Matlab and Programming 1

Programming basics and algorithms

Ivo Roghair, Martin van Sint Annaland

Chemical Process Intensification
Eindhoven University of Technology

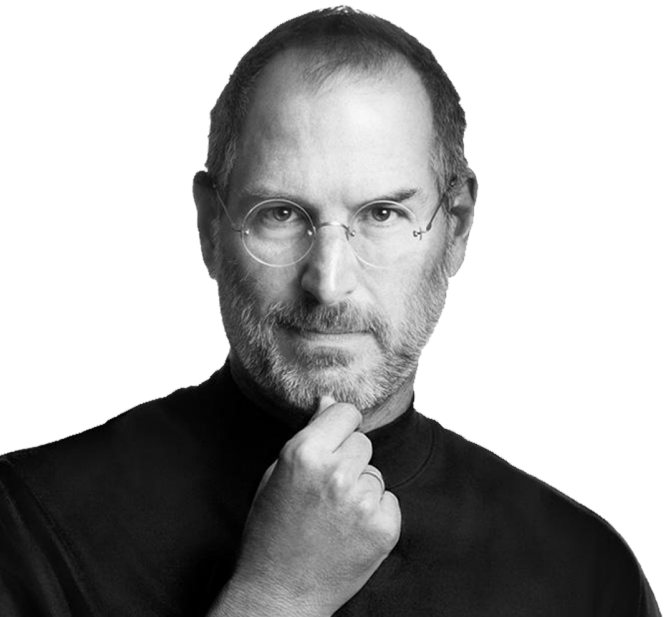
Today's outline

Today's outline

Programming

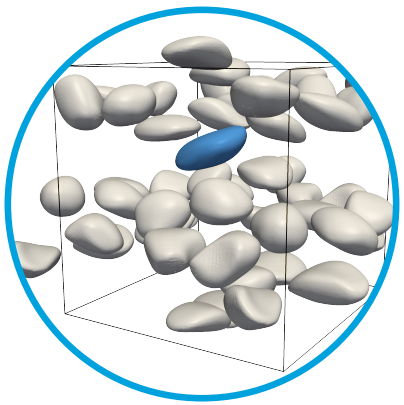
"Everybody in this country should learn to program a computer, because it teaches you to think.."

—Steve Jobs



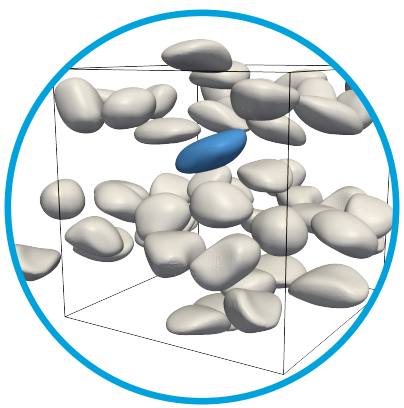
Why?

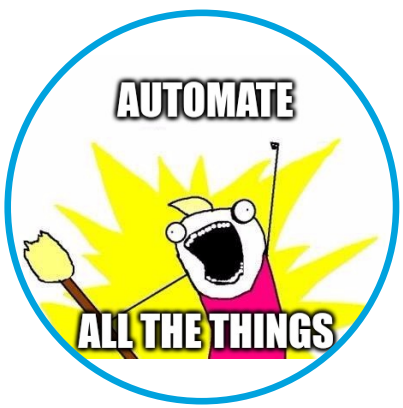
- Scientific techniques depend in an increasing fashion upon computer programs and simulation methods



Why?

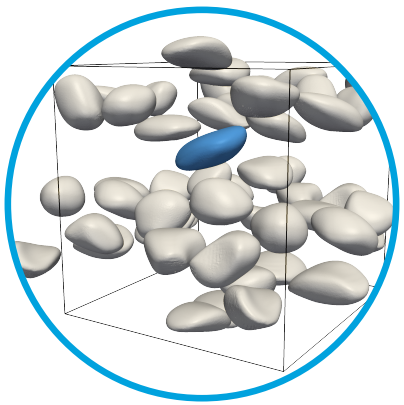
- Scientific techniques depend in an increasing fashion upon computer programs and simulation methods
- Knowledge of programming allows you to automate routine tasks

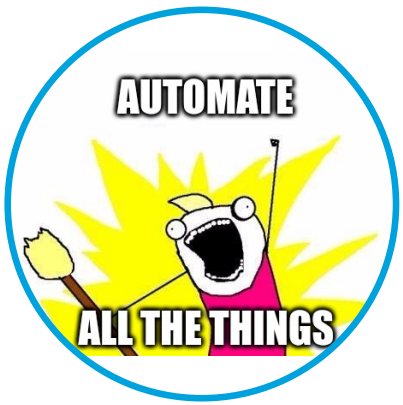




Why?

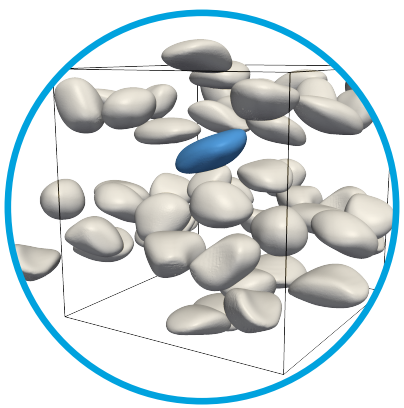
- Scientific techniques depend in an increasing fashion upon computer programs and simulation methods
- Knowledge of programming allows you to automate routine tasks
- Ability to understand algorithms by inspection of the code

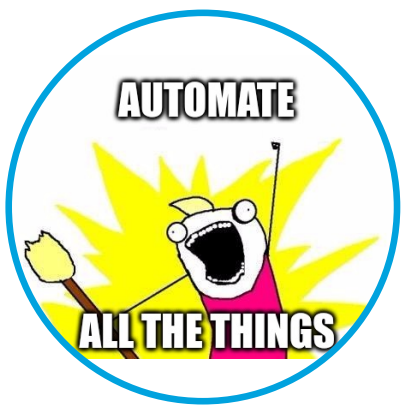




Why?

- 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





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
```

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, such as solving a system of equations or finding the roots of a polynomial
- It can also be a symbolic computation, such as searching and replacing text in a document
- A program may even be used to compile another program
- A program consists of one or more *algorithms*

Getting started

- Use an *integrated development environment*
  - Matlab
  - MS Visual Studio
  - Eclipse
  - Dev C++
  - IDLE, Canopy (express)
- Create a simple program:
  - Hello world
  - Find the roots of a parabola
  - Find the greatest common divisor of two numbers

Some often used programming languages

Python

- Many functionalities available
- Smooth learning curve
- Slow compared to compiled languages
- Many freely available editors

Some often used programming languages

Python

- Many functionalities available
- Smooth learning curve
- Slow compared to compiled languages
- Many freely available editors

Pascal

- Limited number of libraries available
- Steep learning curve
- Compiled language, may be fast
- Some free compilers (fpc)

Some often used programming languages

Python

- Many functionalities available
- Smooth learning curve
- Slow compared to compiled languages
- Many freely available editors

C / C++ / C#

- Many functionalities available
- Steeper learning curve
- Needs compilation, very fast (HPC)
- Freely available (gcc, MSVC)

Pascal

- Limited number of libraries available
- Steep learning curve
- Compiled language, may be fast
- Some free compilers (fpc)

Some often used programming languages

Python

- Many functionalities available
- Smooth learning curve
- Slow compared to compiled languages
- Many freely available editors

Pascal

- Limited number of libraries available
- Steep learning curve
- Compiled language, may be fast
- Some free compilers (fpc)

C / C++ / C#

- Many functionalities available
- Steeper learning curve
- Needs compilation, very fast (HPC)
- Freely available (gcc, MSVC)

Spreadsheet (Excel, Google Docs, ...)

- High availability
- Low learning curve
- Very limited for larger problems, unbeatable for quick calculations
- Not always free

Some often used programming languages

Python

- Many functionalities available
- Smooth learning curve
- Slow compared to compiled languages
- Many freely available editors

C / C++ / C#

- Many functionalities available
- Steeper learning curve
- Needs compilation, very fast (HPC)
- Freely available (gcc, MSVC)

Pascal

- Limited number of libraries available
- Steep learning curve
- Compiled language, may be fast
- Some free compilers (fpc)

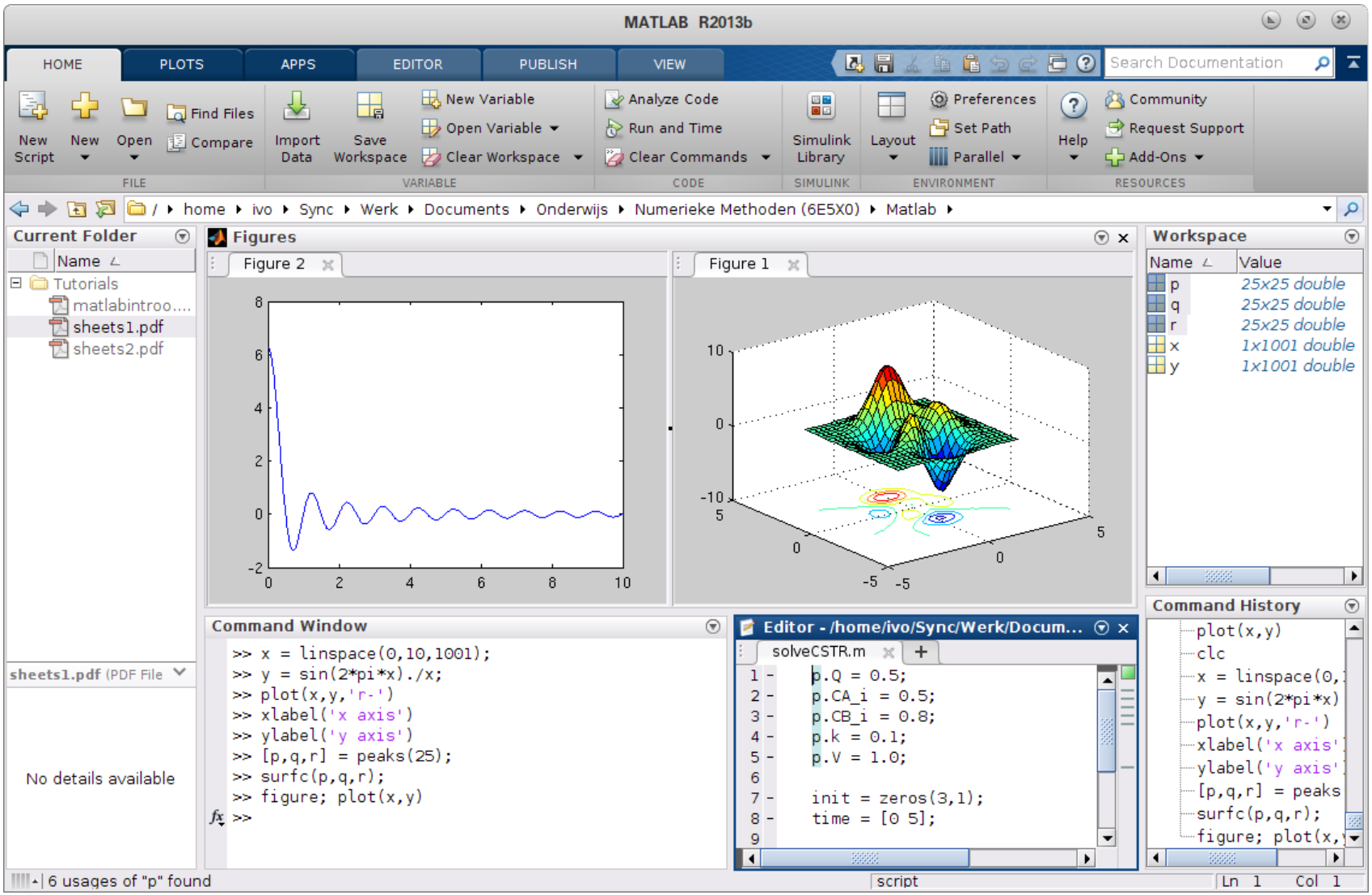
Spreadsheet (Excel, Google Docs, ...)

- High availability
- Low learning curve
- Very limited for larger problems, unbeatable for quick calculations
- Not always free

Matlab

- Many functionalities built-in (80+ toolkits!)
- Slow compared to compiled languages
- Fairly smooth learning curve
- Needs a license (alternatives: SciLab, GNU Octave)

Versatility of Matlab

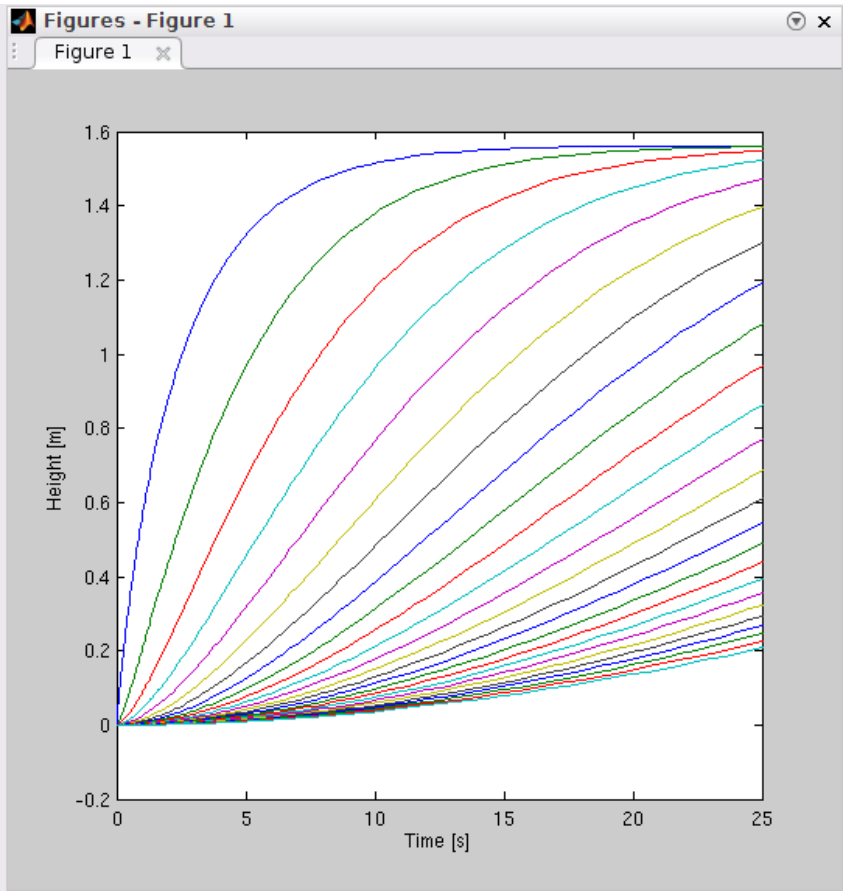


Versatility of Matlab: ODE solver

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

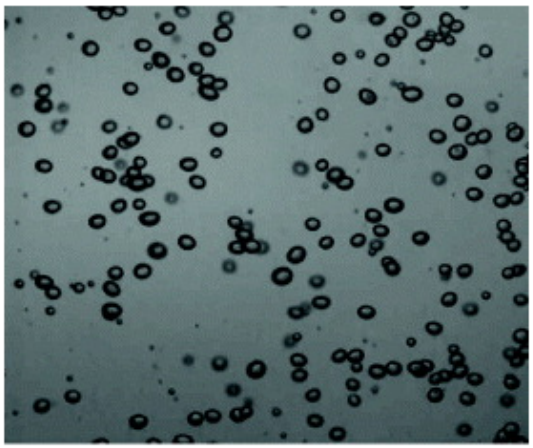
solveTankSeries.m | tankSeries.m

```
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]');
```



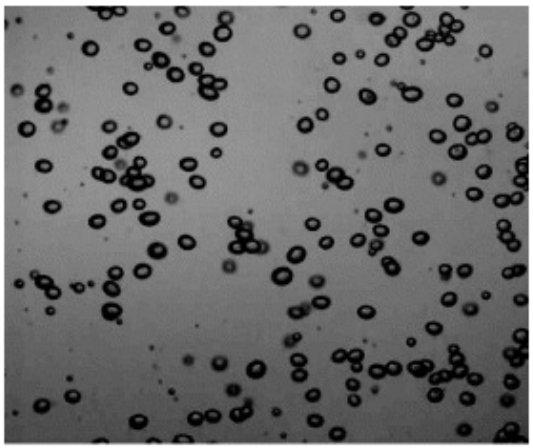
Versatility of Matlab: Image analysis

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



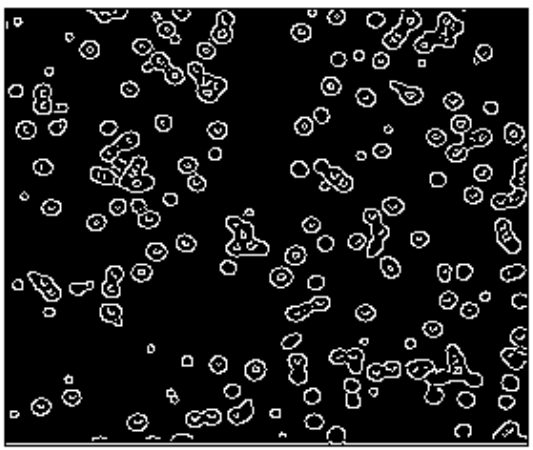
Versatility of Matlab: Image analysis

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



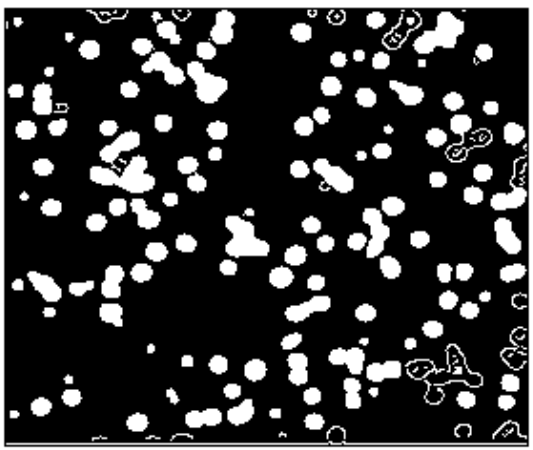
Versatility of Matlab: Image analysis

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

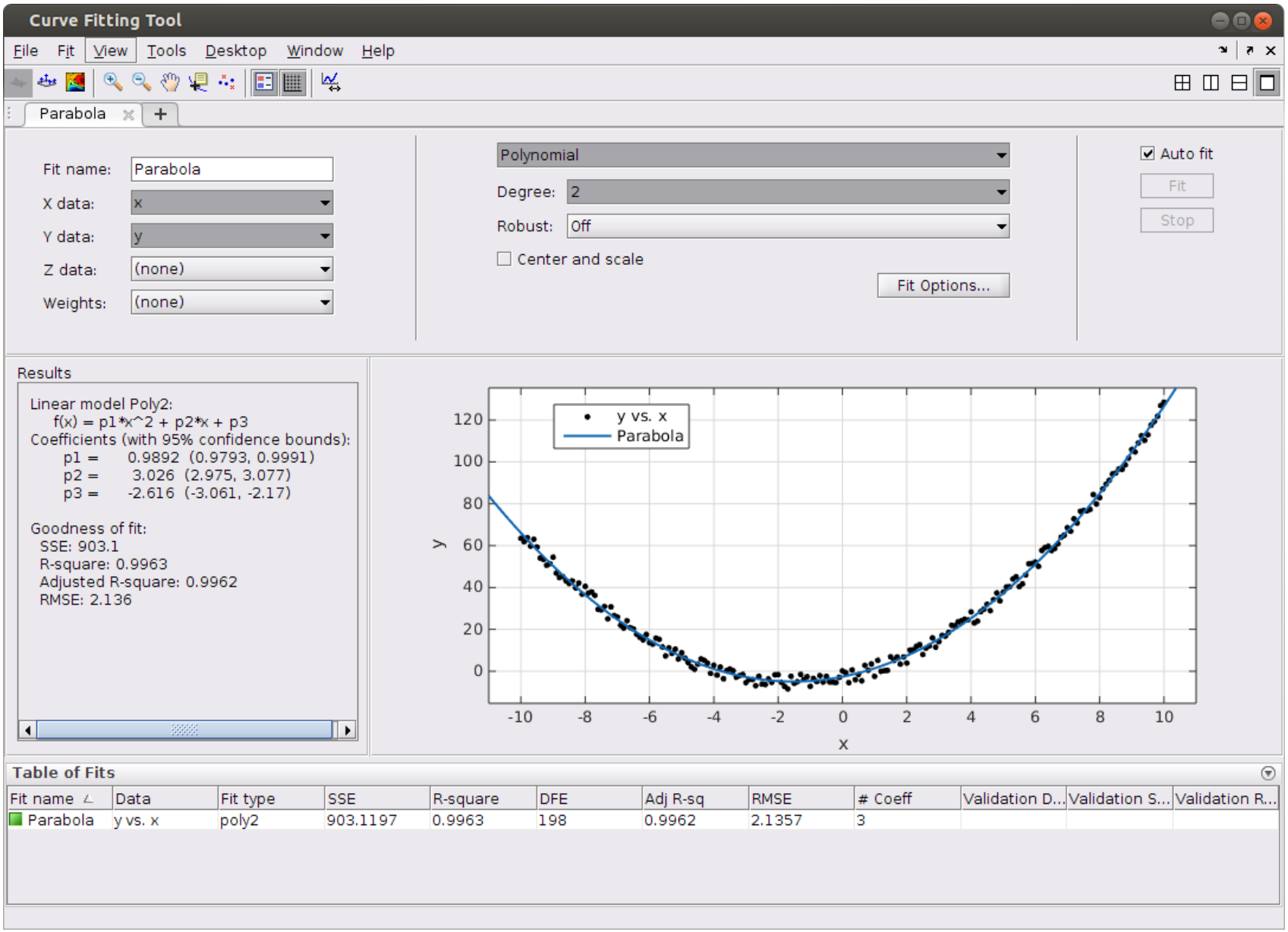


Versatility of Matlab: Image analysis

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



Versatility of Matlab: Curve fitting



Matlab help

- Matlab documentation: `doc` or `help` function

Matlab help

- Matlab documentation: `doc` or `help` function
- Canvas page

Matlab help

- Matlab documentation: `doc` or `help` function
- Canvas page
- Introduction to Numerical Methods and Matlab Programming for Engineers. T. Young and M.J. Mohlenkamp (2015). GNU-licensed document, online

Matlab help

- Matlab documentation: `doc` or `help` function
- Canvas page
- Introduction to Numerical Methods and Matlab Programming for Engineers. T. Young and M.J. Mohlenkamp (2015). GNU-licensed document, online
- Search the web!

Today's outline

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`.

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:

```
>> 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)
```

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

Printing results

You can prevent displaying the outcome of a command by adding a semi-colon at the end of a line:

```
>> c = linspace(0,10,11);
>> length(c)
>> c
>> size(c)
```

Printing results

You can prevent displaying the outcome of a command by adding a semi-colon at the end of a line:

```
>> c = linspace(0,10,11);
>> length(c)
>> c
>> size(c)
```

Altering the display format can be done using the `format` command:

```
>> format compact % loose
>> format long    % short
```

Simple plotting

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

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

```
>> x = [ 5 20 30 50 55 ]
>> y = [ 0.08 0.015 0.009 0.006 0.0055]
```

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

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

Simple plotting

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

```
>> x = [ 5 20 30 50 55 ]
>> y = [ 0.08 0.015 0.009 0.006 0.0055]
```

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

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

Simple plotting

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

```
>> x = [ 5 20 30 50 55 ]
>> y = [ 0.08 0.015 0.009 0.006 0.0055]
```

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

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

Simple plotting

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

```
>> x = [ 5 20 30 50 55 ]
>> y = [ 0.08 0.015 0.009 0.006 0.0055]
```

```
>> plot(x,y,'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
```

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?
- How to obtain the maximum for each row?
- Use a vector multiplication to compute the following matrix:

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{bmatrix}$$

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).

Building blocks: conditional statements

`if`-statement: Check whether a (set of) condition(s) is met.

Building blocks: conditional statements

`if`-statement: Check whether a (set of) condition(s) is met.

```
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.

```
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 less than or equal to
>=	is greater than or equal to
<	is less than
>	is greater than

Combining conditional statements

&&	and
||	or
xor	exclusive or

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!

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
```

Input and output

Many programs require some input 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 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 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');
```

Input and output

Many programs require some input 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');
```

- 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`, ...

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).
  - If functions do not return a result, it is called a procedure
- In Matlab, functions are defined as follows (2 output variables and 3 input arguments):

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).
  - If functions do not return a result, it is called a procedure
- In Matlab, functions are defined as follows (2 output variables and 3 input arguments):

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).
  - If functions do not return a result, it is called a procedure
- In Matlab, functions are defined as follows (2 output variables and 3 input arguments):

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).
  - If functions do not return a result, it is called a procedure
- In Matlab, functions are defined as follows (2 output variables and 3 input arguments):

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

Functions - locality and arguments

- You are supplying arguments to a function because it does not have acces to previously defined variables. This is called *locality*.
  - This does not include global variables - but they're evil!
  - Local variables created in a function are not accessible to other functions unless they are returned or supplied as an argument!

Functions - locality and arguments

- You are supplying arguments to a function because it does not have acces to previously defined variables. This is called *locality*.
  - This does not include global variables - but they're evil!
  - Local variables created in a function are not accessible to other functions unless they are returned or supplied as an argument!

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

Functions - locality and arguments

- You are supplying arguments to a function because it does not have acces to previously defined variables. This is called *locality*.
  - This does not include global variables - but they're evil!
  - Local variables created in a function are not accessible to other functions unless they are returned or supplied as an argument!

Exercise: 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
```

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
```

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 is called by itself (a function within a function)

Recursion

- In order to understand recursion, one must first understand recursion
- A recursive function is called by 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

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

Recursion

- In order to understand recursion, one must first understand recursion
- A recursive function is called by itself (a function within a function)

Recursion

- In order to understand recursion, one must first understand recursion
- A recursive function is called by 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
```

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

- Next lecture: advanced practices (prepare, read instructions on Canvas)
- For now: exercises 1-4 (basics), 5+6 (advanced).

In conclusion...

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

- Next lecture: advanced practices (prepare, read instructions on Canvas)
- For now: exercises 1-4 (basics), 5+6 (advanced).

In conclusion...

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

- Next lecture: advanced practices (prepare, read instructions on Canvas)
- For now: exercises 1-4 (basics), 5+6 (advanced).