2. CONTOL FLOW STUCTURES, DATA IMPORT/EXPORT

LOGICAL OPERATIONS

Some basic knowledge of logical operations (1-true / 0-false) is also very important using Matlab, especially when modifying and querying elements of vectors/matrices. There are many problems that can be solved with matrices and logical variables that would require a loop in other program 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 we query a given property of a vector with a logical variable. Imaginary let's have a university professor who likes to grade the students randomly on the 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 (got mark 1) and exactly who in a given 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 was true, 0 in the other places. If you want to retrieve the names of the candidates who have failed, all you have to do is call the 'students(failed)' command, it will only return the names from students vector, where the value of 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.

CHOICES, LOOPS

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 rows), end; There may 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!

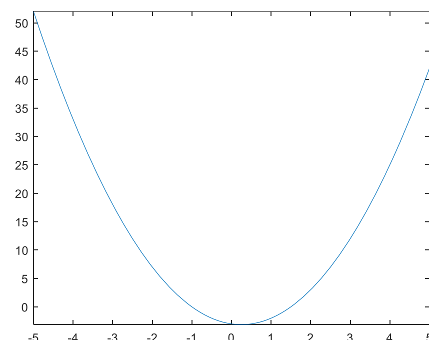
$$2\,x^2 - x - 3 = 0$$

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

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

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**!

```
> a = 2, b = -1, c = -3
> f = @(x) a*x.^2+b*x+c;
> figure; fplot(f);
```

First, we have to check if there is any real solution or not, if there is any then there are 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! To do so, one must consider the different cases of the discriminant D = b ^ 2-4ac. 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],'r*')
>     elseif D==0
>         disp('The equation has 1 real solution')
>         x = -b/(2*a);
>         hold on; plot(x,0,'r*')
>     else
>         disp('The equation has no real solution')
>         x = [];
>     end
> end
```

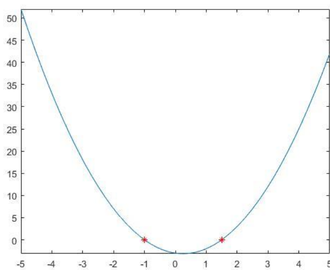
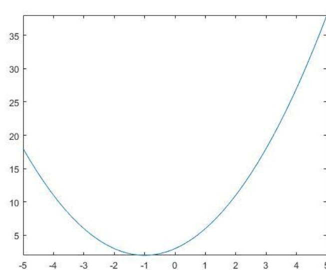
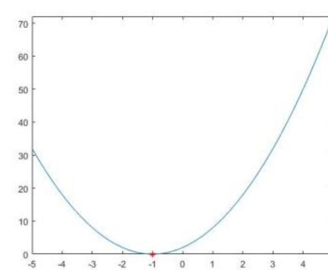
The **disp** command will print a text to the command window.

The functions, compared with the scripts, cannot be run by using 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 where you are working.)

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

However, it is better to work to a script file that can be easily modified later. Go 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 copy the functions to the end of the script file also, 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), when running only a section (with F9) Matlab will not find the function at the end of the script file!

SWITCH STATEMENT

We can use not only two-way branches but also multidirectional ones in more complex cases. Write a program that helps a teacher to grade the students randomly! 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. Try this at 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 command is repeated several times consecutively. In the for loop the number of repetition is predetermined. The structure of a for-end loop is:

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

(Matlab commands)

end

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

Let's see how we can solve the equations described in the first example using a loop if we store the coefficients 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
```

The **size(M)** function has two outputs in default, the first is the number of rows, the second is the number of columns of the M matrix. 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)

It is often necessary to present our results 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 to be the default angle unit. In Matlab/Octave the trigonometric functions use radians as default (e.g. **sin, cos, tan, atan, atan2** ...) but they also have a degree-variant (e.g. **sind, cosd, tand, atand, atan2d** ...), but if you want to display the results in degrees, minutes and seconds, in a format we use in geodesy (deg-min-sec format), or 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 pictures in a loop using the index in the file name e.g. IMG0001.jpg, IMG0002.jpg, etc. then we can use formatted texts for this purpose also.

You can use **fprintf** command to write formatted text to a file or to the command window, and **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 needed. To customize the format, you can use the following specifiers:

- **%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 specifier that determines the type, 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; disp('How old is the captain?')
> % some help: we know his birthday :)
> % Octave has no datetime or between command!
> % in Octave use this instead: y = 35; m = 5; d = 2;
```

```
> birth = datetime(1984,02,28)
> today = datetime('now')   % (at 30.07.1997.)
> age = between(birth, today) % 35y 5mo 2d 13h 58m 47.086s
> [y,m,d] = datevec(age) % ev = 35  ho = 5  nap =  2
> yd = y + m/12 + d/365;
> fprintf('The captain is 35 years old') % does not insert a linebreak
> fprintf('The captain is 35 years old'\r\n')% \r\n - linebreak
> sprintf('The captain is 35 years old')   % results in text variable
> sprintf('The captain is %d years,%d months and %d days',y,m,d) % 'The
  captain is 35 years, 5 months and 2 days'
> sprintf('The captain is %f years old', yd) % 'The captain is
  35.422146 years old'
> sprintf('The captain is %.2f years old', evt) % % 'The captain is
  35.42 years old'
> sprintf('The captain is %8.2f years old', evt) % 'The captain is
  35.42 years old'
> sprintf('The captain is %08.2f years old', evt) % 'The captain is
  00035.42 years old'
> sprintf('The captain is %+6.2f years old', evt) % 'The captain is
  +35.42 years old'
```

In the `%+6.2f` expression `f` denotes 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 will be displayed in case of positive numbers also. If `0` is included in the format, it will fill in the blank 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?

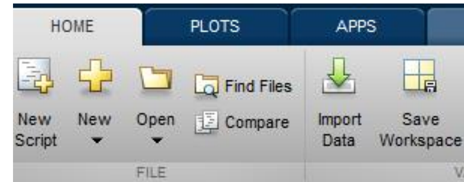
DATA IMPORT/EXPORT

In engineering work, we often have to process the results of some instrumental measurement. These results can be given in a text file in some specific format, so it is good to know how we can obtain the information or numeric data from these files.

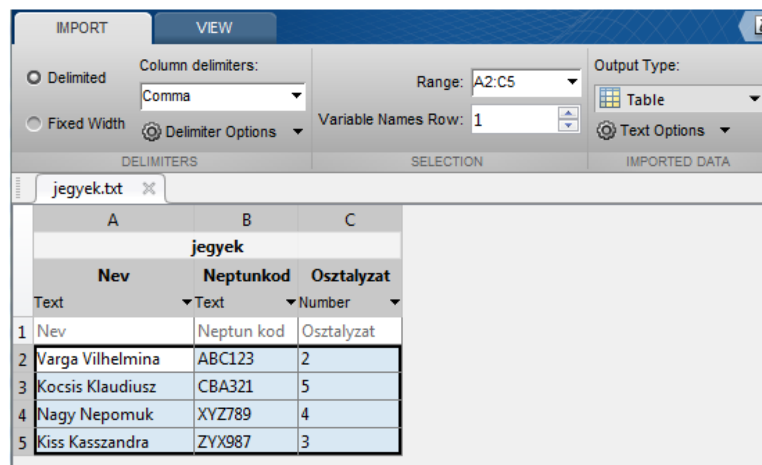
Often, after a complex mathematical calculation, we have to present our result in another specific format for further use. Let's look at some examples of import and export commands to get a little acquainted with file operations.

IMPORT DATA TOOL, DATA TYPES

One tool you can use to import files is Matlab's own import tool, which can be accessed by clicking the 'Import Data' button on the Home tab. Let's import the content of the marks.txt file into Matlab with this tool!



Its use is simple enough, you just have to pay attention to the settings. You can specify the range of your data, whether it is in fixed-width columns, or separated by a specific character. What is important to take care of is the Output type, which is Table by default. Other types can be selected, e.g. Cell array, Numeric matrix. Leave now the default Table type and import the data by clicking on the green check mark (import selection). Then we can close the import window.




```
> %% 'Import Data' tool
> % jegyek.txt -> table forma
> clc; marks
> %   4x3 table
> %          Name              Neptuncode    Mark
> %     ___________________    __________    ____
> %
> %     "Vilhelmina Smith "    " ABC123"      2
> %     "Claudius Jones"       " CBA321"      5
> %     "Desdemona Taylor"     " XYZ789"      4
> %     "Leonidas Davies"      " ZYX987"      3
```

These data will be in 'Table' type, 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 point.

```
> marks(1:2,1:3) % first 2 rows
> name = marks.Name % cella array of Names column
> mark = marks.Mark % number vector of marks
```

A similar form is the 'structure' data type, where you can refer also to a field by its name. However, using this type it is not mandatory to have the same number of rows for each field as for the table. We can store different types of data in the Cell array also, but in this case, nothing is named. You can refer to each element in the same way as in matrices, using their indexes, but you have to use curly brackets {} instead of round brackets (). For example, names are stored in a cell array. Let's ask the second one!

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

BASIC IMPORT/EXPORT (LOAD, SAVE)

Let us now look at an engineering example, again the 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

Our task is to produce a table that contains the strains and stresses from 0 to 25% relative deformation at every 0.1 percent. Now we will not enter the data manually, but will 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, here, too, we could use the import data tool, but we should change the output type to numeric matrix. However, in case of text files containing only numbers in tabular format, there is a more simple and suitable solution, using the **load** command. In more complex formats, it can be worthwhile to read the data line by line and process every line separately.

Now let's look at the **load** and **save** commands as the simplest data read/write commands. Copy the steel.txt file to the current directory and load its contents to Matlab with **load** command. There are two syntax forms, the command form, and the function form. In the command form there is no need for brackets or quotes.

```
> load steel.txt
```

Using function form:

```
> data = load('steel.txt')
```

Using the command form a variable with the same name as the filename (steel) will be created. 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 quotes. Let's use this method now. Check the size and type of the new variable using the **whos** command!

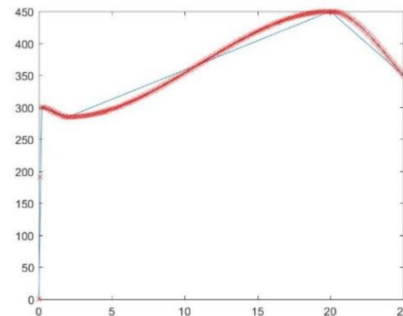
```
> whos adat
> size(data)
```

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

```
> x = adat(:,1); % first column - strain
> y = adat(:,2); % second column - stress
> plot(x,y);
> xlabel('\epsilon');ylabel('\sigma');
```

To solve the original problem, we should calculate the stress values for each deformation between 0-25% using 0.1 intervals. This will require interpolation. We will use a cubic first-order spline interpolation (the theory will be described in later chapters). First, let's determine a vector between 0-25% (max. deformation) with 0.1 intervals, 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
```



Draw 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 to an image for illustration purposes, you can do this either from the Figure/File menu or using **print** command.

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

The variables xi and yi are row vectors (size: 1x271). We should save them to a text file in table 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'];
```

We can use the **save** command to write the content to a file. By default, Matlab saves files in its own binary *.mat extension, which cannot be loaded into another program, only to 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. to 1 or 2 decimal places, we need to use formatted text when saving the data.

FORMATTED FILE EXPORT (FPRINTF)

Let's create a text file from the interpolated strain-stress data, write the strain data to one decimal places and the stress data to 2 decimal places. For this you will need basic file management instructions like opening, writing, closing files. The basic file management instructions generally 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 nothing is 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! Let's use 4 characters to one decimal place for the 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 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, however with **fprintf** we should use its transposed form (2 rows and 271 columns) because **fprintf** reads the associated values per column.

READING MEASUREMENT DATA LINE-BY-LINE (FGETL, FGETS)[1]

In engineering work, it is often the case that measurements of a given instrument need to be processed, which include not only numbers but also texts. For processing, we need to be able to read this data and select the part that interests us. Let's now look at a navigation example! We have a GPS recorded route, stored in the NMEA 0183 format used for navigation (hb_nmea.txt). Read the data and plot the route in a new figure. What vehicle had they used to record this data?

`$GPGLL,5156.9051,N,00117.1178,E*69`

[1] Homework

```
$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 NMEA messages). In the file fix length fields are separated by commas, so this format is easy to read and process. For the geographic latitude, the first two characters are the degree values, followed by decimal minutes, in case of the longitude, the first three characters are the degree values, followed by decimal minutes (as the former extends to ±90°, and the latter to ±180°). For the latitude values N means North, S means South, in case of the 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. Before reading, you should open the file with **fopen**, and obtain a file identifier (fid). By default, **fopen** opens files for read access. When you finish processing the file, close it with **fclose**(fid). For the 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 ending line character, while **fgets** keeps it. The result is stored in a string variable. To read the entire contents of the file, it will require a conditional loop (**while**) to read until we reach the end of the file (**feof** - end-of-file).

Lets read just the first line, and try to acquire the relevant 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 out 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 for latitude, #20-22 are the degree values, and #23-29 are the minutes for longitude. A certain indexed element in a string could be acquired the same way as an indexed element in a vector because 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);
```

Let's convert the values into decimal degrees! First, you need to convert the strings to numbers with 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 in the N/S and E/W characters to determine on which hemisphere is the coordinate: character #18 stands for N (north) or S (south), #31 stands for E (east) or W (west). Letter S and W gives a negative sign to the relevant coordinate. If needed, change the sign using if conditional structure.

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

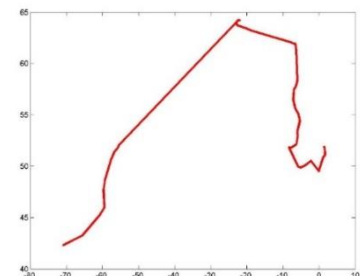
This way, for example, it is possible to extract the relevant information from a more complex structure. There are of course a lot of additional built-in functions in MATLAB to handle different inputs/outputs, if you are interested in, just check the help guide for further details using the **help iofun** 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 **feof(fid)** variable 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. You should initialize them at the beginning as empty vectors, and in every loop simply append the acquired coordinates. Put semicolons (;) at the end of the lines to avoid displaying each sub-result! 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);
```

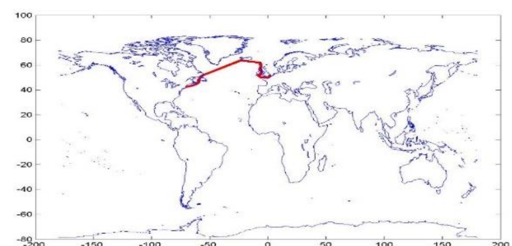
Plot the route in a new figure with a thick red line!

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



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

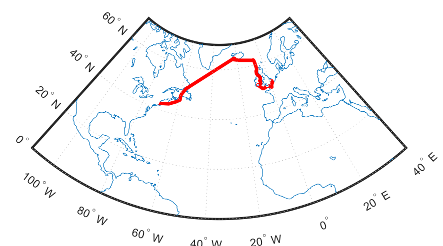
```
> coast = load('coastline.txt');
> hold on;
  plot(coast(:,1),coast(:,2),'b')
```



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)
```



NEW BUILT-IN FUNCTIONS USED IN THE CHAPTER

==	-	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
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