Matlab and Programming 1

Programming basics and algorithms

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2020-2021

Today's outline

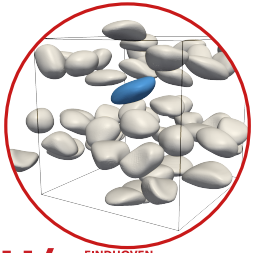
- Introduction
- Data structures
- Plotting
- Creating algorithms
- Functions
- Conclusions
- Exercises

Today's outline

- Introduction
- Data structures
- Plotting
- Creating algorithms
- Functions
- Conclusions
- Exercises

Why should you learn something about programming?

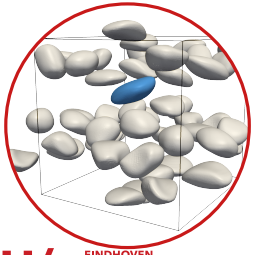
- Scientific techniques depend in an increasing fashion upon computer programs and simulation methods

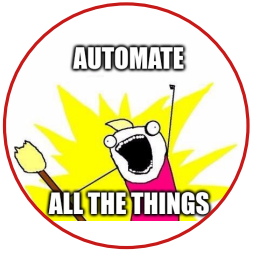


TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Why should you learn something about programming?

- Scientific techniques depend in an increasing fashion upon computer programs and simulation methods
- Knowledge of programming allows you to automate routine tasks

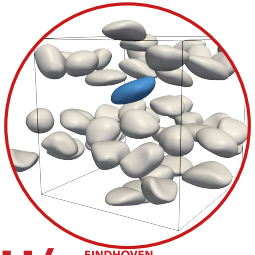


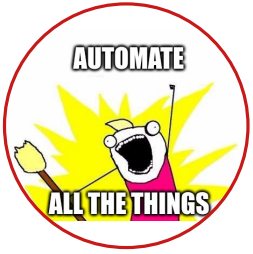


TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Why should you learn something about programming?

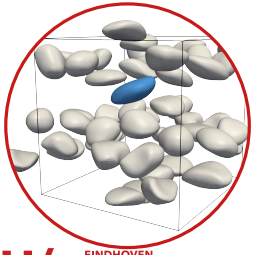
- Scientific techniques depend in an increasing fashion upon computer programs and simulation methods
- Knowledge of programming allows you to automate routine tasks
- Ability to understand algorithms by inspection of the code

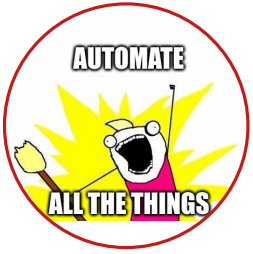




Why should you learn something about programming?

- Scientific techniques depend in an increasing fashion upon computer programs and simulation methods
- Knowledge of programming allows you to automate routine tasks
- Ability to understand algorithms by inspection of the code
- Learn to think by dissecting a problem into smaller bits





Introduction to programming

What is a program?

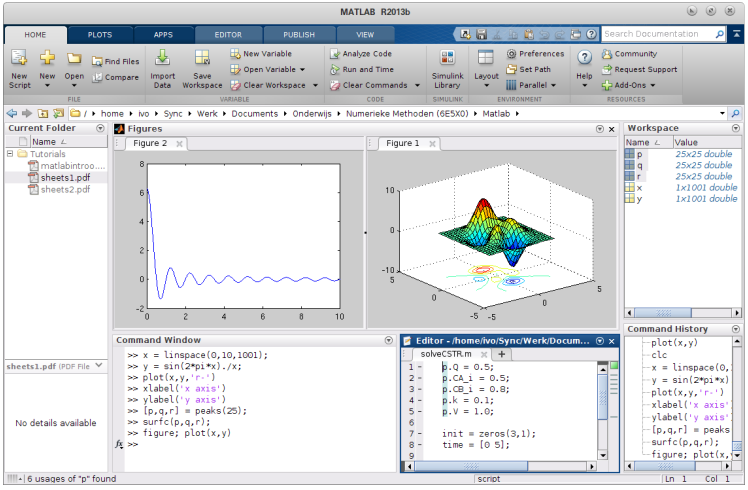
A program is a sequence of instructions that is written to perform a certain task on a computer.

- The computation might be something mathematical, a symbolic operation, image analysis, etc.

Program layout

1. Input (Get the radius of a circle)
2. Operations (Compute and store the area of the circle)
3. Output (Print the area to the screen)

Versatility of Matlab



Versatility of Matlab: ODE solver

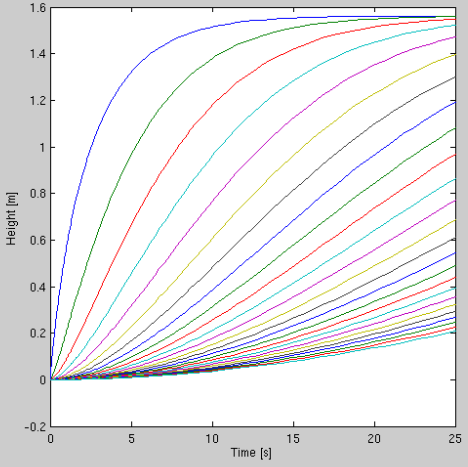
Editor - /mnt/data/BittorrentSync/Werk/Documents/Onderwijs/Pra...

solveTankSeries.m | tankSeries.m

```
1 -    p.F0 = 1.0;
2 -    p.K = 0.8;
3 -    p.N = 25;
4 -    p.A = 1.0;
5
6 -    init = zeros(p.N,1);
7 -    time = [0 25];
8
9 -    [t,h] = ode45(@tankSeries,time,init,[],p);
10
11 -   plot(t,h);
12 -   xlabel('Time [s]');
13 -   ylabel('Height [m]');
```

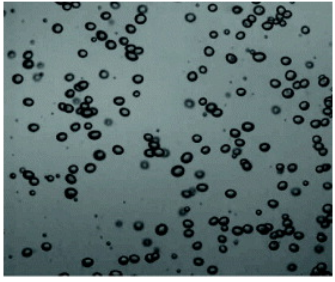
Figures - Figure 1

Figure 1



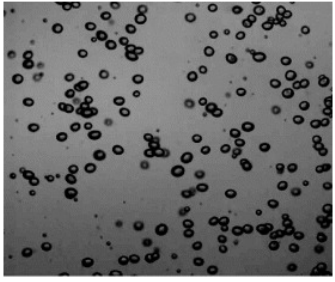
Versatility of Matlab: Image analysis

```
I = imread('bubbles.png');
BW = rgb2gray(I);
E = edge(BW, 'canny');
F = imfill(E, 'holes');
result = regionprops(F);
```



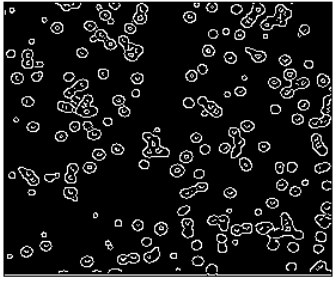
Versatility of Matlab: Image analysis

```
I = imread('bubbles.png');
BW = rgb2gray(I);
E = edge(BW, 'canny');
F = imfill(E, 'holes');
result = regionprops(F);
```



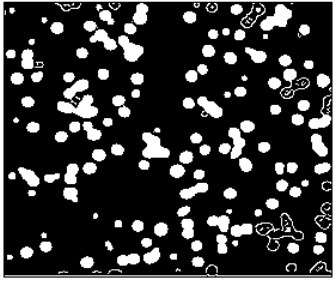
Versatility of Matlab: Image analysis

```
I = imread('bubbles.png');
BW = rgb2gray(I);
E = edge(BW, 'canny');
F = imfill(E, 'holes');
result = regionprops(F);
```

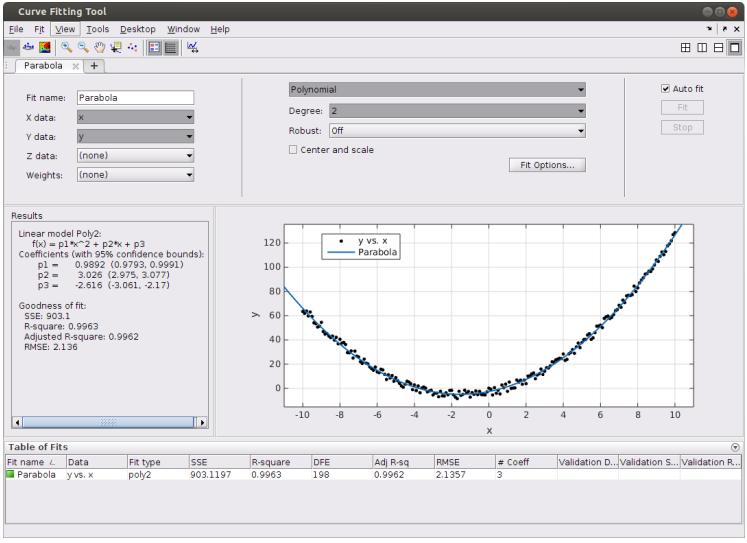


Versatility of Matlab: Image analysis

```
I = imread('bubbles.png');
BW = rgb2gray(I);
E = edge(BW, 'canny');
F = imfill(E, 'holes');
result = regionprops(F);
```



Versatility of Matlab: Curve fitting



Getting started

Start Matlab, and enter the following commands on the command line. Evaluate the output.

Getting started

Start Matlab, and enter the following commands on the command line. Evaluate the output.

```
>> 2 + 3          % Some simple calculations
>> 2*3
>> 2*3^2          % Powers are done with ^
```

Getting started

Start Matlab, and enter the following commands on the command line. Evaluate the output.

```
>> 2 + 3          % Some simple calculations
>> 2*3
>> 2*3^2          % Powers are done with ^
>> a = 2          % Storing values into the workspace
>> b = 3
>> c = (2*3)^2    % Parentheses set priority
>> 8/a-b
```

Getting started

Start Matlab, and enter the following commands on the command line. Evaluate the output.

```
>> 2 + 3          % Some simple calculations
>> 2*3
>> 2*3^2          % Powers are done with ^
>> a = 2          % Storing values into the workspace
>> b = 3
>> c = (2*3)^2    % Parentheses set priority
>> 8/a-b
>> sin(a)         % Mathematical functions can be used
>> sin(0.5*pi)    % pi is an internal Matlab variable
>> 1/0            % Infinity is a thing ...
>> sqrt(-1)       % ... as are imaginary numbers
```

Printing and formatting results

You can control the display format of the output of a command using the `format` statement. This only involves the way the numbers are printed on the screen, behind the scenes the accuracy is the same (some 15 digits).

```
>> format short % This is the default setting
>> a = 19/4
>> b = a^(-6)
>> format long
>> c = sqrt(21)
>> d = exp(-c)
>> format long e
>> d = exp(-c)
>> format short compact
>> d
```

Printing and formatting results

You can control the display format of the output of a command using the `format` statement. This only involves the way the numbers are printed on the screen, behind the scenes the accuracy is the same (some 15 digits).

```
>> format short % This is the default setting
>> a = 19/4
>> b = a^(-6)
>> format long
>> c = sqrt(21)
>> d = exp(-c)
>> format long e
>> d = exp(-c)
>> format short compact
>> d
```

A semi-colon at the end of a line suppresses output entirely:

```
>> f = pi/4;
```

A few helpful things

- Using the ↑ and ↓ keys, you can cycle through recent commands

A few helpful things

- Using the ↑ and ↓ keys, you can cycle through recent commands
- Typing part of a command and pressing Tab completes the command and lists the possibilities

A few helpful things

- Using the ↑ and ↓ keys, you can cycle through recent commands
- Typing part of a command and pressing Tab completes the command and lists the possibilities
- If a computation takes too long, you can press Ctrl + C to stop the program and return to the command line. Note that you may end up with incomplete results in the workspace.

A few helpful things

- Using the ↑ and ↓ keys, you can cycle through recent commands
- Typing part of a command and pressing Tab completes the command and lists the possibilities
- If a computation takes too long, you can press Ctrl + C to stop the program and return to the command line. Note that you may end up with incomplete results in the workspace.
- Sequences of commands (programs, scripts) are contained as m-files, plain text files with the `.m` extension.

A few helpful things

- Using the ↑ and ↓ keys, you can cycle through recent commands
- Typing part of a command and pressing Tab completes the command and lists the possibilities
- If a computation takes too long, you can press Ctrl + C to stop the program and return to the command line. Note that you may end up with incomplete results in the workspace.
- Sequences of commands (programs, scripts) are contained as m-files, plain text files with the `.m` extension.
- Such m-files must be in the *current working directory* or in the Matlab *path*, the locations where Matlab searches for a command. If you try to run a script that is not in the path, Matlab will suggest to add that directory to the *path*, or to switch to that directory. Type `path` to view the current path.

A few helpful things

- Using the ↑ and ↓ keys, you can cycle through recent commands
- Typing part of a command and pressing Tab completes the command and lists the possibilities
- If a computation takes too long, you can press Ctrl + C to stop the program and return to the command line. Note that you may end up with incomplete results in the workspace.
- Sequences of commands (programs, scripts) are contained as m-files, plain text files with the `.m` extension.
- Such m-files must be in the *current working directory* or in the Matlab *path*, the locations where Matlab searches for a command. If you try to run a script that is not in the path, Matlab will suggest to add that directory to the *path*, or to switch to that directory. Type `path` to view the current path.
- Anything following a `%` symbol is regarded as a comment

A few helpful things

- Using the ↑ and ↓ keys, you can cycle through recent commands
- Typing part of a command and pressing Tab completes the command and lists the possibilities
- If a computation takes too long, you can press Ctrl + C to stop the program and return to the command line. Note that you may end up with incomplete results in the workspace.
- Sequences of commands (programs, scripts) are contained as m-files, plain text files with the `.m` extension.
- Such m-files must be in the *current working directory* or in the Matlab *path*, the locations where Matlab searches for a command. If you try to run a script that is not in the path, Matlab will suggest to add that directory to the *path*, or to switch to that directory. Type `path` to view the current path.
- Anything following a `%` symbol is regarded as a comment
- In a script, pressing Ctrl + R comments the current line (or selection), Ctrl + T does the opposite.

A few helpful things

- Using the ↑ and ↓ keys, you can cycle through recent commands
- Typing part of a command and pressing Tab completes the command and lists the possibilities
- If a computation takes too long, you can press Ctrl + C to stop the program and return to the command line. Note that you may end up with incomplete results in the workspace.
- Sequences of commands (programs, scripts) are contained as m-files, plain text files with the `.m` extension.
- Such m-files must be in the *current working directory* or in the Matlab *path*, the locations where Matlab searches for a command. If you try to run a script that is not in the path, Matlab will suggest to add that directory to the *path*, or to switch to that directory. Type `path` to view the current path.
- Anything following a `%` symbol is regarded as a comment
- In a script, pressing Ctrl + R comments the current line (or selection), Ctrl + T does the opposite.
- Clear the workspace by `clear`, clear the command window by `clc`.

Matlab help, documentation, resources

- You can look for a command with a particular purpose using the `lookfor` command:

```
>> lookfor inverse
```

Matlab help, documentation, resources

- You can look for a command with a particular purpose using the `lookfor` command:

```
>> lookfor inverse
```

- Refer to the Matlab documentation using `doc` (pops up a new window) or `help` function

Matlab help, documentation, resources

- You can look for a command with a particular purpose using the `lookfor` command:

```
>> lookfor inverse
```

- Refer to the Matlab documentation using `doc` (pops up a new window) or `help` function
  - Try for instance: `help inv` or `help help`.

Matlab help, documentation, resources

- You can look for a command with a particular purpose using the `lookfor` command:

```
>> lookfor inverse
```

- Refer to the Matlab documentation using `doc` (pops up a new window) or `help` function
  - Try for instance: `help inv` or `help help`.
- We supplied a number of basic Canvas modules: Matlab Crash Course, including small exercises.

Matlab help, documentation, resources

- You can look for a command with a particular purpose using the `lookfor` command:

```
>> lookfor inverse
```

- Refer to the Matlab documentation using `doc` (pops up a new window) or `help` function
  - Try for instance: `help inv` or `help help`.
- We supplied a number of basic Canvas modules: Matlab Crash Course, including small exercises.
- Matlab Academy: https://matlabacademy.mathworks.com

Matlab help, documentation, resources

- You can look for a command with a particular purpose using the `lookfor` command:

```
>> lookfor inverse
```

- Refer to the Matlab documentation using `doc` (pops up a new window) or `help` function
  - Try for instance: `help inv` or `help help`.
- We supplied a number of basic Canvas modules: Matlab Crash Course, including small exercises.
- Matlab Academy: https://matlabacademy.mathworks.com
- Introduction to Numerical Methods and Matlab Programming for Engineers. T. Young and M.J. Mohlenkamp (2015). GNU-licensed document, online

Matlab help, documentation, resources

- You can look for a command with a particular purpose using the `lookfor` command:

```
>> lookfor inverse
```

- Refer to the Matlab documentation using `doc` (pops up a new window) or `help` function
  - Try for instance: `help inv` or `help help`.
- We supplied a number of basic Canvas modules: Matlab Crash Course, including small exercises.
- Matlab Academy: https://matlabacademy.mathworks.com
- Introduction to Numerical Methods and Matlab Programming for Engineers. T. Young and M.J. Mohlenkamp (2015). GNU-licensed document, online
- Search the web, Reddit, YouTube, etc.

Today's outline

- Introduction
- Data structures
- Plotting
- Creating algorithms
- Functions
- Conclusions
- Exercises

Terminology

Variable Piece of data stored in the computer memory, to be referenced and/or manipulated

Function Piece of code that performs a certain operation/sequence of operations on given input

Operators Mathematical operators (e.g. `+` `-` `*` or `/`), relational (e.g. `<` `>`or `==`, and logical operators (`&&`, `||`)

Script Piece of code that performs a certain sequence of operations without specified input/output

Expression A command that combines variables, functions, operators and/or values to produce a result.

Variables in Matlab

- Matlab stores variables in the *workspace*

Variables in Matlab

- Matlab stores variables in the *workspace*
- You should recognize the difference between the *identifier* of a variable (its name, e.g. `x`, `setpoint_p`), and the data that it actually stores (e.g. 0.5)

Variables in Matlab

- Matlab stores variables in the *workspace*
- You should recognize the difference between the *identifier* of a variable (its name, e.g. `x`, `setpoint_p`), and the data that it actually stores (e.g. 0.5)
- Matlab also defines a number of variables by default, e.g. `eps`, `pi` or `i`.

Variables in Matlab

- Matlab stores variables in the *workspace*
- You should recognize the difference between the *identifier* of a variable (its name, e.g. `x`, `setpoint_p`), and the data that it actually stores (e.g. 0.5)
- Matlab also defines a number of variables by default, e.g. `eps`, `pi` or `i`.
- You can assign a variable by the `=` sign:

```
>> x = 4*3
x =
     12
```

Variables in Matlab

- Matlab stores variables in the *workspace*
- You should recognize the difference between the *identifier* of a variable (its name, e.g. `x`, `setpoint_p`), and the data that it actually stores (e.g. 0.5)
- Matlab also defines a number of variables by default, e.g. `eps`, `pi` or `i`.
- You can assign a variable by the `=` sign:

```
>> x = 4*3
x =
     12
```

- If you don't assign a variable, it will be stored in `ans`
- Clearing the workspace is done with `clear`.

Datatypes and variables

Matlab uses different types of variables:

Datatype	Example
`string`	`'Wednesday'`
`integer`	`15`
`float`	`0.15`
`vector`	`[0.0; 0.1; 0.2]`
`matrix`	`[0.0 0.1 0.2; 0.3 0.4 0.5]`
`struct`	`sct.name = 'MyDataName'`
	`sct.number = 13`
`logical`	`0` (false)
	`1` (true)

About variables

- Matlab variables can change their type as the program proceeds (this is not common for other programming languages!):

```
>> s = 'This is a string'
s =
This is a string
>> s = 10
s =
    10
```

- Vectors and matrices are essentially *arrays* of another data type. A vector of `struct` is therefore possible.
- Variables are *local* to a function (more on this later).

Vectors in Matlab (1)

A row vector:

```
>> v = [0 1 2 3]
```

Vectors in Matlab (1)

A row vector:

```
>> v = [0 1 2 3]
```

A column vector by separating elements with semi-colons:

```
>> u = [9; 10; 11; 12; 13; 14; 15]
```

Vectors in Matlab (1)

A row vector:

```
>> v = [0 1 2 3]
```

A column vector by separating elements with semi-colons:

```
>> u = [9; 10; 11; 12; 13; 14; 15]
```

Access (i.e. read) an entry in a vector:

```
>> u(2)
```

Vectors in Matlab (1)

A row vector:

```
>> v = [0 1 2 3]
```

A column vector by separating elements with semi-colons:

```
>> u = [9; 10; 11; 12; 13; 14; 15]
```

Access (i.e. read) an entry in a vector:

```
>> u(2)
```

Manipulate the value of that entry:

```
>> u(2)=47
```

Vectors in Matlab (1)

A row vector:

```
>> v = [0 1 2 3]
```

A column vector by separating elements with semi-colons:

```
>> u = [9; 10; 11; 12; 13; 14; 15]
```

Access (i.e. read) an entry in a vector:

```
>> u(2)
```

Manipulate the value of that entry:

```
>> u(2)=47
```

Get a slice of a vector:

```
>> u([2 3 4]) % With colon operator: u(2:4)
```

Vectors in Matlab (1)

A row vector:

```
>> v = [0 1 2 3]
```

A column vector by separating elements with semi-colons:

```
>> u = [9; 10; 11; 12; 13; 14; 15]
```

Access (i.e. read) an entry in a vector:

```
>> u(2)
```

Manipulate the value of that entry:

```
>> u(2)=47
```

Get a slice of a vector:

```
>> u([2 3 4]) % With colon operator: u(2:4)
```

Transposing vectors:

```
>> w = v'
```

Vectors in Matlab (2)

Manual definition may be cumbersome. A colon (:) generates a list:

```
>> a = 1:10       % Default stride is 1
>> x = -1:.1:1    % start:stride:stop specifies list
```

Vectors in Matlab (2)

Manual definition may be cumbersome. A colon (:) generates a list:

```
>> a = 1:10       % Default stride is 1
>> x = -1:.1:1    % start:stride:stop specifies list
```

Or, when you prefer to set the *number of elements* instead of the step size:

```
>> y = linspace(0,10,11)
>> p = logspace(2,6,5)
```

Vectors in Matlab (2)

Manual definition may be cumbersome. A colon (:) generates a list:

```
>> a = 1:10       % Default stride is 1
>> x = -1:.1:1    % start:stride:stop specifies list
```

Or, when you prefer to set the *number of elements* instead of the step size:

```
>> y = linspace(0,10,11)
>> p = logspace(2,6,5)
```

Manipulating multiple components:

```
>> y([1 4:7]) = 1
```

Vectors in Matlab (2)

Manual definition may be cumbersome. A colon (:) generates a list:

```
>> a = 1:10       % Default stride is 1
>> x = -1:.1:1    % start:stride:stop specifies list
```

Or, when you prefer to set the *number of elements* instead of the step size:

```
>> y = linspace(0,10,11)
>> p = logspace(2,6,5)
```

Manipulating multiple components:

```
>> y([1 4:7]) = 1
```

Or (by supplying a vector instead of a scalar):

```
>> y([1 4:7]) = 16:20 % equivalent to y([1 4 5 6 7]) = [16 17 18 19 20]
```

Practice

Given a vector

$$x = [2\ 4\ 6\ 8\ 10\ 12\ 14\ 16\ 18\ 20\ 30\ 40\ 50\ 60\ 70\ 80]$$

- Find a way to define the vector without typing all individual elements

Practice

Given a vector

$$x = [2\ 4\ 6\ 8\ 10\ 12\ 14\ 16\ 18\ 20\ 30\ 40\ 50\ 60\ 70\ 80]$$

- Find a way to define the vector without typing all individual elements
- Investigate the meaning of the following commands:

```
>> x(3)
>> x(1:5)
>> x(1:end-1)
>> y = x(5:end)
>> y(4)
>> y(4) = []
>> sum(x)
>> mean(x)
>> std(x)
>> max(x)
>> fliplr(x)
>> x(end:-1:1)
>> diff(x)
```

Practice

Given a vector

$$x = [2\ 4\ 6\ 8\ 10\ 12\ 14\ 16\ 18\ 20\ 30\ 40\ 50\ 60\ 70\ 80]$$

- Find a way to define the vector without typing all individual elements

Practice

Given a vector

$$x = [2\ 4\ 6\ 8\ 10\ 12\ 14\ 16\ 18\ 20\ 30\ 40\ 50\ 60\ 70\ 80]$$

- Find a way to define the vector without typing all individual elements
- Investigate the meaning of the following commands:

```
>> y = x(5:end)
>> y(4)
>> y(4) = []
>> sum(x)
>> mean(x)
>> std(x)
>> max(x)
>> fliplr(x)
>> diff(x)
```

Operations on vectors (1)

```
>> e = 1:5
>> f = 2*e
>> g = 4*f + 20
```

Operations on vectors (1)

```
>> e = 1:5
>> f = 2*e
>> g = 4*f + 20
>> h = e^2
```

Operations on vectors (1)

```
>> e = 1:5
>> f = 2*e
>> g = 4*f + 20
>> h = e^2
```

... wait ... what's that?

```
Error using  ^
Inputs must be a scalar and a square matrix.
To compute elementwise POWER, use POWER (.^) instead.
```

Operations on vectors (1)

```
>> e = 1:5
>> f = 2*e
>> g = 4*f + 20
>> h = e^2
```

... wait ... what's that?

```
Error using  ^
Inputs must be a scalar and a square matrix.
To compute elementwise POWER, use POWER (.^) instead.
```

Matlab uses matrix operations by default, we should use a dot operator to make operations element-wise for `*`, `/` and `^`.

```
>> e.^2
```

Operations on vectors (2)

To demonstrate the matrix product:

```
>> p = [1; 1; 1]
>> q = [1 2 3]
>> p*q   % which is not equal to q*p
```

Operations on vectors (2)

To demonstrate the matrix product:

```
>> p = [1; 1; 1]
>> q = [1 2 3]
>> p*q   % which is not equal to q*p
```

All kinds of mathematical functions on vectors typically operate on elements:

```
>> x = linspace(0,2*pi,100);
>> s = sin(x)
>> e = exp(x)
```

Matrices in Matlab

Matrix A is defined as:

$$A = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

Matrices in Matlab

Matrix A is defined as:

$$A = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

In Matlab:

```
>> A = [ 8 1 6; 3 5 7; 4 9 2]
```

Matrices in Matlab

Matrix A is defined as:

$$A = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

In Matlab:

```
>> A = [ 8 1 6; 3 5 7; 4 9 2]
```

Elements can be accessed/manipulated by the following syntax:

```
>> A(3,1) % Third row, first column, also A(3)
>> A(3,:) = [2 4 8] % Set entire third row
>> A(:,3) % Print third column
>> A(A>5) = 2 % Set elements by condition
```

Matrices in Matlab

Matrix A is defined as:

$$A = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

In Matlab:

```
>> A = [ 8 1 6; 3 5 7; 4 9 2]
```

Elements can be accessed/manipulated by the following syntax:

```
>> A(3,1) % Third row, first column, also A(3)
>> A(3,:) = [2 4 8] % Set entire third row
>> A(:,3) % Print third column
>> A(A>5) = 2 % Set elements by condition
```

There are a few functions that help creating matrices:

```
>> A = zeros(4)  % A 4x4 matrix with zeros
>> A = ones(4,1) % A 4-element vector with ones
>> A = eye(3)    % Identity matrix of 3x3
>> A = rand(3,4) % A 3x4 matrix with random numbers
```

Practice

- Find a *short* Matlab expression to create the following matrix:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 9 & 7 & 5 & 3 & 1 & -1 & -3 \\ 4 & 8 & 16 & 32 & 64 & 128 & 256 \end{bmatrix}$$

Practice

- Find a *short* Matlab expression to create the following matrix:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 9 & 7 & 5 & 3 & 1 & -1 & -3 \\ 4 & 8 & 16 & 32 & 64 & 128 & 256 \end{bmatrix}$$

- Investigate the command `max(A)`. What does it give?

Building blocks: Mathematics and number manipulation

Programming languages usually support the use of various mathematical functions (sometimes via a specialized library). Some examples of the most elementary functions in Matlab:

Command	Explanation
`cos(x), sin(x), tan(x)`	Cosine, sine or tangens of x
`mean(x), std(x)`	Mean, st. deviation of vector x
`exp(x)`	Value of the exponential function e^x
`log10(x), log(x)`	Base-10/Natural logarithm of x
`floor(x)`	Largest integer smaller than x
`ceil(x)`	Smallest integer that exceeds x
`abs(x)`	Absolute value of x
`size(x)`	Size of a vector x
`length(x)`	Number of elements in a vector x
`rem(x,y)`	Remainder of division of x by y

Today's outline

- Introduction
- Data structures
- Plotting
- Creating algorithms
- Functions
- Conclusions
- Exercises

Simple plotting

Let's make a plot of the following table

T (°C)	5	20	30	50	55
μ (Pa·s)	0.08	0.015	0.009	0.006	0.0055

Simple plotting

Let's make a plot of the following table

T (°C)	5	20	30	50	55
μ (Pa·s)	0.08	0.015	0.009	0.006	0.0055

```
>> T = [ 5 20 30 50 55 ]
>> mu = [ 0.08 0.015 0.009 0.006 0.0055]
```

```
>> plot(T,mu)
```

```
>> xlabel('Temperature [^\circC]')
>> ylabel('Viscosity [Pa s]')
>> title('Experiment 1')
```

Simple plotting

Let's make a plot of the following table

T (°C)	5	20	30	50	55
μ (Pa·s)	0.08	0.015	0.009	0.006	0.0055

```
>> T = [ 5 20 30 50 55 ]
>> mu = [ 0.08 0.015 0.009 0.006 0.0055]
```

```
>> plot(T,mu,'*')
```

```
>> xlabel('Temperature [^\circC]')
>> ylabel('Viscosity [Pa s]')
>> title('Experiment 1')
```

Simple plotting

Let's make a plot of the following table

T (°C)	5	20	30	50	55
μ (Pa·s)	0.08	0.015	0.009	0.006	0.0055

```
>> T = [ 5 20 30 50 55 ]
>> mu = [ 0.08 0.015 0.009 0.006 0.0055]
```

```
>> plot(T,mu,'r--')
```

```
>> xlabel('Temperature [^\circC]')
>> ylabel('Viscosity [Pa s]')
>> title('Experiment 1')
```

Simple plotting

Let's make a plot of the following table

T (°C)	5	20	30	50	55
μ (Pa·s)	0.08	0.015	0.009	0.006	0.0055

```
>> T = [ 5 20 30 50 55 ]
>> mu = [ 0.08 0.015 0.009 0.006 0.0055]
```

```
>> plot(T,mu,'ko-','LineWidth',2)
```

```
>> xlabel('Temperature [^\circC]')
>> ylabel('Viscosity [Pa s]')
>> title('Experiment 1')
```

Practice

Create plots of the following functions in a single figure for $x \in \{0, 2\pi\}$:

$$y_1 = \cos x$$

$$y_2 = \arctan x$$

$$y_3 = \frac{\sin x}{x}$$

Practice

Create plots of the following functions in a single figure for $x \in \{0, 2\pi\}$:

$$y_1 = \cos x$$

$$y_2 = \arctan x$$

$$y_3 = \frac{\sin x}{x}$$

Strategies to draw multiple graphs in 1 figure:

```
>> plot(x,y1,x,y2,x,y3)
```

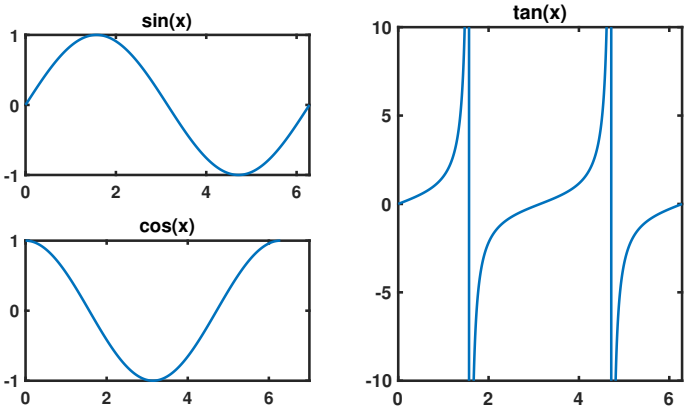
```
>> plot(x,y1)
>> hold on; % Maintain drawn plots in current figure
>> plot(x,y2)
>> plot(x,y3) % The 'hold-property' was already set
```

Multiple graph plotting

- A new figure window can be created using the `figure` command.
- The command `subplot(m,n,p)` divides the figure window into blocks using `m` rows and `n` columns. The number or vector `p` indicates which block (from left to right, top to bottom) or blocks are used for the axis.
- Graph operators (e.g. axis labels, title, hold on, etc) act on the active subplot

```
x = linspace(0,2*pi,1000)
y1 = sin(x);
y2 = cos(x);
y3 = tan(x);
subplot(2,2,1); plot(x,y); title('sin(x)');
subplot(2,2,1); plot(x,y1); title('sin(x)');
subplot(2,2,3); plot(x,y2); title('cos(x)')
subplot(2,2,[2 4]); plot(x,y3); title('tan(x)'); axis([0 2*pi -10 10])
```

Multiple graph plotting



Today's outline

- Introduction
- Data structures
- Plotting
- Creating algorithms
- Functions
- Conclusions
- Exercises

Building blocks: conditional statements

`if`-statement: Check whether a (set of) condition(s) is met, based on relational operations.

Building blocks: conditional statements

`if`-statement: Check whether a (set of) condition(s) is met, based on relational operations.

```
num = floor (10 * rand + 1);
guess = input ('Your guess please : ');
if ( guess ~= num )
  disp (['Wrong, it was ',num2str(num),'. Kbye.']);
else
    disp ('Correct !') ;
end
```

Building blocks: conditional statements

`if`-statement: Check whether a (set of) condition(s) is met, based on relational operations.

```
num = floor (10 * rand + 1);
guess = input ('Your guess please : ');
if ( guess ~= num )
  disp (['Wrong, it was ',num2str(num),'. Kbye.']);
else
    disp ('Correct !') ;
end
```

Other relational operators

==	is equal to
~=	is not equal to
<=	is less than or equal to
>=	is greater than or equal to
<	is less than
>	is greater than

Combining conditional statements

&	and
|	or
~	negation (switches true/false)
xor	exclusive or

Building blocks: conditional statements

You can use nested `if`-statements and combinations of expressions:

```
a = 3;
b = 9;

if (a < b) & (b <= 10)
    disp('a is smaller than 10 (by deduction)')
elseif ((a >= b) & (b > 10))
        disp('a is larger than 10 (by deduction)')
else
    disp('We cannot deduce if a is smaller than 10 or not, unless we perform a direct
        comparison')
    if (a < 10)
        disp('a is smaller than 10')
    elseif (a > 10)
        disp('a is larger than 10')
    else
        disp('a is equal to 10')
    end
end
```

Building blocks: logical indexing

Relational operators return a type "logical", i.e. true (1) or false (0). You can use this to select subsets of existing vectors.

Simple demonstration:

```
>> x = linspace(-3,3,7) % Create a base
     vector
x =
    -3    -2    -1     0     1     2     3
>> x>0 % Illustrate logical result
ans =
  1x7 logical array
   0   0   0   0   1   1   1
>> x2 = x(x>0) % Create a subset
x2 =
     1     2     3
```

Building blocks: logical indexing

Relational operators return a type "logical", i.e. true (1) or false (0). You can use this to select subsets of existing vectors.

Simple demonstration:

```
>> x = linspace(-3,3,7) % Create a base
     vector
x =
    -3    -2    -1     0     1     2     3
>> x>0 % Illustrate logical result
ans =
  1x7 logical array
   0   0   0   0   1   1   1
>> x2 = x(x>0) % Create a subset
x2 =
     1     2     3
```

You can use the same logical index for different vectors (in script-format):

```
% Create a base vector
x  = linspace(-4*pi,4*pi,1000);
y = (sin(x)+0.2*x)+1;
% Select only values where y>0 and plot
xs = x(y>0); ys = y(y>0);
plot(x,y,xs,ys)
```

Practice relational operations and logical indexing

1. Let `x = [1 5 2 8 9 0 1]` and `y = [5 2 2 6 0 0 2]`. Execute and explain the results of the following commands:
   - `x > y`
   - `y < x`
   - `x == y`
   - `x <= y`
   - `y >= x`
   - `x | y`
   - `x & (~y)`
   - `(x > y)| (y < x)`
   - `(x > y)& (y < x)`
2. Clear the workspace. Let `x = 1:10` and `y = [3 5 6 1 8 2 9 4 0 7]`. The exercises here show the techniques of logical-indexing. Execute and interpret the results of the following commands:
   - `(x > 3)& (x < 8)`
   - `x(x > 5)`
   - `y(x <= 4)`
   - `x((x < 2)|(x>=8))`
   - `y((x < 2)|(x>=8))`
   - `x(y<0)`

Building blocks: case selection

`switch`-statement: Selects and runs a block of code.

Building blocks: case selection

`switch`-statement: Selects and runs a block of code.

```
[dnum,dnam] = weekday(now);
switch dnum
    case {1,7}
        disp('Yay! It is weekend!');
    case 6
        disp('Hooray! It is Friday!');
    case {2,3,4,5}
        disp(['Today is ' dnam]);
    otherwise
        disp('Today is not a good day...');
end
```

Building blocks: loops

`for`-loop: Performs a block of code a certain number of times.

Building blocks: loops

`for`-loop: Performs a block of code a certain number of times.

```
>> p(1) = 1;
>> p(2) = 1;
>> for i = 2:10
p(i+1) = p(i)+p(i-1);
end
>> p
p =
    1    1    2    3    5    8   13   21   34   55   89
```

Building blocks: indeterminate repetition

`while`-loop: Performs and repeats a block of code until a certain condition.

Building blocks: indeterminate repetition

`while`-loop: Performs and repeats a block of code until a certain condition.

```
num = floor (10* rand +1) ;
guess = input ('Your guess please : ');

while ( guess ~= num )
    guess = input ('That is wrong. Try again ... ');
end

if (isempty(guess))
    disp('No number supplied - exit');
else
    disp ('Correct!');
end
```

Example algorithm

Compute the factorial of N: $N! = N \cdot (N-1) \cdot (N-2) \cdots 2 \cdot 1$

How to deal with this?

Example algorithm

Compute the factorial of N: $N! = N \cdot (N-1) \cdot (N-2) \cdots 2 \cdot 1$

How to deal with this?

Naive approach

```
Z = 1;
Z = Z*2;
Z = Z*3;
Z = Z*4;
... etc ...
```

Example algorithm

Compute the factorial of N: $N! = N \cdot (N-1) \cdot (N-2) \cdots 2 \cdot 1$

How to deal with this?

Naive approach

```
Z = 1;
Z = Z*2;
Z = Z*3;
Z = Z*4;
... etc ...
```

For-loop

```
Z = 1;
for i = 1:N
    Z = Z*i;
end
```

Example algorithm

Compute the factorial of N: $N! = N \cdot (N-1) \cdot (N-2) \cdots 2 \cdot 1$

How to deal with this?

Naive approach

```
Z = 1;
Z = Z*2;
Z = Z*3;
Z = Z*4;
... etc ...
```

For-loop

```
Z = 1;
for i = 1:N
    Z = Z*i;
end
```

While-loop

```
Z = 1;
i = 1;
while (i<=N)
    Z = Z*i;
    i = i+1;
end
```

Note: `N` must be set beforehand!
Note: Pay attention to the relational operators!

Input and output

Many programs require some input (data) to function correctly. A combination of the following is common:

- Input may be given in a parameters file ("hard-coded")

Input and output

Many programs require some input (data) to function correctly. A combination of the following is common:

- Input may be given in a parameters file ("hard-coded")
- Input may be entered via the keyboard

```
>> a = input('Please enter the number ');
```

Input and output

Many programs require some input (data) to function correctly. A combination of the following is common:

- Input may be given in a parameters file ("hard-coded")
- Input may be entered via the keyboard

```
>> a = input('Please enter the number ');
```

- Input may be read from a file, e.g.

```
>> data = getfield(importdata('myData.txt', ' ', 4), 'data');
>> numdata = xlsread('myExcelDataFile.xls');
```

Importdata also has a GUI equivalent (right-click on file, select *Import Data*)

Input and output

Many programs require some input (data) to function correctly. A combination of the following is common:

- Input may be given in a parameters file ("hard-coded")
- Input may be entered via the keyboard

```
>> a = input('Please enter the number ');
```

- Input may be read from a file, e.g.

```
>> data = getfield(importdata('myData.txt', ' ', 4), 'data');
>> numdata = xlsread('myExcelDataFile.xls');
```

Importdata also has a GUI equivalent (right-click on file, select *Import Data*)

- There are many more advanced functions, e.g. `fread`, `fgets`, ...

Input and output

Output of results to screen, storing arrays to a file or exporting a graphic are the most common ways of getting data out of Matlab:

- Results of each expression are automatically shown on screen as long as the line is not ended with a semi-colon;

Input and output

Output of results to screen, storing arrays to a file or exporting a graphic are the most common ways of getting data out of Matlab:

- Results of each expression are automatically shown on screen as long as the line is not ended with a semi-colon;
- Output may be stored via the GUI:

Input and output

Output of results to screen, storing arrays to a file or exporting a graphic are the most common ways of getting data out of Matlab:

- Results of each expression are automatically shown on screen as long as the line is not ended with a semi-colon;
- Output may be stored via the GUI:
  - Use the 'Export Setup' function

Input and output

Output of results to screen, storing arrays to a file or exporting a graphic are the most common ways of getting data out of Matlab:

- Results of each expression are automatically shown on screen as long as the line is not ended with a semi-colon;
- Output may be stored via the GUI:
  - Use the 'Export Setup' function
  - Save figure (use .fig, .eps or .png, **not** .jpg or .pcx)

Input and output

Output of results to screen, storing arrays to a file or exporting a graphic are the most common ways of getting data out of Matlab:

- Results of each expression are automatically shown on screen as long as the line is not ended with a semi-colon;
- Output may be stored via the GUI:
  - Use the 'Export Setup' function
  - Save figure (use .fig, .eps or .png, **not** .jpg or .pcx)
  - Save variables (right click, save as)

Input and output

Output of results to screen, storing arrays to a file or exporting a graphic are the most common ways of getting data out of Matlab:

- Results of each expression are automatically shown on screen as long as the line is not ended with a semi-colon;
- Output may be stored via the GUI:
  - Use the 'Export Setup' function
  - Save figure (use .fig, .eps or .png, **not** .jpg or .pcx)
  - Save variables (right click, save as)
- Save variables automatically (scripted):

```
>> savefile = 'test.mat';
>> p = rand(1,10);
>> q = ones(10);
>> save(savefile,'p','q')
```

Input and output

Output of results to screen, storing arrays to a file or exporting a graphic are the most common ways of getting data out of Matlab:

- Results of each expression are automatically shown on screen as long as the line is not ended with a semi-colon;
- Output may be stored via the GUI:
  - Use the 'Export Setup' function
  - Save figure (use .fig, .eps or .png, **not** .jpg or .pcx)
  - Save variables (right click, save as)
- Save variables automatically (scripted):

```
>> savefile = 'test.mat';
>> p = rand(1,10);
>> q = ones(10);
>> save(savefile,'p','q')
```

- More advanced functions can be found in e.g. `fwrite`, `fprintf`, ...

Today's outline

- Introduction
- Data structures
- Plotting
- Creating algorithms
- Functions
- Conclusions
- Exercises

Functions - general

A function in a programming language is a program fragment that performs a certain task. Creating functions keeps your code clean, re-usable and structured.

- You can use functions supplied by the programming language, and define functions yourself
- Functions take one or more input parameters (*arguments*), and *return* an output (result).
- In Matlab, functions are defined as follows (2 output and 3 input arguments):

- In the function body, you can perform operations on the input parameters (`in1`, `in2` and `in3`.
- Inside a function, other values are not available (local workspace). Analogously, any variables created are not available after the function returns, except for those explicitly defined as output arguments.
- The name of the function and the name of the file should be identical and (just like variables) be unique.

Functions - general

A function in a programming language is a program fragment that performs a certain task. Creating functions keeps your code clean, re-usable and structured.

- You can use functions supplied by the programming language, and define functions yourself
- Functions take one or more input parameters (*arguments*), and *return* an output (result).
- In Matlab, functions are defined as follows (2 output and 3 input arguments):

- In the function body, you can perform operations on the input parameters (`in1`, `in2` and `in3`.
- Inside a function, other values are not available (local workspace). Analogously, any variables created are not available after the function returns, except for those explicitly defined as output arguments.
- The name of the function and the name of the file should be identical and (just like variables) be unique.

Functions - general

A function in a programming language is a program fragment that performs a certain task. Creating functions keeps your code clean, re-usable and structured.

- You can use functions supplied by the programming language, and define functions yourself
- Functions take one or more input parameters (*arguments*), and *return* an output (result).
- In Matlab, functions are defined as follows (2 output and 3 input arguments):

- In the function body, you can perform operations on the input parameters (`in1`, `in2` and `in3`.
- Inside a function, other values are not available (local workspace). Analogously, any variables created are not available after the function returns, except for those explicitly defined as output arguments.
- The name of the function and the name of the file should be identical and (just like variables) be unique.

Functions - general

A function in a programming language is a program fragment that performs a certain task. Creating functions keeps your code clean, re-usable and structured.

- You can use functions supplied by the programming language, and define functions yourself
- Functions take one or more input parameters (*arguments*), and *return* an output (result).
- In Matlab, functions are defined as follows (2 output and 3 input arguments):

```
function [out1, out2] = myFunction(in1, in2, in3)
```

- In the function body, you can perform operations on the input parameters (`in1`, `in2` and `in3`.
- Inside a function, other values are not available (local workspace). Analogously, any variables created are not available after the function returns, except for those explicitly defined as output arguments.
- The name of the function and the name of the file should be identical and (just like variables) be unique.

Functions - general

A function in a programming language is a program fragment that performs a certain task. Creating functions keeps your code clean, re-usable and structured.

- You can use functions supplied by the programming language, and define functions yourself
- Functions take one or more input parameters (*arguments*), and *return* an output (result).
- In Matlab, functions are defined as follows (2 output and 3 input arguments):

```
function [out1, out2] = myFunction(in1, in2, in3)
```

- In the function body, you can perform operations on the input parameters (`in1`, `in2` and `in3`.
- Inside a function, other values are not available (local workspace). Analogously, any variables created are not available after the function returns, except for those explicitly defined as output arguments.
- The name of the function and the name of the file should be identical and (just like variables) be unique.

Functions - general

A function in a programming language is a program fragment that performs a certain task. Creating functions keeps your code clean, re-usable and structured.

- You can use functions supplied by the programming language, and define functions yourself
- Functions take one or more input parameters (*arguments*), and *return* an output (result).
- In Matlab, functions are defined as follows (2 output and 3 input arguments):

```
function [out1, out2] = myFunction(in1, in2, in3)
```

- In the function body, you can perform operations on the input parameters (`in1`, `in2` and `in3`.
- Inside a function, other values are not available (local workspace). Analogously, any variables created are not available after the function returns, except for those explicitly defined as output arguments.
- The name of the function and the name of the file should be identical and (just like variables) be unique.

Functions - general

A function in a programming language is a program fragment that performs a certain task. Creating functions keeps your code clean, re-usable and structured.

- You can use functions supplied by the programming language, and define functions yourself
- Functions take one or more input parameters (*arguments*), and *return* an output (result).
- In Matlab, functions are defined as follows (2 output and 3 input arguments):

```
function [out1, out2] = myFunction(in1, in2, in3)
```

- In the function body, you can perform operations on the input parameters (`in1`, `in2` and `in3`.
- Inside a function, other values are not available (local workspace). Analogously, any variables created are not available after the function returns, except for those explicitly defined as output arguments.
- The name of the function and the name of the file should be identical and (just like variables) be unique.

Functions - Exercise

Example: write a function that takes 3 variables, and returns the average:

Functions - Exercise

Example: write a function that takes 3 variables, and returns the average:

Approach 1

```
function res = avg1(a,b,c)
    mySum = a + b + c;
    res = mySum / 3;
end
```

Functions - Exercise

Example: write a function that takes 3 variables, and returns the average:

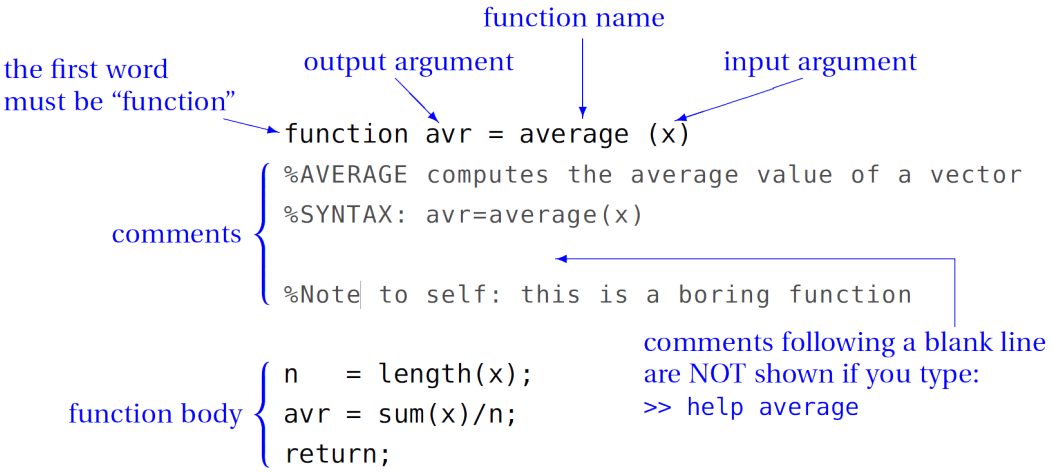
Approach 1

```
function res = avg1(a,b,c)
    mySum = a + b + c;
    res = mySum / 3;
end
```

Approach 2

```
function res = avg2(a,b,c)
    data = [a; b; c];
    res = mean(data);
end
```

Functions - definition overview



2

2 Source: Pekalska, Meinsma and Zwart, Programming with Matlab manual, University of Twente, 2018.

Functions - stack

Calling functions from each other creates a 'stack', like a stack of cards. The stack works with first in, last out (FILO) principle.

```
function [] = a()
disp('Start of a()');
b();
disp('End of a()');
end

function [] = b()
disp('Start of b()');
c();
disp('End of b()');
end

function [] = c()
disp('Start of c()');
disp('End of c()');
end
```

```
>> stackcheck
Start of a()
Start of b()
Start of c()
End of c()
End of b()
End of a()
```

Exercise: create a function

Compute $N! = N \cdot (N-1) \cdot (N-2) \cdots 2 \cdot 1$

Create a function of our while-loop approach with N the argument:

Original script

```
Z = 1;
i = 1;
while (i<=N)
    Z = Z*i;
    i = i+1;
end
```

Exercise: create a function

Compute $N! = N \cdot (N-1) \cdot (N-2) \cdots 2 \cdot 1$

Create a function of our while-loop approach with N the argument:

Original script

```
Z = 1;
i = 1;
while (i<=N)
    Z = Z*i;
    i = i+1;
end
```

Function

```
function Z = fact_while(N)

Z = 1;
i = 1;
while (i<=N)
    Z = Z*i;
    i = i+1;
end

end
```

Functions - checking input

The function we created computes the factorial correctly!

Functions - checking input

The function we created computes the factorial correctly!

- When the supplied argument is positive

Functions - checking input

The function we created computes the factorial correctly!

- When the supplied argument is positive and

Functions - checking input

The function we created computes the factorial correctly!

- When the supplied argument is positive and
- When the supplied argument is a natural number...

Functions - checking input

The function we created computes the factorial correctly!

- When the supplied argument is positive and
- When the supplied argument is a natural number...

Functions - checking input

The function we created computes the factorial correctly!

- When the supplied argument is positive and
- When the supplied argument is a natural number...

- In this case, we should check the user input to prevent an infinite loop:

Functions - checking input

The function we created computes the factorial correctly!

- When the supplied argument is positive and
- When the supplied argument is a natural number...

- In this case, we should check the user input to prevent an infinite loop:

Functions - checking input

The function we created computes the factorial correctly!

- When the supplied argument is positive and
- When the supplied argument is a natural number...

- In this case, we should check the user input to prevent an infinite loop:

```
if (fix(N)~=N) | (N<0)
    disp 'Provide a positive integer number!'
    return;
end
```

- If no check can be done before a while-loop, you may want to stop after x loops

Functions - checking input

The whole factorial function, including comments:

```
function Z = fact_while(N)
%% This function computes a factorial of input value N
% Usage  : fact_while(N)
% N      : value of which the factorial is computed
% returns: factorial of N

% Catch non-integer case
if (fix(N)~=N) | (N<0)
    disp 'Provide a positive integer number!'
    return;
end

Z = 1;
i = 1;
while (i<=N)
    Z = Z*i;
    i = i+1;
end

end
```

Recursion

- In order to understand recursion, one must first understand recursion

Recursion

- In order to understand recursion, one must first understand recursion
- A recursive function includes a call to itself (a function within a function)

Recursion

- In order to understand recursion, one must first understand recursion
- A recursive function includes a call to itself (a function within a function)
  - This could lead to infinite calls;
  - A base case is required so that recursion is stopped;
  - Base case does not call itself, simply returns.

Recursion: example

```
function out = mystery(a,b)
if (b == 1)
    % Base case
    out = a;
else
    % Recursive function call
    out = a + mystery(a,b-1);
end
```

Recursion: example

```
function out = mystery(a,b)
if (b == 1)
    % Base case
    out = a;
else
    % Recursive function call
    out = a + mystery(a,b-1);
end
```

- What does this function do?

Recursion: example

```
function out = mystery(a,b)
if (b == 1)
    % Base case
    out = a;
else
    % Recursive function call
    out = a + mystery(a,b-1);
end
```

- What does this function do?
- Can you spot the error?

Recursion: example

```
function out = mystery(a,b)
if (b == 1)
    % Base case
    out = a;
else
    % Recursive function call
    out = a + mystery(a,b-1);
end
```

- What does this function do?
- Can you spot the error?
- How deep can you go? Which values of b don't work anymore?

Recursion: exercise

Create a function computing the factorial of N, based on recursion.

Recursion: exercise

Create a function computing the factorial of N, based on recursion.

```
function res = fact_recursive(x)

% Catch non-integer case
if (fix(x)~=x) | (x<0)
    disp 'You should provide a positive integer number only'
    return;
end

if (x > 1)
    res = x*fact_recursive(x-1);
else
    res = 1;
end

end
```

Today's outline

- Introduction
- Data structures
- Plotting
- Creating algorithms
- Functions
- Conclusions
- Exercises

In conclusion...

- Matlab: A versatile development environment, with excellent vector and matrix computations
- Programming basics: variables, operators and functions, locality of variables, recursive operations

In conclusion...

- Matlab: A versatile development environment, with excellent vector and matrix computations
- Programming basics: variables, operators and functions, locality of variables, recursive operations
- For now: exercises on slide deck and Matlab modules
- Preparation for next lecture: familiarize with the concepts, use Canvas course or Matlab Academy.

In conclusion...

- Matlab: A versatile development environment, with excellent vector and matrix computations
- Programming basics: variables, operators and functions, locality of variables, recursive operations
- For now: exercises on slide deck and Matlab modules
- Preparation for next lecture: familiarize with the concepts, use Canvas course or Matlab Academy.

In conclusion...

- Matlab: A versatile development environment, with excellent vector and matrix computations
- Programming basics: variables, operators and functions, locality of variables, recursive operations
- For now: exercises on slide deck and Matlab modules
- Preparation for next lecture: familiarize with the concepts, use Canvas course or Matlab Academy.

Practice vectors and matrices

1. Create a vector `x` with the elements:
   - $[2,4,6,8,\ldots,16]$
   - $[0,\frac{1}{2},\frac{2}{3},\frac{3}{4},\ldots,\frac{99}{100}]$
2. Create a vector `x` with the elements: $x_n = \frac{(-1)^n}{2n-1}$ for $n = 1,2,3,\ldots,200$. Find the sum of the first 50 elements $x_1,\ldots,x_{50}$.
3. Let `x = 20:10:200`. Create a vector `y` of the same length as `x` such that:
   - $y_i = x_i - 3$
   - $y_i = x_i$ for every even index i and $y_i = x_i + 11$ for every odd index i.
4. Let `T = [3 4; 1 8; -4 3]` and `A = [diag(-1:2:3),T; -4 4 1 2 1]`. Perform the following operations on A:
   - Retrieve a vector consisting of the 2nd and 4th elements of the 3rd row.
   - Find the minimum of the 3rd column.
   - Find the maximum of the 2nd row.
   - Compute the sum of the 2nd column
   - Compute the mean of the row 1 and the mean of row 4

Practice plotting

1. Plot the functions $f(x) = x$, $g(x) = x^3$, $h(x) = e^x$ and $z(x) = e^{x^2}$ over the interval $[0, 4]$ on the normal scale and on the log-log scale. Use an appropriate sampling to get smooth curves. Describe your plots by using the functions: `xlabel`, `ylabel`, `title` and `legend`.

2. Make a plot of the functions: $f(x) = sin(1/x)$ and $g(x) = cos(1/x)$ over the interval $[0.01, 0.1]$. How do you create `x` so that the plots look sufficiently smooth?

Practice control flow and loops (1)

1. Write a function that uses two logical input arguments with the following behaviour:

$$f(\text{true}, \text{true}) \mapsto \text{false}$$
$$f(\text{false}, \text{true}) \mapsto \text{true}$$
$$f(\text{true}, \text{false}) \mapsto \text{true}$$
$$f(\text{false}, \text{false}) \mapsto \text{false}$$

2. Write a function that computes the factorial of x:

$$f(x) = x! = 1 \times 2 \times 3 \times 4 \times \ldots \times x$$

- Using a loop-construction
- Using recursion

Practice control flow and loops (2)

1. Write a function that computes the exponential function using the Taylor series

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

   until the last term is smaller than 10^{-6}.

2. Use a script to compute the result of the following series:

$$f_n = \sum_{n=1}^{\infty} \frac{1}{\pi^2 n^2}$$

   This should give you an indication of the fraction this series converges to.

   - Now plot in two vertically aligned subplots i) The result as a function of n, and ii) the difference with the earlier mentioned fraction as a function of n. For the latter, consider carefully the axis scale!

Practice logical indexing

1. Let `x = linspace(-4,4,1000)`, $y_1 = 3x^2 - 4x - 6$ and $y_2 = 1.5x - 1$. Use logical indexing to determine function $y_3 = \max(\max(y_1, y_2), 0)$. Plot the function.
2. Consider these data concerning the age (in years), length (in cm) and weight (in kg) of twelve adult men: `A = [41 25 33 29 64 34 47 38 49 32 26 26]; H = [165 186 177 190 156 174 164 205 184 190 165 171]; W = [75 90 97 60 74 65 101 85 91 75 87 70];`.

   - Calculate the average of all vectors (age, weight and length).
   - Combine the command `length` with logical indexing to determine how many men in the group are taller than 182 cm.
   - What is the average age of men with a body-mass index ($B \equiv \frac{W}{L^2}$ with W in kg and L in m) larger than 25? And for men with a $B < 25$?
   - How many men are older than the average and at the same time have a BMI below 25?

Practice algorithm: Fourier series for heat equation

The unsteady 1D heat equation in 1D in a slab of material is given as:

$$\frac{\partial T}{\partial t} = k \frac{\partial^2 T}{\partial x^2}$$

We can express the temperature profile $T(x,t)$ in the slab using a Fourier sine series. For an initial profile T(x,0) = 20 and fixed boundary values T(0,t) = T(L,t) = 0, the solution is given as:

$$T(x,t) = \sum_{n=1}^{n=\infty} \frac{40(1-(-1)^n)}{n\pi} \sin\left(\frac{n\pi x}{L}\right) \exp\left(-kt\frac{n\pi}{L}^2\right)$$

- Create a script to solve this equation using loops and/or conditional statements

Matlab and Programming 2

Programming workflow and advanced features

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2020-2021

Today's outline

- Coding style
- Debugging and profiling
- Visualisation
- Functions: revisited
- Concluding remarks

Today's outline

- Coding style
- Debugging and profiling
- Visualisation
- Functions: revisited
- Concluding remarks

Make a habit of the following adage

MAKE IT **WORK**

MAKE IT **RIGHT**

MAKE IT **FAST**

Make it work

Use the building blocks of previous lecture to create an algorithm:

1. *Problem analysis*
   Contextual understanding of the nature of the problem to be solved
2. *Problem statement*
   Develop a detailed statement of the mathematical problem to be solved with the program
3. *Processing scheme*
   Define the inputs and outputs of the program
4. *Algorithm*
   A step-by-step procedure of all actions to be taken by the program (*pseudo-code*)
5. *Program the algorithm*
   Convert the algorithm into a computer language, and debug until it runs
6. *Evaluation*
   Test all of the options and conduct a validation study

Now it's time to make it right!

Make it work

Use the building blocks of previous lecture to create an algorithm:

1. *Problem analysis*
   Contextual understanding of the nature of the problem to be solved
2. *Problem statement*
   Develop a detailed statement of the mathematical problem to be solved with the program
3. *Processing scheme*
   Define the inputs and outputs of the program
4. *Algorithm*
   A step-by-step procedure of all actions to be taken by the program (*pseudo-code*)
5. *Program the algorithm*
   Convert the algorithm into a computer language, and debug until it runs
6. *Evaluation*
   Test all of the options and conduct a validation study

Now it's time to make it right!

Make it work

Use the building blocks of previous lecture to create an algorithm:

1. *Problem analysis*
   Contextual understanding of the nature of the problem to be solved
2. *Problem statement*
   Develop a detailed statement of the mathematical problem to be solved with the program
3. *Processing scheme*
   Define the inputs and outputs of the program
4. *Algorithm*
   A step-by-step procedure of all actions to be taken by the program (*pseudo-code*)
5. *Program the algorithm*
   Convert the algorithm into a computer language, and debug until it runs
6. *Evaluation*
   Test all of the options and conduct a validation study

Now it's time to make it right!

Make it work

Use the building blocks of previous lecture to create an algorithm:

1. *Problem analysis*
   Contextual understanding of the nature of the problem to be solved
2. *Problem statement*
   Develop a detailed statement of the mathematical problem to be solved with the program
3. *Processing scheme*
   Define the inputs and outputs of the program
4. *Algorithm*
   A step-by-step procedure of all actions to be taken by the program (*pseudo-code*)
5. *Program the algorithm*
   Convert the algorithm into a computer language, and debug until it runs
6. *Evaluation*
   Test all of the options and conduct a validation study

Now it's time to make it right!

Make it work

Use the building blocks of previous lecture to create an algorithm:

1. *Problem analysis*
   Contextual understanding of the nature of the problem to be solved
2. *Problem statement*
   Develop a detailed statement of the mathematical problem to be solved with the program
3. *Processing scheme*
   Define the inputs and outputs of the program
4. *Algorithm*
   A step-by-step procedure of all actions to be taken by the program (*pseudo-code*)
5. *Program the algorithm*
   Convert the algorithm into a computer language, and debug until it runs
6. *Evaluation*
   Test all of the options and conduct a validation study

Now it's time to make it right!

Make it work

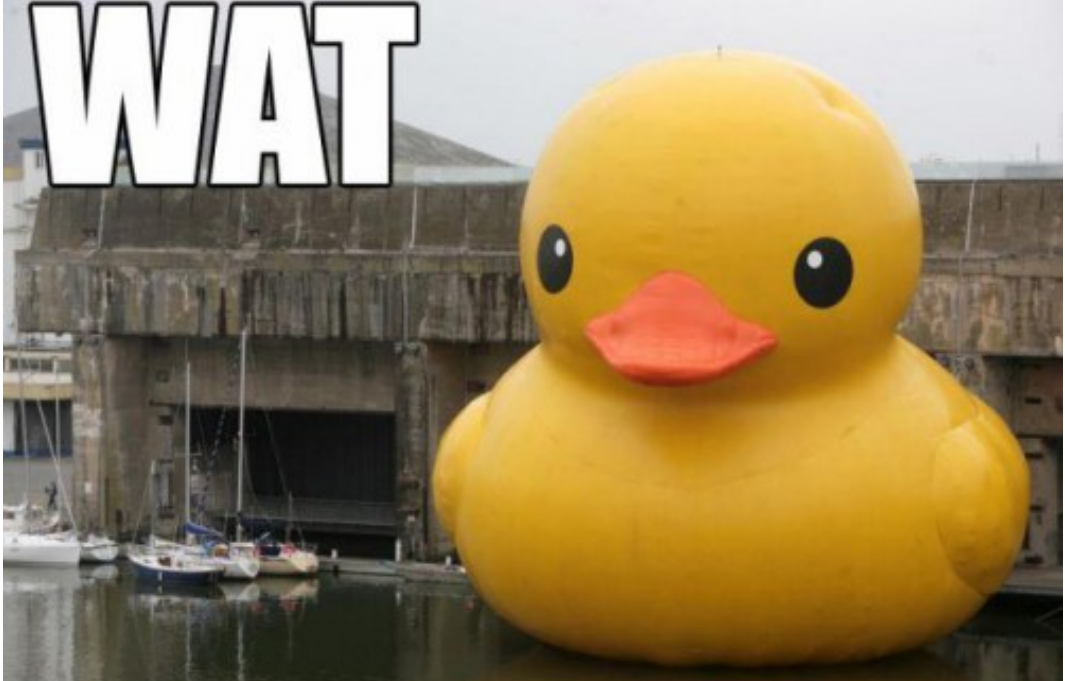
Use the building blocks of previous lecture to create an algorithm:

1. *Problem analysis*
   Contextual understanding of the nature of the problem to be solved
2. *Problem statement*
   Develop a detailed statement of the mathematical problem to be solved with the program
3. *Processing scheme*
   Define the inputs and outputs of the program
4. *Algorithm*
   A step-by-step procedure of all actions to be taken by the program (*pseudo-code*)
5. *Program the algorithm*
   Convert the algorithm into a computer language, and debug until it runs
6. *Evaluation*
   Test all of the options and conduct a validation study

Now it's time to make it right!

Interpret the following code

```
s=checksc();
if(s==true)
a=cb();
b=cfrsp();
if(a<5)
if(b>5)
a=gtbs();
end
if(a>b)
ubx();
end
end
else
brn();
gtbs();
end
```


WAT

Let's change that a bit... Indentation

Let's change that a bit... Indentation

Shown here with 2 spaces of indentation, Matlab uses 4 by default!

```
s=checksc();
if(s==true)
a=cb();
b=cfrsp();
if(a<5)
if(b>5)
a=gtbs();
end
if(a>b)
ubx();
end
end
else
brn();
gtbs();
end
```

```
s = checksc();
if (s == true)
  a = cb();
  b = cfrsp();
  if (a < 5)
    if (b > 5)
      a = gtbs();
    end
    if (a > b)
      ubx();
    end
  end
else
  brn();
  gtbs();
end
```

Readable variables and function names

```
s = checksc();
if (s == true)
  a = cb();
  b = cfrsp();
  if (a < 5)
    if (b > 5)
      a = gtbs();
    end
    if (a > b)
      ubx();
    end
  end
else
    brn();
    gtbs();
end
```

```
IAmFree = checkSchedule();
if (IAmFree == true)
  books = countBooks();
  shelfSize = countFreeSpaceShelf();
  if (books < 5)
    if (shelfSize > 5)
      books = goToBookStore();
    end
    if (books > shelfSize)
      useBox();
    end
  end
else
  burnBooks();
  goToBookStore();
end
```

Get rid of magic numbers in the code

```
IAmFree = checkSchedule();
if (IAmFree == true)
  books = countBooks();
  shelfSize = countFreeSpaceShelf();
  if (books < 5)
    if (shelfSize > 5)
      books = goToBookStore();
    end
    if (books > shelfSize)
      useBox();
    end
  end
else
  burnBooks();
  goToBookStore();
end
```

```
maxShelfSize = 5;
minBooksNeeded = 5;

IAmFree = checkSchedule();
if (IAmFree == true)
  books = countBooks();
  shelfSize = countFreeSpaceShelf();
  if (books < maxShelfSize)
    if (shelfSize > minBooksNeeded)
      books = goToBookStore();
    end
    if (books > shelfSize)
      useBox();
    end
  end
else
  burnBooks();
  goToBookStore();
end
```

That's more like it!

```
s=checksc();
if(s==true)
a=cb();
b=cfrsp();
if(a<5)
if(b>5)
a=gtbs();
end
if(a>b)
ubx();
end
end
else
brn();
gtbs();
end
```

```
maxShelfSize = 5;
minBooksNeeded = 5;

IAmFree = checkSchedule();
if (IAmFree == true)
  books = countBooks();
  shelfSize = countFreeSpaceShelf();
  if (books < maxShelfSize)
    if (shelfSize > minBooksNeeded)
      books = goToBookStore();
    end
    if (books > shelfSize)
      useBox();
    end
  end
else
  burnBooks();
  goToBookStore();
end
```

Writing readable code

Good code reads like a book.

Writing readable code

Good code reads like a book.

- When it doesn't, make sure to use comments. In Matlab, everything following `% is a comment`
- Prevent "smart constructions" in the code
- Re-use working code (i.e. create functions for well-defined tasks).
- Documentation is also useful, but hard to maintain.
- Matlab comes with a function that generates reports from comments

How not to comment

- Useless:

```
% Start program
```

How not to comment

- Useless:

```
% Start program
```

- Obvious:

```
if (a > 5)   % Check if a is greater than 5
    ...
end
```

How not to comment

- Useless:

```
% Start program
```

- Obvious:

```
if (a > 5)    % Check if a is greater than 5
    ...
end
```

- Too much about the life:

```
% Well... I do not know how to explain what is going on
% in the snippet below. I tried to code in the night
% with some booze and it worked then, but now I have a
% strong hangover and some parameters still need to be
% worked out...
```

How not to comment

- Useless:

```
% Start program
```

- Obvious:

```
if (a > 5)   % Check if a is greater than 5
    ...
end
```

- Too much about the life:

```
% Well... I do not know how to explain what is going on
% in the snippet below. I tried to code in the night
% with some booze and it worked then, but now I have a
% strong hangover and some parameters still need to be
% worked out...
```

- ...

```
% You may think that this function is obsolete, and doesn't seem to
% do anything. And you would be correct. But when we remove this
% function for some reason the whole program crashes and we can't
% figure out why, so here it will stay.
```

Adding comments to our program

Use comments to document design and purpose (functionality), not mechanics (implementation).

```
IAmFree = checkSchedule();
if (IAmFree == true)
% Count books and amount of free space on a shelf.
% If minimum number of books I need is less than a
% shelf capacity, go shopping and buy additional
% literature. If the amount of books after the
% shopping is too big, use boxes to store them.
  books = countBooks();
  shelfSize = countFreeSpaceShelf();

  ...

else
  burnBooks();
  goToBookStore();
end
```

What else makes a good program?

- Portability (guaranteed in Matlab)
- Readability
- Efficiency
- Structural
- Flexibility
- Generality
- Documentation

Funny thing is: This list does not mention that the program should be actually working for its intended purposes!

Portability

It should work on your neighbors laptop. Well, that is guaranteed anyway. There is a little caveat though, what if your neighbor is stupid and has a polluted workspace?

```
% trust no one! nuke all evidence! no witnesses!
clear all   % destroy all variables
close all   % close all figures
```

Solution: clear all variables before your program starts

Readability

Don't use meaningless variable or function names. Rule of thumb: use verbs for functions and nouns for variables.

```
% stupid names
x = 5;
xx = myfunction(x);

% proper names
number_dams = 6;
beaver_workforce = allocate_beavers();
dams = build_dams(beaver_workforce, number_dams);
```

Efficiency

This one is difficult. Not much you can do without truly understanding how Matlab is utilizing your processor and memory. A couple of guidelines though:

- Avoid loops
- Especially avoid nested loops
- Use inherent matrix operations when possible
- Reduce IO (i.e. reading / writing to and from files)
- Don't run scripts from network disks
- Pre-allocate your matrices, that means, making it as large as the maximum required size for your particular problem
- Use tic/toc to test the execution times

```
x = linspace(0,10,1000001);
tic;
for cr = 1:length(x)
    y(cr) = sin(2*pi*x(cr));
end;
toc
```

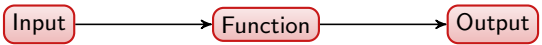
```
x = linspace(0,10,1000001);
tic;

y = sin(2*pi*x);

toc
```

Structural

- Compartmentalize your code.
- Write functions whenever possible.
  - If you have > 15 lines of code, you can probably replace it by one or more functions.
  - In principle, it should not even matter how function works, as long as it gives the expected output.



- Put critical variables at the beginning of your program.

Structural

Write code as if it are paragraphs of a story.

```
% Step 0: Define variables
n_steps = 10000;     % number of steps
n_walks = 1000;      % number of random walk samples

% Step 1: Generate n_walks random_walks
de = zeros(n_walks, 2);
for i=1:n_walks
    angles = get_random_angles(n_steps);
    coord = transform_angles_to_coordinates(angles);
    de(i,1) = calculate_de(coord);  % store de
    de(i,2) = de(i,1)^2;            % store de^2
end

% Step 2: Plot the histogram
histogram(de(:,1),'Normalization', 'pdf')
[D,P] = calculate_pdf(n_steps, 1000);
hold on;
plot(D,P);
```

Flexibility

If you want to add a feature or change something inside the program, it should not require rewriting the whole program. (jargon: non-linear propagation of change).
Solution: Encapsulate your code and "Don't Repeat Yourself"

- Use functions for specific tasks (can you verbalize it? Then it is probably a function)
- Use variables, even for constants (it sounds like a oxymoron, but it's not! In fact, constant variables are a real thing)
- Use abstraction whenever possible

Generalization

If you code is working for one problem, it should also work for a similar problem, in another company, on another planet.

Pro-tip: Separate data from algorithms.
To be honest: I rarely see students making this mistake.

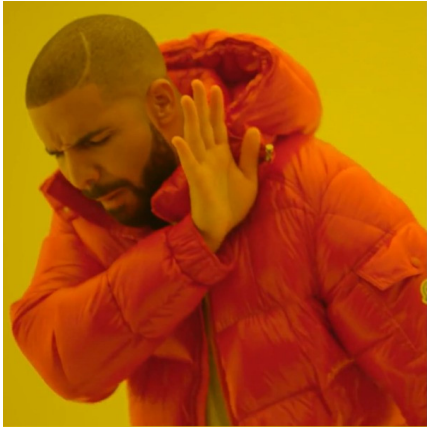
```
% stupid code
A = [1 2 3; 4 5 6; 7 8 9]        % hardcode data in the program

% smart code
load('some_random_dataset.dat') % or the arguably better fopen functions
```

Documentation

Properly document your code. Write your comments in a clear and concise fashion. Help your future self: Write clear and concise documentation.

```
function c = f(a,b)
switch nargin
    case 2
        c = a + b;
    case 1
        c = a + a;
    otherwise
        c = 0;
    end
```



Documentation

Properly document your code. Write your comments in a clear and concise fashion. Help your future self: Write clear and concise documentation.

```
function c = f(a,b)
 % ADDME  Add two values together.
 %   C = ADDME(A) adds A to itself.
 %   C = ADDME(A,B) adds A and B together.
 %
 %   See also SUM, PLUS.
switch nargin  % number of arguments
    case 2     % sum two different numbers
        c = a + b;
    case 1     % double single number
        c = a + a;
    otherwise  % shit in, shit out
        c = 0;
    end
```

Make a habit of the following adage

MAKE IT **WORK**

MAKE IT **RIGHT**

MAKE IT **FAST**

Make a habit of the following adage

1. *Make it work*
   Create an algorithm that does the intended job. Make sure it works, and works repeatedly. Test and verify frequently. Add *todo* comments when you're not sure about a certain decision.
2. *Make it right*
   Refactor the code to improve the code design. Insert functions, comments, compartmentalize it. Get rid of magic numbers, use sensible variable names. Check input. Test and verify. Align with the team!
3. *Make it fast*
   Measure and tune the performance of your code (profiling tool). In Matlab, vectorized calculations are much (!) faster than for-loops. Use sensible numerical techniques (e.g. higher-order integration).

Program by iterating over these aspects multiple times, starting at the fine-grained level, working your way up.

Make a habit of the following adage

1. *Make it work*
   Create an algorithm that does the intended job. Make sure it works, and works repeatedly. Test and verify frequently. Add *todo* comments when you're not sure about a certain decision.
2. *Make it right*
   Refactor the code to improve the code design. Insert functions, comments, compartmentalize it. Get rid of magic numbers, use sensible variable names. Check input. Test and verify. Align with the team!
3. *Make it fast*
   Measure and tune the performance of your code (profiling tool). In Matlab, vectorized calculations are much (!) faster than for-loops. Use sensible numerical techniques (e.g. higher-order integration).

Program by iterating over these aspects multiple times, starting at the fine-grained level, working your way up.

Make a habit of the following adage

1. *Make it work*
   Create an algorithm that does the intended job. Make sure it works, and works repeatedly. Test and verify frequently. Add *todo* comments when you're not sure about a certain decision.
2. *Make it right*
   Refactor the code to improve the code design. Insert functions, comments, compartmentalize it. Get rid of magic numbers, use sensible variable names. Check input. Test and verify. Align with the team!
3. *Make it fast*
   Measure and tune the performance of your code (profiling tool). In Matlab, vectorized calculations are much (!) faster than for-loops. Use sensible numerical techniques (e.g. higher-order integration).

Program by iterating over these aspects multiple times, starting at the fine-grained level, working your way up.

Today's outline

- Coding style
- Debugging and profiling
- Visualisation
- Functions: revisited
- Concluding remarks

Errors in computer programs

The following symptoms can be distinguished:

- Unable to execute the program
- Program crashes, warnings or error messages
- Never-ending loops
- Wrong (unexpected) result

Errors in computer programs

The following symptoms can be distinguished:

- Unable to execute the program
- Program crashes, warnings or error messages
- Never-ending loops
- Wrong (unexpected) result

Three error categories:

Syntax errors You did not obey the language rules. These errors prevent running or compilation of the program.

Runtime errors Something goes wrong during the execution of the program resulting in an error message (problem with input, division by zero, loading of non-existent files, memory problems, etc.)

Semantic errors The program does not do what you expect, but does what have told it to do.

Errors in computer programs

The following symptoms can be distinguished:

- Unable to execute the program
- Program crashes, warnings or error messages
- Never-ending loops
- Wrong (unexpected) result

Three error categories:

Syntax errors You did not obey the language rules. These errors prevent running or compilation of the program.

Runtime errors Something goes wrong during the execution of the program resulting in an error message (problem with input, division by zero, loading of non-existent files, memory problems, etc.)

Semantic errors The program does not do what you expect, but does what have told it to do.

Errors in computer programs

The following symptoms can be distinguished:

- Unable to execute the program
- Program crashes, warnings or error messages
- Never-ending loops
- Wrong (unexpected) result

Three error categories:

Syntax errors	You did not obey the language rules. These errors prevent running or compilation of the program.
Runtime errors	Something goes wrong during the execution of the program resulting in an error message (problem with input, division by zero, loading of non-existent files, memory problems, etc.)
Semantic errors	The program does not do what you expect, but does what have told it to do.

Validation

- Testcases: run the program with parameters such that a known result is (should be) produced.
- Testcases: what happens when unforeseen input is encountered?
  - More or fewer arguments than anticipated? (Matlab uses `varargin` and `nargin` to create a varying number of input arguments, and to check the number of given input arguments
  - Other data types than anticipated? How does the program handle this? Warnings, error messages (crash), NaN or worse: a program that silently continues?
- For physical modeling, we typically look for analytical solutions
  - Sometimes somewhat stylized cases
  - Possible solutions include Fourier-series
  - Experimental data

Validation

- Testcases: run the program with parameters such that a known result is (should be) produced.
- Testcases: what happens when unforeseen input is encountered?
  - More or fewer arguments than anticipated? (Matlab uses `varargin` and `nargin` to create a varying number of input arguments, and to check the number of given input arguments
  - Other data types than anticipated? How does the program handle this? Warnings, error messages (crash), NaN or worse: a program that silently continues?
- For physical modeling, we typically look for analytical solutions
  - Sometimes somewhat stylized cases
  - Possible solutions include Fourier-series
  - Experimental data

But: validation can only tell you *if* something is wrong, not *where* it went wrong.

The debugger (1)

- No-one can write a 1000-line code without making errors
  - If you can, please come work for us
- One of the most important skills you will acquire is debugging.
- Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.
- In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.
- Actually, you are the detective, the murderer and the victim at the same time.

The debugger (1)

- No-one can write a 1000-line code without making errors
  - If you can, please come work for us
- One of the most important skills you will acquire is debugging.
- Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.
- In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.
- Actually, you are the detective, the murderer and the victim at the same time.

The debugger (1)

- No-one can write a 1000-line code without making errors
  - If you can, please come work for us
- One of the most important skills you will acquire is debugging.
- Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.
- In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.
- Actually, you are the detective, the murderer and the victim at the same time.

The debugger (1)

- No-one can write a 1000-line code without making errors
  - If you can, please come work for us
- One of the most important skills you will acquire is debugging.
- Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.
- In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.
- Actually, you are the detective, the murderer and the victim at the same time.

The debugger (1)

- No-one can write a 1000-line code without making errors
  - If you can, please come work for us
- One of the most important skills you will acquire is debugging.
- Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.
- In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.
- Actually, you are the detective, the murderer and the victim at the same time.

"When you have eliminated the impossible, whatever remains, however improbable, must be the truth."

— A. Conan Doyle, The Sign of Four

The debugger (2)

The debugger can help you to:

- Pause a program at a certain line: set a *breakpoint*
- Check the values of variables during the program
- Controlled execution of the program:
  - One line at a time
  - Run until a certain line
  - Run until a certain condition is met (conditional breakpoint)
  - Run until the current function exits
- Note: You may end up in the source code of Matlab functions!
- Check Canvas (Matlab Crash Course section) for a demonstration of the debugger.

Recursive Fibonacci

- Create a program that computes the n-th Fibonacci number using recursion: $F_n = F_{n-1} + F_{n-2}$ with $F_1 = 1$ and $F_2 = 1$

Recursive Fibonacci

- Create a program that computes the n-th Fibonacci number using recursion: $F_n = F_{n-1} + F_{n-2}$ with $F_1 = 1$ and $F_2 = 1$

```
1  function out = fibonacci_recursive(N)
2  %FIBONACCI_RECURSIVE Prints out the Nth Fibonacci number to the screen
3  %SYNTAX: fibonacci_recursive(N)
4
5  if (N>2)
6      Nminus1 = fibonacci_recursive(N-1);
7      Nminus2 = fibonacci_recursive(N-2);
8      out = Nminus1 + Nminus2;
9  elseif (N==1) || (N==2)
10     out = 1;
11 else
12     error('Input argument was invalid')
13 end
```

Recursive Fibonacci

- Create a program that computes the n-th Fibonacci number using recursion: $F_n = F_{n-1} + F_{n-2}$ with $F_1 = 1$ and $F_2 = 1$

```
1  function out = fibonacci_recursive(N)
2  %FIBONACCI_RECURSIVE Prints out the Nth Fibonacci number to the screen
3  %SYNTAX: fibonacci_recursive(N)
4
5  if (N>2)
6      Nminus1 = fibonacci_recursive(N-1);
7      Nminus2 = fibonacci_recursive(N-2);
8      out = Nminus1 + Nminus2;
9  elseif (N==1) || (N==2)
10     out = 1;
11 else
12     error('Input argument was invalid')
13 end
```

- Place a breakpoint line 5 (click on dash or press F12), run `fibonacci_recursive(5)`
- Explore the function of step F10, step into F11, and how the local workspace changes
- Stop the debugger (red stop button on top, or Shift + F5)
- Right-click the breakpoint, select *Set/modify condition*, enter `N==2`, run again.

Profiling the code

We can use the commands `tic` and `toc` to record the time spent in the enclosed code:

```
% Script to call fibonacci recursive for different values of N and
% display the time to process

max_N = 32;

for i = 1:max_N
    tic
    i
    fibonacci_recursive(i);
    toc;
end
```

Profiling the code

We can use the commands `tic` and `toc` to record the time spent in the enclosed code:

```
% Script to call fibonacci recursive for different values and record the
% time to process

max_N = 32;

for i = 1:max_N
    tic
    i
    fibonacci_recursive(i);
    recorded_time(i) = toc;
end

plot(1:max_N,recorded_time)
```

Profiling the code

We can use the commands `tic` and `toc` to record the time spent in the enclosed code:

```
% Script to call fibonacci recursive for different values and record the
% time to process

max_N = 32;

for i = 1:max_N
    tic
    i
    fibonacci_recursive(i);
    recorded_time(i) = toc;
end

plot(1:max_N,recorded_time)
```

- The time needed for a computation increases exponentially. Indeed, this may not be the fastest way to compute the Fibonacci sequence.
- In order to see where the computation time is spent, use the *Run and Time* feature in Matlab (home tab).

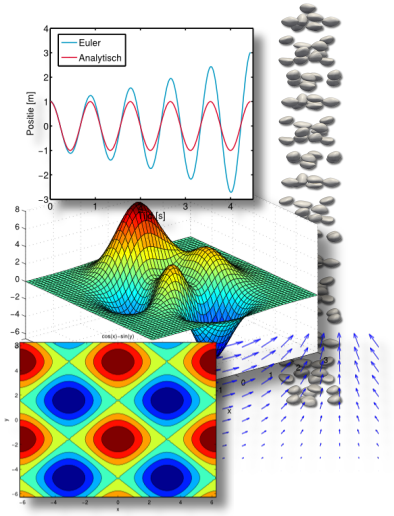
Today's outline

- Coding style
- Debugging and profiling
- Visualisation
- Functions: revisited
- Concluding remarks

Data visualisation

Modeling can lead to very large data sets, that require appropriate visualisation to convey your results.

- 1D, 2D, 3D visualisation
- Multiple variables at the same time (temperature, concentration, direction of flow)
- Use of colors, contour lines
- Use of stream lines or vector plots
- Animations



Plotting

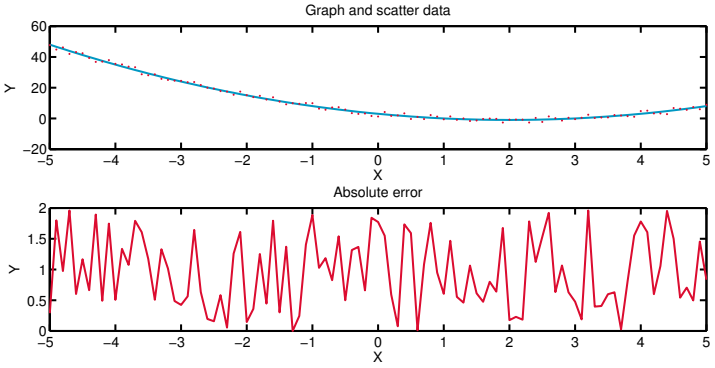
```
x = -5:0.1:5;
y = x.^2-4*x+3;
y2 = y + (2-4*rand(size(y)));
```

Plotting

```
x = -5:0.1:5;
y = x.^2-4*x+3;
y2 = y + (2-4*rand(size(y)));
subplot(2,1,1); plot(x,y,'-',x,y2,'r.');
xlabel('X'); ylabel('Y'); title('Graph and Scatter');
```

Plotting

```
x = -5:0.1:5;
y = x.^2-4*x+3;
y2 = y + (2-4*rand(size(y)));
subplot(2,1,1); plot(x,y,'-',x,y2,'r.');
xlabel('X'); ylabel('Y'); title('Graph and Scatter');
subplot(2,1,2); plot(x,abs(y-y2),'r-');
xlabel('X'); ylabel('Y'); title('Absolute error');
```



Animating plots

The `drawnow` command holds the execution of your program until the graph is updated. It allows to have a live view of a simulation result.

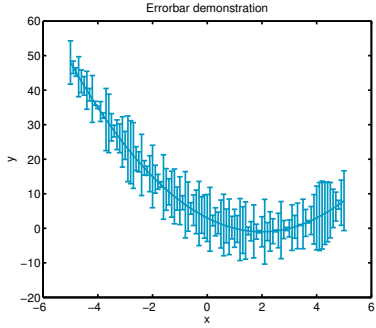
```
x = -5:0.1:5;
y = x.^2-4*x+3;
y2 = y + (2-4*rand(size(y)));
for i = 1:length(x)
    subplot(2,1,1);
    plot(x(1:i),y(1:i),'-',x(1:i),y2(1:i),'r.');
    xlabel('X'); ylabel('Y'); title('Graph and Scatter');
    axis([min(x) max(x) min(y) max(y)])

    subplot(2,1,2);
    plot(x(1:i),abs(y(1:i)-y2(1:i)),'r-');
    xlabel('X'); ylabel('Y'); title('Absolute error');
    axis([min(x) max(x) 0 2])

    drawnow
end
```

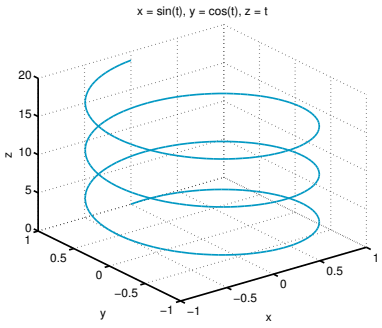
Other plotting tools

- Errorbars: `errorbar(x,y,err)`



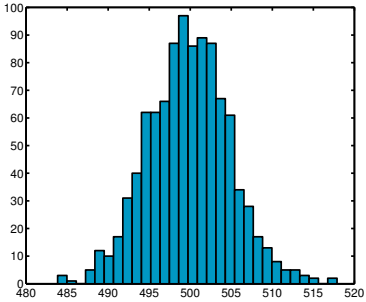
Other plotting tools

- Errorbars: `errorbar(x,y,err)`
- 3D-plots: `plot3(x,y,z)`



Other plotting tools

- Errorbars: `errorbar(x,y,err)`
- 3D-plots: `plot3(x,y,z)`
- Histograms: `histogram(x,20)`



Multi-dimensional data

Matlab typically requires the definition of rectangular grid coordinates using `meshgrid`:

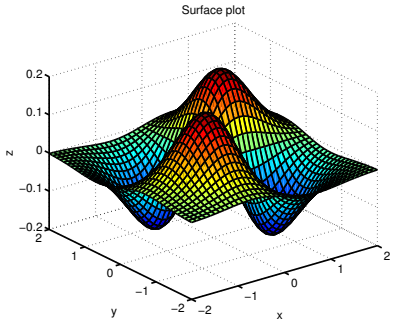
```
[x y] = meshgrid(-2:0.1:2, -2:0.1:2);
z = x .* y .* exp(-x.^2 - y.^2);
```

Multi-dimensional data

Matlab typically requires the definition of rectangular grid coordinates using `meshgrid`:

```
[x y] = meshgrid(-2:0.1:2, -2:0.1:2);
z = x .* y .* exp(-x.^2 - y.^2);
```

- Surface plot



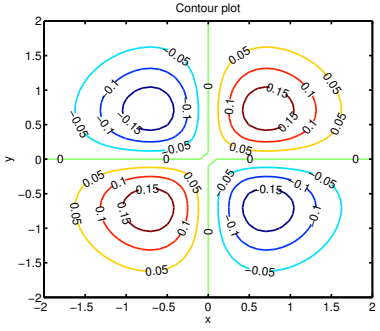

```
surf(x,y,z);
```

Multi-dimensional data

Matlab typically requires the definition of rectangular grid coordinates using `meshgrid`:

```
[x y] = meshgrid(-2:0.1:2, -2:0.1:2);
z = x .* y .* exp(-x.^2 - y.^2);
```

- Surface plot
- Contour plot



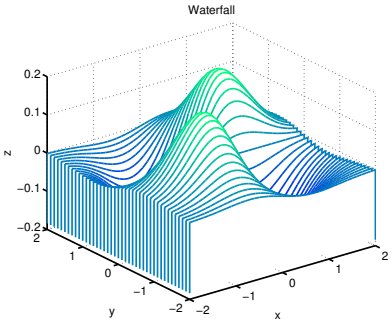

```
v=-0.5:0.05:0.5;
contour(x,y,z,v,'ShowText', 'on');
```

Multi-dimensional data

Matlab typically requires the definition of rectangular grid coordinates using `meshgrid`:

```
[x y] = meshgrid(-2:0.1:2, -2:0.1:2);
z = x .* y .* exp(-x.^2 - y.^2);
```

- Surface plot
- Contour plot
- Waterfall



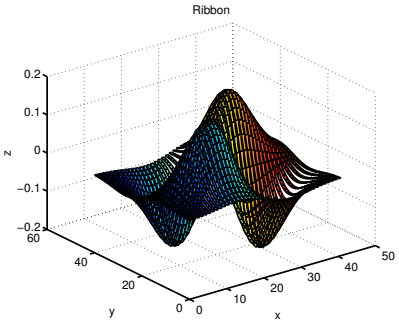

```
waterfall(x,y,z);
colormap(winter);
```

Multi-dimensional data

Matlab typically requires the definition of rectangular grid coordinates using `meshgrid`:

```
[x y] = meshgrid(-2:0.1:2, -2:0.1:2);
z = x .* y .* exp(-x.^2 - y.^2);
```

- Surface plot
- Contour plot
- Waterfall
- Ribbons




```
ribbon(z);
```

Vector data

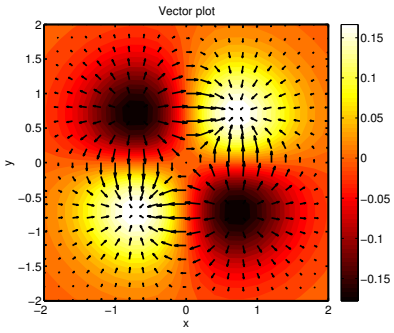
The gradient operator, as expected, is used to obtain the gradient of a scalar field. Colors can be used in the background to simultaneously plot field data:

```
[x y] = meshgrid(-2:0.2:2, -2:0.2:2);
z = x .* y .* exp(-x.^2 - y.^2)
[dx dy] = gradient(z,8,8)

% Background
contourf(x,y,z,30,'LineColor','none');
colormap(hot); colorbar;

axis tight; hold on;

% Vectors
quiver(x,y,dx,dy,'k');
```



Today's outline

- Coding style
- Debugging and profiling
- Visualisation
- Functions: revisited
- Concluding remarks

Functions: revisited

In MATLAB you can define your own functions to re-use certain functionalities. We now define the mathematical function $f = x^2 + e^x$:

```
function y = f(x)
y = x.^2 + exp(x);
```

Functions: revisited

In MATLAB you can define your own functions to re-use certain functionalities. We now define the mathematical function $f = x^2 + e^x$:

```
function y = f(x)
y = x.^2 + exp(x);
```

Note:

- The first line of the file has to contain the `function` keyword
- The variables used are *local*. They will not be available in your Workspace
- The file needs to be saved with the same name as the function, i.e. "f.m"
- The semi-colon prevents that at each function evaluation output appears on the screen
- If `x` is an array, then `y` becomes an array of function values.

Anonymous functions

If you do not want to create a file, you can create an *anonymous function*:

```
>> f = @(x) (x.^2+exp(x))
```

Anonymous functions

If you do not want to create a file, you can create an *anonymous function*:

```
>> f = @(x) (x.^2+exp(x))
```

- `f`: the name of the function
- `@`: the function handle
- `x`: the input argument
- `x.^2+exp(x)`: the actual function

Anonymous functions

If you do not want to create a file, you can create an *anonymous function*:

```
>> f = @(x) (x.^2+exp(x))
```

- `f`: the name of the function
- `@`: the function handle
- `x`: the input argument
- `x.^2+exp(x)`: the actual function

```
>> f(0:0.1:1)
```

Using function handles

A function handle points to a function. It behaves as a variable

```
>> myFunctionHandle = @exp
>> myFunctionHandle(1)
```

Using function handles

A function handle points to a function. It behaves as a variable

```
>> myFunctionHandle = @exp
>> myFunctionHandle(1)
```

Used a.o. for passing a function to another function, for instance for optimization functions.

$$f(x) = x^3 - x^2 - 3\arctan x + 1$$

Using function handles

A function handle points to a function. It behaves as a variable

```
>> myFunctionHandle = @exp
>> myFunctionHandle(1)
```

Used a.o. for passing a function to another function, for instance for optimization functions.

$$f(x) = x^3 - x^2 - 3\arctan x + 1$$

Matlab offers a function `fzero` that can find the roots of a function in a certain range:

```
>> f = @(x) x.^3 - x.^2 - 3*atan(x) + 1;
>> fzero(f,[-2 2])
>> ezplot(f)
>> f(ans)
```

Practice function handles

Consider the function

$$f(x) = -x^2 - 3x + 3 + e^{x^2}$$

The built-in Matlab function `fminbnd` allows to find the minimum of a function in a certain range. Find the minimum of $f(x)$ on $-2 \leq x \leq 2$. Example usage:

```
x = fminbnd(fun,x1,x2)
```

Practice function handles

Consider the function

$$f(x) = -x^2 - 3x + 3 + e^{x^2}$$

The built-in Matlab function `fminbnd` allows to find the minimum of a function in a certain range. Find the minimum of $f(x)$ on $-2 \leq x \leq 2$. Example usage:

```
x = fminbnd(fun,x1,x2)
```

Answer using an anonymous function:

```
>> f = @(x) -x.^2 - 3*x + 3 + exp(x.^2)
>> ezplot(f,[-2 2])
>> fminbnd(f,-2,2)
>> f(ans)
```

Practice function handles

Consider the function

$$f(x) = -x^2 - 3x + 3 + e^{x^2}$$

The built-in Matlab function `fminbnd` allows to find the minimum of a function in a certain range. Find the minimum of $f(x)$ on $-2 \leq x \leq 2$. Example usage:

```
x = fminbnd(fun,x1,x2)
```

Answer using an anonymous function:

```
>> f = @(x) -x.^2 - 3*x + 3 + exp(x.^2)
>> ezplot(f,[-2 2])
>> fminbnd(f,-2,2)
>> f(ans)
```

Various related functions are `fzero`, `feval`, `fsolve`, `fminsearch`. They will be discussed later in the course.

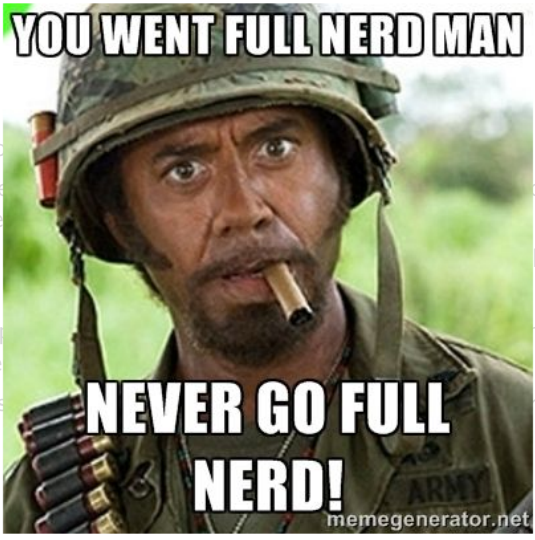
Today's outline

- Coding style
- Debugging and profiling
- Visualisation
- Functions: revisited
- Concluding remarks

Advanced concepts

- Object oriented programming: classes and objects
- Memory management: some programming languages require you to allocate computer memory yourself (e.g. for arrays)
- External libraries: in many cases, someone already built the general functionality you are looking for
- Compiling and scripting ("interpreted"); compiling means converting a program to computer-language before execution. Interpreted languages do this on the fly.
- Parallellization: Distributing expensive calculations over multiple processors or GPUs.

Advanced concepts



- Object oriented pro
- Memory manageme … o allocate computer memory yourself (e
- External libraries: i … l functionality you are looking for
- Compiling and scri … ng a program to computer-language … on the fly.
- Parallellization: Dis … rocessors or GPUs.

Make a habit of the following adage

MAKE IT **WORK**

MAKE IT **RIGHT**

MAKE IT **FAST**

Make a habit of the following adage

1. *Make it work*
   Create an algorithm that does the intended job. Make sure it works, and works repeatedly. Test and verify frequently. Add *todo* comments when you're not sure about a certain decision.
2. *Make it right*
   Refactor the code to improve the code design. Insert functions, comments, compartmentalize it. Get rid of magic numbers, use sensible variable names. Check input. Test and verify. Align with the team!
3. *Make it fast*
   Measure and tune the performance of your code (profiling tool). In Matlab, vectorized calculations are much (!) faster than for-loops. Use sensible numerical techniques (e.g. higher-order integration).

Program by iterating over these aspects multiple times, starting at the fine-grained level, working your way up.

Make a habit of the following adage

1. *Make it work*
   Create an algorithm that does the intended job. Make sure it works, and works repeatedly. Test and verify frequently. Add *todo* comments when you're not sure about a certain decision.
2. *Make it right*
   Refactor the code to improve the code design. Insert functions, comments, compartmentalize it. Get rid of magic numbers, use sensible variable names. Check input. Test and verify. Align with the team!
3. *Make it fast*
   Measure and tune the performance of your code (profiling tool). In Matlab, vectorized calculations are much (!) faster than for-loops. Use sensible numerical techniques (e.g. higher-order integration).

Program by iterating over these aspects multiple times, starting at the fine-grained level, working your way up.

Make a habit of the following adage

1. *Make it work*
   Create an algorithm that does the intended job. Make sure it works, and works repeatedly. Test and verify frequently. Add *todo* comments when you're not sure about a certain decision.
2. *Make it right*
   Refactor the code to improve the code design. Insert functions, comments, compartmentalize it. Get rid of magic numbers, use sensible variable names. Check input. Test and verify. Align with the team!
3. *Make it fast*
   Measure and tune the performance of your code (profiling tool). In Matlab, vectorized calculations are much (!) faster than for-loops. Use sensible numerical techniques (e.g. higher-order integration).

Program by iterating over these aspects multiple times, starting at the fine-grained level, working your way up.

Exercise: finding the roots of a parabola

We are writing a program that finds for us the roots of a parabola. We use the form

$$y = ax^2 + bx + c$$

Exercise: finding the roots of a parabola

We are writing a program that finds for us the roots of a parabola. We use the form

$$y = ax^2 + bx + c$$

What is our program in pseudo-code?

Exercise: finding the roots of a parabola

We are writing a program that finds for us the roots of a parabola. We use the form

$$y = ax^2 + bx + c$$

What is our program in pseudo-code?

1. Input data (a, b and c)

Exercise: finding the roots of a parabola

We are writing a program that finds for us the roots of a parabola. We use the form

$$y = ax^2 + bx + c$$

What is our program in pseudo-code?

1. Input data (a, b and c)
2. Identify special cases ($a = b = c = 0$, $a = 0$)

Exercise: finding the roots of a parabola

We are writing a program that finds for us the roots of a parabola. We use the form

$$y = ax^2 + bx + c$$

What is our program in pseudo-code?

1. Input data (a, b and c)
2. Identify special cases ($a = b = c = 0$, $a = 0$)

   $a = b = c = 0$ Solution indeterminate

Exercise: finding the roots of a parabola

We are writing a program that finds for us the roots of a parabola. We use the form

$$y = ax^2 + bx + c$$

What is our program in pseudo-code?

1. Input data (a, b and c)
2. Identify special cases ($a = b = c = 0$, $a = 0$)

   $a = b = c = 0$ Solution indeterminate

   $a = 0$ Solution: $x = -\frac{c}{b}$

Exercise: finding the roots of a parabola

We are writing a program that finds for us the roots of a parabola. We use the form

$$y = ax^2 + bx + c$$

What is our program in pseudo-code?

1. Input data (a, b and c)
2. Identify special cases ($a = b = c = 0$, $a = 0$)
   $a = b = c = 0$ Solution indeterminate
   $a = 0$ Solution: $x = -\frac{c}{b}$
3. Find $D = b^2 - 4ac$

Exercise: finding the roots of a parabola

We are writing a program that finds for us the roots of a parabola. We use the form

$$y = ax^2 + bx + c$$

What is our program in pseudo-code?

1. Input data (a, b and c)
2. Identify special cases ($a = b = c = 0$, $a = 0$)
   $a = b = c = 0$ Solution indeterminate
   $a = 0$ Solution: $x = -\frac{c}{b}$
3. Find $D = b^2 - 4ac$
4. Decide, based on D:

Exercise: finding the roots of a parabola

We are writing a program that finds for us the roots of a parabola. We use the form

$$y = ax^2 + bx + c$$

What is our program in pseudo-code?

1. Input data (a, b and c)
2. Identify special cases ($a = b = c = 0$, $a = 0$)
   - $a = b = c = 0$ Solution indeterminate
   - $a = 0$ Solution: $x = -\frac{c}{b}$
3. Find $D = b^2 - 4ac$
4. Decide, based on D:
   - $D < 0$ Display message: complex roots

Exercise: finding the roots of a parabola

We are writing a program that finds for us the roots of a parabola. We use the form

$$y = ax^2 + bx + c$$

What is our program in pseudo-code?

1. Input data (a, b and c)
2. Identify special cases ($a = b = c = 0$, $a = 0$)
   - $a = b = c = 0$ Solution indeterminate
   - $a = 0$ Solution: $x = -\frac{c}{b}$
3. Find $D = b^2 - 4ac$
4. Decide, based on D:
   - $D < 0$ Display message: complex roots
   - $D = 0$ Display 1 root value

Exercise: finding the roots of a parabola

We are writing a program that finds for us the roots of a parabola. We use the form

$$y = ax^2 + bx + c$$

What is our program in pseudo-code?

1. Input data (a, b and c)
2. Identify special cases ($a = b = c = 0$, $a = 0$)
   - $a = b = c = 0$ Solution indeterminate
   - $a = 0$ Solution: $x = -\frac{c}{b}$
3. Find $D = b^2 - 4ac$
4. Decide, based on D:
   - $D < 0$ Display message: complex roots
   - $D = 0$ Display 1 root value
   - $D > 0$ Display 2 root values

Example: finding the roots of a parabola

```
function x = parabola(a,b,c)
% Catch exception cases
if (a==0)
    if(b==0)
        if(c==0)
            disp('Solution indeterminate'); return;
        end
        disp('There is no solution');
    end
    x = -c/b;
end

% Compute determinant
D = b^2 - 4*a*c;
if (D<0)
    disp('Complex roots'); return;
    else if (D==0)
        x = -b/(2*a);
        else if (D>0)
                x(1) = (-b + sqrt(D))/(2*a);
                x(2) = (-b - sqrt(D))/(2*a);
                x = sort(x);
        end
    end
end
```

Example: finding the roots of a parabola

```
>> roots([1 -4 -3])
ans =
    4.6458
   -0.6458
```