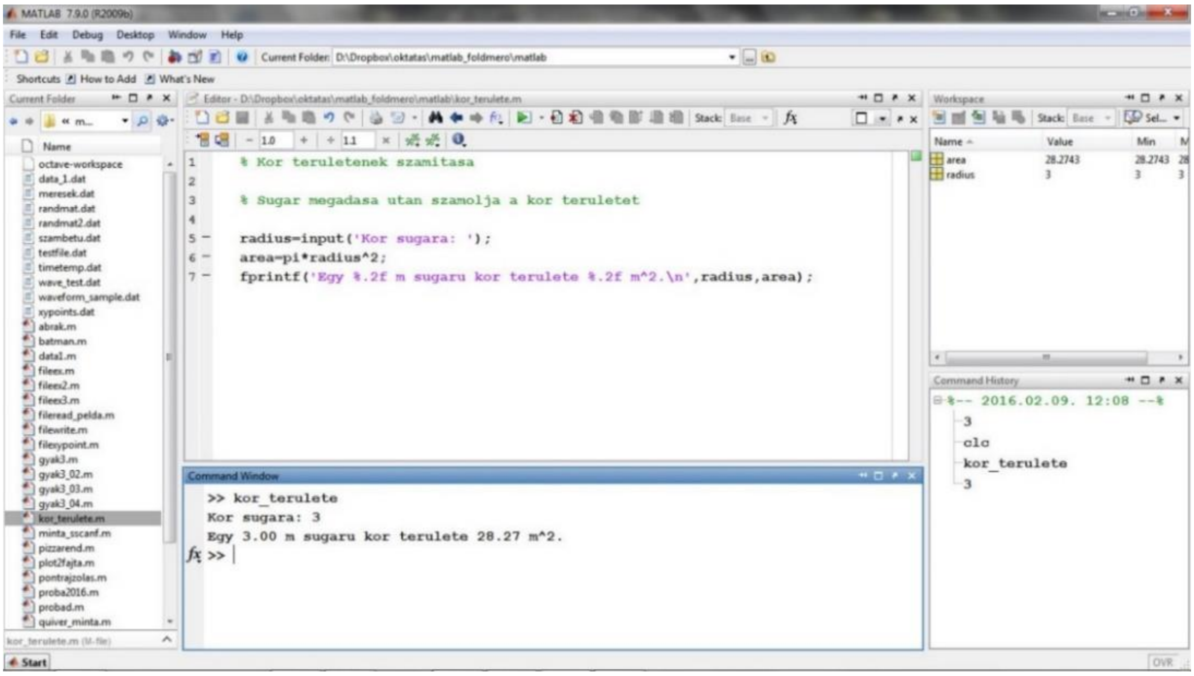
1. MATLAB/OCTAVE BASICS [1]

During the next practices, various engineering problems will be solved with numerical methods algorithms using the Matlab mathematical environment. 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 5.1.0, which came out on Mar 1, 2019.

The students can create a **MathWorks account** on the Matlab site using BME email address, using this account they can access Online Matlab also. With the MathWorks account, students can practice the basics of Matlab 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** site.



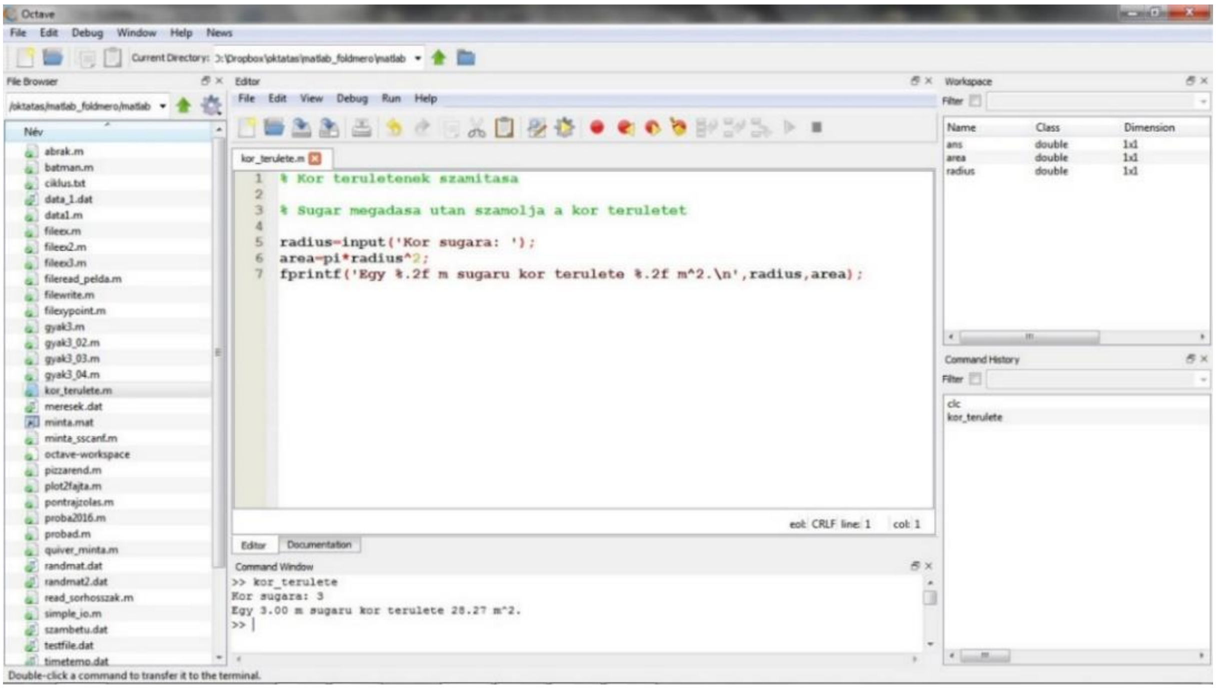
```
% Kor teruletenek szamitasa

% Sugar megadasa utan szamolja a kor teruletet

radius=input('Kor sugara: ');
area=pi*radius^2;
fprintf('Egy %.2f m sugaru kor terulete %.2f m^2.\n',radius,area);
```

```
>> kor_terulete
Kor sugara: 3
Egy 3.00 m sugaru kor terulete 28.27 m^2.
>> 
```

MATLAB 7.9.0 (R2009b)
File Edit Debug Desktop Window Help
Current Folder: D:\Dropbox\oktatas\matlab_foldmero\matlab
Editor - D:\Dropbox\oktatas\matlab_foldmero\matlab\kor_terulete.m
Workspace
Command Window
Command History

1MATLAB GRAPHICAL ENVIRONMENT

[1] Refernce 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


Octave
File Edit Debug Window Help News
Current Directory: D:\Dropbox\oktatas\matlab_foldmero\matlab
File Browser
Editor
File Edit View Debug Run Help
kor_terulete.m
Workspace
Filter
Command History
eol: CRLF line: 1 col: 1
Editor Documentation
Command Window
Double-click a command to transfer it to the terminal.

```
% Kor teruletenek szamitasa

% Sugar megadasa utan szamolja a kor teruletet

radius=input('Kor sugara: ');
area=pi*radius^2;
fprintf('Egy %.2f m sugaru kor terulete %.2f m^2.\n',radius,area);
```

```
>> kor_terulete
Kor sugara: 3
Egy 3.00 m sugaru kor terulete 28.27 m^2.
>> |
```

Name	Class	Dimension
ans	double	1x1
area	double	1x1
radius	double	1x1

```
clc
kor_terulete
```

1 OCTAVE GRAPHICAL ENVIRONMENT

STARTING MATLAB/OCTAVE

The most important parts of the graphical interface are the **current folder** where the current directory is stored, the **command window**, the **workspace** with the used variables, the **command history** with all the used commands and the **editor** (editor). Left-click and hold on the name of a panel to drag and drop them into the blue-colored area. You may want to dock the editor by clicking on the arrow at the top right of the Editor window.

Keep in mind that file names used in Matlab should not begin with a number or contain special characters or spaces! In the program, only those functions and files can be used or run which are in the current directory.

HELP, DOCUMENTATION

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

```
> help elfun
```

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

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

This gives you a description of the command and how to use it. E.g. try:

```
> help rand
```

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

The **doc** command works much like 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. This can be used to search for a word in the name of the command or in its description. Let's try the following:

```
> lookfor rand
```

This will list all the commands whose names or their short description contains the word **rand**. 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 to the command line e.g. **pref**, then press ’**Tab**’. If there is only one command with **pref** beginning, 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 buttons** 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
```

Basic formatting:

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

The basic variable type in Matlab is the matrix. Vectors are special matrices, 1xn row vectors or mx1 column vectors. Square brackets are used in Matlab to define a matrix or vector. If you work with matrices/vectors, be careful with the different parenthesis, because each type is doing different operations: ( ), [ ], { } ! The elements are separated by a comma or space within the row, and the rows are separated by semicolon.

```
>  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
```

When displaying a vector, it can be problematic to have 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** – 5 significant digits, **longG** 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 putting the element 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, line 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 right away. The disadvantage of this format is that this is Matlab's own format, which can only be opened within Matlab.

You can run the script files by typing their names or just pressing **F5**, this will save and run the program right away. 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)
```

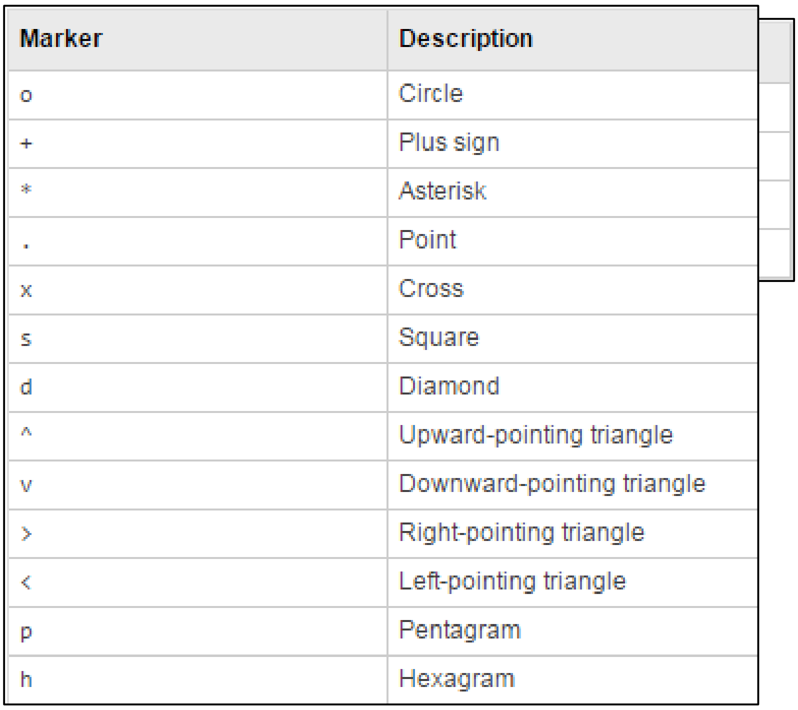
This will connect the points with a line. If you want to mark points with symbols, try the following:

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

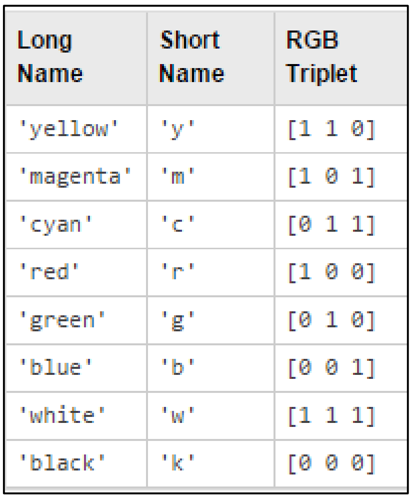
You can add many arguments to define the specifics of a plot, like the shape and size of the markers, specifics of the lines, etc.

```
> 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 % értéke: 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 have seen some earlier. It is important that we could interpret these to correct our mistakes!

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.
```

An easy typing error would be to type 8 or 9 instead of round parentheses:

```
   >  sin(pi9
sin(pi9
        ↑
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 missed:

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

Worst of all, when there is no error message, but the result is still incorrect. Example: Calculate $\frac{1}{2\pi}$ with the following statement. Why is the result wrong?

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

ADDITIONAL ADVICE FOR USING OCTAVE

If you decide to use Octave to practice at home, **Octave** can be downloaded from https://www.gnu.org/software/octave/, the current version is 5.1.0, which came out on Mar 1, 2019. The advantage of Octave over education licensed Matlab is that it is an open source program that can be used not only for educational purposes but also for work. There are many add-on packages available, see https://octave.sourceforge.io/ and https://octave.sourceforge.io/packages.php, many of them are installed, just need to be loaded when you want to use them. You can query what is installed with the **pkg list** command).

INSTALLING THE SYMBOLIC PACKAGE

During the numerical methods practices we will perform many symbolic computations too, which requires installing an extra symbolic package for Octave (this is a separate toolbox in matlab). This package is based on python-sympy and must be installed separately. Installation under Windows (link: https://github.com/cbm755/octsympy/wiki/Notes-on-Windows-installation)

1. Download the symbolic-win-py-bundle-x.y.z.zip file at the github releases page. https://github.com/cbm755/octsympy/releases (here x.y.z is the version number, e.g.symbolic-win-py-bundle-2.8.0.tar.gz)
2. Start Octave, change folder to where your downloads are.
3. Install the package by typing into Octave:

```
   >  pkg install symbolic-win-py-bundle-x.y.z.tar.gz
```

4. Load symbolic package to octave

```
   >  pkg load symbolic
```

5. Check the package by typing a few symbolic commands!

```
   >  syms x
   >  f = sin(cos(x));
```

```
> diff (f)
```

Result ⇒ `-sin(x)·cos(cos(x))`

Functions of the symbolic package in detail:
https://octave.sourceforge.io/symbolic/overview.html

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

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!

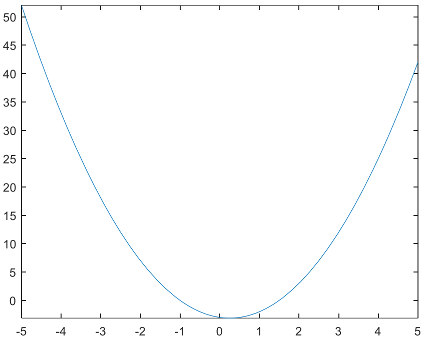
$$2\,x^2 - x - 3 = 0$$

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

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

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
```

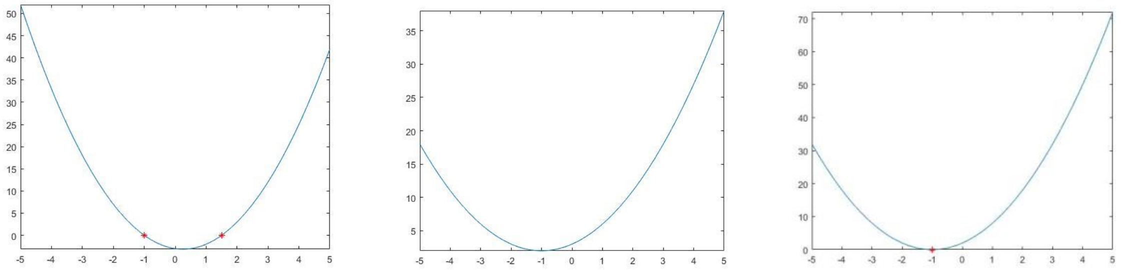
The **disp** command will print a text to the command window.

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?

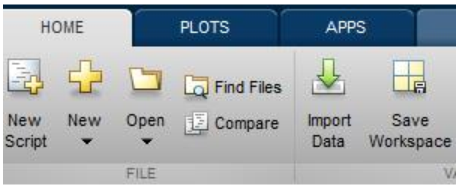
DATA IMPORT/EXPORT

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.

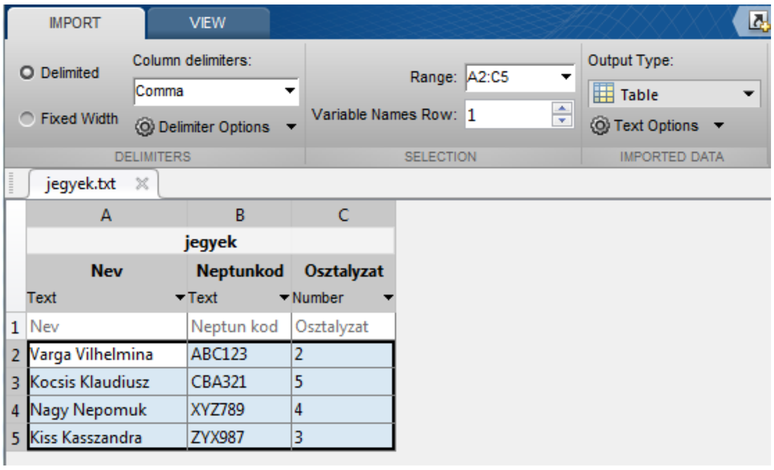
Often, after a complex mathematical calculation, we have to present our result in another specific format for further use. Let's look at some examples of import and export commands to get a little acquainted with file operations.

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.



● Delimited
○ Fixed Width

Column delimiters: Comma

Range: A2:C5

Variable Names Row: 1

Output Type: Table

	A	B	C
	jegyek		
	Nev	Neptunkod	Osztalyzat
	Text	Text	Number
1	Nev	Neptun kod	Osztalyzat
2	Varga Vilhelmina	ABC123	2
3	Kocsis Klaudiusz	CBA321	5
4	Nagy Nepomuk	XYZ789	4
5	Kiss Kasszandra	ZYX987	3

```
>  %% '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.

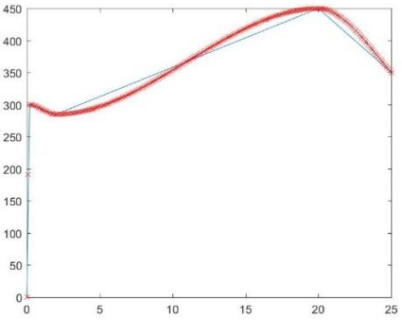
```
>  x = adat(:,1); % first column - strain
>  y = adat(:,2); % second column - stress
>  plot(x,y);
>  xlabel('\epsilon');ylabel('\sigma');
```

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;
```

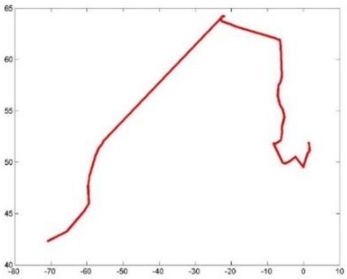
This way, for example, it is possible to extract the relevant information from a more complex structure. There are of course a lot of additional built-in functions in MATLAB to handle different inputs/outputs, if you are interested in, just check the help guide for further details using the **help iofun** 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:

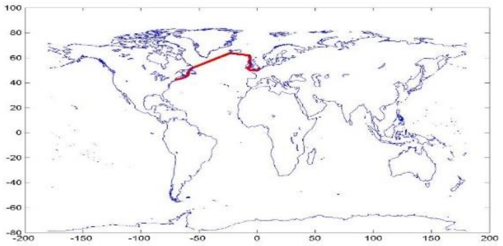
```
> LAT = []; LON = [];
> fid=fopen('hb_nmea.txt');
> while feof(fid)==0
>   line=fgetl(fid); % read a line
>   % aquire latitude, longitude
>   lat_deg = line(8:9);   lat_min = line(10:16);
>   lon_deg = line(20:22);  lon_min = line(23:29);
>   % convert to decimal degree
>   lat = str2num(lat_deg)+str2num(lat_min)/60;
>   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.

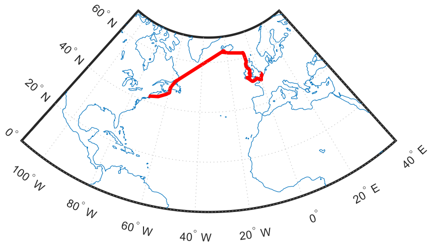


```
> coast = load('coastline.txt');
> hold on;
  plot(coast(:,1),coast(:,2),'b')
```

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

Computational errors

Exercise: Is 0.3 = 0.3 according to MATLAB?

A = 0.3

B = 0.1 + 0.1 + 0.1

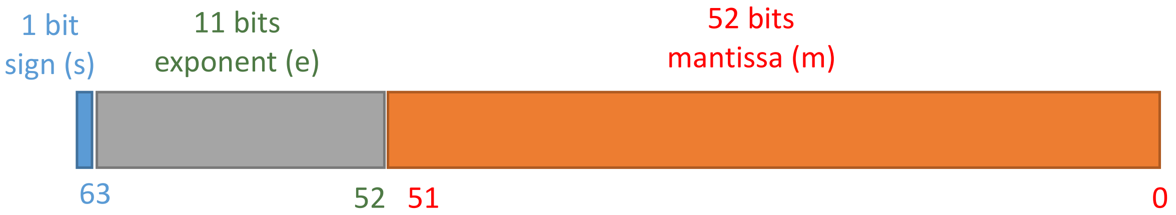
Is A equal to B?

A == B → 0 (false)

How are numbers represented in the computer?

- The default and most common form of storing numbers is the floating point format (double precision floating point).
- Standardized format by the IEEE (IEEE 754).
- Each number is stored in binary format using 64 bits (0-63).

Representation of a number: $(-1)^s \cdot m \cdot 2^{(e-1023)}$



Conversion from decimal to double precision

Let's convert 1055.125 to double precision!

1. Convert to binary: 10000011111.001
2. Normalize: 1.00000111110011 x 2^{10} ads
3. Calculate the exponent: 10 + 1023 = 1033 (10000001001)
4. Sign: 0 (0 – positive, 1 - negative), Mantissa: 111110011

1055.125 = 0 10000001001 00000111110011000...000

So what is the problem with 0.3?

0.3 in binary → 0.010011001100… (infinite)

0.1 in binary → 0.000110001100011… (infinite)

Adding up an infinite fraction three times results in a round-off error, thus the two values will not be exactly equal with the precision used by the representation.

The machine epsilon

- In MATLAB, the variable *eps* and the function *eps()* can be used to query the value of the machine epsilon.

- *eps* by default is $\approx 2 \cdot 10^{-16}$

a = 2

b = $2 + 2 \cdot 10^{-17}$ (which is less than the machine epsilon)

a == b → 1 (True)

eps(1e6) $\approx 1 \cdot 10^{-10}$

eps(1e10) $\approx 2 \cdot 10^{-6}$

As we go higher up the number line, the separation between two distinct numbers gets bigger.

- In other words, you can't add a very small number (smaller than the machine epsilon) to a very big number because in our limited floating-point representation the two numbers will be the same!
- Because of the machine epsilon, we can have a cancellation error as well, when we are subtracting almost equal numbers:

x1 = 4e-15; y1 = 4e-14;

x = 10+x1(14 zeroes before digit 4, just above the machine epsilon)

y = 10+y1 (13 zeroes before digit 4, above the machine epsilon)

(y – 10.0)/(x – 10.0) = 11.5 (not the expected result, which would be 10)

realmin, *realmax* – the biggest and smallest representable numbers

Truncation error

- Happens when we are using numerical approximation of mathematical expressions instead of their exact value.
- E. g. using the Taylor-series representation (to some degree) of a function:

Built-in MATLAB function for the function e^x at x = 1:

f = *exp*(1*)*

Approximation using the Taylor-series with four terms:

g = $1 + x + x^2 + \frac{x^2}{2!} + \frac{x^2}{3!}$

The total error = round-off error + truncation error

Absolute and relative error

- In most cases, the floating-point representation of a decimal number will differ from the exact value of the number due to the rounding and truncation errors.
- Absolute error: $\Delta = |x - \tilde{x}|$
  - x is the exact value and $\tilde{x}$ is the approximation

- Relative error: $\varepsilon = \frac{|x-\tilde{x}|}{|x|}$
    - The relative error is the absolute error with respect to the exact value of the number. It better describes the scale and impact of the error.

Stability and condition number

- Due to the system used to represent numbers, our calculations will most likely contain numerical errors. How can we rely on the results then?
- Condition number (of a mathematical problem): good condition means that a small change to the input of the calculation will only cause a small change in the output or result.
- Stable (algorithm): an algorithm is called numerically stable, if a small change to the input will only cause a small change in the output or result.
- In the opposite case, we talk about ill-conditioned/weakly conditioned problems and unstable algorithms.

Instable algorithm example

Quadratic equation: x^2−100.0001+0.01=0

The exact solutions are: x_1=100 and x_2=0.0001

MATLAB solution:

```
format long;
a = 1; b = -100.0001; c = 0.01;
D = sqrt(b^2 - 4*a*c) % 99.9998999999999997
xl = (-b + D)/(2*a) % 100
x2 = (-b - D)/(2*a) % 1.000000000033197e-04
```

The value of the second solution is not exact. The reason for this is that in that case, two very close numbers were subtracted resulting in a cancellation error.

In many of these cases, we can change the algorithm, to achieve a stable solution.

For example, changing the formula for the second solution the following way:

$$x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a} \cdot \frac{-b+\sqrt{b^2-4ac}}{-b+\sqrt{b^2-4ac}} = \frac{2c}{-b+\sqrt{b^2-4ac}}$$

```
x2m = (2*c)/(-b+D) % 1.000000000000000e-04
```

Stable algorithm example

Let's calculate the value of e^{-x} using two different approaches:

1. $f(x) = e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots$
2. $g(x) = e^{-x} = \frac{1}{e^x} = \frac{1}{1+x+\frac{x^2}{2!}+\frac{x^3}{3!}+\cdots}$

Calculate the value of $e^{-8.3}$ using the built-in function and our custom one:

```
format short
% Built-in function
exp(-8.3) % = 2.4852e-04

% Custom function with different number of terms
[f, g] = emx(8.3, 10) % 10 terms, = 188.0344 & 3.1657e-04
[f, g] = emx(8.3, 20) % 20 terms, = 0.2833 & 2.4856e-04
[f, g] = emx(8.3, 30) % 30 terms, = 2.5151e-04 & 2.4852e-04
```

The speed of convergence is different, but both algorithms give the right answer after a certain number of steps.

Condition number of a problem

Consider the following linear system of equations:

$$6x_1 - 2x_2 = 1011.5x_1 - 3.85x_2 = 17$$

In matrix form ($A \cdot x = b$):

$$\underbrace{\begin{bmatrix} 6 & -2 \\ 11.5 & -3.85 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} 10 \\ 17 \end{bmatrix}}_{b}$$

Solve the system in MATLAB:

```
A = [6, -2; 11.5, -3.85]; b = [10; 17];
% Solution
sol = inv(A)*b % or A\b
sol =
    45.0000
   130.0000
```

Adjust the coefficient for x_2 by a very small amount (-3.85 → -3.84) and see the results again:

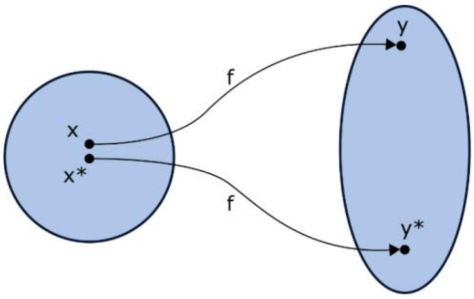
```
A = [6, -2; 11.5, -3.84]; b = [10; 17];
% Solution
sol = inv(A)*b % or A\b
sol =
   110.0000
   325.0000f
```

There is significant change in the results, even though the input was only slightly changed!

- The transformations given by some matrices are very sensitive to the change of input.
- This can be measured by the condition number of the matrix:

The condition number:

$$\kappa = \left| \frac{\frac{f(x) - f(\tilde{x})}{f(\tilde{x})}}{\frac{x - \tilde{x}}{\tilde{x}}} \right| = \left| \frac{\tilde{x}}{f(\tilde{x})} \cdot \frac{f(x) - f(\tilde{x})}{x - \tilde{x}} \right| = \left| \frac{\tilde{x} \cdot f'(\tilde{x})}{f(\tilde{x})} \right|$$



Using MATLAB to get the condition number of a matrix:

```
A = [6, -2; 11.5, -3.84]; b = [10; 17];
% Condition number of matrix A
cond(A); % = 4.6749e+03
```

The grater the condition number is, the bigger the impact of a small change in the inputs has on the results. This effect has to be taken into account in case of engineering problems where the input data are measurements which can often be approximations or values containing errors.

Finding the roots of non-linear equations

(Translation, modification: Bence Ambrus, original hungarian material: Piroska Laky)

Non-linear equations are equations in the form of

$$f(x) = 0$$

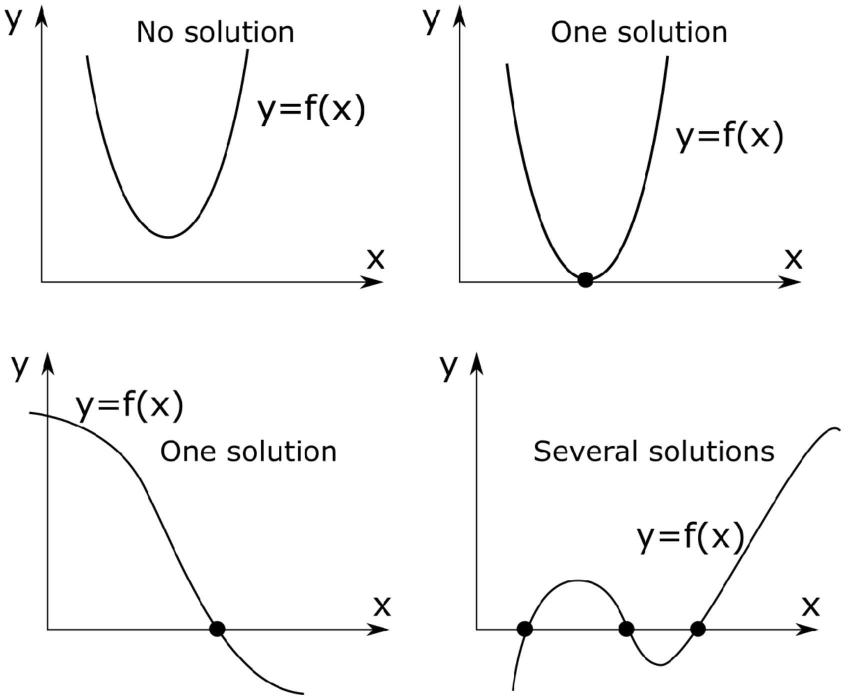
At first, we may have a problem different than $f(x) = 0$, however, we can always convert it to the form above by subtracting the right-hand side from both sides. For example, given the following non-linear equations:

$$e^{x^{2-3}} = 12x$$
$$\sin(x)^2 = 36$$

we can achive the form above by converting them:

$$e^{x^{2-3}} - 12x = 0$$
$$\sin(x)^2 = 36$$

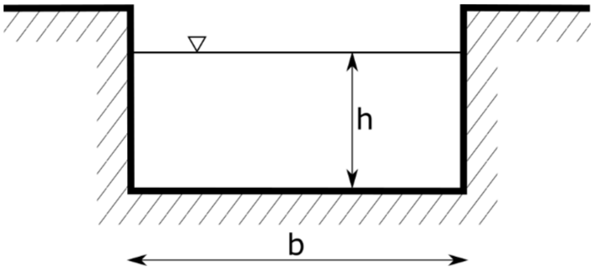
The solutions of equations are also called the roots of the equation. If we substitute the root back into the equation, we get zero, or at least a value close to zero (remember the rounding error from the last practical) within some tolerance level or threshold. If we think about it graphically, the root is where the graph of the equation intersects the x-axis. Depending on the form of the equation, we can have four different possibilities for the roots:



In many practical cases, the root or roots can only be aquired numerically as opposed to the case when there is a symbolic solution (e.g. in the case of the quadratic equation). These methods use iterations to get closer and closer to a solution, until a certain tolerance is met. In other words, instead of solving $f(x) = 0$, we solve $f(x) = \Delta$, where Δ is a usually small value called the tolerance. There are two main types of algorithms for solving non-linear equations: the closed interval methods and the open interval methods.

Example for a non-linear problem

We will use the following hydraulical problem as an example for the major part of this live script:



Depending on the shape, material and slope of a channel and given the height of the water in the channel, the flow of water can be derived using the following formula:

$$Q = \frac{\sqrt{S}}{n} \cdot \frac{(b \cdot h)^{\frac{5}{3}}}{(b + 2 \cdot h)^{\frac{2}{3}}}$$

where Q is the flow of water in (m^3/s), S is the slope of the channel, n is the Gaucker-Manning coefficient (depending on the material), b is the width of the channel in m and h is the height of the water in m. Let's answer the following two questions:

1. How much is the flow of water if the height of the water in the channel is 2 m?
2. How much is the height of the water, if the flow of water is 3 m^3/s?

The following data is given about the channel:

- The slope is 0.08% ($S = 0.0008$)
- The Gaucker-Manning coefficient is 0.02 ($n = 0.02$)
- The width of the channel is 2 m ($b = 2$)

The first question is a simple substitution into the equation. First we have to define the variables in MATLAB, but before we do that, it is always useful to clear out all variables that might exist already and close all figures that might be open:

```
clear all; close all;
S = 0.0008;
n = 0.02;
b = 2;
```

We can define the equation with an anonymous or inline function with h as the variable and using the constants that we have already created.

```
Q = @(h) sqrt(S)/n * (b * h).^(5/3) ./ (b + 2*h).^(2/3);
```

Next, we can substitute 2 into h to get the answer to the first question.

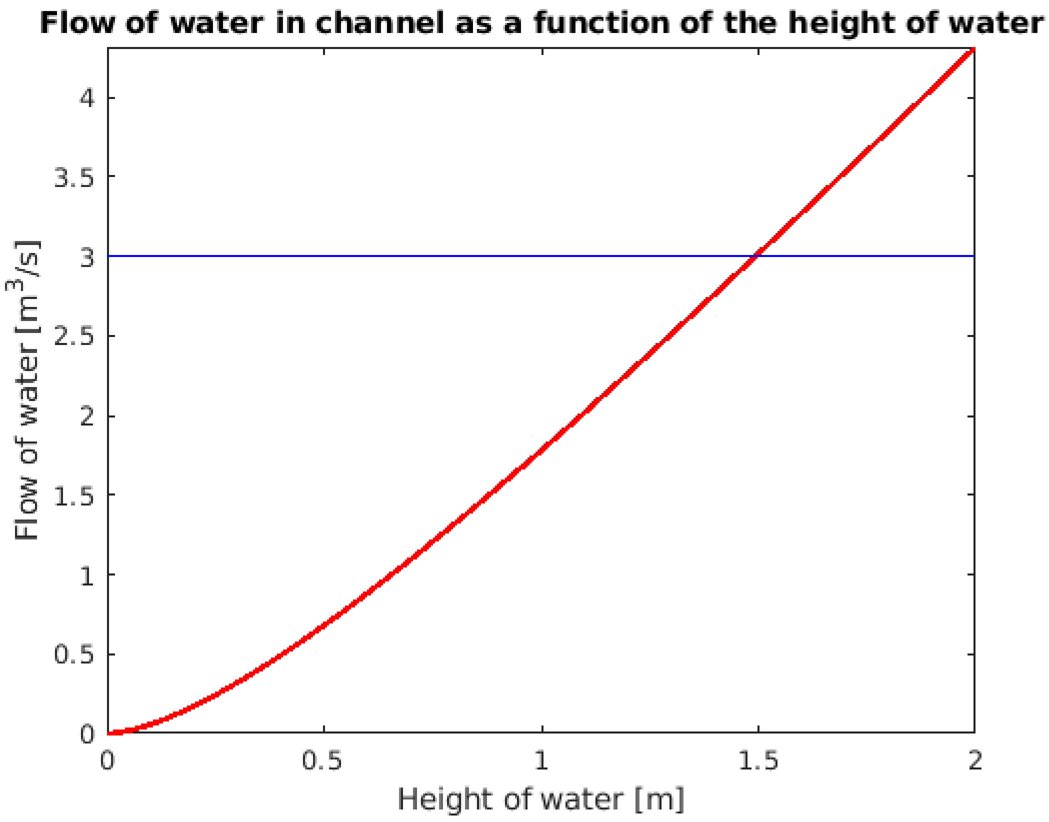
```
Q(2)
```

```
ans = 4.3170
```

The answer is 4.317 m^3/s.

In order to answer the second question, we would have to somehow transform the equation and solved for h. However, it is not possible using equivalent transformations, in other words, the equation cannot be solved for h. The answer can only be given numerically. Before we take a look at the solutions, let's graph the equation between 0 and 2:

```
figure(1);
fplot(Q, [0, 2], 'Color', 'r', 'LineWidth', 2); % red color, 2 point line width
line([0, 2], [3, 3], 'Color', 'b'); % plot of a simple line with endpoint coordinates (0, 3) a
title('Flow of water in channel as a function of the height of water'); % title for the plot
xlabel('Height of water [m]');   % label for the x-axis
ylabel('Flow of water [m^3/s]'); % label for the y-axis
```



We can already see, that the solution will be somewhere between 1.4 and 1.6. To use a numerical algorithm to find the place where the red graph intersects the blue line, we have to transform the equation into the following form:

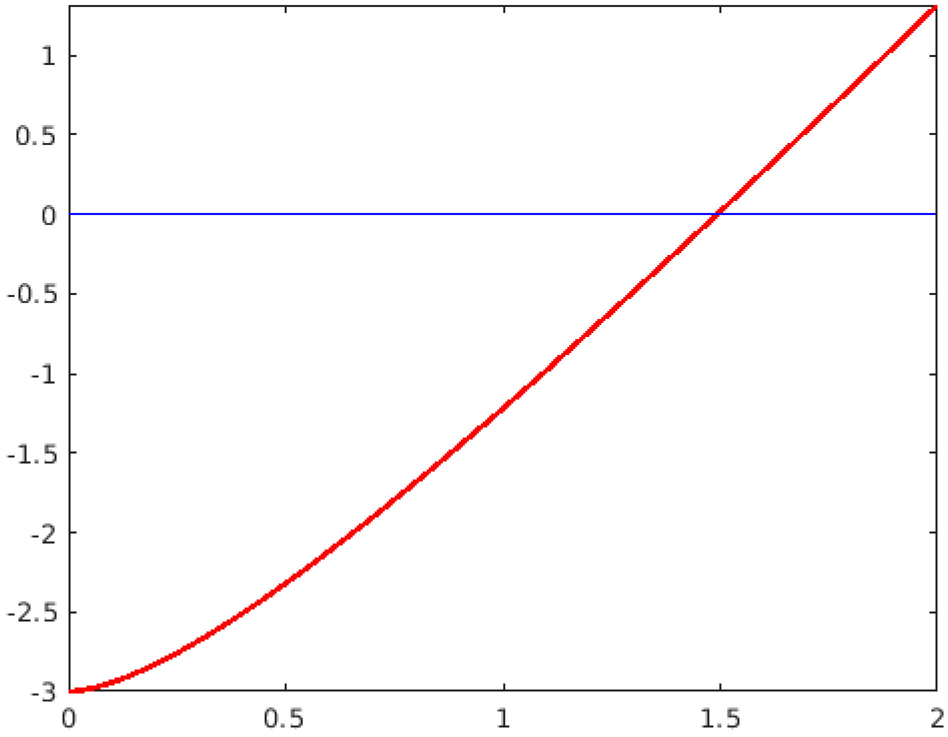
$$Q(h) = 0$$

As we want to the find the h value where Q is equal to 3, right now we have the following equation: $Q(h) = 3$. In order to transform this into the correct form, we simply have to subtract 3 from both sides:

$$f(h) = Q(h) - 3 = 0$$

We define a new anonymous function that gives us the above form and graph it:

```
f = @(h) Q(h) - 3;

figure(2);
fplot(f, [0, 2], 'Color', 'r', 'LineWidth', 2);
line([0, 2], [0, 0], 'Color', 'b');
```



Finding the solution for the problem or the root of the equation can be done using several algorithms, but all of them can be put into two categories: open interval methods and closed interval methods.

Closed interval methods

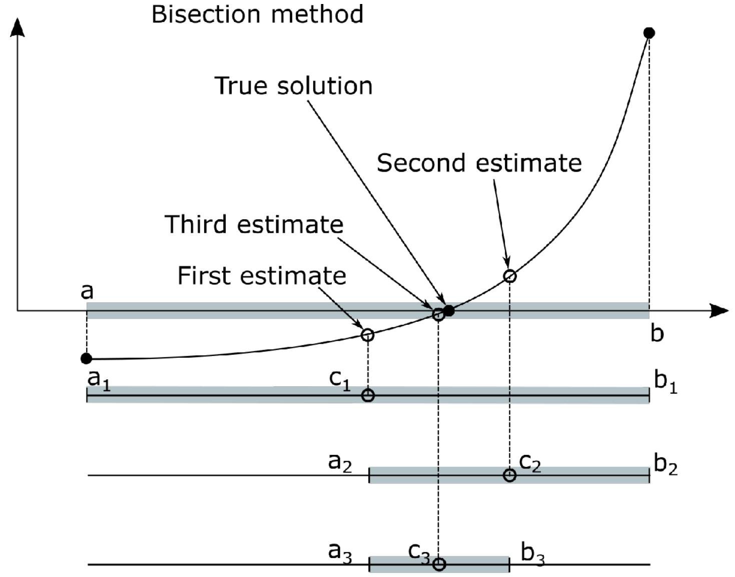
When talking about closed interval methods, we give an [a, b] interval which contains the solution. (The beginning and the end of the interval can come from a guess just like the one we did with the graph.) If the root is inside this interval, the signs of the function values in the beginning and the end of the interval differ, that is: $f(a) \cdot f(b) < 0$ as the function must intersect the 'zero-line' (that is, the x-axis) while going from a to b. If f is continous in the interval, than there is surely a point c which is the root of f. We steadily shrink the interval and check the function values at the sides of the interval until the function values get closer and closer to zero and reach zero within a certain tolerance, or if the interval becomes smaller than a certain tolerance. There are multiple closed interval algorihtms, the difference between them is how they shrink the interval.

There are some restrictions for the initial interval used to start the algorithm:

1. There should be at least one (and preferably one) solution in the interval.
2. The function has to defined at the ends of the interval.
3. The sign of the function value must differ at the ends of the interval.

Bisection method

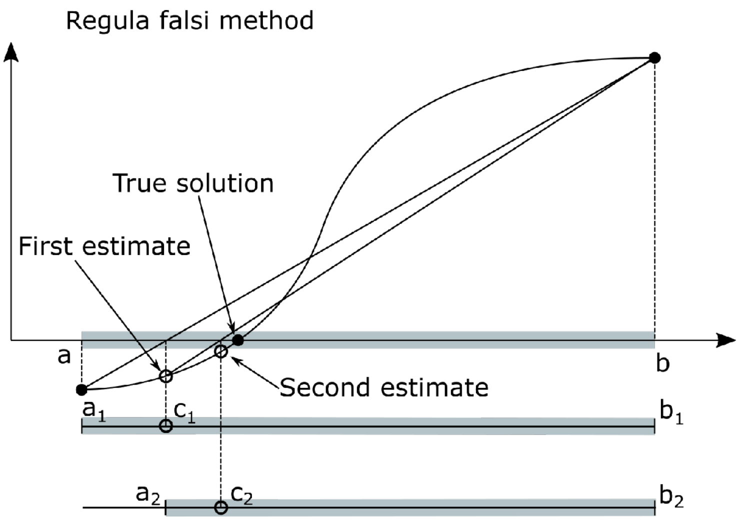
In the case of the bisection method, we halve the initial interval (point c) and see whether the function value in the half point is negative or positive. If it's negative, we change the previous negative end of the interval to the new point, if it is positive, we do the same to the positive side. In other words, we carry out the following steps:



1. Calculate point c by $c = \frac{(a+b)}{2}$.
2. Check if the function value at c is less than the tolerance. If $f(c) < \Delta$, the algorithm ends.
3. Check if $f(a) \cdot f(c) < 0$, if so, then the new b is changed to c, if not, then the new a is changed to c.

Regula falsi method

This method is more efficient then the bisection method, meaning that it converges to the solution faster. In this algorithm, point c is calculated by checking where the line between a and b intersect the x-axis.



$$\frac{b-a}{f(b)-f(a)} = \frac{c-a}{0-f(a)}$$

Point c can be calculated by taking the slope of the line between a and b, multiplying it by the value of the length between c and a and adding it to the function value at point a. This expression has to equal to zero.

$$\frac{f(b)-f(a)}{b-a} \cdot (c-a) + f(a) = 0$$

If we solve this expression for c:

$$c = \frac{a \cdot f(a) - b \cdot f(a)}{f(b) - f(a)}$$

The steps of the algorithm are exactly the same as with the bisection formula, only the way c is calculated changes.

The bisection and the regula falsi methods are not implemented in MATLAB, however, we can write our own custom function to find the root of a non-linear equation using these methods. Check the *bisection.m* and *regulafalsi.m* files for possible implementations.

Solving the channel problem using closed interval method

Solution using the custom bisection function. The first output of the function is the solution, the second is the number of iterations:

```
[xb, ib] = bisection(f, 1.4, 1.6, 1e-9, 100) % arguments: function to solved, start of interval
```

```
xb = 1.4929
ib = 28
```

Solution using the regula falsi method:

```
[xr, ir] = regulafalsi(f, 1.4, 1.6, 1e-9, 100)
```

```
xr = 1.4929
ir = 4
```

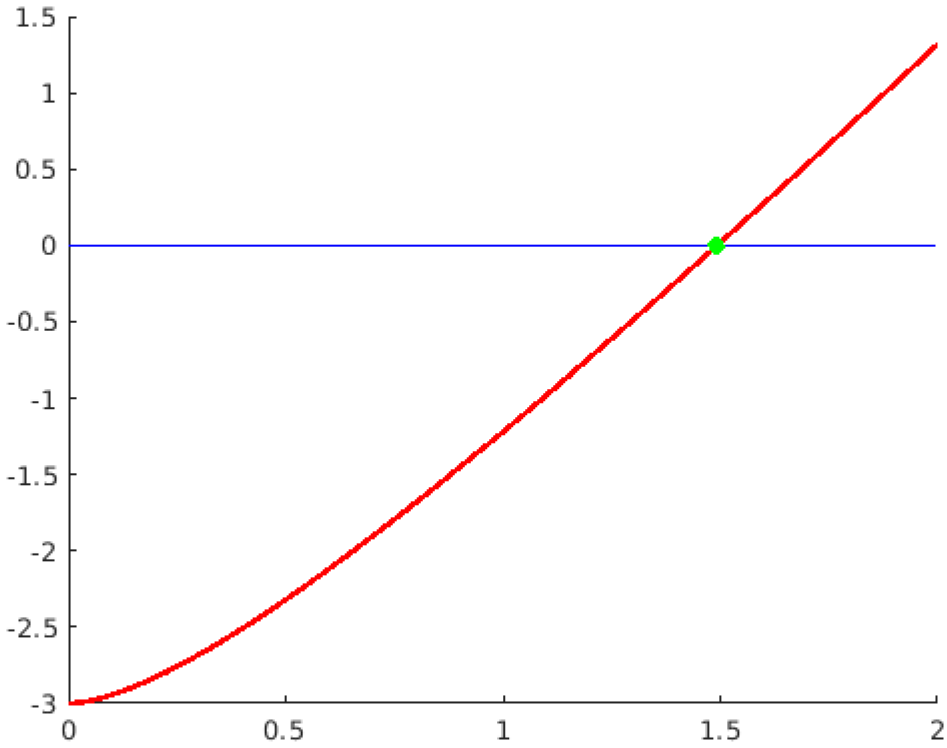
We can see from the results, that the regula falsi method converged much faster than the bisection method, and its solution is much closer to zero. Let's check the results by substituting them into the equation:

```
f(xb), f(xr)
```

```
ans = -4.5629e-10
ans = -1.3299e-10
```

We can also plot the solution on our second figure

```
figure(3);
hold on;
g3 = fplot(f, [0, 2], 'Color', 'r', 'LineWidth', 2);
g2 = line([0, 2], [0, 0], 'Color', 'b');
sol = plot(xr, f(xr), 'go', 'MarkerFaceColor', 'g');
```



Open interval methods

Newton-Raphson method

The Newton-Raphson method (or sometimes only called Newton's method) can be used in the case when the function is continous and differentiable and we have an initial value (x_0) for the solution which is suitably close to final solution. The steps of the first iteration of the algorithm are the following:

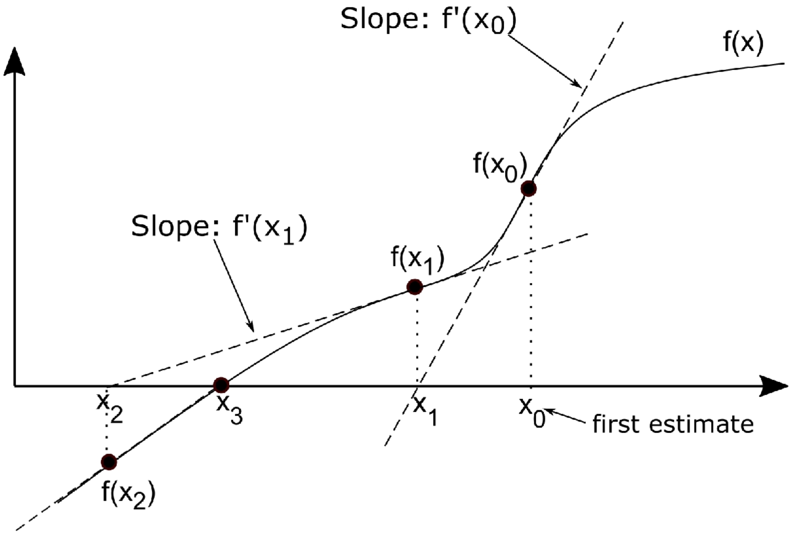
1. We compute the derivative of $f(x)$ at the initial value x_i, where i is zero at first. This gives us the slope of the line that is tangent to the graph of the function at x_i.

2. Using this line, we calculate a new x_{i+1} value, which is the x-coordinate of the intersection of the line in step 1 and the x-axis. The formula for this:

$$f'(x_i) \cdot (x_i - x_{i+1}) = f(x_i) - 0$$

From this expression, we calculate the next x value, which is x_1 in this case:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$



Secant method

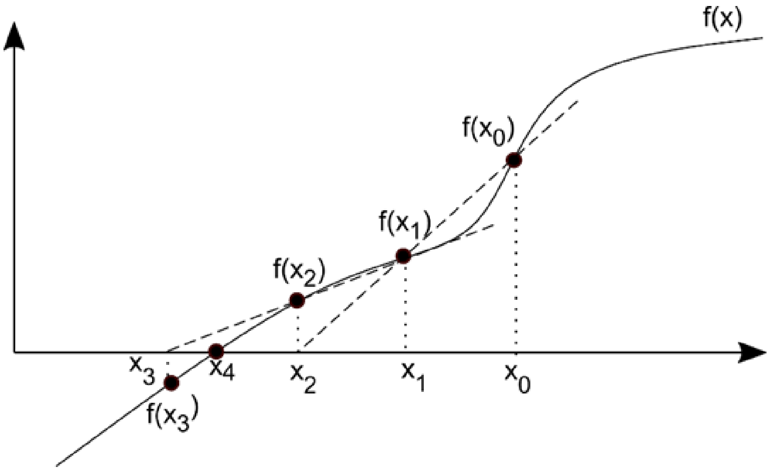
The secant method is a simplified form of the Newton-Raphson method, where we approximate the derivative of $f(x)$ (the slopes of the tangents to the function) using secant lines:

$$f'(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

This means that the formula for calculating the new x_{i+1} becomes:

$$x_{i+1} = x_i - f(x_i) \cdot \frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}$$

This method is viable when we do not know the derivative of $f(x)$ or computing it would cost too many computational resources. The downside is that it usually converges slower and it needs to initial guesses in order to start.



Custom Newton-Raphson algorithm in MATLAB

If we open the *newton.m* file, we can see a custom implementation of the Newton-Raphson algorithm.

Solving the channel problem using the custom Newton-Raphson algorithm

The algorithm requires the derivative of the function whose roots we want to find. Using the symbolic capabilities of MATLAB, we don't need to calculate the derivative by hand, rather we can use symbolic differentiation.

First, we convert the anonymous function $f(h)$ into a symbolic function:

```
sf = sym(f)
```

```
sf =
```

$$\frac{\sqrt{2}\ (2\,h)^{5/3}}{(2\,h+2)^{2/3}}-3$$

Next, we differentiate the symbolic function:

```
sdf = diff(sf, 'h') % h is the variable of the function
```

```
sdf =
```

$$\frac{10\ \sqrt{2}\ \ (2\,h)^{2/3}}{3\ (2\,h+2)^{2/3}}-\frac{4\ \sqrt{2}\ \ (2\,h)^{5/3}}{3\ (2\,h+2)^{5/3}}$$

In order to be able to use this in the custom algorithm, we have to convert it back into a anonymous MATLAB function.

```
df = matlabFunction(sdf)
```

```
df = function_handle with value:
    @(h)sqrt(2.0).*1.0./(h.*2.0+2.0).^(2.0./3.0).*(h.*2.0).^(2.0./3.0).*(1.0e1./3.0)-sqrt(2.0).*1.0./(h.*2.0+2.0).^
```

Now, we have all the input for the Newton-Raphson algorithm:

```
[xnew, inew] = newton(f, df, 1.6, 1e-9, 100)
```

```
xnew = 1.4929
inew = 3
```

The result is of course the same, but the number of iterations shows the efficiency of the method. The downside is that we had to find the derivative of the function which can be much more complicated in certain cases.

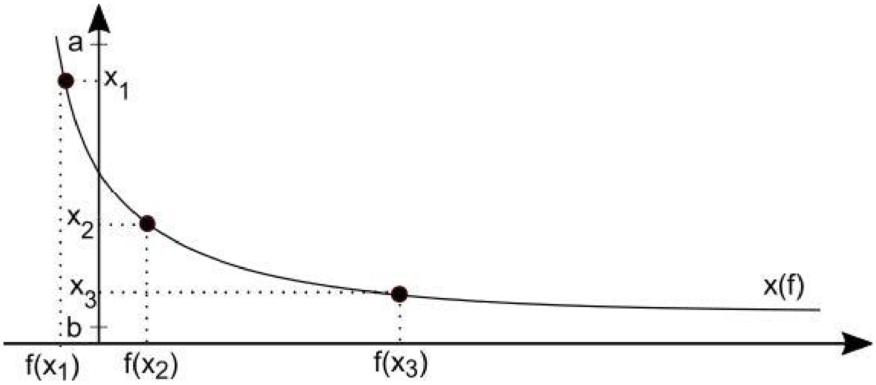
Solution using built-in MATLAB function *fzero*

Solving non-linear equations is an everyday task for a mathematical programming environment such as MATLAB, so there are built-in solutions for such a task. The built-in ***fzero*** function uses a combination of methods that contains the so-called Brent-Dekker method (the original Brent mehod was improved by Dekker), which is also called the inverse quadratic interpolation.

The algorithm kicks off by having 3 initial guesses in an [a, b] interval that contains the root. The coordinates of the points are given in an inversed order: $(f(x_i), x_i)$ and therefore, the function value becomes the unknown variable. We fit a quadratic polynomial onto these three points:

$$x(f) = \alpha_2 \cdot f^2 + \alpha_1 \cdot f + \alpha_0$$

which means that we calculate the coefficients of the polynom. Then, we substitute 0 into the polynomial (as we are looking for the x-value where f becomes 0), which gives us the c point that we can use to update the search interval and start again.



Solving the channel problem using the built-in fzero function

We can call the function using two initial values or just one.

```
x = fzero(f, [1.4, 1.6])
```

```
x = 1.4929
```

```
x = fzero(f, 1.6)
```

```
x = 1.4929
```

If we would like more information about the steps taken to find the solution or wish to control the method by specifying a tolerance value, we can do so by using the *optimset* function. First, we create some control options using the function.

```
opt = optimset('Display', 'iter', 'TolFun', 1e-9) % show iterations and set the tolerance to 1e
```

```
opt = struct with fields:
                   Display: 'iter'
               MaxFunEvals: []
                   MaxIter: []
                    TolFun: 1.0000e-09
                      TolX: []
               FunValCheck: []
                 OutputFcn: []
                  PlotFcns: []
           ActiveConstrTol: []
                 Algorithm: []
    AlwaysHonorConstraints: []
           DerivativeCheck: []
               Diagnostics: []
             DiffMaxChange: []
             DiffMinChange: []
            FinDiffRelStep: []
               FinDiffType: []
         GoalsExactAchieve: []
                GradConstr: []
                   GradObj: []
                   HessFcn: []
                   Hessian: []
                  HessMult: []
               HessPattern: []
                HessUpdate: []
          InitBarrierParam: []
     InitTrustRegionRadius: []
                  Jacobian: []
                 JacobMult: []
              JacobPattern: []
                LargeScale: []
                  MaxNodes: []
                MaxPCGIter: []
             MaxProjCGIter: []
                MaxSQPIter: []
                   MaxTime: []
             MeritFunction: []
                 MinAbsMax: []
        NoStopIfFlatInfeas: []
            ObjectiveLimit: []
      PhaseOneTotalScaling: []
            Preconditioner: []
          PrecondBandWidth: []
            RelLineSrchBnd: []
    RelLineSrchBndDuration: []
              ScaleProblem: []
       SubproblemAlgorithm: []
                    TolCon: []
                 TolConSQP: []
                TolGradCon: []
                    TolPCG: []
                  TolProjCG: []
              TolProjCGAbs: []
                  TypicalX: []
               UseParallel: []
```

```
x = fzero(f, 1.6, opt)
```

```
Search for an interval around 1.6 containing a sign change:
 Func-count    a          f(a)             b          f(b)        Procedure
    1             1.6      0.274131           1.6      0.274131   initial interval
    3         1.55475      0.157999       1.64525      0.390694   search
    5           1.536      0.110027         1.664      0.439097   search
    7         1.50949     0.0423222       1.69051      0.507665   search
    8           1.472    -0.0531432       1.69051      0.507665   search

Search for a zero in the interval [1.472, 1.69051]:
 Func-count    x          f(x)             Procedure
    8           1.472    -0.0531432        initial
    9         1.49271  -0.000458365        interpolation
   10         1.49289   4.30326e-08        interpolation
   11         1.49289   -3.6593e-13        interpolation
   12         1.49289   4.44089e-16        interpolation
   13         1.49289   4.44089e-16        interpolation

Zero found in the interval [1.472, 1.69051]
x = 1.4929
```

Roots of single-variate algebraic polynomial

In many common cases, the non-linear equation we have solve is an algebraic polynomial, meaning that it only contains powers of the unknown variable multiplyed by some constants and can be given in the form:

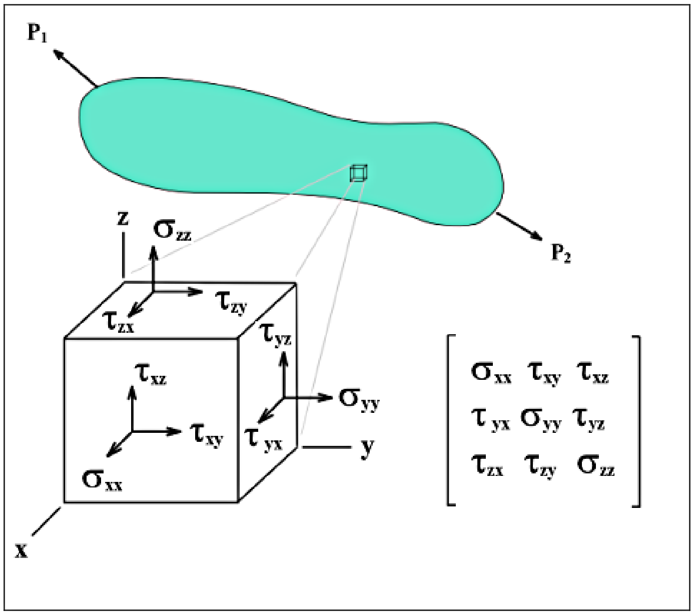
$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

The $a_n, a_{n-1}, \cdots, a_{1,} a_0$ coefficients are real numbers and n is the order of the polynomial.

Finding the roots of a function in the form above can be done in multiple ways in MATLAB. One of these ways is the ***roots*** function, which numerically solves for the polynomials roots by giving it the vector of coefficients (e.g. $[3 \quad -4 \quad 0 \quad 23]$ for the polynomial $3x^3 - 4x^2 - 23 = 0$). Another method is the ***solve*** function, which calculates the symbolic (in other words, exact) solutions.

Finding principal stresses and axes

An example from mechanics would be finding the principal stress values and their axes from the Cauchy stress tensor. The tensor expresses the stresses in a point in the following way (shown on the figure):



The principal axes would be where there is only normal stress, in other words the tensor looks like this:

$$\boldsymbol{F} = \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{bmatrix}, \text{ where } \sigma_1 > \sigma_2 > \sigma_3.$$

Mathematically, this is considered a eigenvalue-eigenvector problem. The form of the solution for calculating the principal stresses and the principal axes can be given as follows:

$$(\boldsymbol{F} - \sigma_e \cdot \boldsymbol{I}) \cdot e = 0$$

The non-trivial solutions are given where the determinant of the coefficient matrix is equal to 0:

$$\det((\boldsymbol{F} - \sigma_e \cdot \boldsymbol{I})) = 0$$

Consider an example, where the stress matrix is following:

$$\boldsymbol{F} = \begin{bmatrix} 50 & 20 & -40 \\ 20 & 80 & -30 \\ -40 & -30 & -20 \end{bmatrix}$$

We can use MATLAB to find the determinant symbolically of matrix $\boldsymbol{F}$:

```
F = [50, 20, -40; 20, 80, -30; -40, -30, -20] % stress matrix
```

```
F = 3×3
    50    20   -40
    20    80   -30
   -40   -30   -20
```

```
syms ps % create a symbolic variable called 'ps'
eq = det(F - eye(3) * ps) % the determinant of the coefficient matrix
```

eq = $-\mathrm{ps}^3 + 110\,\mathrm{ps}^2 + 1500\,\mathrm{ps} - 197000$

This is the so called characteristic equation of the matrix, which happens to be an algebraic polynomial. We can use the ***roots*** function to find its solutions if we give the vector of coefficients as an input fo the solver:

```
pssr = roots([-1, 110, 1500, -197000])
```

```
pssr = 3×1
  106.7674
  -41.3691
   44.6017
```

We can also use the ***solve*** function to find the symbolic solutions of the symbolic expression above:

```
sol1 = solve(eq)
```

sol1 =

$$\begin{pmatrix} \mathrm{root}(z^3 - 110\,z^2 - 1500\,z + 197000, z, 1) \\ \mathrm{root}(z^3 - 110\,z^2 - 1500\,z + 197000, z, 2) \\ \mathrm{root}(z^3 - 110\,z^2 - 1500\,z + 197000, z, 3) \end{pmatrix}$$

We have to convert this double in order to see the numeric values:

```
sol2 = double(sol1)
```

```
sol2 = 3×1
  -41.3691
   44.6017
  106.7674
```

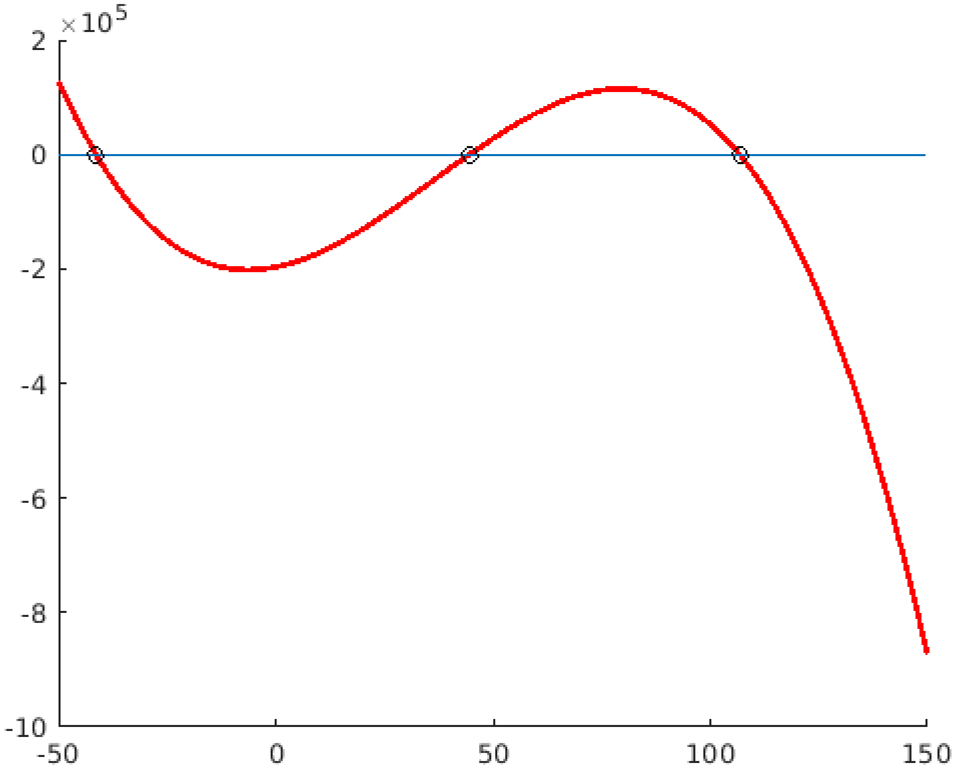
Checking the solution can be done by substituting them into the original equation, however, we must first convert the symbolic expression into an anonymous function:

```
eq2 = matlabFunction(eq);
eq2(sol2)
```

```
ans = 3×1
     0
     0
     0
```

Plot the equation and the solutions:

```
figure(4)
hold on;
g4 = fplot(eq, [-50, 150], 'Color', 'r', 'LineWidth', 2);
g5 = line([-50, 150], [0, 0]);
plot(sol2, eq2(sol2), 'ko');
```


×10^5
2
0
-2
-4
-6
-8
-10
-50
0
50
100
150

Systems of Linear Equations I.

In applied mathematics, systems of linear equations carry great importance as many engineering problems can be modelled using these systems. Working with systems of linear equations tends to be less complicated and therefore favored when compared to other methods. Moreover, many real physics problems can be considered linear in the close vicinity of the state we are investigating, which means that linear model not only supplies with a simple solution but also that the solution adequately approximates the real physical phenomenon.

In numerical methods, many other tools can be traced back and simplified to solving systems of linear equations (such as interpolation or regression). Solving systems of linear equations means doing many matrix operations which can be efficiently carried out using numerical environments such as MATLAB.

Example problem for a system of linear equations

For a construction we need the following materials:

- Sand: 4800 m^3
- Fine gravel: 5810 m^3
- Coarse gravel: 5960 m^3

Three mines supply the materials needed. The transported materials from each mine have the following composition:

1. mine: Sand: 52% - Fine gravel: 30% - Coarse gravel: 18%
2. mine: Sand: 20% - Fine gravel: 50% - Coarse gravel: 30%
3. mine: Sand: 25% - Fine gravel: 20% - Coarse gravel: 55%

Question: How much material should be transported from each mine to satisfy the material needs of the contruction?

Let x_1, x_2, x_3 denote the amount of materials in m^3 units transported from each mine. Each x value is positive of course. The system of equations denoting the materials supplied by all of the mines:

$$\begin{array}{ccccccc} 0.52 \cdot x_1 & + & 0.20 \cdot x_2 & + & 0.25 \cdot x_3 & = & 4800 \\ 0.30 \cdot x_1 & + & 0.50 \cdot x_2 & + & 0.20 \cdot x_3 & = & 5810 \\ 0.18 \cdot x_1 & + & 0.30 \cdot x_2 & + & 0.55 \cdot x_3 & = & 5960 \end{array}$$

This system of equations can be written in matrix form as the following:

$$\boldsymbol{A} = \begin{bmatrix} 0.52 & 0.20 & 0.25 \\ 0.30 & 0.50 & 0.20 \\ 0.18 & 0.30 & 0.55 \end{bmatrix} \qquad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \boldsymbol{b} = \begin{bmatrix} 4800 \\ 5810 \\ 5960 \end{bmatrix}$$

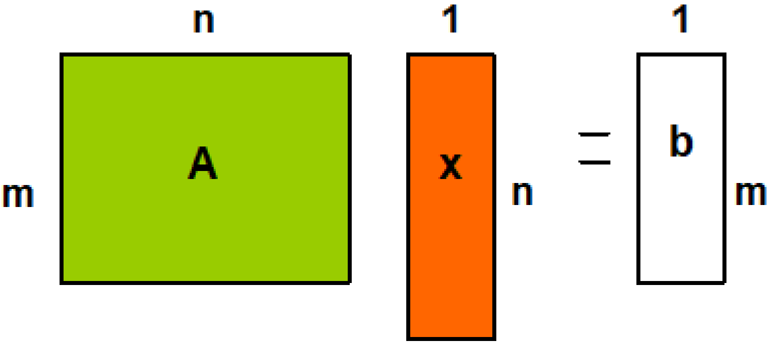
$$\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$$

This is of course a far simpler problem than what usually arises. It is not uncommon in engineering problems that we have to solve equations resulting matrices with thousands of rows and columns. It is because of this fact that we always have to consider the efficiency of the algorithms as well.

Existence of the solution for a linear system of equations

In general form, every system of linear equations can be given using a coefficient matrix ($\boldsymbol{A}$, with size m x n), a vector of unknowns ($\boldsymbol{x}$, with size n x 1) and right-hand side vector ($\boldsymbol{b}$, with size m x 1):

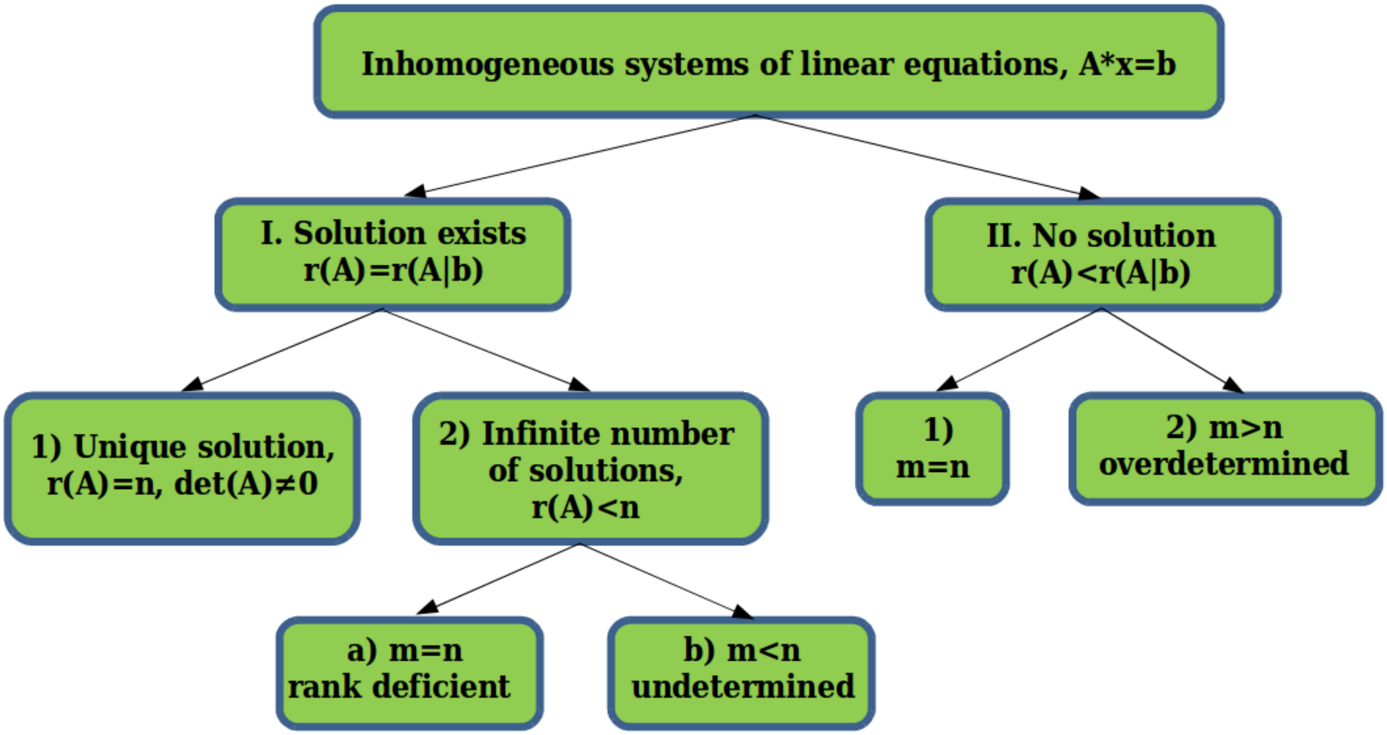
$$\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$$



We can talk about homogeneous systems when the vector $\boldsymbol{b}$ is equal to zero ($\boldsymbol{A} \cdot \boldsymbol{x} = 0$) and inhomogeneous systems when the vector $\boldsymbol{b} \neq 0$. In the homogeneous case, there is a non-trivial solution (when $\boldsymbol{x} \neq 0$) only, when the determinant of $\boldsymbol{A}$ is zero ($\det(\boldsymbol{A}) = 0$), in other words, the matrix $\boldsymbol{A}$ is singular. If there is a non-trivial solution, it means that there are infinitely many solutions (as we can just multiply any solution by an arbitrary number and still get 0).

In the inhomogeneous case, we have no solutions if the rank of matrix $\boldsymbol{A}(\operatorname{rank}(\boldsymbol{A}))$ is less then the rank of augmented coefficient matrix (when we concatenate the right-hand side vector $\boldsymbol{b}$ to the right of matrix $\boldsymbol{A}$, that is, we have $[\boldsymbol{A} \quad \boldsymbol{b}]$), or in mathematical terms $\operatorname{rank}(\boldsymbol{A}) < \operatorname{rank}([\boldsymbol{A} \quad \boldsymbol{b}])$. This basically means, that we cannot construct the right-hand side vector from the linear combination of the columns of $\boldsymbol{A}$, in other words, the right-hand side vector does not lie in the column space of $\boldsymbol{A}$. In these cases we may still have an approximate solution, which does not solve the equations exactly, but strives to tries to solve them with the least amount of residuals (least squares solution). If the number of rows of the coefficient matrix is bigger than the number of columns, we say that the problem is overdetermined (more equations than unknowns).

When we do have a solution, we can have to cases. Firstly, having a solution means, that the rank of the coefficient matrix is equal to the rank of the augmented coefficient matrix, that is $\operatorname{rank}(\boldsymbol{A}) = \operatorname{rank}([\boldsymbol{A} \quad \boldsymbol{b}])$. In this case, the right-hand side vector lies in the column space of $\boldsymbol{A}$ and therefore, the augmented matrix has the same rank as the original matrix because the vector $\boldsymbol{b}$ is not linearly independent of the columns of $\boldsymbol{A}$ (it can be constructed from the columns of $\boldsymbol{A}$). We have an exact solution when the coefficient matrix is full rank, meaning $\operatorname{rank}(\boldsymbol{A}) = n$, where n is the number of columns and the determinant of the matrix is not zero. If the determinant is zero, that is the matrix is singular, we cannot invert is which means we cannot find the solution. (Singular matrices can be inverted as well, more on that later.) If $\operatorname{rank}(\boldsymbol{A}) < n$, then we have infinitely many solutions as there is at least one linearly dependent column in $\boldsymbol{A}$. In this case, we can talk about an underdetermined system, when $\boldsymbol{A}$ has less rows than columns, in other words, there are more unknowns than equations, therefore there have to be dependent uknowns and hence the infinite number of solutions. The other case is when $\boldsymbol{A}$ is a square matrix, if this happens, we say that $\boldsymbol{A}$ is rank deficient. When having and infinite number of solutions, we can always find a solution which has the smallest norm (least norm solution). The solution possibilities are visualized in the figure below:



Existence of solution for our example

Let's check if a solution exists for the example problem. The problem is inhomogeneous as $\boldsymbol{b} \neq 0$.

Defining the variables:

```
clear all; close all;
A = [0.52, 0.20, 0.25; 0.30, 0.50, 0.20; 0.18, 0.30, 0.55]
```

```
A = 3×3
        0.52          0.2         0.25
         0.3          0.5          0.2
        0.18          0.3         0.55
```

```
b = [4800; 5810; 5960]
```

```
b = 3×1
        4800
        5810
        5960
```

Create the augmented coefficient matrix:

```
aug = [A, b] % concatenate matrix A and vector b
```

```
aug = 3×4
```

```
        0.52          0.2          0.25         4800
         0.3          0.5           0.2         5810
        0.18          0.3          0.55         5960
```

Check if the rank of $\boldsymbol{A}$ is equal to the rank of $[\boldsymbol{A} \quad \boldsymbol{b}]$:

```
rank(A) == rank(aug) % logical equality operator: is rank(A) equal to rank(aug)?
```

```
ans = logical
   1
```

The answer is logical 1 (meaning yes). The ranks of the two matrices are equal, therefore we have a solution. Let's check if $\boldsymbol{A}$ is full rank or rank deficient, in other words, whether we have an exact solution of infinitely many:

```
[rownum, colnum] = size(A) % if the size function only has one input, it returns two outputs
```

```
rownum =
     3
colnum =
     3
```

```
rownum = size(A, 1) % only the number of rows (the 1st dimension)
```

```
rownum =
     3
```

```
colunm = size(A, 2) % only the number of columns (the 2nd dimension)
```

```
colunm =
     3
```

The rank of the matrix is equal to the number of columns which means that we have one solution. Does the solution exist? We have to check the determinant of $\boldsymbol{A}$ to find out:

```
det(A)
```

```
ans =
          0.086
```

The determinant is not zero and therefore our one solution exists.

Let's solve the system using multiple built-in methods:

```
% computing the inverse of A directly
x1 = inv(A)*b
```

```
x1 = 3×1
        3744.8
        7071.7
        5753.5
```

```
% using the '\' operator
x2 = A\b
```

```
x2 = 3×1
       3744.8
       7071.7
       5753.5
```

```
% using the linsolve function
x3 = linsolve(A, b)
```

```
x3 = 3×1
       3744.8
       7071.7
       5753.5
```

All methods give the same solution naturally, but let's check if they truly solve the system. The easiest way is to calculate the left-hand side $\boldsymbol{A} \cdot \boldsymbol{x}$, subtract is from the right-hand side $\boldsymbol{b}$ and calculate the norm of the resulting vector. If this is zero, than the solution clearly solves the system.

```
res = A*x1 - b
```

```
res = 3×1
     0
     0
     0
```

```
norm(res)
```

```
ans =
     0
```

Matrix decompositions

When solving systems of linear equations by hand, we use the Gaussian elimination method (or maybe the Gauss-Jordan elimination, where we mutiply the rows of the augmented coefficient matrix by constants and subtract them from each other until we get an upper triangular matrix. For example, we have an augmented coeff. matrix:

$$\begin{bmatrix} 1 & -2 & 3 & 4 \\ 2 & -5 & 12 & 15 \\ 0 & 2 & -10 & -10 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & -2 & 3 & 4 \\ 0 & -1 & 6 & 7 \\ 0 & 2 & -10 & -10 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & -2 & 3 & 4 \\ 0 & -1 & 6 & 7 \\ 0 & 0 & 2 & 4 \end{bmatrix}$$

From the third row:

$$2x_3 = 4 \rightarrow x_3 = 2$$

From the second row:

$$-1x_1 + 6x_2 = 7 \rightarrow -1x_1 + 12 = 7 \rightarrow x_2 = 5$$

From the first row:

$$x_1 - 2x_2 + 3x_3 = 4 \rightarrow x_1 - 10 + 6 = 4 \rightarrow x_1 = 8$$

Sometimes, we have to solve multiple systems of linear equations that have the same coefficient matrices and only the right-hand side changes. In out example, we can imagine that we are simulating using the same mine for different constructions with different material needs. As most of the work during the elimination is done on the matrix $\boldsymbol{A}$, it would be efficient, if we could save the steps of the elimination and just reuse them each time. This is where matrix decompositions come into the picture.

LU decomposition

The LU decomposition splits the matrix into an upper triangular matrix $\boldsymbol{U}$, and a lower triangular matrix $\boldsymbol{L}$, in such a way that:

$$\boldsymbol{P} \cdot \boldsymbol{A} = \boldsymbol{L} \cdot \boldsymbol{U}$$

where $\boldsymbol{P}$ is a permutation matrix denoting the row switches in the matrix (if there were any). The upper triangular matrix contains the form after the Gaussian elimination and the lower triangular matrix contains the operations needed to get there.

We can create the LU decomposition of the coefficient matrix the following way:

```
[L, U, P] = lu(A)
```

```
L = 3×3
            1            0            0
      0.57692            1            0
      0.34615          0.6            1
U = 3×3
         0.52          0.2         0.25
            0      0.38462     0.055769
            0            0         0.43
P = 3×3
     1     0     0
     0     1     0
     0     0     1
```

To solve the system using the decomposition, we first have to permute both sides of the equation using the matrix $\boldsymbol{P}$.

$$\boldsymbol{P} \cdot \boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{P} \cdot \boldsymbol{b} = \boldsymbol{d}$$

Then, we can substitute $\boldsymbol{P} \cdot \boldsymbol{A} = \boldsymbol{L} \cdot \boldsymbol{U}$:

$$\boldsymbol{L} \cdot \boldsymbol{U} \cdot \boldsymbol{x} = \boldsymbol{d}$$

Let $\boldsymbol{y} = \boldsymbol{U} \cdot \boldsymbol{x}$, then we get $\boldsymbol{L} \cdot \boldsymbol{y} = \boldsymbol{d}$. Now we only have to solve the two equations.

```
d = P * b % permuted b vector
```

```
d = 3×1
        4800
        5810
        5960
```

```
opt1.LT = true; % create an option for the solver telling it that the matrix we are using is lo
y = linsolve(L, d, opt1)
```

```
y = 3×1
        4800
      3040.8
        2474
```

```
opt2.UT = true % telling the solver that now we have an upper triangular matrix
```

```
opt2 = struct with fields:
    UT: 1
```

```
xLU = linsolve(U, y, opt2)
```

```
xLU = 3×1
      3744.8
      7071.7
      5753.5
```

```
norm(A * xLU - b) % checking the solution
```

```
ans =
     0
```

Cholesky decomposition

The Cholesky decomposition is similar to the LU decomposition but now the matrix $\boldsymbol{A}$ is split into a upper triangular matrix $\boldsymbol{L}$ in such a way that:

$$\boldsymbol{A} = \boldsymbol{L}^T \cdot \boldsymbol{L}$$

Using this decomposition, the solution becomes: $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{L}^T \cdot \boldsymbol{L} \cdot \boldsymbol{x} = \boldsymbol{L}^T \cdot \boldsymbol{y} = \boldsymbol{b}$.

- $\boldsymbol{L}^T \cdot \boldsymbol{y} = \boldsymbol{b}$, where $\boldsymbol{L}^T$ is a lower triangular matrix.
- $\boldsymbol{L} \cdot \boldsymbol{x} = \boldsymbol{y}$, where $\boldsymbol{L}$ is an upper triangular matrix.

The Cholesky decomposition can only be done if the following criteria are met:

- The matrix $\boldsymbol{A}$ is symmetric, that is $\boldsymbol{A}^T = \boldsymbol{A}$ and
- The matrix $\boldsymbol{A}$ is positive definite, that is every eigenvalue of the matrix is greater than 0: $\lambda_i > 0,\ i = 1, \cdots, n$.

Let us check if $\boldsymbol{A}$ in our example meets the criteria:

```
norm(A' - A) % A is definitely not symmetric
```

```
ans =
      0.1578
```

The norm is not 0, therefore the matrix is not symmetric. We can also see this by simply looking at the values in the matrix.

Consider a different example, let the coefficient matrix be a matrix of binomial coefficients and let's have an arbitrary right-hand side vector vector:

```
B = pascal(4) % creates a 4x4 matrix of binomial coefficients
```

```
B = 4×4
     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20
```

```
r = rand(4, 1) % random 4x1 vector
```

```
r = 4×1
      0.65574
     0.035712
      0.84913
      0.93399
```

Check if our new coefficient matrix is symmetric and positive definite:

```
norm(B' - B) % symmetry check
```

```
ans = 
     0
```

```
[V, D] = eig(B) % eigenvectors and eigenvalues of the matrix
```

```
V = 4×4
      0.30869     -0.78728      0.53037     0.060187
     -0.72309      0.16323      0.64033      0.20117
      0.59455      0.53211      0.39183      0.45808
     -0.16841     -0.26536      -0.3939      0.86375
D = 4×4
     0.038016            0            0            0
            0      0.45383            0            0
            0            0       2.2034            0
            0            0            0       26.305
```

```
min(diag(D)) > 0 % the eigenvalues are in the diagonal of matrix D, check if the minimum if gre
```

```
ans = logical
   1
```

Our new matrix $\boldsymbol{B}$ meets the criteria, so we can use the Cholesky decomposition:

```
L = chol(B)
```

```
L = 4×4
     1     1     1     1
     0     1     2     3
     0     0     1     3
     0     0     0     1
```

```
norm(L'*L - B) % check the decomposition
```

```
ans =
     0
```

```
opt1.LT = true; % first we compute with L' which is lower triangular
y = linsolve(L', r)
```

```
y = 4×1
      0.65574
     -0.62003
       1.4334
       -2.162
```

```
opt1.UP = true; % now we are working with L which is upper triangular
xchol = linsolve(L, y)
```

```
xchol = 4×1
       4.8712
      -9.9729
       7.9194
       -2.162
```

```
norm(B * xchol - r) % check the correctness of the solution
```

```
ans =
   9.7554e-15
```

Infinite number of solutions (underdetermined system)

Sometimes we have fewer equations than unknowns and therefore an underdetermined system. We discussed before that in these cases, the system of equations has infinitely many solutions but we can always find one particular solution which has the smallest norm of all the possibilites.

If we have more unknowns than equations, than some unknowns can be arbitrarily chosen and the values of the other unknows vary depending on these. Mathematically, this can happen, when $\mathrm{rank}(\boldsymbol{A}) = \mathrm{rank}([\boldsymbol{A} \quad \boldsymbol{b}])$, but $\mathrm{rank}(\boldsymbol{A}) < n$. We might have a square coefficient which is rank deficient, for example:

$$\begin{array}{rcrcl} x & + & y & = & 3 \\ 2x & + & 2y & = & 6 \end{array}$$

$$\boldsymbol{A} = \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix} \rightarrow \boldsymbol{A} \text{ is rank deficient}$$

The other case would be the underdetermined system. We can achieve the least norm solution $(\min(|\boldsymbol{x}|)$ using the following formula:

$$\boldsymbol{x} = \boldsymbol{A}^T \cdot (\boldsymbol{A} \cdot \boldsymbol{A}^T)^{-1} \cdot \boldsymbol{b}$$

Let's solve the following system and calculate the least norm solution:

$$7x_1 + 2x_2 + 2x_4 = 1$$
$$x_1 + 8x_2 + x_3 + 8x_4 = 2$$

$$\boldsymbol{A} = \begin{bmatrix} 7 & 2 & 0 & 2 \\ 1 & 8 & 1 & 8 \end{bmatrix}, \qquad \boldsymbol{b} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Check the number of solutions:

```
A = [7, 2, 0, 2; 1, 8, 1, 8]
```

```
A = 2×4
     7     2     0     2
     1     8     1     8
```

```
b = [1; 2]
```

```
b = 2×1
     1
     2
```

```
rank(A)
```

```
ans = 
     2
```

```
rank([A, b])
```

```
ans = 
     2
```

```
size(A, 2)
```

```
ans = 
     4
```

We can clearly see that the rank of the coefficient matrix is equal to the rank of the augmented matrix, which means there is a solution, however, the rank is less than the number of columns of the matrix, which means infinitely many solutions. Let's calculate the least norm solution:

```
x = A' * inv(A * A') * b
```

```
x = 4×1
     0.074546
      0.11954
     0.012736
      0.11954
```

```
norm(A*x - b) % checking the solution
```

```
ans = 
      0
```

```
norm(x)
```

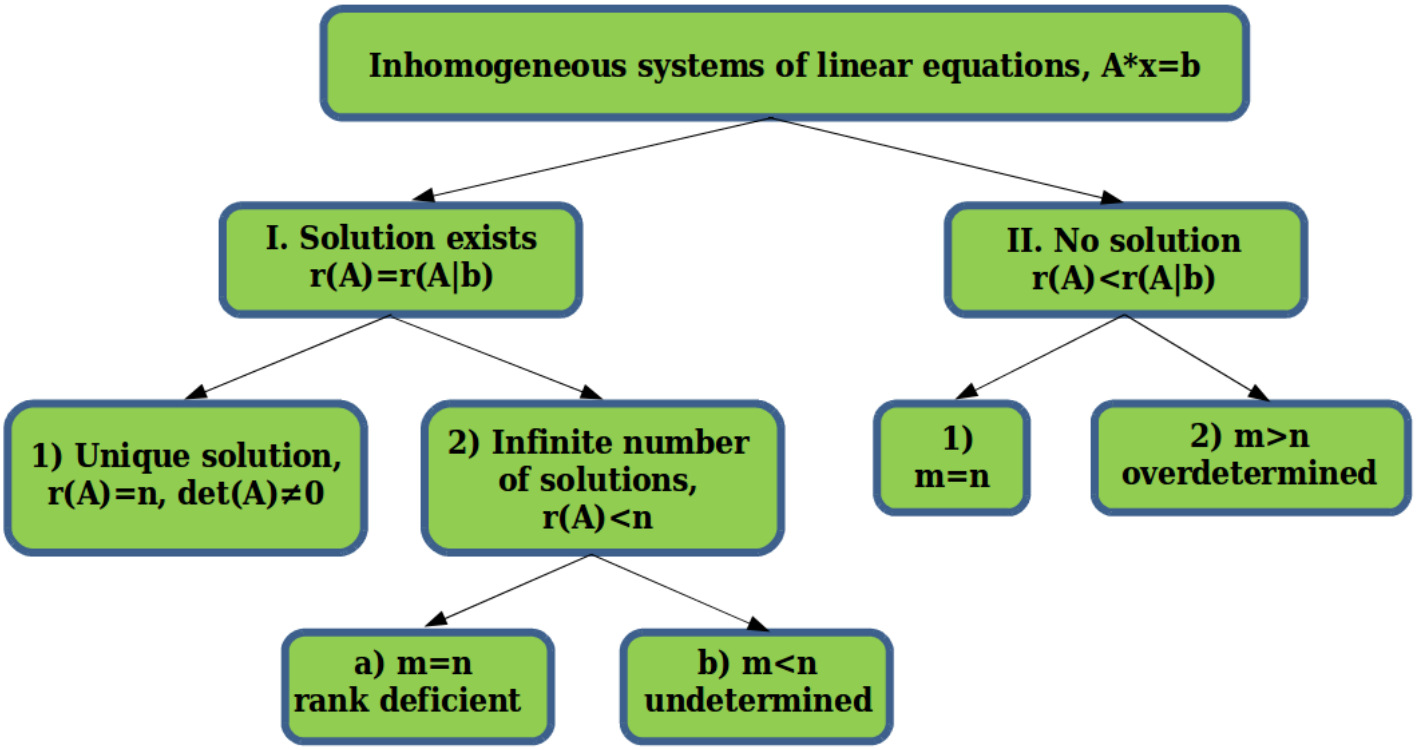
```
ans = 
      0.18521
```

The solution vector does solve the system and norm of the solution vector is 0.1852. This solution has the smallest norm of all the possible solutions.

Calculating the inverse is usually slower than using a matrix decomposition. The problem is that we cannot use the LU and Cholesky decompositions on a rectangular matrix as they only work on square matrices. The problem can only be solved other decompositions, such as the QR decomposition and the Singular Value Decomposition (SVD).

Systems of linear equations II.

During the previous practical, we looked at systems of linear equations which have either one unique solution, or an infinite number of solutions. In the latter case, we can still find a unique solution, namely if we prescribe the constrict of $\|\boldsymbol{x}\| \to \min$, where $\boldsymbol{x}$ is the solution vector.



As we saw, the computation of the inverse of the coefficient matrix used to solve the system of equations can be very computationally expensive, especially as the size of the matrix increases. Moreover, depending on the algorithm used, the result of the algorithm can be numerically less precise, which in certain cases (for example, when calculating the inverse multiple times) can lead to unintended round-off errors. It is always recommended to use some kind of decomposition or factorization of the coefficient matrix when solving linear systems, however, the LU and Cholesky decompositions that we have studied only work on square matrices (with even more constrictions in the case of the Cholesky decomposition). If our matrix is rectangular, as in the case of underdetermined or overdetermined systems, we may use the QR factorization or the Singular Value Decomposition (SVD).

QR factorization

The QR factorization uses the fact that every matrix $\boldsymbol{A}$, independent of its shape, can be decomposed into the product of an orthonormal matrix $\boldsymbol{Q}$ and an upper rectangular matrix $\boldsymbol{R}$, such that $\boldsymbol{A} = \boldsymbol{Q} \cdot \boldsymbol{R}$. The orthonormality of matrix $\boldsymbol{Q}$ means that $\boldsymbol{Q}^T\boldsymbol{Q} = \boldsymbol{I}$, that is, the inverse of $\boldsymbol{Q}$ is equal to its transpose.

The original $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$ system becomes:

$$\boldsymbol{Q} \cdot \boldsymbol{R} \cdot \boldsymbol{x} = \boldsymbol{b}.$$

We can multiply from the left by $\boldsymbol{Q}^T$ (because $\boldsymbol{Q}$ is orthonormal), so we don't have to use invertion.

$$\boldsymbol{Q}^T \cdot \boldsymbol{Q} \cdot \boldsymbol{R} \cdot \boldsymbol{x} = \boldsymbol{Q}^T \cdot \boldsymbol{b}$$

$$\boldsymbol{R} \cdot \boldsymbol{x} = \boldsymbol{Q}^T \cdot \boldsymbol{b} = \boldsymbol{B}$$

The second step is solving the $\boldsymbol{R} \cdot \boldsymbol{x} = \boldsymbol{B}$ equation, but because $\boldsymbol{R}$ is an upper rectangular matrix, we can simply solve this by back-substitution.

There are multiple algorithms to compute this factorization, such as using the Gram-Schmidt process to find the orthonormal basis of the column vectors of $\boldsymbol{A}$, the Household-transformation and more. Calculating the QR factorization using the Gram-Schmidt process is the simplest way, however, it tends to be numerically unstable in certain cases and therefore usually other methods are implemented in numerical algorithms.

Let's solve the example from the previous practical:

$$7x_1 + 2x_2 + 2x_4 = 1$$
$$x_1 + 8x_2 + x_3 + 8x_4 = 2$$

$$\boldsymbol{A} = \begin{bmatrix} 7 & 2 & 0 & 2 \\ 1 & 8 & 1 & 8 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Remember that the example has infinitely many solutions as the rank of the coefficient matrix is only 2. As the coefficient matrix is rectangular, we have to use the QR factorization to decompose it.

```
clear all; close all; format short
A = [7, 2, 0, 2; 1, 8, 1, 8]; b = [1; 2];
rA = rank(A) % rank of A
```

```
rA = 2
```

```
[Q, R] = qr(A) % QR factorization
```

```
Q = 2×2
   -0.9899   -0.1414
   -0.1414    0.9899
R = 2×4
   -7.0711   -3.1113   -0.1414   -3.1113
         0    7.6368    0.9899    7.6368
```

```
norm(Q' * Q) % checking if Q is orthonormal
```

```
ans = 1
```

```
norm(A - Q*R) % checking the correctness of the factorization
```

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

Calculate the new right side of the system:

```
B = Q' * b
```

```
B = 2×1
   -1.2728
    1.8385
```

Solve the second step using the fact that $\boldsymbol{R}$ is an upper rectangular matrix:

```
opts.UT = true
```

```
opts = struct with fields:
    UT: 1
```

```
x = linsolve(R, B, opts)
```

```
x = 4×1
    0.0741
    0.2407
         0
         0
```

```
norm(A*x - b) % checking the solution
```

```
ans = 0
```

```
norm(x) % norm of the solution vector
```

```
ans = 0.2519
```

The solution is correct, but its norm is not the same as the norm of the solution from the end of the previous practical (0.1852). The solution by QR factorization does not give us the solution with the least norm, rather it gives us the solution with the most zero entries. Depending on the concrete problem, one or the other might be more suitable.

Singluar Value Decomposition (SVD)

The SVD is another form of decomposition that can be used arbitrarily shaped matrices. It is very similar to the eigendecomposition which is used in the case of square matrices:

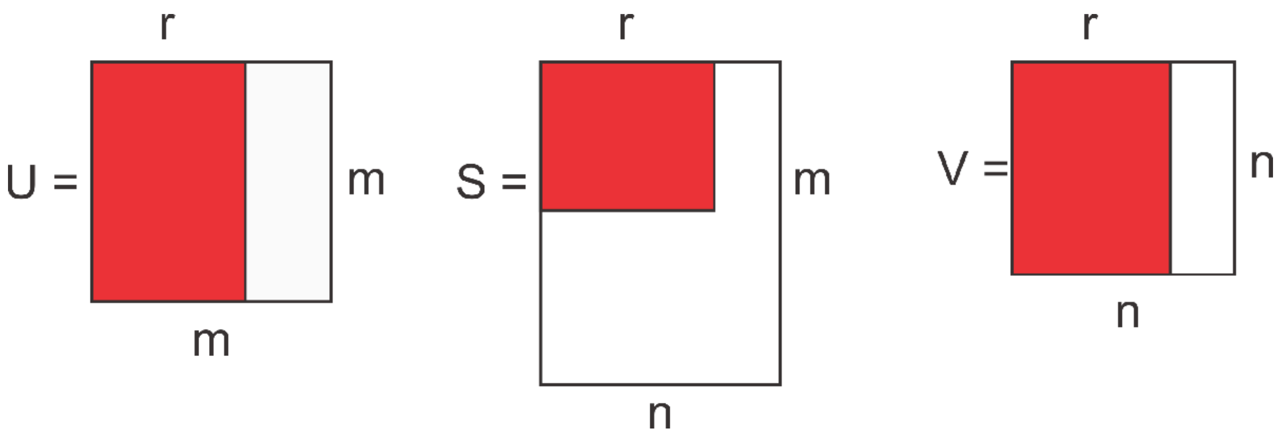
- Eigenvalues: matrix $\boldsymbol{A}$ (with size m x m) has an eigenvalue λ is we can find a vector $\boldsymbol{v}$ ($\boldsymbol{v} = 0$) such, that $\boldsymbol{A} \cdot \boldsymbol{v} = \lambda \cdot \boldsymbol{v}$. In this case, the vector $\boldsymbol{v}$ is called an eigenvector of $\boldsymbol{A}$ corresponding to the eigenvalue λ. The eigenvalues can be found using the characteristic polinomial of the matrix $|\boldsymbol{A} - \lambda \boldsymbol{I}| = 0$. In MATLAB, we can use the `eig` function: `[V, D] = eig(A)`.
- Singular values: the singular values of a matrix $\boldsymbol{A}$ (with size m x n) are the squares of the non-zero eigenvalues of the product $\boldsymbol{A}^T\boldsymbol{A}$ ($\sigma_i = \sqrt{\lambda_i}$, where σ_i is the i-th singular value and λ_i is the i-th eigenvalue of the product $\boldsymbol{A}^T\boldsymbol{A}$). In MATLAB, we can compute the SVD by using the `svd` function: `[U, S, V] = svd(A)`.

The SVD decomposes the matrix $\boldsymbol{A}$ (m x n) into the product of three matrices:

$$\boldsymbol{A} = \boldsymbol{U} \cdot \boldsymbol{S} \cdot \boldsymbol{V}^{\boldsymbol{T}}$$

where:

- $\boldsymbol{U}$ is an orthonormal matrix ($\boldsymbol{U}^{\boldsymbol{T}}\boldsymbol{U} = \boldsymbol{I}$) with size m x m, the first *r* columns of which are the eigenvectors of the product $\boldsymbol{A}\boldsymbol{A}^{\boldsymbol{T}}$.
- $\boldsymbol{S}$ is a diagonal matrix with size m x n that contains the singular values of $\boldsymbol{A}$ on its diagonal.
- $\boldsymbol{V}$ is a orthonormal matrix ($\boldsymbol{V}^{\boldsymbol{T}}\boldsymbol{V} = \boldsymbol{I}$) with size n x n, the first *r* columns of which are the eigenvectors of the product $\boldsymbol{A}^{\boldsymbol{T}}\boldsymbol{A}$.



When using the SVD of a matrix to solve a linear system of equations, the great thing about it is that the inverse of matrix $\boldsymbol{A}$ can be very easily computed because of the structure of the resulting matrices:

$$\boldsymbol{A}^{-1} = \boldsymbol{U}^{-1} \cdot \boldsymbol{S}^{-1} \cdot (V^T)^{-1}$$

Because of the orthogonality of $\boldsymbol{U}$ and V:

$$\boldsymbol{A}^{-1} = \boldsymbol{U}^{\boldsymbol{T}} \cdot \boldsymbol{S}^{-1} \cdot \boldsymbol{V}, \text{ which can be rewritten as } \boldsymbol{A}^{-1} = \boldsymbol{V} \cdot \boldsymbol{S}^{-1} \cdot \boldsymbol{U}^{\boldsymbol{T}}.$$

We can see, that only the matrix $\boldsymbol{S}$ has to be inverted, which is really simple as $\boldsymbol{S}$ is a diagonal matrix:

$$\boldsymbol{S} = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \sigma_r & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \rightarrow \boldsymbol{S}^{-1} = \begin{bmatrix} \frac{1}{\sigma_1} & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \frac{1}{\sigma_2} & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \frac{1}{\sigma_r} & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Let's solve the previous example using the SVD:

```
[U, S, V] = svd(A) % SVD of matrix A
```

```
U = 2×2
   -0.3979    0.9174
   -0.9174   -0.3979
S = 2×4
   12.1209         0         0         0
         0    6.3312         0         0
V = 4×4
   -0.3055    0.9515    0.0368   -0.0015
   -0.6712   -0.2130   -0.0933   -0.7039
   -0.0757   -0.0629    0.9943   -0.0406
   -0.6712   -0.2130   -0.0356    0.7092
```

Check the resulting properties of the matrices:

```
norm(A - U*S*V') % matrix A is correctly factored
```

```
ans = 2.6295e-15
```

```
norm(U' * U) % U is an orthonormal matrix
```

```
ans = 1.0000
```

```
V' * V - eye(4) % V is an orthonormal matrix
```

```
ans = 4×4
10^-15 ×
    0.2220    0.2220    0.0139   -0.0538
    0.2220         0         0    0.0796
    0.0139         0   -0.1110    0.0139
   -0.0538    0.0796    0.0139         0
```

Compute the inverse using the results of the SVD:

```
invS = (1./S)' % take the diagonal elements of S and take the reciprocal of each element
```

```
invS = 4×2
    0.0825       Inf
       Inf    0.1579
       Inf       Inf
       Inf       Inf
```

```
invS(invS == Inf) = 0 % make every Inf element in invS equal to 0
```

```
invS = 4×2
    0.0825         0
         0    0.1579
         0         0
         0         0
```

```
invA = V * invS * U' % inverse of matrix A calculated using the SVD matrices
```

```
invA = 4×2
    0.1479   -0.0367
   -0.0088    0.0642
   -0.0066    0.0097
   -0.0088    0.0642
```

Solve the system of equations:

```
x = invA * b
```

```
x = 4×1
    0.0745
    0.1195
    0.0127
    0.1195
```

```
norm(A*x - b) % check correctness of the results
```

```
ans = 9.1551e-16
```

```
norm(x) % norm of the solution
```

```
ans = 0.1852
```

The norm of the solution is the same as norm from the last practical, in other words, the SVD solution calculates the least norm solution as well.

The inverse matrix calculated with the SVD is also called the pseudo-inverse or generalized inverse. When the matrix is not singular ($\det(\boldsymbol{A}) \neq 0$), the generalized inverse is the same as the normal inverse, but in the case of a singular matrix ($\det(\boldsymbol{A}) = 0$), only the pseudo-inverse can be calculated.

Apart from solving systems of linear equations, the SVD is a widely used method in applied mathematics, computer vision, statistics and signal analysis. Some of its applications include data approximation (when working with very big matrices), dimensionality reduction, trend analysis and forecasting (recommendation services for companies like Netflix, Amazon etc.).

Built-in functions for solving systems of linear equations with rectangular coefficient matrix

The built-in `pinv` command uses the SVD of the matrix to calculate its generalized inverse when the matrix is rectangular. If the matrix is square and non-singular, the `pinv` function shouldn't be used to calculate the inverse (it is computationally expensive).

```
x = pinv(A)*b % solution using the built-in pinv function
```

```
x = 4×1
    0.0745
    0.1195
    0.0127
    0.1195
```

The results are the same as from the 'manual' calculation.

The other built-in functions for solving systems of linear equations:

- The \ operator (`mldivide`) uses the LU or the Cholesky decomposition for square matrices and the QR factorization for rectangular matrices.

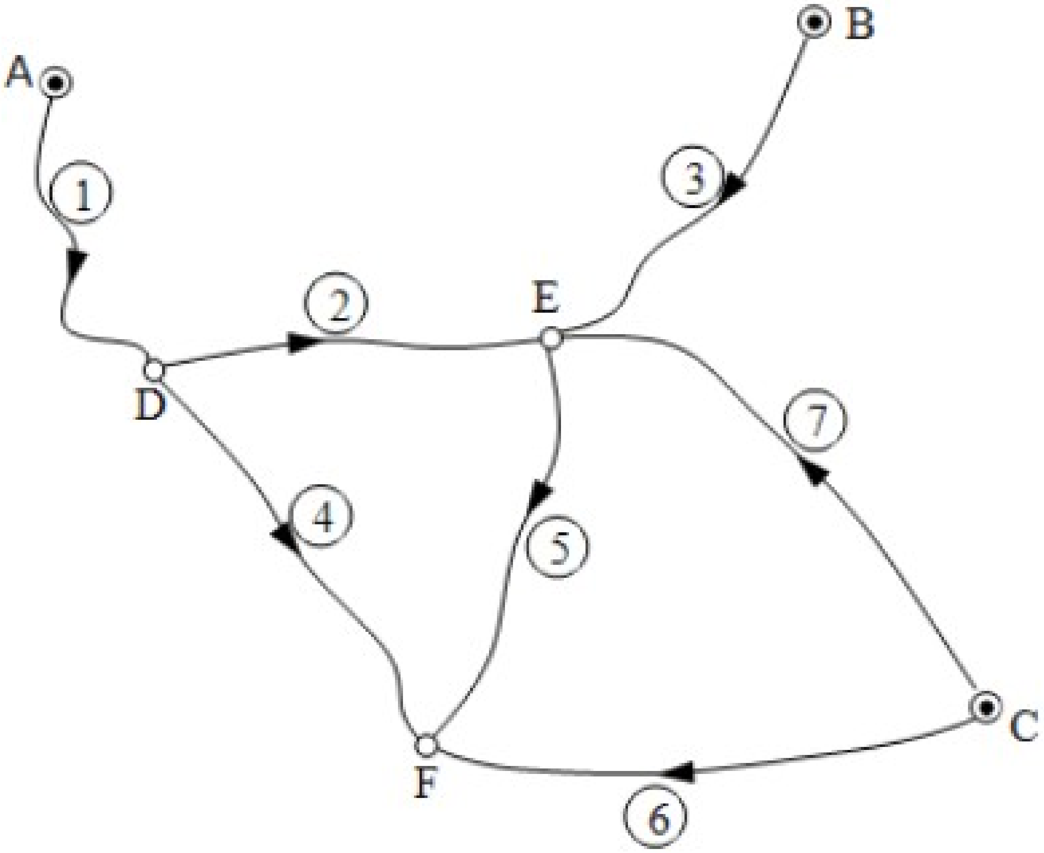
- The `linsolve` uses the same methods as the `\` operator, but it can be parametrized using preliminary knowledge about the structure of the matrix (upper/lower triangular nature).

Overdetermined System (no solution exists)

In most engineering problems, we are using some kind of measurement and calculating parameters values using some form of mathematical model of the given physical phenomenon. It often happens that in order to minimize the effect of measurements errors (at least those, that cannot be eliminated by choosing a specific configuration), we make more measurements than there are unknowns in the mathematical model. If the model is linear, then what we end up with is an overdetermined system of linear equations where we have more equations than unknowns.

As the measurements are never perfect and there are always random fluctuations in the values, we have very little hope of finding an exact solution. However, we can still find a solution which minimizes the sum of the squares of the discrepancies ($\sum_i e_i^2 \to \min$, where e_i is difference between the i-th unknown computed value and its theoretical true value (also called error). This is more commonly called the least-squares solution.

Take a look at an example from surveying. In surveying we almost always have more measurements than what is absolutely necessary, in order to compensate for measurements errors and eliminate blunders. Consider the following levelling network:



We measured the height differences along levelling lines 1-7. We also have three benchmarks (points with known elevations) in the area, A, B and C. Our goal is to calculate the elevation of points D, E and F using all the available measurements. As we have 7 measurements and 3 uknown elevations (m > n in matrix $\boldsymbol{A}$), we can only find an optimal solution when the sum of the squared errors of the elevations are minimalized. (The arrows in the figure show the direction of rise in the lines.)

Our system of equations looks like the following:

$$\begin{aligned}
\text{1st line}: &\quad H_D - H_A = H_D - 183.506 = +6.135 \\
\text{2nd line}: &\quad H_E - H_D = +8.343 \\
\text{3rd line}: &\quad H_E - H_B = H_E - 192.353 = +5.614 \\
\text{4th line}: &\quad H_F - H_D = +1.394 \\
\text{5th line}: &\quad H_F - H_E = -6.969 \\
\text{6th line}: &\quad H_F - H_C = H_F - 191.880 = -0.930 \\
\text{7th line}: &\quad H_E - H_C = H_E - 191.880 = +6.078
\end{aligned}$$

In matrix form:

$$\boldsymbol{A} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \boldsymbol{x} = \begin{bmatrix} H_D \\ H_E \\ H_F \end{bmatrix} \quad \boldsymbol{b} = \begin{bmatrix} 6.135 + 183.506 \\ 8.343 \\ 5.614 + 192.353 \\ 1.394 \\ -6.969 \\ -0.930 + 191.880 \\ 6.078 + 191.880 \end{bmatrix} = \begin{bmatrix} 189.641 \\ 8.343 \\ 197.967 \\ 1.394 \\ -6.969 \\ 190.950 \\ 197.958 \end{bmatrix}$$

Instead of typing the matrix and the vector into MATLAB, we can use the already existing `levelling.txt` file that contains the values:

```
clear all
Ab = load('levelling.txt') % loading the values from a text file, gives back 7 x 4 matrix
```

```
Ab = 7×4
    1.0000         0         0  189.6410
   -1.0000    1.0000         0    8.3430
         0    1.0000         0  197.9670
   -1.0000         0    1.0000    1.3940
         0   -1.0000    1.0000   -6.9690
         0         0    1.0000  190.9500
         0    1.0000         0  197.9580
```

```
A = Ab(:, 1:3) % the first 3 columns are the matrix A
```

```
A = 7×3
     1     0     0
    -1     1     0
     0     1     0
    -1     0     1
     0    -1     1
     0     0     1
     0     1     0
```

```
b = Ab(:, 4) % the 4th column is the vector b
```

```
b = 7×1
```

```
  189.6410
    8.3430
  197.9670
    1.3940
   -6.9690
  190.9500
  197.9580
```

Check the existence of the solution:

```
rA = rank(A)
```

```
rA = 3
```

```
rAug = rank([A, b])
```

```
rAug = 4
```

The rank of $\boldsymbol{A}$ is smaller than the rank of the augmented matrix, which means that no exact solution exits. As we have an overdetermined system, what we can be looking for, is the solution with least error, that is, the least-squares solution. Mathematically this is a solution with the constraint $\|\boldsymbol{A} \cdot \boldsymbol{x} - \boldsymbol{b}\| \rightarrow \min$. It can be derived, that this solution is the following:

$$\boldsymbol{x} = \left(\boldsymbol{A}^{\boldsymbol{T}} \cdot \boldsymbol{A}\right)^{-1} \cdot \boldsymbol{A}^{\boldsymbol{T}} \cdot \boldsymbol{b}$$

In MATLAB:

```
x = inv(A' * A) * A' * b
```

```
x = 3×1
  189.6153
  197.9588
  190.9830
```

As calculating the inverse is computationally expensive, this method is not commonly used in numerical solutions. We can instead use the QR factorization and the SVD decomposition:

Solution using the QR decomposition:

```
[Q, R] = qr(A) % QR decomposition of A
```

```
Q = 7×7
   -0.5774   -0.1741   -0.3077    0.6243    0.2670   -0.0164   -0.2835
    0.5774   -0.3482    0.0615    0.0684    0.5971    0.2659   -0.3312
         0   -0.5222   -0.2462   -0.3761   -0.1927   -0.5809   -0.3882
    0.5774    0.1741   -0.3693    0.5559   -0.3301   -0.2823    0.0478
         0    0.5222   -0.4308   -0.2879    0.5782   -0.3442    0.0776
         0         0   -0.6770   -0.2680   -0.2481    0.6265   -0.1254
         0   -0.5222   -0.2462    0.0198    0.1737   -0.0292    0.7970
R = 7×3
   -1.7321    0.5774    0.5774
         0   -1.9149    0.6963
         0         0   -1.4771
         0         0         0
         0         0         0
         0         0         0
         0         0         0
```

```
B = Q' * b % new righ side vector
```

```
B = 7×1
 -103.8676
 -246.0788
 -282.1006
    0.0344
    0.0083
   -0.0357
   -0.0050
```

```
opts.UT = true; % R is upper triangular
xQR = linsolve(R, B, opts) % solution using linsolve and the upper triangular options
```

```
xQR = 3×1
  189.6153
  197.9588
  190.9830
```

The heights of the unknown points are:

- $H_D = 189.6153\ \mathrm{m}$
- $H_E = 197.9588\ \mathrm{m}$
- $H_F = 190.9830\ \mathrm{m}$

```
norm(A*xQR - b) % the total error of the solution is ~5cm
```

```
ans = 0.0506
```

This means that the errors (random or otherwise) in our measurements only permit us to have a solution where the sum of the squared errors in the solution is around 5 cm. If we had to submit our results, we couldn't really vouch for a precision of millimeters, as far as the heights of the unknown points are considered.

Solution using SVD:

```
[U, S, V] = svd(A) % SVD of matrix A
```

```
U = 7×7
   -0.1494   -0.3536    0.5577    0.6243    0.2670   -0.0164   -0.2835
    0.5577    0.3536   -0.1494    0.0684    0.5971    0.2659   -0.3312
    0.4082   -0.0000    0.4082   -0.3761   -0.1927   -0.5809   -0.3882
    0.0000    0.7071   -0.0000    0.5559   -0.3301   -0.2823    0.0478
   -0.5577    0.3536    0.1494   -0.2879    0.5782   -0.3442    0.0776
   -0.1494    0.3536    0.5577   -0.2680   -0.2481    0.6265   -0.1254
    0.4082   -0.0000    0.4082    0.0198    0.1737   -0.0292    0.7970
S = 7×3
    2.1753         0         0
         0    2.0000         0
         0         0    1.1260
         0         0         0
         0         0         0
         0         0         0
         0         0         0
V = 3×3
   -0.3251   -0.7071    0.6280
    0.8881   -0.0000    0.4597
   -0.3251    0.7071    0.6280
```

```
invS = (1./S)' % inverse of S is the reciprocal of the entries and the transpose of the matrix
```

```
invS = 3×7
    0.4597       Inf       Inf       Inf       Inf       Inf       Inf
       Inf    0.5000       Inf       Inf       Inf       Inf       Inf
       Inf       Inf    0.8881       Inf       Inf       Inf       Inf
```

```
invS(invS == Inf) = 0 % change the Inf values inside the matrix to 0
```

```
invS = 3×7
    0.4597         0         0         0         0         0         0
         0    0.5000         0         0         0         0         0
         0         0    0.8881         0         0         0         0
```

```
invA = V * invS * U' % inverse of A using the SVD matrices
```

```
invA = 3×7
    0.4583   -0.2917    0.1667   -0.2500    0.0417    0.2083    0.1667
    0.1667    0.1667    0.3333   -0.0000   -0.1667    0.1667    0.3333
    0.2083   -0.0417    0.1667    0.2500    0.2917    0.4583    0.1667
```

```
xSVD = invA * b % solution vector
```

```
xSVD = 3×1
  189.6153
  197.9588
  190.9830
```

```
norm(A * xSVD - b) % sum of the squares of the error
```

```
ans = 0.0506
```

The solution and the error are exactly the same, the difference can be found in the computational time. If we had to solve the problem for example a 1000 times, the QR factorization would be reasonably faster.

Solution using built-in functions

Let's solve the system 10.000 times using the \ operator and the SVD and time the results:

```
% the 'tic' keyword starts the timer, the 'toc' keyword ends it
% inside the for loop, the system is solved 1000 times using the \ operator
tic
for i = 1:10000
    x1 = A\b;
end
toc
```

```
Elapsed time is 0.045602 seconds.
```

Solution using the SVD:

```
tic
for i = 1:10000
    x2 = pinv(A)*b;
end
toc
```

```
Elapsed time is 0.213193 seconds.
```

We can see that the time it takes to find the solution using `pinv` is significantly longer (and gets even worse with larger matrices).

Checking the solutions:

```
norm(A*x1 - b)
```

```
ans = 0.0506
```

```
norm(A*x2 - b)
```

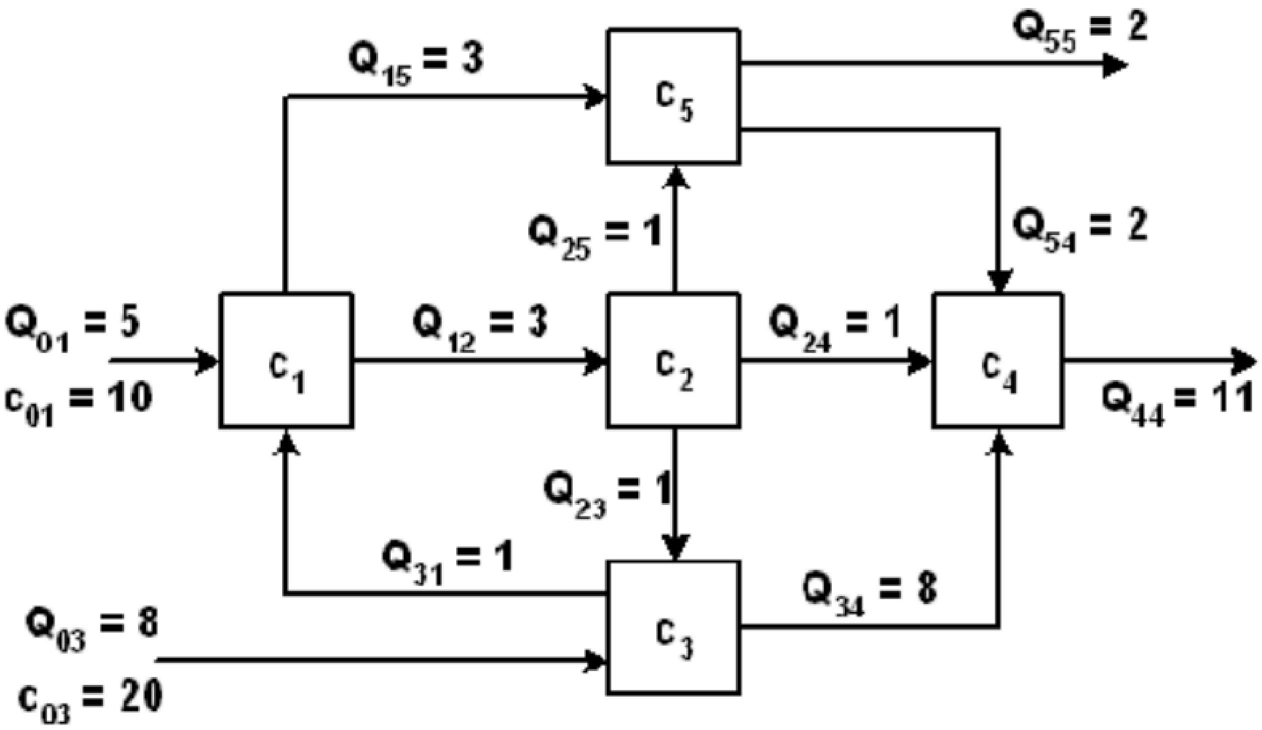
```
ans = 0.0506
```

Both solutions give the same errors of course.

Iterative methods of solving systems of linear equations

Apart from direct solutions, we can also employ iterative methods to solve systems of linear equations. In many cases, especially with sparse coefficient matrices, these solutions can be significantly more efficient. Let's see an example for these types of solutions:

We have a system of 5 reactors and we want to determine the amount of concentration in each reactor. The figure of the connection of the reactors is the following:



As a simplification, the rate of amalgamation in the reactors is considered perfect (the concentration is perfectly equal in every part of the reactor). We have volumetric flow (Q) into and out of every reactor and as the amalgamation is perfect, we can use the fact that the sum of the mass flow (concentration times volumetric flow) into a reactor and out of a reactor is zero. We will use the convention that the outflow is positive. Consider the first reactor for example on the left side: We have 5 units of inflow with concentration of 10 units from an outside source and 1 unit of inflow from reactor 3 with concentration c_3. We also have an outflow of 3 units with concentration c_1 to reactor two and an outflow of 3 units with the same concentration c_1 to reactor 5. The sum of the incoming and outgoing mass flow is zero (with the outgoing being positive as convention now):

$$3 \cdot c_1 + 3 \cdot c_1 - 1 \cdot c_3 - 5 \cdot 10 = 0 \rightarrow 6c_1 - c_3 = 50$$

Similarly, the whole system can be written as the following:

$$\begin{aligned} &6c_1 - c_3 = 50 \\ &-3c_1 + 3c_2 = 0 \\ &-c_2 + 9c_3 = 160 \\ &-c_2 - 8c_3 + 11c_4 - 2c_5 = 0 \\ &-3c_1 - c_2 + 4c_5 = 0 \end{aligned}$$

In matrix form:

$$\boldsymbol{A} = \begin{bmatrix} 6 & 0 & -1 & 0 & 0 \\ -3 & 3 & 0 & 0 & 0 \\ 0 & -1 & 9 & 0 & 0 \\ 0 & -1 & -8 & 11 & -2 \\ -3 & -1 & 0 & 0 & 4 \end{bmatrix} \quad \boldsymbol{c} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} \quad \boldsymbol{b} = \begin{bmatrix} 50 \\ 0 \\ 160 \\ 0 \\ 0 \end{bmatrix}$$

$$\boldsymbol{A} \cdot \boldsymbol{c} = \boldsymbol{b}$$

The coefficient matrix $\boldsymbol{A}$ contains many zero entries and therefore can be considered a sparse matrix. Iterative methods are typically used when the coefficient matrix is sparse and the entries are mostly located along the diagonal and close to it. These type of matrices are called diagonally dominant.

Iterative methods work differently than direct methods as we first have to supply the algorithms with initial guesses for the unknown which values are then iteratively changed as the solutions start to converge. (If they converge.) As an example, let's see the following system:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{12}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned} \rightarrow \begin{aligned} x_1 &= \frac{b_1 - a_{12}x_2 - a_{13}x_3}{a_{11}} \\ x_2 &= \frac{b_2 - a_{21}x_1 - a_{23}x_3}{a_{22}} \\ x_3 &= \frac{b_3 - a_{31}x_1 - a_{32}x_2}{a_{33}} \end{aligned}$$

The initial guesses can be substituted into the right-hand side of the equations and we can calculate a new solution. These are then substituted in the second iteration and so on... We can stop when the changes in the solutions (the differences between two iterations) become adequately small (smaller than some tolerance). The formula of iteration in the example above would be the following:

$$x_i = \frac{1}{a_{\text{ii}}} \cdot \left(b_i - \left(\sum_{j=1, j \neq i}^{j=n} a_{\text{ij}} \cdot x_j \right) \right)$$

This is basically the equation of the Jacobi iterative method.

We take a look at two iterative methods:

- Jacobi method
- Gauss-Seidel method

The main difference between the two methods is that the Jacobi method changes the values of all the unknowns at the end of an iteration, while the Gauss-Seidel method changes the value of each unknown right after they are calculated and uses these new values inside the same iteration. This means, that in the second method, when we refresh the value of x_1, we use this new value to calculate the new $x_2, x_3, \cdots$ values in the same iteration. We can derive the iterative formulas using matrix notation starting from the well known form $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$.

Add $\boldsymbol{B} \cdot \boldsymbol{x}$ and subtract $\boldsymbol{B} \cdot \boldsymbol{x}$ from the left-hand side of the equation. Depending on the choice of matrix $\boldsymbol{B}$, we will have the different methods:

$$\boldsymbol{B} \cdot \boldsymbol{x} + \boldsymbol{A} \cdot \boldsymbol{x} - \boldsymbol{B} \cdot \boldsymbol{x} = \boldsymbol{B} \cdot \boldsymbol{x} + (\boldsymbol{A} - \boldsymbol{B}) \cdot \boldsymbol{x} = \boldsymbol{b}$$

Rearrange the equation:

$$\boldsymbol{B} \cdot \boldsymbol{x} = -(\boldsymbol{A} - \boldsymbol{B}) \cdot \boldsymbol{x} + \boldsymbol{b}$$

Multiply from the left side by $\boldsymbol{B}^{-1}$:

$$\boldsymbol{x} = -\boldsymbol{B}^{-1}(\boldsymbol{A} - \boldsymbol{B}) \cdot \boldsymbol{x} + \boldsymbol{B}^{-1} \cdot \boldsymbol{b} = \boldsymbol{B}^{-1} \cdot \boldsymbol{B} \cdot \boldsymbol{x} - \boldsymbol{B}^{-1} \cdot \boldsymbol{A} \cdot \boldsymbol{x} + \boldsymbol{B}^{-1} \cdot \boldsymbol{b}$$

Group the coefficients of $\boldsymbol{x}$ on the right-hand side:

$$\boldsymbol{x} = (\boldsymbol{I} - \boldsymbol{B}^{-1}\boldsymbol{A})\boldsymbol{x} + \boldsymbol{B}^{-1}\boldsymbol{b}$$

This gives us the formula to compute the k+1-th iteration of vector $\boldsymbol{x}$:

$$\boldsymbol{x}^{(k+1)} = (\boldsymbol{I} - \boldsymbol{B}^{-1}\boldsymbol{A})\boldsymbol{x}^{(k)} + \boldsymbol{B}^{-1}\boldsymbol{b}$$

Let $\boldsymbol{Ai}$ denote $(\boldsymbol{I} - \boldsymbol{B}^{-1}\boldsymbol{A})$ and $\boldsymbol{bi}$ denote $(\boldsymbol{B}^{-1}\boldsymbol{b})$, then we can write the equation in a more concise form:

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{Ai} \cdot \boldsymbol{x}^{(k)} + \boldsymbol{bi}$$

We can create a separate function which handles the iterative solution in the above form given some matrix $\boldsymbol{Ai}$, some vector $\boldsymbol{bi}$, a vector of initial guesses, a tolerance value for the stop condition and a value for the maximum number of iterations. Find this function in `iterative.m`.

Solution using the Jacobi method

In case of the Jacobi method, the matrix $\boldsymbol{B}$ contains the diagonal elements of matrix $\boldsymbol{A}$, which means that it is very simple to calculate the inverse of $\boldsymbol{B}$. We can use a vector of ones as the initial guess. First, let's load the data and check if the solution exists:

```
Ab = load('waterplant.txt')
```

```
Ab = 5×6
     6     0    -1     0     0    50
    -3     3     0     0     0     0
     0    -1     9     0     0   160
     0    -1    -8    11    -2     0
    -3    -1     0     0     4     0
```

```
A = Ab(:, 1:end-1)
```

```
A = 5×5
     6     0    -1     0     0
    -3     3     0     0     0
     0    -1     9     0     0
     0    -1    -8    11    -2
    -3    -1     0     0     4
```

```
b = Ab(:, end)
```

```
b = 5×1
    50
     0
   160
     0
     0
```

```
size(A)
```

```
ans = 1×2
     5     5
```

```
rankA = rank(A)
rankA = 5
```

```
rankAug = rank([A, b])
```

```
rankAug = 5
```

The solution exits and as the matrix $\boldsymbol{A}$ is full-rank, we have a unique solution. We first compute the matrix $\boldsymbol{B}$, which is now the diagonal elements of $\boldsymbol{A}$:

```
B = diag(diag(A)) % the first diag gets the diagonal elements from A as a vector, the second cr
```

```
B = 5×5
     6     0     0     0     0
     0     3     0     0     0
     0     0     9     0     0
     0     0     0    11     0
     0     0     0     0     4
```

```
x0 = ones(5, 1) % vector of initial guesses
```

```
x0 = 5×1
     1
     1
     1
     1
     1
```

```
Ai = eye(5) - B\A % new coefficient matrix
```

```
Ai = 5×5
         0         0    0.1667         0         0
    1.0000         0         0         0         0
         0    0.1111         0         0         0
         0    0.0909    0.7273         0    0.1818
    0.7500    0.2500         0         0         0
```

```
bi = B\b % new right-hand side vector
```

```
bi = 5×1
    8.3333
         0
   17.7778
         0
         0
```

```
X = iterative(Ai, bi, x0, 1e-9, 100)
```

```
X = 5×20
    1.0000   11.3148   11.3148   11.4537   11.5058   11.5058   11.5084   11.5094 ⋯
    1.0000    8.5000   11.3148   11.3148   11.4537   11.5058   11.5058   11.5084
    1.0000   17.8889   18.7222   19.0350   19.0350   19.0504   19.0562   19.0562
    1.0000   13.2828   14.9874   16.5741   16.9295   16.9610   16.9904   16.9970
    1.0000    6.6250   10.6111   11.3148   11.4190   11.4928   11.5058   11.5078
```

```
iters = size(X, 2)
```

```
iters = 20
```

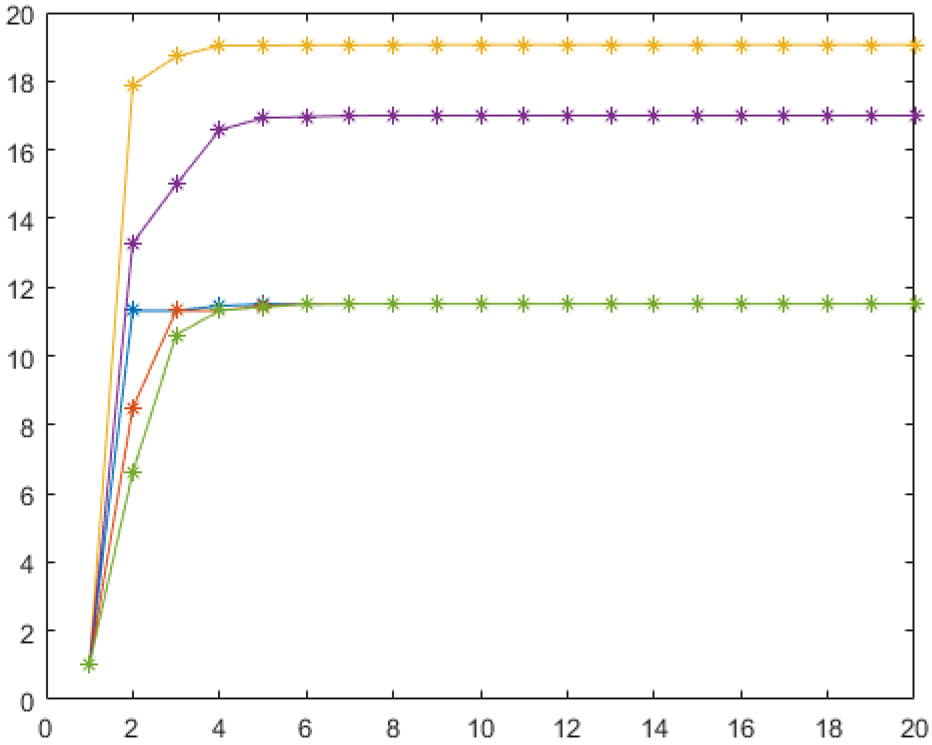
Seems like it took 20 iterations for the algorithm to reach the tolerance. Check the results:

```
norm(A*X(:,end) - b)
```

```
ans = 1.0922e-09
```

We can see, that the check only gives zero with the tolerance specified. Plot the convergence of the solutions:

```
figure(1)
plot(X', '*-') % the plot functions works by going through columns, so we have to transpose the
```



Solution using the Gauss-Seidel method

In order to solve the system using the Gauss-Seidel method, we only have to change the matrix $\boldsymbol{B}$. In this case, it becomes the lower triangular part of matrix $\boldsymbol{A}$.

```
B = tril(A) % lower triangular part of A
```

```
B = 5×5
     6     0     0     0     0
    -3     3     0     0     0
     0    -1     9     0     0
     0    -1    -8    11     0
    -3    -1     0     0     4
```

```
Ai = eye(5) - B\A % new coefficient matrix
```

```
Ai = 5×5
         0         0    0.1667         0         0
         0         0    0.1667         0         0
         0         0    0.0185         0         0
         0         0    0.0286         0    0.1818
         0         0    0.1667         0         0
```

```
bi = B\b % new right-hand side vector
```

```
bi = 5×1
    8.3333
    8.3333
   18.7037
   14.3603
    8.3333
```

```
X = iterative(Ai, bi, x0, 1e-9, 100)
```

```
X = 5×9
    1.0000   11.4537   11.5084   11.5094   11.5094   11.5094   11.5094   11.5094 ⋯
    1.0000   11.4537   11.5084   11.5094   11.5094   11.5094   11.5094   11.5094
    1.0000   19.0504   19.0565   19.0566   19.0566   19.0566   19.0566   19.0566
    1.0000   16.4415   16.9880   16.9981   16.9983   16.9983   16.9983   16.9983
    1.0000   11.4537   11.5084   11.5094   11.5094   11.5094   11.5094   11.5094
```

```
iters = size(X, 2) % number of iterations
```

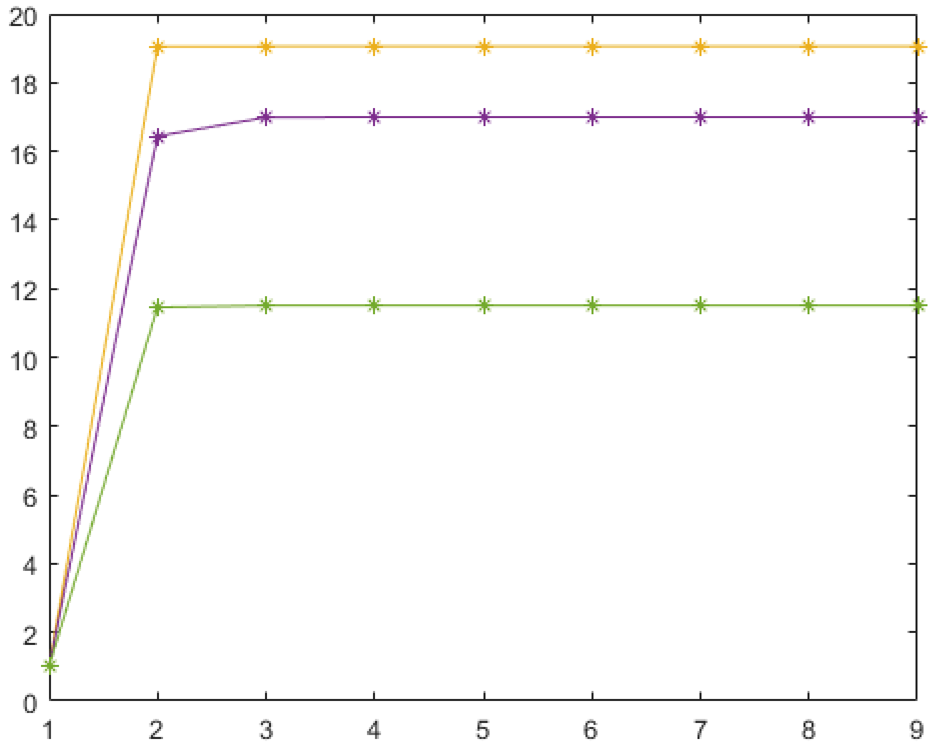
```
iters = 9
```

```
norm(A*X(:, end) - b)
```

```
ans = 4.3947e-12
```

Now, it only took 9 iterations to reach at least the desired tolerance. The convergence of the Gauss-Seidel method is significantly better than the Jacobi method. Plot the convergence:

```
figure(2)
plot(X', '*-')
```



We can see from the figure, that after 3 iterations, the changes to the solutions become very small. If we want to use the built-in function to solve the system iteratively, we can use the `gmres` command:

```
[x, flags, relres, iter, resvec] = gmres(A, b)
```

```
x = 5×1
   11.5094
   11.5094
   19.0566
   16.9983
   11.5094
flags = 0
relres = 4.5153e-17
iter = 1×2
     1     4
resvec = 5×1
  167.6305
  114.7905
   34.1596
   26.1821
    0.0000
```

```
norm(A*x - b)
```

```
ans = 5.0494e-14
```

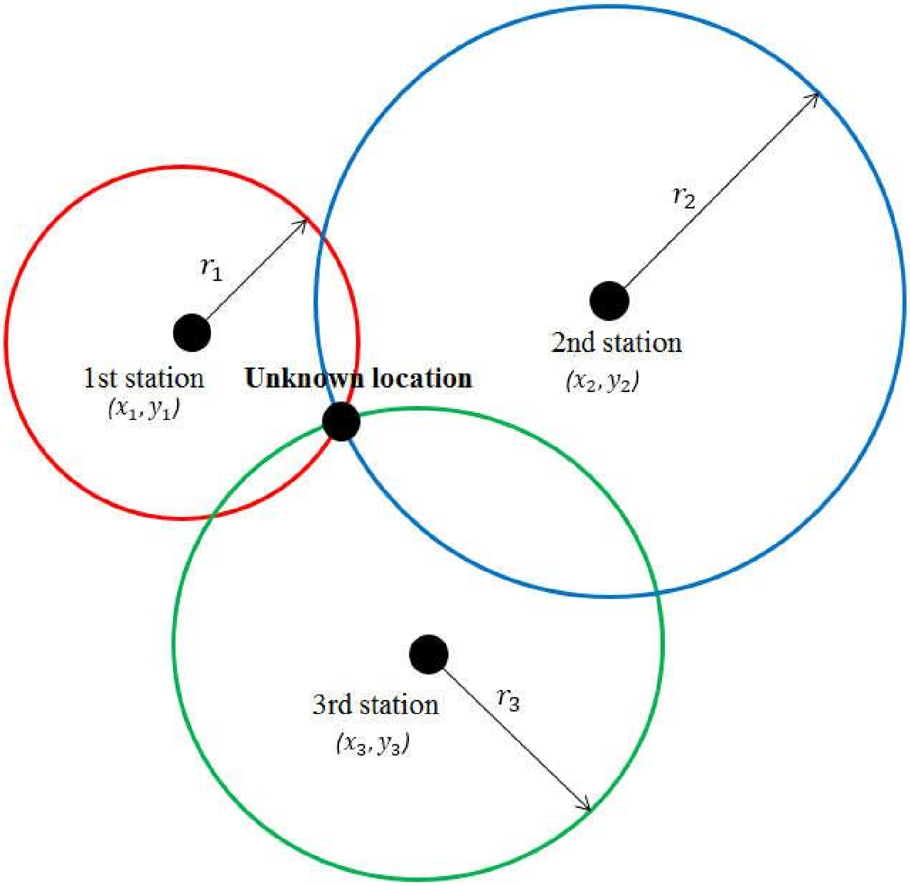
The `gmres` function can give us information about the solution method: `flags` store the exit code of the algorithm (if it is not zero, the algorithm couldn't reach an adequate solution), `relres` returns the residual norm, `iter` returns the number of outer and inner iterations and the `resvec` returns the amount of residual norms at each inner iteration.

If we are working with large sparse matrices, it is better to use the `sparse` function to efficiently store the matrix.

Nonlinear Systems of Equations

In civil engineering practice there are multiple problems that can be only be modeled by systems of nonlinear equations. For example in surveying, many positioning tasks are carried out by finding the intersection of curves that are defined by nonlinear functions of the unknown coordinate values, but other examples can be given from soil mechanics and so on.

Let's look at the following typical problem represented by a nonlinear system. We have 3 cell towers dealing with mobile phone traffic. If we can measure the distance of the device from each tower (using the signal strength for example) and we know the positions of the towers (their coordinates for example) then we can find the position of the device as the intersection of the three circles drawn around the cell towers. The radii of the circles represent the distance of the device from each of the towers. See the figure below.



The equations of the circles are given in the form of quadratic equations and therefore the three circles give us a nonlinear system where x and y denote the **unknown** coordinates of our mobile phone, x_1, x_2, x_3 and y_1, y_2, y_3 represent the **known** coordinates of the cell towers and r_1, r_2, r_3 are the distances of our device from the towers (the radii of the circles). One equation can be written as the following:

$$\sqrt{(x - x_1)^2 + (y - y_1)^2} = r_1$$

Squaring both sides and rearranging so that on the right side we have 0:

$$(x - x_1)^2 + (y - y_1)^2 - r_1 = 0$$

Our system is then the following:

$$(x - x_1)^2 + (y - y_1)^2 - r_1 = 0$$

$$(x - x_2)^2 + (y - y_2)^2 - r_2 = 0$$

$$(x - x_3)^2 + (y - y_3)^2 - r_3 = 0$$

If we first start with only two equations instead of three, we have to find the roots of a polynomial of the fourth degree, which we could solve by hand, but our aim is now to automate the task and harness the power of numerical algorithms to quickly and easily find all solutions.

Vector Notation of Nonlinear Systems

When solving nonlinear systems numerically, we have to use the vector notation to represent the systems. This is mostly due to the fact that the algorithms developed to efficiently solve these problems also use the vector notation as it makes it much simpler to work with the equations and the solutions as vectors instead of individual objects. This means that for example, instead of having each solution as $x_1, x_2, \cdots, x_n$, we have a solution vector $\boldsymbol{x}$, the entries of which are the individual solutions: $\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}$.

The general case is when we have as many equations as unknowns. The equations are given in the form:

$$\begin{aligned} f_1(x_1, x_2, \cdots, x_n) &= 0 \\ f_2(x_1, x_2, \cdots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \cdots, x_n) &= 0 \end{aligned}$$

We create a vector to represent the equations and a vector to represent the solutions:

$$\boldsymbol{f} = \begin{bmatrix} f_1 & f_2 & \cdots & f_n \end{bmatrix}^T$$

$$\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^T$$

Using vector notation, the system becomes (where 0 is the null vector):

$$\boldsymbol{f}(\boldsymbol{x}) = 0$$

When we were solving a single nonlinear equation, we had to supply the algorithm with a single initial guess. Now, as we have *n* equations, we have to give the algorithms a vector of initial guesses that contains the initial values for each of the *n* unknowns.

Visualizing the Mobile Positioning Problem

Let's first use only two cell towers and visualize the problem. The coordinates of the cell towers and their distances from the device (in kilometers) are the following:

1. $x_1 = 1 \quad y_1 = 1 \quad r_1 = 5$

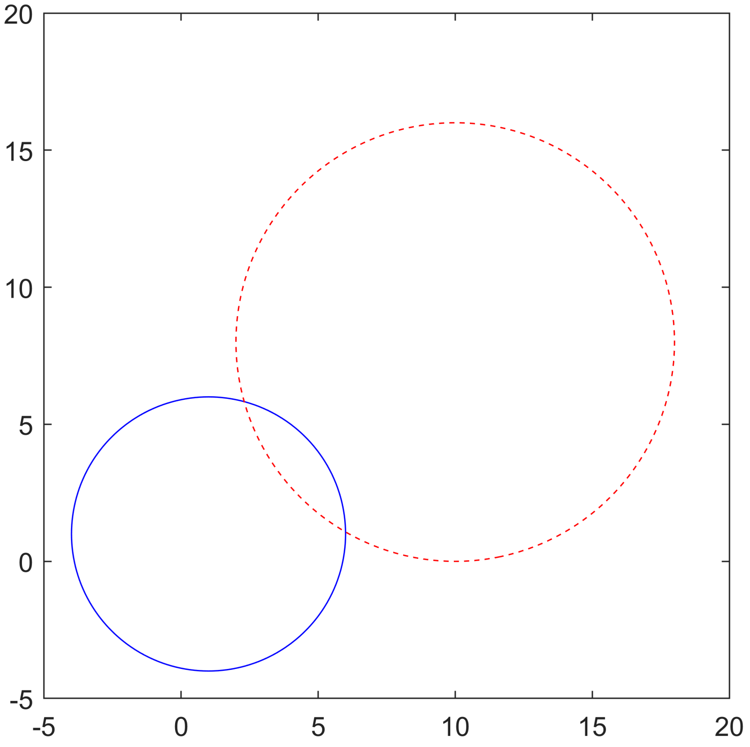
2. $x_2 = 10 \quad y_1 = 8 \quad r_2 = 8$

First, we define the two implicit equations as shown above, using anonymous functions:

```
clear all; close all;

f1 = @(x, y) (x-1).^2 + (y-1).^2 - 5^2;
f2 = @(x, y) (x-10).^2 + (y-8).^2 - 8^2;
```

We can use the MATLAB function `fimplicit` to plot the functions:

```
figure(1)
fimplicit(f1, [-5, 20, -5, 20], 'b-') % second argument is the range
hold on
fimplicit(f2, [-5, 20, -5, 20], 'r--')
axis equal % to make the axis have the same lengths
```



When we only had one equations, we could use the Newton-Raphson method for instance to iteratively find the solution. However, the Newton-Raphson method can also be generalized to solve systems of nonlinear equations.

Newton-Raphson Method for Solving Nonlinear Systems

In case of the Newton-Raphson method for a single equation, we used the initial guess and the derivative of the function to find the next iteration of the solution. As the solution got updated, we got closer and closer to 0 for the function value:

$$f(x) = 0 \approx f(x_0) + f'(x_0) \cdot (x - x_0)$$

where x_0 is the initial guess. Generalizing to a system of equations using the vector notation, we can write:

$$\boldsymbol{f}(\boldsymbol{x}) = 0 \approx \boldsymbol{f}(\boldsymbol{x}_0) + \boldsymbol{J}(\boldsymbol{x}_0) \cdot (\boldsymbol{x} - \boldsymbol{x}_0)$$

where $\boldsymbol{f}$ is the vector of equations, $\boldsymbol{x}_0$ is the vector of initial guesses and $\boldsymbol{J}$ is the Jacobian-matrix of the system. The Jacobian contains the partial derivatives of the equations with respect to each unknown at the values of the initial guesses. Each row corresponds to one equation:

$$\boldsymbol{J}(x_0) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(x_0) & \frac{\partial f_1}{\partial x_2}(x_0) & \cdots & \frac{\partial f_1}{\partial x_n}(x_0) \\ \frac{\partial f_2}{\partial x_1}(x_0) & \frac{\partial f_2}{\partial x_2}(x_0) & \cdots & \frac{\partial f_2}{\partial x_3}(x_0) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1}(x_0) & \frac{\partial f_n}{\partial x_2}(x_0) & \cdots & \frac{\partial f_n}{\partial x_n}(x_0) \end{bmatrix}$$

Let's see the following example:

$$f_1(x, y) = 3x^2 + 2y + 1 = 0$$
$$f_2(x, y) = -x^3 - 5y + 2 = 0$$

The Jacobian of the system contains the partial derivatives of the equations:

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial x} \end{bmatrix} = \begin{bmatrix} 6x & 2 \\ -3x^2 & -5 \end{bmatrix}$$

We want to solve the following system of equations:

$$\boldsymbol{f}(\boldsymbol{x}_{i+1}) = \boldsymbol{f}(\boldsymbol{x}_i) + \boldsymbol{J}(\boldsymbol{x}_i) \cdot (\boldsymbol{x}_{i+1} - \boldsymbol{x}_i) = 0$$

If we rearrange the system and solve for $\boldsymbol{x}_{i+1}$ to find the next iteration of the unknowns, we get:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \boldsymbol{J}^{-1}\boldsymbol{f}(\boldsymbol{x}_i)$$

We can solve this, if the Jacobian can be inverted (not singular). However, as we have seen in the case of linear systems, computing the inverse of a matrix directly is not very computationally efficient, we resort to another way of solving this. Use the $\Delta \boldsymbol{x} = \boldsymbol{x}_{i+1} - \boldsymbol{x}_i$ notation for the original equation:

$$\boldsymbol{f}(\boldsymbol{x}_i) + \boldsymbol{J}(\boldsymbol{x}_i) \cdot \Delta \boldsymbol{x} = 0 \rightarrow \boldsymbol{J}(\boldsymbol{x}_i) \cdot \Delta \boldsymbol{x} = -\boldsymbol{f}(\boldsymbol{x}_i)$$

This gives us a linear system in the form of $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$, which we can solve very efficiently using the methods mentioned in previous practicals. If we find $\Delta \boldsymbol{x}$, we can then find $\boldsymbol{x}_{i+1}$ as well:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \Delta \boldsymbol{x}$$

During each iteration, we solve the following two problems:

1. Calculation of the update of $\boldsymbol{x}_i$ from $\boldsymbol{J}(\boldsymbol{x}_i) \cdot \Delta \boldsymbol{x} = -\boldsymbol{f}(\boldsymbol{x}_i)$ (using QR decomposition for example).
2. Updating the value of $\boldsymbol{x}_i$ and calculating the new solutions: $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \Delta \boldsymbol{x}$.

The iterations continue until the function values become zero with some tolerance or if the solutions become close to each other with some tolerance.

Multivariate Newton-Raphson Method in MATLAB

We can create a custom function in MATLAB that solves a given system of nonlinear equations using the multivariate Newton-method:

```
function [x1, n] = newtonsys (f, J, x0, eps, nmax)
        dx = J(x0)\-f(x0);  % first iteration
        x1 = x0 + dx;       % first iteration
        n = 1;
        while norm(x1-x0)>eps && n<=nmax
            x0 = x1;
            dx = J(x0)\-f(x0);
            x1 = x0 + dx;
            n = n + 1;
        end
end
```

Solution using the custom function

Let's solve the system of two equations that we visualized using the custom function above (located in `newtonsys.m`). We first have to find the Jacobian of the system. We can use symbolic derivation to find it automatically:

First, we create two symbolic variables:

```
syms x1 x2
```

Then, we define our functions as symbolic functions using our new symbolic variables in order to be able to differentiate symbolically:

```
fs1 = f1(x1, x2)
```

fs1 = $(x_1 - 1)^2 + (x_2 - 1)^2 - 25$

```
fs2 = f2(x1, x2)
```

fs2 = $(x_2 - 8)^2 + (x_1 - 10)^2 - 64$

The Jacobian-matrix can be found using the `jacobian` function:

```
js = jacobian([fs1; fs2], [x1, x2]) % the first input is the vector of the equations, the secon
```

```
js =
```

$$\begin{pmatrix} 2\,x_1 - 2 & 2\,x_2 - 2 \\ 2\,x_1 - 20 & 2\,x_2 - 16 \end{pmatrix}$$

This gives us the Jacobian-matrix as a symbolic objects. We need to convert this into an anonymous function in order to be able to calculate with it:

```
J = matlabFunction(js)
```

```
J = function_handle with value:
    @(x1,x2)reshape([x1.*2.0-2.0,x1.*2.0-2.0e1,x2.*2.0-2.0,x2.*2.0-1.6e1],[2,2])
```

We can use this form, to calculate the value of the Jacobian at any (x_1, x_2) point, for example:

```
J(2, 3)
```

```
ans = 2×2
     2     4
   -16   -10
```

The final task before we can find the solutions is the vectorization of the equations and the Jacobian. This means that we want to have only one input variable, so instead of x_1, x_2 we only want to have $\mathbf{x}$ (a vector), the entries of which are x_1 and x_2:

```
f = @(x) [f1(x(1), x(2)); f2(x(1), x(2))] % we substitute the first and second entries of x int
```

```
f = function_handle with value:
    @(x)[f1(x(1),x(2));f2(x(1),x(2))]
```

```
J = @(x) J(x(1), x(2))
```

```
J = function_handle with value:
    @(x)J(x(1),x(2))
```

Now, we can use our custom function to find the solutions. First we define the initial guesses (using the figure above).

The first solution:

```
x01 = [2; 5];
[x1, i1] = newtonsys(f, J, x01, 1e-9, 100)
```

```
x1 = 2×1
    2.3005
```

```
        5.8279
    i1 = 5
```

The second solution:

```
x02 = [5; 0];
[x2, i2] = newtonsys(f, J, x02, 1e-9, 100)
```

```
    x2 = 2×1
        5.9995
        1.0721
    i2 = 5
```

Check the correctness of the solutions:

```
norm(f(x1))
```

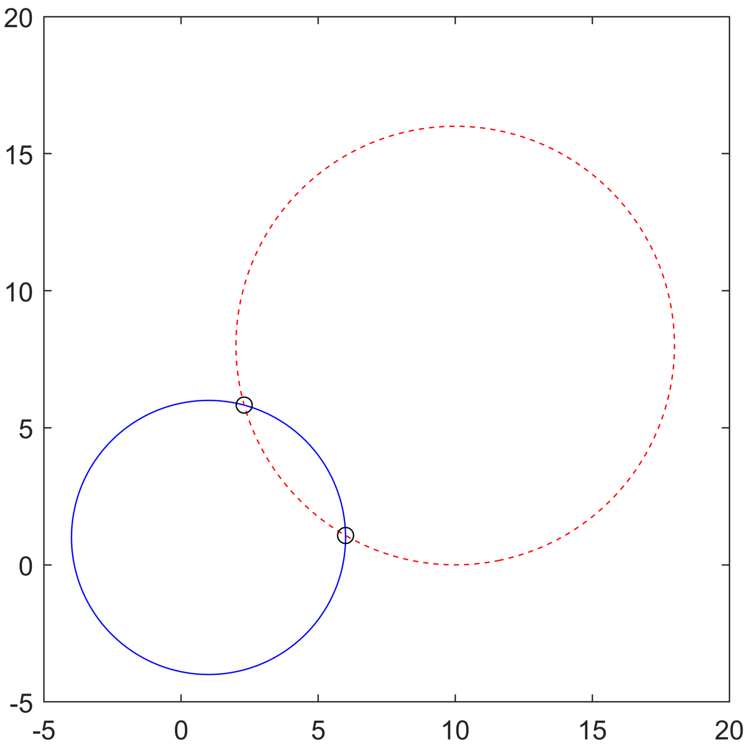
```
    ans = 1.4648e-14
```

```
norm(f(x2))
```

```
    ans = 3.5527e-15
```

Let's visualize the solutions:

```
plot(x1(1), x1(2), 'ko')
plot(x2(1), x2(2), 'ko')
```



What happens if we have $(1,1)$ (the coordinates of the first tower) as our initial guess?

```
x03 = [1; 1];
[x3, i3] = newtonsys(f, J, x03, 1e-9, 100)
```

```
Warning: Matrix is singular to working precision.
Warning: Matrix is singular, close to singular or badly scaled. Results may be inaccurate. RCOND = NaN.
x3 = 2×1
   NaN
   NaN
i3 = 2
```

The Jacobian becomes singular and solution doesn't converge. This illustrates that we have to choose our initial guesses carefully when solving systems of nonlinear equations.

Solution Using Built-in MATLAB method `fsolve`

The built-in `fsolve` method uses a combination of algorithms to find the optimal solution of the system. Similarly to the `fzero` function used for one equation, it can be parametrized to show each iteration step or to have certain tolerance values of the function values and the differences between each iteration of the unknowns. Similarly to the multivariate Newton-method, we have to run the function twice with different initial guesses to find both solutions. Supplying the Jacobian is not necessary however.

First, we give the solver some options to show us the iterations and also specify the tolerance for the function values:

```
opts = optimset('TolFun', 1e-9, 'Display', 'iter');
x1 = fsolve(f, x01, opts)
```

```
                                          Norm of      First-order   Trust-region
 Iteration  Func-count     f(x)           step         optimality    radius
     0          3             145                           160               1
     1          6         1.77485       0.970584           12.4               1
     2          9     0.000981063       0.148822          0.284            2.43
     3         12     3.63457e-10     0.00367156       0.000173            2.43
     4         15     5.56836e-23    2.23746e-06       7.13e-11            2.43

Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the selected value of the function tolerance, and
the problem appears regular as measured by the gradient.

<stopping criteria details>
x1 = 2×1
    2.3005
    5.8279
```

```
x2 = fsolve(f, x02, opts)
```

```
                                          Norm of      First-order   Trust-region
 Iteration  Func-count     f(x)           step         optimality    radius
     0          3             689                           384               1
     1           6         65.1904              1            104                1
     2           9        0.100068       0.472951           3.09              2.5
     3          12     3.61907e-06      0.0366768         0.0184              2.5
     4          15     4.94207e-15    0.000223254        6.8e-07              2.5
     5          18               0    8.25795e-09              0              2.5

Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the selected value of the function tolerance, and
the problem appears regular as measured by the gradient.

<stopping criteria details>
x2 = 2×1
    5.9995
    1.0721
```

The solutions are the same of course.

The method `fsolve` can be used to solve other types of nonlinear systems as well that contain trigonometric, exponential and other kinds of expression. It can also be used to solve a single equation, however, in that case `fzero` is much more efficient.

Symbolic Solution Using the Built-in `solve` method

If our problem can be formulated as an algebraic polynomial, we can also use the `solve` method. The advantages are that we don't have to given an initial guess to the algorithm and we can get back every solution at once. The downside is that we get the solution in algebraic form so we have convert them into numeric values to be able to work with them afterwards.

To use `solve`, we have to specify the equations in symbolic form. We have already done this for the computation of the Jacobian:

```
xs = solve([fs1, fs2])
```

```
xs = struct with fields:
    x1: [2×1 sym]
    x2: [2×1 sym]
```

Symbolic values of the solutions:

```
xs.x1
```

```
ans =
```

$$\begin{pmatrix} \frac{77\sqrt{39}}{260}+\frac{83}{20} \\ \frac{83}{20}-\frac{77\sqrt{39}}{260} \end{pmatrix}$$

```
xs.x2
```

```
ans =
```

$$\begin{pmatrix} \frac{69}{20} - \frac{99\,\sqrt{39}}{260} \\ \frac{99\,\sqrt{39}}{260} + \frac{69}{20} \end{pmatrix}$$

If we convert them into double precision floating point values, we get the same values as with the previously demonstrated methods:

```
xs = [double(xs.x1), double(xs.x2)]
```

```
xs = 2×2
    5.9995    1.0721
    2.3005    5.8279
```

Additional Example

Solution using `fsolve`

Let's practice the solution of nonlinear systems by solving the following two equations:

$$1.2 \cdot \sin(x) \cdot y = 1$$

$$0.8 \cdot \sin(y) \cdot x = 1$$

First, let's define and plot the two functions to get an idea of the solutions:

```
clear all; close all;
f1 = @(x, y) 1.2*sin(x).*y - 1;
f2 = @(x, y) 0.8*sin(y).*x - 1;
fimplicit(f1, [0, 6, 0, 6])
hold on;
fimplicit(f2, [0, 6, 0, 6], 'r')
```

Using the figure, we can find the initial guesses, let the first one be (2, 1) and the second one be (3, 2.5):

```
x01 = [2; 1];
x02 = [3; 2.5];
```

We have to vectorize each function and represent the system as a vector of these functions:

```
f = @(x) [f1(x(1), x(2)); f2(x(1), x(2))];
```

First solution:

```
opts = optimset('TolFun', 1e-9, 'Display', 'iter');
x1 = fsolve(f, x01, opts)
```

```
                                        Norm of      First-order   Trust-region
 Iteration  Func-count     f(x)          step         optimality    radius
```

```
     0          3         0.12827                            0.399               1
     1          6      0.00306652       0.325813          0.0658               1
     2          9     6.53151e-06      0.0742465         0.00366               1
     3         12     1.29533e-12     0.00193483        1.33e-06               1
     4         15      1.3807e-24    1.54508e-06         1.7e-12               1

Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the selected value of the function tolerance, and
the problem appears regular as measured by the gradient.

<stopping criteria details>
x1 = 2×1
    1.6810
    0.8384
```

```
x2 = fsolve(f, x02, opts)
```

```
                                        Norm of      First-order   Trust-region
 Iteration  Func-count     f(x)          step         optimality    radius
     0          3          0.5229                           1.92               1
     1          6     0.000820245        0.25809          0.0869               1
     2          9     5.53249e-09      0.0099228        0.000215               1
     3         12      5.9541e-19    2.84676e-05        2.02e-09               1
     4         15     9.86076e-32    3.26353e-10        7.58e-16               1

Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the selected value of the function tolerance, and
the problem appears regular as measured by the gradient.

<stopping criteria details>
x2 = 2×1
    2.8258
    2.6834
```

Correctness of the solution and plotting them:

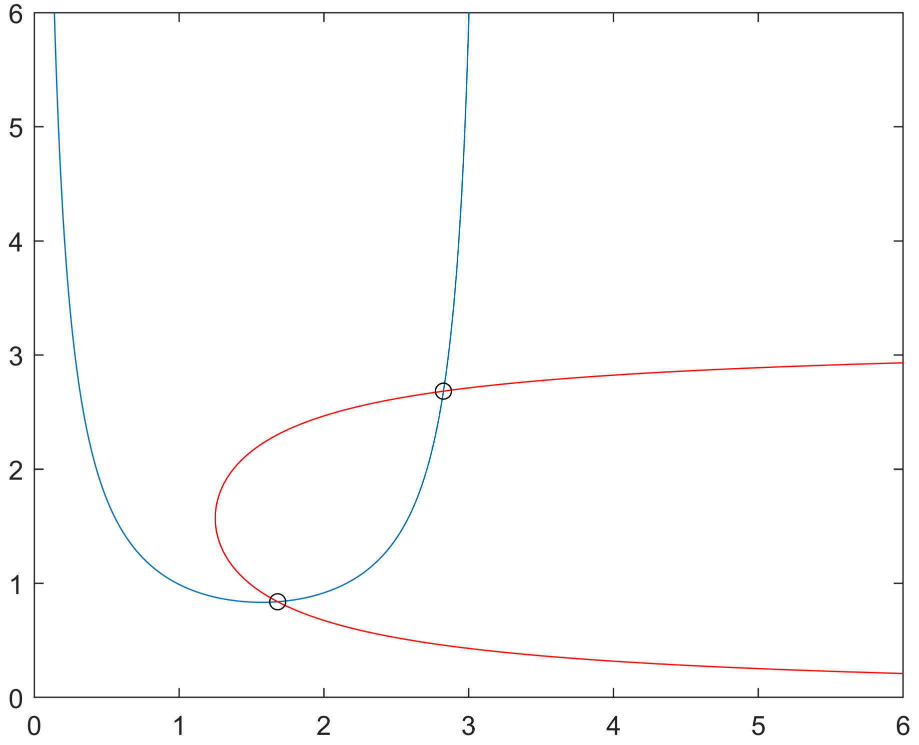
```
norm(f(x1))
```

```
ans = 1.1750e-12
```

```
norm(f(x2))
```

```
ans = 3.1402e-16
```

```
plot(x1(1), x1(2), 'ko')
plot(x2(1), x2(2), 'ko')
```



Using the custom `newtonsys.m` function

We first define the symbolic variables and the symbolic functions:

```
syms x1 x2
xs = [x1, x2]
```

xs = $\begin{pmatrix} x_1 & x_2 \end{pmatrix}$

```
fs = f(xs)
```

fs =

$$\begin{pmatrix} \frac{6\,x_2\,\sin(x_1)}{5} - 1 \\ \frac{4\,x_1\,\sin(x_2)}{5} - 1 \end{pmatrix}$$

Calculating the Jacobian using symbolic derivation and convert it to an anonymous function:

```
js = jacobian(fs)
```

js =

$$\begin{pmatrix} \frac{6\,x_2\cos(x_1)}{5} & \frac{6\sin(x_1)}{5} \\ \frac{4\sin(x_2)}{5} & \frac{4\,x_1\cos(x_2)}{5} \end{pmatrix}$$

```
J1 = matlabFunction(js);
```

Vectorizing the Jacobian:

```
J = @(x) J1(x(1), x(2));
```

Computing the solutions using the custom function:

```
[x1, i1] = newtonsys(f, J, x01, 1e-9, 100)
```

```
x1 = 2×1
    1.6810
    0.8384
i1 = 5
```

```
[x2, i2] = newtonsys(f, J, x02, 1e-9, 100)
```

```
x2 = 2×1
    2.8258
    2.6834
i2 = 4
```

Correctness of the solutions:

```
norm(f(x1))
```

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

```
norm(f(x2))
```

```
ans = 3.1402e-16
```

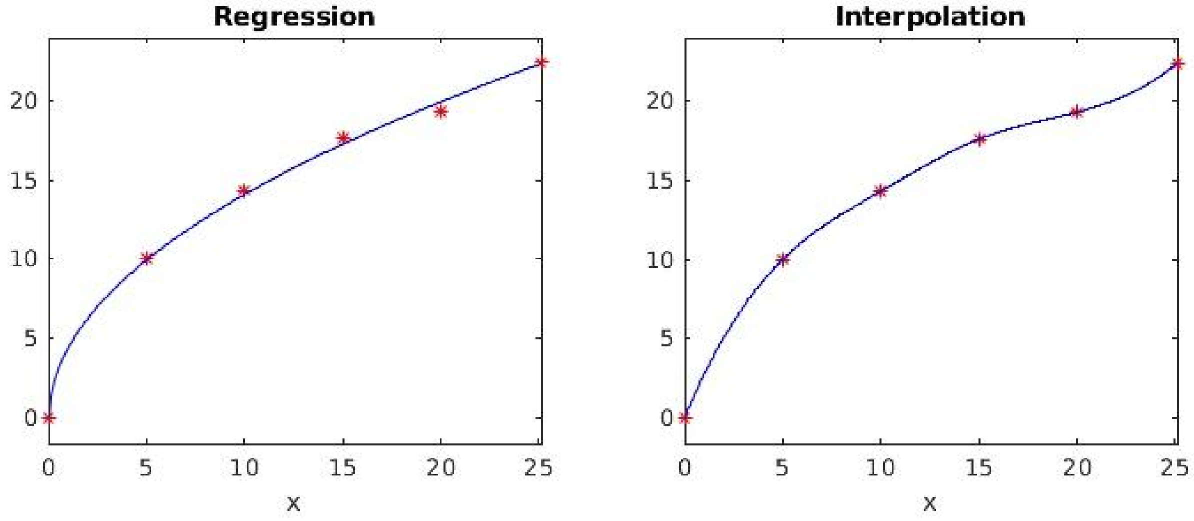
Regression

Regression and interpolation

In engineering practice, countless physical phenomena are measured and examined using digital instruments and the measurements values are stored discreet values. A typical civil engineering example could be the tensile test, when a certain material or structural element is tested to substantiate its tensile strength.

In the case of regression, we already have knowledge about the function (or model) of the physical phenomenon examined and we would like to determine the parameters of the function from our measured values. For example, if the phenomenon can be modelled using a quadratic function in the form $ax^2 + bx + c$, we can use the measured values to determine the parameters a, b and c. Usually, we have more measurements then parameters, so these problems lead to overdetermined systems which are also many times linear. The overdetermined nature also means that our resulting function will not go through any of the measurement points, but it will try to get as close to all of them as possible.

As for interpolation, we would like to have values where we have no measurements. This can mean values between measured points (interpolation) or values beyond our measurement interval (extrapolation). In these cases, we are trying to find a function that has the measured values as its function value at the measured points and adequately describes the behaviour between the points.



Goodness of fit parameters

During regression, we are trying to find the parameters of the function so that our function has the best fit to the measurement points. How can we determine the goodness of fit of a function? The most typical solution is to use the sum of the residuals, that is, the sum of the differences between the measurement values and the function values at each measured point: $r_i = y_i - f(x_i),\ i = 1 \cdots n$, where y_i is the measurement value at x_i and $f(x_i)$ is the function value of the resulting regression model.

However, only summing the residuals might not work well in all cases. If we have similar positive and negative residuals, they will cancel out in the summation which will result in a very high goodness of fit, but falsely. We could instead use the absolute value of the residuals, but this would result in an ambiguous solution, that is, more than one parameter set of a certain model would have the same best fit. The best solution is to use the sum of the squares of the residuals, as this way the solution will be unique (only parameter set of a certain model can have lead to the best fit to the data).

Local goodness of fit can therefore be measured using the residuals, while globally we can use the standard deviation of the residuals:

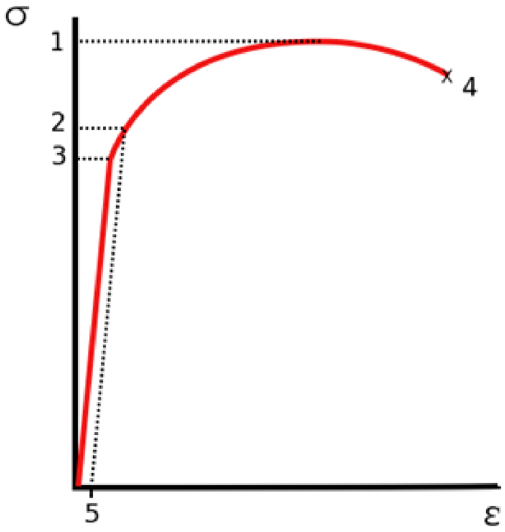
$$\sigma = \sqrt{\frac{1}{n-n_p} \cdot \sum_{i=1}^{n} (y_i - f(x_i))^2}$$

where n is the number of measurements and n_p is the number of parameters in the regression model.

Sometimes, the $f = n - n_p$ is also called the number of surplus measurements (the number of measurements beyond what is absolutely needed to solve the problem).

Regression line

Let's take a look at an example from structural engineering, the tensile test. In the example, we will use the data from the tensile test of a steel rod. In the tensile test, stress on the material is increased until it reaches its maximum, after which the stress declines and in the end the material ruptures. On the figure below, which shows the stress-strain diagram from a tensile test, we can identify certain points:



1. shows the ultimate tensile strength of the material (the maximum of the stress values).

2. shows the yield strength, before which the deformations are completely recoverable.

3. shows the proportional limit stress, up to which the stress-strain diagram is basically linear.

4. shows the point where the material ruptures.

5. shows the offset strain which is typically set as 0.2%)

Our task will be to determine the following:

1. the ultimate tensile strength of the material,

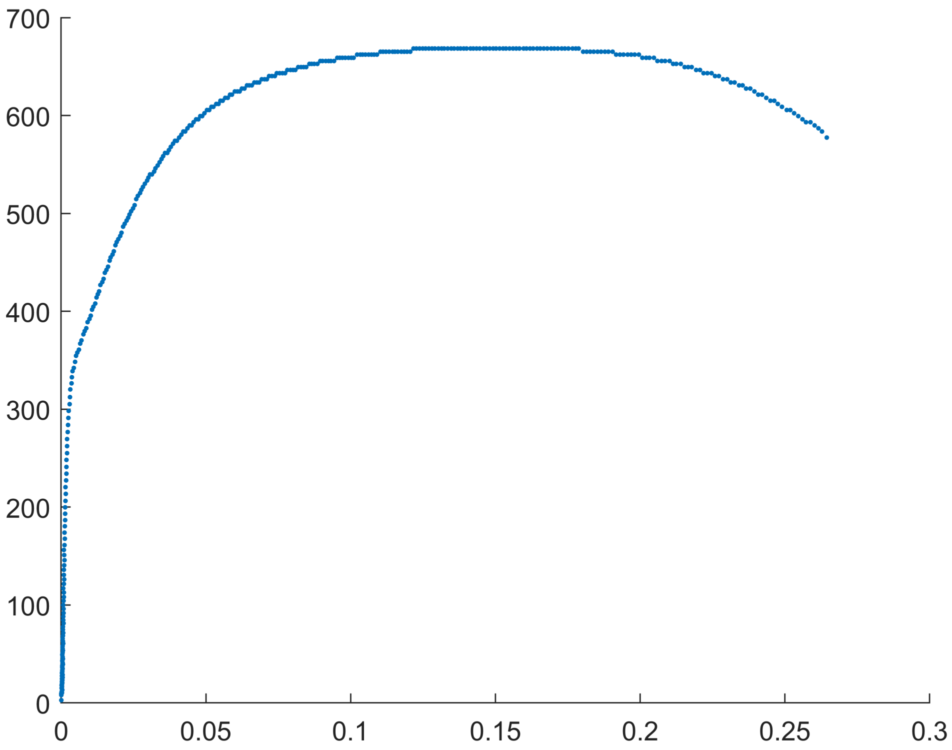
2. the strain of the material at the point of rupture,
3. the Young's modulus of the material from the linear part of the diagram,
4. to determine the stress corresponding to the offset strain.

The data is located in the `tensiletest.txt` file, so first, we have to load it and separate the stress and the strain values into variables. The stress values are in MPa, the strain values are in %.

```
clear all; close all
data = load('tensiletest.txt');
x = data(:, 1);
y = data(:, 2);
```

Plot of the data points:

```
fig1 = figure(); hold on;
plot(x, y, '.')
```



The answers to the first and second questions are easily found. The ultimate tensile strength of the material is the maximum of the stress values, while the strain at the point of rupture is the maximum of the strain values:

```
uts = max(y)
```

```
uts = 
                   668.3606
```

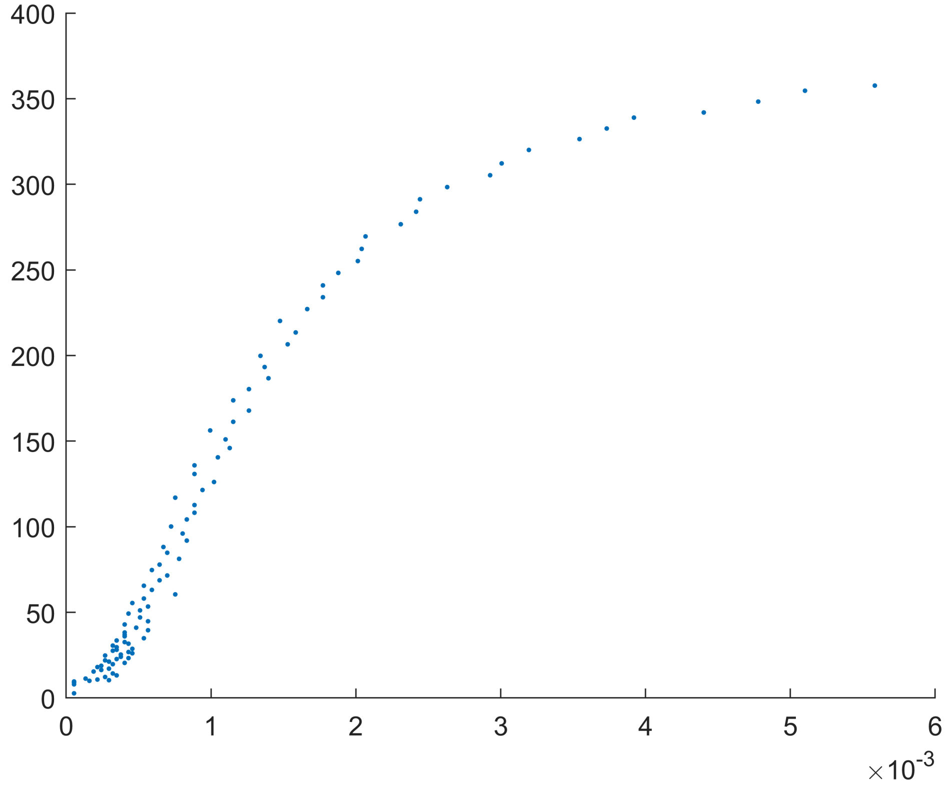
```
max_strain = max(x) % or x(end), the last strain value
```

```
max_strain = 
        0.2644
```

The ultimate tensile strength is around 668 MPa and the maximum strain is around 26%.

In order to determine the Young's modulus, we first have to find the measurement points corresponding to the linear part of the diagram. Then, we can use those points to fit a regression line, the slope of which will tell us the value of the modulus. To better see the points at the linear part of the diagram, we can zoom in (either interactively or by changing the axes limits):

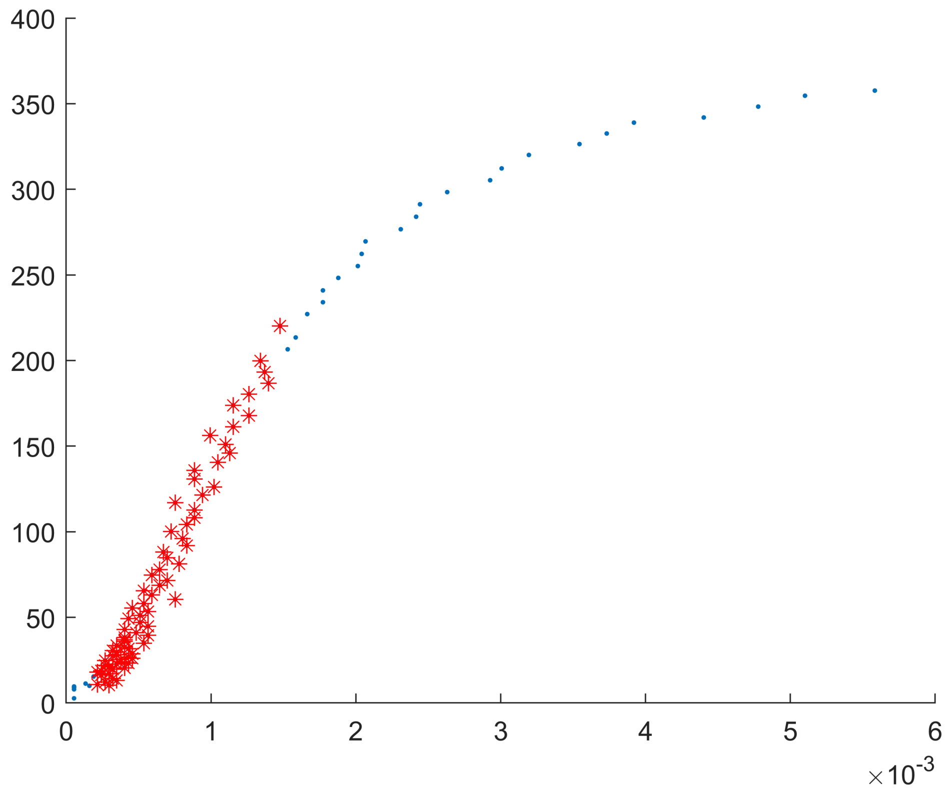
```
axis([0, 0.006, 0, 400]);
```



From the figure above, we can see that at the beginning of the test the diagram is not linear due to the inaccuracy of the instrument used. Therefore, these measurements have to be discarded before the line fitting. We also have to find the top of the linear part. We can use the data cursor to interactively find the x-values for the beginning and the end of the linear part (0.02% and 0.15% can do for now).

First, we create a logical array that contains our filter:

```
filter = x > 0.0002 & x < 0.0015; % && - logical AND
xf = x(filter);
yf = y(filter);
plot(xf, yf, 'r*')
```



We can check if our assumption hold and the filtered points are truly located along a line (that is, the connection between them is linear). To do this, we have to calculate the linear correlation coefficient:

$$r = \frac{\sum_{i=1}^{n} (x_i - \overline{x}) \cdot (y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \overline{x})^2 \sum_{i=1}^{n} (y_i - \overline{y})^2}}$$

In MATLAB, we can use the **`corr2`** command:

```
r = corr2(xf, yf)
```

```
r = 
        0.980487015777272
```

The closer the value of the correlation coefficient to 1 the more linear the connection is between the values. We got 0.98, so we can assume linearity and fit a line to the data. The equation of a line is the following:

$$y = a \cdot x + b$$

where a is the slope of the line and b is the y-intercept.

We filtered 75 points, so we have 75 line equations from which we can find the same a and b parameters:

$$\begin{aligned} y_1 &= a \cdot x_1 + b \\ y_2 &= a \cdot x_2 + b \\ &\vdots \\ y_{75} &= a \cdot x_{75} + b \end{aligned} \quad \text{or in matrix form: } A = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_{75} & 1 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{75} \end{bmatrix}$$

This is an overdetermined linear system (with more equations than unknowns) and we would like to solve it, to minimize the sum of the residual squares. We can use the QR decomposition or the SVD.

First, we create the coefficient matrix:

```
A = [xf, ones(length(xf), 1)] % first column is x, second column is just ones
```

```
A = 75×2
                  0.000295                          1
                  0.000215                          1
                  0.000268                          1
                  0.000349                          1
                  0.000322                          1
                  0.000242                          1
                  0.000295                          1
                  0.000215                          1
                  0.000242                          1
                  0.000322                          1
     ⋮
```

We get the parameters of the line by solving the linear system:

```
line = A\yf;
a = line(1)
```

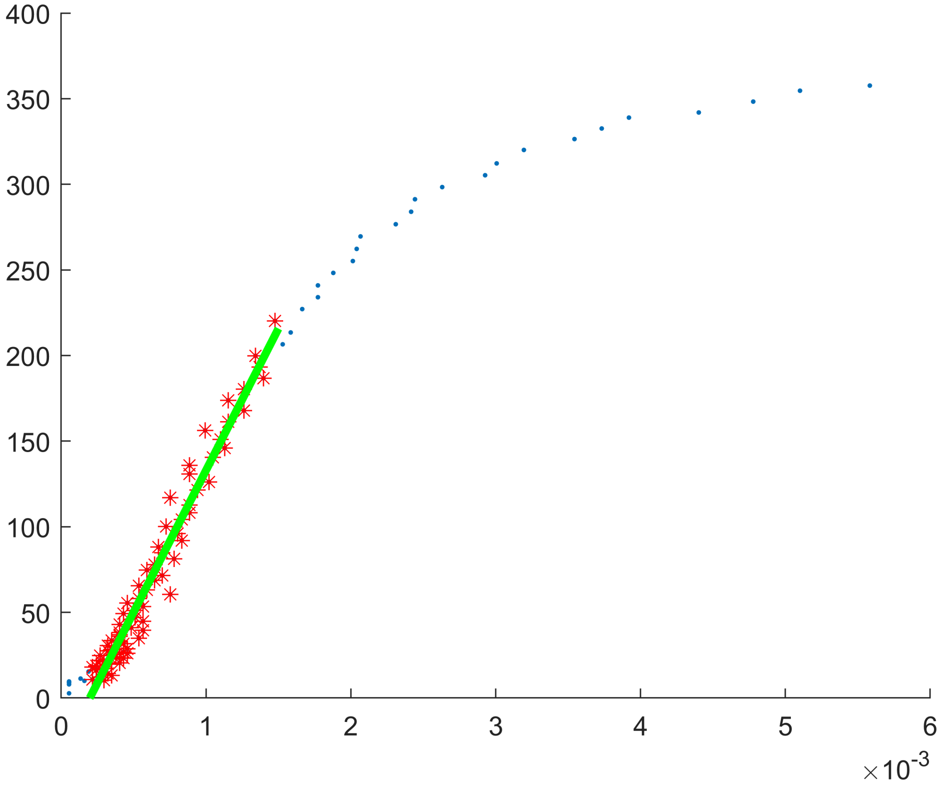
```
a = 
          165626.174495478
```

```
b = line(2)
```

```
b = 
         -32.6778764824923
```

Create the function for the line and plot the results into the figure:

```
l = @(x) a*x + b;
fplot(l, [0, 0.0015], 'g', 'LineWidth', 3)
```



The value of the Young's modulus is the slope of the regession line:

```
fprintf('Young''s modulus: %.5f MPa\n', a)
```

```
Young's modulus: 165626.17450 MPa
```

Regression polynomial

Determining the stress value of the offset strain cannot be adequately done using the diagram itself as the material doesn't have a well defined yielding stress. In these cases, we can use the typical value of the offset strain (0.2%), draw a parallel line with the linear part of the diagram starting at 0.2% strain and find out where it intersects with the measurements.

It would be best to fit a curve to the data after the linear part and find the intersection of the curve and the line parallel to the linear part of the diagram. In order to do this, let us now filter the data points that are located between 0.15% and 0.60%:

```
filter2 = x > 0.0015 & x < 0.006;
xf2 = x(filter2);
yf2 = y(filter2);
length(xf2)
```

```
ans = 
    23
```

We have filtered 23 points. Let's fit a parabola to the data points in the form of: $y = c_0 + c_1x + c_2x^2$.

$$\begin{aligned} y_1 &= c_0 + c_1x_1 + c_2x_1^2 \\ y_2 &= c_0 + c_1x_2 + c_2x_2^2 \\ &\vdots \\ y_{23} &= c_0 + c_1x_{23} + c_2x_{23}^2 \end{aligned} \qquad \text{in matrix form: } \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_{23} & x_{23}^2 \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{23} \end{bmatrix}$$

Again, we first create the coefficient matrix:

```
A = [ones(length(xf2), 1), xf2, xf2.^2]
```

```
A = 23×3
                         1               0.00153             2.3409e-06
                         1              0.001584           2.509056e-06
                         1              0.001664           2.768896e-06
                         1              0.001772           3.139984e-06
                         1              0.001772           3.139984e-06
                         1              0.001879           3.530641e-06
                         1              0.002013           4.052169e-06
                         1               0.00204             4.1616e-06
                         1              0.002067           4.272489e-06
                         1              0.002309           5.331481e-06
      ⋮
```

Solve the linear system:

```
c = A\yf2
```

```
c = 3×1
          69.3456670880163
          115829.75939882
         -11790386.3230783
```

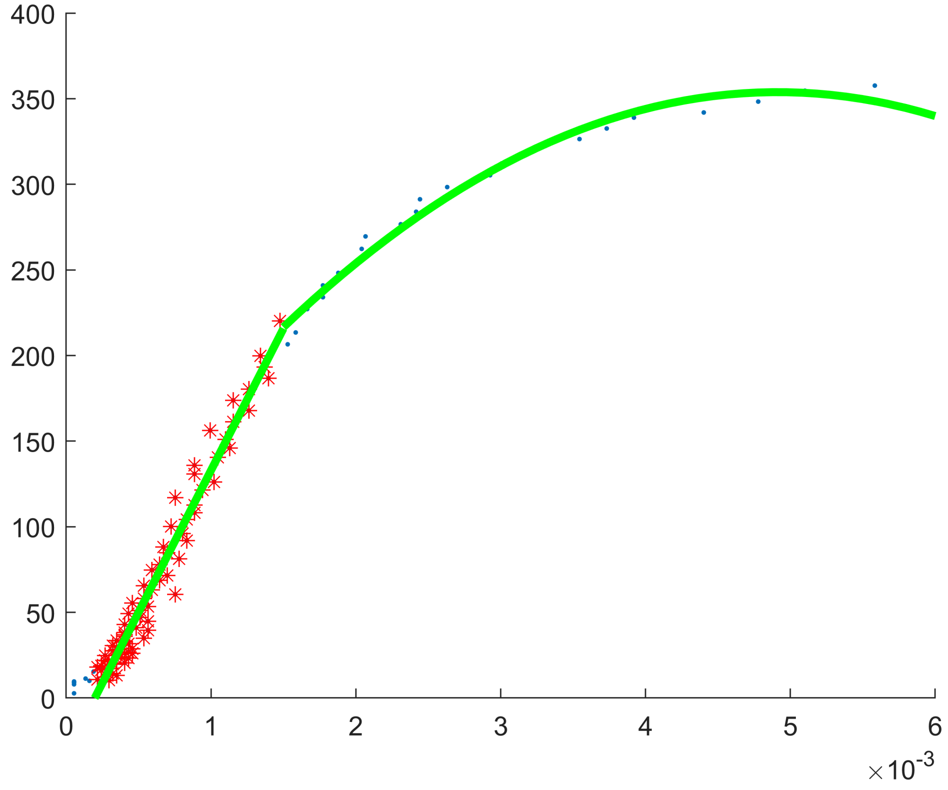
The equation of the parabola:

```
par = @(x) c(1) + c(2)*x + c(3)*x.^2
```

```
par = function_handle with value:
    @(x)c(1)+c(2)*x+c(3)*x.^2
```

Plotting the parabola into the figure:

```
fplot(par, [0.0015, 0.006], 'g', 'LineWidth', 3)
```

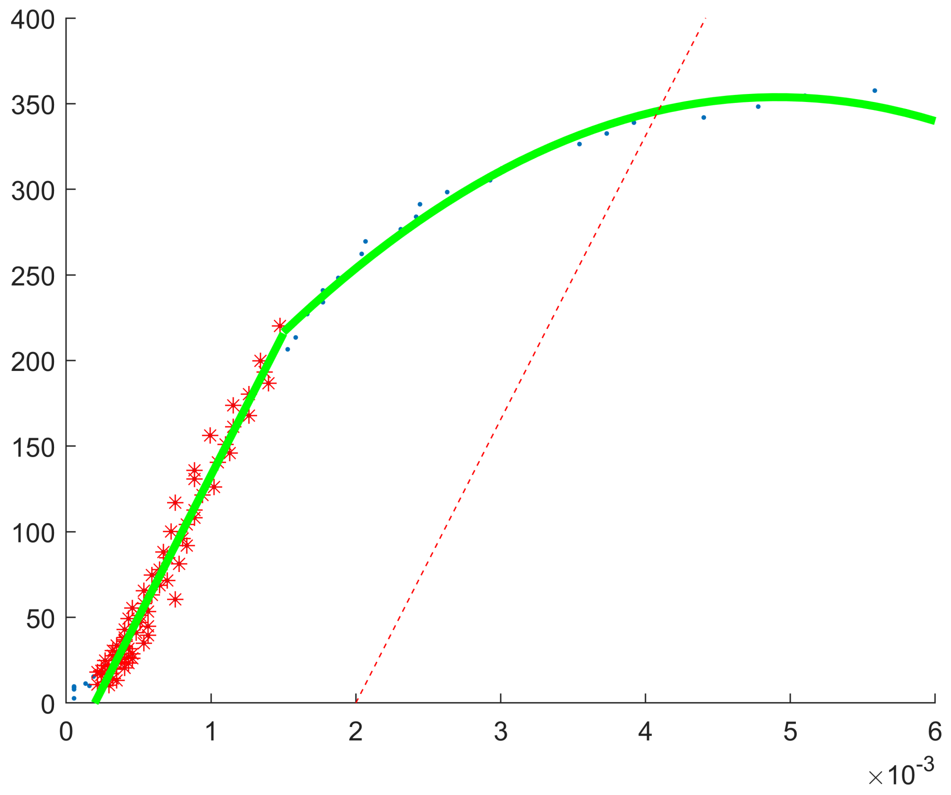


Now, we have to define the function of the line that starts at 0.2% strain and is parallel to the linear part of the diagram. This basically means that we have to create a new line with the same slope as the previous one starting at $x = 0.002$.

```
l2 = @(x) a*(x - 0.002)
```

```
l2 = function_handle with value:
    @(x)a*(x-0.002)
```

```
fplot(l2, [0.002, 0.006], 'r--')
```



To find the answer to our initial question, the stress corresponding to the offset strain, we only have to find the intersection of the green parabola and the red dashed line. This means that the two function have to equal:

$$parabola(x) = redline(x) \rightarrow parabola(x) - redline(x) = 0$$

First, we can define the equation using the already found regression models and then use **`fzero`** to find the root. We can find an initial value using the figure, let it be 0.004:

```
inters = @(x) par(x) - l2(x)
```

```
inters = function_handle with value:
    @(x)par(x)-l2(x)
```

```
xi = fzero(inters, 0.004)
```

```
xi = 
       0.00408794769464755
```

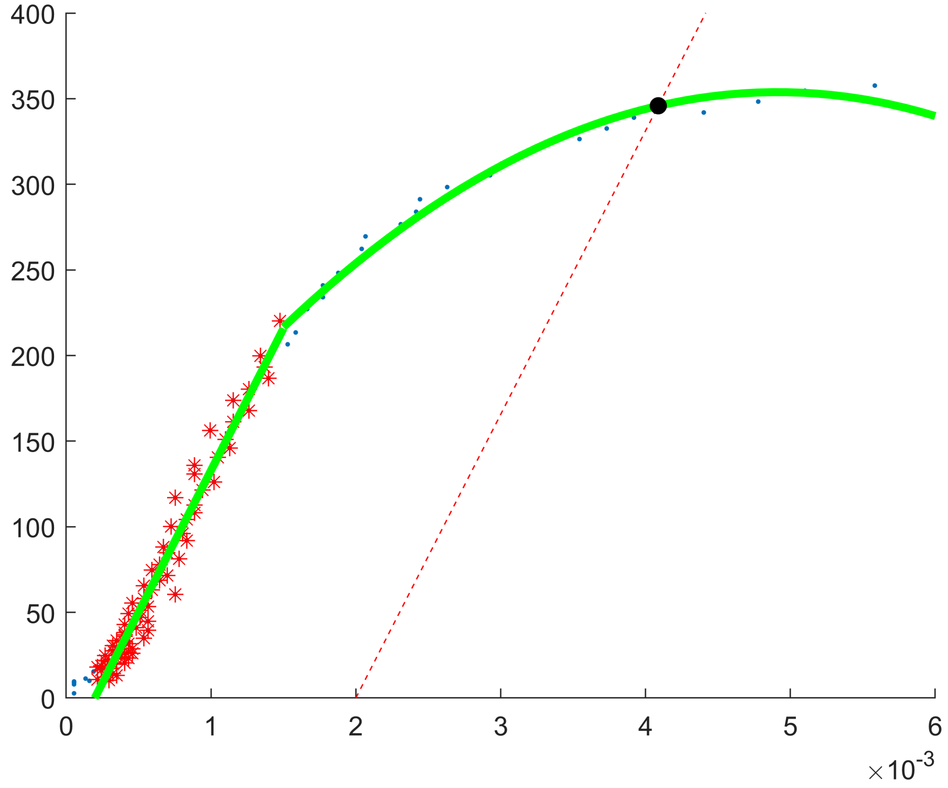
The stress value at this point:

```
offset_stress = par(xi)
```

```
offset_stress = 
          345.818789211111
```

Plot the intersection:

```
plot(xi, offset_stress, 'ko', 'MarkerFaceColor', 'k')
```



Polynomial fit using MATLAB's built-in functions

In the example above, we first fitted a line to the first part of the diagram and a quadratic function to the second part. The regression line is basically a polynomial of degree 1, while the quadratic function is a polynomial of degree 2. We could of course fit higher degree polynomials to the data as well but we always have to be careful when increasing the degree of the regression polynomial. As the degree increases, the global error of the fit will become smaller as the polynomial starts getting closer and closer to the data points, however, between the data points, oscillation will occur (as we will see with the interpolations).

In MATLAB, there is an easier way to fit polynomials to a set of data points using the built-in **`polyfit`** and **`polyval`** functions. The **`polyfit`** function determines the coefficients of a regression polynomial of an arbitrary degree (as long as it is feasible given the data) for an input dataset. The **`polyval`** function on the other hand calculates the value of a polynomial given its coefficients at any point *x*.

Let's see the above example using the two built-in functions. First we fit a degree 1 polynomial to the linear part of the diagram:

```
c1 = polyfit(xf, yf, 1) % inputs: x data, y data, degree
```

```
c1 = 1×2
        165626.174495478          -32.6778764824923
```

The coefficients are given back sorted by the degree of their terms. The first is always the highest degree, which in this case is the slope of the line. The second is the y-intercept in this case.

Fit a quadratic polynomial to the second half of the diagram:

```
c2 = polyfit(xf2, yf2, 2)
```

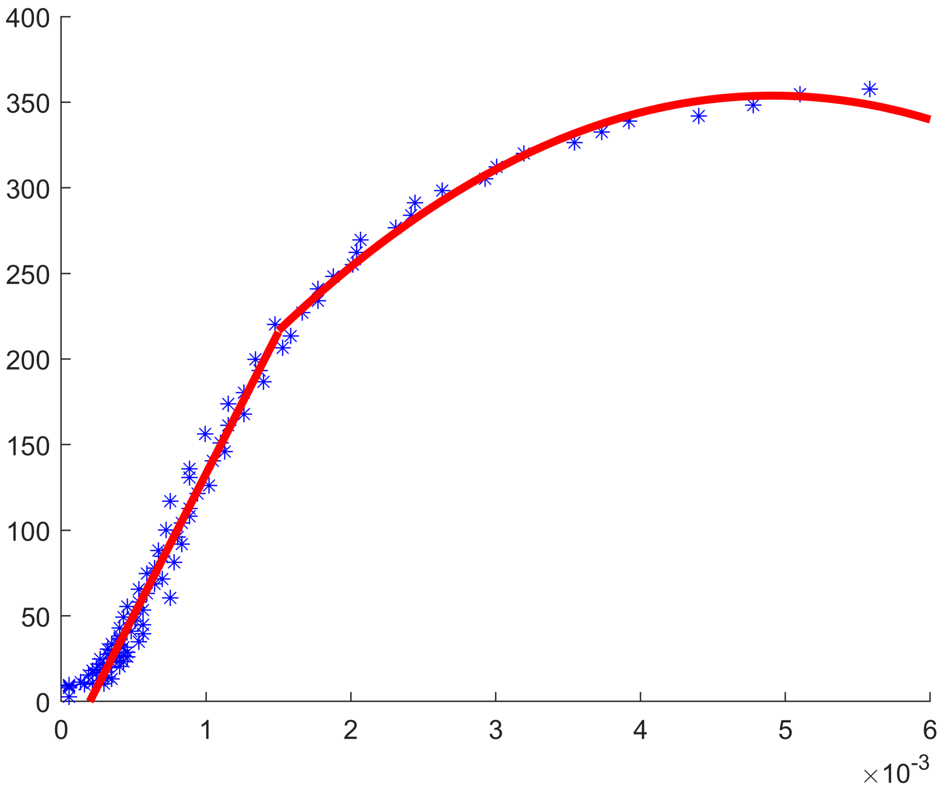
```
c2 = 1×3
        -11790386.3230783            115829.75939882           69.3456670880163
```

We can define two functions that evaluate these polynomials at a given x value using the **`polyval`** function:

```
p1 = @(x) polyval(c1, x); % first input is the vector of coefficients (from the highest degree
p2 = @(x) polyval(c2, x);
```

Create the same plot as before using these new functions:

```
figure(2); hold on;
plot(x, y, 'b*');
fplot(p1, [0, 0.0015], 'r', 'LineWidth', 3);
fplot(p2, [0.0015, 0.006], 'r', 'LineWidth', 3);
axis([0, 0.006, 0, 400])
```



Nonlinear regression using linearization

Many physical phenomena cannot be accurately modelled using only polynomials and therefore we have to use more complex nonlinear functions. For example, when trying to model air density (ρ) as a function of height (h), we have to use an exponential function in the form of $\rho = k \cdot e^{m \cdot h}$, where k and m are parameters of the model. When trying to fit some kind of model to our data, the simples way to do it is to have models that are linear in their parameters. In this section, we deal with nonlinear models that can be linearized in terms of their parameters using some kind of transformation.

The following very common nonlinear models can be easily linearized:

- power functions: $y = k \cdot x^m$
- exponential functions: $y = k \cdot e^{m \cdot x}$ or $y = k \cdot 10^{m \cdot x}$
- reciprocal functions: $y = \frac{1}{m \cdot x + c}$

The linearization is done using new variables that are connected to the original parameters of the regression model. For example, when linearizing a power function, we can take the natural logarithm of both sides:

$$\ln(y) = \ln(k \cdot x^m)$$

$$\ln(y) = \ln(x^m) + \ln(k)$$

$$\ln(y) = m \cdot \ln(x) + \ln(k)$$

If we introduce some new variables: $Y = \ln(y)$, $X = \ln(x)$, $c_1 = m$, $c_2 = \ln(k)$, we can rewrite the expression as a linear model in terms of c_1 and c_2:

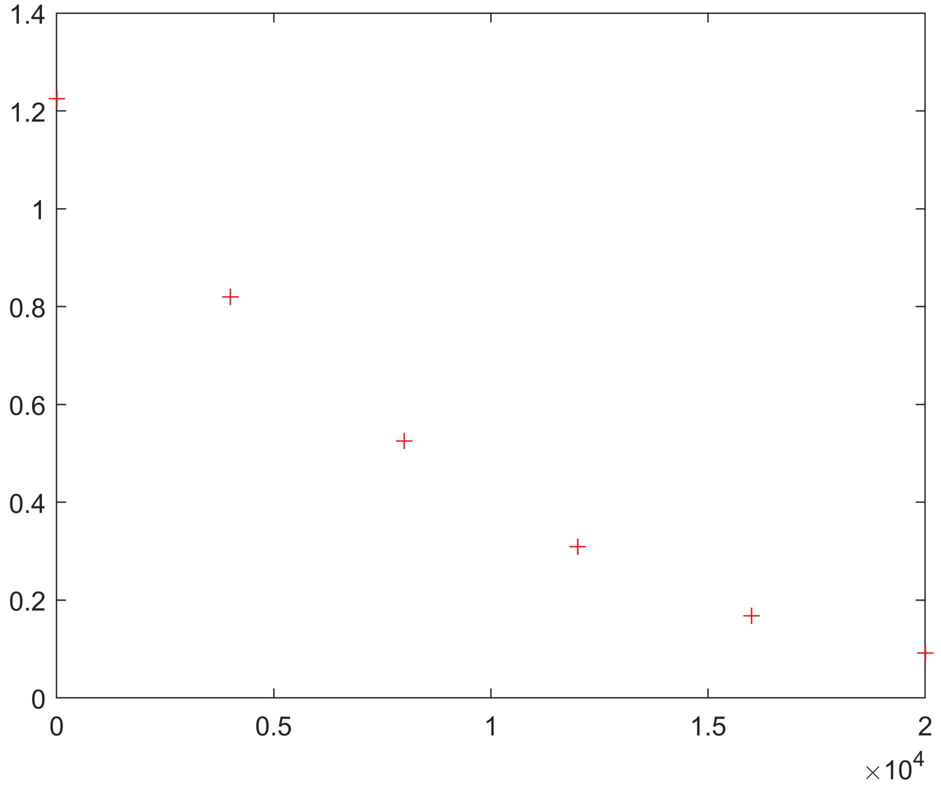
$$Y = c_1 \cdot X + c_2$$

After fitting the model and having found c_1 and c_2, we can find the original parameters of the original model using the relationship between those and the introduces variables:

$$m = c_1 \qquad k = e^{c_2}$$

Let's see an example for this calculation. We have height (m) and air density ($\mathrm{kg/m^3}$) measurements in the file `airdensity.txt`.

```
clear all; close all;
data = load('airdensity.txt');
h = data(:, 1);
r = data(: ,2);
figure(1)
plot(h, r, 'r+')
hold on
```



We are trying to fit the function $\rho = k \cdot e^{m \cdot h}$ to our data, so we will use the linear form $Y = c_1 \cdot X + c_2$ given above where $Y = \ln(\rho)$, $X = h$, $c_1 = m$, $c_2 = \ln(k)$.

```
Y = log(r); % in MATLAB, the log function calculates ln and the log10 calculates logarithm with
X = h;
```

The coefficient matrix:

```
A = [X, ones(length(X), 1)]
```

```
A = 6×2
           1           1
        4000           1
        8000           1
       12000           1
       16000           1
       20000           1
```

Calculating the coefficients from the linear system:

```
c = A\Y
```

```
c = 2×1
     -0.000130223698859203
         0.304918840089502
```

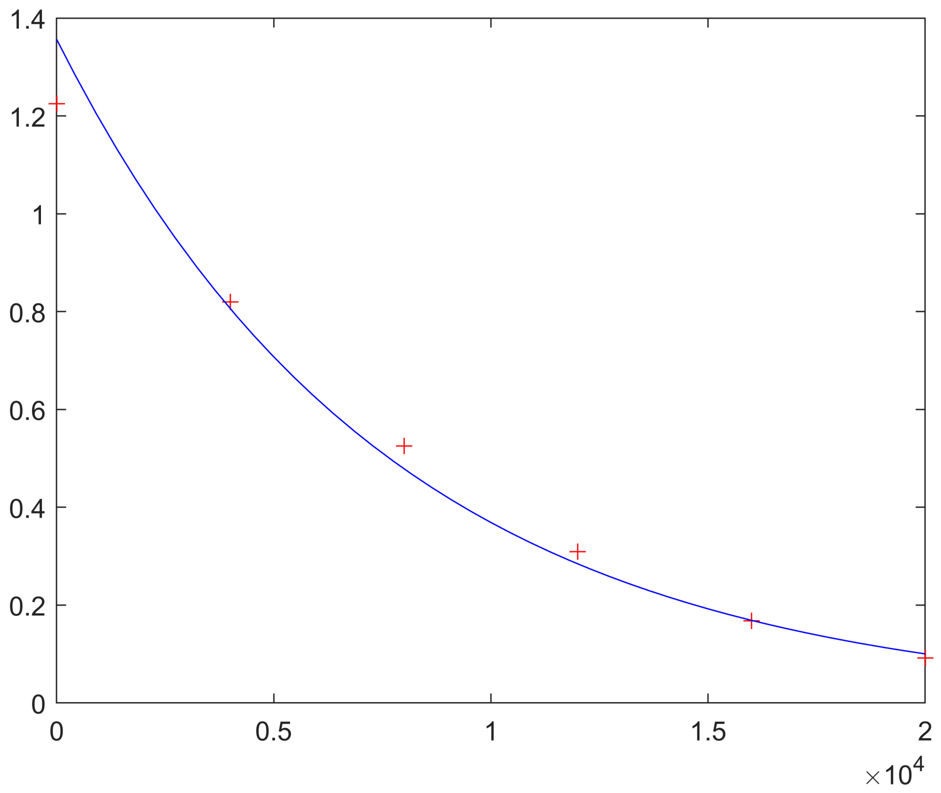
Calculating the parameters of the original model and defining the function of the original model:

```
m = c(1);
```

```
k = exp(c(2));
rho = @(h) k * exp(m * h);
```

Plotting the function with the data points:

```
fplot(rho, [0, 20000], 'b')
```



Using our fitted model for calculation, how much would be the air density on the top of Mount Everest (at ~8850 m)? How high do we have to go in order to reach an air density of 1 kg/m^3?

The answer to the first question is a simple substitution:

```
r8850 = rho(8850)
```

```
r8850 =
         0.428458714355692
```

The air density would be 0.4285 kg/m^3 on Mount Everest according to our measured data.

As for the second question, we have to solve the following equation:

$$\rho(h) = 1$$

$$\rho(h) - 1 = 0$$

We can define this as a new function and find its root by using the **fzero** function:

```
rho1 = @(h) rho(h) - 1;
```

```
h1 = fzero(rho1, 2000) % 2000 meters is the initial guess
```

```
h1 = 
          2341.50037789341
```

We would have to go to 2315 meters to reach an air density of 1 kg/m^3.

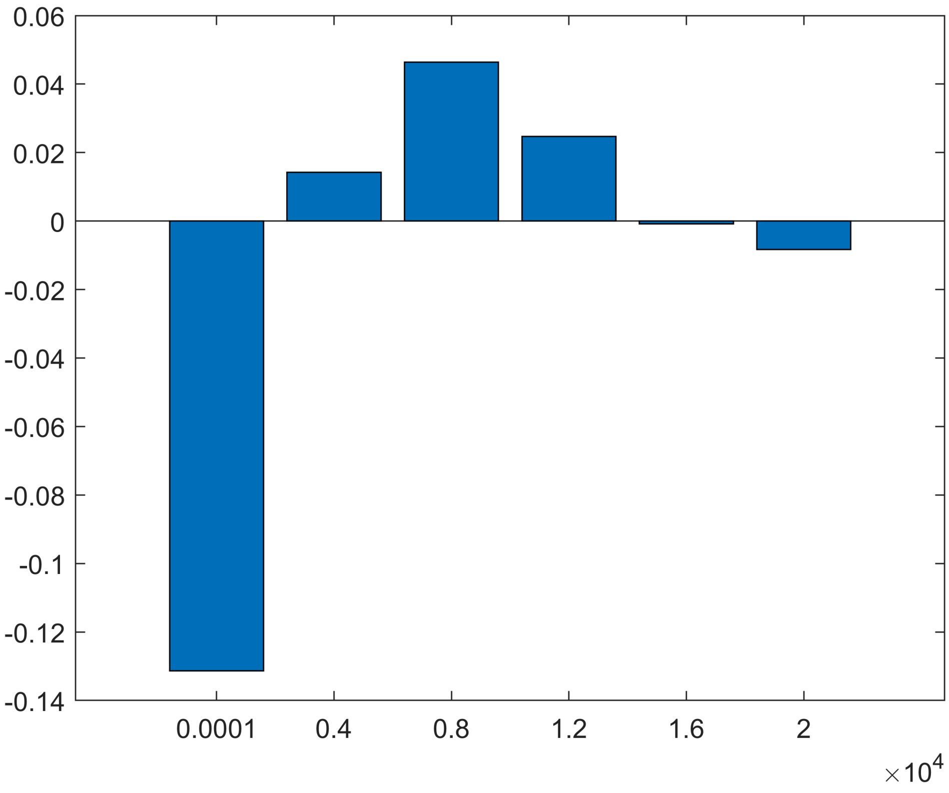
Let's calculate the residuals of the regression model and the square root of the residuals to get an idea about the global goodness of fit. First, we have to calculate the difference between the data point values and the function values of the model at the data points:

```
res = r - rho(h)
```

```
res = 6×1
        -0.131338265023467
        0.0142453242681089
        0.0463907376967921
        0.0247114530494624
     -0.000864216163136117
      -0.00830345508555295
```

An ideal way to visualize these would be a bar plot:

```
figure(2);
bar(h, res);
```



The sum of the residual squares:

```
S = sum(res.^2)
```

```
S =
        0.0202851198146807
```

We have to calculate the difference in the number of measurements and the number of model parameters and then we can find the standard deviation of the residuals:

```
n = length(r) % the number of measurements is the length of the data vector
```

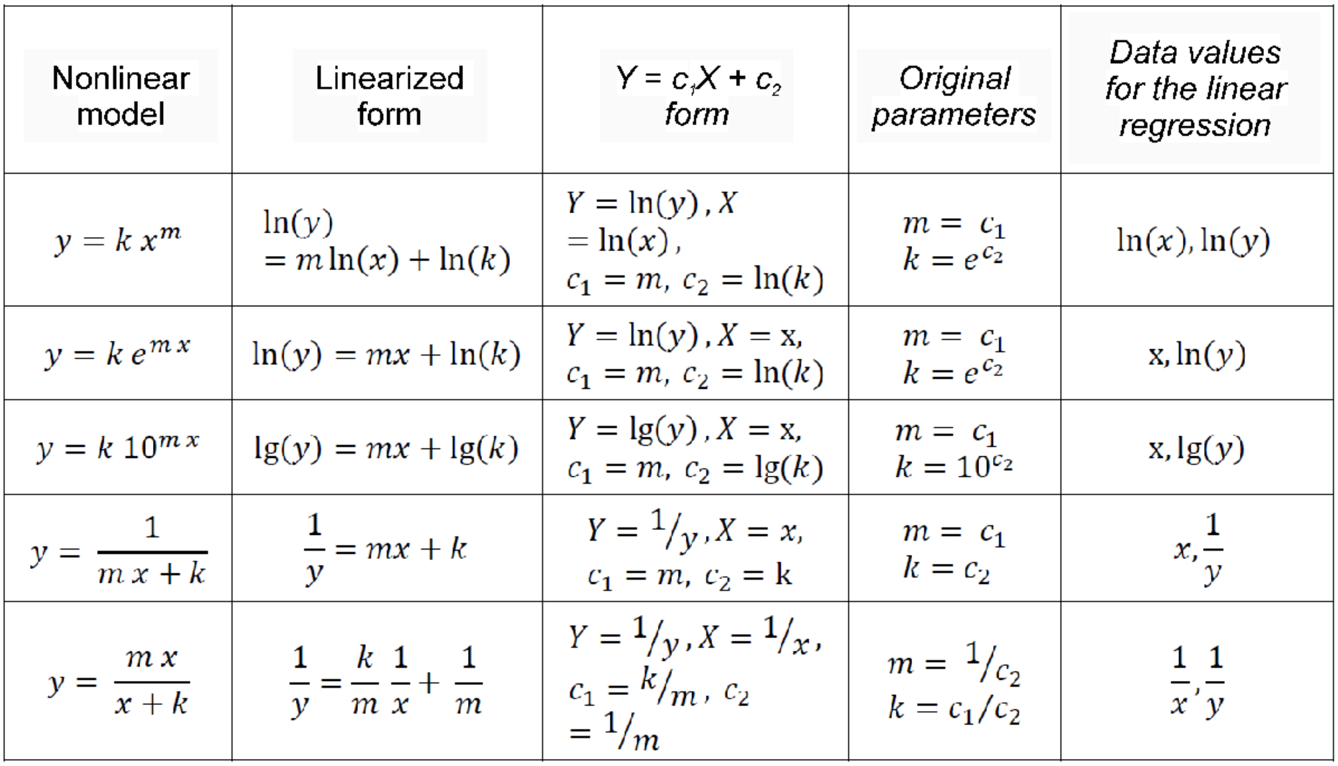
```
n =
     6
```

```
np = 2; % our model had two parameters (m, k)
std = sqrt(S/(n - np))
```

```
std =
        0.0712129198507559
```

Table of the linearization of common nonlinear models

When using some of the most common nonlinear models, the following table shows the transformations we can apply to make the models linear.



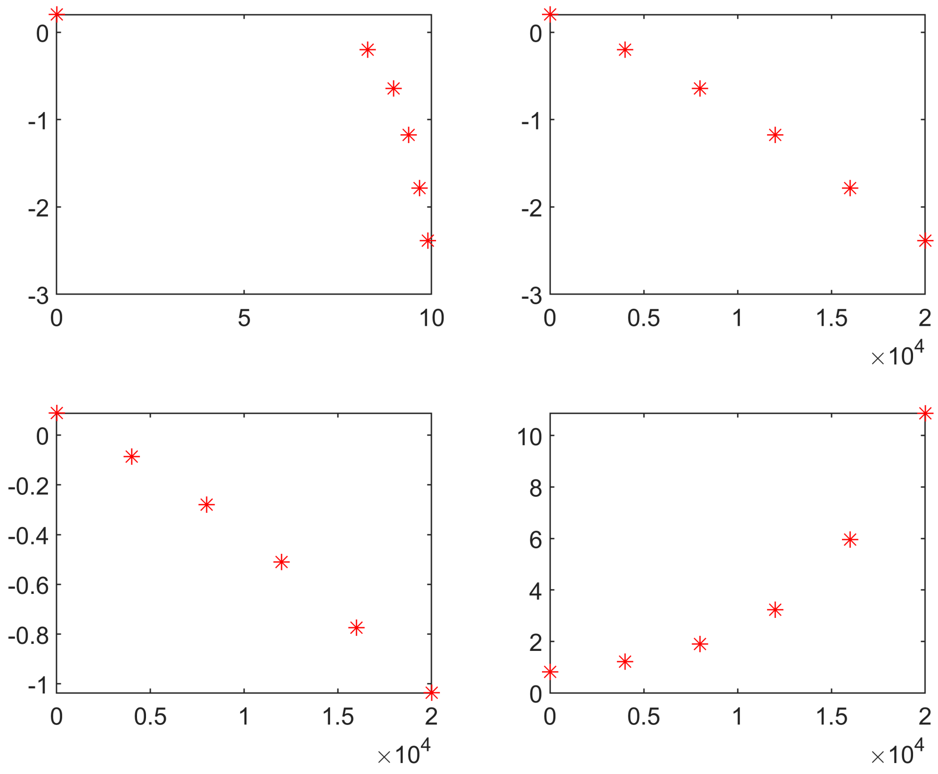
Nonlinear model	Linearized form	*$Y = c_1X + c_2$ form*	*Original parameters*	*Data values for the linear regression*
$y = k\,x^m$	$\ln(y) = m\ln(x) + \ln(k)$	$Y = \ln(y), X = \ln(x), c_1 = m,\ c_2 = \ln(k)$	$m = c_1$ $k = e^{c_2}$	$\ln(x), \ln(y)$
$y = k\,e^{m\,x}$	$\ln(y) = mx + \ln(k)$	$Y = \ln(y), X = \text{x}, c_1 = m,\ c_2 = \ln(k)$	$m = c_1$ $k = e^{c_2}$	$\text{x}, \ln(y)$
$y = k\,10^{m\,x}$	$\lg(y) = mx + \lg(k)$	$Y = \lg(y), X = \text{x}, c_1 = m,\ c_2 = \lg(k)$	$m = c_1$ $k = 10^{c_2}$	$\text{x}, \lg(y)$
$y = \dfrac{1}{m\,x + k}$	$\dfrac{1}{y} = mx + k$	$Y = {}^1/_y, X = x, c_1 = m,\ c_2 = \text{k}$	$m = c_1$ $k = c_2$	$x, \dfrac{1}{y}$
$y = \dfrac{m\,x}{x + k}$	$\dfrac{1}{y} = \dfrac{k}{m}\dfrac{1}{x} + \dfrac{1}{m}$	$Y = {}^1/_y, X = {}^1/_x, c_1 = {}^k/_m, c_2 = {}^1/_m$	$m = {}^1/_{c_2}$ $k = c_1/c_2$	$\dfrac{1}{x}, \dfrac{1}{y}$

Choosing the type of linear regression model for our data

In the previous examples, we initially had models that we wanted to fit to the data points. In can be the case sometimes that we do now have information about the actual model to use. When this happens, we try finding a nonlinear model by plotting our data as finding out which transformation of the data points cause them to be located along a straight line.

For the sake of example, let's imagine that we do not know what exact model to fit to our height-air density data. We can try transforming the data points according to the table above and see which transformation linearizes the data best.

```
figure(3)

x = h; y = r; % data values (heights and air density values)
subplot(2, 2, 1) % 2 x 2 subplot, using the first plot now
plot(log(x), log(y), 'r*') % doesn't seem to be linear

subplot(2, 2, 2) % second subplot
plot(x, log(y), 'r*') % seems linear

subplot(2, 2, 3) % third subplot
plot(x, log10(y), 'r*') % seems linear

subplot(2, 2, 4) % fourth subplot
plot(x, 1./y, 'r*') % not linear
```



The second and the third transformation caused the data points to be linear, therefore our best choice would be to use the models in the second and the third rows of the table above. So, even if we didn't have prior knowledge about the actual model, it would still be a good idea to use the exponential model for the regression.

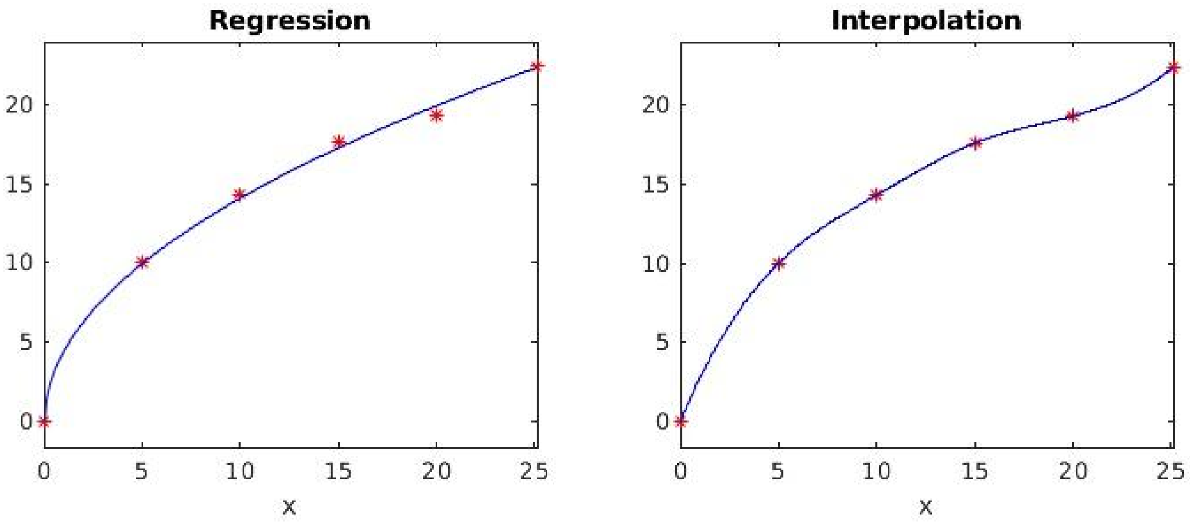
Interpolation

In case of regression, we were looking for the optimal value of the parameters of a certain function that represents our data points in such a way that it minimizes the sum of the squared residuals between the function value and the data point values.

When we talk about interpolation, we mean a function that exactly returns the data values in the data points and can adequately be used to approximate values at unmeasured points. We do not have any prior knowledge about the interpolating function and therefore multiple different choices can give us good results. Depending on the actual problem, we may prefer one or the other.

We will look at two different kind of interpolation:

- Global interpolation: using only one function (a polynomial) to interpolate the data.
- Local interpolation: using multiple functions (lines or splines) to interpolate between certain points of the dataset.



Global interpolation using a polynomial

The generic form of a polynomial of degree n can be given as:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

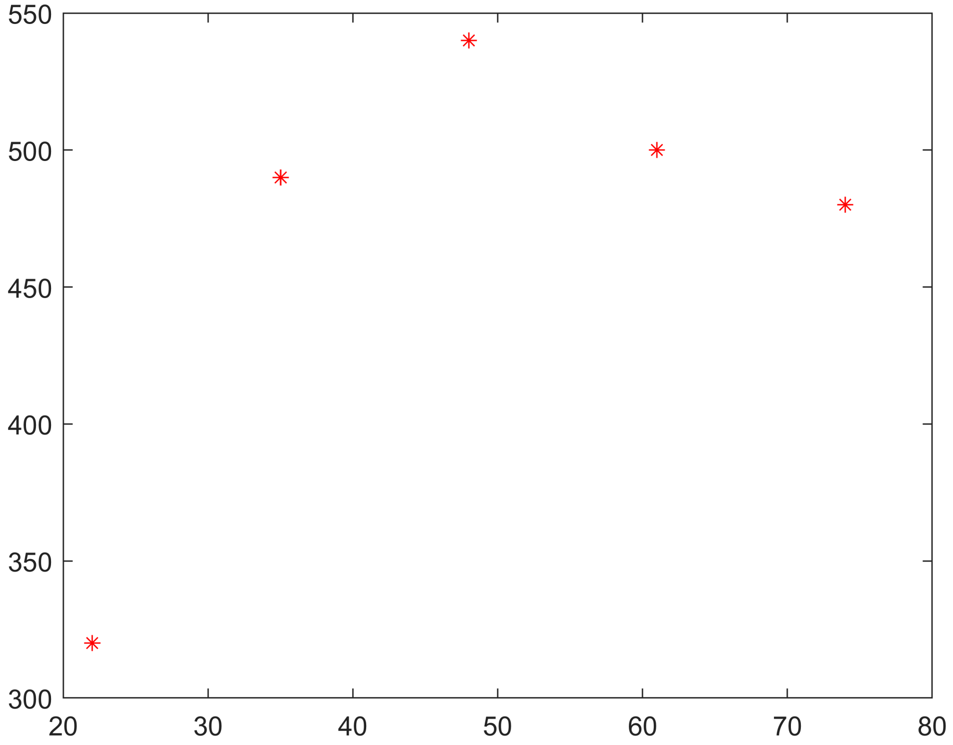
where $a_n, a_{n-1}, \cdots, a_1, a_0$ are real valued coefficients and n is a nonnegative number. A degree 1 polynomial is a simple line, a degree 2 is a parabola and so on. Polynomials of a higher degree are represented by some sort of curve, the higher the degree, the more inflection points the curve has.

If our dataset has n number of points, we can use a polynomial of degree $m = 1, \cdots, n-1$ to approximate the data. When we use a polynomial whose degree is less than $n-1$, we are creating a regression function. When the polynomial has degree $n-1$, we are interpolating the data and the function will go through every data point. This is a global interpolation, as only a single function is used to represent the dataset.

Example 1

The performance of wind turbines is a function of the wind speed. We measured the wind speed (in km/h) and the performance (W) of a turbine and got the following dataset (located in **`windturbine.txt`**).

```
clear all; close all;
data = load('windturbine.txt');
s = data(:, 1);
p = data(:, 2);
plot(s, p, 'r*');
hold on
```



The dataset contains 5 measurements, so we can fit a degree 4 polynomial to the points. We should answer the following questions using the interpolation function:

- How much is the performance of the turbine at wind speeds of 42 and 68 km/h?
- What is the wind speed at 400 W?

A degree 4 polynomial can be given in the form of $f(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$. The coefficient can be calculated from a linear system of equations:

$$\begin{aligned} y_1 &= a_4{x_1}^4 + a_3{x_1}^3 + a_2{x_1}^2 + a_1x_1 + a_0 \\ y_2 &= a_4{x_2}^4 + a_3{x_2}^3 + a_2{x_2}^2 + a_1x_2 + a_0 \\ y_3 &= a_4{x_3}^4 + a_3{x_3}^3 + a_2{x_3}^2 + a_1x_3 + a_0 \\ y_4 &= a_4{x_4}^4 + a_3{x_4}^3 + a_2{x_4}^2 + a_1x_4 + a_0 \\ y_5 &= a_4{x_5}^4 + a_3{x_5}^3 + a_2{x_5}^2 + a_1x_5 + a_0 \end{aligned} \quad \rightarrow \quad A = \begin{bmatrix} x_1^4 & x_1^3 & x_1^2 & x_1 & 1 \\ x_2^4 & x_2^3 & x_2^2 & x_2 & 1 \\ x_3^4 & x_3^3 & x_3^2 & x_3 & 1 \\ x_4^4 & x_4^3 & x_4^2 & x_4 & 1 \\ x_5^4 & x_5^3 & x_5^2 & x_5 & 1 \end{bmatrix} \qquad b = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}$$

The coefficient matrix containing the powers of the x-values from the highest degree to the lowest is called the **Vandermonde matrix**.

Instead of solving the linear system manually, we can use the built-in **`polyfit`** and **`polyval`** commands to find the coefficients and to evaluate the polynomial at any x-value.

```
c = polyfit(s, p, 4)
```

```
c = 1×5
    0.0001   -0.0171    0.5627   12.0190  -62.0517
```

`polyfit` returns the coefficients from the highest degree to the lowest, that is $a_4 = 0.001, a_3 = -0.0171, \cdots$. Using the **`polyfit`** function, we can determine the performance at 42 km/h and 68 km/h wind speeds.

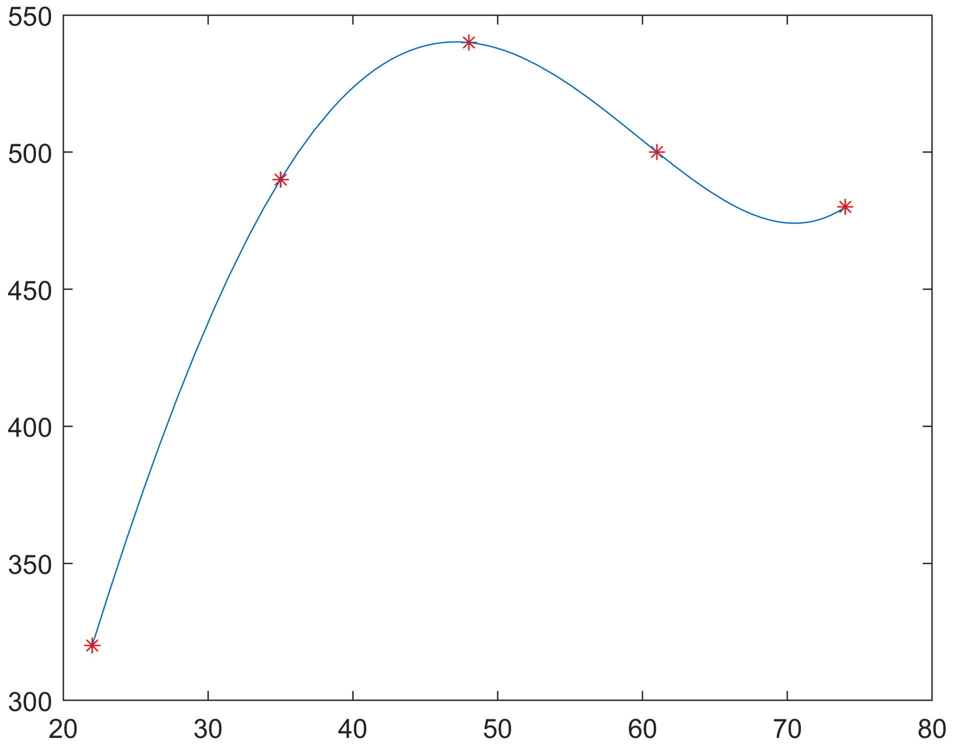
```
polyval(c, 42)
```

```
ans = 531.7853
```

```
fp = @(x) polyval(c, x);
fp(68)
```

```
ans = 476.5008
```

```
fplot(fp, [min(s), max(s)])
```



In order to determine the wind speed at 400 W, we have to solve the $fp(x) = 400 \rightarrow fp(x) - 400 = 0$ equation with, for example, using **`fzero`:**

```
h = @(x) fp(x) - 400;
s400 = fzero(h, 30)
```

```
s400 = 27.1296
```

The turbine reaches a performance of 400 W at a wind speed of ~27 km/h.

To find the coefficients of the interpolating function, we have to solve a linear system of $n+1$ equations (in case of a degree n polynomial) where the coefficient matrix of the system contains the powers of the x-values. As the degree of the polynomial gets higher, the matrix becomes ill-conditioned as it will contain very large and comparably small numbers as well. We can check the condition number of our Vandermonde matrix:

```
A = vander(s)
```

```
A = 5×5
      234256       10648         484          22           1
     1500625       42875        1225          35           1
     5308416      110592        2304          48           1
    13845841      226981        3721          61           1
    29986576      405224        5476          74           1
```

```
cond(A)
```

```
ans = 1.5378e+09
```

Even though our polynomial is only degree 4, the condition number is already in the order of 10^9, which is very high. To ensure numeric stability, interpolating polynomials can be formulated in two other ways than the one seen above, namely the Lagrange and the Newton polynomials.

Lagrange polynomials

The Lagrange polynomials give us the same interpolation polynomial as the one seen before but it can formulated using only the coordinates of the data points and we can avoid solving a linear system to find the coefficients. The general form of the Lagrange polynomials is the following:

$$f(x) = \sum_{i=1}^{n} y_i L_i(x) = \sum_{i=1}^{n} y_i \prod_{j=1, j\neq i}^{n} \frac{(x - x_j)}{(x_i - x_j)}$$

where $\prod_{j=1, j\neq i}^{n} \frac{(x - x_j)}{(x_i - x_j)}$ are called the Lagrange polynomials. Using two data points, the interpolating polynomial will look like the following:

$$f(x) = \frac{(x - x_2)}{(x_1 - x_2)} \cdot y_1 + \frac{(x - x_1)}{(x_2 - x_1)} \cdot y_2$$

For three points:

$$f(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} \cdot y_1 + \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} \cdot y_2 + \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} \cdot y_3$$

Advantage of the Lagrangian form: we do not have to solve a linear system, we can simply use the coordinates of the data points.

Disadvantages: it is somewhat cumbersome to work with as the whole expression has to be written for each approximation of a new data point, we can't just simply use the calculated coefficients. Each time we measure a new data point and want to include it in the interpolation, we have to calculate the whole Lagrange polynomial again.

Newton polynomials

The Newton form for the interpolation polynomial lets us the polynomial in a recursive way using the so-called divided differences. The polynomial can be given in the following form:

$$f(x) = a_1 + a_2(x - x_1) + a_3(x - x_1)(x - x_2) + \cdots + a_n(x - x_1)(x - x_2) \cdots (x - x_{n-1})$$

Using two points, the coefficients are:

$$a_1 = y_1, \quad a_2 = \frac{y_2 - y_1}{x_2 - x_1}$$

For more points, we first have to define the divided differences between two points as the following:

$$f[x_{i+1}, x_i] = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

For three points, we can define the second order divided difference $f[x_{i+1}, x_i, x_{i-1}]$ which is the difference of the two first order divided differences divided by $(x_{i+1} - x_{i-1})$, for three points, this will become the a_3 coefficient:

$$a_3 = f[x_3, x_2, x_1] = \frac{f[x_3, x_2] - f[x_2, x_1]}{x_3 - x_1} = \frac{\frac{y_3 - y_2}{x_3 - x_2} - \frac{y_2 - y_1}{x_2 - x_1}}{x_3 - x_1}$$

Similarly, for four points, we have the third order divided difference, which is the difference between the two third order differences divided by $(x_4 - x_1)$ and so on:

$$f[x_{i+k}, x_i] = \frac{f[x_{i+k}, \cdots, x_{i+1}] - f[x_{i+k-1}, \cdots, x_i]}{x_{i+k} - x_i}$$, where $k = 1, 2, \cdots, n$ and $i = 0, \cdots, n - k$

Advantages of the Newton form are that we only have to determine the coefficients once and after that we can use them to determine any point of the interpolations polynomial. Moreover, if our dataset receives a new measurement, we don't have to recompute the previously found coefficients, only the coefficient corresponding to the new point.

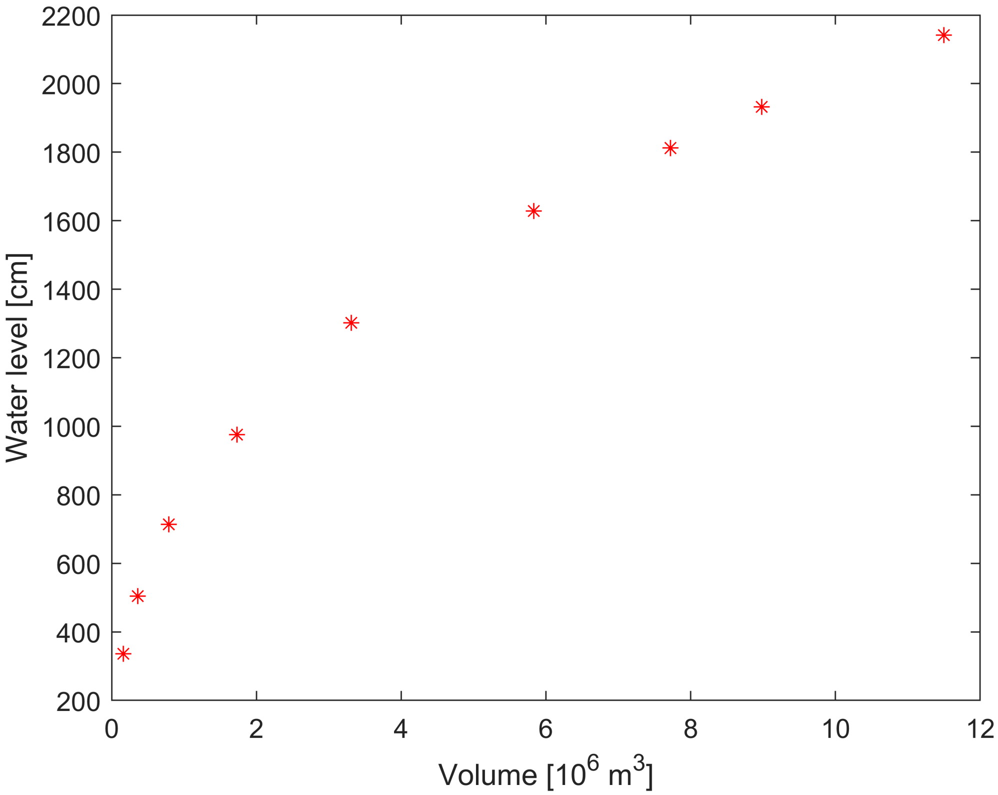
Example 2

The following dataset represents the characteristic curve of a water reservoir. Given a certain water level (H) in the reservoir in cm, using the characteristic curve, we can find the volume (V) of the water in the reservoir in $10^6\,\mathrm{m}^3$ and the area of the water surface (F) in km^2. The dataset is located in the **`reservoir.txt`** file.

Plot the water level vs volume curve and use an interpolation function to find the following:

- the volume corresponding to a water level of 15 m,
- the water level corresponding to a volume of 12 million m^3.

```
clear all; close all;
data = load('reservoir.txt');
H = data(:, 1);
V = data(:, 2);
F = data(:, 3);
figure(1);
plot(V, H, 'r*');
hold on;
xlabel('Volume [10^6 m^3]');
ylabel('Water level [cm]');
```



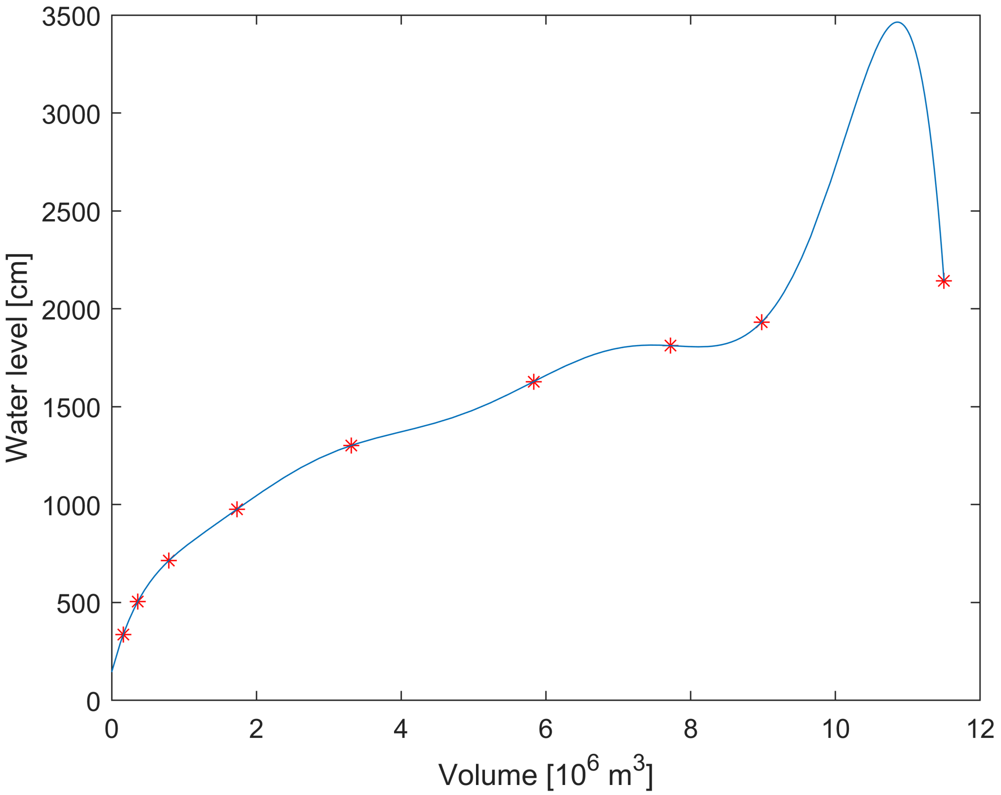
We can determine a polynomial of degree $n-1$, which means, that with 9 data points we will have a degree 8 polynomial.

```
n = length(V); % number of data points
a1 = polyfit(V, H, n-1);
```

```
Warning: Polynomial is badly conditioned. Add points with distinct X values, reduce the degree of the
polynomial, or try centering and scaling as described in HELP POLYFIT.
```

```
p1 = @(x) polyval(a1, x);
```

```
fplot(p1, [0, max(V)]);
```



It is obvious that something is not correct with the interpolation function above. MATLAB is also warning us that the condition number of the Vandermonde matrix is very high and therefore the polynomial is badly conditioned. Because the degree of the polynomial is very high, it starts to oscillate at the end of the dataset. This oscillation is called the **Runge's phenomenon.** This shows us that increasing the degree of the interpolation polynomial is not always a suitable way to go forward and we will have to find another solution to the problem.

We can check the condition number of the Vandermonde matrix to reassure our presumption:

```
A = vander(V);
cond(A)
```

```
ans = 2.7260e+10
```

It is in the order of 10^{10} which is obviously very large.

Local interpolation - Splines

Instead of using one global interpolation function, if we have a larger dataset, it is better to use multiple lower degree polynomials. Each of these polynomials will only use a small number of data points and therefore will only interpolate the dataset between those points. This is why these methods are called local interpolation methods as the coefficients of each interpolation function are only determined by the points in the section surrounding the function. Each of the interpolation polynomials have the same number of coefficients only their values differ.

The simplest local interpolation we can think of is when we connect the points using lines (linear polynomials). We can also use quadratic or third-degree curves (**splines**). The degree of a spline tells us the number of coefficients in the function, that is, a spline of degree n has at most n number of coefficients. It is often desirable for the interpolation function to be not only continuous at the connecting points (vertices), but also be smooth. The order of the spline therefore tells us which order differential of the adjacent splines are made equal. For example, a first order spline of degree 2 is a quadratic polynomial whose first order derivatives are equal at the vertices (data points).

The most common type of spline is the second order cubic spline, which means that we have cubic polynomials between the points and first and second order derivatives of each spline section are equal in the vertices. Sometimes, the so-called Hermite splines are also used, these are first order cubic splines.

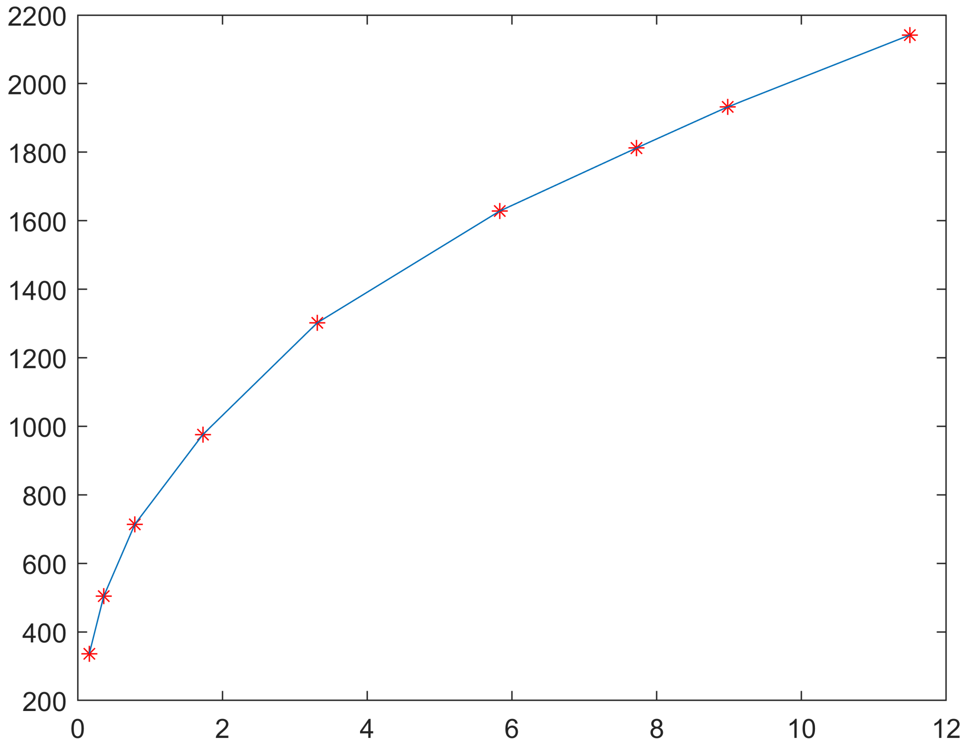
Linear interpolation

Given n data points, we can have $n-1$ linear interpolation functions between the points. Each function is in the following form:

$$f(x) = \frac{(x - x_{i+1})}{(x_i - x_{i+1})} \cdot y_1 + \frac{(x - x_i)}{(x_{i+1} - x_i)} \cdot y_2$$

In MATLAB, we can use the **`interp1`** command carry out any kind of spline interpolation. Let's use lines first to interpolate the dataset:

```
sp = @(x) interp1(V, H, x, 'linear'); % the 'linear' keyword can be omitted
figure(2)
plot(V, H, 'r*'); hold on;
fplot(sp, [min(V), max(V)])
```



Using linear interpolation, our function is continous but not smooth. If smoothness is required, we have to use higher order splines.

Second order cubic spline interpolation

We fit cubic polynomials between the points. Each polynomial can written in the form:

$$y = ax^3 + bx^2 + cx + d$$

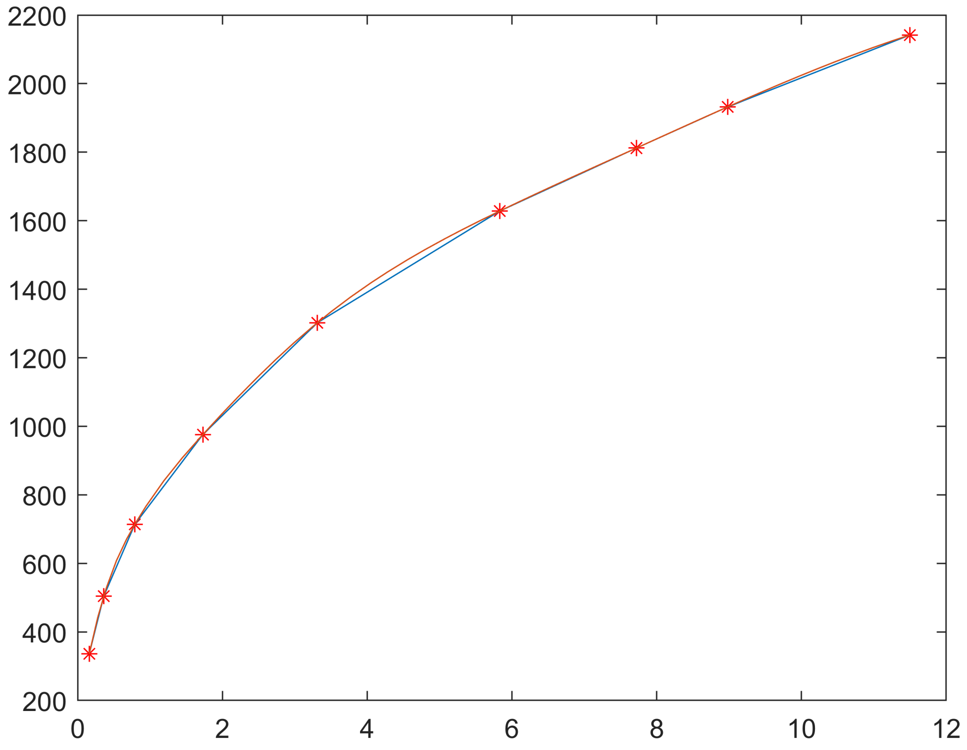
At the vertices, the first and second order derivatives of the adjacent splines are set to be equal.
Given n number of points, we have $n - 1$ sections and we have to compute a spline for each section. Each spline has 4 unknown coefficients, which results in $4 \cdot (n - 1) = 4n - 4$ uknowns. We can use the following equations determined by the contraints:

- Each spline has to go through the starting and the end point of the section. This gives us $2 \cdot (n - 1) = 2n - 2$ equations in the following form: $y_i = a_i x^3 + b_i x^2 + c_i x + d_i$.
- There are $(n - 2)$ central points (not counting the starting and the end point). At each of these points, the first order differentials ($y' = 3ax^2 + 2bx + c$) of the adjacent splines are set equal: $3a_i x^3 + 2b_i x^2 + c_i = 3a_{i+1} x^3 + 2b_{i+1} x + c_{i+1}$.
- At each of the central points, the second order differentials ($y'' = 6ax + 2b$) are also set equal: $6a_i x + 2b_i = 6a_{i+1} x + 2b_{i+1}$.

These give us $4n - 6$ equations to solve for $4n - 4$ unknowns, so we still have to set two constraints. One possibility is to set the second order differentials at the end and the starting points to zero, in this case, we get the so-called natural cubic spline. The other choice can be the "not-a-knot" solution, which means that at the end and the starting points, the third order differentials are set equal as well.

Let's fit a second order cubic spline to the reservoir dataset. We can use the **interp1** command, but now we have to specify the **spline** keyword:

```
sp2 = @(x) interp1(V, H, x, 'spline');
fplot(sp2, [min(V), max(V)])
```



For spline fitting, we can use the **`spline`** command, which can only fit a second order cubic spilne but otherwise works exactly the same way the **`interp1`**.

```
sp2 = @(x) spline(V, H, x);
```

Let us answer the question concerning the example. The water level corresponding to the volume of 12 million m^3:

```
H12 = sp2(12)
```

```
H12 = 2.1741e+03
```

Answer: 2174 cm.

The volume corresponding to a water level of 15 m:

```
f = @(x) sp2(x) - 1500;
V1500 = fzero(f, 5)
```

```
V1500 = 4.6699
```

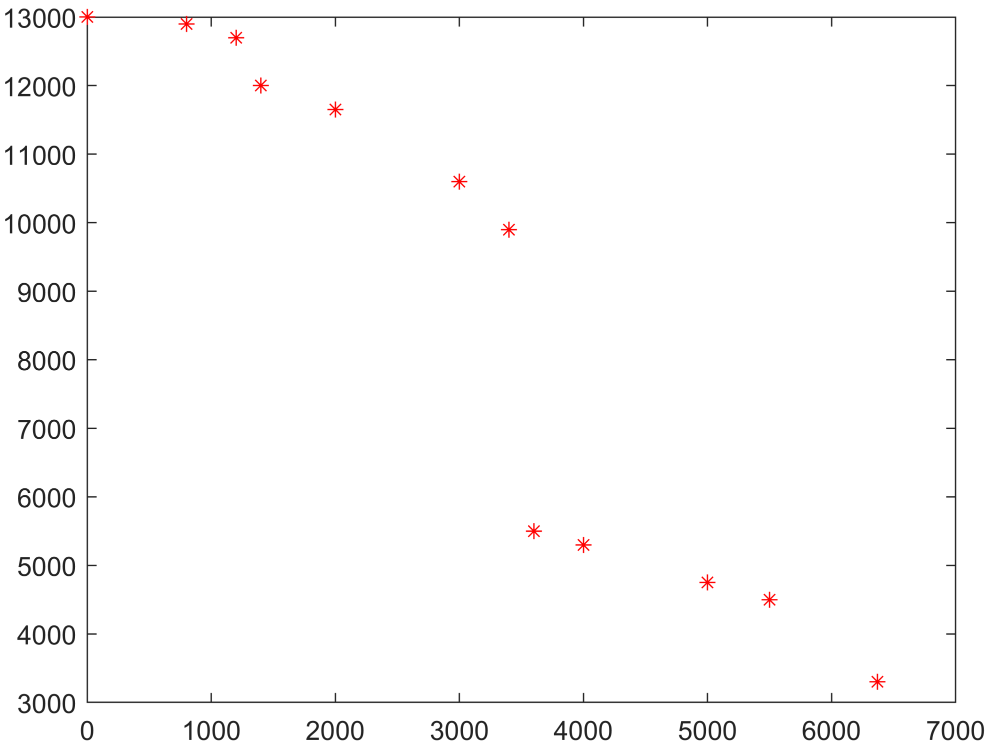
Answer: 4.6699 million m^3.

First order cubic spline interpolation

In some cases, it is beneficial to use the Hermite splines for interpolation, let's see an example for that.

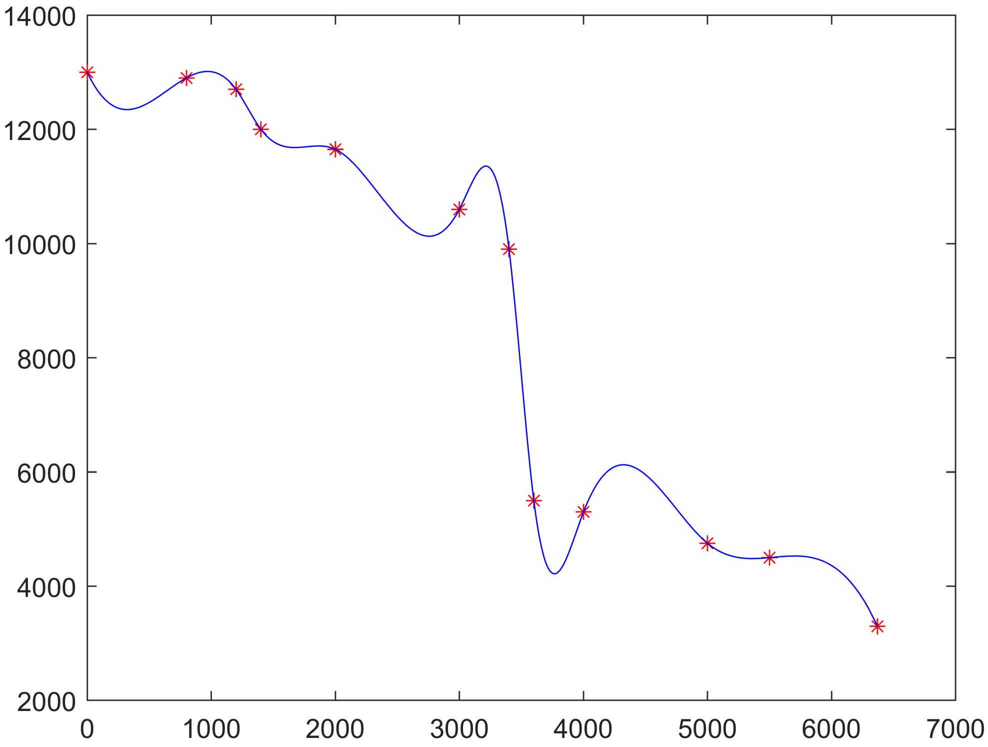
The density ($\rho\ [\mathrm{kg/m^3}]$) of the Earth is a function of the distance (radius, $r\ [\mathrm{km}]$) from the center of the planet. The dataset in **earth_density.txt** contains corresponding values of radii and density measurements. Using the data, determine the density at the radius of 3200 km and find the radius that corresponds to 4000 $\mathrm{kg/m^3}$.

```
clear all; close all;
data = load('earth_density.txt');
r = data(:, 1);
rho = data(:, 2);
figure(1);
plot(r, rho, 'r*'); hold on;
```



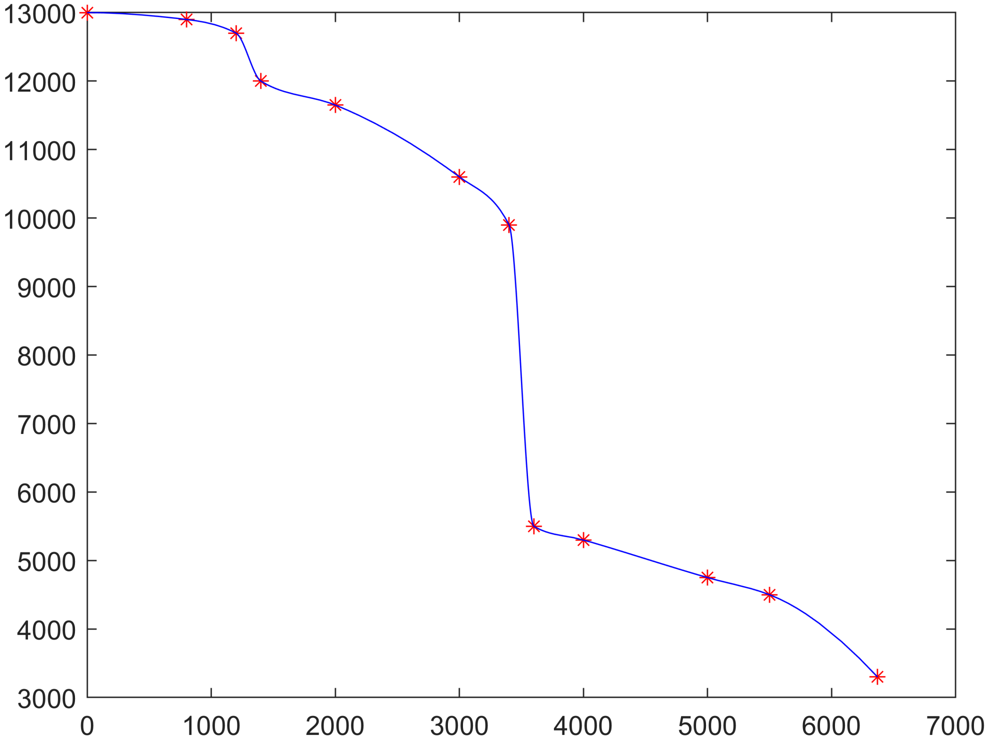
Let's try to fit a second order cubic spline to the data first:

```
sp1 = @(x) interp1(r, rho, x, 'spline');
fplot(sp1, [min(r), max(r)], 'b');
```



We can see that setting the second order derivatives equal at the vertices causes the splines to oscillate between the points. Better results can be achieved if only the first order derivatives are set equal:

```
sp2 = @(x) interp1(r, rho, x, 'pchip'); % pchip = piecewise Hermite interpolation polynomial
figure(2);
plot(r, rho, 'r*'); hold on;
fplot(sp2, [min(r), max(r)], 'b');
```



We can answer the questions using the Hermite interpolation:

```
rho3000 = sp2(3200)
```

```
rho3000 = 1.0361e+04
```

The density at 3200 km is 10361 $\mathrm{kg/m^3}$.

```
f = @(x) sp2(x) - 4000;
R4000 = fzero(f, 5500)
```

```
R4000 = 5.9589e+03
```

The density is equal to 4000 $\mathrm{kg/m^3}$ at a radius of approximately 6000 km.

2D Interpolation and regression

In the previous practicals, we looked at regression models, as well as global and local one dimensional interpolation of data, such as using a polynomial, linear sections or splines. In many cases, our problem is two dimensional in nature, so we cannot use one dimensional regression or interpolation, but luckily, many previously studied methods can be extended into higher dimensions as well.

Interpolation of parametric curves

Most of the curves, that are used to model some data in practice, cannot be described using a single mathematical function, as there exist values of the independent variable (x, for example) where the curve has two distinct "function values". For example, when modelling a path or road using measured data points along the path, if it turns back on itself or loops, it cannot be modelled by a single function. In these cases, we use parametric curves consisting of independent functions for the different coordinates, where the parameter can be the number of points or the arc length along the curve.

As an example, let's see the following problem. We used a GPS to measure a path between two points. The GPS measured and saved data (with a certain frequency) as we went along the path resulting in a data matrix containing the x and y coordinates of the points.

```
close all; clear all; format longG;
data = load('path.txt');
x = data(:, 1);
y = data(:, 2);
figure(1);
plot(x, y, 'b-');
```

We would like to find the coordinates of the point that is 500 meters from the beginning of the path using cubic, second order spline interpolation. Our parameter will be the arc length along the path which we can approximate by using the distances between the measured points. As the path cannot be modelled by a single function, we have to use a different spline interpolation for the x and the y coordinates:

$$x(t) = c_1 + c_2 \cdot t + c_3 \cdot t^2 + c_4 \cdot t^3$$

$$y(t) = c_5 + c_6 \cdot t + c_6 \cdot t^2 + c_7 \cdot t^3$$

We first have to calculate the distances between the points, to approximate the arc length along the curve. To do this, we can use the built-in **`diff`** command that calculates the differences between points in a vector. If we have ***n*** number of points, we get back a vector containing ***n-1*** values:

```
dx = diff(x)
```

```
dx = 24×1
           84.3399999999674
           46.4600000000792
           31.5299999999115
           40.7100000000792
           24.5499999999302
           24.3399999999674
          0.200000000069849
          -19.9500000000698
          -40.7099999999627
          -9.97999999998137
```

```
        ⋮
```

```
dy = diff(y)
```

```
dy = 24×1
         -42.4800000000396
          -22.929999999993
         -22.1500000000233
          -61.859999999986
          -12.570000000007
          15.1600000000326
          27.3399999999674
          25.9400000000023
          39.1199999999953
           27.140000000014
        ⋮
```

Now we can compute the distances between the points using the differences. The first distance is 0, as the first point is to 0 distance from itself.

```
d0 = [0; sqrt(dx.^2 + dy.^2)]
```

```
d0 = 25×1
                         0
          94.4340298832887
          51.8103898847233
          38.5326277328117
          74.0537892346146
           27.580924567475
          28.6750972099381
          27.3407315190769
          32.7243960983684
          56.4595297535908
        ⋮
```

If we cumulatively sum up the distances using the **`cumsum`** function, we get the distance of each point from the starting point:

```
d = cumsum(d0)
```

```
d = 25×1
                         0
          94.4340298832887
          146.244419768012
          184.777047500824
          258.830836735438
          286.411761302913
          315.086858512851
          342.427590031928
          375.151986130297
          431.611515883887
        ⋮
```

We can use the built in **`spline`** command to define the spline interpolations using the **`d`** variable as our independent variable:

```
xs = @(u) spline(d, x, u);
ys = @(u) spline(d, y, u);
hold on;
fplot(xs, ys, [0, d(end)], 'r--')
```

The coordinates of the 500 meter marker:

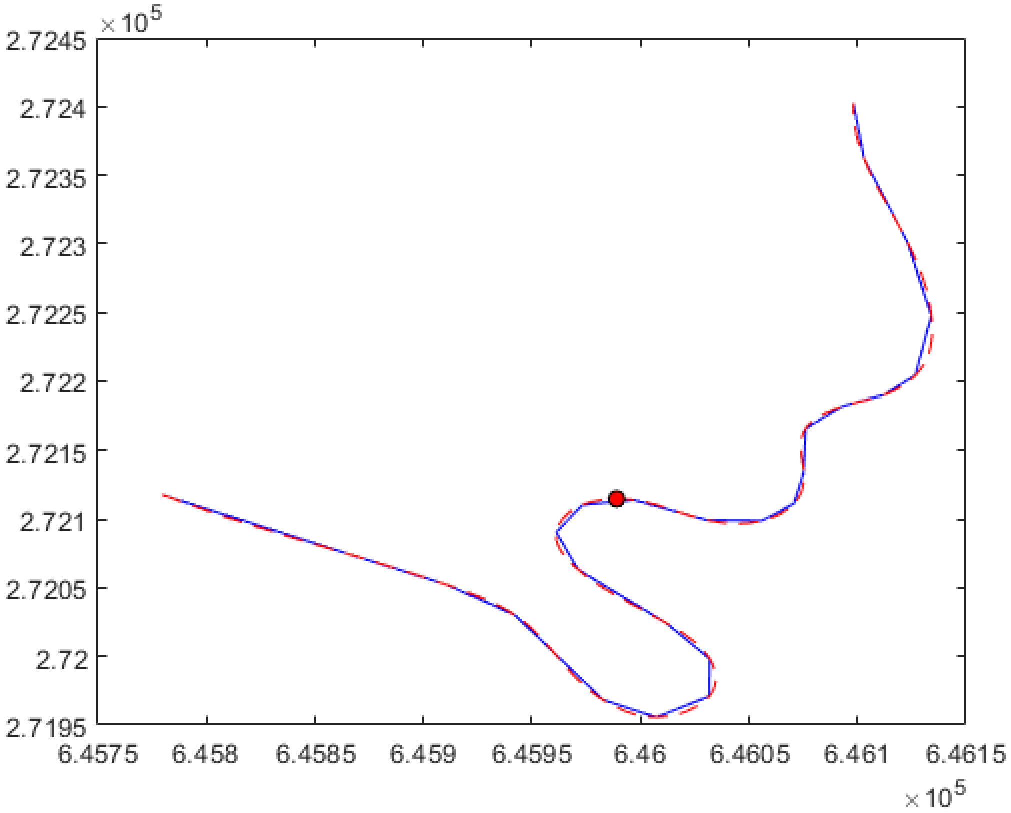
```
x500 = xs(500)
```

```
x500 =
         645989.003262212
```

```
y500 = ys(500)
```

```
y500 =
         272114.837059394
```

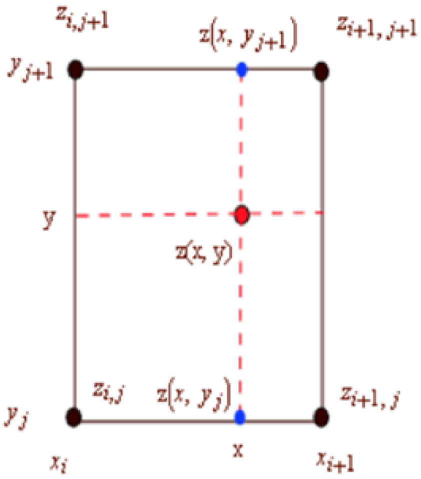
```
plot(x500, y500, 'ko', 'MarkerFaceColor', 'r');
```



2D interplation of data given on a regular grid

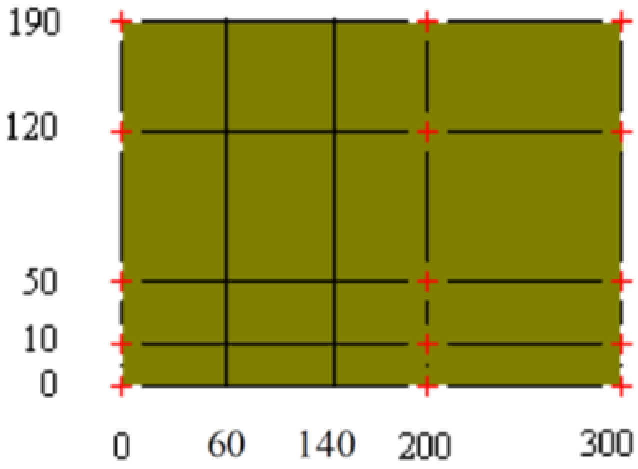
If we have data points that are located in a regular grid, the simplest form of interpolation we can use the so-called bilinear interpolation. The interpolation incorporates three computational steps:

1. We linearly interpolate along $y = y_j = \text{const.}$ and calculate $z(x, y_j)$.
2. We linearly interpolate along $y = y_{j+1} = \text{const.}$ and calculate $z(x, y_{j+1})$.
3. We linearly interpolate along x, between $z(x, y_j)$ and $z(x, y_{j+1})$ and find the value $z(x, y)$.



If we introduce more points to the interpolation process, we can use higher order polynomials as well (such as the bicubic interpolation).

As an example let's look at terrain heights measured in a grid. The layout of the measurements, with their x and y coordinates, is given in the following figure:



The matrix containing the measurements are located in the **`terrain.txt`** file. Let's load them:

```
clear all; close all;
x = [0, 60, 140, 200, 300];
y = [190, 120, 50, 10, 0];
Z = load('terrain.txt');
```

The built-in function calculating the interpolation works on matrices, so we first have to create a regular grid from the vectors of the x and y coordinates. We can use the built-in **`meshgrid`** command for this.

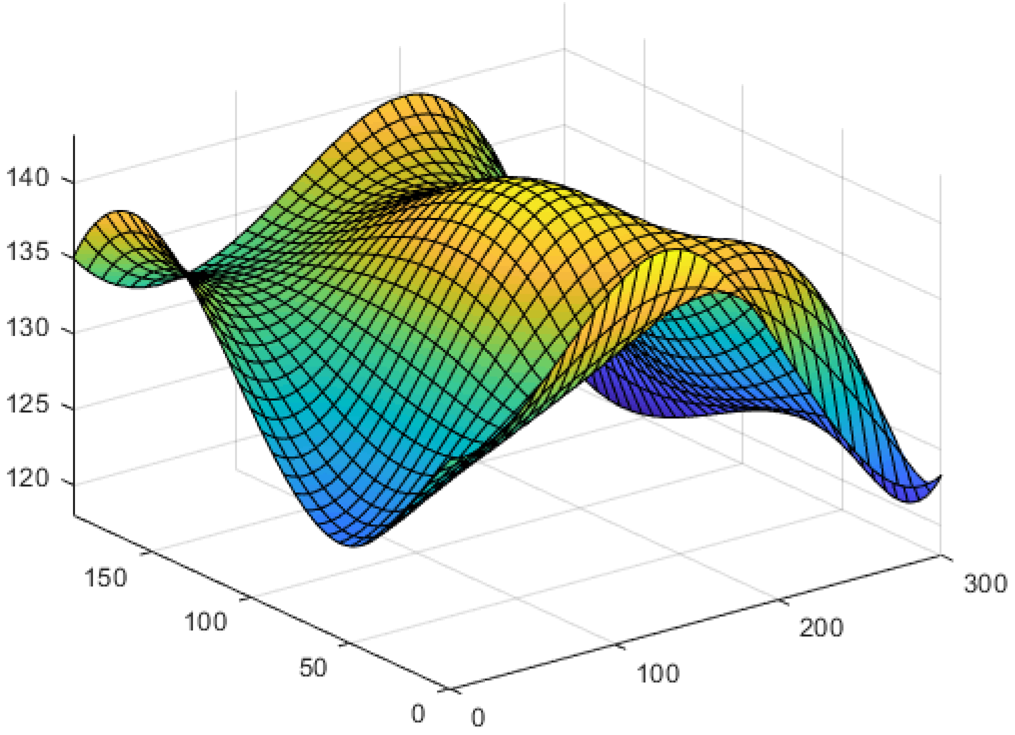
```
[X, Y] = meshgrid(x, y)
```

```
X = 5×5
     0    60   140   200   300
     0    60   140   200   300
     0    60   140   200   300
     0    60   140   200   300
     0    60   140   200   300
Y = 5×5
   190   190   190   190   190
   120   120   120   120   120
    50    50    50    50    50
    10    10    10    10    10
     0     0     0     0     0
```

Now, using the X, Y and Z matrices, we have 3 dimensional coordinates for each terrain height. For two dimensional interpolation, we can use the **`interp2`** command, similarly to the **`interp1`** from the previous practical. The **`interp2`** command can only work with data given in a regular grid (the grid points don't necessarily have to be equidistant). We specify different methods for the interpolation:

- `nearest` - nearest neighbor interpolation,
- `linear` - 2D linear interpolation (bilinear),
- `spline` - cubic spline,
- `cubic` - 2D cubic interpolation (bicubic).

```
F = @(u, v) interp2(X, Y, Z, u, v, 'spline');
figure(1);
fsurf(F, [min(x), max(x), min(y), max(y)]);
```



Using our interpolation, we can answer the following questions:

1. What is the height in (100, 100)?
2. What does an West-East section in this point look like?
3. How much cut and fill is needed if we want to level the surface at 135 meters?

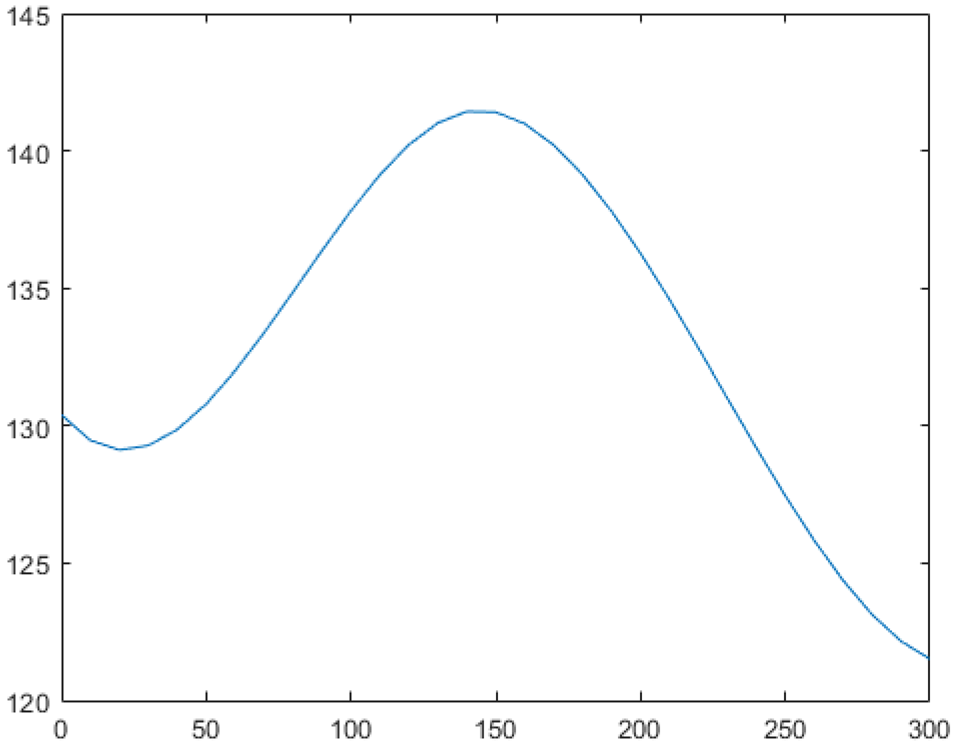
The answer to the first question can be found by a simple query using the interpolation:

```
F(100, 100)
```

```
ans =

        137.807868245719
```

The W-E section at (100, 100) can be drawn by creating two vectors corresponding to the coordinates in the section. The x coordinates go from 0 to 300 with some resolution (let's say, 10 meters) and the y values are a constant 100:

```
xs = 0:10:300; % x coordinates at every 10 meters from 0 to 300
ys = 100*ones(1, length(xs)); % y coordinates are always 100
zs = F(xs, ys); % terrain height from the interpolation
figure(2);
plot(xs, zs);
```



To answer the third question, the amount of cut and fill can be calculated using 2D integration (covered in subsequent practicals). If we calculate the volume between the terrain and a 0 reference level and then subtract it from the volume under the horizontal surface at 135 m, we get the total amount of cut and fill needed. The volume under the terrain can be calculated using the following integral:

$$V_T = \int_0^{190} \int_0^{300} F(x, y)\,\mathrm{dx}\,\mathrm{dy}$$

```
VT = integral2(F, 0, 300, 0, 190)
```

```
VT =
               7614016.55348964
```

The volume under the level surface at 135 m:

```
VS = 135 * 300 * 190
```

```
VS =
          7695000
```

The sum of the cut and the fill volumes:

```
V = VS - VT
```

```
V =
80983.4465103578
```

2D regression of data given on an irregular grid

When our data is not located on a regular grid, interpolating it becomes somewhat harder. In case of polynomial regression however, data location can be arbitrary, although, as the degree of the polynomial gets higher, we see similar oscillation as in the one dimensional case (Runge's phenomenon). Moreover, in the two dimensional case, the number of coefficients grows very rapidly as the degree goes up, which means that higher degree polynomials require a lot more data points.

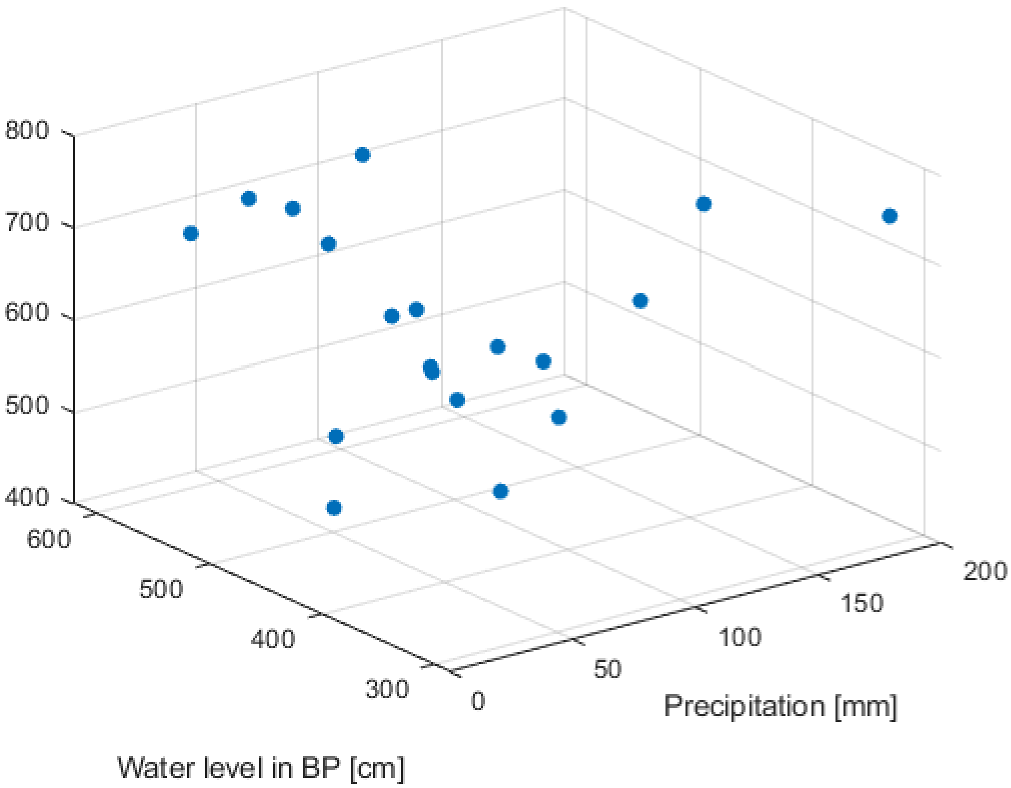
Let's look at the following example. The catchment area (the source of water) of the section of the Danube in Hungary is located in Austria and Bavaria. Depending on the precipitation in these areas, and the water level of the Danube in Budapest at the same time, we can forecast the height of the flood wave at Budapest caused by the precipitation.

Our data matrix located in **`flood.txt`** contains 3 columns:

1. The precipitation in the catchment area that caused flood waves on the Danube, in mm.
2. The water level of the Danube at Budapest during the precipitation, in cm.
3. The height of the flood wave at Budapest caused by the precipitation in cm.

Let's load the data and visualize it:

```
clear all; close all;
data = load('flood.txt');
x = data(:, 1);  % precipitation in the catchment area [mm]
y = data(:, 2); % water level in BP during the precipitation [cm]
z = data(:, 3); % forecast flood wave height in BP [cm]
figure(1);
scatter3(x, y, z, 'filled'); % 3D scatter plot
xlabel('Precipitation [mm]');
ylabel('Water level in BP [cm]')
```



The simplest regression polynomial we can use is a regression plane:

$$z(x, y) = c_1 + c_2 x + c_3 y$$

The generic form of a more complicated, 2D quadratic regression polynomial:

$$z(x, y) = c_1 + c_2 x + c_3 y + c_4 x^2 + c_5 xy + c_6 y^2$$

Our task is to approximate the height of the flood wave in Budapest, if the precipitation in the catchment area was 100 mm and the water level in Budapest at that time was 400 cm. In order to do this, we can use a 2D quadratic polynomial to model the data and query the value of the model at the given values.

The regression polynomial can be written in matrix form (where ***n*** is the number of data points):

$$A = \begin{bmatrix} 1 & x_1 & y_1 & x_1^2 & x_1 y_1 & y_1^2 \\ 1 & x_2 & y_2 & x_2^2 & x_2 y_2 & y_2^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & y_n & x_n^2 & x_n y_n & y_n^2 \end{bmatrix} \qquad c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_6 \end{bmatrix} \qquad b = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_6 \end{bmatrix}$$

$$\boldsymbol{A} \cdot \boldsymbol{c} = \boldsymbol{b}$$

In MATLAB:

```
n = length(x); % number of data points
A = [ones(n, 1), x, y, x.^2, x.*y, y.^2]
```

```
A = 19×6
```

```
            1          58         405        3364       23490      164025
            1          52         450        2704       23400      202500
            1         133         350       17689       46550      122500
            1         179         285       32041       51015       81225
            1          98         330        9604       32340      108900
            1          72         400        5184       28800      160000
            1          72         550        5184       39600      302500
            1          43         480        1849       20640      230400
            1          62         450        3844       27900      202500
            1          67         610        4489       40870      372100
      ⋮
```

```
c = A\z
```

```
c = 6×1
         -1339.90488568953
          17.1883973493066
          4.95723840977286
       -0.0372951053113031
       -0.0179199642155538
      -0.00333877203214026
```

We can also solve the fitting of the regression model using the built-in **`regress`** function. The function can give us back the 95% confidence intervals the residual values at the points as well:

```
[c2, int95, res] = regress(z, A)
```

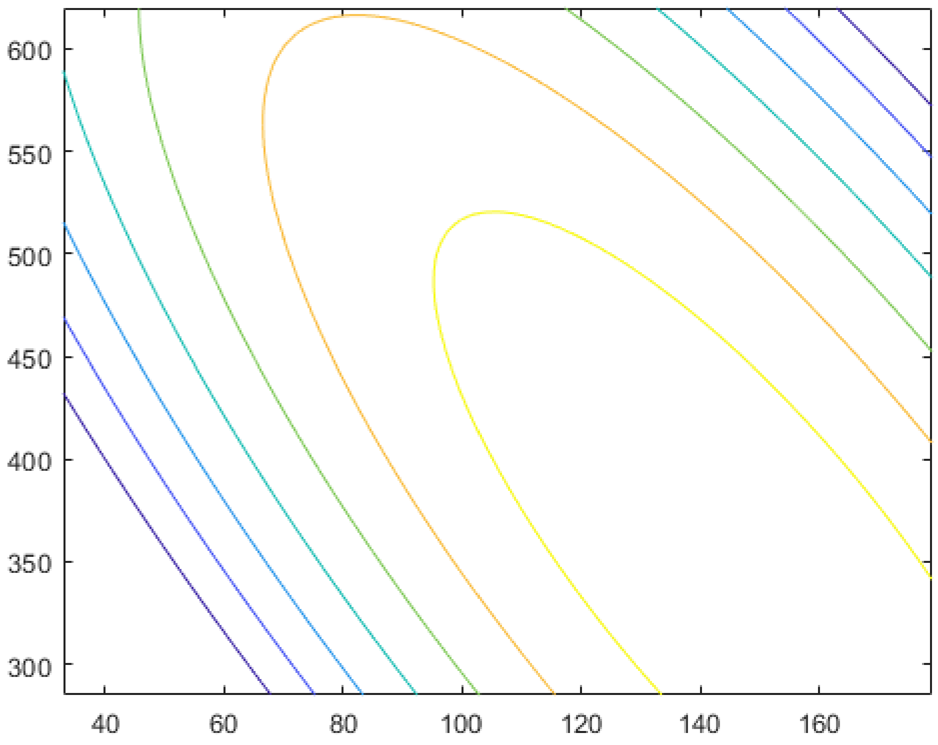
```
c2 = 6×1
         -1339.90488568954
          17.1883973493066
          4.95723840977287
       -0.0372951053113031
       -0.0179199642155538
      -0.00333877203214028
int95 = 6×2
         -3269.18333399781          589.373562618741
          2.34025664610191          32.0365380525113
         -1.31208695454193          11.2265637740877
       -0.0702443913017601       -0.0043458193208462
       -0.0397719281370144       0.00393199970590674
      -0.00860910072975285        0.0019315566654723
res = 19×1
          19.3390597341336
          71.6254030419311
          1.70162083411185
         0.720015424926714
          30.8593786732861
          3.08123311450356
          -41.193900243456
         -50.5908496868752
          22.8976885737432
         -3.46942828393469
      ⋮
```

We can define the regression surface using the coefficients:

```
f = @(x, y) c(1) + c(2)*x + c(3)*y + c(4)*x.^2 + c(5)*x.*y + c(6)*y.^2;
```

Visualizing the model using a contour plot first:

```
figure(2);
cont = fcontour(f, [min(x), max(x), min(y), max(y)]);
cont.LevelList = 450:50:800; % contour lines between 450 and 800
```



3D visualization:

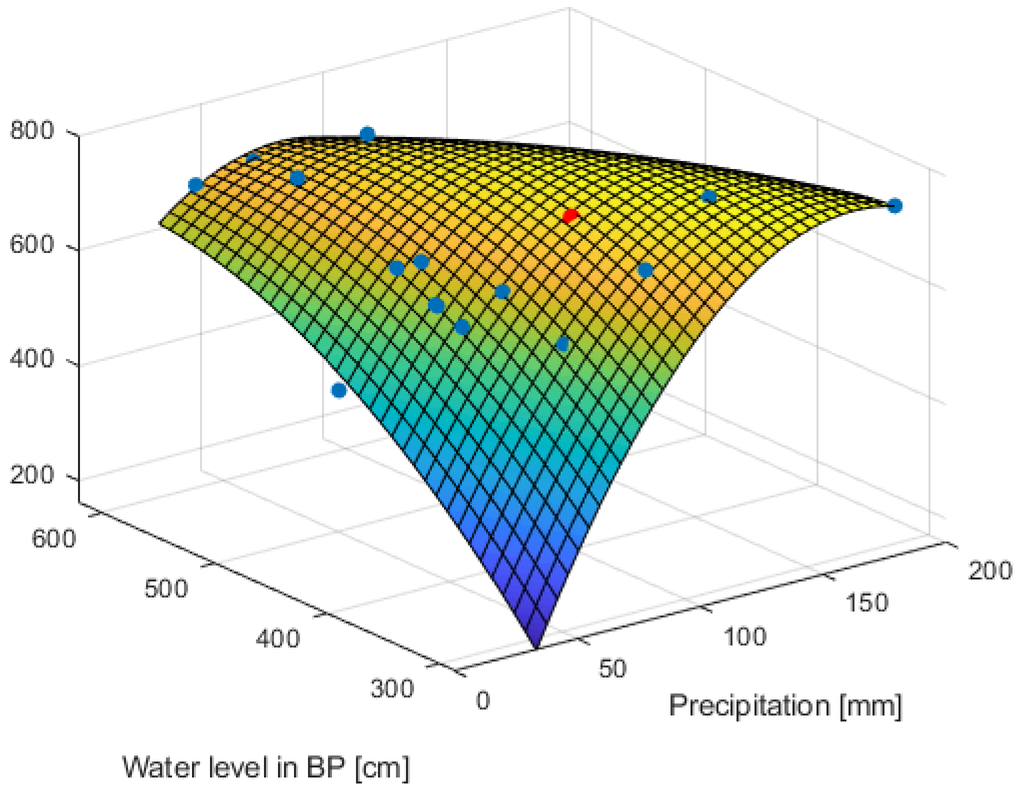
```
figure(1); hold on;
fsurf(f, [min(x), max(x), min(y), max(y)])
```

Forecasting the height of the flood wave at 100 mm of precipitation and 400 cm of water level:

```
xfw = 100;
yfw = 400;
flood_wave = f(xfw, yfw)
```

```
flood_wave = 
          737.877066272642
```

```
plot3(xfw, yfw, flood_wave, 'ro', 'MarkerFaceColor', 'r');
```



2D interpolation of data on an irregular grid

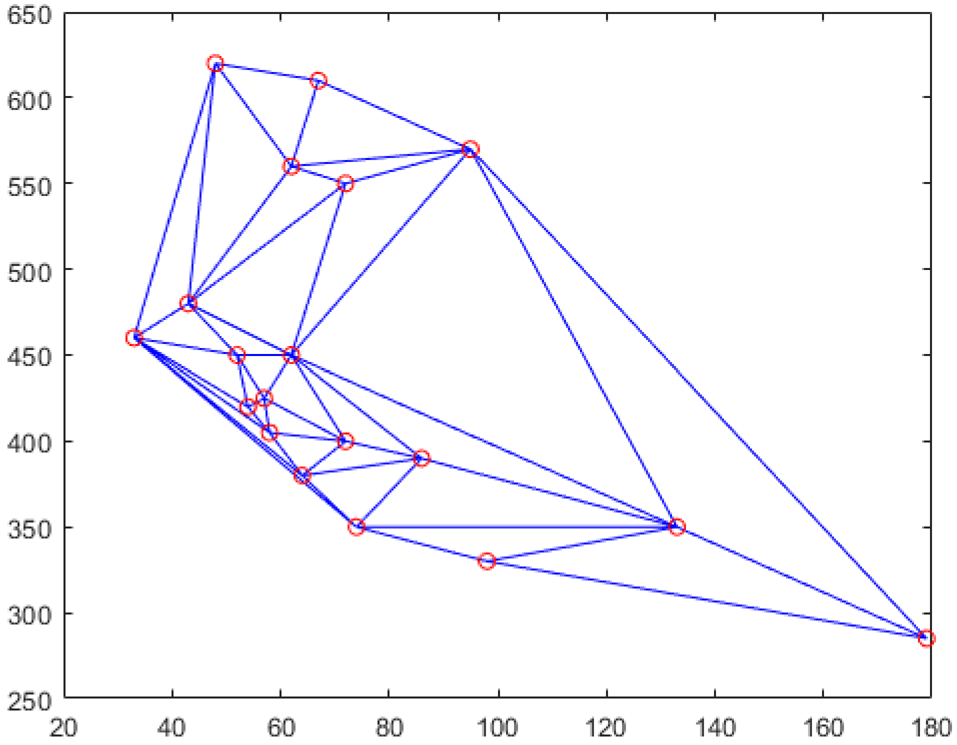
When interpolating data on an irregular grid, we have multiple methods to choose from:

- the nearest neighbor method, when the interpolated point gets the value of the closest data point,
- kriging, when the distance from data point gets taken into account with some weighting,
- radial basis functions, when multiple basis functions are fitted to the data points,
- division of the irregular grid with simpler elements.

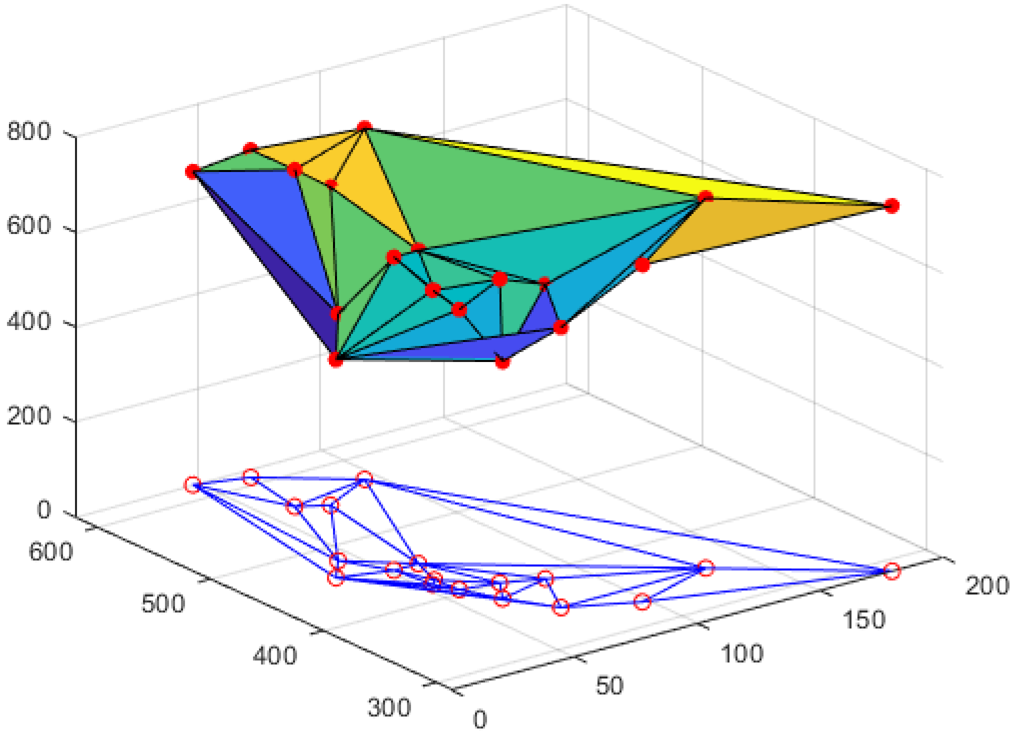
In MATLAB, the built-in **`griddata`** function uses the last method, the so-called Delaunay triangulation. The method creates triangles between the data points in such a way that if we drew the circumcircle for each triangle, no data points would be inside the circles. When choosing the `linear` option for the **`griddata`** command, the surface of the triangles are used for the interpolation. The **`griddata`** command can also use other types of interpolations, such as the `nearest` (nearest neighbor), or the `cubic` (bicubic spline).

The following two figures show the Delaunay triangulation of the data points from the previous example.

```
tri = delaunay(x, y); % creates a Delaunay triangulation from the data points

figure();
triplot(tri, x, y); hold on; % plot of the triangles
plot(x, y, 'ro');
```



```
figure();
trisurf(tri, x, y, z); % 3D plot of the triangles
hold on;
scatter3(x, y, z, 'ro', 'filled')
plot(x, y, 'ro');
triplot(tri, x, y);
```



How much is the predicted height of the flood wave if the precipitation was 100 mm and the water level was 400 cm, using linear and cubic interpolation?

In order to answer the question, we first have to define the interpolating functions:

```
zlin = @(u, v) griddata(x, y, z, u, v, 'linear');
zcub = @(u, v) griddata(x, y, z, u, v, 'cubic');
```

Using the linear interpolation, the answer is:

```
flood_lin = zlin(100, 400)
```

```
flood_lin =
          724.737732656514
```

From the cubic interpolation, we get:

```
flood_cubic = zcub(100, 400)
```

```
flood_cubic =
          735.393860996604
```

Using the regression polynomial, the answer is ~738 cm.

In order to better see the difference between the two types of interpolations, we can plot the corresponding surfaces. First, we create 1000 points between the smallest and largest x and y values and then we use these vectors to create a mesh grid.

```
xv = linspace(min(x), max(x), 1000);
yv = linspace(min(y), max(y), 1000);
[xq, yq] = meshgrid(xv, yv);
```

We can query the value of the interpolation at each of these points. By default, the **`griddata`** function does not extrapolate, so if our query point is out of the convex perimeter of the dataset, we get a NaN (not a number) value.

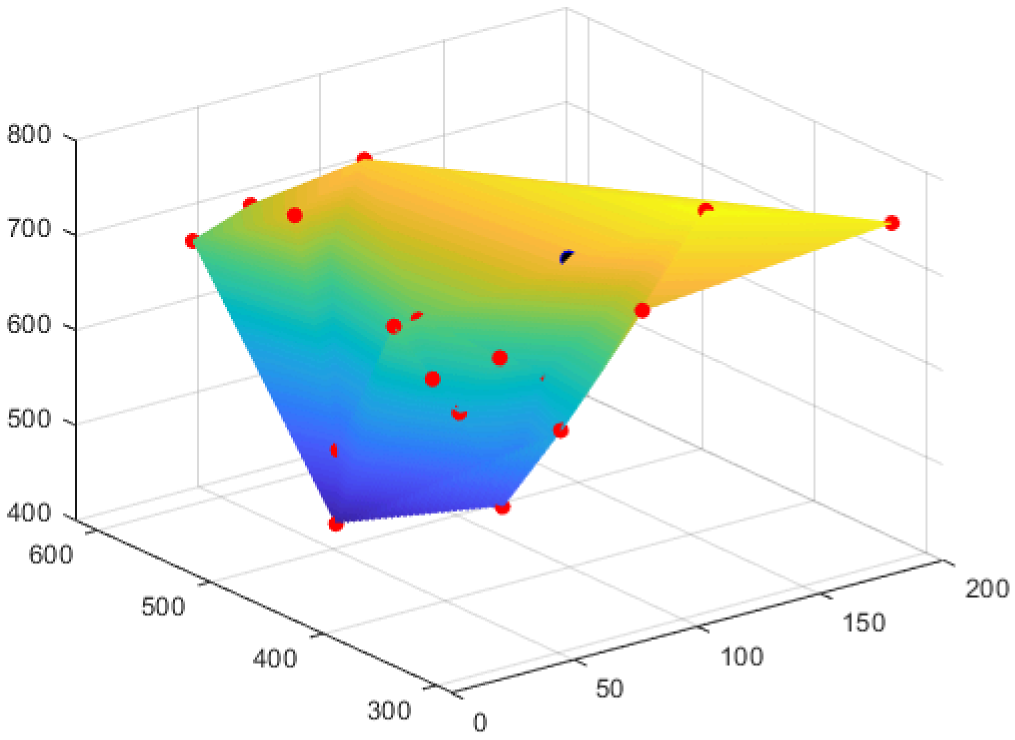
```
zql = zlin(xq, yq)
```

```
zql = 1000×1000
    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN ⋯
    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
      ⋮
```

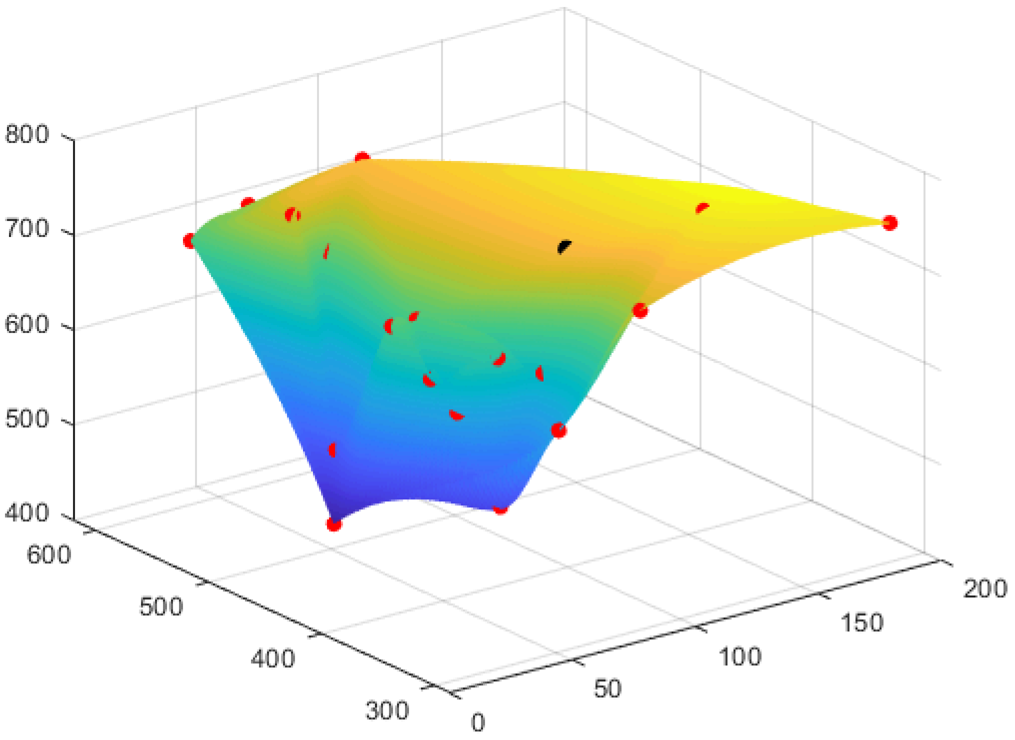
```
zqc = zcub(xq, yq);
```

Now, using the new variables, we can plot the interpolating surfaces. In case of the linear interpolation, we get the same surface as in the Delaunay triangulation plot:

```
% Linear interpolation
figure();
scatter3(x, y, z, 'ro', 'filled');
hold on;
mesh(xq, yq, zql);
plot3(100, 400, flood_lin, 'bo', 'MarkerFaceColor', 'k');
```



```
% Cubic interpolation
figure();
scatter3(x, y, z, 'ro', 'filled');
hold on;
mesh(xq, yq, zqc);
plot3(100, 400, flood_cubic, 'ko', 'MarkerFaceColor', 'k');
```


800
700
600
500
400
600
500
400
300
0
50
100
150
200

Numerical differentiation

Differentiation of a quantity makes us able to track its change with respect to some variable. The most common example is the calculation of velocity from the function of position. If the position is given as a function $x = f(t)$ of time, the velocity is calculated as the derivative of this function

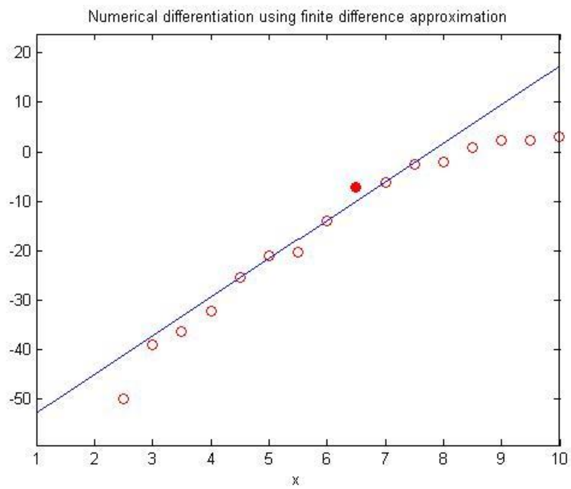
$$v = \frac{df(t)}{dt}$$

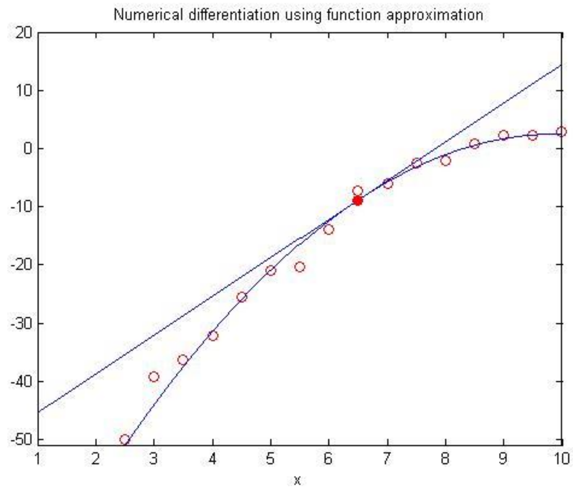
and the acceleration can be calculated as the second derivative of the velocity function:

$$a = \frac{dv(t)}{dt} = \frac{d^2 f}{dt}.$$

The derivative can also be used to find the minimum or maximum of a function. Analytic (symbolic in MATLAB) derivation can be used when the function can be given in analytic form that is somewhat easy to differentiate. If the function is given as a set of data points or if it is not possible to differentiate it analytically, numeric differentiation is the only option. Numeric differentiation also plays a very important role in solving differential equations.

If the function is acquired as a set of data points, the differential can be approximated by finite differences or another option can be to first approximate the function using some form of interpolation or regression and then differentiate this resulting analytic (and mostly simple) function.





Finite difference approximation

Let's suppose that we can only acquire the function values in a limited set of data points. In such a case, we can approximate the differential by the slope of the lines connecting the points. This can be done in two ways:

- using the right side or forward difference:

$$f'(x_i) = y'_i \approx \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

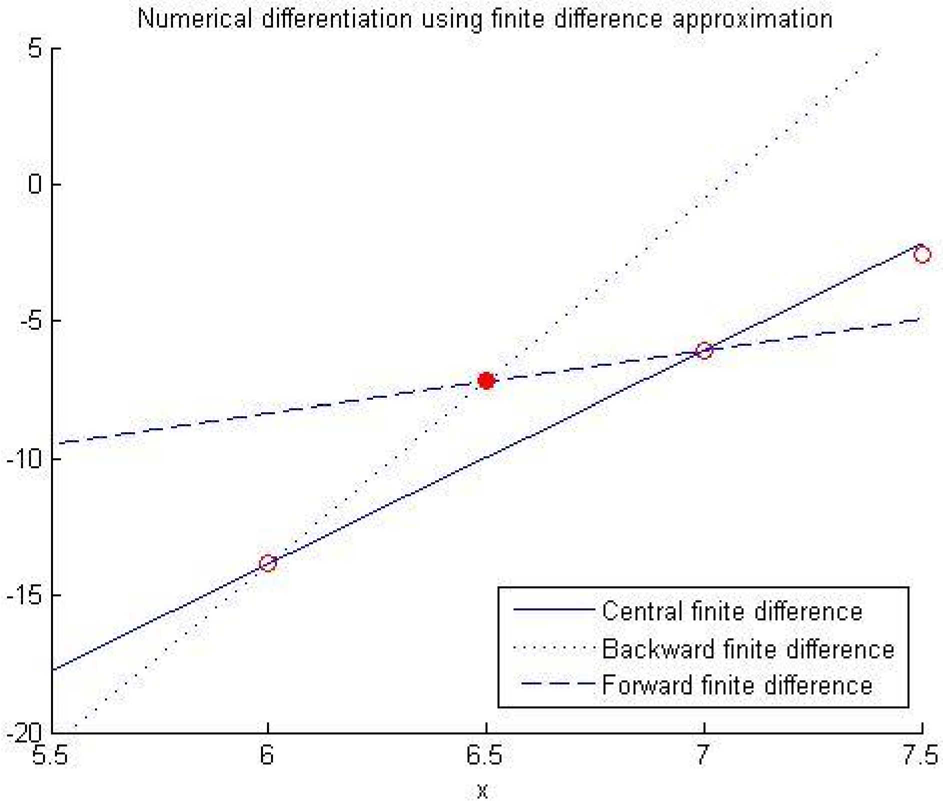
- using the left side of backward difference:

$$f'(x_i) = y'_i \approx \frac{y_i - y_{i-1}}{x_i - x_{i-1}}$$

As the errors of the forward and backward differences usually have the opposite sign, it is better to take the average of the two. If the spacing of the data points is homogeneous and $x_{i+1} - x_i = x_i - x_{i-1} = h$, this will result in the central difference:

$$f'(x_i) = y'_i \approx \frac{y_{i+1} - y_{i-1}}{x_{i+1} - x_{i-1}} = \frac{y_{i+1} - y_{i-1}}{2h}$$

Generally, the central difference is a better approximation of the derivative then the other two. As the spacing of the points becomes smaller, the accuracy further increases.



Errors of finite difference approximations

Taylor series can be used to approximate the truncation errors corresponding to the forward and backward differences:

$$f(x+h) = f(x) + h \cdot f'(x) + \frac{h^2}{2} \cdot f''(c)$$

where c is an unknown value between x and $x+h$. Let $x = x_i$ and $x+h = x_{i+1}$ and solve the equation for $f'(x)$:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} - \frac{h}{2} \cdot f''(c)$$

The formula above is the one used to calculate the forward difference with an extra subtraction on the left side, the error term $-\frac{h}{2} \cdot f''(c)$. That is, the magnitude of the truncation error is $\mathrm{O}(h)$ (big Ordo *h*). If we change h to $-h$ in the formula, we get the truncation error of the backward difference. The error of the central difference can be computed using third order Taylor series:

$$f(x_{i+1}) = f(x_i + h) = f(x_i) + h \cdot f'(x_i) + \frac{h^2}{2} \cdot f''(x_i) + \frac{h^3}{3!} \cdot f'''(c_1)$$

$$f(x_{i+1}) = f(x_i - h) = f(x_i) - h \cdot f'(x_i) + \frac{h^2}{2} \cdot f''(x_i) - \frac{h^3}{3!} \cdot f'''(c_2)$$

where $x_i < c_1 < x_{i+1}$ and $x_{i-1} < c_2 < x_i$. Subtract the two equations from each other and solve it for $f'(x_i)$:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h} - \frac{h^2}{3!} \cdot \frac{f'''(c_1) + f'''(c_2)}{2}$$

which shows that the truncation error of the central difference is $\mathrm{O}(h^2)$, which results in a much better approximation if h can be reduced.

We can approximate the differential using even more points, using 5 points for example gives us a central formula with $\mathrm{O}(h^4)$

$$f'(x_i) = y_i' = \frac{y_{i-2} - 8y_{i-1} + 8y_{i+1} - y_{i+2}}{12h} + \mathrm{O}(h^4)$$

The forward and backward difference can also be made better by involving more points. Using three points, the formulas for the forward and the backward differences become:

$$f'(x_i) = y_i' \approx \frac{-3y_i + 4y_{i+1} - y_{i+2}}{2h} + \mathrm{O}(h^2)$$

$$f'(x_i) = y_i' \approx \frac{y_{i-2} - 4y_{i-1} + 3y_i}{2h} + \mathrm{O}(h^2)$$

Higher order differentials

Similar numeric formulae can be derived for higher order derivatives as well. For example, the second derivative using central difference and 3 points:

$$f''(x_i) = y_i'' \approx \frac{y_{i+1} - 2y_i + y_{i+1}}{h^2} + \mathrm{O}(h^2)$$

Central differences for higher order derivatives will always have an error term that is proportional to h^2.

The second order derivatives using forward and backward differences:

$$f''(x_i) = \frac{y_i - 2y_{i+1} + y_i}{h^2} + \mathrm{O}(h)$$

$$f''(x_i) = \frac{y_{i-2} - 2y_{i-1} + y_i}{h^2} + \mathrm{O}(h)$$

How can these formulae be derived? Let's take a look at the 3 point forward difference formula. We can start from the Taylor series of the function at the three data points used (y_i, y_{i+1}, y_{i+2}):

$$y_i = f(x_i) = f(x_i) + 0 \cdot h \cdot f'(x_i) + 0 \cdot \frac{h^2}{2!} \cdot f''(x_i) + \mathrm{O}(h^3)$$

$$y_{i+1} = f(x_i + h) \approx f(x_i) + h \cdot f'(x_i) + \frac{h^2}{2!} \cdot f''(x_i) + \mathrm{O}(h^3)$$

$$y_{i+2} = f(x_i + 2 \cdot h) \approx f(x_i) + 2 \cdot h \cdot f'(x_i) + \frac{4 \cdot h^2}{2!} \cdot f''(x_i) + \mathrm{O}(h^3)$$

The above is a linear system of equations with 3 unknowns and 3 equations. We need to figure out a linear combination of the rows that cancel out all the term except the second derivatives. If we denote the coefficients (the multiplicators) of the lines as c_1, c_2, c_3, we can write:

$$c_1 \cdot y_i + c_2 \cdot y_{i+1} + c_3 \cdot y_{i+2} = 0$$

$$c_1 \cdot 0 \cdot y_i + c_2 \cdot 1 \cdot y_{i+1} + c_3 \cdot 2 \cdot y_{i+2} = 0$$

$$c_1 \cdot 0 \cdot y_i + c_2 \cdot \frac{1}{2} \cdot y_{i+1} + c_3 \cdot 2 \cdot y_{i+2} = 1$$

The first row corresponds to the first column of the previous system (the column for $f(x_i)$), the second row to the column containing $f'(x_i)$ and the third row to the column containing $f''(x_i)$. As in the end we only need the second derivative, we equate every row except the last with 0. By making the last row equal to one, we find a combination which will results in $1 \cdot f''(x_i)$ (+ the error term).

(Similarly, if we wanted to find the approximation of the first derivative, we would need to make the second row - and only the second row - equal to 1.)

Writing the system using matrix notation:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 2 \\ 0 & \frac{1}{2} & 2 \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

If we solve this using Gauss elimination for example and find the values of the coefficients, we get: $c_1 = 1,\ c_2 = -2,\ c_3 = 1$. In other words:

$$1 \cdot y_i - 2 \cdot y_{i+1} + 1 \cdot y_{i+2} = 0 \cdot f(x_i) + 0 \cdot h \cdot f'(x_i) + 1 \cdot h^2 \cdot f''(x_i) + \mathrm{O}(h^3) = h^2 \cdot f''(x_i) + \mathrm{O}(h^3)$$

$$f''(x_i) \approx \frac{y_i - 2y_{i+1} + y_{i+2}}{h^2} + \mathrm{O}(h)$$

(The sign of the $\mathrm{O}(h)$ is arbitrary as it only symbolizes the proportion of the error term to the step size h.)

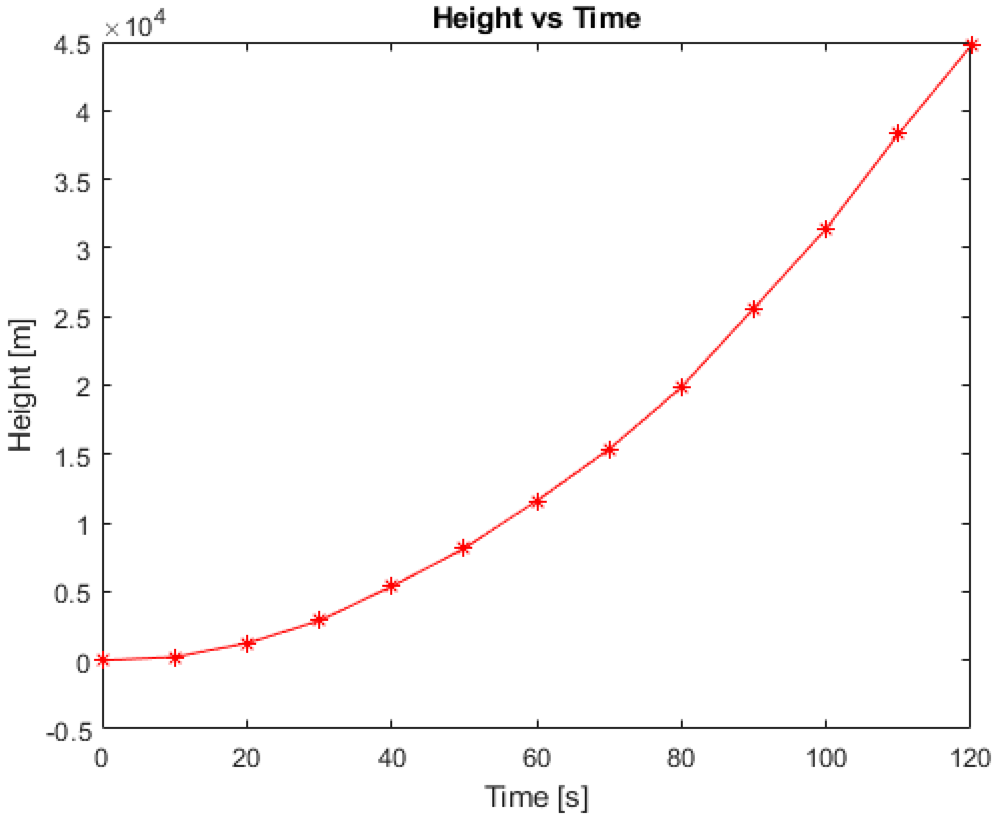
Numeric differentiation is practice

The following table contains the launch data of the space shuttle:

t(s)	0	10	20	30	40	50	60	70	80	9
h(m)	-8	241	1244	2872	5377	8130	11617	15380	19872	256

The task is to find the velocity function of the shuttle as a function of time. We can load the data from the **shuttle.txt** file and plot the data first:

```
data = load('shuttle.txt');
t = data(:, 1);
h = data(:, 2);
figure(1);
plot(t, h, 'r*-');
title('Height vs Time');
xlabel('Time [s]');
ylabel('Height [m]');
```

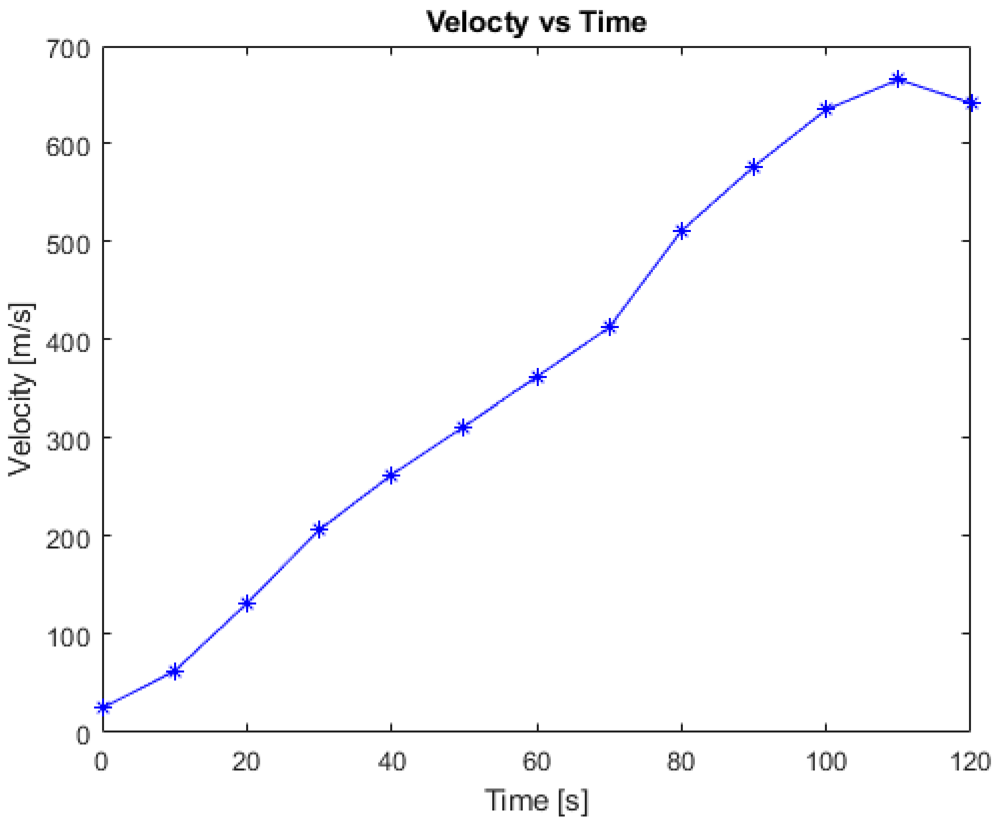


The velocity function is the first derivative of the position function (height in our case). The central difference formula can only be used from the second point to the next to last point, as it takes into account the point to the left and to the right of the current data point as well. In the first point, we can only use a forward difference and in the last point only a backward difference. Let's create a custom function that takes care of this calculation for us:

```
function dx = derivative(x, y)
% Numeric differentiation using finite differences

    n = length(x);
    dx(1) = (y(2)-y(1))/(x(2)-x(1)); % forward difference

    for i = 2:n-1
        dx(i) = (y(i+1)-y(i-1))/(x(i+1)-x(i-1)); % central difference
    end

    dx(n) = (y(n)-y(n-1))/(x(n)-x(n-1)); % backward difference

end
```

This function is located in the **`derivative.m`** file. Let's use it to calculate the velocity function and plot its graph:

```
v = derivative(t, h);
figure(2);
plot(t, v, 'b*-');
xlabel('Time [s]');
ylabel('Velocity [m/s]');
title('Velocty vs Time');
```

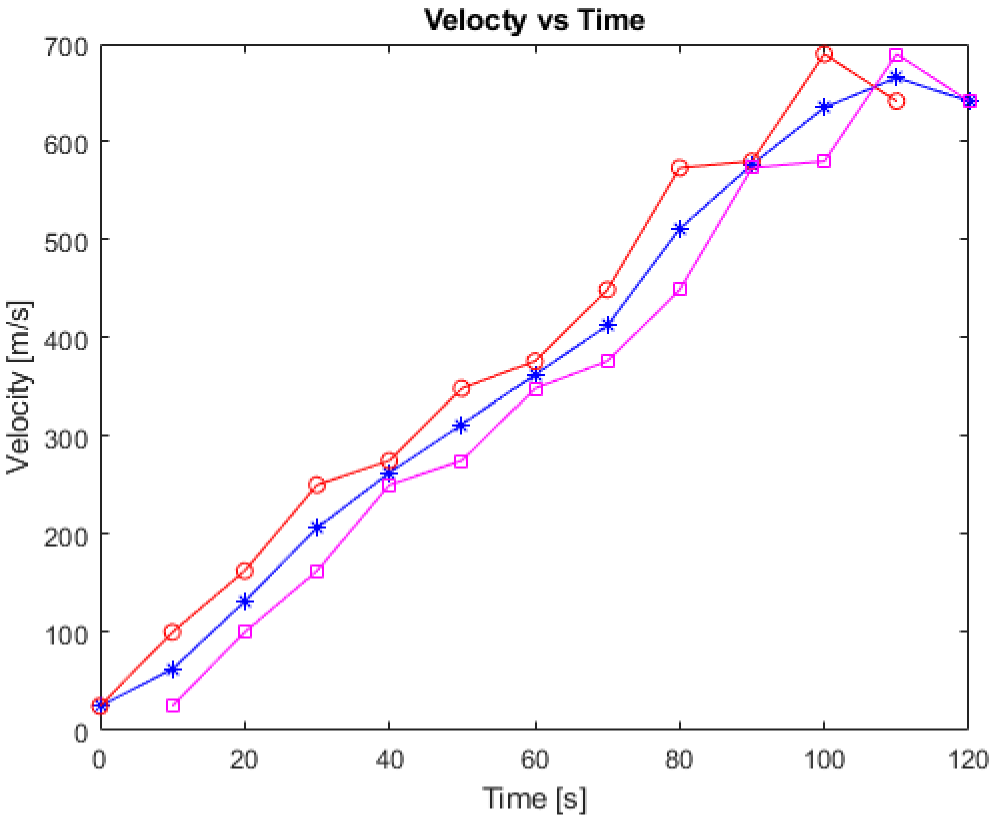


The built-in **diff** command can also be used to calculate the finite differences. We can first compute the differences in time and height and divide the two to get the finite differences for approximating the derivative:

```
dh = diff(h);
dt = diff(t);
dhdt = dh./dt;
```

The calculation of the forward and backward differences are essentially the same. The forward differences can be computed until the next to last point, while the backward differences can be computed from second point:

```
figure(2);
hold on;
plot(t(1:end-1), dhdt, 'ro-');
plot(t(2:end), dhdt, 'ms-');
```



From the figure, we can see that the central differences give a somewhat smoother curve that the other two.

The applied **derivative.m** function approximated the derivative from the second point to the next to last point an error of $\mathrm{O}(h^2)$, while the error is $\mathrm{O}(h)$ in the first and the last point. We could make the approximation more accurate by using 3 point approximations in the first and the last point as well, giving an error of $\mathrm{O}\left(h^2\right)$. Using the previously mentioned formula for the 3 point forward and backward differences:

```
v1 = (-3*h(1) + 4*h(2) - h(3))/(t(3) - t(1))
```

```
v1 = -12.8000
```

```
vn = (h(end-2) - 4*h(end-1) + 3*h(end))/(t(end) - t(end-2))
```

```
vn = 617.7000
```

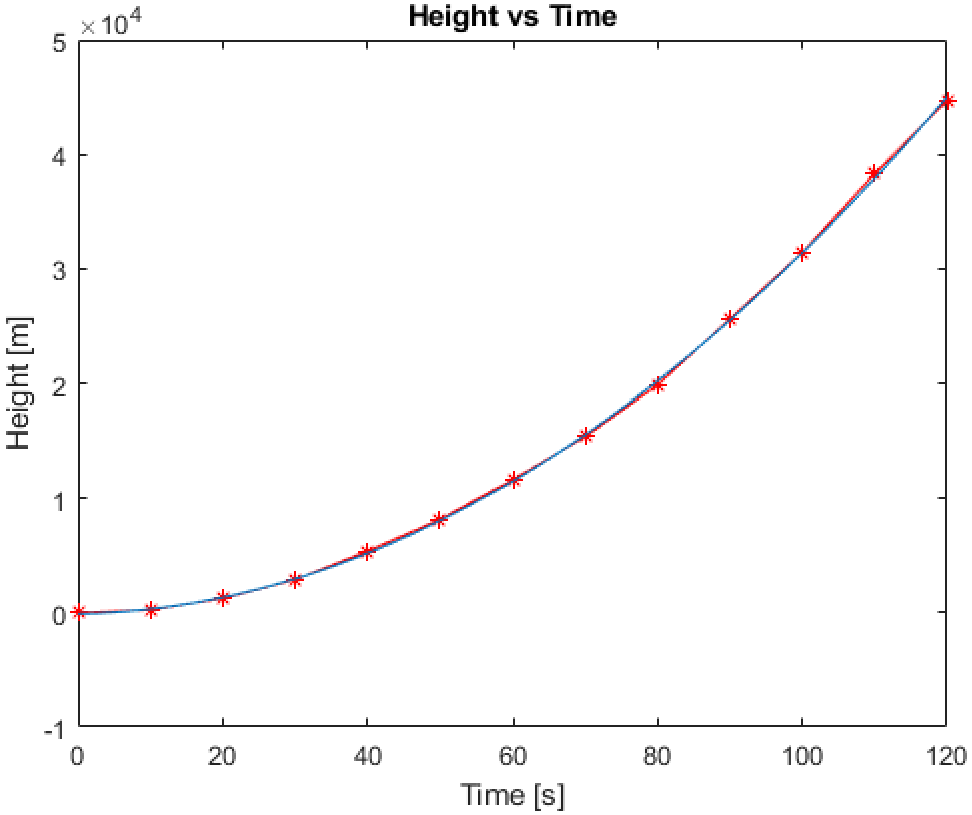
Numeric differentiation by function approximation

Let's look at the other approach for approximating the derivative. In this case, we first fit a function to the data points, preferably one that can be differentiated easily, and we compute the analytic derivative of that function. Let's use a quadratic polynomial as our model:

```
c = polyfit(t, h, 2)
```

```
c = 1×3
    3.0470   10.1151  -89.3626
```

```
p = @(x) polyval(c, x);
figure(1);
hold on;
fplot(p, [min(t), max(t)]);
```



We can see that the quadratic polynomial fits the data really well.

If we defined the polynomial in symbolic form, we could use the same **`diff`** command to differentiate the function symbolically. In the case of polynomials, we can also use the built-in **`polyder`** command and if we use this, we don't even have to define the polynomial symbolically. Let's take a look at both solutions:

```
syms x
ps = c(1)*x.^2 + c(2)*x + c(3)
```

```
ps =
```

$$\frac{3050\,x^2}{1001}+\frac{50626\,x}{5005}-\frac{6288336567612549}{70368744177664}$$

```
v2 = diff(ps, x) % using symbolic derivation
```

```
v2 =
```

$$\frac{6100\,x}{1001}+\frac{50626}{5005}$$

```
v2 = matlabFunction(v2)
```

```
v2 = function_handle with value:
    @(x)x.*6.093906093906094+1.011508491508492e1
```

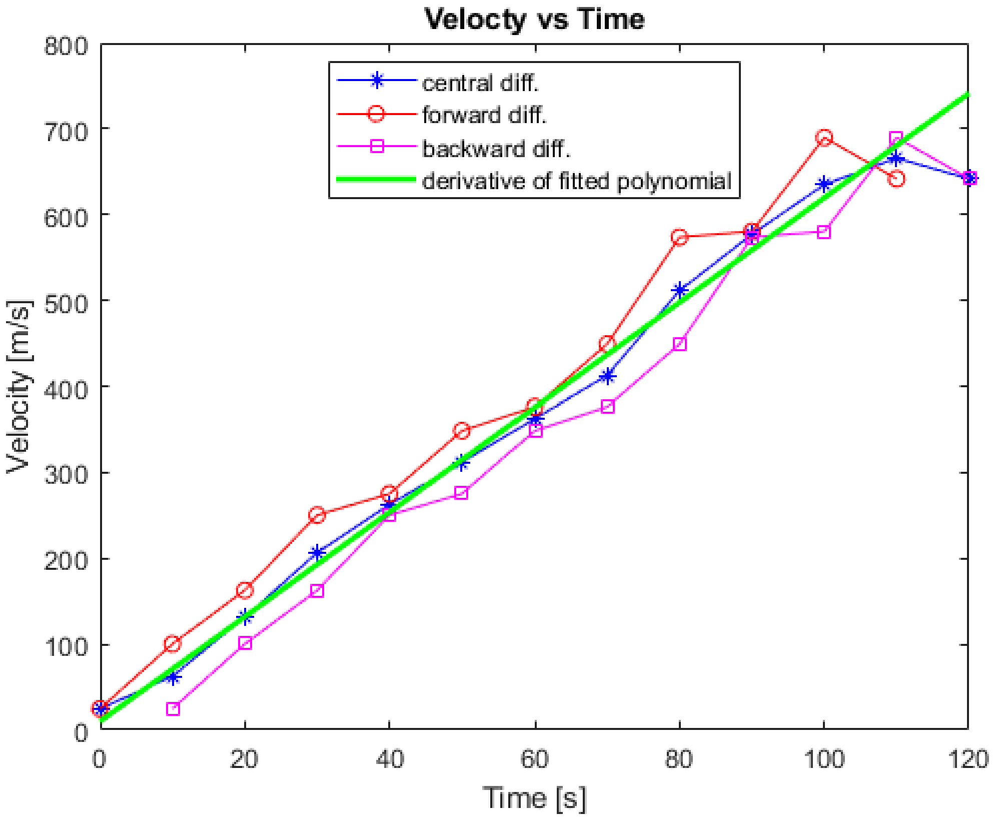
```
c1 = polyder(c) % using the polyder command which takes the coefficient of the original polynom
```

```
c1 = 1×2
    6.0939   10.1151
```

```
v2 = @(x) polyval(c1, x)
```

```
v2 = function_handle with value:
    @(x)polyval(c1,x)
```

```
figure(2);
fplot(v2, [min(t), max(t)], 'g', 'LineWidth', 2);
legend('central diff.', 'forward diff.', 'backward diff.', 'derivative of fitted polynomial',
```



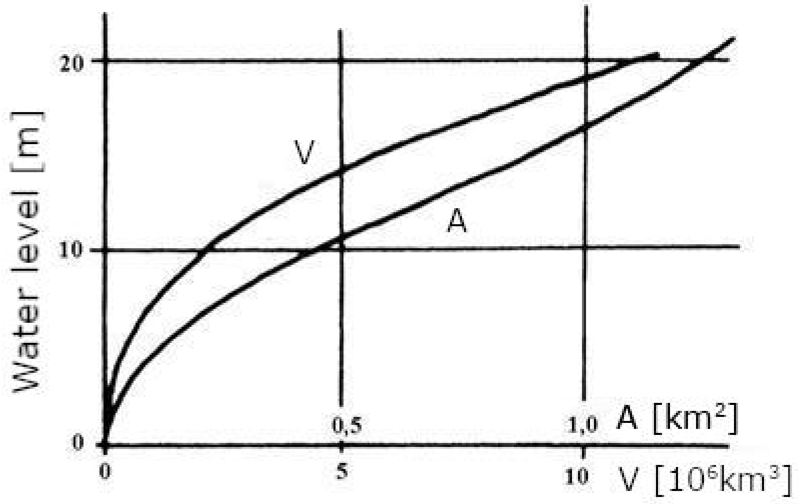
The derivative function became even smoother, resulting in a line.

Example - Evaporation and leakage losses of irrigation water reservoir

Let's look at the following example. We have a water storage used for irrigation that suffers losses due to evaporation and leakage. Our task is to approximate the quantities of these losses ($Q_{\text{loss}}(t)\ [m^3/s]$) as a function of time if we know the value of the inlet ($Q_{\text{in}}(t)\ [m^3/s]$) the value of the outlet (the water used for irrigation, $Q_{\text{out}}(t)\ [m^3/s]$), the level of water inside the reservoir ($z(t)$) and the characteristic curve of the reservoir. The inlet, the outlet and the water level are all given as time series and are hence functions of time. The equation determining the volume of water inside the reservoir:

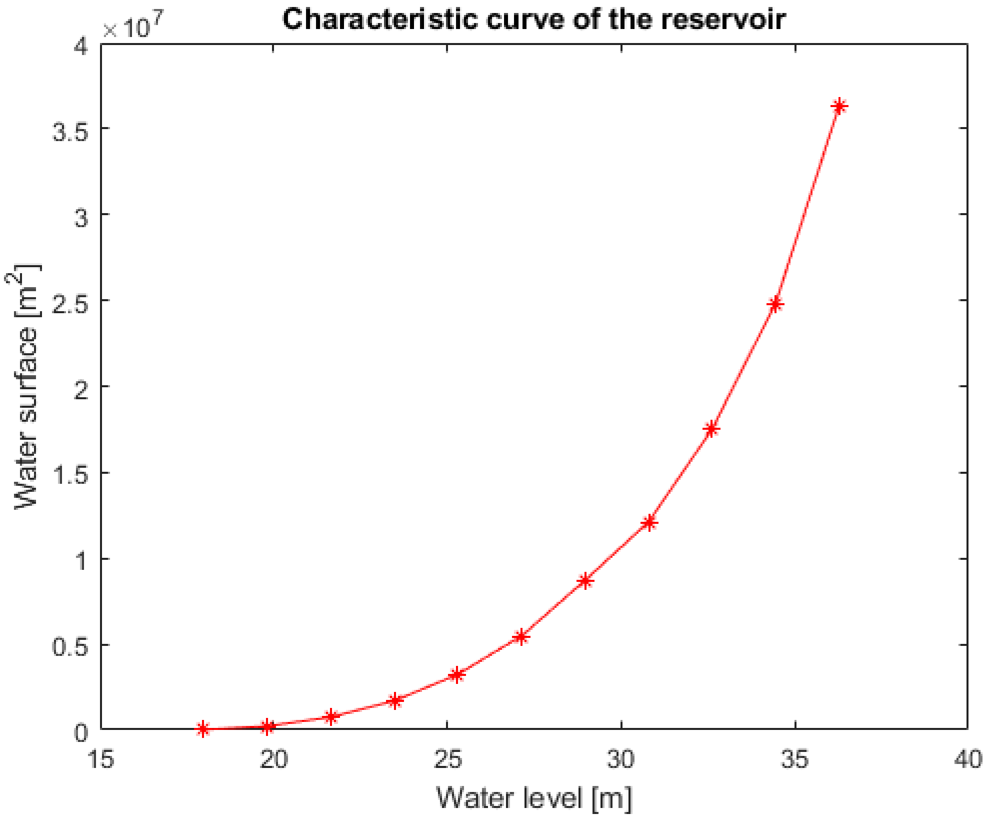
$$Q_{\text{in}}(t) - Q_{\text{out}}(t) - Q_{\text{loss}}(t) = A(z) \cdot \frac{dz}{dt}$$

where $\frac{dz}{dt}$ is the change in the water level and $A(z)$ is the area of the water inside the reservoir for a given water level (it can be determined from the characteristic curve of the reservoir):



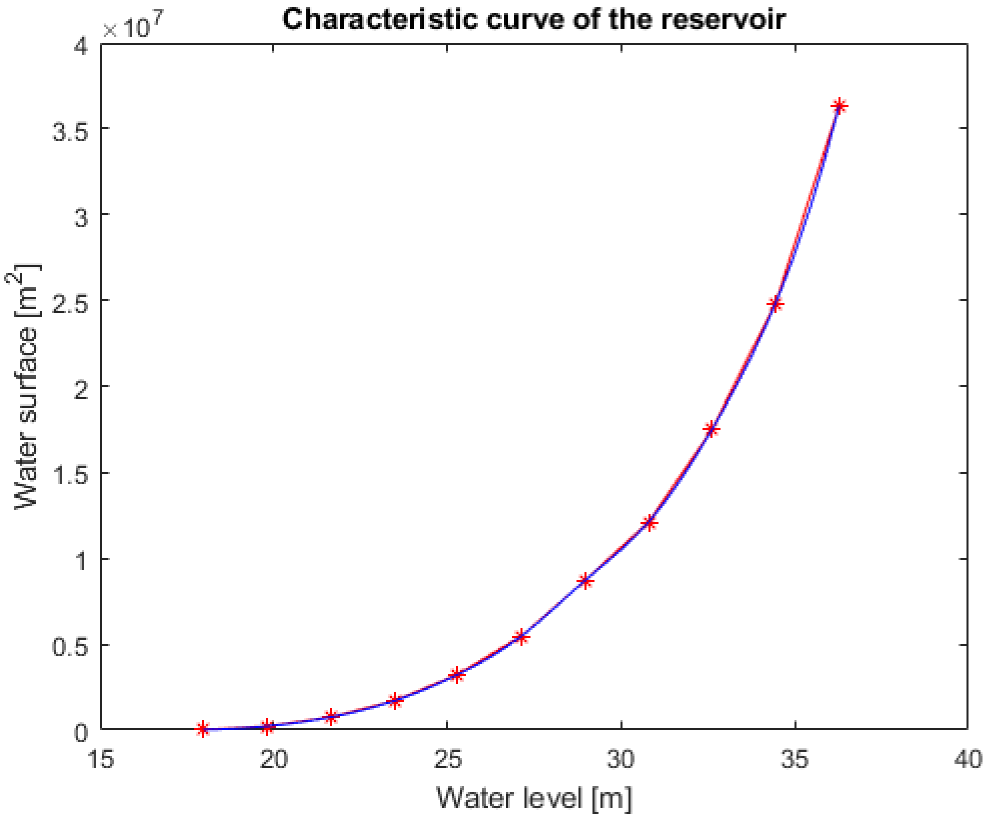
The figure above shows the characteristic curve of a reservoir for both water surface and the volume as function of the water level. The data for the $A(z)$ function for our reservoir is available in the **irrigation.txt** file. The water levels are given in m and the water surface values are given in m^2. Let's load the data and plot the characteristic curve.

```
clear all; close all;
data = load('irrigation.txt');
z = data(:, 1);
A = data(:, 2);
figure(1);
plot(z, A, 'r*-');
title('Characteristic curve of the reservoir');
xlabel('Water level [m]');
ylabel('Water surface [m^2]');
```



In the example, we will use cubic, second order splines to interpolate the data.

```
F = @(x) spline(z, A, x);
hold on;
fplot(F, [min(z), max(z)], 'b')
```

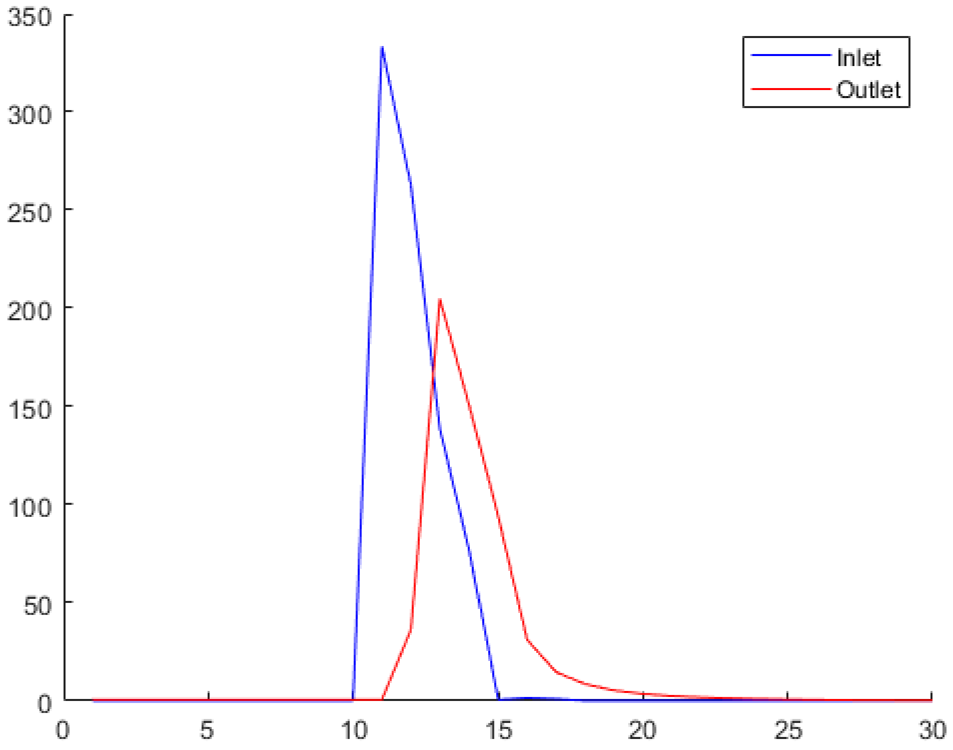


Data for the inlet ($Q_{\text{in}}(t)$), the water level ($z(t)$) and the outlet ($Q_{\text{out}}(t)$) are also available for a 30-day period in the **timeseries.txt**. The first column represents the number of the days, the second the inlet, the third the water level and the fourth the outlet. The water level is given in m, while the inlet and outlet values are given in m^3/s. Let's load this data into MATLAB as well:

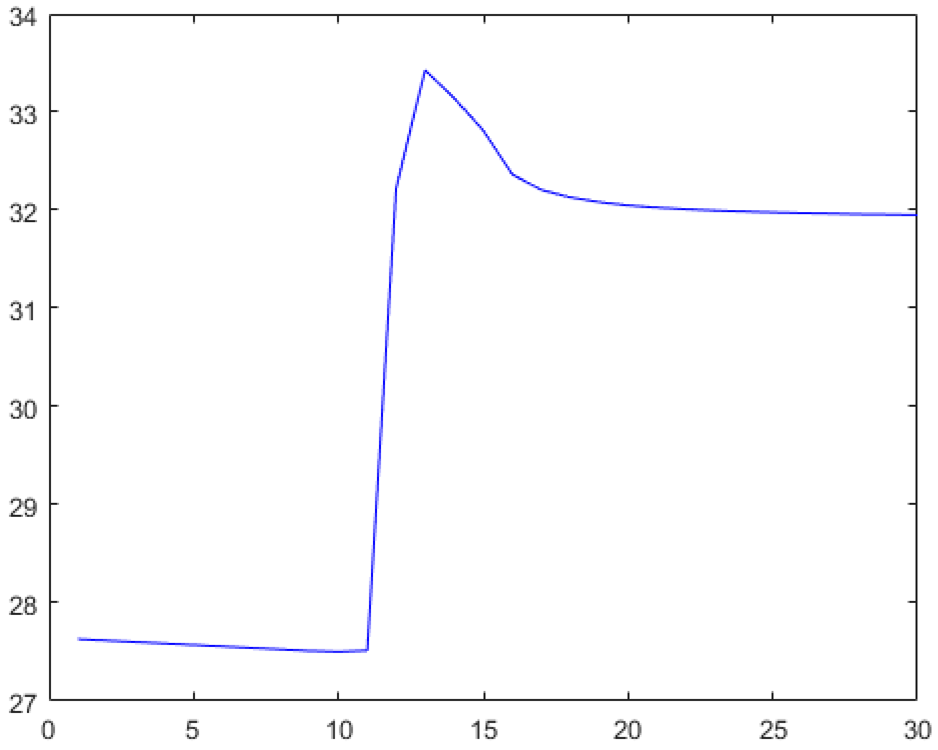
```
data = load('timeseries.txt');
t = data(:, 1);
Qin = data(:, 2);
z = data(:, 3);
Qout = data(:, 4);
```

We can create two plots showing the inlet/outlet and the change in the water level:

```
figure(2); hold on;
plot(t, Qin, 'b');
plot(t, Qout, 'r');
legend('Inlet', 'Outlet');
```



```
figure(3);
plot(t, z, 'b');
```



Using the data of the characteristic curve, let's compute the water surface values (A) at each water level value (z):

```
Az = F(z)
```

```
Az = 30×1
10^7  ×
    0.6312
    0.6288
    0.6261
    0.6235
    0.6209
    0.6183
    0.6157
    0.6131
    0.6105
    0.6090
      ⋮
```

Our equation for the reservoir is the following:

$$Q_{\text{in}}(t) - Q_{\text{out}}(t) - Q_{\text{loss}}(t) = A(z) \cdot \frac{dz}{dt}$$

We know the $Q_{\text{in}}(t)$, $Q_{\text{out}}(t)$ and $A(z)$ time series. In order to find the $Q_{\text{loss}}(t)$ function, we have to compute the first derivative of the water level, $\frac{dz}{dt}$. Because the water level values are measured at the beginning of each day, the effect of change in the water level during a day will only visible in the data for the next day. This means that we have to use forward differences. We can create a new custom function that calculates the derivative using only forward differences (one backward difference at the very last point), or we can use the **diff** command.

First, we convert the time values from day to second, as all the other values are given in $[\mathrm{m}^3/s]$:

```
ts = t*24*3600;
dz = diff(z)./diff(ts);
dz = [dz; dz(end)] % derivative using forward differences (the last value is copied)
```

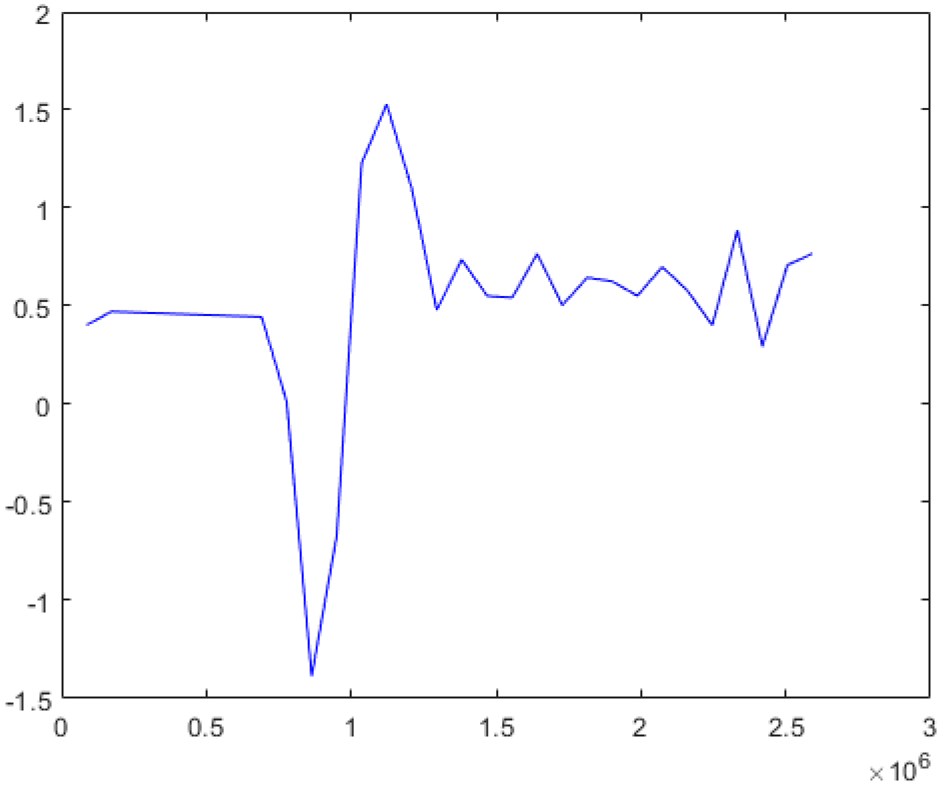
```
dz = 30×1
10^-4 ×
    -0.0016
    -0.0017
    -0.0017
    -0.0017
    -0.0017
    -0.0017
    -0.0017
    -0.0017
    -0.0010
     0.0014
       ⋮
```

Now, we can use the equation above to find the loss function:

```
dV = Az.*dz;
Qloss = Qin - Qout - dV;
```

Plotting the loss function:

```
figure(4);
plot(ts, Qloss, 'b-');
```



The evaporation (h_v) is proportional to the water surface and is usually given in mm/day. The values of the loss function are now in m^3/s, but we can change this to mm/day:

$$h_v = \frac{Q_{\mathrm{loss}} \cdot 86400 \cdot 1000}{A}$$

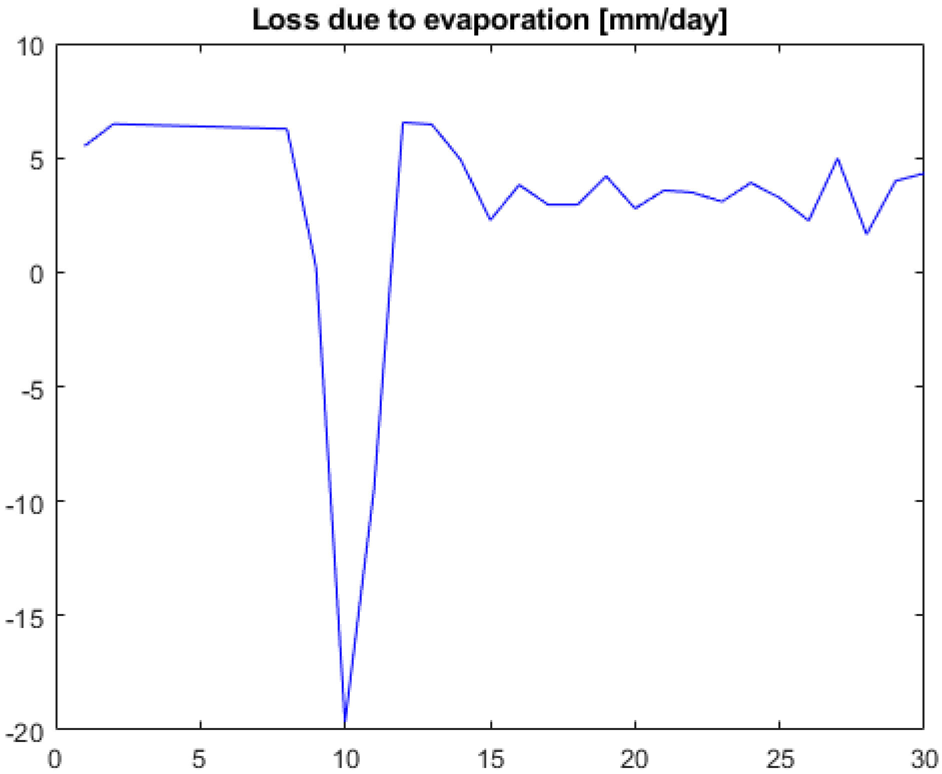
where 86400 is the number of seconds in a day and we multiply by 1000 to give the results in mm and not in m. Let's plot the transformed evaporation function (it's value is generally under 10 mm/day for our conditions and its value can be negative in case of precipitation):

```
evap_mm = Qloss * 86400 * 1000 ./ Az
```

```
evap_mm = 30×1
    5.5137
    6.4805
    6.4448
    6.4088
    6.3726
    6.3362
    6.2996
    6.2628
    0.2258
  -19.6615
      ⋮
```

```
figure(5);
plot(t, evap_mm, 'b');
```

```
title('Loss due to evaporation [mm/day]');
```



Numerical integration

Numerical integration is the approximation of the definite integral of a function $f(x)$. The definite integral $\int_a^b f(x)$ is equal to the area under the function's graph in the interval $[a, b]$. Some of the methods used to numerically approximate the integral will try to approximate this area under the function instead. Many engineering problems use integrals, the most common ones include area/volume calculation, arc length calculation or solving differential equation.

The trapezoidal rule

One of the most common numerical methods to approximate the integral is the trapezoidal rule. We take the function of the graph in an interval $[a, b]$ and approximate the area under the function using a trapezoid:

$$\int_a^b f(x)\,dx \approx \frac{f(a) + f(b)}{2} \cdot (b - a)$$

where $(b - a)$ is the length of the interval (the "height" of the trapezoid). As using just one big trapezoid is very inaccurate, we have to divide the interval into $N = n - 1$ (n being the number of dividing points in the interval) smaller intervals, calculate the area of the corresponding trapezoids and then sum up these areas:

$$\int_a^b f(x)\,dx \approx \frac{1}{2} \cdot \sum_{i=1}^{N} (f(x_i) + f(x_{i+1})) \cdot (x_{i+1} - x_i)$$

It is not necessary to have equally spaced data points, but if this condition is met, the above formula simplifies to:

$$\int_a^b f(x)\,dx \approx \frac{h}{2} \cdot \sum_{i=1}^{N} (f(x_i) + f(x_{i+1}))$$

where h is the distance between two data points along the x-axis.

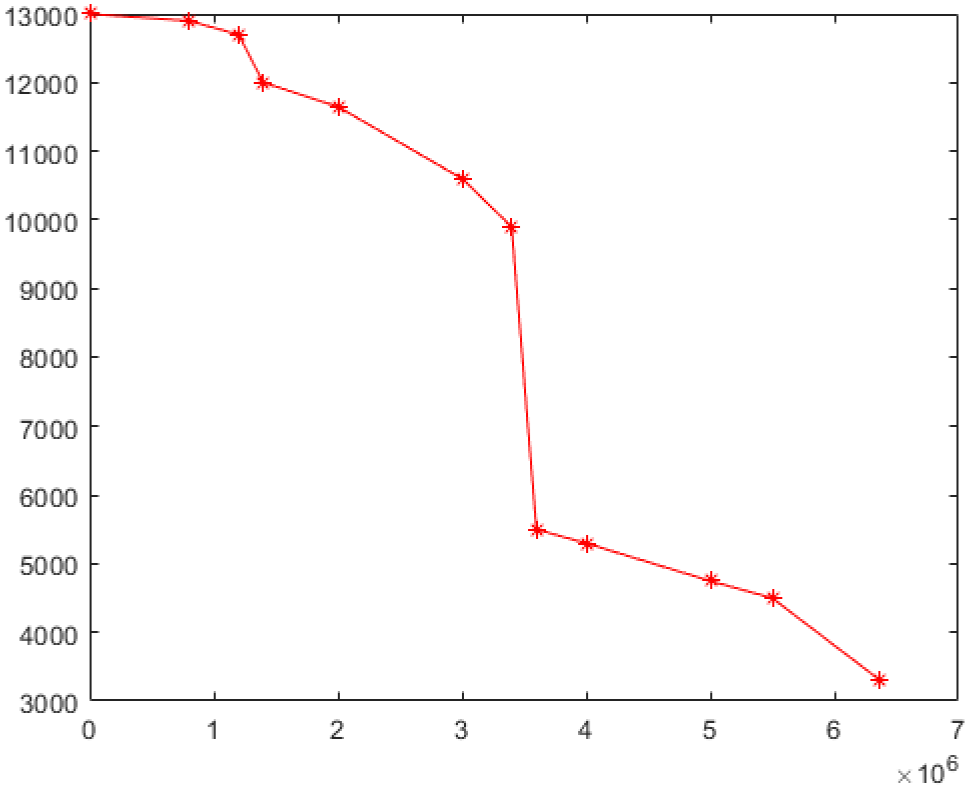
Let's take a look at the following example:

The density of the Earth (ρ) is a function of the distance from its center (the radius, R). The data points describing this relationship are located in the **`earth_density.txt`** file. Let's load the data and compute the approximate mass of the Earth using the following integral:

$$M_{\text{Earth}} = \int_0^{6370} \rho(R) \cdot 4 \cdot \pi \cdot R^2\,dR$$

The radii are given in km and the density values are given in $[\mathrm{kg/m^3}]$, which means that we have to change the radii into m as well:

```
data = load('earth_density.txt');
R = data(:, 1) * 1000;
rho = data(:, 2);
figure(1);
plot(R, rho, 'r*-');
```



First, we need to calculate the values of the function that we want to integrate, so let's compute the function values for the given R and ρ values:

```
fx = 4 * pi * rho .* R.^2;
```

MATLAB has a built-in command to approximate the integral using the trapezoidal rule,
the **`trapz`** command:

```
M = trapz(R, fx) % 1st argument is variable with respect to which we integrate, second
```

```
M =
      6.02609577351443e+24
```

```
% the function values
```

The approximation seems to be OK as the current estimate for the mass of the Earth is $(5.9722 \pm 0.0006) \cdot 10^{24}\,\mathrm{kg}$.

Numerical integration using Simpson's rule

The trapezoidal rule basically uses a line between the function values to approximate the function. Simpson's rule improves this approximation by using quadratic or cubic polynomials (Simpson's 1/3 rule, Simpson's 3/8 rule).

In case of the 1/3 rule, that uses quadratic polynomials, we compute the function values in 3 points and fit a degree 2 polynomial to the points. We can do this easily using the Lagrange or the Newton form. Let's use the Newton form is our case:

$$p(x) = a_1 + a_2(x - x_1) + a_3(x - x_1)(x - x_2)$$

The coefficients a_1, a_2, a_3 are as follows (computed using the divided differences):

$$a_1 = y_1 \qquad a_2 = \frac{y_2 - y_1}{x_2 - x_1} \qquad a_3 = \frac{\frac{y_3 - y_2}{x_3 - x_2} - \frac{y_2 - y_1}{x_2 - x_1}}{x_3 - x_1}$$

If we have equally spaced points, with a distance h between them:

$$a_1 = y_1 \qquad a_2 = \frac{y_2 - y_1}{h} \qquad a_3 = \frac{\frac{y_3 - y_2}{h} - \frac{y_2 - y_1}{h}}{2h} = \frac{y_3 - 2y_2 - y_1}{2h^2}$$

As the function is approximated using this polynomial, we approximate the integral of the function by the integral of the polynomial:

$$\int_{x_1}^{x_3} f(x)\,dx \approx \int_{x_1}^{x_3} p(x)\,dx = \frac{h}{3}\left(f(x_1) + 4f(x_2) + f(x_3)\right)$$

In a general form:

$$\int_{x_{i-1}}^{x_{i+1}} f(x)\,dx \approx \frac{h}{3}\left(f(x_{i-1}) + 4f(x_i) + f(x_{i+1})\right)$$

Let's have n equally spaced points in the interval $[a, b]$. This means that $x_1 = a$ and $x_n = b$. The n number of points divide the interval into $N = n - 1$ parts. Simpson's rule uses 3 point to fit the parabola to the function, so we always have to divide the integral into even number of parts:

$$\int_a^b f(x)\,dx = \int_{x_1=a}^{x_3} f(x)\,dx + \int_{x_3}^{x_5} f(x)\,dx + \cdots + \int_{x_{n-2=N-1}}^{x_n} f(x)\,dx = \sum_{i=2,4,6,\ldots}^{N} \int_{x_{i-1}}^{x_{i+1}} f(x)\,dx$$

The approximation using Simpson's rule:

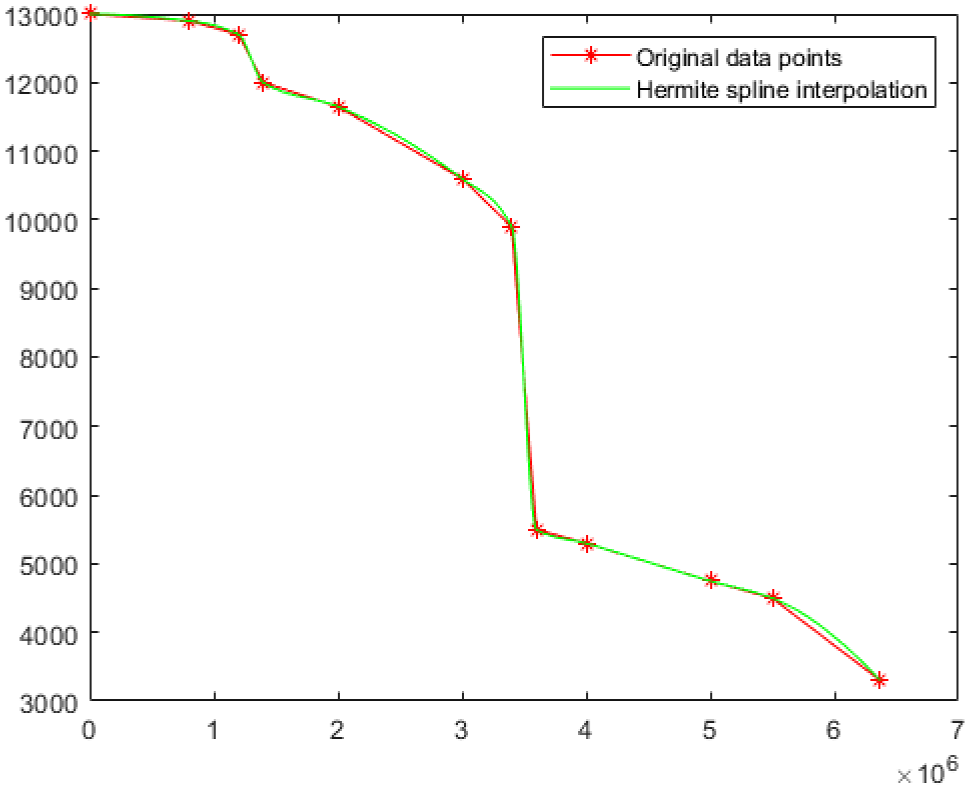
$$\int_a^b f(x)\,dx \approx \frac{h}{3} \cdot \left[f(a) + 4\sum_{i=2,4,6,\ldots}^{n-1} f(x_i) + 2\sum_{j=3,5,7,\ldots}^{n-2} f(x_j) + f(b) \right]$$

As an example, let's compute the mass of the Earth from the previous exercise using Simpson's rule as well. In MATLAB, we can use the **quad** function to estimate the integral using Simpson's rule. However, in this case, we have to supply a function as an input to the algorithm and not a vector of data points. This means, that we first have to approximate the data points using either a form of interpolation or some form of regression. Previously we have looked at this dataset and found that it is best approximated using cubic Hermite interpolation (only the first derivatives are equal at the internal points).

We can fit the Hermite spline using the **`interp1`** command and specifying the **`pchip`** method:

```
rho_cubic = @(x) interp1(R, rho, x, 'pchip');
```

```
figure(1);
hold on;
fplot(rho_cubic, [0, 6370000], 'g');
legend('Original data points', 'Hermite spline interpolation');
```



Now, we can use the **quad** command to estimate the integral:

```
fx_cubic = @(R) 4 * pi * rho_cubic(R) .* R.^2;
M2 = quad(fx_cubic, 0, 6370000)
```

```
Warning: Maximum function count exceeded; singularity likely.
M2 =
      5.96575465878443e+24
```

We can see that the estimation is a lot better, that is closer to the currently accepted value.

Remark: a method called Richardson's extrapolation can be used to further improve the estimation. The method uses the combination of two less accurate results to improve the approximation by one order of magnitude. Due to time constraints, we do not cover this in the class.

Estimation of multidimensional integrals or a regular grid

It is common to have two or three dimensional integrals in many engineering problems. A two dimensional definite integral is given in the following form:

$$I = \int_{y_1}^{y_2} \int_{x_1}^{x_2} f(x, y)\, dx\, dy$$

The computation of such an integral can be done in two parts, an inner and an outer integral. Numerically, we can start the computation of the outer integral with one of the methods above, but every function value inside results from a numerical estimation of the inner integral. This way, we can generalize the integral of a univariate function to higher dimensions (more variables), when the limits of the integrals can be given on a regular grid (rectangular area, volume etc.)

In MATLAB, if the integral has to be computed on a regular grid, we can use the **`integral2`** for 2D and the **`integral3`** for the estimation of 3D integrals:

```
q = integral2(fun, xmin, xmax, ymin, ymax)
q = integral3(fun, xmin, xmax, ymin, ymax, zmin, zmax)
```

Previously we have looked at an example where we used 2D integration and computed the volume below the terrain given on a regular grid.

If the limits of the integral are defined by an irregular shape, we have to use a different approach.

Multidimensional integral on irregular shapes

In such a case, we can use the so called **Monte Carlo method**. This method is stochastic, which means that it evaluates the function in random points as opposed to a regular grid. The method is most commonly used to estimate area or volume of irregular shapes.

Let's look at the following example. We have the area of a water reservoir defined by the coordinates of its vertices. The coordinates are located in the **`reservoir.txt`** file. First, let's load the data and plot the area:

```
clear all; close all;
data = load('reservoir.txt');
x = data(:, 1);
y = data(:, 2);
figure(1);
hold on;
plot(x, y, 'r-', 'LineWidth', 2);
axis equal
```

If we want to estimate the area of the reservoir using the Monte Carlo method, we have to do the following:

1. We need to define the rectangular area that covers the whole reservoir.
2. We have to generate N number of random points in this rectangular area.
3. We have to count how many points fall inside the area of the reservoir. Let this number be n. We can calculate the proportion of the points that are inside the area of the reservoir to the total number of points: $\rho = \frac{n}{N}$.
4. If we know the area of the rectangle containing the reservoir, let it be $T = a \cdot b$, then the estimated area of the reservoir is given by the following equation:

$$T_v = \int_T f(x)\ dT = \int_T 1\ dT \approx \rho \cdot T = \frac{n}{N} \cdot (a \cdot b)$$

Let's compute this in MATLAB. First, we have to define the rectangle containing the reservoir:

```
a = max(x) - min(x)
```

```
a = 
        6669
```

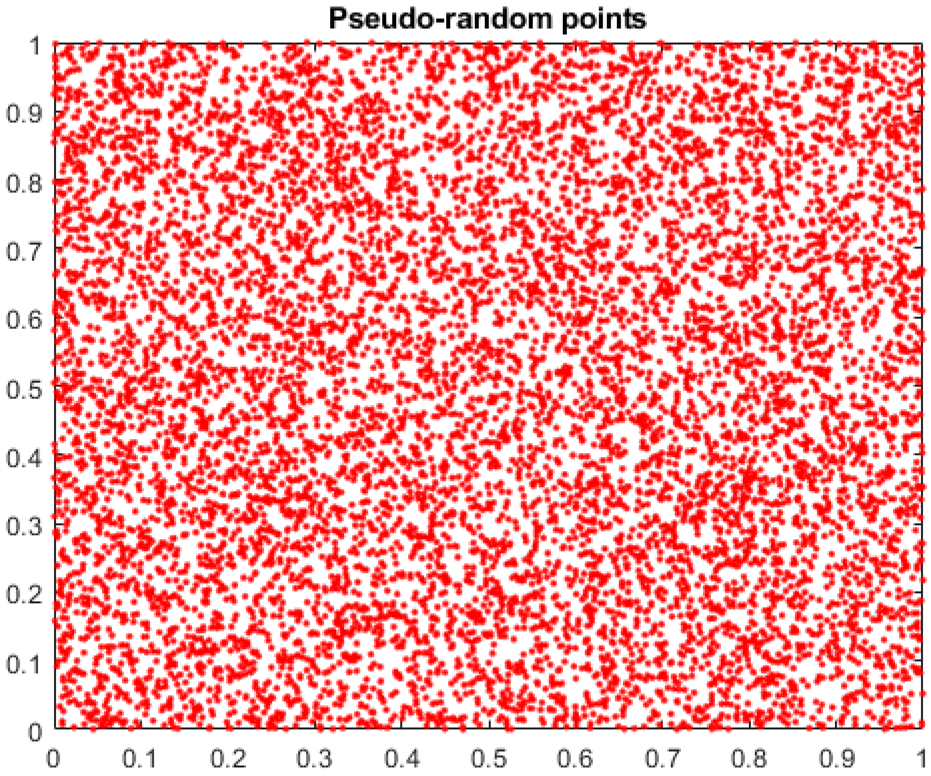
```
b = max(y) - min(y)
```

```
b = 
       13169
```

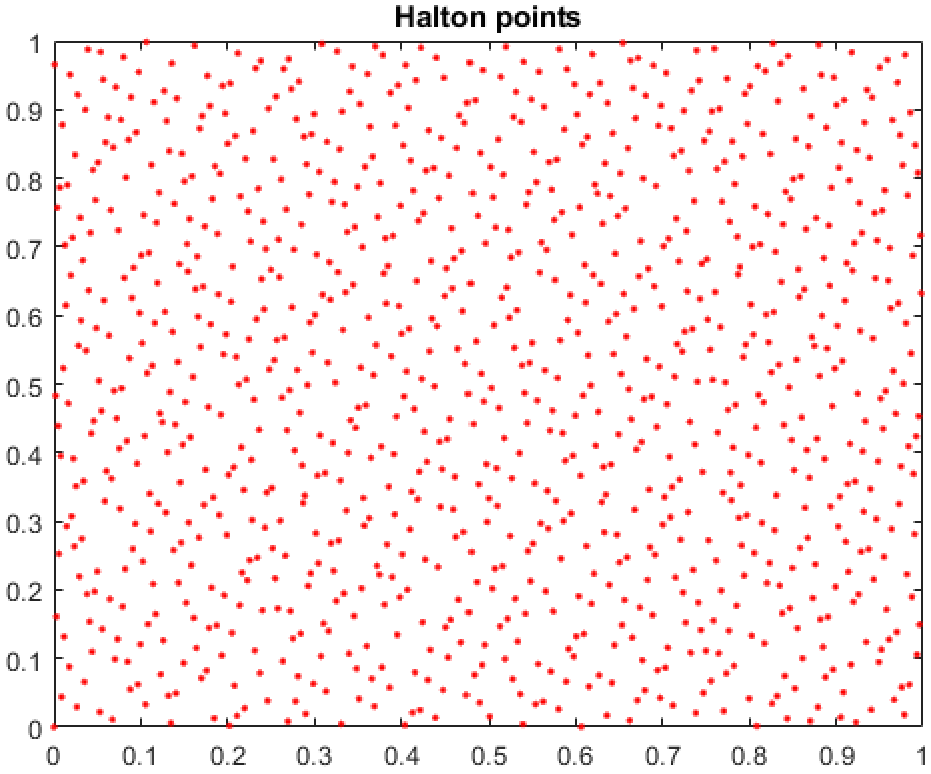
```
rectangle('Position', [min(x), min(y), a, b]);
```

Next, we have to generate 2 dimensional random points. One way to do this would be to call the **`rand`** function, another is use the so called **Halton points** (**`haltonset`** command). The Halton points are based on the *van der Corput* series and are generally cover an area more homogeneously. Both will result in points between $[0, 1]$. Let's generate 1000 points with both methods:

```
% Pseudo-random points
xyr = rand(10000, 2);
figure(2);
plot(xyr(:, 1), xyr(:, 2), 'r.');
title('Pseudo-random points');
```



```
% Halton points
hs = haltonset(2); % this command generates the 2D Halton sequence
xyh = net(hs, 1000); % we sample the sequence 1000 times
figure(3);
plot(xyh(:, 1), xyh(:, 2), 'r.');
title('Halton points');
```



For further computation, we will use the Halton points as they cover the area a bit more homogeneously.

As the points are currently given in the $[0, 1]$ interval, we have to first transform them to our actual interval:

```
xh = xyh(:, 1)*a + min(x);
yh = xyh(:, 2)*b + min(y);
figure(1);
plot(xh, yh, 'b.');
```

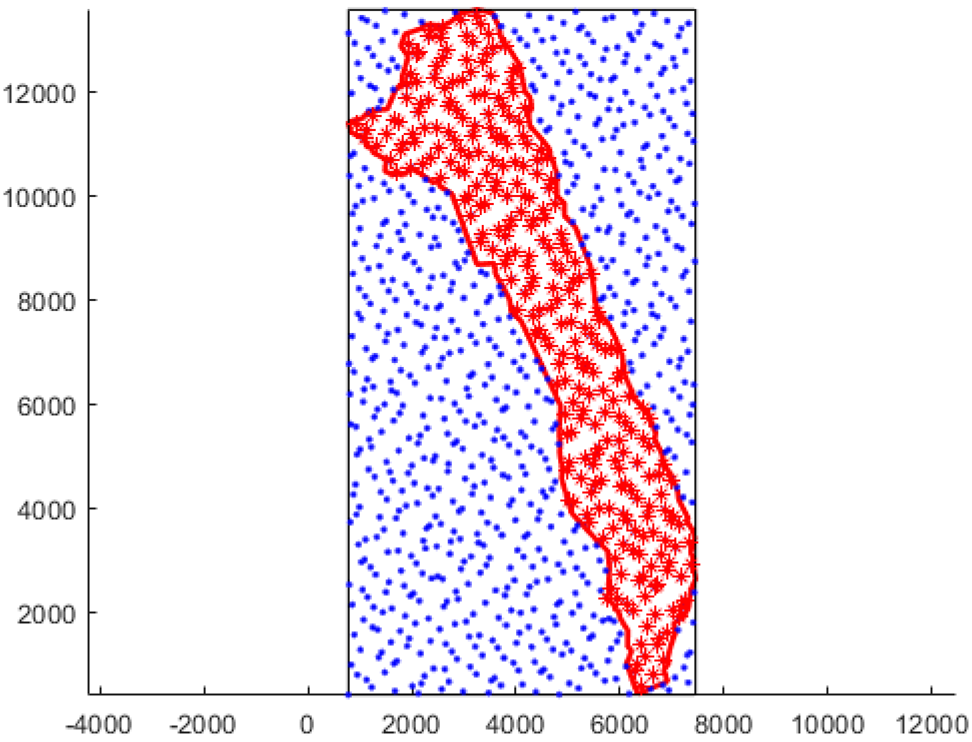
Now, we have to count the number points that are inside our polygon. To do this, we can use the **inpolygon** command in MATLAB. The result will be a vector that contains one in the rows corresponding to points that are inside the polygon and zeros for the ones that outside. We can count the number of nonzero elements by just summing up the elements in the vector, or using the **nnz** command:

```
k = inpolygon(xh, yh, x, y)
```

```
k = 1000×1 logical array
    0
    0
    0
    0
    0
    0
    0
    0
    0
    0
```

⋮

```
plot(xh(k), yh(k), 'r*');
```



```
n = nnz(k)       % number of points inside the polygon
```

```
n =
   280
```

```
N = length(k)    % total number of points
```

```
N =
         1000
```

```
T = a*b          % area of the rectangle containing the polygon
```

```
T =
    87824061
```

```
t = T*n/N        % estimated area of the polygon
```

```
t =
                   24590737.08
```

The Monte Carlo estimate can be improved by using more points. When estimating integrals over irregular shapes, the greatest advantage of the Monte Carlo method is that it can be generalized to any kind of function.

Generalized Monte Carlo method for integration

The general form of the Monte Carlo method for integration can be given as follows: let $f(x)$ be defined on an $x \in V_T$ region. We wish to compute the definite integral of $f(x)$ on a subregion $V_R \subset V_T$:

$$\int_{V_R} f(x)\ dV$$

The average of the function over the subregion can be computed as:

$$\overline{f}_V = \frac{1}{V_R} \int_{V_R} f(x)\ dV$$

But it can also be estimated by:

$$\overline{f_V} \approx \frac{1}{n} \sum_{i=1}^{n} f(x_i)$$

where $x \in V_R$ and n is the number of points inside the subregion. Equating the formulae for the average:

$$\frac{1}{V_R} \int_{V_R} f(x)\ dV \approx \frac{1}{n} \sum_{i=1}^{n} f(x_i)$$

If we solve for the integral, we get:

$$\int_{V_R} f(x)\ dV \approx \frac{V_R}{n} \sum_{i=1}^{n} f(x_i)$$

The estimated value of the subregion V_R can be computed as follows. If the randomly distributed points follow a homogeneous distribution, as the number of points increases, the number of points inside the subregion V_R divided by the total number of points V_T will be proportional to the areas of the two regions:

$$\frac{V_R}{V_T} = \frac{n}{N} \to V_R = V_T \cdot \frac{n}{N}$$

where n is the number of points inside the subregion and N is the total number of points. The estimation of the integral:

$$\int_{V_R} f(x)\ dV \approx \frac{\frac{V_T \cdot n}{N}}{n} \sum_{i=1}^{n} f(x_i) = \frac{V_T}{N} \sum_{i=1}^{n} f(x_i)$$

In other words, the integral can be estimated by taking the sum of the function values at the points inside the subregion and multiplying it by the area of the whole region containing the subregion, divided by the total number of points.

Estimating total precipitation using the Monte Carlo method

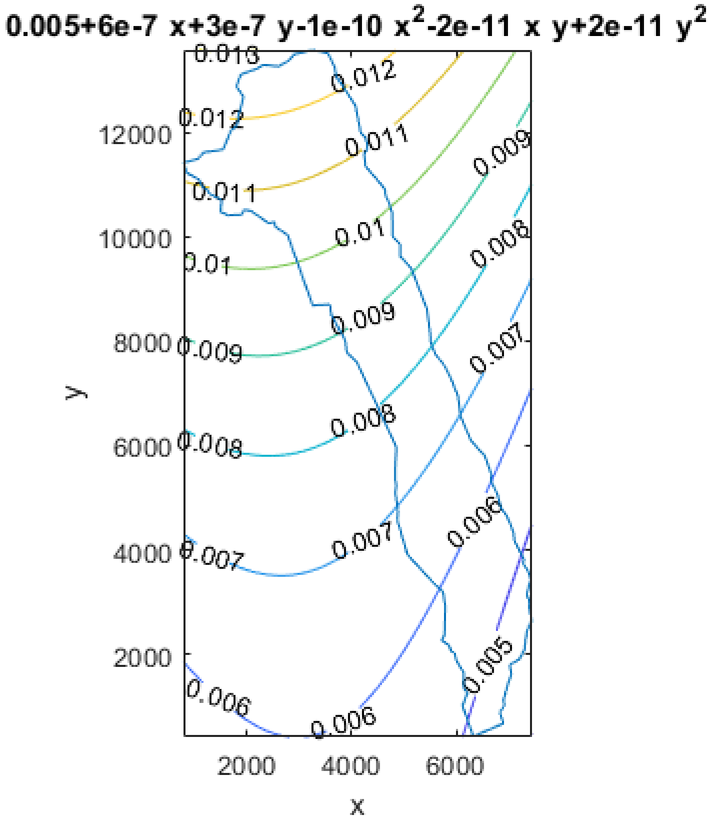
Weather stations were set up in the reservoir from the previous example that measure the precipitation in the area. Given the data of the weather stations, how much was the total precipitation in the whole reservoir?

The precipitation values in the area of the reservoir were estimated using the data from the weather stations and the following 2D quadratic polynomial:

$$f(x) = 0.005 + 6 \cdot 10^{-7}x + 3 \cdot 10^{-7}y - 1^{-10}x^2 - 2 \cdot 10^{-11}xy + 2 \cdot 10^{-11}y^2$$

To find the total precipitation in the area of the reservoir, we have to compute the integral of the polynomial above the area of the reservoir. First, we can define the function and plot its contour lines:

```
f = @(x, y) 0.005 + 6e-7*x + 3e-7*y - 1e-10*x.^2 - 2e-11.*x.*y + 2e-11*y.^2;
figure(4);
h = ezcontour(f, [min(x), max(x), min(y), max(y)]);
set(h, 'Show', 'on');
hold on;
plot(x, y);
axis equal
```



We already have 1000 random points in the area from the previous example, so we can reuse those. Next, we can proceed in two ways. We can compute the average precipitation value from the function values of the points inside the reservoir and multiply it by the area of the reservoir, as we have already computed it from the previous example:

```
xb = xh(k);
yb = yh(k);                       % coordinates of points inside the reservoir
n = length(xb);                   % number of points inside the reservoir
N = length(xh);                   % total number of points
p = 1/n * sum(f(xb, yb))          % average precipitation from the points inside
```

```
p = 
       0.00861282022495253
```

```
sum_p = p * t                     % total precipitation from the average and the area
```

```
sum_p = 
          211795.597669114
```

Or we can use the generalized Monte Carlo method. In this case, we needn't have calculated the area of the reservoir, what is required is the area of the rectangle containing the reservoir, the total number of points and the function values at the points inside the reservoir:

```
p2 = T/N * sum(f(xb, yb))
```

```
p2 = 
          211795.597669114
```

As an example, if our region is regular, for instance, we would like to compute the integral of the function above the rectangular area, we can use the **`integral2`** in 2D:

```
p_rectangle = integral2(f, min(x), max(x), min(y), max(y))
```

```
p_rectangle = 
          719767.774288615
```

Optimization

Mathematical optimization means that we have an objective function $f = f(\boldsymbol{x})$, where $\boldsymbol{x}$ is a vector of inputs ($\boldsymbol{x} = [x_1, x_2, \cdots, x_n]$) and we want to find the place where the function value reaches minimum or maximum. There are many variations of optimization problems, but the majority of the methods that have been developed are looking for the minimum of the objective function:

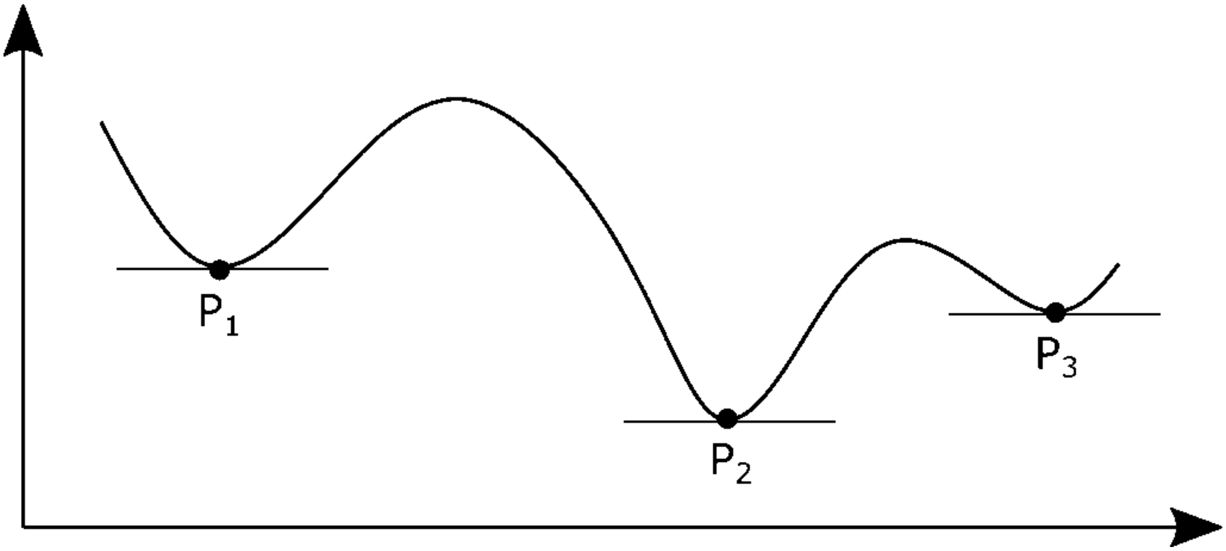
$$f = f(\boldsymbol{x}) \rightarrow \min.$$

If our problem is to find the maximum of some function $g(\boldsymbol{x})$, we can turn this into a minimization problem by taking the function multiplied by -1, that is $-g(\boldsymbol{x})$. By finding the minimum of $-g(\boldsymbol{x})$, we have found the maximum of $g(\boldsymbol{x})$.

Optimization problems are always defined on a certain interval of the input variables. In this interval there can be either

- local minima or maxima, or
- a global minimum or maximum.

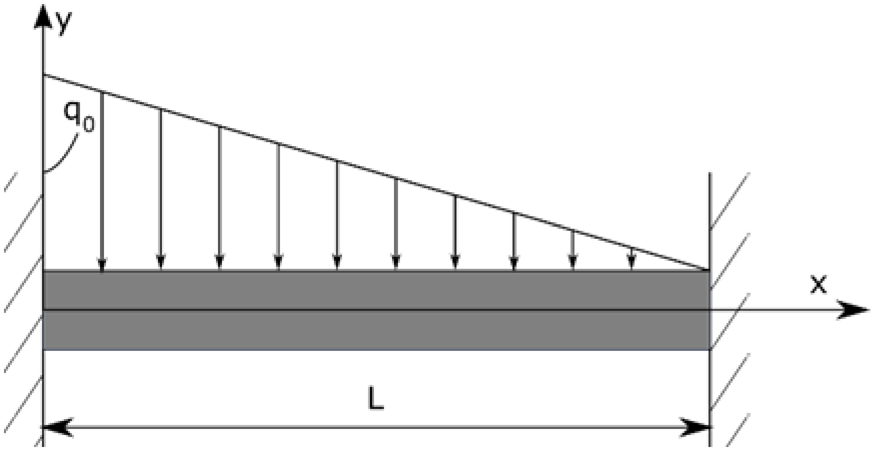
Local minima or maxima are places where the function value is smaller than in any arbitrarily close point. If we have multiple local minima or maxima in the interval, the one with the smallest (or largest) value is the global minimum or maximum in the interval. In the figure below, P1 and P3 are local minima, while P2 would be the global minimum of the interval.



Optimization problems can also be classified by fact whether they are constrained or unconstrained. Constrained optimization means that apart from the objective function that we are minimizing, we have certain linear or nonlinear equations or inequalities that the input variables have to satisfy. For example, one of the most common linear inequality constraint is setting the input variables positive. Different methods exist for different types of constrained optimization problems, whether the constraints are linear or nonlinear, whether there are inequalities and so on. The methods include Lagrange multipliers, the penalty method, the Karush-Kuhn-Tucker conditions and more. In the following, we are only dealing with unconstrained optimization problems, where we only have one objective function that has to be minimized.

Finding the minimum of a univariate function

Let's look at the following example: we have a fixed I-beam with a linearly distributed load according to the following figure:



The specifications of the beam and the load are as follows:

- length of the beam, $L = 3000\,\mathrm{mm}$
- moment of inertia of the beam, $I = 5.29 \cdot 10^7\,\mathrm{mm}^4$
- Young's modulus of the beam, $E = 70\,000\,\mathrm{N/mm}^2$
- The maximum of the distributed load, $q_0 = 15\,\mathrm{kN}/m = 15\,\mathrm{N/mm}$

The deflection of the beam is given by the following function:

$$y = \frac{q_0}{120 \cdot L \cdot E \cdot I} \cdot (x^5 - 5 \cdot L \cdot x^4 + 7 \cdot L^2 \cdot x^3 - 3 \cdot L^3 \cdot x^2)$$

The questions are the following:

1. How much is the deflection (in mm) at 1 m and at 2 m?
2. Where is the maximum (along the x-axis) of the deflection (in mm)?

First, we have to define the variables:

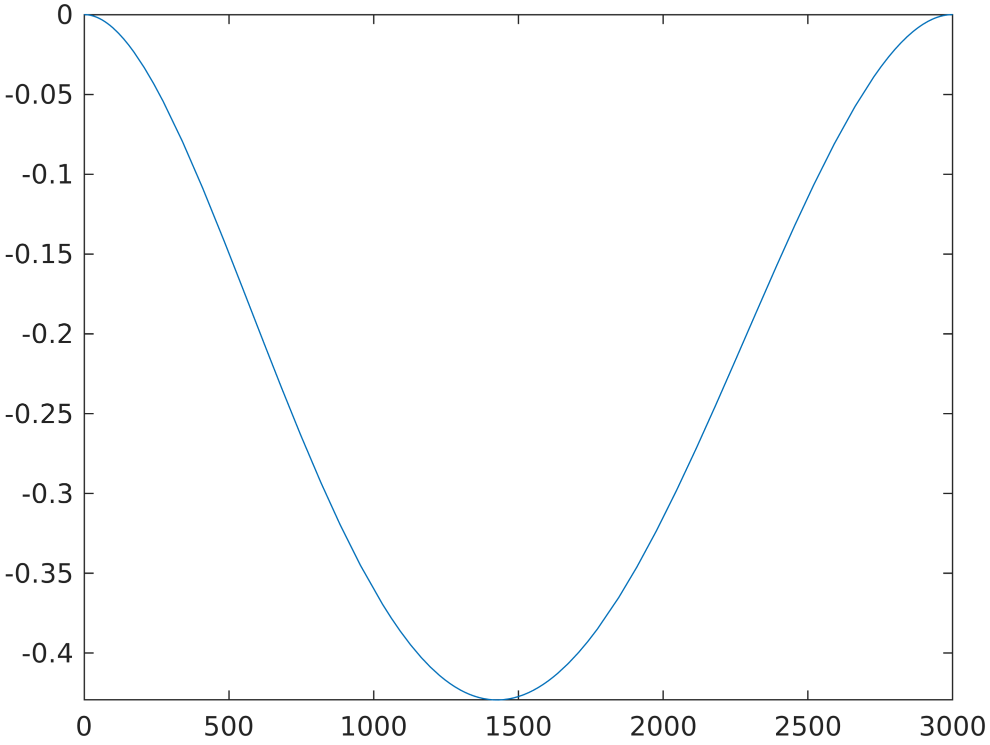
```
format longG
E = 70000;
I = 5.29e7;
q0 = 15;
L = 3000;
```

Next, we can define the function of the deflection. Our only variable is the x distance:

```
y = @(x) q0/(120*L*E*I) * (x.^5 - 5*L*x.^4 + 7*L^2*x.^3 - 3*L^3*x.^2);
```

We can visualize the function to get a better idea about the deflection:

```
figure(1);
fplot(y, [0, 3000]);
```



To find the answers to the first question, we just simply have to substitute the x values into the deflection function. As we have defined everything in mm, the results are in mm as well:

```
y(1000)
```

```
ans =
         -0.36006841299847
```

```
y(2000)
```

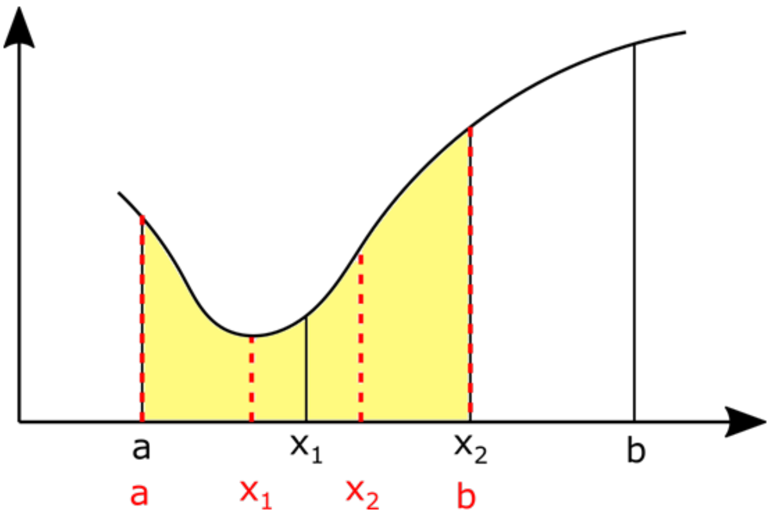
```
ans =
        -0.315059861373661
```

Before we can answer the second question, we have to take a look at some optimization methods that will help us find the minimum of the deflection function.

Interval methods -- Ternary search algorithm

The interval methods are very similar to the ones used to find the roots of functions. We have to specify a certain closed interval [a, b] that contains one minimum of the function (in other words, the function is unimodal in the interval), but instead of calculating only one function value inside the interval, we will calculate two (x_1, x_2).

As the function only has one minimum in the interval, its value is monotonically decreasing until the minimum and then it is monotonically increasing. If the function value in x_1 is smaller than that in x_2, it means that the minimum is in the interval [a, x_2]. Similarly, if the function value in x_2 is smaller than the function value in x_1, then the minimum has to be in the interval [x_1, b]. Therefore, we have shrunk the interval and found our new a and b values. We can find a new x_1 and x_2 in this new interval and continue the algorithm while the interval is bigger than some tolerance value or we have reached a certain number of iterations.



As long as the function is unimodal in the interval, the method will converge. The more interesting question is, that how should we define the x_1 and x_2 values to be most efficient (that is, to find the minimum with the least amount of computations)? One approach is to distribute the points equally, x_1 will be in the 1/3 mark of the interval and x_2 will be in the 2/3 mark. This is called the ternary search algorithm, which can be implemented in MATLAB using the following code:

```
function [x, i] = interval(f, a, b, tol)

 i = 1;
 x1 = a + 1/3*(b-a);
 x2 = b - 1/3*(b-a);
 while abs(x2-x1) > tol
   if f(x1) < f(x2)
     b = x2;
   else
     a = x1;
   end
   i = i+1;
   x1 = a + 1/3*(b-a);
   x2 = b - 1/3*(b-a);
 end
 x = (x1+x2)/2;
end
```

Let's use this algorithm to find the maximal deflection (which is a minimum value of the deflection function, as the deflection is negative). From the figure we can see that the minimum is inside the [1000, 2000] interval, so we can use that as our initial interval:

```
[x1, i1] = interval(y, 1000, 2000, 1e-6)
```

```
x1 =
          1425.91478062954
i1 =
    50
```

It took 50 iterations for the algorithm to find the minimum which turned out to be at ~1426 mm. It's value is:

```
y(x1)
```

```
ans =
        -0.429347419445165
```

-0.4293 mm.

Golden-section search

The golden-section method improves upon the interval method and uses the golden ratio to distribute x_1 and x_2 inside the search interval. We can find the golden ratio by diving the length of the interval into two parts ($L = L_1 + L_2$) in such a way, that L_2 is proportional to L the same way as L_1 is to L_2:

$$R = \frac{L_2}{L} = \frac{L_1}{L_2}$$

If we rearrange the equation and solve for L_2 and L_1:

$$L_2 = R \cdot L \quad \text{and} \quad L_1 = R \cdot L_2 = R^2 \cdot L$$

Substitute L_2 and L_1 into $L = L_1 + L_2$:

$$L = R \cdot L + R^2 \cdot L$$

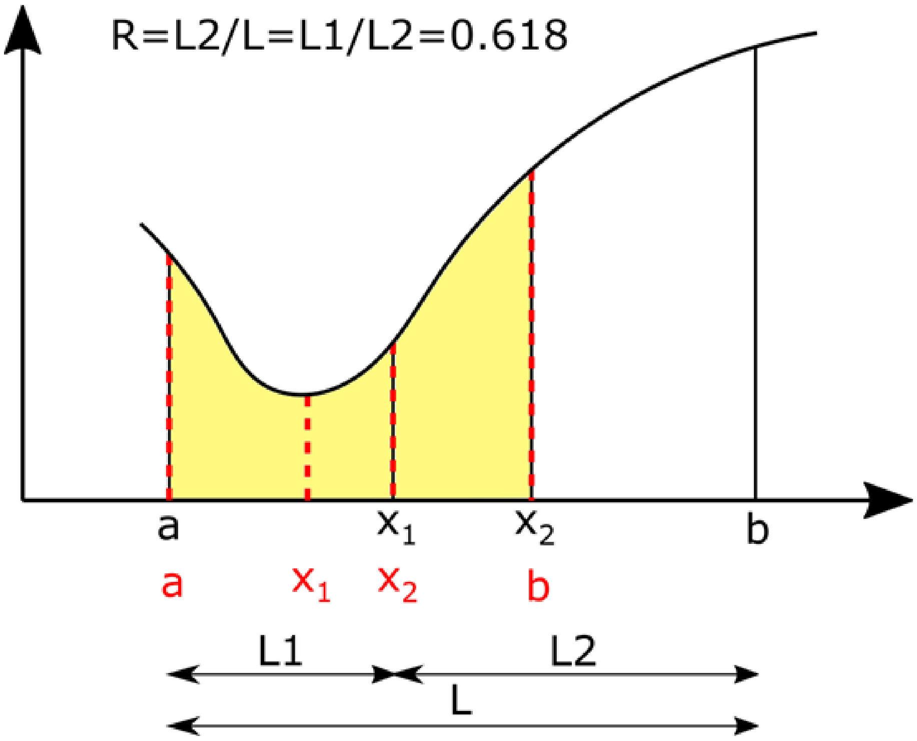
Dividing this by L and bringing everything to the left side:

$$R^2 + R - 1 = 0$$

The positive solution of this quadratic equation is the golden ratio:

$$R = \frac{\sqrt{5} - 1}{2} \approx 0.618$$

If we choose the place of x_1 and x_2 according to the golden ratio, that is, x_1 is $0.618 \cdot L$ from b and x_2 is symmetrically $0.618 \cdot L$ from a, then what happens is that as we decrease the interval between [a, x_2] or [b, x_1], one of the internal points of the new, decreased interval will already be given, as it will be the same as x_1. In other words, we only have to compute the function value inside one other point in the new [a, b] interval, the place of which is chosen according to the golden ratio. This results in relatively faster convergence and a lot less function evaluations.



In MATLAB, we could use the following code to do the golden-section search:

```
function [x, i] = goldensect(f, a, b, tol)

i = 1;
R = (sqrt(5)-1)/2;
x1 = b - R*(b-a);
x2 = a + R*(b-a);
f1 = f(x1); f2 = f(x2);
while abs(x2-x1) > tol
  if f1 < f2
     b = x2;
     x2 = x1; f2 = f1; % f2 comes from the previous iteration!
     x1 = b - R*(b-a);
     f1 = f(x1);
  else
     a = x1;
     x1 = x2; f1 = f2; % f1 comes from the previous iteration!
     x2 = a + R*(b-a);
     f2 = f(x2);
  end
  i = i+1;
end
x = (x1+x2)/2;
```

x_1 and x_2 only have to be computed in the first iteration and in all of the subsequent ones, only one of them will be updated. Let's use the golden-section search to find the maximum of the deflection:

```
[x2, i2] = goldensect(y, 1000, 2000, 1e-6)
```

```
x2 =

          1425.91478565744

i2 =
    42
```

```
y(x2)
```

```
ans =
       -0.429347419445165
```

Instead of 50 iterations, the algorithm only needed 42 now. However, what is more important is that instead of the previous $50 \cdot 2 = 100$ function evaluations, the new algorithm only carried out $41 + 2 = 43$ function evaluations, which is a huge gain when it comes to complicated functions.

Newton's method for optimization

If we can easily compute the derivative of the objective function, then we can use a modified version of Newton's algorithm that we used to find roots of nonlinear functions. The trick is, that we are looking for the roots of the first derivative of the function now as those are the places where the function can have a minimum or a maximum. The iteration formula for the regular Newton's method was the following:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

If instead we are looking for the root of the derivative function, the iteration formula becomes:

$$x_{i+1} = x_i - \frac{f'(x_i)}{f''(x_i)}$$

This means, that we have to be able to calculate the first and the second derivative of the objective function. In the case of a problem like the one above, it is not too much work, even without using the computer. We can of course use MATLAB to do the work for us:

First, we have to convert the function into a symbolic expression. If we do not want to lose precision, it is better to define the symbolic function using only symbolic variables (for E, I, q0 and L as well):

```
syms E I L x
ys = q0/(120*E*I*L) * (x.^5 - 5*L*x.^4 + 7*L^2*x.^3 - 3*L^3*x.^2)
```

ys =

$$-\frac{3\,L^3\,x^2 - 7\,L^2\,x^3 + 5\,L\,x^4 - x^5}{8\,\mathrm{E}\,\mathrm{I}\,L}$$

Now, we can use symbolic derivation. As the second argument, we define the variable with respect to which we differentiate:

```
dx = diff(ys, x)
```

dx =

$$-\frac{6\,L^3\,x - 21\,L^2\,x^2 + 20\,L\,x^3 - 5\,x^4}{8\,\mathrm{E}\,\mathrm{I}\,L}$$

```
ddx = diff(dx, x)
ddx =
```

$$-\frac{6\,L^3 - 42\,L^2\,x + 60\,L\,x^2 - 20\,x^3}{8\,\mathrm{E}\,\mathrm{I}\,L}$$

To substitute the numeric values into the symbolic expression, we first have to define the variables again and then use the **subs** command:

```
E = 70000; I = 5.29e7; L = 3000; q0 = 15;
dx = subs(dx);
ddx = subs(ddx);
```

Convert the expressions back to anonymous functions:

```
dxf = matlabFunction(dx)
```

```
dxf = function_handle with value:
    @(x)x.*(-1.822846340804753e-6)+x.^2.*2.126654064272212e-9-x.^3./1.4812e12+x.^4./1.77744e16
```

```
ddxf = matlabFunction(ddx)
```

```
ddxf = function_handle with value:
    @(x)x.*4.253308128544423e-9-x.^2.*2.025384823116392e-12+x.^3./4.4436e15-1.822846340804753e-6
```

Solution using Newton's method:

```
[xn, in] = newton(dxf, ddxf, 2000, 1e-6, 100)
```

```
xn =
          1425.68322611289
in =
     3
```

```
y(xn)
```

```
ans =
        -0.429347398633028
```

It only took 3 iterations to reach a similar tolerance. We can see from this that this method has a much faster convergence rate, if it converges.

MATLAB's built-in algorithm

There are of course built-in methods in MATLAB to solve optimization problems. One of the function that works for unconstrained optimization is **`fminsearch`**, which uses the Nelder-Mead simplex algorithm. Let's use it on the same problem:

```
xmin = fminsearch(y, 2000)
xmin =
           1425.9147644043
```

```
y(xmin)
```

```
ans =
        -0.429347419445165
```

The function can also be called with extra outputs that provide information about the procedure:

```
[xmin, fval, exitflag, output] = fminsearch(y, 2000)
```

```
xmin =
           1425.9147644043
fval =
        -0.429347419445165
exitflag =
     1
output = struct with fields:
    iterations: 25
     funcCount: 50
     algorithm: 'Nelder-Mead simplex direct search'
       message: 'Optimization terminated:↵ the current x satisfies the termination criteria using OPTIONS.To
```

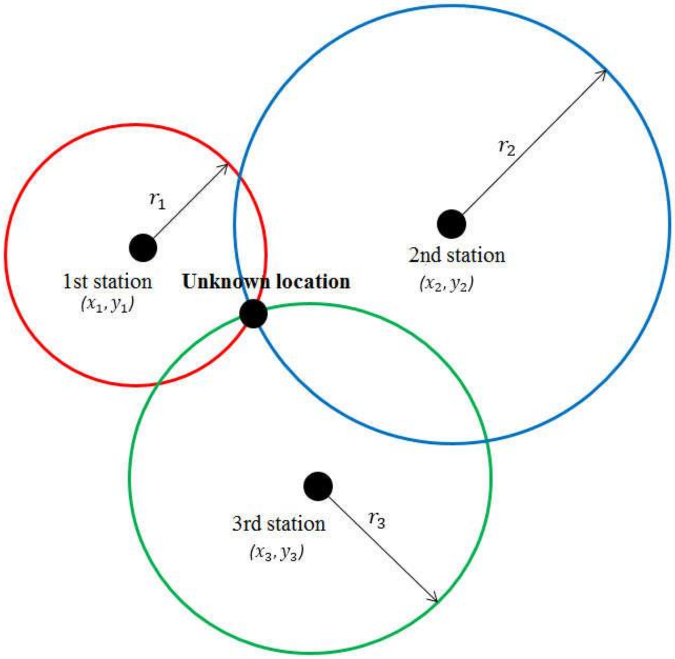
Multivariate function optimization

In many cases, the objective function that have to be optimized have multiple input variables. For example, finding the highest or lowest point of a surface, minimizing the distances from a road intersection etc. For unconstrained multivariate optimization, we can use the multivariate Newton's method, the gradient method or the Nelder-Mead simplex method.

Finding the position in an overdetermined case

Previously we have looked at a positioning problem where distances are measured from three cell towers and the coordinates of the point at the intersection of the circles (represented by the distances) have to be found. In the case when we have more than 3 distances to more than 3 cell towers (an overdetermined case), due to the perturbation in the data, we cannot hope to find a single point of intersection. Instead, we can only find a point in which the sum of the squared distances from the given circles (the sum of the squared residuals) is minimal. We have seen this least squares problem before in the case of regression and solved it by linearizing the data. However, it can also be solved by minimizing the objective function (the sum of the squared residuals), which we wil do now.

We have the following data for 4 cell towers (x, y - coordinates of the tower [m], r - distance from the tower [m]):



Mobile mast number	Coordinate X (x_i) [m]	Coordinate Y (y_i) [m]	Mast-terminal distance (r_i) [m]
1	561	1487	2130
2	5203	4625	5620
3	5067	-5728	6040
4	1012	5451	5820

The equation of each circle around the towers can be given in the following explicit form:

$$(x - x_i)^2 + (y - y_i)^2 - r_i^2 = 0$$

Let's define the data for the towers and the measurements first:

```
xt = [561, 5203, 5067, 1012];
yt = [487, 4625, -5728, 5451];
rm = [2130, 5620, 6040, 5820];
```

We can use a for loop to plot the position of the towers and circles around them. First, we define the equation of the circles in a general form:

```
circle = @(x, y, xi, yi, ri) (x - xi).^2 + (y - yi).^2 - ri.^2
```

```
circle = function_handle with value:
    @(x,y,xi,yi,ri)(x-xi).^2+(y-yi).^2-ri.^2
```

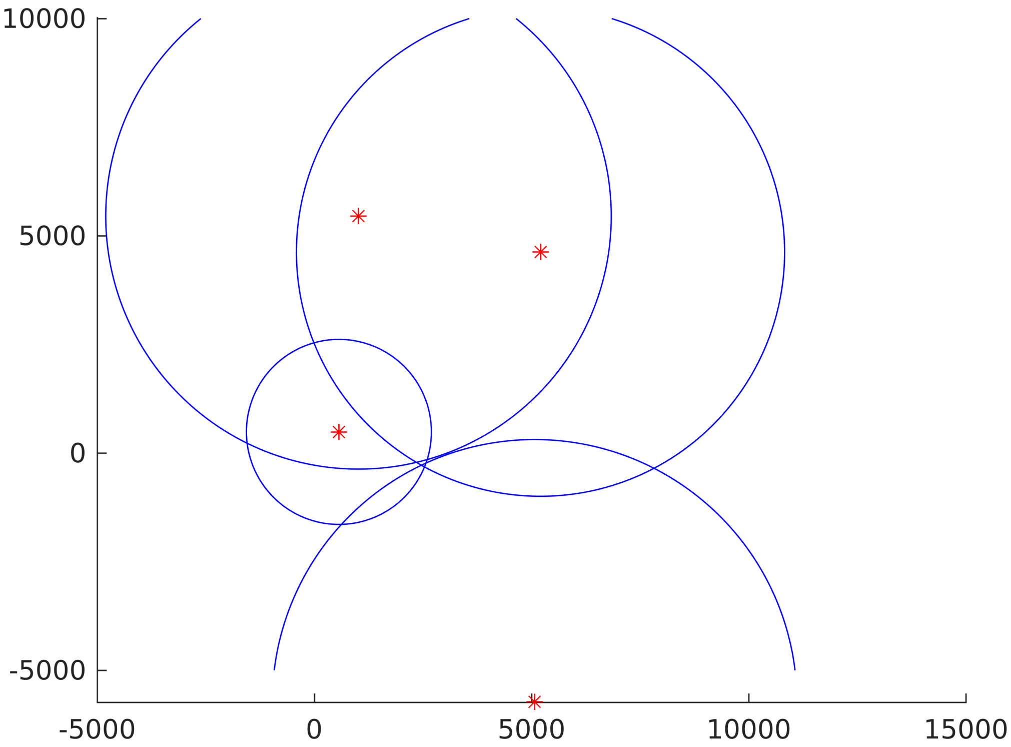
We can convert the equations into symbolic form by substituting a symbolic x and y variables and the data for each tower:

```
syms x y
Eqs = circle(x, y, xt, yt, rm)
```

Eqs

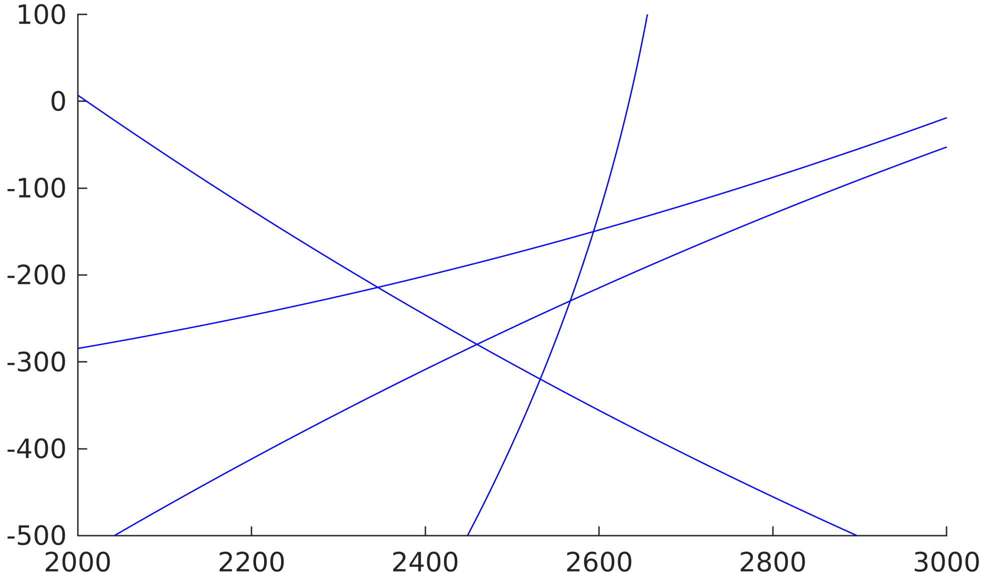
$$= \left((x-561)^2+(y-487)^2-4536900 \quad (x-5203)^2+(y-4625)^2-31584400 \quad (x-5067)^2+(y+5728)^2-36481600 \quad (x-\right.$$

```

figure(2); hold on;
for i = 1:length(Eqs)
    plot(xt(i), yt(i), 'r*')
    fimplicit(Eqs(i), [-5000, 15000, -5000, 10000], 'b-')
end
axis equal;
```



Zoom in on the figure to better see the intersection:

```
axis([2000, 3000, -500, 100])
```



We can clearly see that there is no defined point of intersection, we have to find the optimal point by minimizing the sum of the squared residuals. This objective function can be defined as follows:

$$f(x, y) = \sum_{i=1}^{n} \left((x - x_i)^2 + (y - y_i)^2 - r_i^2\right)^2$$

Define the objective function and plot it:

```
f = sum(circle(x, y, xt, yt, rm).^2)
```

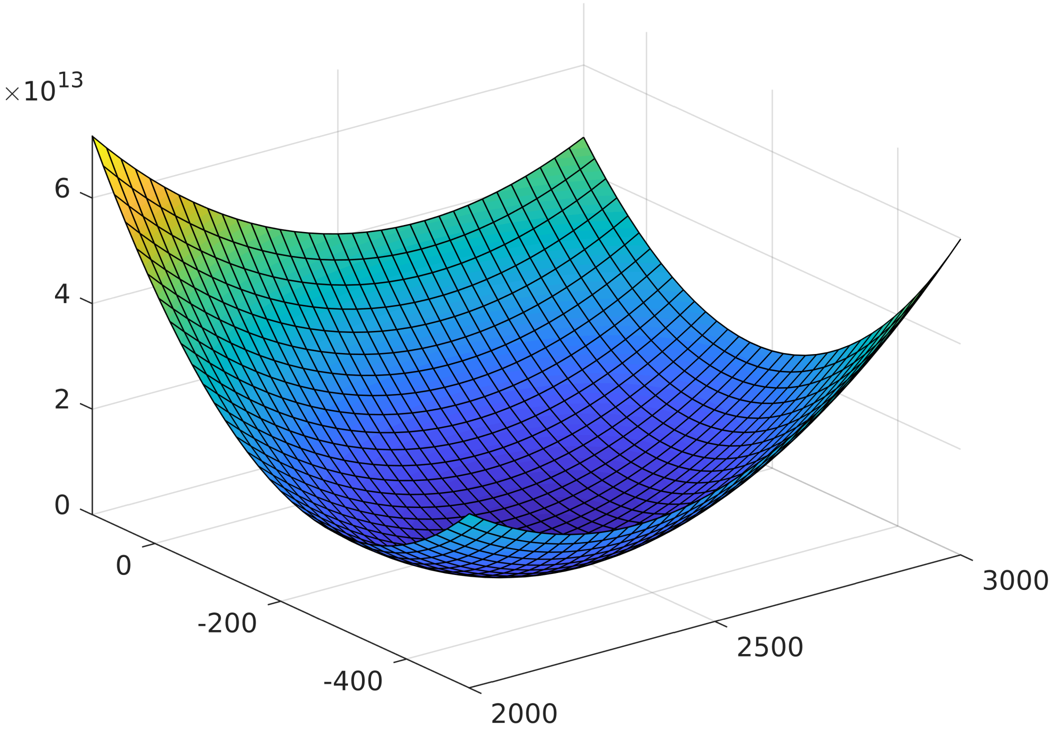
f

$$= \left((x-561)^2 + (y-487)^2 - 4536900\right)^2 + \left((x-5203)^2 + (y-4625)^2 - 31584400\right)^2 + ((x-1012)^2 + (y-5451)^2 - 3387240$$

```
ff = matlabFunction(f)
```

```
ff = function_handle with value:
    @(x,y)((x-5.61e2).^2+(y-4.87e2).^2-4.5369e6).^2+((x-5.203e3).^2+(y-4.625e3).^2-3.15844e7).^2+((x-1.01
```

```
figure(3);
fsurf(ff, [2000, 3000, -500, 100])
```



To find the lowest point of the surface (the minimum of our objective function), we can use the following method.

Multivariate Newton's method

In the univariate case, the iteration formula for the optimization was the following:

$$x_{i+1} = x_i - \frac{f'(x_i)}{f''(x_i)}$$

In the multivariate case, instead of the first derivative of the function, we have to use the gradient vector of the function (∇f), which contains the partial derivatives with respect to each variable (x and y in our case). In place of the second derivative, we use the function Hessian matrix, that contains the second partial derivatives:

$$\nabla \boldsymbol{f}(x, y) = \begin{bmatrix} \frac{\partial}{\partial x} f \\ \frac{\partial}{\partial y} f \end{bmatrix} \qquad \boldsymbol{H}(x, y) = \begin{bmatrix} \frac{\partial^2}{\partial x^2} f & \frac{\partial^2}{\partial x \partial y} f \\ \frac{\partial^2}{\partial x \partial y} f & \frac{\partial^2}{\partial y^2} f \end{bmatrix}$$

The iteration formula becomes the following:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \boldsymbol{H}^{-1}(\boldsymbol{x}_i) \cdot \nabla \boldsymbol{f}(\boldsymbol{x}_i)$$

The variables have to given in vector form of course. An example for the multivariate Newton's algorithm is given below:

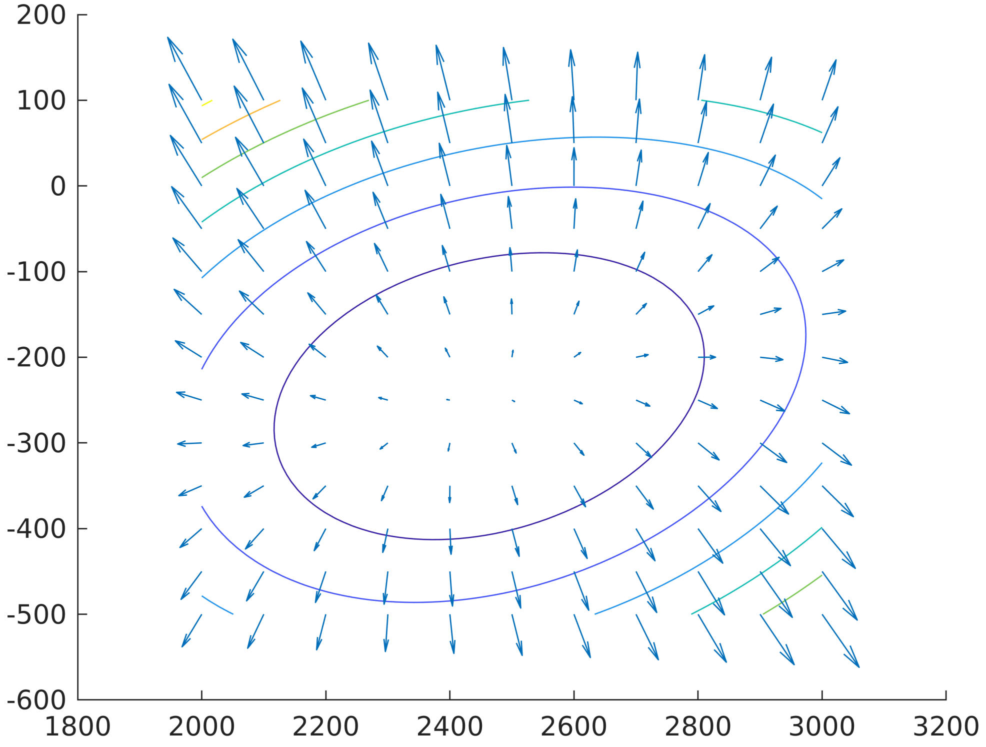
```
function [x i X] = gradmulti(grad, hesse, x0, eps, nmax)

x1 = x0 - pinv(hesse(x0))*grad(x0);
i=1;
X=[x0 x1];

% Stop conditions:
% 1. the step size is smaller than the given eps value, or
% 2. the maximum number of iterations has been reached
while and(norm(x1 - x0) > eps, i < nmax)
 x0 = x1;
 x1 = x0 - pinv(hesse(x0))*grad(x0);
 i = i + 1;
 X = [X x1];
end
x = x1;
```

Before we find the minimum of the function, we can visualize the gradient vectors of the surface. In MATLAB, we can calculate the gradient both numerically and symbolically. In order to calculate it numerically, we first have to create a mesh grid and evaluate the function on it:

```
[X, Y] = meshgrid(2000:100:3000, -500:50:100);
Z = ff(X, Y);
```

We can use the **`gradient`** function to compute the gradients in the points of the mesh grid and the **`quiver`** function to plot the vectors:

```
[gx, gy] = gradient(Z);
figure(4); hold on;
fcontour(ff, [2000, 3000, -500, 100]);
quiver(X, Y, gx, gy)
```



To use our multivariate Newton's algorithm, we have to supply the gradient and the Hessian in vector form. If we have our function in symbolic form, we can use the **`gradient`** and the **`hessian`** functions:

```
g = gradient(f)
```

```
g =
```

$$\begin{pmatrix} 2\ (2\,x-1122)\,\sigma_4+2\ (2\,x-10406)\,\sigma_3+2\ (2\,x-2024)\,\sigma_2+2\ (2\,x-10134)\,\sigma_1 \\ 2\ (2\,y-974)\,\sigma_4+2\ (2\,y-9250)\,\sigma_3+2\ (2\,y-10902)\,\sigma_2+2\ (2\,y+11456)\,\sigma_1 \end{pmatrix}$$

where

$$\sigma_1=(x-5067)^2+(y+5728)^2-36481600$$

$$\sigma_2=(x-1012)^2+(y-5451)^2-33872400$$

$$\sigma_3=(x-5203)^2+(y-4625)^2-31584400$$

$$\sigma_4=(x-561)^2+(y-487)^2-4536900$$

```
h = hessian(f)
```

```
h =
```

$$\left(\begin{array}{c} \sigma_5+\sigma_4+\sigma_3+\sigma_2+\sigma_9+\sigma_8+\sigma_7+\sigma_6+2\ (2\,x-1122)^2+2\ (2\,x-2024)^2+2\ (2\,x-10134)^2+2\ (2\,x-10406)^2-4259 \\ \sigma_1 \end{array}\right.$$

where

$$\sigma_1 = 2\ (2\,x-1122)\ (2\,y-974)+2\ (2\,x-2024)\ (2\,y-10902)+2\ (2\,x-10406)\ (2\,y-9250)+2\ (2\,x-10134)\ (2$$

$$\sigma_2 = 4\ (x-5203)^2$$

$$\sigma_3 = 4\ (x-5067)^2$$

$$\sigma_4 = 4\ (x-1012)^2$$

$$\sigma_5 = 4\ (x-561)^2$$

$$\sigma_6 = 4\ (y+5728)^2$$

$$\sigma_7 = 4\ (y-5451)^2$$

$$\sigma_8 = 4\ (y-4625)^2$$

$$\sigma_9 = 4\ (y-487)^2$$

Convert these to anonymous functions:

```
gf = matlabFunction(g)
```

```
gf = function_handle with value:
    @(x,y)[(x.*2.0-1.122e3).*((x-5.61e2).^2+(y-4.87e2).^2-4.5369e6).*2.0+(x.*2.0-1.0406e4).*((x-5.203e3).
```

```
hf = matlabFunction(h)
```

```
hf = function_handle with value:
    @(x,y)reshape([(x-5.61e2).^2.*4.0+(x-1.012e3).^2.*4.0+(x-5.067e3).^2.*4.0+(x-5.203e3).^2.*4.0+(y-4.87
```

Write them using vector notation:

```
gv = @(x) gf(x(1), x(2));
hv = @(x) hf(x(1), x(2));
```

We can choose an initial guess from the figure and find the minimum:

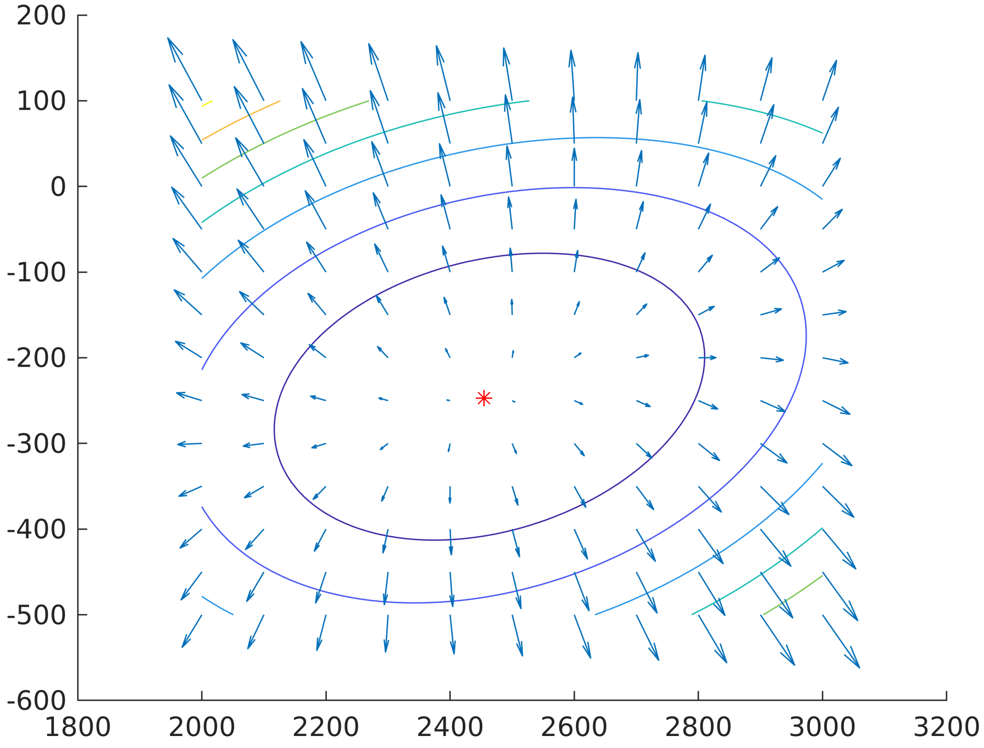
```
x0 = [2400; -300];
```

```
[s, i, S] = gradmulti(gv, hv, x0, 1e-6, 100)
```

```
s = 2×1
          2454.65697065448
           -246.9483059606
i = 
     4
S = 2×5
                        2400          2452.89756162567          2454.65599642035 ···
                        -300         -247.608991028719         -246.948598495572
```

Plot the solution:

```
plot(s(1), s(2), 'r*')
```



Built-in methods in MATLAB

There are two main algorithms in MATLAB for solving multivariate optimization:

- **`fminsearch`** that uses the Nelder-Mead simplex method and
- **`fminunc`** that uses the quasi-Newton algorithm.

If the derivatives cannot be computed efficiently, then it is recommended to use the simplex method. In this case, we use a simplex (a polytope of n+1 vertices in n dimensions, i.e. a triangle in two dimensions) and by operating on the simplex (streching, shrinking, mirroring) we change the location of the vertices in such a way that they always conform to the shape of the surface. This causes the simplex to move towards the minimum of the function. A GIF showing the movement of the simplex can be found at: https://en.wikipedia.org/wiki/File:Nelder-Mead_Himmelblau.gif.

In order to solve the problem using the simplex method, we have to define our function using the vector notation:

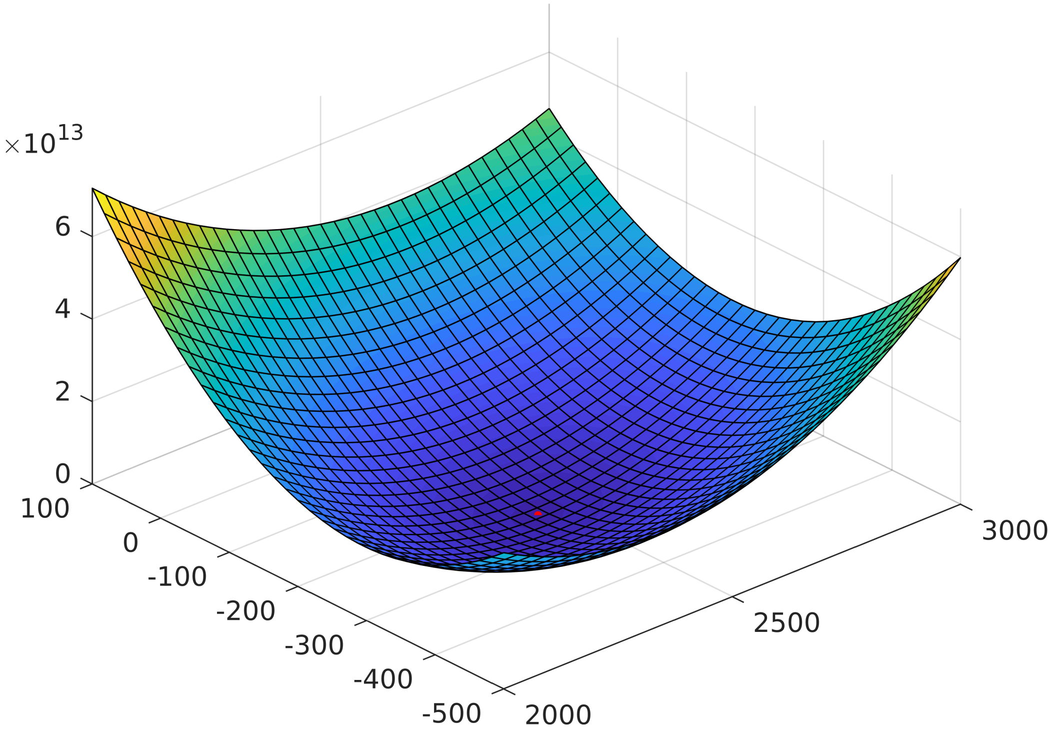
```
fv = @(x) ff(x(1), x(2))
```

```
fv = function_handle with value:
    @(x)ff(x(1),x(2))
```

```
sol = fminsearch(fv, x0)
```

```
sol = 2×1
          2454.6569708984
        -246.948306492714
```

```
figure(3); hold on;
plot3(sol(1), sol(2), ff(sol(1), sol(2)), 'r.', 'MarkerSize', 10)
view([-42 43])
```



The distances of the solution from each of the circles:

```
ex = xt - sol(1);
ey = yt - sol(2);
er = rm - sqrt(ex.^2 + ey.^2)
er = 1×4
          99.0847284941651         26.3187696519035        -31.7595282253087 ···
```

Remark: All the methods shows above are for finding local minima or maxima. They will generally (not always) converge to the closes minimum or maximum from the initial guess. If we have multiple minima or maxima in the interval and our goal is to find the global minimum for example, we have to find each of the local minima or maxima and compare their function values. There are stochastic methods that can be used to find global minima or maxima on a given interval (genetic algorithms, simulated annealing, particle swarm etc.), but due to time constraints, we do not cover these in class.

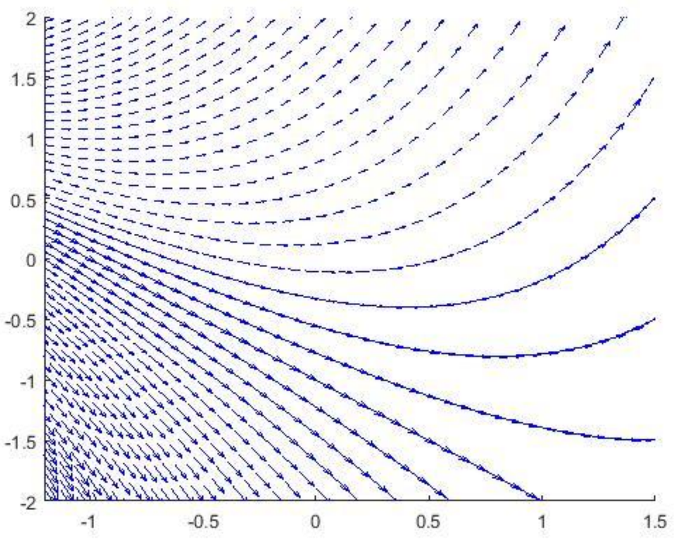
Ordinary Differential Equations - Initial value problems

Differential equations

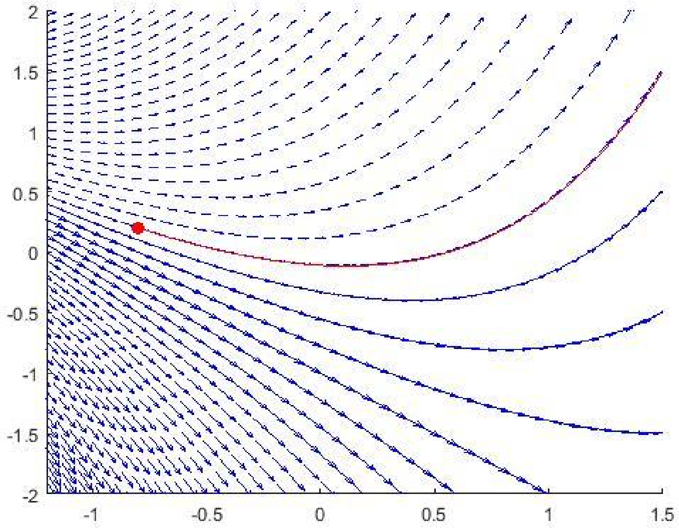
Differential equations are equations containing a function $y = f(\boldsymbol{x})$ and its derivatives. The ordinary differential equations (ODEs) contain a univariate function and its derivatives. The order of the ODE equals the highest order of derivatives in the equation. As opposed to an ODE, there are partial differential equations (PDEs) as well that contain a multivariate function's partial derivatives.

The solution for a differential equation is not a number but a function. In order for the solution to be unique, some constraints have to be met by the solution function. One constraint can be that the function and its derivative take some value at the beginning of the interval of interest, hence the initial value problem. Another constraint would be that at least one of the values the function or its derivative takes is given at the end of the interval.

Generally, we have a solution set for a given differential equation. For example, take the following ODE: $\frac{dy}{dx} = y + x$. We are looking for a function that solves this equation. All the functions in the figure below solve the equation, so they make up the solution set for the equation. The figure below is also called the trajectory or direction field of the ODE.



If we specify the initial conditions for the function and derivative's value at the beginning of our search interval, e.g. the function has to go through the point $x = -0.8, y = 0.2$, that we have reduced the solution to a specific solution, that is the function given by the red line below:



There are linear ODE's and nonlinear ones. In a linear ODE, the linear expression of the unknown function and its derivatives are given. For example:

$$e^x \cdot \frac{dy}{dx} + a \cdot x^2 + x^4 \cdot y = 0$$ -- This is a linear ODE.

$$\frac{dy}{dx} + a \cdot x \cdot y + b \cdot y^2 = 0$$ -- This is a nonlinear ODE.

In the case of many differential equations and systems (especially if they are nonlinear), the solution function can only be found numerically. The numeric solution give us the data points of the function computed using numeric integration. The aim of these algorithms is to accurately approximate the data points of the solution function using as few steps and function evaluations as possible.

Ordinary Differential Equations - The initial value problem

The general form of a first order ODE with the independent variable t:

$$y' = \frac{dy}{dt} = f(t, y)$$

In the univariate case, we have one independent variable (t) and one dependent variable (y). The derivative of the function y is given by $f(t, y)$. In the case of the initial value problem, we know that the solution goes through the (t_0, y_0) point:

$$y(t_0) = y_0$$

Euler's method

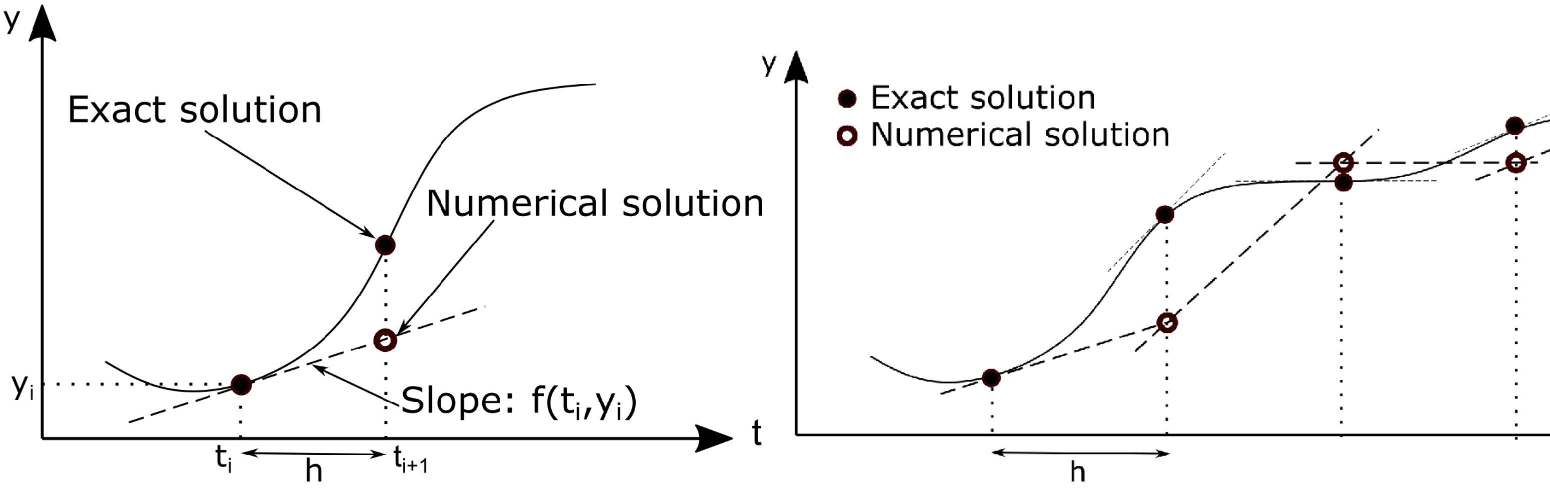
We wish to find the values of the solution function on a given interval using a predefined step size (h). We suppose that the slope of the function (m) on the interval defined by the h step size is constant. If we know the functions value at the beginning of the interval and the value of its derivative (which denotes the slope of the function) than we can approximate the value of the function at the end of the h step size using a line with slope m.

In case of Euler's method, we suppose that the $m = f(t, y)$ value is constant for each subinterval ($h = t_{i+1} - t_i$) where we integrate and its value is given at the beginning of the interval.

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y)dt \approx y_i + f(t_i, y_i) \cdot h = y_i + m_i \cdot h$$

$$t_{i+1} = t_i + h$$

where m is the slope at the beginning of each subinterval and its value is constant on the subinterval. The local error of the method is $\mathrm{O}(h)$, its global error is $\mathrm{O}(h)$ meaning that it is a first-order method.



Let's look at an example custom code that implements Euler's method in MATLAB:

```
function [t,y] = euler (f, y0, a, b, h)
n = round((b - a)/h);
t(1) = a;
y(1) = y0;
for i = 1 : n - 1
    y(i + 1) = y(i) + h*f(t(i), y(i));
    t(i + 1) = t(i) + h;
end
```

The inputs to the function above:

1. f - the handle of the function that defines the right-hand side of the ODE.
2. y0 - the initial value of the solution function at the beginning of the search interval.
3. a - the beginning of the search interval.
4. b - the end of the search interval.
5. h - the step size for the computation.

Solving an ODE using Euler's method

Let's look at the following example. A spherical water tank with radius $R = 10\,\mathrm{m}$ is drained using a hole on its bottom at $h = 0$ height. The hole has a radius $r = 5\,\mathrm{cm}$. The contains approximately 4000 m^3 of water. At the beginning of the draining ($t = 0$), the height of the water inside the tank is 17.44 m. The coefficient of contraption of the hole is $\mu = 0.85$.

The water level inside the tank, measured from the bottom is given by the following first-order ODE:

$$f(t,h) = \frac{dh}{dt} = \frac{-\mu\, r^2 \sqrt{2gh}}{2hR - h^2}$$

where $R = 10\,\mathrm{m}, r = 0.05\,\mathrm{m}, g = 9.81\ m/s^2, \mu = 0.85$. We have two questions:

1. How high will be the water level in the tank after 12 hours?
2. How much time does it take to completely drain the tank?

Solving a differential equation always starts with rearranging the equation into a form where we have the derivative on side and all the other variables and constant on the other. In our case, the ODE is already given in this form, so we can skip this step. The function that is equal to the derivative will be our f function.

We have to define the function in MATLAB and solve it using Euler's method and a step size of 60 seconds. It is possible that the derivative function f itself does not contain the independent variable t. Nevertheless, we still have to define the function in MATLAB using the independent variable t as well.

Definition of the constants:

```
R = 10;
r = 0.05;
g = 9.81;
mu = 0.85;
```

The derivative function:

```
f = @(t, h) -mu*r^2*sqrt(2*g*h)/(2*h*R - h.^2) % Note that the expression is given as a functio
```

```
f = function_handle with value:
    @(t,h)-mu*r^2*sqrt(2*g*h)/(2*h*R-h.^2)
```

Defining the initial value, the interval of the solution and the step size:

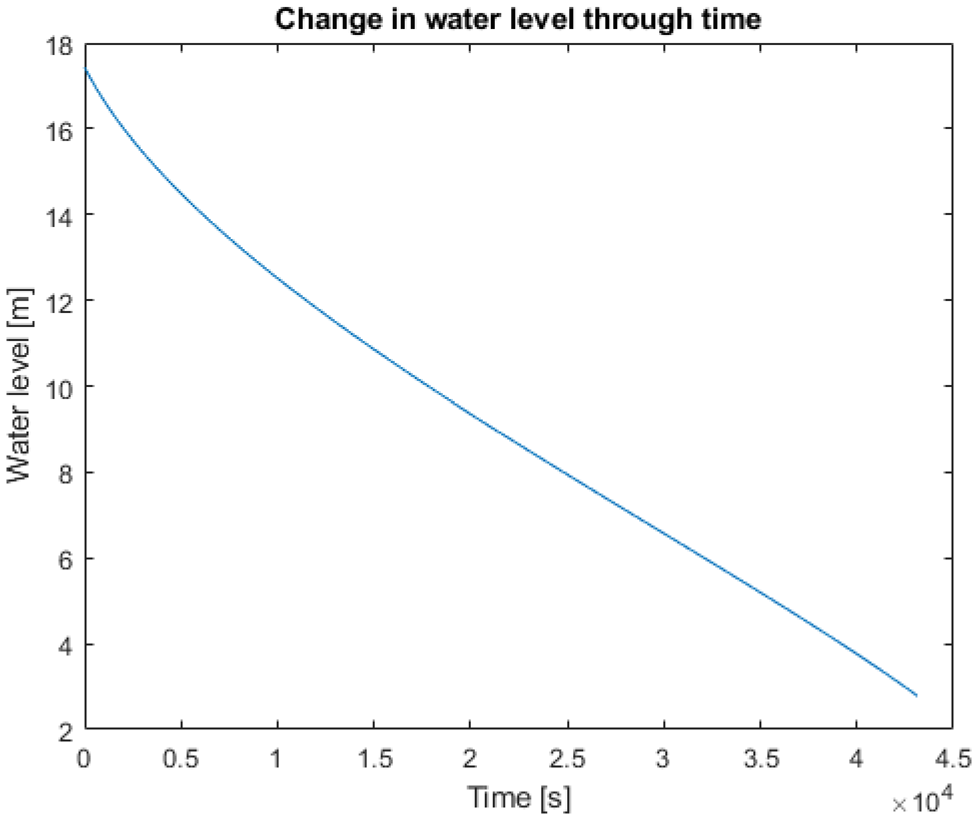
```
h0 = 17.44; % initial water level
t0 = 0; % 0 hours is the beginning of the interval
t1 = 12*3600; % 12 hours is the end of the interval (given in seconds)
s = 60; % step size in seconds
```

The solution using Euler's method:

```
[T, H] = euler(f, h0, t0, t1, s);
```

Plotting the solution:

```
figure(1);
plot(T, H);
xlabel('Time [s]');
ylabel('Water level [m]');
title('Change in water level through time');
```



As the computation was carried out to 12 hours, the last element inside the vector **H** will give us the answer to the first question:

```
H(end)
```

```
ans = 2.7810
```

The water level after 12 hours is ~2.78 m. The method used is a first-order method ($\mathrm{O}(h)$). Before we try to answer the second the question, let's see if we can do better than just a first-order approximation.

Improvements of Euler's method (Heun's method, midpoint method, Runge-Kutta method)

A similar approach is used to estimate the function values in all of the methods mentioned in the section title, the difference lies in the way of approximating the slope of the function.

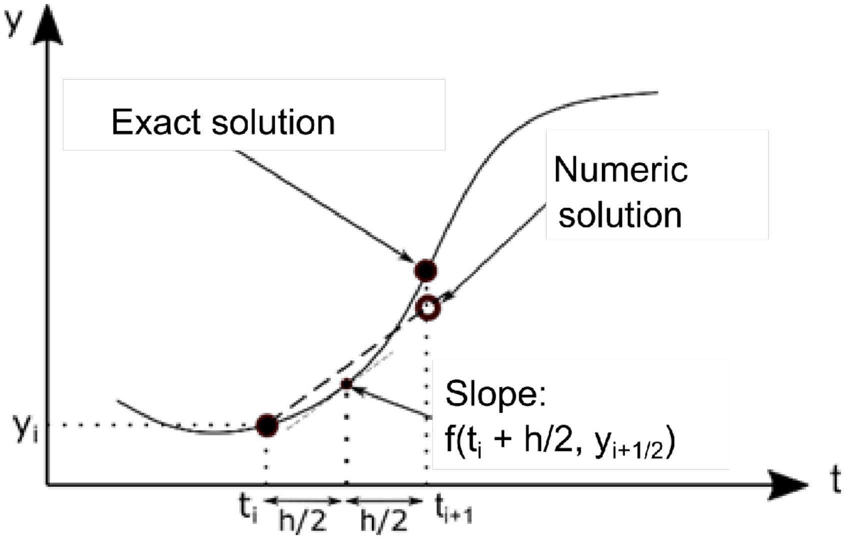
In Euler's method, the slope is calculated at the beginning of the step.

In Heun's method, the slope is calculated at the beginning (m_i) and at the end of the step interval (m_{i+1}) as well. The final value of the slope that is used is the average of the two computed values. In order to be able to calculate the slope at the end of the step size, we have to know the function value there as $m_{i+1} = f(t_{i+1}, y_{i+1})$. To be able to achieve this, a so-called predictor step is carried out first using Euler's method and the result of the predictor step is used to compute the final slope value:

1. Predictor step (Euler's method): $y_{i+1}^{(0)} = y_i + m_i \cdot h = y_i + \cdot f(t_i, y_i) \cdot h$
2. Corrector step: $t_{i+1} = t_i + h$, $m_{i+1} = f\left(t_{i+1}, y_{i+1}^{(0)}\right)$

3. Final slope value: $y_{i+1} = y_i + \frac{(m_i + m_{i+1})}{2} \cdot h = y_i + \frac{f(t_i, y_i) + f\left(t_{i+1}, y_{i+1}^{(0)}\right)}{2} \cdot h$

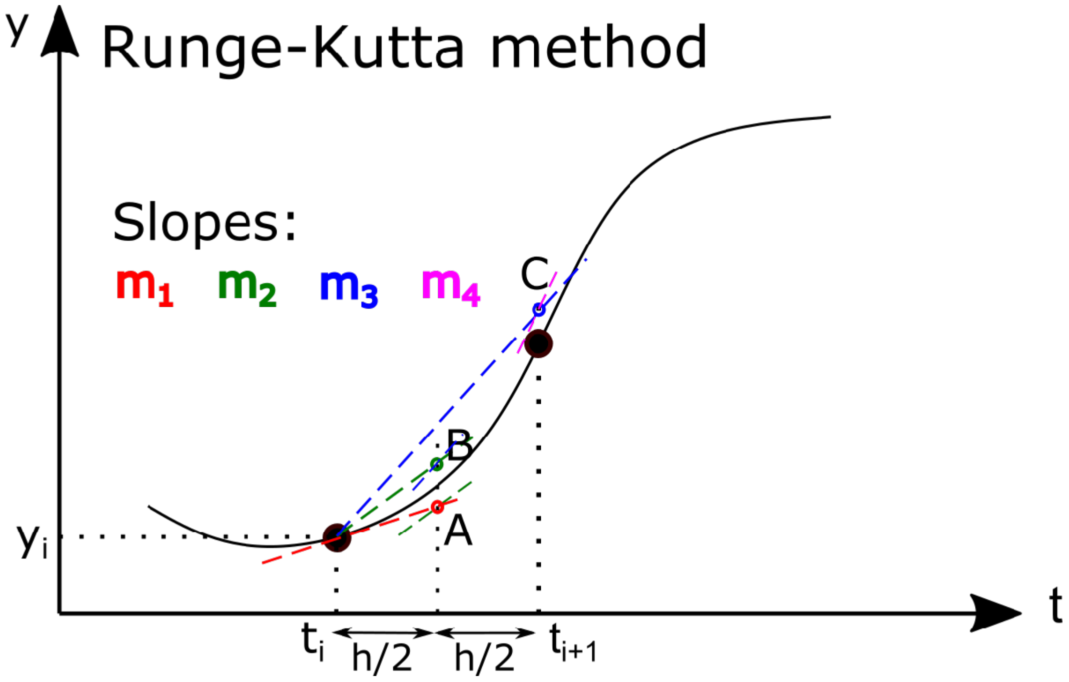
The local error of the method is $\mathrm{O}(h^3)$, its global error is $\mathrm{O}(h^2)$, which means that the method is a second-order method and is a magnitude more accurate than Euler's method.



In case of the midpoint method, the derivative is computed in the midpoint of the step and its value will be considered constant for the given step. To achieve this, the preliminary value of the function is computed for the midpoint using Euler's method and then the value of the slope is calculated:

1. Function value at the midpoint (Euler's method): $y_{i+\frac{1}{2}} = y_i + m_i \cdot \frac{h}{2} = y_i + f(t_i, y) \cdot \frac{h}{2}$
2. Slope in the midpoint: $t_{i+\frac{1}{2}} = t_i + \frac{h}{2}, \quad m_{i+\frac{1}{2}} = f\left(t_{i+\frac{1}{2}}, y_{i+\frac{1}{2}}\right)$
3. Function value at the end of the step: $y_{i+1} = y_i + m_{i+\frac{1}{2}} \cdot h$

Euler's method can be further improved if we use even more points to compute the slope and use the weighted average of these values. The most common of these methods is the fourth-order Runge-Kutta method which has a global truncation error of $\mathrm{O}(h^4)$. In MATLAB, we can use the built-in **ode45** command.



The function value at t_{i+1} is computed from the formula:

$$y_{i+1} = y_i + \frac{1}{6} \cdot (m_1 + 2m_2 + 2m_3 + m_4) \cdot h$$

where:

- $m_1 = f(t_i, y_i)$ - the slope at the beginning of the step size, point A is computed using this slope,
- $m_2 = f\left(t_i + \frac{h}{2}, y_i + m_1 \cdot \frac{h}{2}\right)$ - the slope in point A, point B is calculated using this slope,
- $m_3 = f\left(t_i + \frac{h}{2}, y_i + m_2 \cdot \frac{h}{2}\right)$ - the slope in point B, point C is computed using this slope,
- $m_4 = f(t_i + h, y_i + m_3 \cdot h)$ - the slope in point C.

The steps of the computation:

1. $m_1 = f(t_i, y_i)$
2. $y_A = y_i + m_1 \cdot \frac{h}{2} \rightarrow m_2 = f\left(t_i + \frac{h}{2}, y_A\right)$
3. $y_B = y_i + m_2 \cdot \frac{h}{2} \rightarrow m_3 = f\left(t_i + \frac{h}{2}, y_B\right)$
4. $y_C = y_i + m_3 \cdot h \rightarrow m_4 = f(t_i + h, y_C)$
5. $y_{i+1} = y_i + \frac{1}{6} \cdot (m_1 + 2m_2 + 2m_3 + m_4) \cdot h$

Solution of a first-order ODE using the Runge-Kutta method

In MATLAB, we can use the fourth-order Runge-Kutta method in the following form:

```
[T, Y] = ode45(odefun, tspan, y0)
```

where `T` and `Y` are the output of the independent variable and the solution, `odefun` is the function handle for the derivative function, `tspan` is the interval of the independent variable and `y0` is the initial value of the function at the start of the interval. The value of `tspan` can be given in two ways:

- as a vector of the start and the end of the interval, in this case, the `ode45` algorithm decides the step size automatically,
- as a vector of the values of the independent variable, in this case, we decide the step size and define it explicitely.

Solution using `ode45`:

```
% Letting the algorithm deicde the step size
[T1, H1] = ode45(f, [0, 12*3600], h0);
H1(end) % water level after 12 hours
```

```
ans = 2.7779
```

```
% Defining the step size explicitely
[T2, H2] = ode45(f, [0:60:12*3600], h0);
H2(end)
```

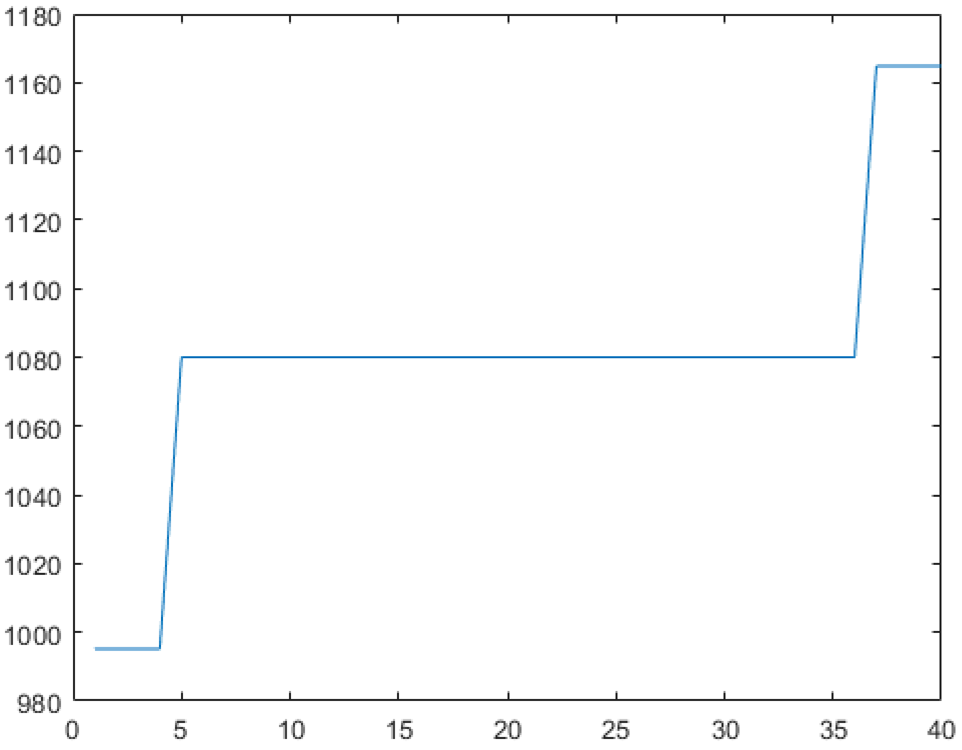
```
ans = 2.7713
```

Plotting the results:

```
figure(1);
hold on;
plot(T1, H1, 'r');
plot(T2, H2, 'g');
```

The difference in the values is very minor, only a couple of millimeters. It is worth noting however, that when the algorithm decided the step size, it wasn't constant throughout the whole interval:

```
figure(2);
plot(diff(T1));
```



The step sized varied between 995 and 1165 seconds. Smaller step sizes can usually make the approximation more accurate, however, it increases the number of function evaluations. The algorithm itself tries to decide when to use a smaller and when to use a bigger step size, depending on the difference between the function values.

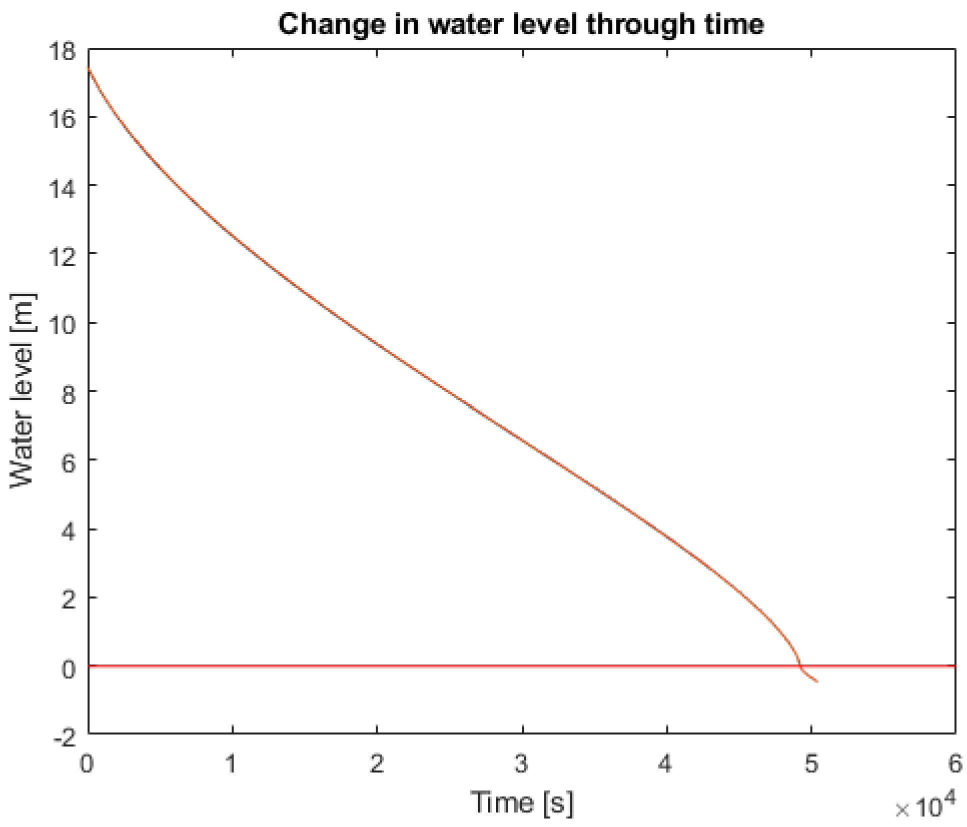
Let's anwer the second question now: how much time is needed to completely drain the tank?

To answer this, we first have to compute the function values using a longer period than 12 hours, as we have seen that after 12 hours, the water level is still higher than 2 meters. Let's use 14 hours (in seconds) for the end of our interval. We have to be careful, because of the square root inside the formula, the results become complex after the $h = 0$ line is passed. We first have to convert the complex into real ones, fit a spline to our solution data and find the intersection of our spline and the horizontal line at $h = 0$:

```
[T3, H3] = ode45(f, 0:60:14*3600, h0);
figure(1);
H3 = real(H3); % ignoring the imaginary parts
plot(T3, H3, 'k');
```

Fitting of a spline to the data:

```
sp = @(t) spline(T3, H3, t);
fplot(sp, [min(T3), max(T3)])
rl = refline(0, 0);
rl.Color = 'r';
```



Findig the intersection of the two lines (solution of $h = 0$ equation), the initial guess is from the figure:

```
h_0 = fzero(sp, 50000)
```

```
h_0 = 4.9192e+04
```

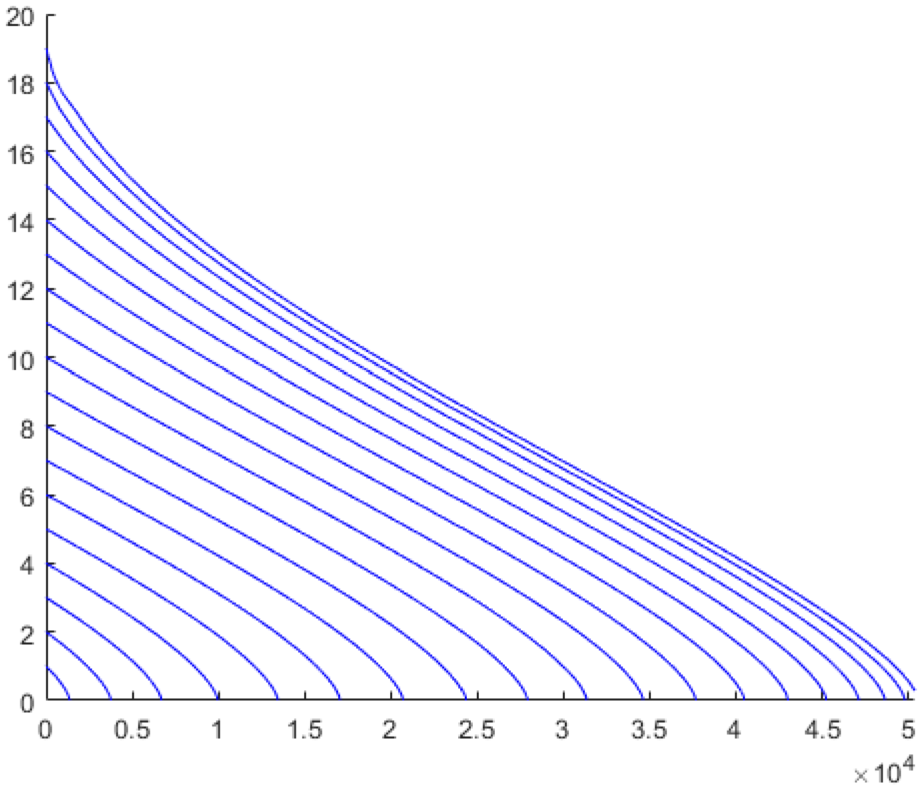
In hours:

```
h_0 / 3600
```

```
ans = 13.6644
```

What does the draining of the tank looks like if we have different initial water levels? We can visualize this by creating the trajectory or direction field of the ODE by solving it using initial values from 1 m to 20 m for example:

```
figure(3);
hold on;

for i = 20:-1:1 % go from 20 to 19, 18, ... 1
    [T, H] = ode45(f, [0, 14*3600], i);
    plot(T, real(H), 'b');
end
axis([0, 14*3600, 0, 20])
```



Solution of a system of first-order ODEs

In many cases the phenomenon we are investigating is dependent on multiple variables that can alter each other. In such cases we have to solve not a single ODE but a system of ODEs. Let the dependent variables be $y_1, y_2, \cdots, y_n$ and the independent variable be t. In a general form, the system of first-order ODEs can be written as:

$$\frac{dy_1}{dt} = f_1(t, y_1, y_2, \cdots, y_n)$$

$$\frac{dy_2}{dt} = f_2(t, y_1, y_2, \cdots, y_n)$$

$$\vdots$$

$$\frac{dy_n}{dt} = f_n(t, y_1, y_2, \cdots, y_n)$$

The initial values on the interval $[a, b]$:

$$y_1(a) = Y_1, y_2(a) = Y_2, \cdots, y_n(a) = Y_n$$

Some systems of ODEs can be solved using the generalized forms of the explicit methods mentioned before: $t_{i+1} = t_i + h$, in case of Euler's method for example:

$$
\begin{aligned}
&y_{1,i+1} = y_i + f_1(t, y_1, y_2, \cdots, y_n) \cdot h \\
&\vdots \\
&y_{n,i+1} = y_i + f_n(t, y_1, y_2, \cdots, y_n) \cdot h
\end{aligned}
$$

The improved methods can be similarly generalized. Let's see the following simple example. We are looking for the solution in the interval $[0, 1.2]$ using a step size of $h = 0.4$.

$$
\begin{aligned}
f_1 &= \frac{dx}{dt} = x \cdot t - y \qquad & x(0) &= 1 \\
f_2 &= \frac{dy}{dt} = y \cdot t + x \qquad & y(0) &= 0.5
\end{aligned}
$$

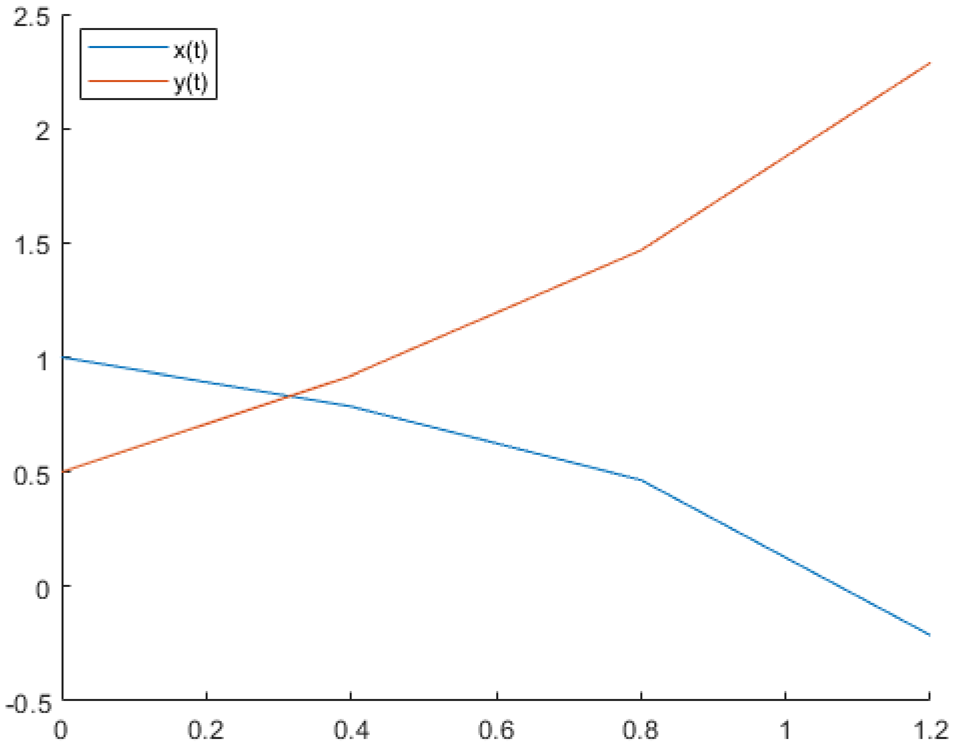
We have two equations, f_1 denotes the derivative of the first variable with respect to t and f_2 denotes the same for the second variable. Let's use the Runge-Kutta method to solve the system. We have to use vector notation when specifying the variables, for example we can use $v = [x, y]$, so that $v(1) = x$ and $v(2) = y$.

If the system is not too complicated, we can define it as an anonymous function in MATLAB:

```
clear all; close all;
odesys = @(t, v) [v(1)*t - v(2); v(2)*t + v(1)];
```

Before we can solve it, we have to define the interval, the step size and the initial values.

```
t = 0:0.4:1.2;
init = [1; 0.5];
[T, Y] = ode45(odesys, t, init);
figure(1);
hold on;
plot(T, Y(:, 1), T, Y(:, 2));
legend('x(t)', 'y(t)', 'Location', 'best');
```



If the system is more complicated, it is better practice to define it in a separate file as a regular function. Let's do this for the above example. We can create a function file called **`sysdiff.m`** and define the system:

```
function dvdt = sysdiff(t, v)
  f1 = v(1)*t - v(2);
  f2 = v(2)*t + v(1);
  dvdt = [f1; f2];
end
```

If our system is defined in a separate function file, we have to put an "@" symbol before its name when referring to it in the solver:

```
[T, Y] = ode45(@sysdiff, t, init)
```

```
T = 4×1
         0
    0.4000
    0.8000
    1.2000
Y = 4×2
    1.0000    0.5000
    0.7868    0.9207
    0.4655    1.4676
   -0.2130    2.2870
```

Second-order ODEs

Second-order ODEs with independent variable t and dependent variable y can be given in the following general form:

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right)$$

The equation can be solved on the interval $[a, b]$ if we have two know values. If these values are given at the beginning of the interval, than we call it an initial value problem. The two initial values that have to be given are the value of y and $\frac{dy}{dt}$ at the beginning of the interval. Let these initial values be denoted using A and B:

$$y(a) = A \qquad \text{and} \qquad \frac{dy}{dt}(a) = B$$

This type of second-order ODE can be transformed into a system of two first-order ODEs that can be solved similarly to the previous case. The first step of the solution is to rearrange the equation in such a way that the second derivative is expressed as a function of the other variables and constants $\left(f\left(t, y, \frac{dy}{dt}\right)\right)$. Of course, it can be the case that the second derivative is not dependent on all of the variables. As t is the independent variable, it always has to be specified in MATLAB. We can substitute new variables instead of y and $\frac{dy}{dt}$:

$$w_1 = y \qquad \text{and} \qquad w_2 = \frac{dy}{dt}$$

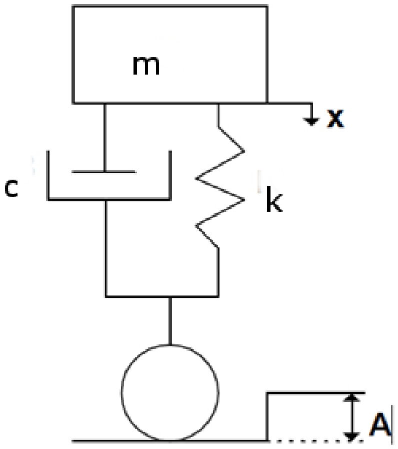
Now, we have to write to equations for the first derivatives of the two new variables and give their initial values:

$$\begin{aligned} f_1 &= \frac{dw_1}{dt} = \frac{dy}{dt} = w_2 & \qquad w_1(a) &= A \\ f_2 &= \frac{dw_2}{dt} = \frac{d^2y}{dt^2} = f(t, w_1, w_2) & \qquad w_2(a) &= B \end{aligned}$$

Using these new definitions, the second-order equation is given as a system of two first-order ODEs.

Solution of a second-order ODE in MATLAB

Let's look at the following example. We are simulating the suspension of a vehicle using the following simple model, where the vehicle travels through an obstacle with height A.



In the modell, m is the mass of the vehicle, k is the spring constant (the force in the spring is proportional to the displacement), c is the dampening factor (the dampening force is proportional to the velocity of the mass). The following data is used:

- $m = 1000\,\mathrm{kg}$
- $k = 1000\,\frac{\mathrm{kg}}{s^2}$
- $c = 500\,\frac{\mathrm{kg}}{s}$
- $A = 0.1\,\mathrm{m}$

Summing up the forces in the dampened system, we get the following second-order ODE:

$$m \cdot \ddot{x} + c \cdot \dot{x} + k \cdot (x - A) = 0$$

where x is the vertical position of the vehicle, $\dot{x}$ is the position's first derivative with respect to time (the vertical velocity of the vehicle) and $\ddot{x}$ is the second derivative of the position (the vertical acceleration of the vehicle). Defining the equation in the vehicle's coordinate system:

$$m \cdot \frac{d^2x}{dt^2} + c \cdot \frac{dx}{dt} + k \cdot (x - A) = 0$$

The initial values are the vertical displacement of the vehicle at the begining of our interval and the initial verical velocity at the beginning of the interval, both are zero:

$$x(0) = 0 \qquad \text{and} \qquad \frac{dx}{dt}(0) = 0$$

The first step is to express the second derivative from the equation:

$$\frac{d^2x}{dt^2} = \frac{1}{m} \cdot \left(kA - kx - c \cdot \frac{dx}{dt}\right) = f\left(t, x, \frac{dx}{dt}\right)$$

We have to transform this second-order equation into a system of two first-order ODEs. Before we do this, let's introduce two substitutions:

$$w_1 = x$$

$$w_2 = \frac{dx}{dt}$$

Using these substitutions, we can write two equations and their initial values:

$$f_1 = \frac{dw_1}{dt} = \frac{dx}{dt} = w_2 \qquad w_1(0) = 0$$

$$f_2 = \frac{dw_2}{dt} = \frac{d^2x}{dt^2} = \frac{1}{m}(kA - kw_1 - cw_2) \qquad w_2(0) = 0$$

We can create our system of ODEs in a separate function file **`vehicdiff.m`**. We can use the vector notation $w = [w_1, w_2]$, so that w_1 is the vertical displacement and w_2 is the vertical velocity.

```
function f = vehdiff(t, w)
    % Constants
    m = 1000; k = 1000; A = 0.1; c = 500;
    f1 = w(2);
    f2 = 1/m * (k*A - k*w(1) - c*k*w(2));
    f = [f1; f2];
end
```

Note that the variable t is given among the input variables, however, it is not usd explicitely in the equations. Let's solve the system using the Runge-Kutta method (**`ode45`** in MATLAB) specifying an absolute and a relative tolerance of 10^{-4}, on the interval 0-15 seconds.

Optional parameters can be specified for the **`ode45`** solver similarly to previous solvers, using the **`odeset()`** function. Some of the most relevant parameters:

- `RelTol` - scalar relative tolerance that is valid for each component of the function y, it measures the error relative to the magnitude of each solution function;
- `AbsTol` - scalar or vector of tolerance values that is valid for all or some of the solution functions, controls the step size of the solver;
- `MaxStep` - the largest acceptable step size,
- `InitialStep` - recommended starting step size.

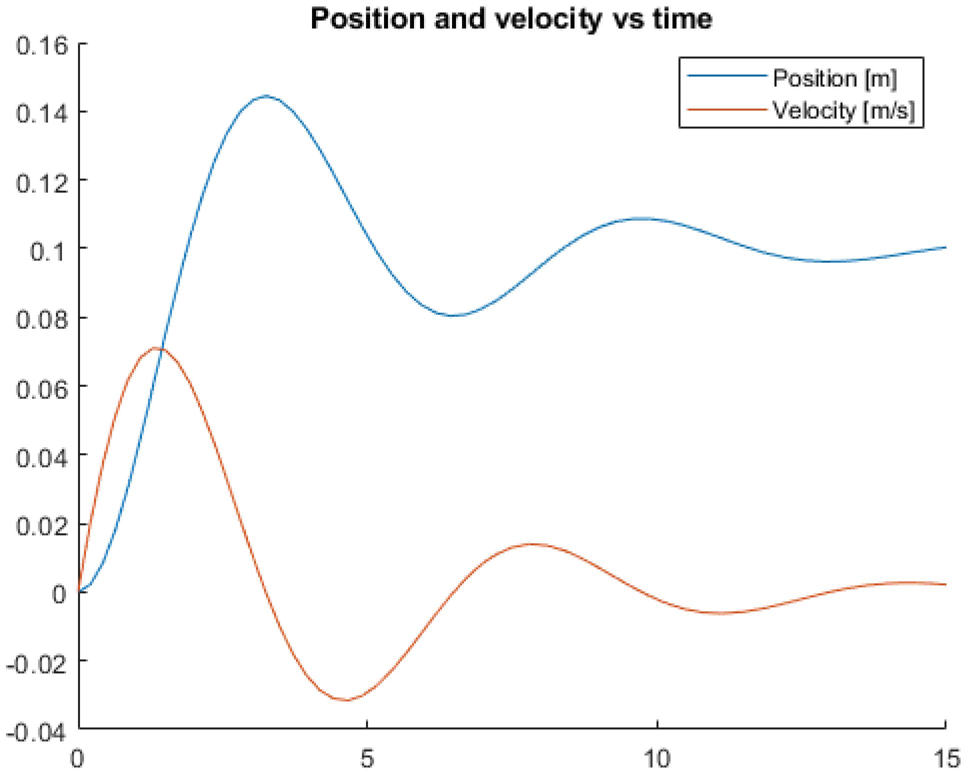
```
clear all; close all;
options = odeset('RelTol', 1e-4, 'AbsTol', [1e-4, 1e-4]);
x0 = 0; % initial position
v0 = 0; % initial vertical velocity
[T, W] = ode45(@vehdiff, [0, 15], [x0; v0], options);
```

The first column of the solution vector W contains the values of the position ($w_1 = x$), the second column contains the values of the vertical velocity $\left(w_2 = \frac{dx}{dt}\right)$.

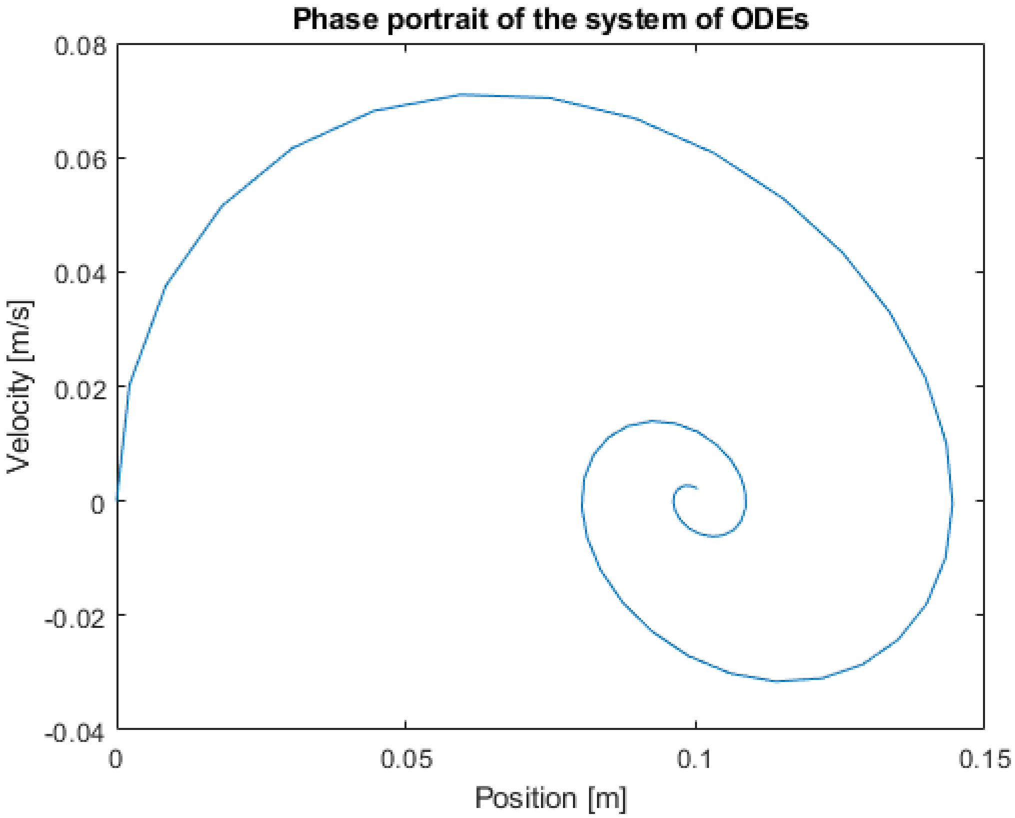
Important remark: **always include the independent variable (t in our case) as a variable of the function file that defines the system of ODEs. If the system is defined in a separate file, always include the '@' symbol before its name.**

To visualize our results, we can plot the position and the velocity values as functions of time. We can also plot the velocity as a function of the position (where time becomes a parameter of the curve), this is called the phase portrait of the system of ODEs:

```
figure(1);
hold on;
x = W(:, 1);
v = W(:, 2);
plot(T, x, T, v);
legend('Position [m]', 'Velocity [m/s]', 'Location', 'best');
title('Position and velocity vs time');
```



```
figure(2);
plot(x, v);
xlabel('Position [m]');
ylabel('Velocity [m/s]');
title('Phase portrait of the system of ODEs');
```



N-th order ODEs

When dealing with third-order, fourth-order to even higher order ODEs, let's assume an n-th order ODE, the solutions can be reduced to solving a system of n number of first-order ODEs by introducing new variables. In order to solve such an equation, we need n initial values, e.g. in the third-order case 3, in the fourth-order case 4.

The general form of an n-th order ODE:

$$\frac{d^n y}{dt^n} = f\left(t, y, \frac{dy}{dt}, \frac{d^2 y}{dt^2}, \cdots, \frac{d^{(n-1)} y}{dt^{(n-1)}}\right), \qquad a \le t \le b$$

The initial conditions:

$$y(a) = A_1, \qquad \frac{dy}{dt}(a) = A_2, \qquad \frac{d^2 y}{dt^2}(a) = A_3, \qquad \ldots, \qquad \frac{d^{(n-1)} y}{dt^{(n-1)}}(a) = A_n$$

If we introduce n new variables and their initial values, we can write the following system of ODEs:

$$\begin{aligned}
&y = w_1 \\
&\frac{dw_1}{dt} = \frac{dy}{dt} = w_2 \\
&\frac{\mathrm{dw}_2}{dt} = \frac{d^2y}{dt^2} = w_3 \\
&\vdots \\
&\frac{dw_{n-1}}{dt} = \frac{d^{(n-1)}y}{dt^{(n-1)}} = w_n \\
&\frac{dw_n}{dt} = \frac{d^ny}{dt^n} = f\left(t, y, \frac{dy}{dt}, \frac{d^2y}{dt^2}, \cdots, \frac{d^{(n-1)}y}{dt^{(n-1)}}\right)
\end{aligned}$$

$$\begin{aligned}
w_1(a) &= A_1 \\
w_2(a) &= A_2 \\
w_3(a) &= A_3 \\
&\vdots \\
w_n(a) &= A_n
\end{aligned}$$

As an example, let's solve the following third-order ODE on the interval $[0, 1]$:

$$2x - 3y + 4 \cdot \frac{dy}{dx} + x \cdot \frac{d^2y}{dx^2} - \frac{d^3y}{dx^3} = 0$$

The initial values:

$$\begin{aligned}
&y(0) = 3 \\
&\frac{dy}{dx}(0) = 2 \\
&\frac{d^2y}{dx^2}(0) = 7
\end{aligned}$$

The first step is to express the highest order derivative in the equation:

$$\frac{d^3y}{dx^3} = 2x - 3y + 4 \cdot \frac{dy}{dx} + x \cdot \frac{d^2y}{dx^2}$$

Next, we can introduce the new variables and write the system of 3 first-order ODEs:

$$w_1 = y, \qquad w_2 = \frac{dy}{dx}, \qquad w_3 = \frac{d^2y}{dx^3}$$

In the new system of ODEs, we have to define the derivatives of the new variables:

$$\begin{aligned}
f_1 &= \frac{dw_1}{dx} = \frac{dy}{dx} = w_2 \\
f_2 &= \frac{dw_2}{dx} = \frac{d^2y}{dx^2} = w_3 \\
f_3 &= \frac{dw_3}{dx} = \frac{d^3y}{dx^3} = 2x - 3y + 4 \cdot \frac{dy}{dx} + x \cdot \frac{d^2y}{dx^2}
\end{aligned}$$

$$\begin{aligned}
w_1(0) &= 3 \\
w_2(0) &= 2 \\
w_3(0) &= 7
\end{aligned}$$

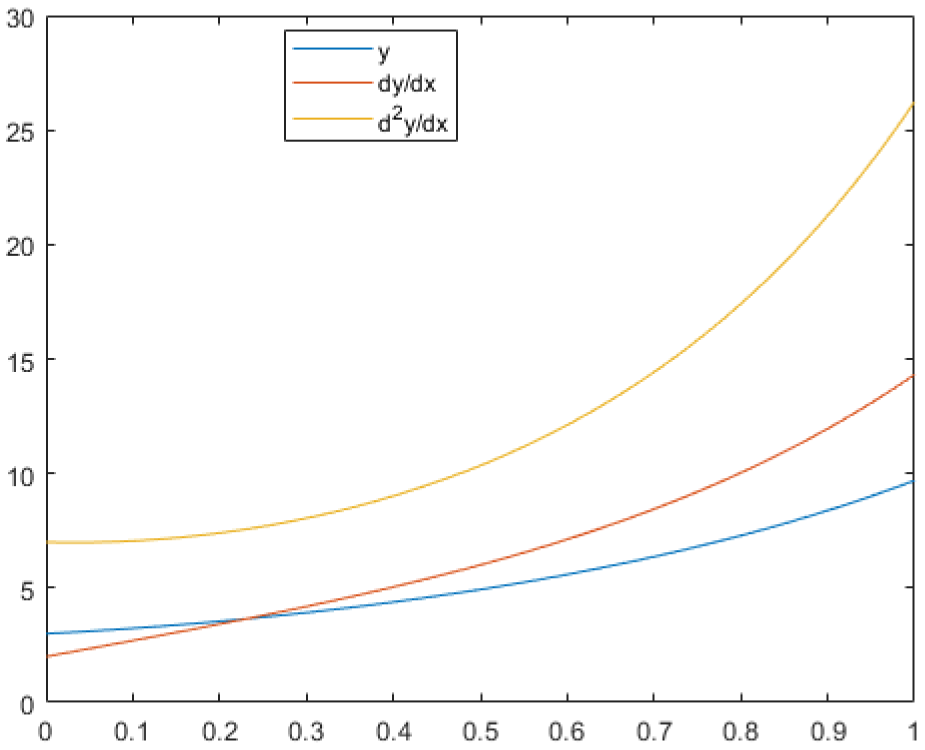
The system is created in a separate file, **`diff3.m`**:

```
function dwdx = diff3(x, w)
    f1 = w(2);
    f2 = w(3);
    f3 = 2*x - 3*w(1) + 4*w(2) + x*w(3);
    dwdx = [f1; f2; f3];
```

```
        end
```

Solution in MATLAB:

```
clear all; close all;

% Initial values
w10 = 3; w20 = 2; w30 = 7;

% Solution
[X, W] = ode45(@diff3, [0, 1], [w10; w20; w30]);

% Plotting the results
figure(1);
plot(X, W(:, 1), X, W(:, 2), X, W(:, 3));
lgd = legend('y', 'dy/dx', 'd^2y/dx', 'Location', 'best');
```



N-th order systems of ODEs

Given an n-th order system of ODEs, the solution can be reduced similarly to the example shown above by introducing new variables. For the sake of example, take the following general form of a second-order system of ODEs:

$$\frac{d^2x}{dt^2} = F_1\left(t, x, y, \frac{dx}{dt}, \frac{dy}{dt}\right)$$
$$\frac{d^2y}{dt^2} = F_2\left(t, x, y, \frac{dx}{dt}, \frac{dy}{dt}\right)$$

We have to define 4 new variables:

$$w_1 = x, \qquad w_2 = y, \qquad w_3 = \frac{dx}{dt}, \qquad w_4 = \frac{dy}{dt}$$

Using these new variables, we can write a system of four first-order ODEs:

$$f_1 = \frac{dw_1}{dt} = \frac{dx}{dt} = w_3$$
$$f_2 = \frac{dw_2}{dt} = \frac{dy}{dt} = w_4$$
$$f_3 = \frac{dw_3}{dt} = \frac{d^2x}{dt^2} = F_1\left(t, x, y, \frac{dx}{dt}, \frac{dy}{dt}\right)$$
$$f_4 = \frac{dw_4}{dt} = \frac{d^2y}{dt^2} = F_2\left(t, x, y, \frac{dx}{dt}, \frac{dy}{dt}\right)$$

The solution is analogous to the previous example.

Differential equations - Boundary Value Problems

Last time we studied initial value problems of ordinary differential equations (ODEs). In the first order case, the value of the unknown function, in the second and higher order case, the value of the unknown function and derivatives up to the order of the equation were given at the start of the solution interval.

In the case of boundary value problems, at least one of these given values are for the end of the solution interval. As an example, let's look at the following second order ODE on the interval $[a, b]$.

$$\frac{d^2y}{dt^2} = f(t, y, \frac{dy}{dt})$$

As this is a second order DE, we need two values to find a unique solution. If it is an initial value problem, one of these is the value of the unknown function, the other is the value of the first derivative of the function and both of these values are given at a (the start of the interval).

However, in the case of the boundary value problem, there can be more than one option. If the value of the function is given at the start and at the end of the interval as well, it is called a Dirichlet problem.

$$y(a) = Y_a \qquad y(b) = Y_b$$

Another possibility is to specify the values of the first derivative on the boundaries, this is called the Neumann problem:

$$\frac{dy}{dt}(a) = D_a \qquad \frac{dy}{dt}(b) = D_b$$

Conditions specifying the value of the function and the derivative can be mixed as well, e.g. the function value is given at the beginning of the interval and the derivative is given at the end of the interval.

We will deal with the general cases, when the problem can be transformed into an explicit system of ODEs. In such a case, we have to solve the following system of ODEs:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y})$$

On the two boundaries ($t = a$ and $t = b$), the following values are given:

$$\begin{aligned} \mathbf{y}_i &= \mathbf{A}_i \qquad i = 1, \cdots, k \\ \mathbf{y}_j &= \mathbf{B}_j \qquad i = k + 1, \cdots, n \end{aligned}$$

One method of solution is that we reduce the problem into solving an initial value problem multiple times. This is called the shooting method.

Shooting method

The idea behind this method is that we solve the initial value problem for an arbitrarily specified $\mathbf{y}_j(a) = \mathbf{u}_j$ value and check whether the $\mathbf{y}_j(b) = \mathbf{B}_j$ condition is satisfied (by solving the system of ODEs numerically). If the condition is not met, we modify the value of u. Let's suppose that the connection between the unknown $\mathbf{u}_j$ values and the $\mathbf{y}_j(b)$ values are given by a function $\mathbf{g}(u)$, that is:

$$\mathbf{y}_j(b) = \mathbf{g}_j(\mathbf{u})$$

Given this function, the unknown initial values are given by finding the solutions of the following system containing $(n - k)$ number of equations:

$$\mathbf{h}_j(\mathbf{u}) = \mathbf{g}_j(\mathbf{u}) - \mathbf{y}_j(b) = 0$$

Example for the shooting method

Let's look at the following example. We shoot a fireworks pellet into the air that explodes after 5 seconds. The vertical movement of the pellet (omitting drag and other effects) is given by the following second order ODE:

$$\frac{d^2y}{dt^2} = -g$$

What is the needed initial velocity if we want the pellet to explode at exactly 40 meters from the ground?

First, we have to reduce the problem into solving a system of two first order ODEs, as seen in the previous practical. In order to do this, we introduce two new variables ($\mathbf{w} = [w_1, w_2]$). $w_1 = y$ denotes the vertical position of the pellet and $w_2 = \frac{dy}{dt}$ denotes the vertical velocity of the pellet. Now, the system of first order ODEs is given by the derivatives of the new variables:

$$\begin{aligned} f_1 &= \frac{dw_1}{dt} = \frac{dy}{dt} = w_2 \\ f_2 &= \frac{dw_2}{dt} = \frac{d^2y}{dt^2} = -g \end{aligned}$$

The boundary conditions:

$$y(0) = 0 \qquad y(5) = 40$$

That is, the vertical position at the time of launch is 0 m, after 5 seconds, when the pellet explodes, it is 40 m. The question is that what value should the first derivative have at the time of launch in order for the $y(5) = 40$ to be satisfied.

Let's first solve the system using the Runge-Kutta method and by defining different initial values for the velocity. Let $w_2(0) = 20, 30, 40, 50\text{m/s}$. Let the vector $\mathbf{w} = [w_1, w_2], where$ is the vertical position and w_2 is the vertical velocity (the first derivative of the position).

The system can be given in a separate file (diff_pellet.m):

```
function F = diff_pellet(t, w)
    g = 9.81;
    f1 = w(2);
    f2 = -g;
    F = [f1; f2];
end
```

Or we can also define it as an anonymous function, given that the system is not too complex:

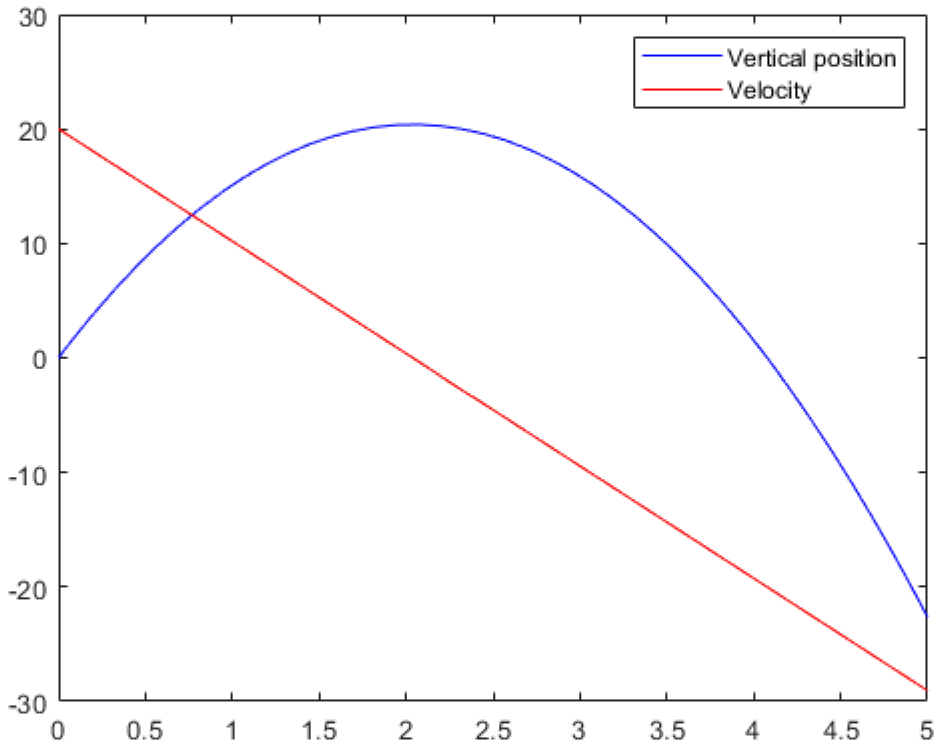
```
g = 9.81;
dwdt = @(t, w) [w(2); -g];
```

To solve the system with the Runge-Kutta method, use the **ode45** command in MATLAB (if the system is defined in a separate file, the "@" symbol has to be used before the name of the file, when calling **ode45**). Let's specify 20 m/s as the initial value of the velocity and $0 \le t \le 5$ as our solution interval:

```
ta = 0;
tb = 5;
y0 = 0;
v0 = 20;
% if the system is defined in a separate file
% [T, W] = ode45(@diff_pellet, [ta; tb], [y0; v0]);

% if the system is defined as an anonymous function
[T, W] = ode45(dwdt, [ta; tb], [y0; v0]);
```

The vector $\mathbf{T}$ contains the steps of the independent variable on the interval $[0,5]$. The matrix $\mathbf{W}$ contains two columns and as many rows as there are elements in $\mathbf{T}$. The first column of $\mathbf{W}$ contains the vertical position values (y), the second column contains the vertical velocities $\left(\frac{dy}{dt}\right)$. Let's visualize the results on a plot:

```
figure(1);
Y = W(:, 1);
V = W(:, 2);
plot(T, Y, 'b', T, V, 'r');
legend('Vertical position', 'Velocity');
```

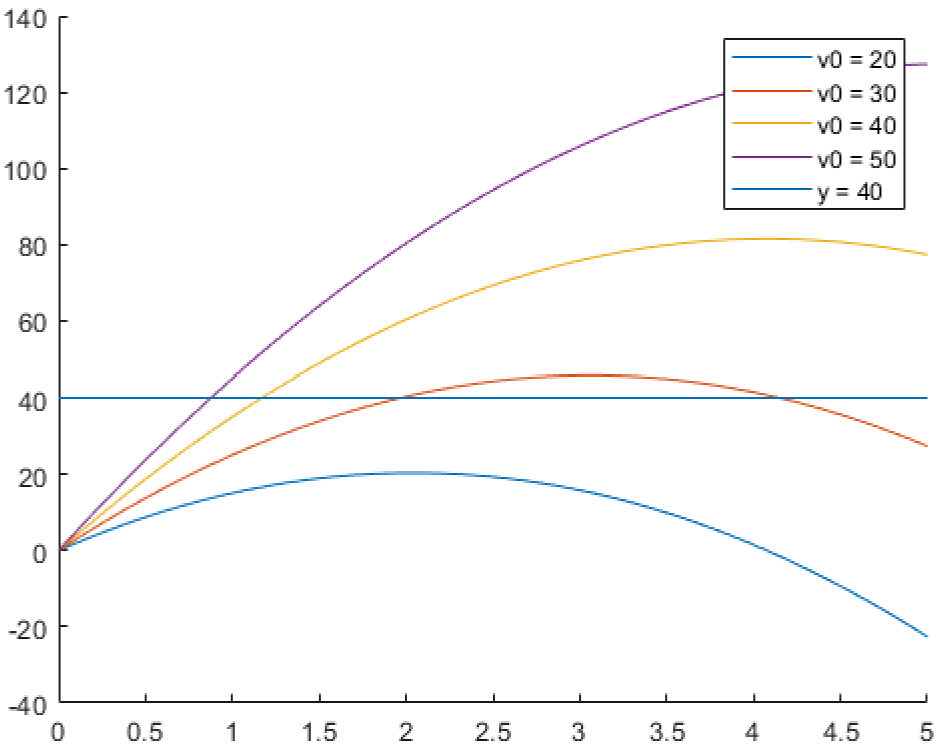


From the figure, we can see, that using 20 m/s as the initial velocity takes us very far from reaching the desired height of 40 m after 5 seconds. Let's try to increase the initial velocity (to 30, 40 and 50 m/s) and plot the vertical positions. Print the values of the position after 5 seconds for each initial velocity:

```
figure(2);
hold on;
for vi = 20:10:50
    [T, W] = ode45(dwdt, [ta; tb], [y0; vi]);
    fprintf('v0 = %d m/s -- y(5) = %.3f m\n', vi, W(end, 1));
    plot(T, W(:, 1));
end
```

```
v0 = 20 m/s -- y(5) = -22.625 m
v0 = 30 m/s -- y(5) = 27.375 m
v0 = 40 m/s -- y(5) = 77.375 m
v0 = 50 m/s -- y(5) = 127.375 m
```

```
refline(0, 40);
legend('v0 = 20', 'v0 = 30', 'v0 = 40', 'v0 = 50', 'y = 40');
```



From the figure and the printed values, we can see if the initial velocity is 30 m/s, the pellet fall under 40 m when it explodes and when the initial velocity is 40 m/s, the explosion happens above 40 m. The solution is somewhere between the two values.

Let the unknown initial velocity be u and define the position at the end of the 5 seconds as a function g of this u unknown velocity. The value of this function has to be equal to 40:

$$w_1 = g(u) = 40$$

In other words, we are looking for the root of the $h = g(u) - 40$ equation.

Let's first define this function g in a separate file, that is a function of the unknown initial velocity and name it **pellet_height.m**. This will use the **diff_pellet.m** file inside:

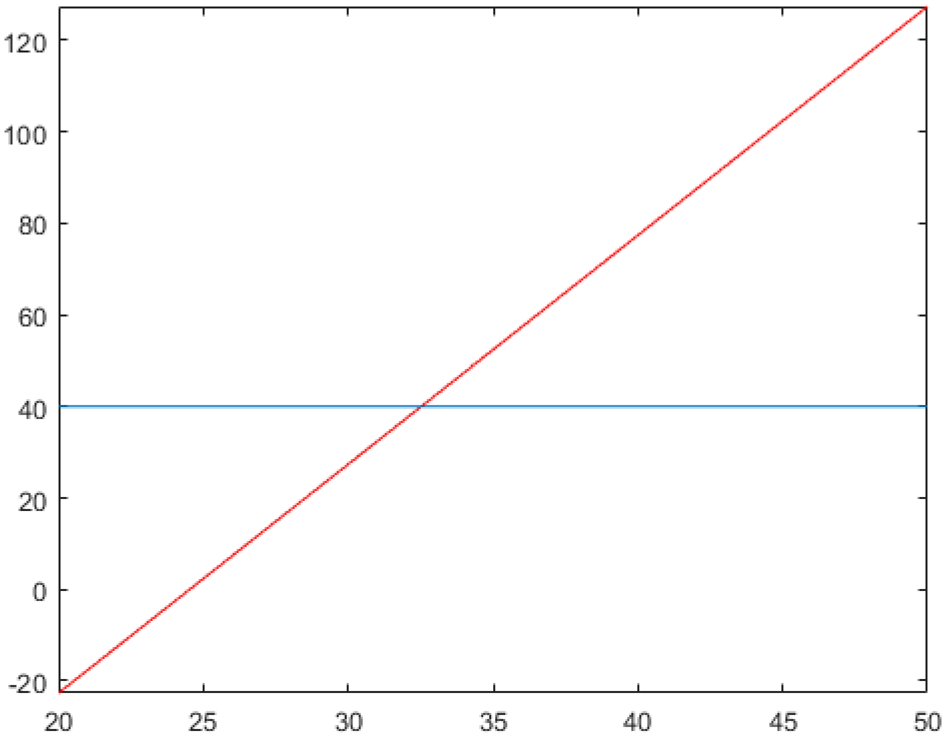
```
function y = pellet_height(u)
    ta = 0;
    tb = 5;
    y0 = 0;
    [T, W] = ode45(@diff_pellet, [ta; tb], [y0; u]);
    y = W(end, 1); % the final position is given in the
                   % last element of the first column of W
end
```

Plot the final heights as a function of the initial velocity between 20 and 50 m/s:

```
figure(3);
fplot(@pellet_height, [20, 50], 'r');
```

```
Warning: Function behaves unexpectedly on array inputs. To improve performance, properly vectorize your
function to return an output with the same size and shape as the input arguments.
```

```
hold on;
refline(0, 40);
```



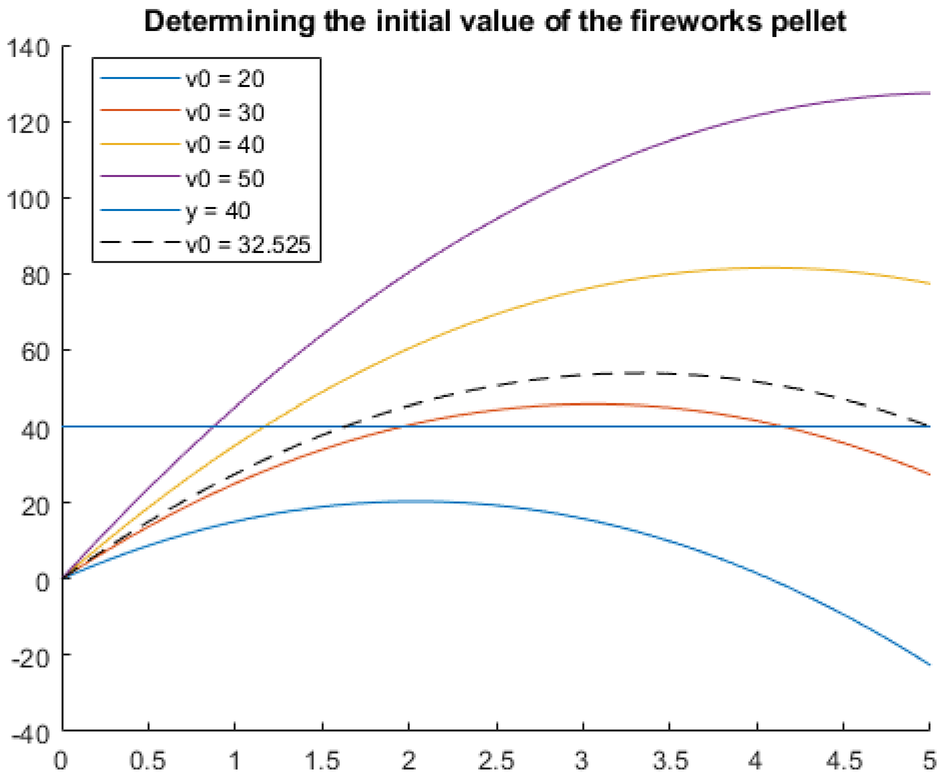
According to the figure, the solution is somewhere between 30 and 35 m/s. Let our initial guess be $u_0 = 32$.

```
h = @(u) pellet_height(u) - 40;
v0 = fzero(h, 32)
```

```
v0 = 32.5250
```

Let's plot the solution using dashed line on figure 2:

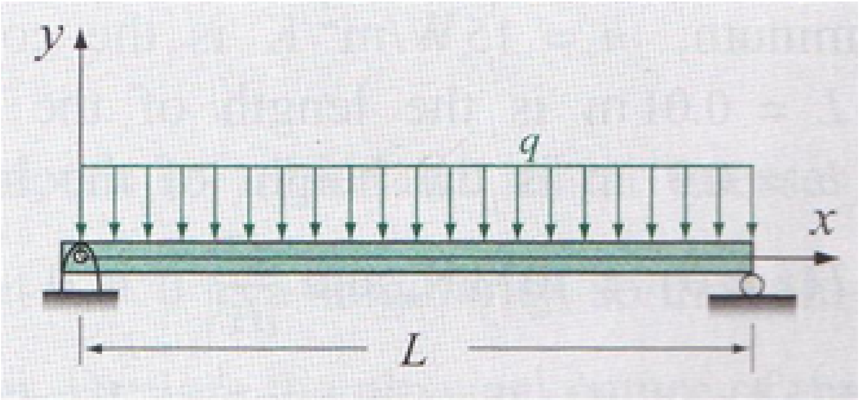
```
[T, W] = ode45(@diff_pellet, [ta; tb], [y0; v0]);
figure(2);
plot(T, W(:, 1), 'k--');
legend('v0 = 20', 'v0 = 30', 'v0 = 40', 'v0 = 50', 'y = 40', ...
    sprintf('v0 = %.3f', v0), 'Location', 'best');
title('Determining the initial value of the fireworks pellet');
```



There are many other methods for solving boundary value problems (minimizing global/local residuals, homotopy method, finite difference method etc). Due to time constraints, we do not deal with these now.

Using the built-in solver in MATLAB

MATLAB contains a built-in method for solving ODEs with boundary value problems, the **bvp4c** command (bvp = boundary value problem). In the following, we will look at an example using this method:



A 4m long beam is given with supports at both ends and a constant load (q) on top. For large deflections, the value of the deflection (y) along the beam can be computed using the following ODE:

$$EI\,\frac{d^2y}{dx^2} = \left[1 + \left(\frac{dy}{dx}\right)^2\right]^{\frac{3}{2}} \cdot \frac{1}{2} \cdot q \cdot (L \cdot x - x^2)$$

with the boundary values $y(0) = 0$ and $y(L) = 0$. The constants in the equation:

- EI - flexural rigidity: $1.4 \cdot 10^7\,\mathrm{N}/m^2$
- q - constant load: $10 \cdot 10^3\,\mathrm{N}/m$

Determine the function of the deflection with respect to x using the built-in solver in MATLAB. Plot the results and the first derivative as well. How much is the deflection at 1.35 m? What is the maximum deflection? At which x value is the deflection exactly 1 mm?

The built-in **bvp4c** command uses the following syntax:

```
SOL = bvp4c(ODEFUN, BCFUN, SOLINIT)
```

The command has one output, which is a structure containing the independent variable (`sol.x`), the function values and the derivatives as well (`sol.y`). The command takes three inputs:

- `ODEFUN`: the system of first order ODEs (as a function handle),
- `BCFUN`: the boundary values given as a function,
- `SOLINIT`: the interval of the solution and the approximated average values of the function and its derivatives (can be given using the **bvpinit** command).

Definition of the system of first order ODEs (ODEFUN)

Similarly to when using the **ode45** command, we first have to define the system of first order ODEs the second order equation is reduced to. To do this, we have to express the second derivative from the equation first:

$$\frac{d^2y}{dx^2} = \left(\left[1 + \left(\frac{dy}{dx}\right)^2\right]^{\frac{3}{2}} \cdot \frac{1}{2} \cdot q \cdot (L \cdot x - x^2)\right) \cdot \frac{1}{EI}$$

Then, we can introduce our new variables and reduce the equation into a system of first order ODEs. Let $w_1 = y$ and $w_2 = \frac{dy}{dx}$. Our system of first order ODEs look like the following:

$$
\begin{aligned}
f_1 &= \frac{dw_1}{dx} = \frac{dy}{dx} = w_2 \\
f_2 &= \frac{dw_2}{dx} = \frac{d^2y}{dx^2} = \left(\left[1 + w_2^2\right]^{\frac{3}{2}} \cdot \frac{1}{2} \cdot q \cdot (L \cdot x - x^2) \right) \cdot \frac{1}{EI}
\end{aligned}
$$

We can define the system of equations in a separate file called **`diff_beam.m`**:

```
function F = diff_beam(x, w)
   EI = 1.4e7;
   q = 10e3;
   L = 4;
   f1 = w(2);
   f2 = ((1 + w(2)^2)^(3/2)*(1/2)*q*(L*x - x^2))/EI;
   F = [f1; f2];
end
```

Definition of the boundary values as a function (BCFUN)

The boundary values for the problem are the following:

$$y(0) = 0 \qquad y(L) = 0$$

In order to use **`bvp4c`**, these boundary values have to be given as a function as well (BCFUN). The boundary conditions on the interval $[a, b]$ have to be given as a vector w_a and vector w_b. The vector w_a contains the conditions for the start of the interval, while w_b contains the conditions for the end of the interval. The first elements of these vectors $(w_a(1), w_b(1))$ are the values of the unknown function ($w_1 = y$) at the start/end of the interval, the second elements are the values of the first derivatives $\left(w_2 = \frac{dy}{\mathrm{dx}}\right)$ at the start/end of the interval and so on.

In the case of our problem, only the function values are given, therefore w_a and w_b only contain one value:

$$w_a(1) = 0 \qquad w_b(1) = 0$$

Let the function of the boundary value be denoted by g. The function of the boundary value can be defined similarly to when searching for the root of an equation, its expression has to have zero on one of the sides:

$$
\begin{aligned}
g_1 = w_a(1) - 0 = 0 \\
g_2 = w_b(1) - 0 = 0
\end{aligned}
$$

These functions in essence define the residuals. If these are 0, the solution functions satisfies the given criteria. In our case, as the boundary values are 0 for both boundaries the above equations simplify to:

$$
\begin{aligned}
g_1 = w_a(1) \\
g_2 = w_b(1)
\end{aligned}
$$

For the sake of example, if the boundary value at the beginning of the interval for the first derivative is 2 and the boundary value at the end of the interval for the function value is 3, then our functions would look like the following:

$$g_1 = w_a(2) - 2$$
$$g_2 = w_b(1) - 3$$

We can define the boundary conditions in a separate file called **beam_bv.m**:

```
function G = beam_bv(wa, wb)
   g1 = wa(1);
   g2 = wb(1);
   G = [g1; g2];
end
```

Definition of the solution interval and the approximating the average values (SOLINIT)

The third input to the **bvp4cm** command is the interval of the solution and the approximated average values of the function and its derivatives (as constants, specified using the **bvpinit** command). Let our solution interval be $0 \leq x \leq 4$ in this case and the step size be 10 cm.

```
x0 = 0:0.1:4;
```

Now, we have to approximate the average value of the function. As we only have values on the boundaries, that are both zero, let our approximated average be zero as well for both the function and its first derivative. (If we had different boundary values, the average could be the mean of those two values.)

```
% Approximated averages
w10 = 0;
w20 = 0;

% Solution interval and approx. averages using the bvpinit command
solinit = bvpinit(x0, [w10; w20]);
```

Solution using the bvp4c command

Before we solve the ODE, specify a relative tolerance of 10^{-4}. Now, this is specified by the **bvpset** command (and not the **odeset**):

```
opts = bvpset('RelTol', 1e-4);
```

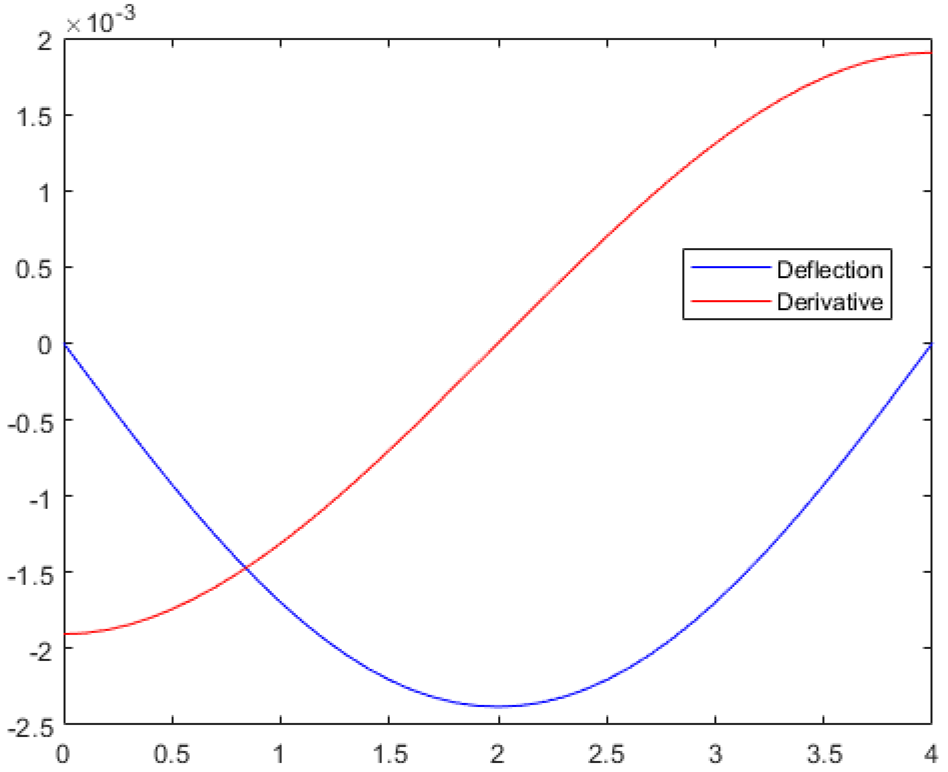
The solution of the ODE:

```
sol = bvp4c(@diff_beam, @beam_bv, solinit, opts);
X = sol.x;
W = sol.y;
```

Plot of the solution:

```
figure(4)
```

```
plot(X, W(1, :), 'b', X, W(2, :), 'r');
legend('Deflection', 'Derivative', 'Location', 'best');
```



One of the question asked about the value of the deflection at $x = 1.35\ \mathrm{m}$. Another question was that at which value of x is the deflection exactly 1 mm?

These could be answered by interpolating the data points of the function using splines for example, but we can also use the **`deval`** command. The command evaluates the solution from the **`bvp4c`** solver at any arbitrary point (in the solution interval). This command can also be used in the case of **`ode45`**. The **`deval`** command cannot be used for extrapolation!

Let's first answer the first question, that is, the value of the deflection at $x = 1.35\ \mathrm{m}$:

```
e1 = deval(sol, 1.35)    % this will return the value of the function and its derivative as wel
```

```
e1 = 2×1
   -0.0021
   -0.0009
```

```
e2 = deval(sol, 1.35, 1) % this will only return the value of the function
```

```
e2 = -0.0021
```

The deflection is 2.1 mm at $x = 1.35\ \mathrm{m}$.

We can define a new function called `defl` as an anonymous function using the **`deval`** command to compute the deflection at any point:

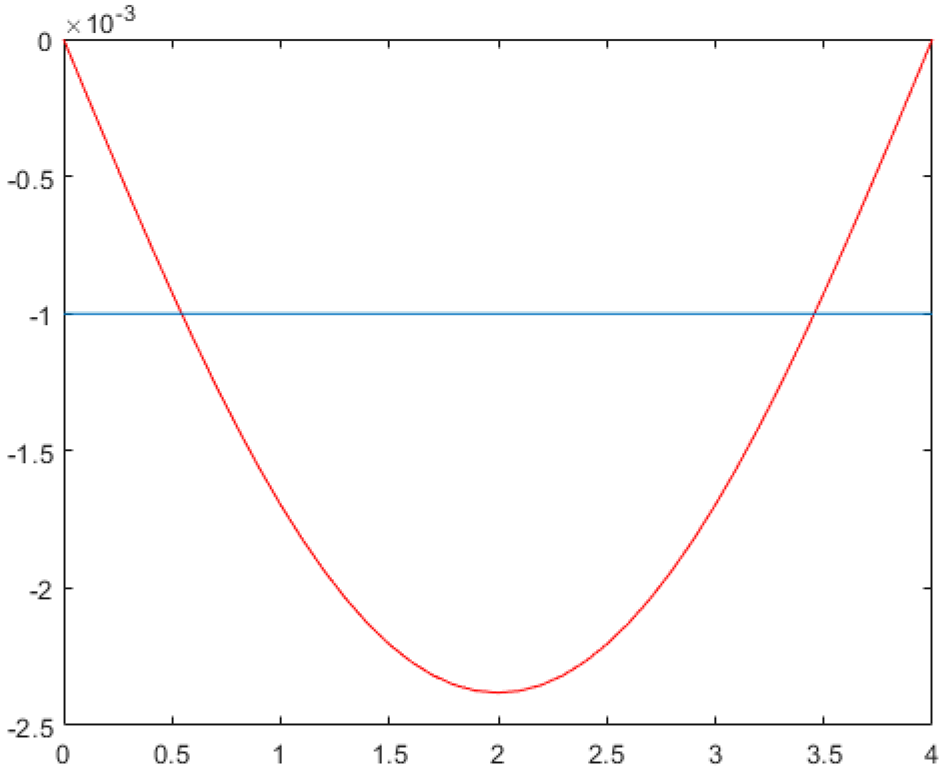
```
defl = @(x) deval(sol, x, 1);
```

The second question can be answered by defining a new function $h(x)$ and solving the following equation (remember, the deflections are negative values!):

$$\mathit{defl}(x) = -0.001 \rightarrow h(x) = \mathit{defl}(x) + 0.001 = 0$$

Before we do that, let's look at the figure to get an idea about the initial guesses:

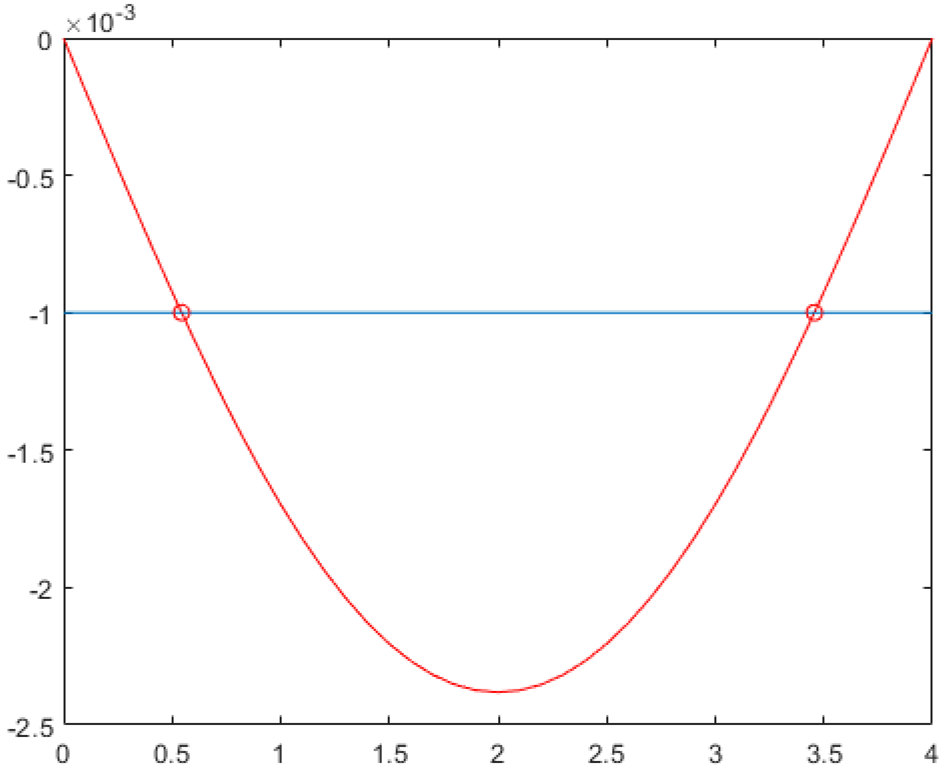
```
figure(5);
plot(X, W(1, :), 'r')
rl = refline(0, -0.001);
```



```
% initial guesses
x01 = 0.5;
x02 = 3.5;

% definition of h(x)
h = @(x) defl(x) + 0.001;

% solution
x1 = fzero(h, x01);
x2 = fzero(h, x02);
```

Plotting the solutions:

```
figure(5);
hold on;
plot(x1, defl(x1), 'ro');
plot(x2, defl(x2), 'ro');
```



The last question was the position of maximum deflection. As the load was symmetric, we can already tell that the maximum deflection happened in the middle of the beam. Let's verify this numerically. As the deflection values are negative, we can use the optimization command **fminsearch** to find the minimum of the function. For the initial guess, we can use 2:

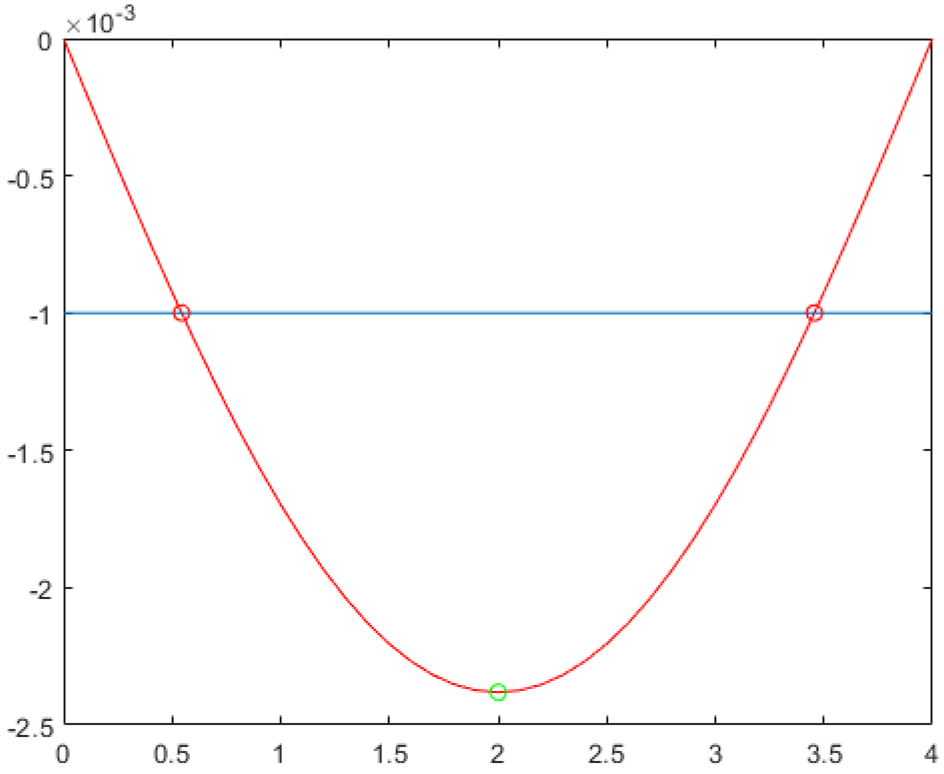
```
xmax = fminsearch(defl, 2)
```

```
xmax = 2
```

```
dmax = defl(xmax)
```

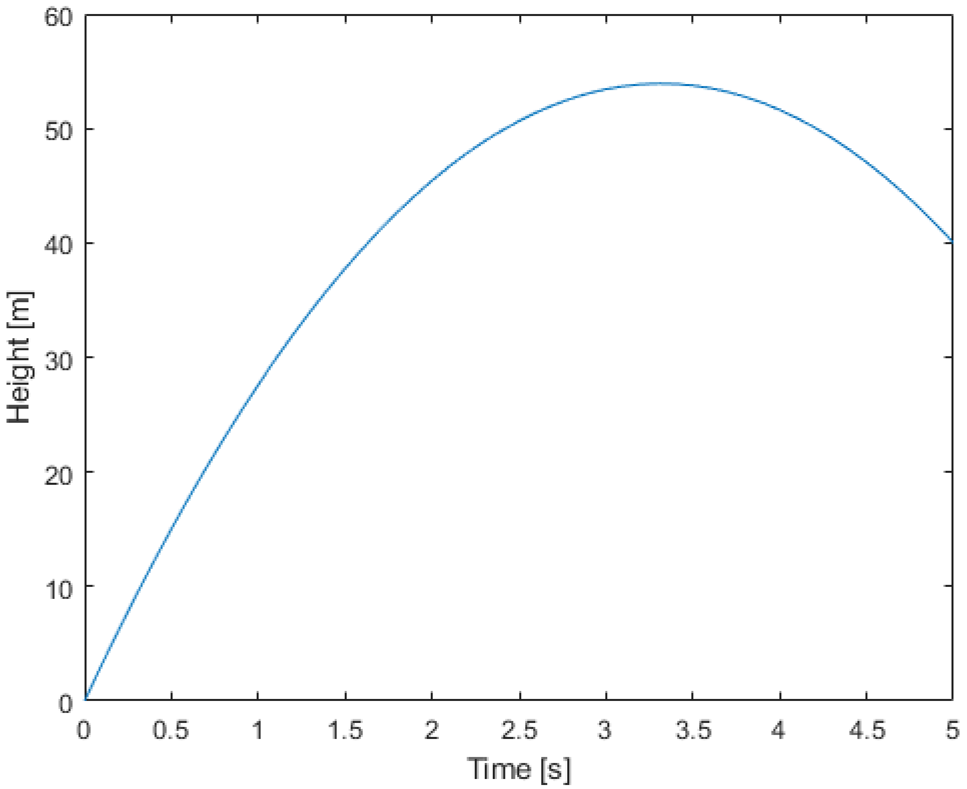
```
dmax = -0.0024
```

```
figure(5);
plot(xmax, dmax, 'go')
```



For the sake of example, let's look at how our first exercise could be solved using the built-in **bvp4c** command:

```
clear all; close all;

% Solution interval
t0 = 0:0.1:5;

% Approximated averages for the function value and its derivative
w10 = 20; % average function value (0+40)/2
w20 = 0;  % average value of the derivative
solinit = bvpinit(t0, [w10; w20]);

% Solution using the bvp4c command
opts = bvpset('RelTol', 1e-4); % relative tolerance
sol = bvp4c(@diff_pellet, @pellet_bv, solinit, opts);
X = sol.x;
W = sol.y;

% Plotting the solution
plot(X, W(1, :));
xlabel('Time [s]');
ylabel('Height [m]');
```



Practice example

Solve the following boundary value problem on the interval $[0, 1]$:

$$\frac{d^2y}{dx^2} + y = 0$$

Rearranging the expression:

$$\frac{d^2y}{dx^2} = -y$$

The given boundary conditions:

$$y(0) = 1 \qquad \frac{dy}{dx}(1) = 3$$

At the beginning of the interval, the value of the function is given, at the end of the interval, the value of the derivative is specified. We can reduce this equation into a system of first order ODEs by introducing two new variables: $w_1 = y$ and $w_2 = \frac{dy}{dx}$:

$$f_1 = \frac{dw_1}{dx} = \frac{dy}{dx} = w_2$$

$$f_2 = \frac{dw_2}{dx} = \frac{d^2y}{dx^2} = -w_1$$

The boundary conditions using the new variables, in the form required by the **bvp4c** solver:

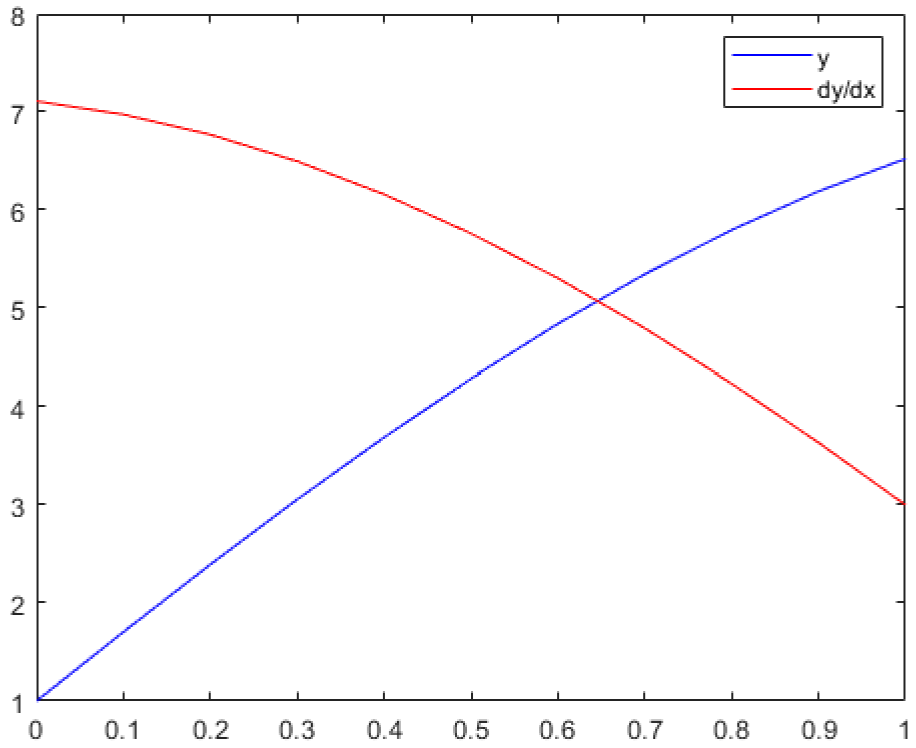
$$w_a(1) = 1 \qquad w_b(2) = 3$$

Rearranging the boundary conditions:

$$g_1 = w_a(1) - 1 \qquad g_2 = w_b(2) - 3$$

The solution in MATLAB:

```
clear all; close all;

% Solution interval (the step size is arbitrary)
x0 = 0:0.1:1;

% Approximating the function value and its derivative
w10 = 1;
w20 = 2;
solinit = bvpinit(x0, [w10; w20]);

% Solution
sol = bvp4c(@diff_practice, @practice_bv, solinit);
X = sol.x;
W = sol.y;

% Plotting the solution
plot(X, W(1, :), 'b', X, W(2, :), 'r');
legend('y', 'dy/dx');
```


8
7
6
5
4
3
2
1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
y
dy/dx