1 INTRODUCTION

Contents

```
% make sure the workspace is clear
if ~exist('____code____','var') ; clear ; end
% reduce needless whitespace
format compact
% reduce irritations (pausing and buffering)
more off
% start a diary (in the actual lecture)
%diary lecture1.txt
```

LESSON SUMMARY

- This lesson explains basic use of Matlab. It explains how to do basic computations, and how to document your code using comments starting with a percent. It also explains how to separate your final results into parts using doubled-percent comment lines.

- Variables are named storage locations. A numeric variable contains a number. An assignment statement puts a value into a variable. Such a statement looks like an equality, but it is not.

- You can prevent a command from printing out the result by appending a semicolon.

- Standard numerical functions like `sin`, `cos`, `log` (for ln), `exp`, `sqrt`, ... are available. However, in Matlab you can also write your own functions, and often need to. Normally a function takes in one or more values ("input arguments"), to produce ("return") one or more results ("output arguments"). For example, `sqrt` takes in a nonnegative number as input argument and returns the square root of that number as its output argument.

- Matlab stores numbers to only about 16 decimal digits accurate, starting from the first nonzero digit. As a result, storing a number in Matlab introduces a relative error of up to about 10^{-16}. To get the absolute error, multiply the relative error by the magnitude of the number. Besides inaccuracy, there are also limits on the magnitude of numbers.

- The order in which computations are done is given by "precedence". Matlab does exponentiation first, followed by multiplication and division in any order, followed by addition and subtraction in any order. Use parentheses to ensure that Matlab does things in the order you want, and for more readable code.

- In Matlab, you cannot just use single numbers, but also lists of numbers called "arrays". To create an array, you can put the numbers between square brackets. To create a long list of equally spaced numbers easily, use `START:STEP:END` notation. If you apply various basic operations on entire arrays, you typically need to prefix the operator $*$, /, or ^ by a point (.$*$, ./, or .^).

Key areas of the online book

NOTE: *PA=Participation Activity* and *CA=Challenge Activity*

WARNING: *When you complete any PA or CA in the online book, you should see a colored banner. If you do not see one for a CA, be sure to press "refresh" on your browser (Firefox?). Or you will not get credit.*

Before the first lecture, in the online book do:

- 1.3 Matlab and the interpreter: all.
- 2.1 Variables and assignments: all.
- 2.2 Identifiers: all.
- 2.5 Numeric expressions: skip the stuff beyond CA 2.5.2.
- 3.1 Scripts: all except PA 3.1.4.

Before the second lecture, in the online book do:

- 2.4 Mathematical constants: all.
- 3.2 Comments and clear code: all except the ugly CA.
- 3.4 Internal mathematical functions: skip the CAs.
- 4.1 Introduction to arrays: skip the CA.

BASIC CONCEPTS

Basic computations

The basic numerical operators are +, -, $*$, /, and ^, for plus, minus, times, divide, and power respectively.

```
2+3
```

```
ans =   5
```

```
2-3
```

```
ans = -1
```

```
2*3
```

```
ans =   6
```

```
2/3
```

```
ans =   0.66667
```

```
2^3
```

```
ans =   8
```

```
1.5/.5
```

```
ans =   3
```

Comments

NOTE: *In the lecture, this section will be covered while doing the example homework.*

Comments are required in the homeworks to make things simpler on the grader. They should also be used in your own saved code for the benefit of whoever uses it, including you, later.

1. Lines in your script that start with a percent are ignored by Matlab. These "comment" lines do not really "do" anything. But they do allow you to explain your code to whoever reads it.

2. In addition, a line starting with a *double* percent followed by a title start new section in the resulting pdf file. For example, a "%% Comments" line was used in my script to start this section.

3. If there is *only* the double percent, without a title, then no new section is started. But in the pdf file, your code is executed at that point and the results shown. Use this to force showing results at suitable locations.

4. Single-percent comment lines that *immediately* follow a double-percent line, (with no blank lines or code in between), allow you to use "mark-up". Mark-up allows you to insert such things in the resulting pdf as *italics* (enclose the italic text between underscores), **bold text** (between stars), `typewriter font` (between bars), and math like 2^3 (between dollar signs). You can also insert other files, using "include tags" (see the homework solution files for examples). And you can enumerate items, like the ones in this section, using #, or itemize them using $*$. Note: math looks great in Octave but pretty horrible in Matlab.

```
% let's try a simple division!
1.5/.5
% and another
2.5/.5
```

```
ans =  3
ans =  5
```

(A double percent was used here to force the preceding output)

```
% and a third
3.5/.5
```

```
ans =  7
```

Exponential notation

```
% exponential notation for Planck's constant
1.0546e-34
```

```
ans =     1.0546e-34
```

```
% or not
0.00000000000000000000000000000000010546
```

```
ans =     1.0546e-34
```

```
% the Rydberg constant contains Planck's constant cubed
ans^3
```

```
ans =    1.1729e-102
```

Basic functions

```
% getting the square root of a number
sqrt(9)
```

```
ans =  3
```

```
% matlab (and all science) uses radians by default
sin(30)
```

```
ans = -0.98803
```

```
% this is one way to get the sine of 30 degrees
sin(30/180*pi)
```

```
ans =  0.50000
```

```
% this is a simpler way
sind(30)
```

```
ans =  0.50000
```

Variables

Variables are named storage locations. To put a number `VALUE` into a storage location named `VARIABLENAME`, use

```
VARIABLENAME = VALUE
```

On that "assignment statement", Matlab will see what number `VALUE` is, and then put that number into the storage location called `VARIABLENAME`. If no such storage location exists as yet, it is created.

```
% there is no variable named x yet (no response)
who x
```

Check also the workspace window for `x`.

The statement (or command) `"x=3"` is an example assignment statement. It tells Matlab to create a variable named `x`, if it does not yet exist, (like now), and then put the value 3 in that storage location:

```
% create x and store 3 in it
x=3
```

```
x =  3
```

Now we have a variable named `x`. Check the workspace window or use the `who` command:

```
% look for x
who x
```

```
Variables in the current scope:
x
```

Use `whos` to show more info on x.

```
% get more info on x
whos x
```

```
Variables in the current scope:
   Attr Name        Size     Bytes  Class
   ==== ====        ====     =====  =====
        x           1x1          8  double
Total is 1 element using 8 bytes
```

(You might wonder where the name `double` comes from. Well, long before Matlab was created, normal numbers on a computer had 7 significant digits. Numbers with 16 significant digits were called "double precision". Computer scientists were not yet aware that 2 times 7 is 14, not 16. Matlab choose numbers with 16 significant digits to be the normal ones. But they kept the name "double" to confuse you. It means a normal Matlab floating point number.)

Computing with variables

```
% print out the current value of x
x
```

```
x =  3
```

```
% we can print out twice the value of x
2*x
```

```
ans =   6
```

```
% we can print out the value of x plus 7
x+7
```

```
ans =   10
```

```
% x is still the same
x
```

```
x =   3
```

Please note that the notation `VARIABLENAME=VALUE` is confusing.

REMEMBER: *The command* `VARIABLENAME = VALUE` *is NOT an equality. It is an assignment statement. Think of the equals sign as really being a left pointing arrow, like in* `VARIABLENAME` $\Leftarrow$ `VALUE`. `VALUE` *is pushed into* `VARIABLENAME`.

(I seem to recall that one of my first, very primitive, personal computers actually had a left arrow on the keyboard for assignment statements. And a now largely forgotten programming language, "Algol", used := as an approximation of a left arrow.)

(To instead tell Matlab that `VARIABLENAME` above is *the same* as `VALUE`, or to test for that, use `VARIABLENAME==VALUE`. Note the doubled equal signs.)

As an example of an assignment statement, consider:

```
% add 7 to the current value of x
x=x+7
```

```
x =   10
```

If you think of the above command as an equality, it is complete nonsense: x cannot be equal to x+7! But if you think of it as an assignment statement, it is perfectly OK:

In the above command, `x` was initially 3. Then *first* the *right*-hand side $x + 7$ was evaluated. Then the result, 10, was put in the storage location named `x`. The old value 3 that was in `x` is now *lost*.

```
% we can double x
x=x+x
```

```
x =   20
```

Just for fun, try using the Up-Arrow key a few times now!

Suppressing printed results

Normally, a Matlab command prints out the result of that command. You can suppress that by appending a semicolon to the command.

```
% no final semicolon so the result is printed
1.5/0.5
```

```
ans =   3
```

```
% with a final semicolon nothing is printed
2.5/0.5;
```

```
% the same for variables or other expressions
x;
```

```
% The same for an assignment statement; without semicolon
x=3
```

```
x =   3
```

```
% with semicolon
x=4;
```

```
% even if not printed, x has still been set to 4:
x
```

```
x =   4
```

CREATING YOUR OWN FUNCTIONS

Often you need to create your own functions in Matlab. This section explains how to do that.

Example function: sqr

Matlab provides a function `sqrt(x)` that produces ("returns" or "outputs") the square root of any `x`. But suppose you would like a function `sqr(x)` that returns the square instead of square root of `x`.

The *general* way to do it is to write a function file. For our `sqr` function, the function file *must* be called `sqr.m`. A *minimal* example of the contents of that file is shown below. In the code, `x` is used as symbol for the "input argument", and `x2` for the "output argument", the square of `x`:

Contents of `sqr.m`:

```
function x2 = sqr(x)

x2 = x*x;

end
```

Let's try out our new function!

```
% try finding the square of 2
sqr(2)
```

```
ans =   4
```

```
% try the square of 3 too
sqr(3)
```

```
ans =   9
```

```
% or the square of a variable
x=4
sqr(x)
```

```
x =   4
ans =   16
```

```
% you can use it in assignment statements
x=5
y=sqr(x)
```

```
x =   5
y =   25
```

An improved function: Square

(Note: depending on the time left, cover doing the homeworks first.)

There are several improvements you should always make when creating your own functions:

- The online book recommends that the names of functions that you write *start with a capital*. This avoids potential confusion with a Matlab function of the same name. What if there already *was* a function `sqr` in Matlab? Calling you function `Square` avoids that danger; all normal Matlab functions are lowercase.

- You must use *well-chosen variable names*. For example, the output argument name `x2` can be improved. People might think that `x2` means `x` times 2, rather than `x` to the power 2. Or they might think that it simply means a second `x` value. If you call the output argument `xSqr` instead of `x2`, you avoid such potential confusion.

- You must properly comment your function. For one, the Matlab command `help Square` *must* show what your function does, and how to use it.

Below is an example of how a well-written function `Square` would look:

Contents of `Square.m`:

```
function xSqr = Square(x)

%
% Function that returns the square of its input argument.
%
%                      xSqr = Square(x)
%
% Input:
%    x: must be a real number or variable
%
% Output:
%   xSqr: square of x
%

% set xSqr equal to the square of x
xSqr=x*x;

end
```

```
% test the function
Square(3)
```

```
ans =  9
```

```
% test the "help Square" command
help Square
```

```
'Square' is a function from the file /home/dommelen/
   store/courses/tll/matlab/lessons/Square.m

 Function that returns the square of its input argument.

                    xSqr = Square(x)

 Input:
    x: must be a real number or variable

 Output:
   xSqr: square of x
```

HOW TO DO HOMEWORKS

First consider the METLab Matlab web page. To find it:

- Open https://www.eng.famu.fsu.edu/%7Edommelen/
- Click on `Courses`, `METLab Matlab`.
- Bookmark this page! You can find transcripts of the eight lessons here. You can also find descriptions of the exercises for these lessons here. And you can find example old exams here. Finally there is a semester specific folder where you can find the homeworks.

In principle, you can find the homeworks to do in the semester-specific folder, and the description of the assigned lesson exercises, and templates for them in the top Matlab folder. However, the recommended procedure is to use Secure Shell Client. This will download the templates automatically in a folder METLAB inside MATLAB, put your name and due date in them so that you do not forget or mess it up, and get whatever additional info might be useful for the homework. And you will have to learn how to use Secure Shell Client anyway, for the exams. The procedure is:

- Open Secure Shell Client from "All Programs" in the Windows Start Menu. Click on "Quick Connect", enter `wolf` for the host and your *college of engineering* (not university) username. Hit Enter, click "Yes" or "OK", and enter your *college of engineering* password. Then in the created terminal enter the command
  ```
  ~dommelen/metlab/gethw0
  ```

to get homework 0. (Or `~dommelen/metlab/gethw1` for homework 1, etcetera; later homeworks go similar to the example homework 0. To always get the latest available one, leave out the number.). If all is well, use an `exit` command to close the window and then terminate Secure Shell Client.

- Enter Matlab and if you are not yet in the METLAB folder, double-click it. The homework 0 assignment should now be in file `hw0_LOCAL.pdf` in your workspace. A homework will normally involve *(a)* online book sections, and *(b)* lesson exercises. The descriptions of the lesson exercises can be found in file `lesson_exercises_LOCAL.pdf`.

- In Matlab, now double-click `l0_Test_x1.m` so that Matlab opens it in its editor. Put your solution to the exercise at the *end* of the file. Add comments as appropriate as described earlier.

- Ready for testing? Select "Save" from the file menu, or more quickly use `Ctrl+s` from the keyboard. Then inside the command window, type
  ```
  l0_Test_x1
  ```
  (no `.m`) to run the script and check that all is well. (The Matlab editor has a "Run" button to simplify this process.) Check for problems. Fix them and try again.

- When all is well, in the command window type
  ```
  publish l0_Test_x1.m pdf
  ```
  (with `.m`). This should create a pdf file in the `html` subfolder called `l0_Test_x1.pdf`. Check out this file, fix any problems.

- *Warning:* Immediately exit the pdf reader after viewing `l0_Test_x1.pdf`. Otherwise Matlab will refuse subsequent `publish` commands!

- Warning: If you use the "Publish" button, make sure you are publishing to `pdf`. The default is `html`. Html is not acceptable.

- The same for the remaining exercises. For homeworks beyond homework 0, print out the created exercise pdf files, staple them together in proper order and hand that in at the *start* of the class at which it is due. The TAs will not be able to pick up homeworks handed in after class starts.

- *Warning:* If the TAs note that they spend noticeably more time on grading a certain student than they are comfortable with, the first time they will offer a warning to that student with suggestions for improvement. The second time, the messy answers will no longer be graded and a zero assigned.

Note: The exam questions will be based on (be loose variations of) the lesson exercises. But you will also be expected to have picked up the basic Matlab skills in the covered sections of the online book too.

MORE ON BASIC CONCEPTS

Here are some additional points about simple computations.

Bad numbers

```
% Inf(inity)
1/0
```

```
warning: division by zero
ans = Inf
```

```
% N(ot)aN(umber)
0/0
```

```
warning: division by zero
ans = NaN
```

Overflow and underflow

Numbers that are too big to store are set to infinity. This is called "overflow". In Matlab it happens for numbers bigger in magnitude than roughly 10^{308}. (See `realmax` for the precise value.)

```
% 1/hbar to the power 10 is trouble
(1/1.0546e-34)^10
```

```
ans = Inf
```

Numbers that are too small to store are set to zero. This is called "underflow". In Matlab it happens for numbers smaller in magnitude than roughly 10^{-308}. (See `realmin` for the precise value.) It may be just as bad as overflow.

```
% underflow loses all quantum mechanics here
1.0546e-34^10
```

```
ans = 0
```

Some examples of inaccuracy

Matlab does *not* store infinitely many digits of an arbitrary floating point number. In normal use it stores only about 16 "significant digits". In other words, only about 16 digits are stored correctly, *counting from the first nonzero one.*

In still other words, the way Matlab stores floating point numbers produces a "relative error" of about 10^{-16}.

The below is an expanded discussion of the examples shown in the lecture.

```
% is 4/3 really 4/3???
4/3
% tell Matlab to show all reasonable digits
format long
% try again
4/3
```

```
ans =  1.3333
ans =  1.33333333333333
```

Note that only 15 digits are shown. Is the rest also equal to 3??? No! The 16th digit might still be, but the rest will not.

```
% the next should produce 1/3
(4/3)-1
% so the next should produce zero, but it does not
ans*3-1
```

```
ans =  0.333333333333333
ans =   -2.22044604925031e-16
```

The result is not zero due to the fact that 4/3 had about a 10^{-16} error. The operations we did left *only* the error. While the final `ans` is clearly very small, it is *not* zero. Whether this error is acceptable depends on your application. Note that the *relative* error in `ans` (compared to zero) is infinite.

In general, the *relative* error in a number due merely to storing it is about 10^{-16} in Matlab. To find the actual, or "absolute" error, multiply the relative error by the magnitude of the number.

So bigger numbers have about the same 10^{-16} relative error as smaller numbers, but a bigger absolute error.

Watch what happens if we make the numbers in the above computation about 1,000 times as big:

```
% the next should produce 1000/3
(4000/3)-1000
```

```
ans =  333.333333333333
```

We still have about 16 correct digits, *counting from the first nonzero one.* So the relative error is still about 10^{-16}. But the absolute error is now about 1,000 times as large since the number is 1,000 times as large!

```
% the next should produce zero, but it does not
ans*3-1000
```

```
ans =    -2.27373675443232e-13
```

Note that this produced about 10^{-13} rather than about 10^{-16}; about 1000 times the earlier error.

```
% show just a few digits again
format short
format compact % for at least Octave
```

Note that the relative error is not exactly constant, it can vary a bit. To get the maximum relative error due to storing a number in Matlab, use the bare Matlab `eps` function:

```
% the maximum relative error for storing normal numbers
eps
```

```
ans =     2.2204e-16
```

(Actually, the above assumes that the final digits are simply ignored. If the number is properly rounded before storing, the maximum relative error will be twice as small.)

To get the maximum *absolute* error due to storing a number `NUMBER` in Matlab, (or twice that if rounded), use `eps(NUMBER)`:

```
% the maximum absolute error in a number stored as 1
eps(1)
% the corresponding relative error is the same
eps(1)/1
```

```
ans =     2.2204e-16
ans =     2.2204e-16
```

```
% the maximum absolute error in a number stored as 1000
eps(1000)
% the corresponding relative error
eps(1000)/1000
```

```
ans =     1.1369e-13
ans =     1.1369e-16
```

```
% the maximum absolute error in a number stored as 1024
eps(1024)
% the corresponding relative error
eps(1024)/1024
```

```
ans =    2.2737e-13
ans =    2.2204e-16
```

```
% the maximum absolute error in a number stored as 1023
eps(1023)
% the corresponding relative error
eps(1023)/1023
```

```
ans =    1.1369e-13
ans =    1.1113e-16
```

Since Matlab does not store numbers by their normal ("decimal") digits, even a simple decimal number like $11/10 = 1.1$ still has a 10^{-16} relative error after Matlab has stored it.

```
% the next should produce 1/10
1.1-1
% so the next should produce zero, but it does not
ans*10-1
```

```
ans =  0.10000
ans =    8.8818e-16
```

However, integers can be stored exactly. (At least they can if not extremely big, see `flintmax`.) And fractions like 0.5, 0.25, 0.75, 0.125, ... can be stored exactly too.

```
% the next should produce 1/8, and does
(9/8)-1
% so the next should produce zero, and does
ans*8-1
```

```
ans =  0.12500
ans = 0
```

Note that while integers can be stored exactly, that does not mean that if a stored number is an integer, it is exact. It may simply be a non-integer that happened to round to an integer.

Another inaccuracy to watch for: very big values for the argument of trig functions

```
% should be zero:
sin1=sin(10*pi)
% this too (16 zeros):
sin2=sin(10000000000000000*pi)
```

```
sin1 =    -1.2246e-15
sin2 = -0.37521
```

```
% should be 1:
cos1=cos(10*pi)
% so should be zero:
error1=cos1-1
% should be 1 (17 zeros):
cos2=cos(100000000000000000*pi)
% so should be zero:
error2=cos2-1
```

```
cos1 =  1
error1 = 0
cos2 = -0.53004
error2 = -1.5300
```

Precedence

If no parentheses are used, the following order of precedence applies to basic computations:

```
PRECEDENCE:
                    highest: ^
                    lower:    *  /
                    lowest:  +  -
```

```
% without parentheses
2+3*4
% since * takes precedence over +, this is the same as
val1=2+(3*4)
% and not the same as
val2=(2+3)*4
```

```
ans =  14
val1 =  14
val2 =  20
```

```
% without parentheses
12/2*3
% since / and * have equal precedence, this is
val1=(12/2)*3
% and not
val2=12/(2*3)
```

```
ans =  18
val1 =  18
val2 =  2
```

Manipulating variables

Always keep track of *what* is stored in a variable. Consider the following simple example of what can go wrong.

```
% store 1 in x and 2 in y
x=1
y=2
```

```
x =  1
y =  2
```

Let's try to swap these two values naively.

```
y=x;
x=y;
```

Note in the above that the trailing semi-colons prevent the new values of x and y to be printed. We were keeping them secret. But now look at the results:

```
x
y
```

```
x =  1
y =  1
```

We did not correctly swap the values; the 2 got lost.

Let's try again, this time without semicolons

```
% store 1 in x and 2 in y
x=1
y=2
% swap naively
```

```
y=x
x=y
```

```
x =  1
y =  2
y =  1
x =  1
```

When x was set to y in the final command, the original value of y had already been lost. To fix this, we must prevent the original value of y from becoming lost by storing it in some temporary variable:

```
% store 1 in x and 2 in y
x=1
y=2
% save the original value of y
ySaved=y
% now give y the value of x
y=x
% and give x the *saved* value of y
x=ySaved
```

```
x =  1
y =  2
ySaved =  2
y =  1
x =  2
```

Should we simplify pi?

Note that there is no variable `pi` in the workspace.

```
% show pi
pi
% show more digits
format long
pi
% reset the number of printed digits
format short
```

```
ans =  3.1416
ans =  3.14159265358979
```

The decimals of pi are much too complicated! So the Indiana "pi bill" was introduced to redefine pi as 3.2.

```
% simplify pi
pi=3.2
```

```
pi =   3.2000
```

Note that there is now a `pi` in the workspace.

Unfortunately, the legislature changed it mind after loud protests from mathematicians. So we should undo this.

```
% get rid of variable pi
clear pi

% we have the old value back
pi
```

```
ans =   3.1416
```

MORE ON FUNCTIONS

Many students are confused by functions. Let's see whether we can figure out exactly what Matlab does when a simple function like `sqr` is used.

Contents of `sqr.m`:

```
function x2 = sqr(x)

x2 = x*x;

end
```

Contents of `trysqr.m`:

```
disp('Start of trysqr.m')

sqr(3)

y=4

sqr(y)
```

To run:

- Observe the workspace with the *saved* `trysqr.m` open in the editor.
- Using the "Breakpoint" edit toolbar button, set a break point in `trysqr.m` just before the first use of `sqr`.

- Press the "Run" button.
- Observe the workspace.
- Use "Step-into".
- Observe the workspace. (Matlab uses "Pass-by-Value")
- Use "Step"
- Observe the workspace.
- Etcetera.

ARRAYS

Arrays are tables of numbers. They are usually created using square brackets.

Some examples

```
% create a row of numbers
list=[1 2 4 9 16]
```

```
list =
     1     2     4     9    16
```

Matlab functions can handle entire lists!

```
% take the square root of each element in the list
sqrt(list)
```

```
ans =
    1.0000    1.4142    2.0000    3.0000    4.0000
```

Another example:

```
% a list of special angles in degrees
list=[0 30 45 60 90]
% take the sine, cosine, and tangent
sinlist=sind(list)
coslist=cosd(list)
tanlist=tand(list)
```

```
list =
     0    30    45    60    90
sinlist =
   0.00000   0.50000   0.70711   0.86603   1.00000
coslist =
   1.00000   0.86603   0.70711   0.50000   0.00000
tanlist =
   0.00000   0.57735   1.00000   1.73205       Inf
```

A trick: using the colon operator

You can create some simple types of arrays more easily using START:END notation.

```
% the straightforward way
list=[1 2 3 4 5 6 7 8 9 10]

% the quickest way to do this: START:END
list=1:10
```

```
list =
     1     2     3     4     5     6     7     8     9    10
list =
     1     2     3     4     5     6     7     8     9    10
```

More generally, you can use START:STEP:END notation.

```
% the straightforward way
list=[1 3 5 7 9]

% create a list with a step (increment) of 2
list=1:2:9
list=1:2:10
```

```
list =
   1   3   5   7   9
list =
   1   3   5   7   9
list =
   1   3   5   7   9
```

Negative values of STEP create decreasing lists

```
% a decreasing list
list=10:-1:1
```

```
list =
   10     9     8     7     6     5     4     3     2     1
```

Let's have an example for a different range of numbers.

```
% even numbers from -4 to 12
list=-4:2:12
```

```
list =
   -4    -2     0     2     4     6     8    10    12
```

Elementwise basic operators

When operating on entire lists with ^, ∗, or /, you may need to precede these operators with a point. That tells Matlab to do the operation on each list element (for .^), or pair of list elements (for .∗ and ./), separately.

```
% a simple list
list1=[1 2 3]
% square the list
list1Sqr=list1.^2
% without the point you get an error
%list1^2 % don't, produces an error
```

```
list1 =
   1   2   3
list1Sqr =
   1   4   9
```

Let's also try multiplying and dividing lists

```
% the first list
list1=[1 2 3]
% the second list *of the same size* (3 elements)
list2=[2 4 6]
% multiply the lists
product=list1.*list2
% divide the lists
ratio=list1./list2
```

```
list1 =
   1   2   3
list2 =
   2   4   6
product =
     2     8    18
```

```
ratio =
   0.50000    0.50000    0.50000
```

Fixing our sqr function

Our earlier function `sqr` does not work on lists since the . is missing before the ∗:

Contents of `sqr.m`:

```
function x2 = sqr(x)

x2 = x*x;

end
```

This is fixed in function `sqrFixed`:

Contents of `sqrFixed.m`:

```
function x2 = sqrFixed(x)

x2 = x.*x;

end
```

```
% try a simple list
list=[1 2 4 9 16]

% the Matlab sqrt function works fine
goodSqrt=sqrt(list)

% our function sqr creates an error
%badSqr=sqr(list)  % don't, produces an error

% our fixed function sqr works OK
goodSqr=sqrFixed(list)
```

```
list =
    1     2     4     9    16
goodSqrt =
   1.0000    1.4142    2.0000    3.0000    4.0000
goodSqr =
     1      4     16     81    256
```