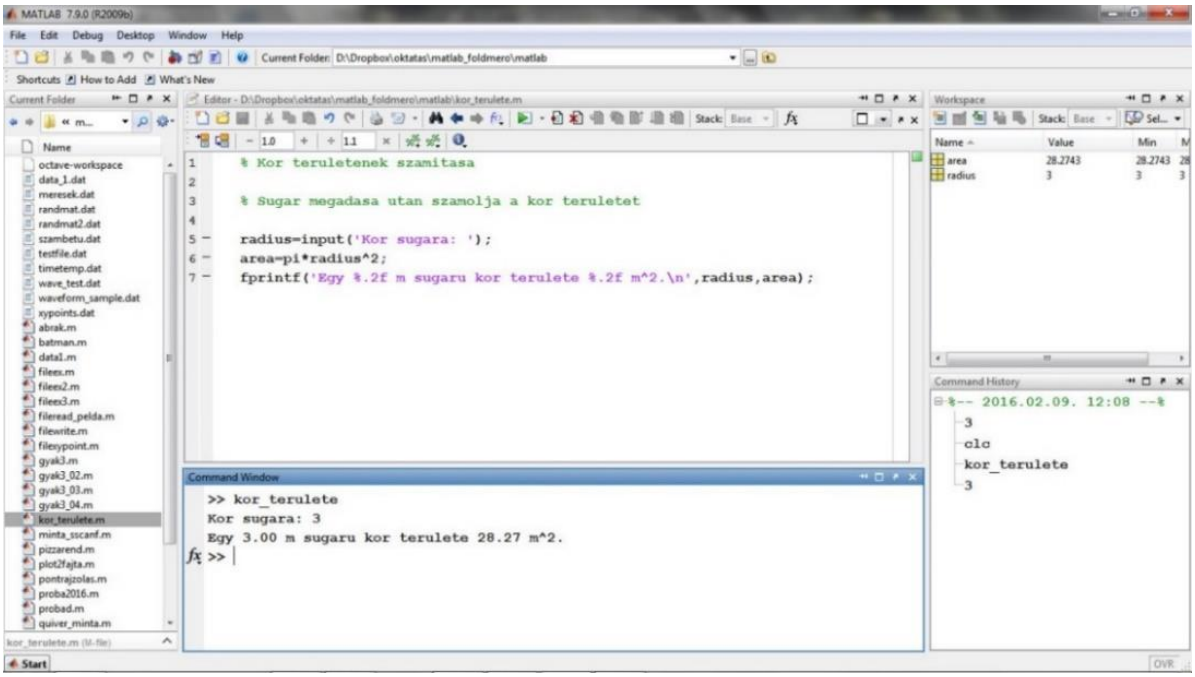
1. MATLAB/OCTAVE BASICS [1]

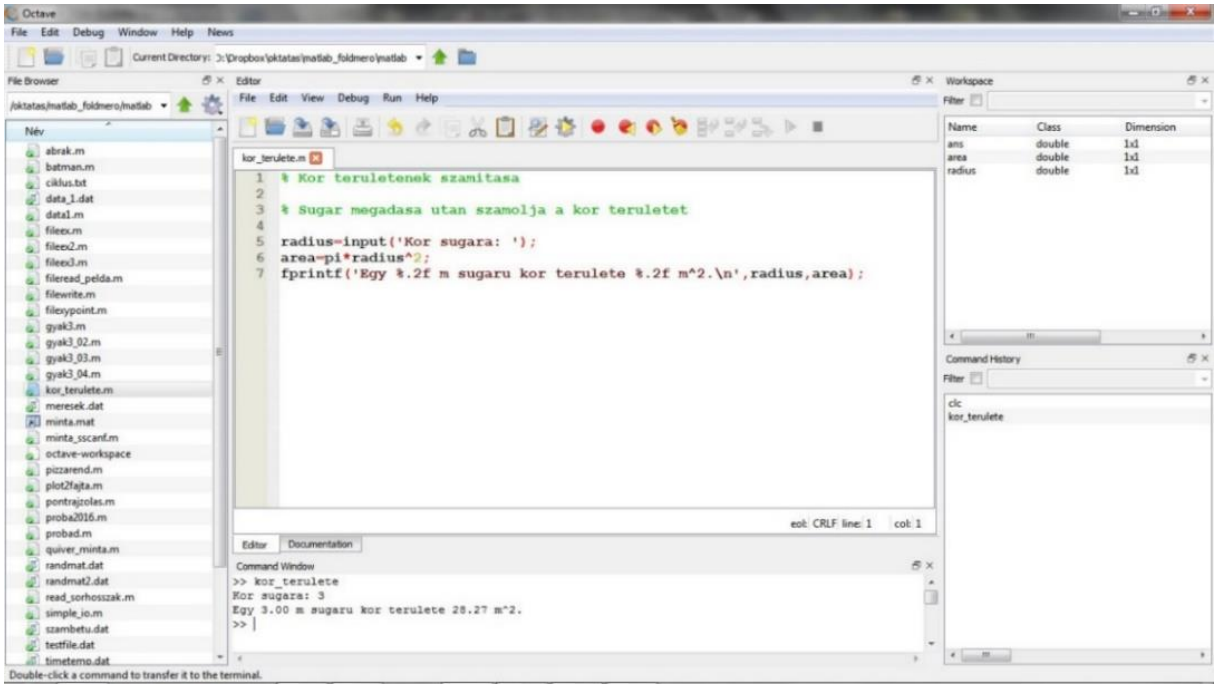
In the following exercises, various engineering problems are solved with numerical methods algorithms using the Matlab mathematical environment. The Matlab software is available for the students of Budapest University of Technology and Economics (BUTE) for educational purposes from March 2017. A good alternative might be the Octave mathematical environment, which is a free, open-source program where you can use essentially the same commands as in Matlab. Even the graphical interface is very similar in the two programs, with optional rearrangeable windows (see Figures 1, 2). **Octave** can be downloaded from https://www.gnu.org/software/octave/, the current version is 7.1.0, which came out on Apr 6, 2022.

Students can create a **MathWorks account** on the Matlab site with their BME email address, with this account they can also access Online Matlab. With the MathWorks account, students can practice the basics of Matlab by solving the tasks of the **Matlab Onramp** online study site, which is recommended for everyone (it takes about 2-3 hours). There are other good online practice options at **Matlab Cody** website.



1MATLAB GRAPHICAL ENVIRONMENT

[1] Reference for Matlab/Octave Basics: Todd Young and Martin J. Mohlenkamp: Introduction to Numerical Methods and Matlab Programming for Engineers, Department of Mathematics, Ohio University, July 24, 2018, http://www.ohiouniversityfaculty.com/youngt/IntNumMeth/book.pdf



1 OCTAVE GRAPHICAL ENVIRONMENT

STARTING MATLAB/OCTAVE

The most important parts of the graphical interface are the **current folder** where the current directory is located, the **command window**, the **workspace** with the used variables, the **command history** with all the used commands and the **editor**. Left click and hold the panel names and drag them to the blue area. You may want to dock the editor by clicking the arrow in the upper right corner of the Editor window.

Remember that file names used in Matlab cannot start with a number or contain special characters or spaces! Only those functions and files that are in the current directory can be used or run in the program.

HELP, DOCUMENTATION

We can learn a lot from the proper use of documentation. In older Matlab versions (prior to 2019), typing **help** to the Command Window listed the categories of various built-in functions. Here you could either double-click the category name or type the category name after the help command (in this book, the > symbol indicates Matlab commands). From Matlab R2019a, typing the '**help**' command will display the help for the last command used. Try the following:

```
> help elfun
```

This will list the 'Elementary math functions', e.g. trigonometric, exponential, complex and rounding functions. If you know the name of a function, you can type it after the help:

```
> help 'function name'
```

This describes the command and how to use it. For example try it:

```
> help rand
```

This command describes how the **rand** function generates a uniformly distributed random number between 0 and 1, how to use it to call with one or more inputs, and lists some related commands (e.g. **randn**, which generates standard normal random numbers with 0 expected values and 1 standard deviation).

The **doc** command works similarly to help, but is a much more detailed description of the command, with many examples

```
> doc randn
```

Let's try the **pwd** function! Use the **help** command to figure out what the command does!

Another useful function is the **lookfor** command. Use this to search for a word in the command name or description. Let's try the following:

```
> lookfor rand
```

This lists all the commands that have the word **rand** in their name or short description. If the search takes too long, it can be aborted by pressing CTRL + C.

ONLINE DOCUMENTATION

Help for Octave: https://www.gnu.org/software/octave/doc/interpreter/

Help for Matlab: http://www.mathworks.com/help/matlab/

Useful tips, functions in Matlabcentral: http://www.mathworks.com/matlabcentral/

Practice material: https://matlabacademy.mathworks.com/

Other practicing examples to solve: https://www.mathworks.com/matlabcentral/cody/

SOME USEFUL COMMANDS

`clc` – clears the content of the command window

`clear` – deletes variables (see workspace)

`close` – closes current figure or all (close all)

`CTRL+C` – interrupts the given command (e.g. exit an infinite loop)

% – comment (the program ignores everything in the line after this sign)

%% – you can open a new section in your script

; – using this sign at the end of the command the result will not be displayed in the Command window (cancels echo)

USING THE 'TAB' AND ARROW BUTTONS IN THE COMMAND WINDOW

The '**Tab**' button is very useful. If we do not know exactly the name of a particular command, just the beginning, let's start typing the beginning into the command line e.g. **pref**, then press '**Tab**' key. If there is only one command starting with **pref**, the '**Tab**' button will complete the command, if there are more, the '**Tab**' will list them. In this case, there are several commands with the beginning of **pref**, e.g. **prefdir** (the

name of the directory where all settings, history, etc. are located) or **preferences**, which opens the settings window.

The **up and down arrow keys** are also very useful in the command line. With them, the previous commands can be retrieved, executed and modified. Previous commands can also be executed using the **command history** by double-clicking on the command name.

TAB	- use it to finish the started function/variable etc. name
ARROW KEYS	- you can scroll around the previously processed commands (you can access those from the command history too)
CTRL+ENTER	- run the whole section
F9	- run the selected part
F5	- run the whole script

ENTERING NUMBERS/VECTORS, VARIABLE TYPES, FUNCTIONS

```
> %% Entering numbers
> a = 0.01
> b = 1e-2
> c = 1d-2;
> a+c
> clear a % deletes the variable 'a'
> clear % (or 'clear all') clears all variables
> pi % built-in value (3.14)
> e = exp(1) % e^1 = e = 2.71
> b = e^-10 % exponentiation
```

The basic variable type in Matlab is the matrix. Vectors are special matrices, 1xn row vectors or mx1 column vectors. In Matlab, square brackets are used to define a matrix or vector. When working with matrices/vectors, be careful with the different brackets, because each type performs a different operation: ( ), [ ], { }! Elements within a row are separated by commas or spaces, and rows are separated by semicolons

```
> z = [1 3 45 33 78] % rowvector
> z = [1,3,45,33,78] % rowvector
> t = [2; 4; 22; 66; 21] % column vectors
> M = [1,2,3; 4,5,6] % matix, with 2x3 size
```

Basic formatting:

```
> format long % 14 decimal digits
> e, b
> format short % 4 decimal digits
> e, b
```

When displaying a vector, it can be a problem to display very small and very large numbers at the same time. Let's look at the following vector for example

```
> x = [25 56.31156 255.52675 9876899999];
> format short
> x
```

The result is difficult to read and not very informative:

```
x = 1.0e+09 *
```

```
    0.0000    0.0000    0.0000    9.8769
```

In this case, it is better to use a different display format, such as **shortG** or **longG**, which displays numbers of each different magnitude in their compact format (**shortG** with 5 significant digits, **longG** with 15 significant digits).

```
> format shortG
> x
> format longG
> x
```

The results are:

```
x = 25       56.312       255.53          9.8769e+09 % format shortG
x = 25       56.31156     255.52675       9876899999 % format longG
```

You can query any element of a vector by enclosing the element's index number in round brackets.

```
> t(2) % result: 4
> M(2,3) % result: 6
> z(end) % last element of z: 78
```

Or you can override the value of any element in the same way:

```
> t(2)=47
> p = [] % empty vector
> z(3)=[]; % deletes the 3rd element, after: z = 1 3 33 78
```

You can query a part of the vector or the matrix

```
> t(2:4) % result after the previous override command: 47 22 66
> t(1:29 % error message after mistyping
  t(39
     ?
 Error: Expression or statement is incorrect--possibly unbalanced (, {, or
[.
> t(1:29)
Index exceeds matrix dimensions.

> M(1:2,2:3) % result: [2,3; 5,6]
```

Transposed vector/matrix (switched rows and columns):

```
> tt = t' % t transposed, row vector
> Mt = M' % result: [1,4; 2,5; 3,6]
```

There are some useful commands that you can use to generate vectors:

```
> x1 = 1:10 % line vector: integers from 1 to 10
> x2 = 1:0.3:10 % line vector from 1 to 10, with 0.3 spacing
> x3= (1:0.3:10)' % column vector from 1 to 10, with 0.3 spacing
> x4 = linspace(1,10,4) % 4 numbers between 1 and 10, equaly spaced
```

It is easy to concatenate vectors/matrices horizontally/vertically when they have the same number of rows or columns.

```
> X = rand(2,3) % A matrix of random numbers between [0,1], size: 2x3
> Y = ones(2,4) % matrix of ones, size: 2x4
> Z = eye(3) % 3x3 identity (unit) matrix
> W = zeros(2,4) % Matrix of zeros, size: 2x4
> XY = [X, Y] % 2x7 sized matrix, X and Y horizontally concatenated
> XZ = [X; Z] % 5x3 sized matrix, X and Z vertically concatenated
> XY2 = [X; Y] % error message
```

```
Error using vertcat
Dimensions of matrices being concatenated are not consistent.
   > XZ2 = [X,Z] % error message
Error using horzcat
Dimensions of matrices being concatenated are not consistent.
```

Accessing one row/column in a matrix:

```
> XY(1,:) % first row of XY (the : sign means all items in that row)
> XY(end,:) % last row of XY
> XY(:,1) % first column of XY (all items in that column)
> XY(:,end-1) % penultimate column of XY
```

Texts as vectors of characters

```
> s = 'p' % a text/character (string) variable of size 1x1
> u = 'University of Technology' % string type variable, size 1-by-24
> b = 'Budapest '
> bu = [b,u] % you can combine them using square brackets '[]'
> % Budapest University of Technology
> bu(24:29) % The strings could be handled as vectors
> % Techno
```

MOST COMMON VARIABLE TYPES

- Double: represents a double-precision floating-point number, mostly used to represent rational numbers. This is the default data type for numbers.
- Integer: represent an integer number (no fractional part). Be careful when calculating with both integers and doubles.
- Array (vector/matrix): a (multi-dimensional) collection of numbers. Can only contain data of similar type.
- Character array: textual data enclosed in single quotes, e.g. 'vehicle'.
- Cell array: similar to the regular, multi-dimensional array, but the types of the contained data can be different.
- Structure: an array with named fields, can contain varying data types.
- Table: array in a tabular form whose columns can be named and can be of varying data types.

WRITING AND RUNNING SCRIPT FILES

So far, we have been working from the command line, but solving a more complex calculation would be very difficult using only the command window. Therefore it may be better to collect the necessary commands/functions into a script file. The default file type for Matlab is the * .m file, a simple text file that can be edited with any text editor. In addition, in recent versions of Matlab, you can use the LiveScript file type (* .mlx), in which you can mix Matlab instructions and formatted text, images, and see the results immediately. The disadvantage of this format is that this is Matlab's own format, which can only be opened within Matlab and its execution is much slower.

You can run the script files by typing their names or just pressing **F5**, this will save and run the program immediately. Make sure that the filename does not start with a number or contain spaces or special characters! Filenames must start with a letter and contain only the English alphabet, numbers, and underscores (_).

If you don't want to run the whole program, you can write a **return** command somewhere, and after pressing **F5** the program will only run until that point. Alternatively, pressing **F9** will run only the selected part. You can divide the file into sections using double **%** characters (**%%**) followed by a section title. You can execute commands in a particular section by pressing **CTRL+Enter**. Let's start a new script file by clicking on the plus sign (new) in the upper left corner and this will open a blank page in the Editor. Save it to your current directory as practice1.m file!

PLOTTING - THE BASICS

The next table contains the data of a stress-strain diagram (σ-ε) of a steel bar for reinforced concrete:

ε [%]	0	0.2	2	20	25
σ [N/mm^2=Mpa]	0	300	285	450	350

1. TABLE, STRESS-STRAIN DIAGRAM OF A STEEL BAR FOR REINFORCED CONCRETE

Type the next commands to **practice1.m** script file!

```
> %% STRESS-STRAIN DIAGRAM OF STEEL
> x = [0,0.2,2,20,25] % prints its contents to the command window
> y = [0,300,285,450,350]; % does not display its contents in the
  command window
```

Run either the entire file with **F5** or a selected section with **F9** or the actual section with **CTRL + Enter**. You can check the content of any variable by typing its name into the command window or to the script file.

```
> x, y
```

To plot data stored in vectors, use the **plot** command:

```
> plot(x,y)
```

This will connect the points with a line. If you want to mark points with symbols, try the following:

```
> plot(x,y,'x')
> plot(x,y,'o-')
> plot(x,y,'r*-')
```

You can add many arguments to define the specifics of a plot, like the shape and size of the markers, specifics of the lines, etc.

```
> plot(x,y,'--gs','LineWidth',2,'MarkerSize',10,...
>       'MarkerEdgeColor','b','MarkerFaceColor',[0.5,0.5,0.5])
```

Useful specifiers:

Marker	Description
o	Circle
+	Plus sign
*	Asterisk
.	Point
x	Cross
s	Square
d	Diamond
^	Upward-pointing triangle
v	Downward-pointing triangle
>	Right-pointing triangle
<	Left-pointing triangle
p	Pentagram
h	Hexagram

Long Name	Short Name	RGB Triplet
'yellow'	'y'	[1 1 0]
'magenta'	'm'	[1 0 1]
'cyan'	'c'	[0 1 1]
'red'	'r'	[1 0 0]
'green'	'g'	[0 1 0]
'blue'	'b'	[0 0 1]
'white'	'w'	[1 1 1]
'black'	'k'	[0 0 0]

LineWidth	Line width
MarkerEdgeColor	Color of the outline of the symbol
MarkerFaceColor	Color of the symbol fill
MarkerSize	Size of the symbol

You can name the axes, or add a legend or a title:

```
> xlabel('Strain')
> ylabel('Stress')
> title('STRESS-STRAIN DIAGRAM OF STEEL')
```

ADDITIONAL USEFUL TIPS FOR PLOTTING

Each figure and also the plotted elements can be named with a handle (an identifier). You can use this to access any figure/element later if you want to set a property or clear them. If you don't do further settings, by plotting a new element, the previous one will be deleted.

```
> clf         % clear figure
> f1 = figure; % the figure function creates a new figure
> p1 = plot(x,y,'r*');
> hold on  % you can fix the diagram, to add new elements to it without
  cleaning it first
> p2 = plot(x,y);
> p3 = plot(x,y,'bo');
> delete(p1)  % you can delete elements by using its handle
> figure(1)   % by referring to the number of the figure, you can work
  on any previous figure
> close % close figure
```

FUNCTIONS

There are many mathematical and other built-in functions in Matlab, it is useful to browse the documentation to get to know them. Let's look at some of the basic math functions, from 'Elementary math functions' in the documentation, using the '**help elfun**' command! The input variables of the functions are always enclosed in round brackets. The default angular unit for trigonometric functions is radian.

```
> sin(pi) % its value is 0 within the accuracy of the numerical
  representation
> cos(pi) % -1
> tan(pi) % a large number instead of infinite
> log(100) % natural logarithm
> log10(100) % 10 based logarithm
> 3^4 % value: 81
> sqrt(81) % 9
> abs(-6) % 6
> exp(0)
```

exp(0) is e^0, its value is of course 1. The built-in functions work not only on numbers but also on vectors.

```
> x = linspace(0,2*pi,40)
> y = sin(x)
> figure(1); plot(x,y)
```

For more settings see **help plot**!

USER DEFINED ANONYMOUS ('SINGLE-LINE') FUNCTIONS

There are many ways to specify your own functions in Matlab, in simple cases we use the so-called anonymous function, which is not saved as a separate program, only assigned to a variable. Let's define cos(2x) function in this way!

```
> f = @(x) cos(2*x)
```

Here, after the @ symbol, you must specify the independent variables of the function, and after a space you can write the formula. Let's plot this function also next to the sine function you have just drawn. When drawing the sine function, we first calculated some points of the function and plotted them using the **plot** command. We can plot functions without calculating their points beforehand, using symbolic **fplot** or **ezplot** commands. (Note: In Octave and older Matlab versions you can use only the **ezplot** command, in newer Matlab **fplot** is recommended).

```
> hold on
> fplot(f,[0,2*pi]) % or ezplot(f,[0,2*pi])
```

Without the **hold on** command, the previously drawn elements will be deleted from the figure, the **hold off** command restores this default mode. For **fplot**, you can specify an interval where you want to display the function (it is optional, there is a default interval also).

Calculate the squares of the integers between 1-10 using your own function! First, let's define a function to calculate the square of any number!

```
> f1 = @(x) x^2
```

Check for a given value of x:

```
> f1(3) % value: 9
```

And now let's calculate the squares of the integers between 1-10!

```
> x = 1:10;
> y = f1(x)
```

We got an error message:

```
Error using  ^
One argument must be a square matrix and the other must be a scalar. Use
POWER (.^) for elementwise power.
Error in gyak1>@(x)x^2
Error in gyak1 (line 100)
y = f(x)
```

Why? Because our variable x is a line vector, and when it is squared we try to multiply two line vectors, which is mathematically incorrect (mathematically the line vector could be multiplied by a column vector). If we do not want to perform a vector/matrix operation, but element by element operation, we must put a dot before the * operation symbol (elementwise operation).

```
> f1 = @(x) x.^2
> y = f1(x) % 1  4   9  16  25  36  49  64  81  100
```

Since addition/subtraction, division/multiplication with a scalar for vectors/matrices is done element by element, there is no need to put a dot before the operation symbol in these cases, only in case of multiplication, division and power of vectors:. *,. /,. ^.

One of Matlab's strengths is its ability to perform operations with vectors and matrices, so in many cases, we can avoid the use of the much slower loops with vectorized operations.

FUNCTIONS IN SEPARATE FILES

Matlab's default file type is a text file with the *.m extension. There are two main types, script file and function. We have used the former in our work so far, now let's see the differences between functions and script files!

One of the advantages of writing the functions into a separate file compared with the anonymous functions is that it can be invoked from any other script file also. Another advantage is that it can be used to perform more complex calculations, it is easy to parameterize the input and output variables and a description can be added for help.

Let's rewrite the previous square function into a separate function file! Click the plus sign (new) in the upper left corner and a blank page will open in the Editor. Type in the following, and save it as **squarefun.m** to the current directory. Important: The name of the function (written in bold in the following code) must be the same as the file name, otherwise the function cannot be called!

```
> function y = squarefun(x)
> % Calculate the square of a number
>     y = x.^2;
> end
```

Some characteristics of the functions are:

- begins with the word function
- There are at least one output and one input
- The output, function name and input arguments are on the first line and the function name must match the *.m filename
- We need to assign value to the output somewhere inside the function
- Internal variables in a function are local variables, they will not appear in the workspace, and the function will not have access to the variables in the workspace except to the defined input arguments.
- A function cannot be executed, it can only be called from another file or command line! To be called, the function must be in the current directory (or in a directory that is in the path).
- The comments written after the first line of the function are listed when using the help command with this function name.

Let's call the written function on our vector x! To do this, switch back to the **practice1.m** script file!

```
> squarefun(11) % 121
> squarefun(x) % 1  4  9  16  25  36  49  64  81  100
> help squarefun %  Calculate the square of a number
```

A function can have multiple inputs, listed after the name of the function in round brackets. Let's modify our previous function as follows and save it as **powerp.m**!

```
> function y = powerp(x,p)
>     y = x.^p
> end
```

A function can have multiple outputs collected in a vector in square brackets (powers.m):

```
> function [x2 x3 x4] = powers(x)
>     x2 = x.^2;
>     x3 = x.^3;
>     x4 = x.^4;
> end
```

Call the above functions also from our script file!

```
> powerp(x,3) % 1  8  27  64  125  216  343  512  729  1000
> [a b c] = powers(x)
> % a = 1 4  9 16 25 36 49 64 81 100
> % b = 1 8 27 64 125 216 343 512 729 1000
> % c = 1 16 81 256 625 1296 2401 4096 6561 10000
```

Plot the results in a new Figure (number 3) using the figure command. You can list several graphs to plot in the same figure in one **plot** command!

```
> figure(3)
> plot(x,a,x,b,x,c)
```

We can add custom colors and legend. The text of the legend should be in the same order as we plotted the graphs.

```
> plot(x,a,'black',x,b,'blue',x,c,'green')
```

```
> plot(x,a,'k',x,b,'b',x,c,'g')
> legend('square','x^3','x^4','Location','North')
> legend('square','x^3','x^4','Location','Best')
```

COMMENTS AND HELP TO USER DEFINED FUNCTIONS

Comments are an important part of multi-line programs. On the one hand, others also can understand our code, and on the other hand, we will also remember it if we want to use it again or modify later. It is advisable to write comments not only at the beginning of the program but also for each new section. In Matlab, you can write comments after the % sign. For a function, it is useful to specify in the comments what the purpose of the function is, what input and output values are used. In the case of a function, the comments written after the first line will be displayed when calling the help command with this function name.

MATLAB ERROR MESSAGES

When writing a script we encounter many error messages, as we saw earlier. It is important that we can correct our mistakes by interpretation!

Let's look a mistyping error in '**clear all**'!

```
> cler all
Undefined function or variable 'cler'.
Did you mean:
>> clear all
```

Matlab is case sensitive:

```
> X = 3/4; x
Undefined function or variable 'x'.
```

Let's look an example of a syntactically incorrect Matlab statement:

```
> 1 X
1 X
  ↑
Error: Unexpected MATLAB expression.
```

Wrong number of input parameters:

```
> sin(pi,3)
Error using sin
Too many input arguments.
```

The number of rows/columns in the matrices does not match:

```
> M = [1 2;3]
Dimensions of matrices being concatenated are not consistent.
> [3, 4, 5] * [1; 2; 3; 4]
Error using  *
Inner matrix dimensions must agree.
> a = 1:5, b = 1:3
> [a;b]
Error using vertcat
Dimensions of matrices being concatenated are not consistent.
```

A common typo is to write 9 or 0 instead of parentheses:

```
>  sin(pi0
sin(pi0
       ↑
Error: Expression or statement is incorrect--possibly unbalanced (, {, or [.
```

Missing parenthesis:

```
>  abs(sin(rand(2))
abs(sin(rand(2))
                ↑
Error: Expression or statement is incorrect--possibly unbalanced (, {, or [.
```

We want to perform element-by-element operation on a vector, but the point is missing:

```
>  v =1:4;
>  1/v
Error using  /
Matrix dimensions must agree.
```

The worst is when there is no error message but the result is still wrong. Example: Calculate $\frac{1}{2\pi}$ with the following statement. Why is the result bad?

```
>  1/2*pi %  1.5708
```

2. CONTOL FLOW STUCTURES, DATA IMPORT/EXPORT

LOGICAL OPERATIONS

Certain basic knowledge of logical operations (1-true / 0-false) is also very important using Matlab, especially when modifying and querying the elements of vectors/matrices. Many problems can be solved with matrices and boolean variables that would require a loop in other programming languages. Create the practice2.m script file in the current directory!

```
>  clc; clear all; close all;
>  % equal ==, non equal ~=
>  a = 3==4 % false - 0
>  whos a
>  b = 5~=6 % true - 1
>  vs = [1 2 3 4 5 6] % row vector
>  vs(5)>5
>  vs(5)>=5
>  % or: ||, and: &&
>  a || b % true because one of the 2 conditions are true
>  a && b % false because only one condition is true
```

Let's look at an example where a given property of a vector is queried with a logical variable. Imagine a university professor who likes randomly grade the students on an exam. There are 6 students on the exam, their names are a, b, c, d, e, f. Everyone got a mark between 1-5 randomly. The question is how many people failed (a grade of 1) and who exactly in that exam?

```
>  students = ['a';'b';'c';'d';'e';'f']
>  marks = ceil(rand(1,6)*5)
>  failed = marks<2
>  students(failed)
```

The result of the failed vector will be a 6-element vector with 1 in the places where the condition is true, 0 in the other places. If you want to get the names of the failed candidates you just need to call '`students(failed)`' command, which returns only the names of the students where the value of the failed vector was 1. Such a query can be solved in Matlab without a loop using logical variables. Note: we used a rounding command: **ceil**, see for details others in help: round, floor, fix.

LOOPS, BRANCHES

IF -ELSE CONDITIONAL STATEMENTS

The if-else conditional statement is a two-way conditional branch. The structure of the basic 'if statement' is: condition; what to do if the condition is true (can be in one or more lines), end; There can be other branches before the end using elseif (otherwise if ...) or else (otherwise). The structure of an if-else conditional statement in Matlab is:

```
if  (conditional expression)
    (Matlab commands)
elseif (conditional expression)
    (Matlab commands)
else
    (Matlab commands)
end
```

Let's see an example! Plot the following quadratic equations, determine the number of real roots and give them if there is any!

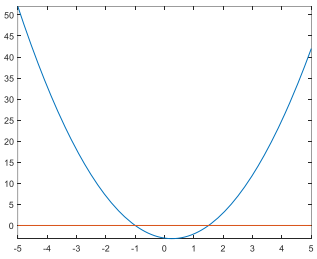
$$2\,x^2 - x - 3 = 0$$

$$x^2 + 2\,x + 3 = 0$$

$$2\,x^2 + 4\,x + 2 = 0$$

The general form of the quadratic equation is: $ax^2 + bx + c = 0$. And the solution form:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$



Plot the first equation as a function using **fplot**!

```
> close all;
> a = 2, b = -1, c = -3
> f = @(x) a*x.^2+b*x+c;
> figure; fplot(f);
> hold on; plot(xlim,[0,0])
```

First we need to check whether there is a real solution or not, if so, is there 1 or 2 solutions? Let's look at the following user-defined function (quadratic.m), which examines the number of solutions of a quadratic equation (x^2+bx+c=0), plots the function and gives the real solutions if there is any! For this, different cases of the discriminant D = b ^ 2-4ac must be considered. Save the file to the current directory.

```
function x = quadratic(a,b,c)
% Solving a*x^2+b*x+c=0 equation, input: a,b,c
    f = @(x) a*x.^2+b*x+c;
    figure; fplot(f);
    D = b^2-4*a*c; % discriminant
    if D>0
        disp('The equation has 2 real solutions')
        x(1) = (-b+sqrt(D))/(2*a);
        x(2) = (-b-sqrt(D))/(2*a);
        hold on; plot(x,[0,0],'k*'); plot(xlim,[0,0],'r');
    elseif D==0
        disp('The equation has 1 real solution')
        x = -b/(2*a);
        hold on; plot(x,0,'k*'); plot(xlim,[0,0],'r');
    else
        disp('The equation has no real solution')
        x = []; hold on; plot(xlim,[0,0],'r');
    end
end
```

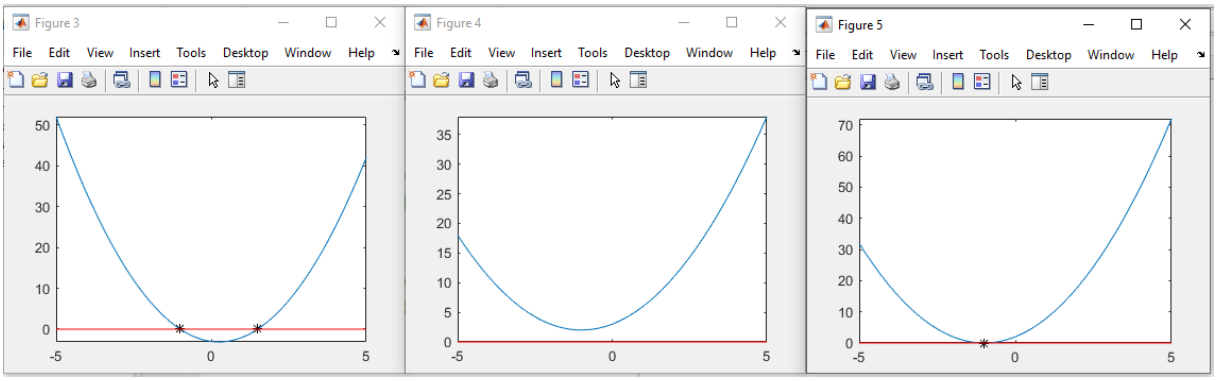
The **disp** command will print a text to the command window.

Unlike scripts, functions cannot be run with F5, they can only be called from the command line or from a script file. Solve the first equation from the command line by calling the quadratic function! (You can do this if the file is in the same directory you are working in.)

```
quadratic(2,-1,-3)
```

However, it is better to work from a script file that can be easily modified later. Type the following to the practice2.m script file!

```
%% Branches - IF
disp('Solving quadratic equations: ax^2+bx+c=0')
a = 2, b = -1, c = -3,
x = quadratic(a,b,c)
a = 1, b = 2, c = 3,
x = quadratic (a,b,c)
a = 2, b = 4, c =2,
x = quadratic (a,b,c)
```



Note: In case of the latest Matlab versions, you can also copy the functions to the end of the script file, it is not necessary to save them to a separate file. In this case, the function can be called when running the whole script (with F5), but when running only a section (with F9) Matlab will not find the function at the end of the script file!

SWITCH STATEMENT

In more complex cases, we can use not only two-way but also multi-way branches. Write a program to help the teacher randomly grade students! The **randi(n)** command can be used to generate random integers between 1 and n. Generate a number between 1-5 and display a message based on the result! Let's put this into a new section using double **%%** characters. We can run a separate section using CTRL + Enter key combination. Try this 3 times. Did you get an excellent (5) mark from 3 runs?

```
> %% Branches - SWITCH
> disp('Mark:')
> mark = randi(5);
> switch mark
>   case 1
>     disp('Fail')
>   case 2
>     disp('Pass')
>   case 3
>     disp('Satisfactory ')
>   case 4
>     disp('Good')
>   case 5
>     disp('Excellent')
> end
```

ITERATIONS - FOR LOOP

In a loop, the execution of a group of commands is repeated several times consecutively. The number of repetitions in the for loop is predetermined. The structure of a for-end loop is as follows:

for *i* = *f:s:t*

(Matlab commands)

end

Where *i* is the loop index variable, *f* is the value of *i* in the first step, *s* is the increment of *i* after each step and *t* is the value of *i* in the last step.

Let's see how we can solve the equations described in the first example using a cycle, if the coefficients are stored in a matrix!

```
> %% Loops - FOR
> close all; clc; clear all;
> disp('Solve the next equations: 2x^2-x-3=0, x^2+2x+3=0, 2x^2+4x+2=0')
> M = [2,-1,-3;
>      1,2,3;
>      2,4,2]
> n = size(M,1) % number of rows
>
> for i = 1:n
>     a = M(i,1), b = M(i,2), c = M(i,3),
>     quadratic(a,b,c)
> end
```

By deafult, the **size(M)** function has two outputs, the first is the number of rows, the second is the number of columns of the matrix M. The **size (M,1)** returns the number of rows and **size(M,2)** returns the number of columns. There are other similar functions, **length** returns the number of elements in a vector or the larger size of a matrix, and **numel** returns the number of all elements in the matrix or vector.

ITERATIONS – WHILE LOOP

The while loop allows commands to be executed repeatedly based on a given conditional expression. The commands inside the loop will be executed until the condition is true. The structure of a while loop is:

```
while (conditional expression)
  (matlab commands)
end
```

Let's look again our imaginary example: a subject is randomly scored in the exam between 0-100. The excellent mark is above 88%. We'll try the exam until we get five. Let's write a program that randomly generates our grades for each exam. How many exams needed to get an excellent mark?

```
> %% Loops - WHILE
> disp('How many exams do you need to get more than 88 %?')
> i = 0; point = 0;
> while point<=88
>     i=i+1;
>     point = rand()*100
> end
> i
```

FORMATTED STRINGS (FPRINTF, SPRINTF)

Our results often need to be presented in a specific format. Take, for example, the operation of angles. Most software that performs mathematical operations (e.g. Matlab, Octave, Excel ...) considers the radian as the default angle unit. In Matlab/Octave the trigonometric functions use radians by default (e.g. **sin, cos, tan, atan, atan2** ...) but these commands also have degree versions (e.g. **sind, cosd, tand, atand, atan2d** ...). However, if you want to display the results in degrees, minutes and seconds, in the format we use in geodesy (deg-min-sec), or up to a certain number of decimal places (23-03-48.5831), you need to use the so-called formatted texts. Similarly, if we want to automatically name saved images in a loop using the index in the file name e.g. IMG0001.jpg, IMG0002.jpg, etc. then we can also use formatted texts for this purpose.

You can use the **fprintf** command to write formatted text to a file or to the command window, and the **sprintf** command to save a formatted string. There is always a **%** sign in a formatted text, which indicates the variable to be formatted. We will have as many **%** signs in the text as many formatted numbers we need. You can use the following specifications to customize the format:

- **%d** – integer number

- **%s** – string
- **%f** – float - floating point number
- **%c** – character
- **%u** – unsigned integer
- **%e** or **%E** – normal form e.g. 3.14e+00,
- **%g** – compact form, i.e. the shorter from **%f** or **%e**, without the unnecessary zeros

Before the type specifier, you can add:

- **+** sign, to make it a signed value;
- number of characters;
- number of decimals;
- **0**, it will fill with zeros the undefined characters.

Let's try the following! The basic question is 'How old is the captain?'

```
clc; clear all; close all;
disp('How old is the captain?')
% some help: we know his birthday :)
birth = datetime(1984,02,28) %   datetime:  28-Feb-1984
today = datetime('now')   %   datetime: 12-Jul-2022 14:41:54
age = between(birth, today) % calendarDuration: 38y 4mo 14d 14h 41m
54.56s
[y,m,d] = datevec(age) % year = 38  month = 4  day =  14
% Octave has no datetime or between command!
% in Octave use this instead: y = 38; m = 4; d = 14;
yd = y + m/12 + d/365;
fprintf('The captain is 35 years old') % does not insert a linebreak
fprintf('The captain is 37 years old\r\n')% \r\n - linebreak
% The captain is 35 years oldThe captain is 37 years old
sprintf('The captain is 38 years old')   % results in text variable,
ans = 'The captain is 38 years old'
sprintf('The captain is %d years,%d months and %d days',y,m,d) % 'The
captain is 38 years,4 months and 14 days'
sprintf('The captain is %f years old', yd) %  'The captain is
38.371689 years old'
sprintf('The captain is %.2f years old', yd) % 'The captain is 38.37
years old'
sprintf('The captain is %8.2f years old', yd) %  'The captain is
38.37 years old'
sprintf('The captain is %08.2f years old', yd) % 'The captain is
00038.37 years old'
sprintf('The captain is %+6.2f years old', yd) %  'The captain is
+38.37 years old'
```

In the `%+6.2f` expression `f` represents a floating point number, `6` means field width (6 characters including decimal point and a sign), and `.2` denotes 2 decimal places. The `+` sign means that the sign symbol also appears for positive numbers. If `0` is included in the format, it fills the empty spaces with 0. If the result is longer than the field width, then the specified field width is ignored.

Let's look at the following function, which calculates and displays decimal degree angles in degrees-minutes-seconds in ddd-mm-ss format (e.g. 192-03-12)

```
> function str = dms(x);
> % Calculates and displays decimal degree angles in
> % degrees-minutes-seconds in ddd-mm-ss form used in geodesy
>   d = fix(x);
>   m = fix((x-d) .* 60);
>   s = round(((x-d).*60-m).*60);
>   str = sprintf('%3d-%02d-%02d', d, abs(m), abs(s));
> end
```

The **fix** function always rounds towards 0 (this is important because of the negative angles), the **round** function rounds towards nearest integer, the **floor** function rounds towards minus infinity and **ceil** function rounds towards plus infinity. At the end, we take the absolute value of minutes and seconds so that the negative sign is written only at the first place, before the degree value.

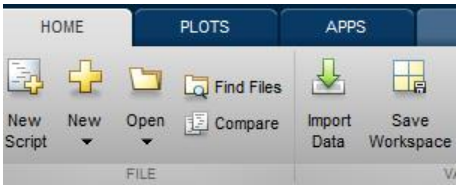
```
> a = 123.123, b = -123.123
> dms(a)  % '123-07-23'
> dms(b)  % '-123-07-23'
```

Replace the **fix** command to **floor** in dms function when calculating degrees (d), then save it and run the dms(a) and dms(b) commands again! What's happening?

DATA IMPORT/EXPORT

In engineering work, we often have to process the results of some kind of instrumental measurement. These results can be given in a text file in some specific format, so it is good to know how to extract information or numerical data from these files. Often, after a complex mathematical calculation, we need to present our result in another specific format for further use. Let's look at some examples of import and export commands to learn a little about file operations.

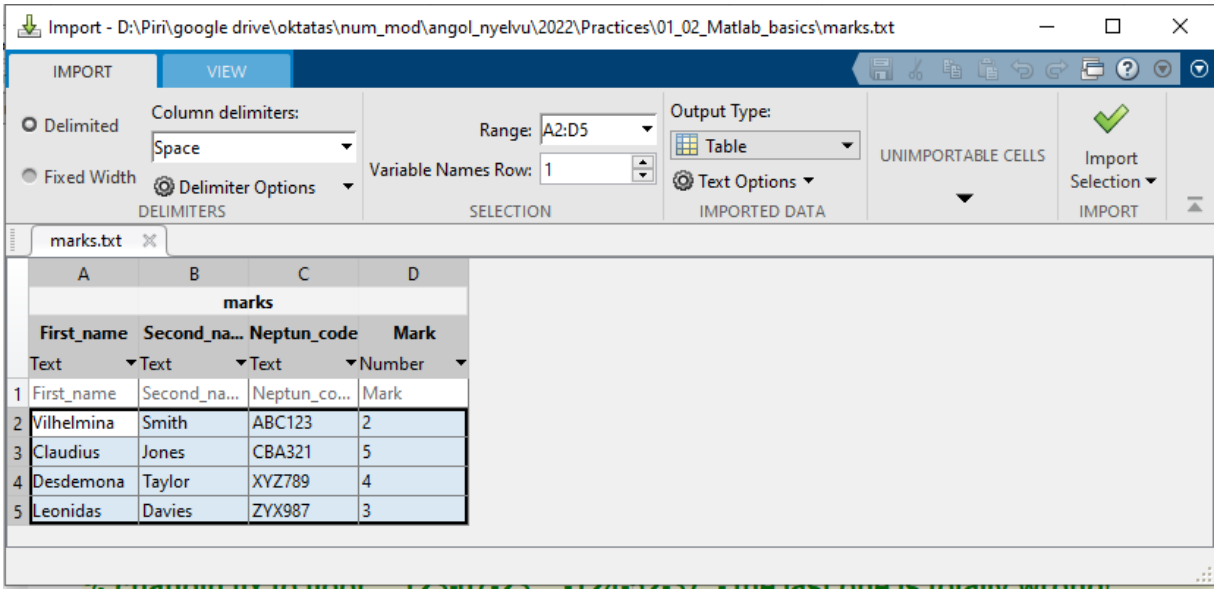
IMPORT DATA TOOL, DATA TYPES



One tool that can be used to import files is Matlab's own import tool, which can be accessed by clicking the 'Import Data' button on the Home tab. Use this tool to import the contents of the marks.txt file into Matlab!

```
First_name Second_name Neptun_code Mark
Vilhelmina Smith ABC123 2
Claudius Jones CBA321 5
Desdemona Taylor XYZ789 4
Leonidas Davies ZYX987 3
```

It is quite simple to use this tool, you only need to pay attention to some settings. You can specify the range of data, either in fixed-width columns or separated by a specific character. The thing to watch out for is the output type, which is a table by default. Other types can also be selected, e.g. Cell array, Numeric matrix. Now leave the default Table type and import the data by clicking on the green check mark (import selection). Then we can close the import window. Or you can use the '**readtable**' command from a script (the default column delimiters are spaces).




```
> % 'Import Data' tool
> clc; clear all; close all;
> marks = readtable('marks.txt'); % default column delimiters: spaces
> % Or use: Home Tab/ Import Data tool
> marks
> %       First_name      Second_name    Neptun_code    Mark
> %     ______________    ___________    ___________    ____
> %
> %     {'Vilhelmina'}    {'Smith' }     {'ABC123'}      2
> %     {'Claudius'  }    {'Jones' }     {'CBA321'}      5
> %     {'Desdemona' }    {'Taylor'}     {'XYZ789'}      4
> %     {'Leonidas'  }    {'Davies'}     {'ZYX987'}      3
```

These data will be of type 'Table', which can store different types of data at the same time, including texts and numbers (as well as Structure and Cell array types). Each column can be named, and a column can be referenced with its name written after the name of the Table and a dot.

```
> marks(1:2,1:3) % first 2 rows, 3 columns
> %   2×3 table
> %       First_name      Second_name    Neptun_code
> %     ______________    ___________    ___________
> %     {'Vilhelmina'}     {'Smith'}     {'ABC123'}
> %     {'Claudius'  }     {'Jones'}     {'CBA321'}
> name = marks.First_name % cella array of First_names column
> %   4×1 cell array
> %     {'Vilhelmina'}
> %     {'Claudius'  }
> %     {'Desdemona' }
> %     {'Leonidas'  }
> mark = marks.Mark %  vector of marks (numbers)
> %      2; 5; 4; 3
```

A similar form is the 'structure' data type, where a field can be referred to by its name. However, when using this type, it is not necessary to have the same number of rows in each field, unlike the table type. Different types of data can also be stored in a Cell array, but in this case there are no named fields. Individual elements can be referred to in the same way as in arrays, using their indices, but curly brackets {} should be

used instead of round brackets (). For example, first names are stored now in the 'name' cell array. Let's query the second one!

```
> name2 = name{2}  % 'Claudius'
```

BASIC IMPORT/EXPORT (LOAD, SAVE)

Now let's look at an engineering example, again the stress-strain diagram (σ-ε) of a reinforced concrete steel bar.

ε [%]	0	0.2	2	20	25
σ [N/mm^2=Mpa]	0	300	285	450	350

2. TABLE, STRESS-STRAIN DIAGRAM OF A STEEL BAR FOR REINFORCED CONCRETE

Our task is to create a table that contains the relative deformations between 0-25% in 0.1 percent increments and the correspondent stresses. Now we no longer enter the data manually, but read it from the steel.txt file:

```
0    0
0.2  300
2    285
20   450
25   350
```

This file contains only numbers, in 2 columns and 5 rows. Of course, we could also use the data import tool here, but it is worth changing the output type to a numeric matrix. However, in the case of text files containing only numbers in tabular format, there is also a simpler and more suitable solution, using the **load** command. For more complex formats, it may be worth reading the data line by line and processing each line separately.

Now let's look at the **load** and **save** commands as the simplest data read/write commands. Copy the file steel.txt to the current directory and load its contents into Matlab with the **load** command. There are two ways to call the **load** command, as a command or as a function The command form does not require parentheses or quotation marks. Command form:

```
> load steel.txt % command form
```

Using function form:

```
> data = load('steel.txt') % function form
```

Using the command form, a variable with the same name as the file name is created (in this case, steel). In the second version, by calling **load** as a function, we can assign the result to a new variable, but we should use brackets and quotation marks. Let's use this method now. Check the size and type of the new variable using the **whos** command!

```
> whos adat
> size(data)
```

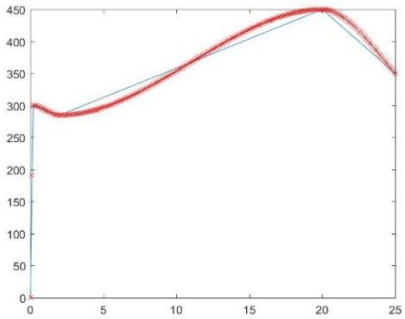
We got a 5x2 matrix. First, separate the variables (let x be the strain, y be the stress) and plot the (σ-ε) diagram.

```
> x = data(:,1); % first column - strain
```

```
> y = data(:,2); % second column - stress
> plot(x,y);
> xlabel('\epsilon');ylabel('\sigma');
```

To solve the original problem, we need to calculate stress values corresponding to deformations between 0-25% per 0.1 interval. This will require interpolation. We will use a cubic first-order spline interpolation (the theory is explained in later chapters). First, define a vector between 0-25% (max. deformation) with intervals of 0.1. Then calculate the stress values at these points by interpolation using the **interp1** command, with **'pchip'** method (piecewise cubic Hermite interpolating polynomial)!

```
> % cubic first-order spline
  interpolation
> xi = 0:0.1:max(x); % calculate points
  between 0-25%
> yi = interp1(x,y,xi,'pchip'); %
  interpolation
```



Plot the calculated points to the previous figure.,

```
> hold on;
> plot(xi,yi,'rx'); % 'rx' - red x-s
```

If you want to save the plotted graph as an image for illustration purposes, you can do this either from the Figure/File menu or with the **print** command.

```
> print('steel.jpg','-djpeg')
```

The variables xi and yi are row vectors (size: 1x271). We have to save them in a text file in tabular format, with the deformation in the first column and the stress in the second. To do this, we need to transpose the row vectors (') and then concatenate them with a simple matrix operation, since they have the same size.

```
> data2 = [xi' yi'];
```

The content can be written to a file with the **save** command. By default, Matlab saves files in its own binary *.mat extension, which cannot be loaded into any other program but Matlab.

```
> save('steel2.mat','data2')
```

To save the content to a text file, we should use the extra '**-ascii**' parameter.

```
> save('steel2.txt','data2','-ascii');
```

Note: **save** can be used in command format also:

```
> save steel2 data2
> save steel2.txt data2 -ascii
```

Let's open the saved text file!

```
0.0000000e+00   0.0000000e+00
1.0000000e-01   5.2521666e+01
2.0000000e-01   1.0943166e+02
3.0000000e-01   1.6158083e+02
4.0000000e-01   1.9982000e+02
…
```

The **save** command uses scientific notation as number format. If we want to print the numbers in a different format, e.g. up to 1 or 2 decimal places, we must use formatted text when saving the data.

FORMATTED FILE EXPORT (FPRINTF)

Create a text file from the interpolated strain-stress data, write the strain data to one decimal place and the stress data to 2 decimal places. To do this, you will need basic file handling instructions such as open, write, close files. Basic file handling instructions usually look like this:

- open file (**fopen**)
- read, write, append to file, process
- close file (**fclose**)

When using **fopen**, you can specify how to open the file, 'r' read-only (default if not specified), 'w' write, 'a' append, e.g.:

`fileID = fopen (filename, 'w')` - Open file for writing

You can close files individually: **fclose**(fileID), or all at once: **fclose**('all').

Write the data to a file using a **for** loop! Use 4 characters to one decimal place for deformation and 6 characters to 2 decimal places for the stress data. The **length** command returns the number of elements in the vector.

```
> n = length(xi); % vector length
> fid = fopen('diagramtable.txt','w');
> for i=1:n
>     fprintf(fid,'%4.1f %6.2f\r\n',xi(i),yi(i));
> end
> fclose(fid);
```

The problem can also be solved without a loop by using the data2 variable:

```
> fid = fopen('diagramtable2.txt','w');
> fprintf(fid,'%4.1f %6.2f\r\n',data2');
> fclose(fid);
> type diagramtable2.txt % print the content of file to the screen
```

Data2 variable has 2 columns and 271 rows, but **fprintf** must use its transposed form (2 rows and 271 columns) since **fprintf** reads its values column by column.

READING MEASUREMENT DATA LINE-BY-LINE (FGETL, FGETS)[2]

In engineering work, it often happens that measurements of a specific instrument need to be processed, which contain not only numbers but also texts. In order to process it, you must be able to read this data and select the part that interests you. Let's look at a navigation example! We have a GPS recorded route, stored in the NMEA 0183 format[3] used for navigation (hb_nmea.txt). Read the data and plot the route in a new figure. What vehicle was used to record this data?

```
$GPGLL,5156.9051,N,00117.1178,E*69
$GPGLL,5156.9194,N,00117.1482,E*61
…
```

In the NMEA standard, the word $GPGLL at the beginning of the line means that it contains GPS Geographic Latitude, Longitude information (there are many different

[2] To be read at home
[3] http://www.nmea.org/content/nmea_standards/nmea_0183_v_410.asp

NMEA messages). Fixed-length fields in the file are separated by commas, making this format easy to read and process. For latitude, the first two characters are the degree value, followed by the decimal minutes. For longitude, the first three characters are the degree value, followed by the decimal minutes (since the former extends to ±90°, and the latter to ±180°). For latitude values N means North, S means South, for longitude E stands for East and W stands for West. For example, 5156.9051, N means north latitude 51° 56.9051'.

This file has a more complex structure, the simple load function cannot be used. You should use Matlab's low-level input/output functions instead. Before reading, open the file with **fopen**, and obtain a file identifier (fid). By default, **fopen** opens files for read access. When you have finishes processing the file, close it with **fclose**(fid). For processing, in this case, it is useful to know the line-by-line file reading commands: **fgetl**, **fgets**. **fgetl** reads a line and cuts off the end-of-line character, while **fgets** keeps it. The result is stored in a string variable. To read the entire contents of the file, a conditional loop (**while**) will be needed until we reach the end of the file (**feof** - end-of-file).

Lets just read the first line, and try to get the data for the route. Note: After opening the file, a file pointer monitors how many bytes of the file have been read, which can be queried with the **ftell**(fid) command.

```
> fid=fopen('hb_nmea.txt');
> line=fgetl(fid) % read one line
> % $GPGLL,5156.9051,N,00117.1178,E*69
```

The result will be a string variable containing the first line. Filter the information we are interested in, latitude (lat) and longitude (lon)! To do this, you need to know that characters #8-9 are the degree values, #10-16 are the minutes of latitude, #20-22 are the degres, and #23-29 are minutes of longitude. A certain indexed element in a string can be obtained the same way as an indexed element in a vector since strings are character vectors in Matlab!

```
> lat_deg = line(8:9),   lat_min = line(10:16),
> lon_deg = line(20:22), lon_min = line(23:29),
> % lat_deg =  '51'; lat_min = '56.9051'
> % lon_deg = '001'; lon_min = '17.1178'
```

Convert the values to decimal degrees! First, you need to convert the strings to numbers using the str2num command.

```
> lat = str2num(lat_deg)+str2num(lat_min)/60  % 51.9484
> lon = str2num(lon_deg)+str2num(lon_min)/60  % 1.2853
```

Now read the characters N/S and E/W to determine which hemisphere the coordinate is in: character #18 stands for N (north) or S (south), #31 stands for E (east) or W (west). The letters S and W gives a negative sign to the corresponding coordinate. If necessary, change the sign with an if conditional structure.

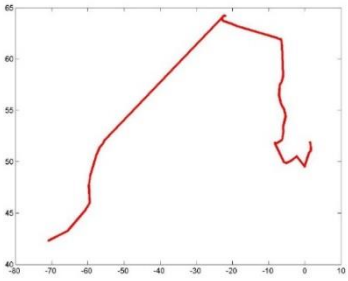
```
> NS = line(18);   if NS=='S'; lat=lat*-1; end;
> EW = line(31);   if EW=='W'; lon=lon*-1; end;
```

Thus, for example, relevant information can be extracted from a more complicated structure. Of course, MATLAB has many other built-in functions to handle various

inputs/outputs, if you are interested, check the help guide for more details with the **help iofun** command.

Now let's process the whole file in one go. This will require a condition-controlled loop (**while** loop). In this case, the condition is to check whether the process reached the end of the file or not? The variabl **feof(fid)** is 1 at the end of the file and 0 before it. Therefore if feof (fid) == 0 the loop will run. You will need two more vector variables (LAT,LON) where you can store the acquired coordinates. They should be initialized as empty vectors at the beginning and simply append the acquired coordinates in each cycle. Put a semicolons (;) at the end of the lines to avoid displaying partial-results! The whole process:

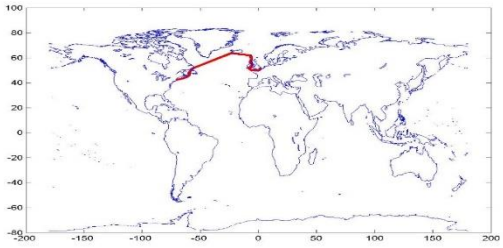
```
> LAT = []; LON = [];
> fid=fopen('hb_nmea.txt');
> while feof(fid)==0
>   line=fgetl(fid); % read a line
>   % aquire latitude, longitude
>   lat_deg = line(8:9);   lat_min = line(10:16);
>   lon_deg = line(20:22);  lon_min = line(23:29);
>   % convert to decimal degree
>   lat = str2num(lat_deg)+str2num(lat_min)/60;
>   lon = str2num(lon_deg)+str2num(lon_min)/60;
>   % signs
>   NS = line(18);   if NS=='S'; lat=lat*-1; end;
>   EW = line(31);   if EW=='W'; lon=lon*-1; end;
>   % append subresults to LAT,LON vectors
>   LAT = [LAT; lat]; LON = [LON; lon];
> end
> fclose(fid);
```



Plot the route in a new figure with a thick red line!

```
> figure(2)
> plot(LON, LAT,'r','LineWidth',3)
```

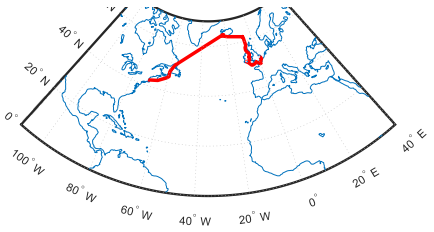
Based on the figure, it would be difficult to decide where the vehicle was going, in order to facilitate localization we also plot the coastlines in blue.



```
> coast = load('coastline.txt');
> hold on;
  plot(coast(:,1),coast(:,2),'b')
```

What kind of vehicle could it be?

Alternative plot:



```
> figure
> worldmap([0 70],[-110 40])  %
  worldmap('World')
> load coastlines
> plotm(coastlat,coastlon)
> hold on
> plotm(LAT, LON,'r','LineWidth',3)
```

USED MATLAB BUILT-IN FUNCTIONS

help	-	matlab help categories, or help of a specific topic or function

rand	-	Random numbers between 0-1, evenly distributed
randn	-	Random numbers in standard normal distribution with 0 expected value and 1 standard deviation
doc	-	detailed documentation for a given function, in a new window
lookfor	-	search for a word, word fragment in help
clc	-	clears the contents of the command window
clear, clear all	-	deletes the specified variables or all variables
close, close all	-	closes the current graph or all
CTRL+C	-	interrupts the given command (e.g. exit an infinite loop)
%	-	comment (the program ignores what is next in the line)
;	-	at the end of the command; the result will not appear in the Command Window
Tab gomb	-	completes a command that has been started
preferences	-	opens the settings window
prefdir	-	the name of the directory where the settings, history, etc. are stored
↑↓ buttons	-	previous commands can be returned to the Command Window
pi	-	3.14… (pi number)
exp(1), exp(n)	-	e^1 = 2.71…, e^n, Euler's number
^	-	exponentiation
format long, format short	-	displaying multiple (14) or less decimal digits (4)
format shortG, format longG	-	displays numbers of different magnitudes in compact format (5 or 15 significant digits).
[1, 2, 3; 4, 5, 6]	-	vector/matrix definition
'	-	transposed vector/matrix
[A,B] vagy [A B]	-	join matrices side by side (equal number of rows)
[A;B]	-	join matrices under each other (equal number of columns)
A(1,:)	-	first row of matrix
A(:,1), A(:,end)	-	first / last column of matrix
linspace(x1,x2,n)	-	Vector between [x1, x2] with n points evenly distributed
ones	-	matrix of ones
zeros	-	matrix of zeros
eye	-	identity matrix
figure	-	Open a new graph
plot	-	drawing point vectors

xlabel, ylabel	-	x, y axis annotation
title	-	Figure title
sin, cos, tan	-	angle functions (default unit: radian)
log, log10	-	natural logarithm, 10 based logarithm
sqrt	-	square root
abs	-	absolute value
hold on, hold off	-	overwrite or do not overwrite the existing figure with the new one
fplot, ezplot	-	plotting functions
.* ./ .^	-	multiplication, division, power with vectors element-by-element
clf	-	deleting elements of the figure (does not close the window)
legend	-	legend
return	-	return point - F5 executes the program only up to this point
==	-	Equality, logical operator
~=	-	Inequality, 'not equal', logical operator
&&	-	Logical AND
||	-	Logical OR
disp	-	Print string or string variable contents to Command Window
if, elseif, else, end	-	Two-way conditional branch
switch, case	-	Multi-directional branch
for	-	Counting controlled loop
while	-	Conditional controlled loop
size	-	The number of rows, columns of a matrix
length	-	The number of elements of a vector, or the larger size of a matrix
numel	-	Total number of elements in a matrix/vector
randi	-	Generate random integers
fprintf	-	Write formatted texts to a file or screen
sprintf	-	Write formatted texts to a string variable or screen
\r\n	-	End of line symbol for formatted text
fix	-	Rounding function, rounds towards 0
round	-	Rounding function, rounds towards nearest integer
floor	-	Rounding function, rounds towards minus infinity
ceil	-	Rounding function, rounds towards plus infinity
readtable	-	Create a table by reading from a file
load	-	Loading data (from Matlab file (*.mat) or plain text file)
save	-	Save data (to Matlab file (*.mat) or plain text file)
print	-	Save the content of the figure to a file

interp1	-	One-variable interpolation
fopen	-	Open file
fclose	-	Close file
type	-	List the contents of a text file in the Command window
fgetl	-	It reads a line and cuts the end of line character from it.
fgets	-	It reads a line and retains the end of line character.
feof	-	End-of-file
ftell	-	Pointer to check how many bytes of the file have been read
str2num	-	Converts text to number
atan, atan2	-	Inverse tangent function, result in radians.
sind, cosd, tand, atand, atan2d	-	Trigonometric functions working with degrees