# Algorithms, validation and visualisation

## Introduction to programming in Matlab

Ivo Roghair, Edwin Zondervan, Martin van Sint Annaland

Chemical Process Intensification
Process Systems Engineering



## Presentation outline

1. Introduction
2. Programming basics
3. Eliminating errors
4. Visualisation
5. Examples
6. Conclusions



## 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*



## Algorithm design

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

## About programming

**What is programming?**

Constructing a (series of) algorithm(s) that fulfill a certain function

- Translate your problem to a formal procedure (recipe, pseudo-code)
  - What steps do I need to do?
  - Can you break down a step further?
  - Is the order of the steps of importance?
  - Can I re-use certain parts?
- Translate your formal procedures to machine instructions
  - Learning a programming language: syntax



## Getting your hands dirty

- 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



## 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,LibreOffice Calc, ...)**

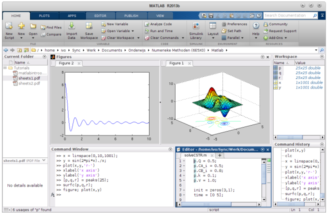
- High availability
- Low learning curve
- Very limited for larger problems, unbeatable for quick calculations
- Not always free

**Matlab**

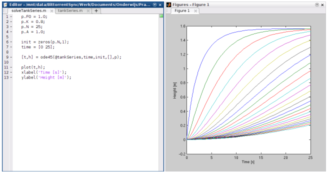
- Many functionalities built-in (80+ toolkits!)
- Slow compared to compiled languages
- Fairly smooth learning curve
- Needs a license, not available everywhere (alternatives: SciLab, GNU Octave)



## Versatility of Matlab



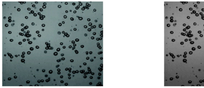
## Versatility of Matlab: ODE solver





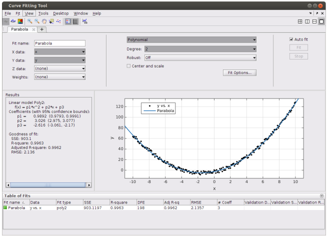

## Versatility of Matlab: Image analysis

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





## Versatility of Matlab: Curve fitting






## Matlab resources

- Matlab documentation
- Introduction to Numerical Methods and Matlab Programming for Engineers. Todd Young and Martin J. Mohlenkamp (2014). GNU-licensed document, online
- Interactive Matlab Course. Pieter van Zutven (2010). See also http://www.imc.tue.nl/ (check website for the PDF)
- Search the web!

## Presentation outline

1 Introduction

2 Programming basics

3 Eliminating errors

4 Visualisation

5 Examples

6 Conclusions



## Syntax and semantics

**Syntax (the form)**

Correctness of the structure of symbols

```
x = 3 * 4; %ok
x + c = 5 car %wrong
```

**Semantics (the meaning)**

Opposed to natural language, programming languages are designed to prevent ambiguous, non-sensical statements
"Giraffes wait ravenously because the King of Scotland touched March"



## Programming basics

Programs consist of a number of expressions that form the algorithm.

- An expression is a command, combining functions, variables, operators and/or values to produce a result.
- Variables contain one or more value(s)
- Operators act on the data in variables (compare, add, multiply)
- Functions perform an operation on one or more variables and return one or more result(s).

The following will very shortly discuss some important aspects of variables, operators and functions that can be of use when creating your algorithms.



## Variables

- Data is stored in the memory of your computer, and can be read/updated using *variables*.
  - Matlab stores variables in the *workspace*
- A variable is not always the same as the mathematical concept of variable (i.e. part of an equation).
- You should recognize the difference between the *identifier* of a variable (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`

## 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'<br>sct.number = 13 |
| logical | 0 (false)<br>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: 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 $y$ |



## 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: conditional statements

`if`-statement: Performs a block of code if a certain condition is met.

```
num = floor(10*rand+1);
guess = input('Your guess please: ');
if (guess ~= num)
    disp('That is wrong. Have a nice day');
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: 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: 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
disp('Correct!');
```



## 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, ' ', 5)
    , '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;
- Output may be stored via the GUI:
  - 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

A function in a programming language is a program fragment that 'knows' how to perform a defined task. Creating good functions keeps your code clean, re-usable and structured.

- You will 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

- 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. These are often considered evil.
  - The locality of variables also causes that local variables created in a function are not accessible to other functions unless they are returned or supplied as an argument!

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

```
function result = avg(a,b,c)
    mySum = a + b + c;
    result = mySum / 3;
end
```

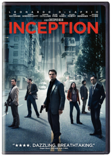


## Recursion

- 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
- 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
```

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



## Presentation outline

1. Introduction
2. Programming basics
3. Eliminating errors
4. Visualisation
5. Examples
6. Conclusions



## Errors in computer programs

Computer programs often contain errors (bugs): buildings collapse, governments fall, kittens will die.

## 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.



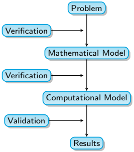
## Verification and validation

**Verification**

Verification is the process of mathematically and computationally assuring that the model computes what you have entered.

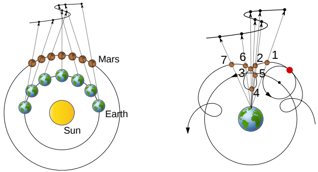
**Validation**

Validation is the process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model






## Be aware of your uncertainties




- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)
- Even though they were not 'right', Earth-centered models (Ptolemy) were still valid



## Be aware of your uncertainties

**Aleatory uncertainty**

Uncertainty that arises due to inherent randomness of the system, features that are too complex to measure and take into account

**Epistemic uncertainty**

Uncertainty that arises due to lack of knowledge of the system, but could in principle be known

## A convenient tool: the debugger

- 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.

*"When you have eliminated the impossible, whatever remains, however improbable, must be the truth."*
— A. Conan Doyle, The Sign of Four



## A convenient tool: the debugger

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!



## About testcases (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 continuing program)?
- For physical modeling, we typically look for analytical solutions
  - Sometimes somewhat stylized cases
  - Possible solutions include Fourier-series
  - Experimental data



## 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.
- Profiling, optimization, parallellization: Checking where your program spends the most of its time, optimizing (or parallellizing) that part.

## If anything sticks today, let it be this

### Your code will not be understood by anyone

That includes future-you

This can be prevented somewhat by the following

- Use comments! In Matlab, everything following `% is a comment`
- Prevent "smart constructions". You will spend a day tinkering why it does what it does...
- If you write unmaintainable code, you'll have a job for life.
- Use comments! Documentation is also useful (though hard to maintain)



## Presentation outline

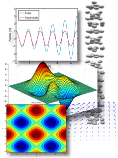
1. Introduction
2. Programming basics
3. Eliminating errors
4. Visualisation
5. Examples
6. Conclusions



## Data visualisation

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

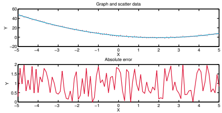
- 1D, 2D, 3D visualisation
- Multiple variables at the same time (temperature, concentration, direction of flow)
- Use of colors, contour lines
- Use of stream lines or vector plots
- Animations





## Plotting

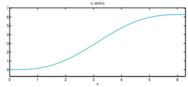
```
x = -5:0.1:5;
y = x.^2-4*x+3;
y2 = y + (2-4*rand(size(y)));
subplot(2,1,1); plot(x,y,'-',x,y2,'r.');
xlabel('X'); ylabel('Y'); title('Graph and scatter
    data');
subplot(2,1,2); plot(x,abs(y-y2),'r-');
xlabel('X'); ylabel('Y'); title('Absolute error');
```



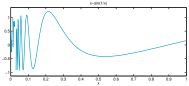
## Plotting (2)

Easy plotting of functions can be done using the ezplot function:

```
ezplot('x-sin(x)', [0 2*pi]);
```



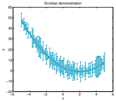

Be careful with steep gradients: `ezplot('x-sin(1/x)', [0 1])`

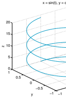





## Other plotting tools

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



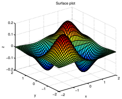


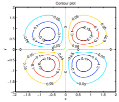
## Multi-dimensional data

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);
```

- Surface plot
- Contour plot
- Waterfall
- Ribbons



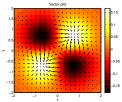

```
surf(x,y,z);
v=-0.5:0.05:0.5;
contour(x,y,z,v,'ShowText'
, 'on');
waterfall(x,y,z);
colormap(winter);
ribbon(z);
```



## 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');
```



## Presentation outline

1 Introduction
2 Programming basics
3 Eliminating errors
4 Visualisation
5 Examples
6 Conclusions



## Example: 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

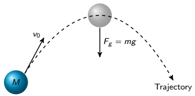
```
function x = parabola(a,b,c)
% Catch exception cases
if (a==0)
    if(b==0)
        if(c==0)
            disp('Solution indeterminate'); return;
        end
        disp('There is no solution');
    end
    x = -c/b;
end

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
```

## Example: projectile trajectory

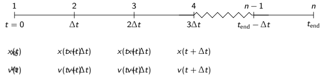



- A ball with mass $M$ is thrown at time $t = 0$ with a certain velocity $v(t) = v(0) = v_0$
- We need to describe the trajectory of the ball over time
- It is given that the only force acting on the ball is gravity: $F = Mg$



## Example: projectile trajectory

Computers cannot solve a continuous equation; we need to *discretize* the time into steps of size $\Delta t$. Create a time line:






## Example: projectile trajectory

- A Taylor expansion shows how the $x$-position is obtained at discrete time intervals:

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \ldots$$

$$x(t+\Delta t) = x(t) + \frac{\frac{dx}{dt}(t)}{1!}(t+\Delta t - t) + \frac{\frac{d^2x}{dt^2}}{2!}(t+\Delta t - t)^2 + O(\Delta t^3)$$

$$x(t+\Delta t) = x(t) + v(t)\Delta t + \frac{F}{2M}\Delta t^2 + O(\Delta t^3)$$

- Taking small time steps, we can discard $\Delta t^2$ and subsequent terms:

$$x(t+\Delta t) = x(t) + v(t)\Delta t$$

- A similar approach is taken for the velocity:

$$v(t+\Delta t) = v(t) + a(t)\Delta t$$

$$F = Ma \Rightarrow a = \frac{F}{M} \Rightarrow v(t+\Delta t) = v(t) + \frac{F(t)}{M}\Delta t$$



## Example: projectile trajectory

Our mathematical model is as follows:

1. Initialisation of parameters ($x_0$, $v_0$, $g$, $\Delta t$, $t_end$, $M$)
2. Create storage vectors for time, position, velocity
3. Start a time-marching loop
   - Calculate $x(t+\Delta t)$, then $F$, then $v(t+\Delta t)$:

     $$x(t+\Delta t) = x(t) + v(t)\Delta t$$

     $$F = Mg$$

     $$v(t+\Delta t) = v(t) + \frac{F}{M}\Delta t$$

   - Store current solution
4. Draw result and return solution vector $x$
   - Exact solution:

     $$x(t) = x_0 + v_0 t + (\frac{1}{2} - 9.81t^2)$$

# Example: projectile trajectory - solution (initialisation)

```
function [pos,tim] = projectile(v0,M)

% Initialise parameters
t_end = 2;                  % End time
deltat = 0.01;              % Time step
x0 = 1;                     % Initial position

nsteps = fix(t_end/deltat); % Number of time steps
pos = zeros(nsteps,1);      % Position vector
vel = zeros(nsteps,1);      % Velocity vector
tim = zeros(nsteps,1);      % Time vector

% Default values for mass and velocity
if (nargin < 2)
    M = 10;
    if (nargin < 1)
        v0 = 1;
    end
end
```



# Example: projectile trajectory - solution (main program)

```
pos(1) = x0;                % Store initial position
vel(1) = v0;                % Store initial velocity

% The time loop
for n = 1:nsteps-1
    pos(n+1) = position(pos(n),vel(n),deltat);
    vel(n+1) = velocity(vel(n),M,deltat);
    tim(n+1) = tim(n) + deltat;
end

% Plot results
figure; plot(tim,pos, 'o');

% Compare to analytical solution
compareToExact(x0,v0,tim,pos);

end
```



# Example: projectile trajectory - solution (added functions)

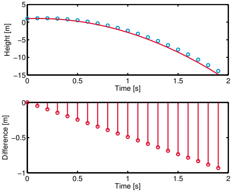
```
function F = force(M)
% M:    mass of particle
g = -9.81;
F = M * g;
end

function v = velocity(vt,mass,dt)
% vt:   velocity at previous time
% mass: mass of particle
% dt:   time step size
v = vt + force(mass)/mass * dt;
end

function x = position(xt,vel,dt)
% xt:   position at current time step
% vel:  velocity at current time step
% dt:   time step size
x = xt + vel * dt;
end
```



# Example: projectile trajectory - solution (verification)

```
function compareToExact(x0,v0,tim,pos)

% Exact solution
pos_ex = x0 + v0 * tim + ( 0.5 * -9.81 * tim .* tim );

% Draw comparative figure
figure;
subplot(2,1,1)
plot(tim,pos, 'o');
hold on;
plot(tim,pos_ex,'r-')
subplot(2,1,2)
stem(tim,pos_ex-pos,'r-')

% Print the L2-error norm
norm(pos_ex - pos)
end
```

## Example: projectile trajectory - solution






## In conclusion...

- Algorithm design: define your problem, think ahead, make a scheme, sketch the interplay between variables and functions, then start programming
- Programming basics: variables, operators and functions, locality of variables, recursive operations
- Dealing with complex programs, verification of your algorithms, use of the debugger
- Visualisation: how to make 1D and 2D/3D plots, create a sensible and intuitive presentation of your data.
- Examples: a few practice cases