Functions in MATLAB

The arguments of functions are always inside the regular parenthesis, and while you are typing MATLAB is constantly giving you hints what kind of additional inputs can you add. Use the help guide anytime by pressing F1 on the function!

\[ \sin(\pi) \quad \% \text{result: 0 (or at least within the bounds on floating point representation error)} \]
\[ \text{ans} = 1.2246e-16 \]

\[ 3^4 \quad \% \text{result: 81} \]
\[ \text{ans} = 81 \]

\[ \exp(0) \quad \% \text{this is } e^0, \text{ which is 1. } e \text{ means the euler number 2.7183} \]
\[ \text{ans} = 1 \]

The built-in functions are also working on vectors:

\[ x = \text{linspace}(0, 2\pi, 40) \]
\[ x = \]
\[ \begin{array}{ccccccccc}
0 & 0.1611 & 0.3222 & 0.4833 & 0.6444 & 0.8055 & 0.9666 & 1.1278 & 1.2889 & 1.4500 & 1.6111 & 1.7722 & 1.9333 \\
\end{array} \]

\[ y = \sin(x) \]
\[ y = \]
\[ \begin{array}{ccccccccc}
0 & 0.1604 & 0.3167 & 0.4647 & 0.6007 & 0.7212 & 0.8238 & 0.9035 & 0.9605 & 0.9927 & 0.9992 & 0.9798 & 0.9350 \\
\end{array} \]

\[ \text{figure} \]
\[ \text{plot}(x, y, 'bx-') \]

You can also write your very own functions, and you have more options. For the more simple functions you can create anonymous functions, that are not saved as a separate program; they are just assigned to a variable! For example, if you want to create a function for the following:

\[ f(x) = 2x^2 - 3x + 1 \]

The MATLAB code will look like this:

\[ f = @(x) 2\times^2 - 3\times + 1 \]

\[ f = \text{function handle with value:} \]
\[ @(x)2\times^2-2.3\times+1 \]
This means the function is assigned to the variable \( f \), and it has a single argument. You define the input parameters with the @ sign, and you can put multiple arguments inside the parenthesis if you wish.

If you want to call the function, you can refer to its variable, as follows:

\[
\begin{align*}
f(1.2345) \\
\text{ans} &= 0.3445
\end{align*}
\]

This function works also on vectors, not only on single values! Its because we added a dot operator before we squared \( x \). If you want to make your own function to work similarly, you should add a "." before multiplication, division and in exponential expressions as follows:

\[
\begin{align*}
\ast & \quad \div & \quad ^\wedge
\end{align*}
\]

If you add/subtract, or multiply your inputs by a scalar value, this is not necessary.

If you want to write a separate function, you should apply the following structure in a new script file:

```matlab
function [output1, output2] = functionname(input1, input2, ...)
    output1 = formula1
    output2 = formula2
end
```

```matlab
x1 = -2:0.1:2;
y1 = probafv(x1);
clf
plot(x1, y1)
```

About the functions:

- It should start with the keyword `function`
- It should have at least one input and one output
- In the first row you should have the output, the functions name, and the input; the functions name should be the same as, the *.m file name
- Inside the function you should assign a value to the output
- The variables defined inside the functions are local variables, those won't appear in the workspace, also the running function has no access to the variables inside the workspace, only if you added them, as an input

Your own function could have multiple inputs, just modify the previous function as follows:

```matlab
function y = probafv2(x,p)
    y = 2*x.^p - 3*x + 1;
end
```
Save it as a separate function, with its name, and recall it:

```matlab
y1 = probafv2(x1,3);
plot(x1,y1)
```

A function could have multiple outputs as well, defined as a vector:

```matlab
function [x2, x3, x4] = hatvany(x)
    x2 = x.^2;
    x3 = x.^3;
    x4 = x.^4;
end
```

Save this as a separate file and lets present the results on a figure:

```matlab
x = -1:.1:1

[x] = hatvany(x);
plot(x,x,x2,x, x3, x, x4)
```
On the figure you will see each element with separate colors, but you can customize them on your own too:

```matlab
plot(x,x,'black',x,x2,'blue',x,x3,'green',x,x4,'red')
```

You can add additional information too e.g. legend, title, axis label as follows:

```matlab
legend('x','x^2','x^3','x^4','Location','SouthEast') % adding legend
title('Power functions') % adding title for the figure
xlabel('x'); % adding labels for the horizontal axis
ylabel('Functions value') % adding labels for the vertical axis
```

**Basic control flow structures**

**Branching point - IF statement**

In our calculations we need to apply different formulas based on certain conditions. Here is a small summary about the possibilities for this:

**If statement:**

You should apply the following form:

```matlab
if (condition)
```
(matlab commands)

end

This will execute the matlab command between the if - end keywords, if the condition is true. This condition is a logical test with an output of 1 (TRUE) or 0 (FALSE).

Summary for logical tests:

```matlab
x=2.5;

% Simple logical tests:
% Output: IGAZ = 1 (TRUE) or HAMIS = 0 (FALSE)
   x > 2          % is x greater than 2?
   ans = logical
   1

   x < 2          % is x less than 2?
   ans = logical
   0

   x >= 2         % is x greater or equal than 2?
   ans = logical
   1

   x <= 2         % is x less or equal than 2?
   ans = logical
   0

   x == 2         % is x equal to 2?
   ans = logical
   0

   x ~= 2         % is x anything else than 2?
   ans = logical
   1

% Combined logical tests - you can combine the previous logical tests using logical operators:
% These are the & (AND), | (OR), and ~ (NOT) operators
   (x > 2) && (x > 3)  % both test should be TRUE (AND)
   ans = logical
   0

   (x > 2) || (x > 3)  % at least one test should be TRUE (OR)
   ans = logical
   1

   (x > 2) & ~(x < 3)  % the first condition should be TRUE and the second NOT
   ans = logical
   0

% These are working on vectors/matrices too:
   v=1:20;
   v > 5

   ans = 1x20 logical array
   0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1

% The output is a logical list describing which element satisfied the condition, and which not
% You can even use this kind of logical indexing to select the specified elements:
   v(v>5)
```
Let's check a simple decision:

```matlab
% If the value of x is less than 0, let's redefine it as 0:
ans = 6
x=-6;
if x<0
    % if the condition is true, than the command between the if-end keywords will be processed
    disp(['x=',num2str(x),', which is less than 0'])
    % redefining x:
    x = 0;
end
```

`x=-6`, which is less than 0

If we want to define what happens, if the condition was false, we should apply the following form:

**If - Else statement:**

This works with the following form:

```matlab
if (condition)
    % (matlab commands)
else
    % (matlab commands)
end
```

If the condition was false, only those commands will be executed, which are between the `else-end` keywords:

```matlab
% If the value of x is less than 0, let's redefine it as 0, otherwise write out a small message:
ans = 6
x=-6;
if x<0
    % if the condition is true, than the command between the if-end keywords will be processed
    disp(['x=',num2str(x),', which is less than 0'])
    % redefining x:
    x = 0;
else
    disp(['x=',num2str(x),', that is greater than 0, so we don''t have to redefine'])
end
```

`x=-6`, which is less than 0

If the decision tree is more complex, you can apply further possibilities with the `elseif` keyword instead of `else`. You can make various choices with that, but a better alternative is to use the switch case statement:

**Switch - case**

Let's check a random classification task! Function `randi(n)` can generate integers between 1 and the given input (n). Generate a number between 1-5 and display a message based on the result:

```matlab
disp('Exam mark: ')

Exam mark:

mark = randi(5);
switch mark
    case 1
        disp('Failed')
    case 2
        disp('Passed')
    case 3
        disp('Satisfactory')
    case 4
        disp('Good')
    case 5
```
disp('Excellent')
end

Excellent

**Iterations - For loop**

This allows the code to be executed repeatedly, and its typically used when the number of iterations is known before entering the loop. You can apply it in the following form:

```matlab
for iteration_variable = initial_value:end_value
    (matlab commands)
end
```

You can use this for repeated processes, or indexing through the element of a vector/matrix:

```matlab
for i=1:3
    i*2
end
```

ans = 2
ans = 4
ans = 6

```matlab
a = 2;
for i = 1:5
    a = a*2
end
```

a = 4
a = 16
a = 256
a = 65536
a = 4.295e+09

```matlab
v= [10:10:50]'
```

```matlab
v =
  10
  20
  30
  40
  50
```

```matlab
for i = 1:length(v)
    v(i)=v(i)+3
end
```

```matlab
v =
  13
  20
  30
  40
  50
```

```matlab
v =
  13
  23
  30
  40
  50
```

```matlab
v =
  13
  23
  33
  40
  50
```

```matlab
v =
```
Iterations - While loop

This allows the code to be executed repeatedly, and contains a condition. The commands inside the loop will be executed till the condition is true. You can apply it in the following form:

```matlab
while condition
    (matlab commands)
end
```

Let's check a simple task! Generate random numbers below 100, till we reach a specified value! Also count how many tries was necessary to achieve it!

```matlab
i = 0; number = 0;
while number<=88
    i=i+1;
    number = randi(100)
end

number = 91
```

1

1 = 1

Import Data

You can use the Matlab's import tool (Home/Import Data) to load in datasets. It is really simple, you can set various parameters (delimiters, variable names, data types, range, etc.) what really important is: be careful with the output! Let's load the `marks.txt` data with the default table settings:

Let's check what's inside! The table type variable can contain multiple type of data (Cells, Numeric Arrays, Strings, etc)

```matlab
marks
```

<table>
<thead>
<tr>
<th></th>
<th>Name</th>
<th>Neptunco</th>
<th>Marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;Varga Vilh...&quot;</td>
<td>&quot;ABC123&quot;</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>&quot;Kocsis Kla...&quot;</td>
<td>&quot;CBA321&quot;</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>&quot;Nagy Nep...&quot;</td>
<td>&quot;XYZ789&quot;</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>&quot;Kiss Kassz...&quot;</td>
<td>&quot;ZYX987&quot;</td>
<td>3</td>
</tr>
</tbody>
</table>

```matlab
marks(1:2,1:3)
```

<table>
<thead>
<tr>
<th></th>
<th>Name</th>
<th>Neptunco</th>
<th>Marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;Varga Vilh...&quot;</td>
<td>&quot;ABC123&quot;</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>&quot;Kocsis Kla...&quot;</td>
<td>&quot;CBA321&quot;</td>
<td>5</td>
</tr>
</tbody>
</table>

```matlab
name = marks.Name % cell array
```

```matlab
name = 4x1 string array
"Varga Vilhelmina"
"Kocsis Klaudiusz"
"Nagy Nepomuk"
"Kiss Kasszandra"
```

```matlab
mark = marks.Marks % numeric vector
```
Saving your data/results

```matlab
clear all; clc; close all
% this only works, if you see the specified file in the current folder, otherwise you should add the full path
xy = load('stressstrain.txt');
% xy = [0, 0.2, 2, 20, 25; 0, 300, 285, 450, 350];
x = xy(:,1);
y = xy(:,2);
plot(x,y,'bo');
xlabel('\sigma');
ylabel('\epsilon');

Lets add a cubic spline interpolated function to the plot:

this is how you can define it in the script, and apply it more automatically, but in a Figure window, you can also use the Tools/Basic Fitting option

```matlab
xi = 0:0.1:max(x); % adding intermediate points
yi = interp1(x,y,xi,'cubic'); % interpolation
```

Warning: INTERP1(...,'CUBIC') will change in a future release. Use INTERP1(...,'PCHIP') instead.

```matlab
hold on;
plot(xi,yi,'r');
```

If you want to save the figure, its often better to use the print function with additional parameters, rather than to use the options in the figure window:

```matlab
print('stressstrain.jpg','-djpeg');
```

Saving the result data:

```matlab
result = [xi' yi'];
```
save('stressstrain2.mat','result')
save('stressstrain2.txt','result','-ascii');

If you use the `save` function, your output file will contain the numbers in normal form. If you want to save the results in a more conventional way e.g. a previously defined number of decimals, you should apply alternative methods, and create pre-formatted data.

The most software designed for mathematical processes (Matlab, Octave, Excel, etc.) are using radians by default to handle angles, and if you want to use something else, you need to work on it a bit more. In Matlab (or in Octave) you have for the trigonometric functions (sin, cos, tan, atan, etc.) an alternative to calculate with degrees (sind, cosd, tand, atand, etc.). If you wish to present the results in deg-min-sec format, as used conventionally in geodesy (123-45-67 or with a determined number of decimals e.g. 123-45-67.8910), you should create formatted strings. Formatted strings are multi-purpose, if you want to save several figure automatically in a loop, you can create with it the filename too using the loop variable (e.g. IMG0001.jpg, IMG0002.jpg etc.).

For this you can apply the `fprintf` (which can write inside files or on the screen) or the `sprintf` (which can create a string, or write on the screen) function. To customize the format, you can use the following specifiers:

- `%d` - digit
- `%s` - string
- `%f` - float - floating point representation of numbers
- `%c` - character
- `%u` - unsigned integer
- `%e` - normal form e.g. 3.14e+00. (exp)
- `%E` - 3.14E+00
- `%g` - compact form, i.e. the shorter out of `%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 followings:

```plaintext
fprintf('pi') % pi>> - without line break
pi
fprintf('pi') % pi - with line break on screen
ans = 'pi'
fprintf('%f',pi) % 3.141593 - pi represented as float
ans = '3.141593'
fprintf('%2f',pi) % 3.14 - float, with two decimals
ans = '3.14'
fprintf('%6.2f',pi) % 3.14 - 2 decimals in 6 character
ans = '3.14'
fprintf('%+6.2f',pi) % +3.14 - the same float, signed
ans = '+3.14'
fprintf('%0+6.2f',pi) % +03.14 - the empty characters are filled by zeros
ans = '+03.14'
fprintf('%0+12.8f',pi) % +03.14 - the empty characters are filled by zeros, 8 decimals
ans = '+03.1459265'
```

The last expression is a floating point number (f), that consist of 12 characters (including the + sign, and the decimal point!), the number of decimals is 8 (12.8), it is signed (+) and all the rest of the undefined characters are filled by zeros (0), to have 12 characters altogether (without 0, it would be empty space), and pi is represented in this form. If the result would be longer than 12 characters with the specified
amount of decimals, than the number of characters is ignored.

The same way, you can create the path and filename you wish to read or write:

```matlab
i = 1

i = 1

sprintf('Project1-IMG-%04d.JPG',i) % 'Project1-IMG-0001.JPG'

ans = 'Project1-IMG-0001.JPG'

for i=1:5
    filenev = sprintf('Project-IMG-%04d.jpg',i)
end

filenev = 'Project-IMG-0001.jpg'
filenev = 'Project-IMG-0002.jpg'
filenev = 'Project-IMG-0003.jpg'
filenev = 'Project-IMG-0004.jpg'
filenev = 'Project-IMG-0005.jpg'

Lets write a new function, saved separately, that transforms our result in degrees with decimals into the deg-minute-seconds form used in geodesy! The minutes and seconds should be presented in two-digit form e.g. 192-03-12.

```matlab
fpm(124.345)
```

```
ans = '124-20-42'
```

**Saving formatted results in file**

Lets write the results of our stress-strain (sigma-epsilon) diagrams with regular formatted values into a new file. The strain (deformation) should have one decimal, and the relevant stress should have two decimals. You will need the basic file managing functions i.e. opening, writing, closing in the following order:

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

You can determine with `fopen` what you wish to do with the file: `r` - just for reading (by default if you don’t define anything), `w` - for writing, `a` - to append:

```matlab
fileID = fopen(filename,'w') - opening a file for writing
```

You can close an open file by its ID:

```matlab
close(fileID, or all opened file together:
close(all). ‘\n\n’ the end of line character in Windows.
```

Writing the data into a file could be done line-by-line in a loop:

```matlab
n = length(xi); % length of a vector
fid = fopen('table_1.txt','w');
for i=1:n
    fprintf(fid,'%4.1f %6.2f\n',xi(i),yi(i));
end
close(fid);
```

Or all together in one step:

```matlab
fid = fopen('table_1.txt','w');
fprintf(fid,'%4.1f %6.2f\n',result);
close(fid);
type table_1.txt % present the content of the file in the command window
```

```
0.0 0.00
0.1 191.27
0.2 308.08
0.3 299.87
0.4 299.49
```
The variable 'result' contains all data in 2 columns and 271 lines, but to write the formatted results into the file the transpose matrix was used (2 rows, 271 columns), because fprintf function reads the the data values by columns!

Reading measurement data

As an engineer often you need to work with measurement data, which contains not only numbers, but additional characters, texts too. To work with this kind of data you need to read the raw file, and extract the relevant information you need. Let's take a look on a navigational example:

In this exersize you have a GPS recorded route, and its using a navigational standard format (NMEA 0183). Read in the file (hb_nmea.txt) and print out the route on a new figure!

Sample from the dataset:

```plaintext
$GPGLL,5156.9051,N,00117.1178,E*69
$GPGLL,5156.9194,N,00117.1482,E*61
$GPGLL,5157.0772,N,00117.4576,E*68
```

In the standard NMEA at the beginning of a line the $GPGLL means, its containing GPS Geographic Latitude, and Longitude data (there are multiple NMEA messages in general). It contains fixed length, comma separated fields, therefor this is as easily readable file. For the geographic latitude, the first two characters are the degree values, after that comes the minutes, in case of the longitude, the first three characters are the degree values, and then comes the minutes (because the former is in the range +90, the latter is in the range of +180). For the latitude values N means Nord, S means South, in case of the longitude E stands for East and W stands for West.

E.g: 5156.9051,N means nordic latitude 51°56.9051'.

This is a bit more complex, therefor the simple load function won't help you. The loading in this case will be similar to the writing, where at first you need to open the file (fopen), then you need to do the reading process, and after that finish it by closing the file (fclose). In this case there are some other useful functions too, that are coming handy to read the dataset line-by-line: fgets, fget. Function fgets reads a single line and truncates the ending line character, while function fget keeps it. The result of each function is a string type variable. To process the whole dataset in the file, you'll need a condition-driven loop (while), that reads the data till the end of the file sign (feof - end-of-file).

Let's read just the first line, and try to acquire the relevant data for the route. Remark: After fileopening, a pointer tracks the position, till which point did we read the file. If you want to check this pointer use the function ftell(fid).

```c
fid=fopen('hb_nmea.txt');
line=fget(fid) % reading a single line

line = '$GPGLL,5156.9051,N,00117.1178,E*69'
% $GPGLL,5156.9051,N,00117.1178,E*69
```

The result is a string type variable, that contains the first line of the file. Let's filter the relevant data, the geographical latitude (phi) and longitude (lambda) values! For this you should know that in each line, characters #8-9 are the degree values for phi, #10-16 are the minutes, #20-22 are the degree values for lambda, and #23-29 are the minutes values for lambda. A certain numbered element in a string could be acquired the same way as an indexed element in a vector:
phi_deg = line(8:9); phi_min = line(10:16);
lambda_deg = line(20:22); lambda_min = line(23:29);

Lets convert the values into degrees with decimals! For this you should first convert from string to numbers the phi and lambda values. Use the function str2num to manage this:

\[
\text{phi} = \text{str2num}(\text{phi\_deg})+\text{str2num}(\text{phi\_min})/60 \times 51.9484
\]

\[
\text{phi} = 51.9484
\]

\[
\text{lambda} = \text{str2num}(\text{lambda\_deg})+\text{str2num}(\text{lambda\_min})/60 \times 1.2853
\]

\[
\text{lambda} = 1.2853
\]

Now read in the N/D and E/W characters to determine where the coordinates are: character #18 stands for N (nord) or S (south) latitude, #31 stands for E (east) or W (west) longitude. Letter D and W gives a negative sign to the relevant coordinate. For this you will need an if statement, which is a conditional assignment that will change the signes if necessary. The structure for an if statement:

\[
\text{If condition}
\]

processes that should be run if the condition is true (it could be written in single/multi lines)

\[
\text{end};
\]

If its necessary you can add more branching points before the end operator, with the elseif (to define more conditions) or else statements.

\[
\text{NS = line(18); if NS=='S'; phi=phi*-1; end;}
\]

\[
\text{EW = line(31); if EW=='W'; lambda=lambda*-1; end;}
\]

This way you have managed to to acquire all the information from a single line, that's necessary to get a coordinate of the route, though it had a bit more complex structure. There are of course a lot of additional built-in method in MATLAB that can handle different inputs/outputs, if you are interested, just check the help guide for further details: help iofun

help iofun

File input and output.

File import/export functions.

matfile - Load or save parts of variables in MAT-files.

dlmread - Read ASCII delimited file.
dlmwrite - Write ASCII delimited file.
csvread - Read a comma separated value file.
csvwrite - Write a comma separated value file.
importdata - Load data from a file into MATLAB.
daqread - Read Data Acquisition Toolbox (.daq) data file.
matinfo - Text description of MAT-file contents.
fileread - Return contents of file as string vector.

Table functions.

readtable - Create a table by reading from a file.

writetable - Write a table to a file.

Spreadsheet support.

xlsread - Get data and text from a spreadsheet in an Excel workbook.
xlswrite - Stores numeric array or cell array in Excel workbook.
xlsinfo - Determine if file contains Microsoft Excel spreadsheet.

Internet resource.

urlread - Returns the contents of a URL as a string.

urlwrite - Save the contents of a URL to a file.

ftp - Create an FTP object.

sendmail - Send e-mail.

Zip file access.

zip - Compress files into zip file.

unzip - Extract contents of zip file.

Tar file access.

tar - Compress files into tar file.

untar - Extract contents of tar file.

Gzip file access.

gzip - Compress files into GNU zip files.
gunzip  - Uncompress GNU zip files.

Formatted file I/O.
fgetl  - Read line from file, discard newline character.
fgets  - Read line from file, keep newline character.
fprintf - Write formatted data to file.
scanf  - Read formatted data from file.
textscan - Read formatted data from text file.

File opening and closing.
 fopen - Open file.
 fclose - Close file.

Binary file I/O.
 fread - Read binary data from file.
 fwrite - Write binary data to file.

File positioning.
 feof - Test for end-of-file.
 ferror - Inquire file error status.
 frewind - Rewind file.
 fseek  - Set file position indicator.
 ftell - Get file position indicator.

Memory-mapped file support.
 memmapfile  - Construct memory-mapped file object.

File name handling.
 filepath  - Filename parts.
 filesel  - Directory separator for this platform.
 filemarker - Character that separates a file and a within-file function name.
 fullfile  - Build full filename from parts.
 matlabroot - Root directory of MATLAB installation.
 mexext  - MEX filename extension for this platform.
 partialpath - Partial pathnames.
 pathsep  - Path separator for this platform.
 prefdir  - Preference directory name.
 tempdir  - Get temporary directory.
 tempname  - Get temporary file.

XML file handling.
 xmlread - Parse an XML document and return a Document Object Model node.
 xmlwrite - Serialize an XML Document Object Model node.
 xslt - Transform an XML document using an XSLT engine.

Serial port support.
 serial - Construct serial port object.
 instrfindall - Find all serial port objects with specified property values.
 instrfind - Find serial port objects with specified property values.

Timer support.
 timer - Construct timer object.
 timerfindall - Find all timer objects with specified property values.
 timerfind - Find visible timer objects with specified property values.

Command window I/O.
 clc   - Clear command window.
 home - Send the cursor home.

SOAP support.
 createClassFromWsd1 - Create a MATLAB object based on a WSDL-file.
 callSoapService - Send a SOAP message off to an endpoint.
 createSoapMessage - Create the SOAP message, ready to send to the server.
 parseSoapResponse - Convert the response from a SOAP server into MATLAB types.

See also general, lang, audiovideo, imagesci, graphics, uitools.

Now to process the whole file, you just need to combine the previous materials with a conditional-driven cycle (while loop). The condition is to check whether the fprocess reached the end of the file. Therefor: feof(fd)==0 (feof = end of file). You'll also need two vector variable (lat, long), where you can store the acquired coordinates. You should initialize them right at the beginning as empty vectors, and you will simply append it in each cycle by the acquired values. If you are making larger calculations, its really useful to end each line with a semicolon (;) to avoid displaying each sub-result. The whole process:

```matlab
lat = []; long = [];
```
Present the route on a new figure, with thick red line:

```matlab
gure
plot(long, lat,'r','LineWidth',3)
```

You can't really locate the route yet. Let's load in somehow an additional map:

```matlab
part = load('partvonal.txt');
hold on; plot(part(:,1),part(:,2),'b')
```

Alternative:

```matlab
gure
worldmap([0 70],[-110 40])
load coastlines
plotm(coastlat,coastlon)
hold on
plotm(lat, long,'r','LineWidth',3)
```
function y = probafv(x)
    y = 2*x.^2 - 3*x + 1;
end

function y = probafv2(x,p)
    y = 2*x.^p - 3*x + 1;
end

function [x2, x3, x4] = hatvany(x)
    x2 = x.^2;
    x3 = x.^3;
    x4 = x.^4;
end

function str = fpm(x);
% The function changes from degree with decimals to degree-minute-seconds format
% The output is a formatted string (ddd-mm-ss)
    f = fix(x);
    p = fix((x-f) .* 60);
    m = round(((x-f).*60-p).*60);
    str = sprintf('%3d-%02d-%02d', f, p, m);
end