Math 116 – Programming with Matlab®

Lukas Burch, Elise Le Meledo, Fatemeh Nassajianmojarrad

10 – 14th February, 2020

Universitat Zurich

Course organisation

Aim of the course

- Understand the spirit of Matlab® language
- Be fluent in the syntax and in the basic usage of Matlab®
- Develop a programming-oriented thinking
- Write your own (simple) scripts and functions

Organisation

- Lectures are held in H12, H46, H52
- Morning session: 9:00 to 12:00
- Afternoon session: 13:00 to 17:00

Course organisation

Exam

- Programming exam, similar to exercises
- Venue: 14th of February, 9:00 to 11:00, Y27
- Rooms: H52 and H46

Course material

- Course material (slides, example files, exam info):
  UZH website → Vorlesungen → Programmierung in MatLab
- Matlab® software:
  https://www.zi.uzh.ch/de/students/software-elearning/softwareinstructions/Matlab.html

Course organisation

Covered topics

- Variables definition
  Basic operations
- Scripts
  User-defined functions
- Conditional and loops
- Arrays, matrices
  Vectorial computations
- Results visualisation
- Advanced data types
- Input import, output export
  Data handling
- (Linear systems solving)
  (Symbolic toolbox)

Course organisation

Lecture workflow

1. A theoretical notion is introduced
2. Related Matlab® commands are presented
3. Examples are given
4. Autonomous exercise sessions are held

→ Try the content of any such blue box in the Matlab® console

→When you see the symbol , work on the exercise in autonomy

→ When you see , guess the answer to the asked question

Pay a special attention when you see the symbol . It points out a practical information that will make your coding habits better.

Course organisation

How to access and use Matlab®

Using Matlab® at the university

1. Connect to Thinlinc client with your math account credentials
   https://www.math.uzh.ch/li/index.php?id=thinlinc
2. Go to Applications $\rightarrow$ Science $\rightarrow$ Matlab
   or open a terminal and type `matlab --nodesktop`
3. Enter your Thinlinc password

Using Matlab® on your own computer

1. Download, install and activate Matlab®
2. Launch Matlab® from either the console or the created icon

Course organisation

Installing Matlab®

- Download Matlab® from the Mathwork's website: `https://ch.mathworks.com/academia/tah-portal/universitaet-zuerich-970406.html#get`
- Download the TAH Activation key from the university website `https://www.zi.uzh.ch/de/teaching-and-research/software-elearning/softwareinstructions/Matlab.html`
- Install Matlab® by following the installer instructions
- Activate the license with your TAH activation key.
  Warning: if you are using a UNIX system, your username should match your session's login name

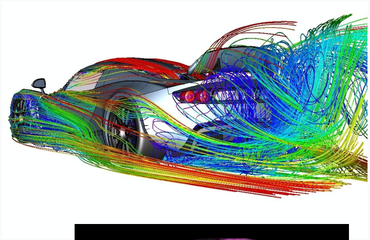
Course organisation

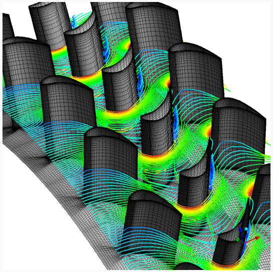
Getting help

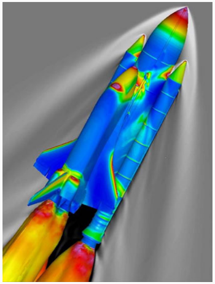
- Built-in help of Matlab®
  Type `help help` *in the Matlab® prompt*
- Matlab®online help
  `https://www.mathworks.com/help/matlab`
- Extra material: script of Felix Fontein
  Available on the HS15 MATH116 website

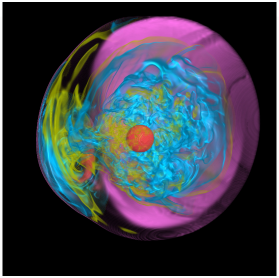
Introduction

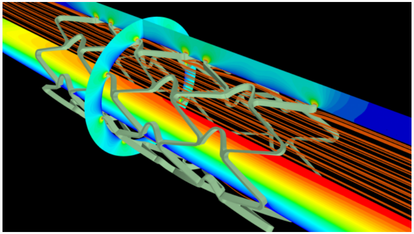
Motivation: numerical simulations



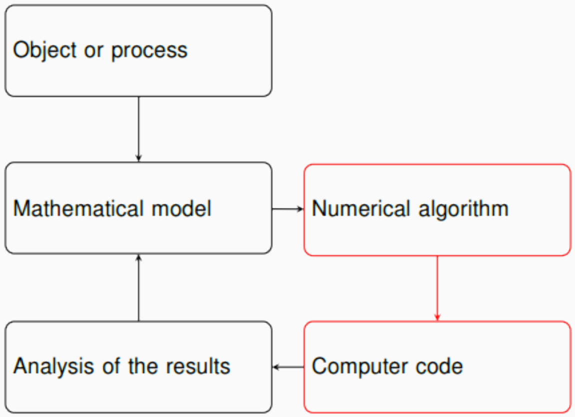




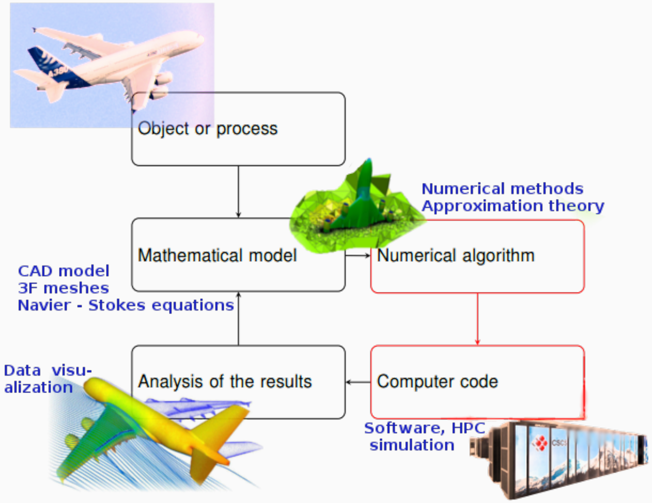




Motivation: mathematical modeling



Motivation: mathematical modeling

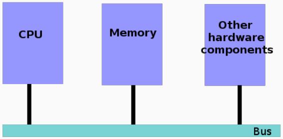


Tool: computers

Computer structure

- **Memory (ROM):** Stores data and instructions
- **CPU:** Elaborates instructions in a machine code
- **Bus:** Data communication system between components

One can use a computer with an operating system and software



Development

Software (including numerical solvers) are developed via **programming languages**, than can be of high or low level

Tool: programming languages

Low-level languages are non-portable, *i.e.* they are specific to each processor architecture, and have very few/low abstraction

- Machine code
- Assembly language

High-level languages are characterised by a strong abstraction

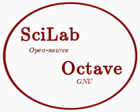
- Compiled: C, C++, C# , Fortran, Delphi, Rust
- Bytecode compiled: Java, Python, Common Lisp
- Interpreted: JavaScript, Mathematica, Python, Matlab®
- Source-to-source translated: Haxe, Dart, TypeScript

About MatLab®

Matlab®, from *MATrix LABoratory*, is a proprietary **software** containing ready-to-use **Toolboxes**. It is commonly used for

- Scientific computing
- Image processing
- Data analysis and post processing
- Symbolic computations and exact arithmetic,

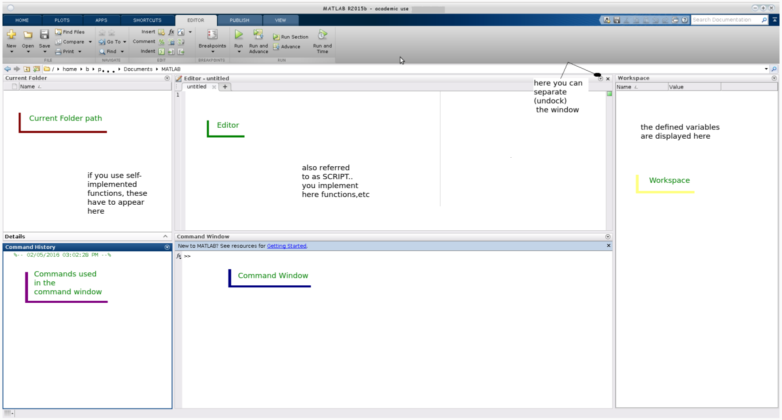
  though Mathematica is preferable for this...



It originates from a FORTRAN program and is notably good with

- Data elaboration with floating points
- Matrix based computations

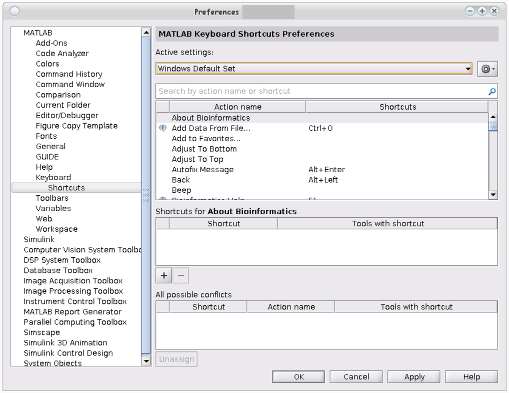
The Matlab® interface


MATLAB R2015b - academic use
HOME
PLOTS
APPS
SHORTCUTS
EDITOR
PUBLISH
VIEW
Current Folder
Current Folder path
if you use self-implemented functions, these have to appear here
Editor - untitled
Editor
also referred to as SCRIPT.. you implement here functions,etc
here you can separate (undock) the window
Workspace
the defined variables are displayed here
Workspace
Details
Command History
Commands used in the command window
Command Window
New to MATLAB? See resources for Getting Started.
Command Window

The Matlab® shortcuts

Default Matlab® shortcuts are fairly different from the usual ones

Preferences → *Matlab* → *Keyboard* → *Shortcuts*



Preferences

- MATLAB
  - Add-Ons
  - Code Analyzer
  - Colors
  - Command History
  - Command Window
  - Comparison
  - Current Folder
  - Editor/Debugger
  - Figure Copy Template
  - Fonts
  - General
  - GUIDE
  - Help
  - Keyboard
    - Shortcuts
  - Toolbars
  - Variables
  - Web
  - Workspace
- Simulink
- Computer Vision System Toolbc
- DSP System Toolbox
- Database Toolbox
- Image Acquisition Toolbox
- Image Processing Toolbox
- Instrument Control Toolbox
- MATLAB Report Generator
- Parallel Computing Toolbox
- Simscape
- Simulink 3D Animation
- Simulink Control Design
- System Objects

MATLAB Keyboard Shortcuts Preferences

Active settings: Windows Default Set

Search by action name or shortcut

Action name	Shortcuts
About Bioinformatics	
Add Data From File...	Ctrl+O
Add to Favorites...	
Adjust To Bottom	
Adjust To Top	
Autofix Message	Alt+Enter
Back	Alt+Left
Beep	

Shortcuts for **About Bioinformatics**

Shortcut	Tools with shortcut

All possible conflicts

Shortcut	Action name	Tools with shortcut

The Matlab® precision

Real numbers representation at the computer level

Matlab uses *floating-point numbers* $\mathbb{F}$, which is different from $\mathbb{R}$

```
>> 1/7
ans =
      0.1429
```

Matlab® computes on floats, in single or double precision

Single precision:

`1/3 ≈ 0.33333333`

Double precision:

`1/3 ≈ 0.3333333333333333`

Note that the numbers of decimals is not the same as the one prompted above: **precision** is different from **output format**

The Matlab® precision

Output formats

One can prescribe the output format that Matlab® should use for **showing** the results by using the command `format Type`

```
>> 2.105
ans =
    2.1050
>> format long
>> 2.105
ans =
    2.105000000000000
```

Type	
rat	1/7
short	0.1429
short g	0.14286
short e	1.4286e-01
long	0.142857142857143
long g	0.142857142857143
long e	1.428571428571428e-01

The Matlab® precision

Output formats

The choice of the output format **does not** impact the computer precision, which is either simple or double (float)

```
>> format short
>> 2.01
ans =
    2.0100
>> 2.01 + 0.000000002
ans =
    2.0100
>> format long
>> ans
ans =
    2.010000002000000
```

Type	
`rat`	1/7
`short`	0.1429
`short g`	0.14286
`short e`	1.4286e-01
`long`	0.142857142857143
`long g`	0.142857142857143
`long e`	1.428571428571428e-01

Theory: representation of floating-point numbers

How to represent any decimal number on the computer regardless its order of magnitude?

Theory: representation of floating-point numbers

How to represent any decimal number on the computer regardless its order of magnitude?

$\rightarrow$ Using a floating-point representation

In base 2, only 23 or 52 (single or double precision) digits can be stored. Thus, one *shifts* the decimal point according to the order of magnitude.

Needs three digits to be represented accurately

The exponent expresses the position of the coma (floating point)

$$\begin{aligned} 0.145000 &= 0.145 \times 10^{0} \\ 0.000145 &= 0.145 \times 10^{-3} \end{aligned}$$

Both need three digits and an exponent to be represented accurately

Needs six digits to be represented accurately

Assuming one can only store 4 digits, one can represent exactly 0.000001 but not 0.0010001. This is due to **machine precision**.

Theory: representation of floating-point numbers

Floating-point representation in any base

In practice, a computer represents any (signed) floating-point number x on a basis β (usually 2) depending on its architecture

$$\begin{aligned} x &= 0.\underbrace{00\cdots\ 0}_{e \text{ times}}\ \underbrace{a_1a_2a_3\cdots a_t}_{t \text{ significant digits}} \quad \text{in the basis } \beta \\ &= (-1)^s \times a_1a_2a_3\cdots a_t \times \beta^{e-t} \\ &= (-1)^s \sum_{j=1}^{t} a_j \beta^{e-j} \end{aligned}$$

Vocabulary

- The p first *significant digits* are $a_1a_2, \cdots a_p$, where $a_1 \neq 0$
- The integer $m = a_1a_2\ldots a_t$ composed of all the t significant digits of x is the *mantissa*

Further reference: Quarteroni, Saleri, Gervasio - Scientific Computing with Matlab and Octave, IVth ed.

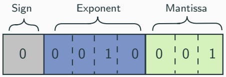
Theory: representation of floating-point numbers

At the computer level

A computer stores in memory any floating point number

$$x = (-1)^s \times a_1 a_2 a_3 \cdots a_t \times \beta^{e-t}$$

by assigning the sign s, each coefficient a_i, and the exponent to a contiguous allocation in memory.



Exemple of the representation of $0.25 = 1 \times 2^{-2}$ on 8 bits in base 2

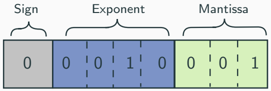
No. of Bits	Sign Bits	Exponent Bits	Mantissa Bits
8	1	4	3
16	1	6	9
32	1	8	23
64	1	11	52
128	1	15	112

The number of allocated storage spots per block type depends on the computer's architecture

Theory: representation of floating-point numbers

Range of representable values

As the number of significant digits, the exponents that can be used in are limited by the number of bits of its storage



Exemple of the representation of $0.25 = 1 \times 2^{-2}$ on 8 bits in base 2

Practically, the exponent e can take any integer value in $\{L, \cdots, U\}$, where L (*resp.* U) is the smaller (resp. bigger) number that can be expressed in base 2 on the number of allocated exponent's bits

Range of representable values: $x_{min} = \beta^{L-1}$

$$x_{max} = \beta^{U}(1 - \beta^{-t})$$

Theory: representation of floating-point numbers

Exercise

Knowing that a floating-point number is represented as

$$x = (-1)^s \times a_1 a_2 a_3 \cdots a_t \times \beta^{e-t}$$
$$= (-1)^s \sum_{j=1}^{t} a_j \beta^{e-j}_{,}$$

and assuming that $\beta = 2$, $t = 3$ and $e \in \{-4, \ldots, 3\}$, derive:

- the smallest representable number:
- the biggest representable number:
- the smallest representable absolute value of a number:
- the number of bits you need to store one number:

Theory: representation of floating-point numbers

Exercise

Knowing that a floating-point number is represented as

$$x = (-1)^s \times a_1 a_2 a_3 \cdots a_t \times \beta^{e-t}$$
$$= (-1)^s \sum_{j=1}^{t} a_j \beta^{e-j}_{,}$$

and assuming that $\beta = 2$, $t = 3$ and $e \in \{-4, \ldots, 3\}$, derive:

- the smallest representable number:
  -7
- the biggest representable number:
  7
- the smallest representable absolute value of a number:
  $\frac{1}{32}$
- the number of bits you need to store one number:
  $s : 1, a_i : t-1 = 2, e : 3 \Rightarrow 6$ *bits*

Storing in Matlab

Space of floating-point numbers

The set of a floating-point numbers $\mathbb{F}$ that one can represent on a computer is characterised by $\mathbb{F}(\beta, t, L, U)$,

where β is the basis, t the number of significant digits, and L, U are integers giving the range's bounds of the used exponents

Space used in Matlab®

Numbers are represented on $\mathbb{F} = \mathbb{F}(2, 53, -1021, 1024)$ and are stored on 8 bytes ($=$ 64 bits, exponent on 11 bits: *double precision*)

```
>> realmin
ans =
    2.2250738585072e-308
```

```
>> realmax
ans =
    1.79769313486232e+308
```

Storing in Matlab

Exercise

Represent the following floating points in base 2, or explain why this is not possible to be done exactly.

a) 15

b) 0.5

c) 0.1

d) 1/3

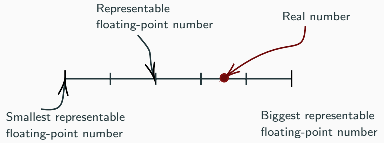
e) 1/256

Theory: round-off error

Definition

A *round-off* error is generated when a real number is replaced with a floating-point number

$$\frac{|x - fl(x)|}{|x|} \leq \frac{1}{2}\,\epsilon_M$$



with $\epsilon_M = \beta^{1-t}$ the *machine epsilon*

In Matlab®

The machine epsilon is given by $\epsilon_M = 2^{-52} \sim 2.22 \cdot 10^{-16}$

```
>> eps
ans =
    2.22044604925031e-16
```

Good to know

Sources of numerical errors

1. Fixed-size data storage format on computers

   (e.g. round-off errors)
2. Errors in the initial or input data

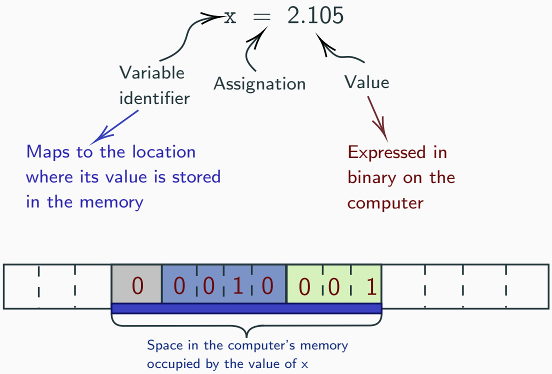
   (e.g. from experimental measurements)
3. Incompleteness of mathematical model
4. Approximate methods to solve the equations of the mathematical model
5. . . .

Getting started

Scalar assignment

Reusing values

The values that will be reused are stored in *variables*



x = 2.105
Variable identifier
Assignation
Value
Maps to the location where its value is stored in the memory
Expressed in binary on the computer
0
0 0 1 0
0 0 1
Space in the computer's memory occupied by the value of x

Here, `x` is a variable, and we *assigned* the *value* 2.105 to it

Scalar assignment

Type in the command line:

```
>> a = 2.45
>> A = 3.1
>> a
```

Note: >> is called a *prompt*

Scalar assignment

Type in the command line:

```
>> a = 2.45
>> A = 3.1
>> a
```

```
ans =
      2.4500
```

→ Distinction between capital and small letters (case sensitivity)

Note: >> is called a *prompt*

Scalar assignment

Type in the command line:

```
>> A = 7.2
>> A
```

Scalar assignment

Type in the command line:

```
>> A = 7.2
>> A
```

```
ans =
      7.2000
```

→ Watch out! The variable *A* has been *overwritten* (the previous value 3.1 has been replaced by 7.2)

Comma and semicolon

Use of the comma

```
>> a= 1.2,
a =
    1.2000
>> a= 1.7, b=2.45
a =
    1.7000
b =
    2.4500
```

- Useful to **separate multiple commands** on a same line
- No difference with/without the comma at the end of the line

Comma and semicolon

Use of the semicolon

```
>> a= 1.2;
>> a= 1.7; b=2.45
b =
    2.4500
>> a;
>> a
a =
    1.7000
```

- Useful to **separate multiple commands** on a same line
- Suppresses the **output visualisation**

Memory management

Exploring memory

To check what variables are stored in the memory:

```
>> who
Your variables are:
a  A  ans
```

To know the stored variables, their dimension, the dimension of their memory storage and their typology:

```
>> whos
Name      Size            Bytes  Class     Attributes

A         1x1                 8  double
a         1x1                 8  double
ans       1x1                 8  double
```

Memory management

Clearing memory

Delete variable *A* from workspace:

```
>> clear A
```

Delete **all variables** from workspace:

```
>> clear
>> clear all
```

Clean up the visualization window:

```
>> clc
>> home
```

Best practice

Always clean your workspace and visualization window before starting a new exercise

Getting help about instructions

To see how to use an instruction, ex. called `instruction`:

```
help instruction
```

Example:

```
>> help format
format Set output format.
    format with no inputs sets the output format to the
    default appropriate for the class of the variable.
    For float variables, the default is format SHORT.

    format does not affect how MATLAB computations
    are done. Computations on float variables, ...
```

Getting started: trying the command line

Exercise

Type in the command line what follows and see what happens

```
>> doc sin
>> sin <F1>
>> si <TAB>
```

Exercise

Type in the command line what follows and explain the results

```
>> 32 * 3 + 5
>> 32 * (3 + 5)
>> sin (3 * pi)
>> pi
```

Expressions and predefined variables

Expressions

Arithmetic operators

Arithmetical operators	`^ * /`	power, multiplication, division
	`+ -`	addition, subtraction
Equivalence operators	`== ~=`	equivalent to, unequal to
	`< <= > >=`	less, greater than
Logic operators	`& | ~`	and, or, not

The order in which the operations are done is called *precedence*, whose rule is given by the previous table (decreasing order)

- Two operations with a same priority rank are done from left to right
- One can specify the desired precedence by using brackets ()

```
>> 32 * 3 + 5
>> 32/2*3
>> 32 * (3 + 5)
```

To get more information: `doc 'operator precedence'`

Expressions

Numerical expressions

Expressing integer and decimal numbers

```
>> 1
```

```
>> 0.23
```

```
>> .23
```

Expressing decimal numbers in their exponential form

```
>> 23*10^(-2)
```

```
>> 23e-2
```

Expressing complex numbers

```
>> 5+4i
```

```
>> 5+4j
```

Expressions

Built-in functions

Predefined functions exist in Matlab®

`sin`	`sqrt`	`log`	`floor`
`cos`	`abs`	`log2`	`ceil`
`tan`	`exp`	`log10`	`round`

Call them with `functionName(parameter1, parameter2, ...)` and know their arguments' type by typing `help functionName`

Example:

```
>> help sin    % You learn that the angle is in radians
>> sin(3 * pi)
```

Expressions

Predefined variables

Some commonly used values are stored in *predefined variables*

`ans`	result of the last computation
`pi`	$\pi = 3.14..$
`eps`	machine precision
`i,j`	$\sqrt{-1}$
`nan`	not a number (ex. as result of 0/0)
`inf`	infinity (ex. as result of 1/0)
`realmin, realmax`	smallest and biggest floating-point numbers

Best practice

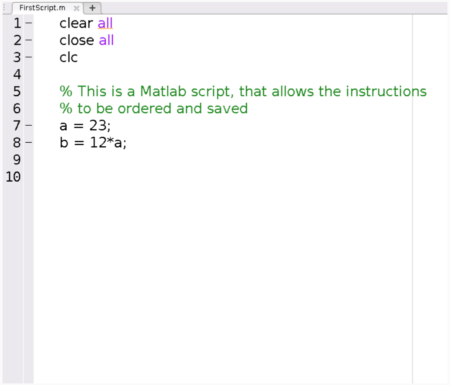
Do not overwrite any of the predefined functions or variables!

Further *keywords* should not be used as variable names. The list of keywords is accessible with the command `>> iskeyword`

Scripts

Scripts

In order to organise a succession of instructions and reuse a previous work, one uses *scripts*



```
clear all
close all
clc

% This is a Matlab script, that allows the instructions
% to be ordered and saved
a = 23;
b = 12*a;
```

A Matlab® script is a file with a *.m* extension that contains all your ordered instructions

Scripts

Create your first script:

1. Click on New $\rightarrow$ Script
2. Write within the file

```
% This is my first script
a = 23;
b = 12*a;
```

3. Save the script with the name *"1_FirstScript.m"*

   What is happening?

Scripts

Create your first script:

1. Click on New → Script
2. Write within the file

```
% This is my first script
a = 23;
b = 12*a;
```

3. Save the script with the name *"1_FirstScript.m"*

   What is happening?

→ A script's name should always begin with a letter

→ The extension should also always be ".m", otherwise Maltab® won't recognise it

Scripts

Run your first script:

4. Save the script with the name *"L1_FirstScript.m"*
5. Run the script by clicking on the run button
6. Look at the variables present in the workspace, and check the output of `whos`

Scripts

Run your first script:

4. Save the script with the name *"L1_FirstScript.m"*
5. Run the script by clicking on the run button
6. Look at the variables present in the workspace, and check the output of `whos`

→ Nothing is prompted during the run (use of semicolons)

→ The two variables `a` and `b` appear in the workspace and have the desired values

Scripts

Create and call your first script from the prompt:

7. Run the script from command line by typing `L1_FirstScript`
8. Type `L1_Fi` and press TAB
9. Close L1_FirstScript by clicking on the cross on the top right of the built-in text editor
10. Type `edit L1_FirstScript` in the prompt
11. Type `edit L1_SecondScript` in the prompt

Scripts

Create and call your first script from the prompt:

7. Run the script from command line by typing `L1_FirstScript`
8. Type `L1_Fi` and press TAB
9. Close L1_FirstScript by clicking on the cross on the top right of the built-in text editor
10. Type `edit L1_FirstScript` in the prompt
11. Type `edit L1_SecondScript` in the prompt

→ Tabbing after having written the first letters of a script's name in the prompt *autofills* its name

→ The command `edit` opens or creates a new script, automatically with a ".m" extension

Comments on the top of the file

Always give indications on how to use your code to any potential reader (usually yourself)

- Summarises the purpose of the code
- Specifies the variables taken in input and their format
- Specifies the variables given in output and their format
- Further useful information for using the script

When written at the very top of the file, those comments are accessible externally through the commands

```
>> help L1_FirstScript
>> L1_FirstScript + F1
```

Documentation

Comments through the file

Give indications on how you though your code as you are writing it, in particular in technical areas where the code understanding is not straightforward

- Eases the understanding of the code for an external reader
- Eases the maintenance of the code

Types of comments

```
% single line comment

%% Describes a code block
```

```
%{
multi line
comment
%}
```

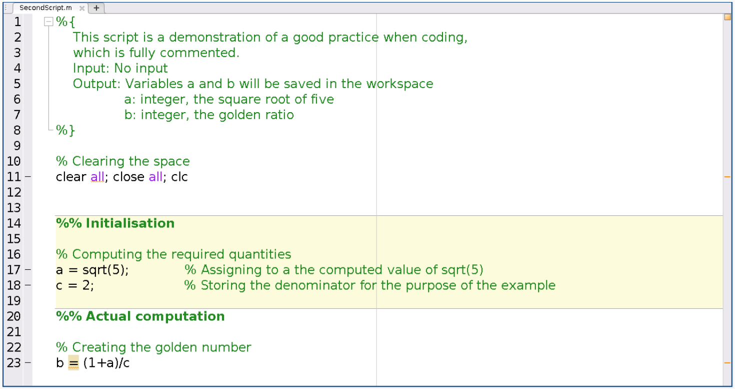
Scripts and documentation

Best practice

- Always start a script with a comment block describing its usage (documentation)
- Then add `clear all; close all; clc;` to clear the prompt and previous workspace variables
- Separate the parts of your code having distinct aims with a block separator `%% A comment` that describes the main goal of the block below the separator
- Comment the technical parts thorough your code
- Write in a legible way and use coherent indentation

Think your code to be efficient and write with a Matlab® spirit

A dream code



```
%{
    This script is a demonstration of a good practice when coding,
    which is fully commented.
    Input: No input
    Output: Variables a and b will be saved in the workspace
                a: integer, the square root of five
                b: integer, the golden ratio
%}

% Clearing the space
clear all; close all; clc


%% Initialisation

% Computing the required quantities
a = sqrt(5);            % Assigning to a the computed value of sqrt(5)
c = 2;                  % Storing the denominator for the purpose of the example

%% Actual computation

% Creating the golden number
b = (1+a)/c
```

Note that Matlab® helps you with *syntax highlighting*

Functions

Functions

Create your own functions

A *function* is a group of instructions that executes a given task

- it takes none, one or many *arguments* (input variables)
- it performs the tasks according to the instructions
- it returns none, one or many *output variables* to the script or prompt that calls it

In Matlab®, the function should be named as the containing file, and its structure is as follows

```
function [output1, output2] = TheFunction(arg1, arg2)
    % The help of the function
    The main content of the function
end
```

Functions

Create your first function:

1. Click on New → Function
2. Change the content to:

```
function y = L1_power8(x)
    % Compute y=x^8 (comment displayed in the help)
    y=x^8;
end
```

3. Save it to `L1_power8.m`
4. Type in command line `L1_power8(2)`

Functions

Create your first function:

1. Click on New → Function
2. Change the content to:

```
function y = L1_power8(x)
    % Compute y=x^8 (comment displayed in the help)
    y=x^8;
end
```

3. Save it to `L1_power8.m`
4. Type in command line `L1_power8(2)`

→ The instructions within the function are executed on the value $x = 2$ and the result is printed in the prompt

→ No variable `y` in the workspace, but `ans` contains 256

Functions

Create your first function with multiple arguments:

1. Create a function `L1_ThreeArguments` as follows

```
function [y1, y2] = L1_ThreeArguments(x1,x2,x3)
    % example function with more than one input
    y1=x1^2;
    y2=x2+x3;
end
```

2. Save it to `L1_ThreeArguments.m`
3. Call the function from the prompt with `L1_ThreeArguments(2,3,4)`. What does `ans` value?

Functions

Create your first function with multiple arguments:

1. Create a function `L1_ThreeArguments` as follows

```
function [y1, y2] = L1_ThreeArguments(x1,x2,x3)
    % example function with more than one input
    y1=x1^2;
    y2=x2+x3;
end
```

2. Save it to `L1_ThreeArguments.m`
3. Call the function from the prompt with `L1_ThreeArguments(2,3,4)`. What does `ans` value?

→ The variable ans only contains the returned value for `y1`, the return value for `y2` is not automatically given

Functions

Call your first function with multiple arguments:

4. Store the output of the function by

```
[a, b] = L1_ThreeArguments(2,3,4)
```

5. Observe the values of `a` and `b` if you simply write

```
a, b = L1_ThreeArguments(2,3,4)
```

6. Retrieve only some outputs among all of those returned

```
a = L1_ThreeArguments(2,3,4)
[a, ~] = L1_ThreeArguments(2,3,4)
[~, b] = L1_ThreeArguments(2,3,4)
```

Functions

Call your first function with multiple arguments:

4. Store the output of the function by

```
[a, b] = L1_ThreeArguments(2,3,4)
```

5. Observe the values of `a` and `b` if you simply write

```
a, b = L1_ThreeArguments(2,3,4)
```

6. Retrieve only some outputs among all of those returned

```
a = L1_ThreeArguments(2,3,4)
```

```
[a, ~] = L1_ThreeArguments(2,3,4)
```

```
[~, b] = L1_ThreeArguments(2,3,4)
```

→ Warning: using `a, b = L1_ThreeArguments(2,3,4)` assigns the first returned value (`y1`) to both `a` and `b`

Functions

Accessing the number of input and output variables

Within a function, the number of input and output arguments specified in its declaration are accessible with the keywords

`nargin`
`nargout`

From outside the function, use the command `nargin(@function)`

```
>> nargin(@sin)
>> nargin(@L1_ThreeArguments)
>> nargout(@L1_ThreeArguments)
```

Note: Functions may be called with less inputs, but not with more

Functions

Multiples functions in a file

Defining multiple functions in a single file is possible. However, only the function whose name is matching the file name can be called from the command line or from other scripts

Example: *Content of the file "L1_power8_2.m"*

```
function y = L1_power8_2(x)
    y=mypower8(x);
end

function y = mypower8(x)
    y=x^8;
end
```

The prompt can only execute `L1_power8_2(2)`, not `mypower8(2)`

Scoping and local variables

The interface between the function and the workspace (or script) is done only through the input and output arguments

```
function y = power8_local(x)
    % Power eight function with a local variable
    y=x^8      % Is sent back to the workspace
    myVar = 5; % Does not appear in the workspace
end
```

Any variable that does not appear in the function declaration is a *local variable* and stays within the *scope* of the function

→ Local variables can be named the same in different functions

Functions

Global variables

When a variable needs to be shared between different scripts, functions, workspace scopes, one can declare a global variable by

`global global_var`

It can then be called everywhere, specifying its global nature

```
function y = power8_global(x)
    % Power eight function with a global variable
    y=x^8;           % Is sent back to the workspace

    global my_global_var % Generates a global variable
    my_global_var=2*y;   % appearing in the workspace
end
```

Note: Using global variables is usually a *bad* practice: prefer arguments

Functions

Anonymous functions

Whenever the definition of the function is simple (typically when it takes a single line), it is convenient to use *Anonymous functions*

```
>> f = @(x,y) x^y
>> f(2,3)
```

- The right hand side is an anonymous function
- The variable `f` is a *function handle*

Note: To witness the type of the variable `f`, use `class(f)`

Remark: The old *inline* syntax (e.g. `g = inline('sin(2*pi*f + theta)', 'f', 'theta')`) is depreciated and will be cleared

Functions

Closures

Let an anonymous function be defined from a parameter a. Then, if this parameter changes later in the code, the function does not automatically change. This is due to *closures*

```
>> clear all;
>> a=2;
>> f = @(x) x+a;
>> f(2) % Gives out 4
>> a=3;
>> f(2) % Still gives out 4
```

Functions

Best practice

- Always write documentation for your functions
- Choose the most efficient and straightforward way to perform a task: do not reinvent the wheel
- Test your functions by themselves as you write it
- Use the minimum (or none) global variables
- Use variable names that are meaningful and coherent with each other (even for local variables)
- Indent carefully the code blocks

Tips and tricks

Matlab® hacks

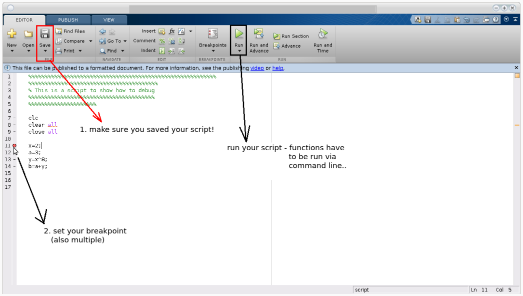
- The keyword `return` leaves a script or function
- Three dots **...** splits an equation into multiple lines

```
function y = power8(x)
   y=... % very long expression over multiple lines
     x^8;
end
```

User interaction

- The instruction `disp(x)` displays the value of a variable or expression of `x` in the prompt
- The instruction `error('my message')` raises an error message to the user and exits the function or script
- The keyword `pause` suspends the run until a key is pressed

Debugging

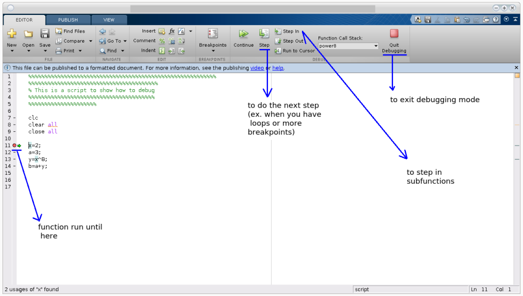


```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This is a script to show how to debug
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%

clc
clear all
close all

x=2;
a=3;
y=x^8;
b=a+y;
```

1. make sure you saved your script!

run your script - functions have to be run via command line..

2. set your breakpoint (also multiple)



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This is a script to show how to debug
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%

clc
clear all
close all

x=2;
a=3;
y=x^8;
b=a+y;
```

to do the next step (ex. when you have loops or more breakpoints)

to exit debugging mode

to step in subfunctions

function run until here

Exercises

Exercises: Command Line

Exercise

1. Compute the number of hours and seconds of a year
2. Use the help tool of Matlab to display the square root (`sqrt`)
3. Find the function for computing n-roots and compute $\sqrt[3]{10}$
4. Compute the multiplicative inverse of $1 + 2i$

Exercises: Basic operations

Exercise

5. Overwrite the variables `i,j` and observe the output of

```
>> 3i
>> 3*i
```

6. Determine for which number of the following expressions there exist $(a_1, a_2, a_3) \in \mathbb{Z}^3$ such that

$$(a_1 \text{ op}_1\ a_2) \text{ op}_2\ a_3 \neq a_1 \text{ op}_1\ (a_2 \text{ op}_2\ a_3)$$

   a) $(\text{op}_1, \text{op}_2) = (-, -)$
   b) $(\text{op}_1, \text{op}_2) = (-, +)$
   c) $(\text{op}_1, \text{op}_2) = (+, \&)$
   d) $(\text{op}_1, \text{op}_2) = (\&, |)$
   e) $(\text{op}_1, \text{op}_2) = (*, ==)$
   f) $(\text{op}_1, \text{op}_2) = (\&, >=)$

Exercises: Basic operations

Exercise

7. Add as many (simple) **brackets** as possible, without changing the meaning of the expression. Then, type the initial expression and your final suggestion in Matlab.

   a) `2 + 4 * - 1 / 3 / 4`
   b) `2 + 4 == 5 | 3 + 5 ^ - 1 & 2`
   c) `- 3 < 7 < 5`
   d) `sin ( n + 1 ) / ( n + 2 )`

Exercises: Basic operations

Exercise

8. Which value has `ans` after each of the following commands?

```
>> 23;
>> ans^2;
>> x = sqrt(ans);
>> x^2 - ans;
>> x^2 - ans;
>> ans - x^2;
```

9. Think about how complex numbers could be rewritten from polar coordinates (angle in radian and radius) to Cartesian coordinates (real and imaginary part). Then, take radius `r=0.2` and the angle `p=1.32` and write the corresponding complex number.

Exercises: Arguments in functions

Exercise

10. Try to implement the following function:

```
function y = noinput()
   x=2;
   z=3;
   y=x+z;
end
```

Exercises: Functions

Exercise

11. Write a function, that gets as input the integer a and b and gives as output both the **integer division** a/b and **remainder**.
12. Write a function `curry(f,g,x)`, which takes as input two anonymous functions and a parameter and gives as output $f(g(x))$.

Control flow

Control flow

A *control flow* is the order in which individual statements, instructions or function calls of the code are executed

To control which statements should be executed, and in which order, *control structures* are predefined in keywords

The main control structures are the structures of

- Conditions (if then else)
- Loops (count-controlled, condition-controlled, . . .)

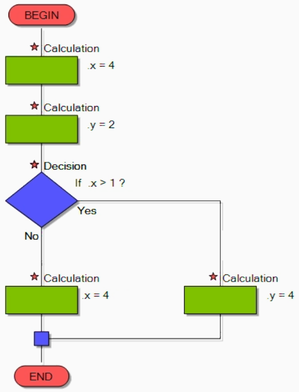


Figure 1: Chart of a control flow

Condition statements

Single condition

The instructions given within a if-statement *block* are only considered if the statement is fulfilled

```
if (condition statement which outputs a boolean)
    (matLab commands)
end
```

Otherwise, the instruction block is simply disregarded

Example:

```
>> x = 2
>> if x == 5  % The block will be executed only if x=5
>>   y=1;     % otherwise the instructions are ignored
>> end
```

Condition statements

Condition from multiple sub-statements

Multiple conditions can be expressed in a statement giving a single boolean output, formed by several sub-statements connected through the operators `|`, `&` and `~`

```
>> x=-3; y=5;
>> if ~(x == 5 & y >= 2) | x<-2  % Executed only if
      y=1;                       % not (x=5 and y>=2)
   end                           % or if x<-2
```

- Know your logical equivalences to simplify the test statement
- Watch out the precedence order of the sub-statements
- The operators `||` and `&&` are the short-cutting ones

Condition statements

Nested conditions

Conditions can be *nested*, meaning that several condition blocks may be stacked one under another

```
>> if x >= 5        % Executed only if x>=5
       y = x+2
       if y == 9;   % Executed only if y=9
          x = 2
       end
   end
```

Note: If the condition statement ruling the nested sub-block does not depend on the instructions of the outer block, it is usually preferable to use a single line multiple condition

Condition statements

Default case

One can specify a *default* case to execute instructions in case the wished statement is not fulfilled, with the keyword `else`

```
>> if x == 5   % Executed if x=5
      y=1;
   else        % Executed in all the
      y=3;     % other cases
   end
```

Multiple conditions (1/2)

A sequence of instructions blocks that should be executed upon ordered tests statements is defined using `if`, `elseif` and `else`

```
>> if x == 5   % Executed if x=5
       y=1;
   elseif x==6 % If the previous block was not entered,
       z=2;    % enter this one if x=5
   else
       y=3;    % Enter here if none of the above case
   end         % was successful
```

- The `else` statement is not mandatory
- Watch out blocks that are never considered

```
>> x=6; if x>=5; y=1; elseif x==6; y=0; end
```

Multiple conditions (2/2)

If the multiple conditions are such that

- they act on the same variable
- the tests' natures are the same, focusing on the output's value

a convenient way to write down the conditions is to use the keyword `switch`. The runtime execution will also be faster

```
>> if x+2==1
>>    y = 2
>> elseif (x^2==2)
>>    y = x+2+4
>> else
>>    error("Oops")
>> end
```

```
>> switch(x+2)
>>    case 1
>>        y = 2
>>    case 2
>>        y = x+2+4
>>    otherwise
>>        error("Oops")
>> end
```

Condition statements

Statements involving predefined variables

When the obtained results are close to machine precision or do not have a floating point representation on 8 bytes, one can test them against predefined variables

```
>> eps
>> 1+eps == 1
>> 1+3*eps/4 == 1
>> 1+eps/2 == 1
>> 1-eps/4 == 1
```

```
>> nan == Inf
>> 1/0 == NaN
>> 10^308 == Inf
>> 10^309 == Inf
```

Mathematical specificities or errors are also handled similarly

```
>> 0 == -0
```

```
>> 1/0
```

```
>> 0/0
```

Condition statements

Statements involving predefined variables

When the obtained results are close to machine precision or do not have a floating point representation on 8 bytes, one can test them against predefined variables

```
>> eps
>> 1+eps == 1
>> 1+3*eps/4 == 1
>> 1+eps/2 == 1
>> 1-eps/4 == 1
```

```
>> nan == Inf
>> 1/0 == NaN
>> 10^308 == Inf
>> 10^309 == Inf
```

→ It is linked to `realmin`

Mathematical specificities or errors are also handled similarly

```
>> 0 == -0
```

```
>> 1/0
```

```
>> 0/0
```

Loops

Loops are instructions that repeat a same block of instruction upon an evolving variable. Each step of the loop is called an *iteration*

```
TypeOfLoop (IterationControl)
     Block of instructions that should be
     repeated, possibly depending on the
     iterated variable’s value
EndOfLoop
```

- Useful to carry out the same command multiple times
- Possible to create nested loops (though usually not advised)
- Different types of control on the iterated variable depending on the aim of the loop: `for` and `while` loops

Loops

For loops

A *for* loop is a loop that iterates a predefined number of times. The iteration is controlled by the definition of a given list of iterates (in a vector format)

```
for iterate=start:end
   (Instructions)
end
```

```
>> for k=1:10
      disp(k^2)
   end
```

```
>> for k=[4.0,2.1]
      disp(k^2)
   end
```

```
>> for k=1
      disp(k^2)
   end
```

Loops

For loops

A *for* loop is a loop that iterates a predefined number of times. The iteration is controlled by the definition of a given list of iterates (in a vector format)

```
for iterate=start:end
   (Instructions)
end
```

```
>> for k=1:10
      disp(k^2)
   end
```

```
>> for k=[4.0,2.1]
      disp(k^2)
   end
```

```
>> for k=1
      disp(k^2)
   end
```

$\rightarrow$ 1

Loops

While loops

A *while* loop executes the code's block until a specified condition is fulfilled. Only the change in the values of the variables used within the loop can be used to define a *stopping criterion*

```
>> a=0;
>> while a<5
     a=a+1;
   end
```

```
>> a=6;
>> while a>5
     a=a+1;
   end
```

```
>> a=0;b=1;
>> while a<b
     a=a+1;
     b=0.5*a;
   end
```

Note:

- The variables involved in the stopping criterion should be *initialized* before the loop
- `ctrl + c` interrupts a script

Loops

Control keywords

On the top of the iteration instruction that defines the loop, it is possible to control the loop flow from the instruction block itself by the keywords

`break`: interrupts the iteration and jumps to below the loop

`continue`: jumps to the instruction block's star. In a for-loop, the iterated value is updated to the next one

```
>> for k=1:10
     disp(k);
     continue; % Skips below
     a=1/0;    % Not done
   end
```

```
>> k=0;
>> while k<10
     disp(k);
     break; % Breaks
     a=1/0; % Not done
   end
```

Variables scope

In Matlab®, the variables defined or updated inside a condition statement or a loop are accessible from outside the code's block

```
>> clear x
>> for k=1:3
       x=2*k;
   end

>> disp(x)
>> disp(k)
```

The value retrieved outside the loop corresponds to:

- the last value assigned within the loop
- the last value of the iterated variable

Control flow

Best practice

- *Indent* the code's blocks that are subject to conditions or loops, and keep the *indentation level* consistent

- Always write a safety condition that stops a while loop
- Try not to call the iterated variable(s) as i, j
- Avoid nested loops

Exercises

Exercises: Condition statements & Loops

Exercise

1. Write a script that contains each of the keywords
    - if
    - for
    - while
    - continue
    - break
    - a call to a self implemented function

   and check its right execution through the debugger.

Exercises: Condition statements & Loops

Exercise

2. Write a function that takes a positive integer n and computes the members of the Fibonacci sequence $1, 1, 2, 3, 5, 8, ..$
   a) once without using loops as a recursive function $f(n) = f(n-1) + f(n-2) \quad \forall n > 2, \quad f(1) = f(2) = 1.$
   b) once using loops

   Do not forget to write a documentation for the implemented functions

Exercises: Condition statements & Loops

Exercise

3. Check numerically that:

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

   a) Write a function that takes a continuous function f and two values (boundary) a, b, such that $f(a) \cdot f(b) < 0$. The function will return the x for which $f(x) = 0$. This is done through the bisection method on the interval $[a, b]$.
   b) Check the function on `sin` to compute `pi`.

Exercises: Condition statements & Loops

Exercise

4. We are interested in finding numerically the zeros of a function.
   a) Write a function, that takes as input a function f, its derivative f' and an initial value x_0. This function will give as output a zero of f found with the Newton method, that reads
   $$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$
   Be careful, $f'(x) \neq 0$, so, choose properly x_0.
   b) Check your implemented function using the function `sin` to compute `pi`.

Matrix Calculus

Matrix Calculus

Matrix calculus is the field dealing with multivariate calculus over spaces of matrices and vectors

Matlab® is optimised for those computations at the numerical level: whenever possible one should use *vectorized computations*

- Eases the lecture and understanding of the code's
- Increase in the speed performance
- Consistency with Matlab®'s spirit

Arrays

Arrays are the core of Matlab®. As Matlab® states it itself,

> While other programming languages mostly work with numbers one at a time, Matlab® is designed to operate primarily on whole matrices and arrays.

All variables are multidimensional arrays, regardless the type of data. There are two main categories

- Numerical arrays

  Scalar (1×1)

  Vector $(n \times 1$ or $1 \times n)$

  Matrix $(n \times m)$
- Cell array of objects

Vector arrays

Simple creation of a vector array

1. Try the following in the command line:

```
>> a=[8 2 23 5]
>> b=[8, 2, 23, 5]
>> c=[8; 2; 23; 5]
```

2. Check the workspace and look at the information on a, b, c.
3. What do `space` or `,` or `;` do?

Vector arrays

Simple creation of a vector array

1. Try the following in the command line:

```
>> a=[8 2 23 5]
>> b=[8, 2, 23, 5]
>> c=[8; 2; 23; 5]
```

2. Check the workspace and look at the information on a, b, c.
3. What do `space` or `,` or `;` do?

→ The separators `space` and `,` are building a row vector
→ The separator `;` is building a column vector
→ The storage's amount of a vector corresponds to its number of elements times the amount required by one element

Vector arrays

Automatic creation of a vector array

Matlab® has built-in constructors and functions that *initialises* automatically a vector array defined on a predefined sequence

```
>> a = 1:10
>> a = 1:2:10
>> a = 100:-2:-1
>> a = 1.2:0.2:10.5
```

```
>> b = zeros(1,10)
>> b = ones(10,1)
>> b = linspace(1,10,10)
>> b = linspace(1.2,10.4,47)
```

The *indexing* starts at 1 and finishes at `end`. The elements of a vector array are accessible through their index

```
>> a(1)
```

```
>> a(end)
```

```
>> a(2:end-1)
```

Note: Be careful to the bounds!

```
>> b(45:end-5)
```

```
>> a(-1)
```

Vector arrays

Getting the properties of a vector array

To perform the desired algebra operations it is crucial to know the orientation (row or column) of a vector array and its length

```
>> size(a)
```

```
>> length(a)
```

Operations over a vector array: addition

```
>> a = 1:5;
>> b = 5:3:17;
>> c = a + b
```

The addition and subtraction are done element-wise

```
>> a = 1:5; b = 5:3:20;
>> c = a - b
```

→ The two vectors should have the same length

Vector arrays

Operations over a vector array: transpose

The symbol `'` transposes a given vector: a row vector array will become a column array and *vis-versa*

```
>> a = 1:5;
>> b = a';
>> size(a)
>> size(b)
```

Operations over a vector array: broadcasting

In the new versions of Matlab® (from 2016b on), the operations are *broadcast*: adding two vectors that do not have the same orientation creates a matrix

```
>> b = [3, 2];
>> a = [1; 4];
>> a+b
```

```
ans =
      4     3
      7     6
```

Operations over a vector array: multiplication

The operator `*` is the matrix multiplication. Between row and column vectors, it is the scalar product. The operator `.*` is the element-wise multiplication

```
>> a = 1:5; b = 5:3:17;
>> c = a*b                  % Error
>> c = a.*b                 % Element-wise multiplication
>> c = a*b’                 % Scalar product
>> c = b’*a                 % Broadcast multiplication
```

Note: Be careful to the orientation of the vectors. Because of broadcasting, you may not see the errors

Vector arrays

Operations over a vector array: right division

The operator `/` is the matrix right division: A/B determines the matrix C such that $C * B = A$. It applies to vectors by considering that a vector is a $n \times 1$ or $1 \times n$ matrix. The operator `./` is the element-wise usual division

```
>> a = 1:5; b = 5:3:17;
>> c = a/b          % Least-squares solution of x*a = b
>> c = b'/a'        % Least-squares solution of x*a = b
>> c = a./b         % Element-wise usual division
>> c = a/b'         % Error
>> c = a'/b         % Error
```

Matrices

Simple creation of a matrix

1. Try the following in the command line:

```
>> clear all; clc;
>> a=[1 2 3; 4 5 6]
>> b=[1,2,3; 4,5,6]
```

2. Check the workspace and look at the information on *a*, *b*.
3. What do `space` or `,` or `;` do?

Simple creation of a matrix

1. Try the following in the command line:

```
>> clear all; clc;
>> a=[1 2 3; 4 5 6]
>> b=[1,2,3; 4,5,6]
```

2. Check the workspace and look at the information on *a*, *b*.
3. What do `space` or `,` or `;` do?

→ The separators `space` and `,` are separating the columns
→ The separator `;` is separating the rows
→ The storage's amount of a matrix corresponds to its number of elements times the amount required by one element

Matrices

Automatic creation of a matrix

Matlab® has built-in functions and assembling techniques that *initialises* automatically a matrix defined on a predefined structure

```
>> % Using constructors
>> A = []           % Empty
>> A = eye(5)       % Identity
>> A = zeros(4,3) % Zeros
>> A = ones(4,3)   % Ones
>> A = rand(5)      % Uniform
>> A = randn(3,4) % Normal
>> A = magic(5)
```

```
>> % Assembling
>> A = eye(2)
>> B = [1:5]+[1:3]'
>> C = ones(2,3)
>> D = [A C; B]'
```

```
>> a = 2*1:5+[1:3]'
>> a = 2*[1:5]+[1:3]'
```

Matrices

Knowing the size of a matrix

As for vectors, the (multidimensional) size of a matrix is given by the command `size`. For a given matrix A, the command `length(A)` returns `max(size(A))`

```
>> size(D)
>> size(D,1) % Number of rows
>> size(D,2) % Number of columns
```

```
>> length(D)
>> max(size(D))
```

Reshaping matrices

One can construct a matrix from a single vector or stack a matrix into a vector. **The total number of elements must be preserved**

```
>> D2 = reshape(D,1,(size(D,1)*size(D,2)))
>> a = 1:16
>> A = reshape(a,4,4)  % Fills the columns successively
```

Matrices

Accessing elements

As vector arrays, the *indexing* starts at 1 and finishes at `end`. The matrix's elements are accessible giving one index per dimension

```
>> D(2,3)            % Second line, third column
>> D(1,end)          % Last element of the first line
```

Extracting ang glueing submatrices

Extract a submatrix by giving a list of indices for each dimension

```
>> D(1:3, 1:2:6)
>> D(1:3, [1 4 3])
>> D(1:5)
>> D(1, 1:end)
```

```
>> D(:)
>> D(:,1:3)
>> D(1:3,:)
>> D(:,end)
```

Paste multiple copies of a same matrix with `repmat(D,2,2)`

Matrices

Operations over matrices

Element-wise sum/substraction	`+ -`
Element-wise multiplication/division	`.* .\ ./`
Matrix multiplication (ex. `B*A, 3*B`)	`*`
Element-wise power	`.^`
Power for square matrix	`^`
Conjugate transposed (ex. `A'`)	`'`
Transposed (ex. `A.'`)	`.'`
Inverse of matrix	`inv`
Linear system solver :$A\backslash b$ solves $A * x = b$	`\`
Element-wise logic operations	`| & ~`
Element-wise comparison operators	`== ~= < > <= >=`

Matrices

Operations over a matrix's diagonal

To retrieve the values of a matrix's diagonal in a vector shape, use the function `diag`. The same function can also create a diagonal matrix from a single vector

```
>> A = diag([1,2,3])
>> a = diag(A)
```

Retrieving the k^{th} diagonal is done in a similar way

```
>> k = 1
>> A = diag([1,2,3],k)
>> A = diag([1,2,3],-k)
```

```
>> A = magic(5)
>> a = diag(A,2)
```

Matrices

Broadcast operations

In the new versions of Matlab® (from 2016b on), the operations are *broadcast*: one can add scalars or vectors to matrices

```
>> A = [1, 2; 3, 4];
>> A + 1
   ans =
           2     3
           4     5
```

```
>> A = [1, 2; 3, 4];
>> b = [3, 2];
>> A+b
   ans =
           4     4
           6     6
```

Matrices

1. Define in Matlab® the following matrix and call it A.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 5 & 4 & 3 & 2 & 1 \\ 5 & 8 & 2 & 0 & 1 & 4 \end{pmatrix}$$

2. What happens when using a single index?

```
>> A3(5)
```

3. What happens here?

```
>> A=zeros(4,5);
>> A(:)=1:numel(A)
```

Matrices

1. Define in Matlab® the following matrix and call it A.
$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 5 & 4 & 3 & 2 & 1 \\ 5 & 8 & 2 & 0 & 1 & 4 \end{pmatrix}$$
2. What happens when using a single index?
```
>> A3(5)
```
3. What happens here?
```
>> A=zeros(4,5);
>> A(:)=1:numel(A)
```

→ The first element of the 5th row is returned

→ An empty matrix is created. Then, the elements are filled by the integers 1 to the number of elements of A

Multidimensional arrays

In Matlab®, vector and matrices arrays behave alike. And more generally, any array has the same Matlab® structure

Initialisation

```
>> A=rand(4, 5, 7, 4)
>> size(A)
```

Access to values

```
>> A(1,2,3,4)
>> A(1,:,:,4)
```

The syntax introduced above apply in a similar way to multidimensional arrays. The operators also act alike provided that there exists an algebraic meaning to the given instruction

Operations on arrays

Modifying array's elements

The elements of any array can be changed by overwriting the values stored at locations pointed by given indices

```
>> D = [1,2,3;4,5,6;7,8,9]
>> D(2,3)=7           % Changes this specific element
>> D(:,2)= [12,13,14] % Changes the full row
```

Extending an array

An array extension is done by storing a value at a new location

```
>> D = [1,2,3;4,5,6;7,8,9]
>> D(4:5,:) = [7,8,9;10,11,12]  % Two rows are added
>> size(D)                      % The dimension changed
```

Note: Mind the dimension compatibility when changing arrays

Operations on arrays

Testing an array element-wise

Testing the value of the array's elements against some condition is done through logical operators. It returns a *logical array*

```
>> A = [1 2 3 4; 5 4 3 2]
>> A<4
```

Finding elements in an array

The command `find` returns the index of the array's elements that satisfy a wished condition. The array's structure is not preserved

```
>> find([1,2,3,4]>6)      % Returns empty vector
>> find([2,3,4,5]>3)      % [3,4]
>> find(A<4)              % [1 3 5 6 8]'
>> A(find(A<4))           % [1 2 3 3 2]'
```

Operations on arrays

Clearing elements in an array

Removing a row or a column is done by assigning `[]` as a new value to the column or row one wishes to delete

```
>> A(:,[2,4]) = []  % Remove the 2nd and 4th columns
>> A(1,:) = []      % Remove the first row
>> A(1:3,1:2) = []  % Error: cannot remove block
>> A(1,1) = []      %        portions of the matrix
```

Removing spurious dimensions

The function `squeeze` removes the dimensions that are not necessary for representing the array

```
>> D2=reshape(D,[3,3,1,1,2,1,1])
>> D3=squeeze(D2)
```

```
>> size(D2)
>> size(D3)
```

Operations on arrays

Shifting dimensions of multidimensional array

The dimensions of any array can be shifted, rolling the matrix to the left (right) when the shift index is positive (negative)

```
>> C=shiftdim(D3,1);
>> size(C)
```

```
>> D3(3,2,1)
>> C(2,1,3)
```

Generic functions on multidimensional arrays

Most of the functions operate on multidimensional arrays with the same syntax as for vector and matrix arrays

```
>> A=rand(3);
>> B=repmat(A,[1,1,2])
>> size(B)
```

```
>> A=rand(3);
>> sin(A)
>> log(A)
```

Operations on arrays

Functions designed for arrays

Some functions used to extract and infer information from a given array are already implemented. By default, they act column-wise

`max`	Find the maximum
`min`	Find the minimum
`sum`, `mean`, `median`	Statistical quantities
`std`	Standard deviation

```
>> max(b(:))
>> max(b)
```

```
>> max(A(:))
>> max(A)
```

```
>> s = sum(D3)
>> size(s)
```

To specify the dimension along which the function acts:

```
>> max(A,2)
```

```
>> sum(D3,1)
>> sum(D3,[1,2])
```

Operations on arrays

Functions that act on two multidimensional arrays

`isequal` Checks if two matrices are equal

`kron` Computes the outer tensor product of two matrices:

$$kron(A, B) = \begin{pmatrix} A_{11}B & A_{12}B & ... & A_{1n}B \\ ... & ... & ... & ... \\ A_{m1B} & ... & ... & A_{mn}B \end{pmatrix}$$

```
>> isequal(D2, D3)
```

```
>> kron(A, A')
```

Vectorisation spirit

1. Initialise the three quantities
   `A = magic(200); b = ones(200,1)` and `c=0;`
2. Create two scripts containing the following instructions

```
% Intuitive code
tic
c = 0;
for k=1:200
  for l=1:200
    c =c +...
       A(k,l)*b(l);
  end
end
speed1 = toc;
```

```
% Vectorised code
tic
c = sum(A*b);
speed2 = toc
```

3. Check that the two results for `c` match and compare the values of `speed1` and `speed2`.

Exercises

Exercises: Matrix calculus

Exercise

1. Construct in Matlab the following matrices in an efficient way (use `help diag`):

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 1 & 1 & 1 & 3 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 4 & 0 \\ 2 & 2 & 2 & 2 & 0 & 0 & 5 \\ 2 & 2 & 2 & 2 & 6 & 6 & 7 \\ 2 & 2 & 2 & 2 & 6 & 6 & 8 \end{pmatrix} \quad C = \begin{pmatrix} 5 & 1 & 0 & 0 & 0 & 5 & 5 & 6 & 7 & 8 \\ 6 & 0 & 2 & 0 & 0 & 6 & 1 & 0 & 3 & 0 \\ 7 & 0 & 0 & 3 & 0 & 7 & 0 & 2 & 0 & 4 \\ 8 & 0 & 0 & 0 & 4 & 8 & 5 & 6 & 7 & 8 \end{pmatrix}$$

Exercises : Matrix calculus

Exercise

2. Generate a 10×10 random matrix. Then, find the values and indexes of the biggest element of each row.

   Possible hints: `sort, max, find`

3. Compute the inverse of a random vector (entry–wise).

4. Write a function, which shifts cyclically a vector or matrix by a certain number of rows down and columns to the right (**do not use** `circshift`).

Exercises: Matrix calculus

Exercise

5. Checkboard:
   a) Generate a random (`rand`) matrix A with size $n \times n$ and a second matrix with the same size as A and which takes some elements of A in the black fields, while zeros everywhere else. The structure of the second matrix should be similar to a chessboard.
   b) Make sure that your code works for the cases when n is even and odd. To check if you succeeded try `spy` on your matrix.

Exercises: Matrix calculus

Exercise

6. Give the in-built functions of MatLab that can you use to compute, given a matrix of type $m \times n$,
   a) the average,
   b) the median,
   c) the mode.
7. Check what the following functions do.
   a) `rank`
   b) `null`
   c) `rref`
   d) `orth`

Exercises: Matrix calculus

Exercise

8. Guess and verify on the computer what is happening.

```
>> hist(rand(1,100000),100)
>> hist(randn(1,100000),100)
```

9. Let us consider the square matrix A defined as

```
>> A=[1+i, 1+2i; 2, 4i]
```

Check the following expressions.

```
>> A’
>> A.’
>> A.^A
```

Exercises: Matrix calculus

Exercise

10. Consider `A=rand(10)` and check the result of

```
>> diag(diag(A)) + diag(diag(A,1),1)+...
   diag(diag(A,2),2)
```

11. Transform the matrix

```
>> A=reshape(1:20, 4, 5)';
```

through elementar row changes in a reduced row-echelon form (germ.: Zeilenstufenform). You should set to 0 any number smaller than `tol=10^(-13)`. You can compare your results with `rref`.

Exercises: Matrix calculus

Exercise

12. Guess the type and value of the following expressions, where `A=[1 2; 3 4*i]`. Check it then in the prompt.

    a)
    ```
    >> isequal(A, A');
    ```

    b)
    ```
    >> sum(sum(A - A')') == 10;
    >> sum(diag(A == A')) & abs(sum(sum(A)));
    ```

Exercises: Matrix calculus

Exercise

13. Generate a table with N values of the function $f(x)$ on the interval $[a, b]$, s.t. you get an $N \times 2$ matrix having on each row x and $f(x)$. Consider the following functions:
    a) $f(x) = x^5 - 4x + 1$
    b) $f(x) = exp(i\frac{x}{10})$
14. Find and cure the error in the script on the webpage `search_error.m`. The script should give an ordered list of numbers generated by a random vector.

Visualisation

Visualisation

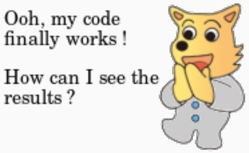
Scientific visualisation is the field that developps graphical tools to illustrate scientific results or data to help their understanding

As a general rule, any graphical representation of data should be

- Meaningful: it emphases the information present in the data
- Self-containing: the graphical output is intelligible by itself

In Matlab®, many visualisation tools are already implemented

- Vectorial tools *(2D and 3D plots, ...)*
- Statistical tools *(bar, pie, ...)*



Note: The set of instructions that extracts intelligible information from the *raw data* is called *post-processing*

Vector plots

Create a simple plot of a vector

1. Observe the behaviour of the following instructions

```
>> x=linspace(0,2*pi,200); y1=sin(x); y2=cos(x);
>> plot(y1)
>> plot(y2)
```

2. Try to plot each of the following and see what changes

```
>> plot(y1, y2)
>> plot([y1',y2'])
>> plot([y1; y2])
```

Vector plots

Create a simple plot of a vector

1. Observe the behaviour of the following instructions

```
>> x=linspace(0,2*pi,200); y1=sin(x); y2=cos(x);
>> plot(y1)
>> plot(y2)
```

2. Try to plot each of the following and see what changes

```
>> plot(y1, y2)
>> plot([y1',y2'])
>> plot([y1; y2])
```

→ The command `plot` plots a given vector with respect to the vector's indices: from 1 to `length(y)`

→ It plots a given matrix by plotting each of its columns against the row indices

Vector plots

Create a simple plot of a vector

3. To plot against a vector of antecedents, include it in the arguments of the plot function. Plot with respect to the variable `x` by writing

```
>> x=linspace(0,2*pi,200); y1=sin(x); y2=cos(x);
>> plot(x,y1)
>> plot(x,y1,x,y2)
>> plot(x,[y1', y2'])
```

Vector plots

Create a simple plot of a vector

3. To plot against a vector of antecedents, include it in the arguments of the plot function. Plot with respect to the variable x by writing

```
>> x=linspace(0,2*pi,200); y1=sin(x); y2=cos(x);
>> plot(x,y1)
>> plot(x,y1,x,y2)
>> plot(x,[y1', y2'])
```

→ When two vectors are given, the second vector is plot against the first vector. If there is more arguments, this behaviour repeats itself until the end of the argument list.

→ When a vector and a matrix are given, each of the column vectors of the matrix are plot against the vector.

→ Pay attention to the order of the arguments!

Functions plots

Create an easy plot of a function

1. Plot an anonymous function evaluated on a vector

```
>> f = @(x) -x^2+2
>> x=linspace(0,1,10);  plot(x, f(x));
>> x=linspace(0,1,200); plot(x, f(x));
```

2. Plot an anonymous function between bounds

```
>> ezplot(f, [0, 1])    % Depreciated
>> fplot(f, [0,1])
```

Functions plots

Create an easy plot of a function

1. Plot an anonymous function evaluated on a vector

```
>> f = @(x) -x^2+2
>> x=linspace(0,1,10);  plot(x, f(x));
>> x=linspace(0,1,200); plot(x, f(x));
```

2. Plot an anonymous function between bounds

```
>> ezplot(f, [0, 1])    % Depreciated
>> fplot(f, [0,1])
```

→ Plotting by hand an anonymous function after evaluating it on a vector makes the resolution of the plot dependent on the predefined vector. Using a function designed for anonymous functions is usually preferable

Different plot scales

Use specific functions to plot on a different scale

1. Plot all the functions
$$f(x) = e^{3x},\ g(x) = x^x,\ h(x) = \log(x),\ j(x) = 3x$$
on $[0.01, 100]$ using each of the following functions (use the command `help` to access their documentation)

```
plot     semilogx     semilogy     loglog
```

2. Which scale suits the best each function?

Different plot scales

Use specific functions to plot on a different scale

1. Plot all the functions
   $$f(x) = e^{3x},\ g(x) = x^x,\ h(x) = \log(x),\ j(x) = 3x$$
   on $[0.01, 100]$ using each of the following functions (use the command `help` to access their documentation)
   ```
   plot     semilogx     semilogy     loglog
   ```
2. Which scale suits the best each function?

→ Those commands are useful when plotting logarithmic and exponential function, which are very common in error plots.

Multiple plots on one figure

Create several vectors or functions on a single plot

1. We already saw that `plot(x, y1, x, y2)` plots `y1` and `y2` with respect to `x` on the same figure. Try now

```
>> hold on
>> plot(x, y1)
>> plot(x, y2)
>> hold off
```

2. What would happen here?

```
>> close all; plot(x, y1)
>> hold on
>> for a = 1:10; plot(x, a.*x); end
>> hold off
```

Multiple plots on one figure

Create several vectors or functions on a single plot

1. We already saw that `plot(x, y1, x, y2)` plots `y1` and `y2` with respect to `x` on the same figure. Try now

```
>> hold on
>> plot(x, y1)
>> plot(x, y2)
>> hold off
```

2. What would happen here?

```
>> close all; plot(x, y1)
>> hold on
>> for a = 1:10; plot(x, a.*x); end
>> hold off
```

→ The commands `hold on` and `hold off` are useful when generating plots in a loop. Be careful where to specify them!

The attributes of a useful figure

For a figure to be meaningful and intelligible, thus *useful*, several features providing information on the plot have to be added

Title
: *The global title of your figure, telling what you are focusing on*

Colors and line typology
: *Identifies a specific plot in a figure containing many. By default, automatic colors are selected. The line typology (plain, dashed) further helps color-blinds and allows to print in black and white*

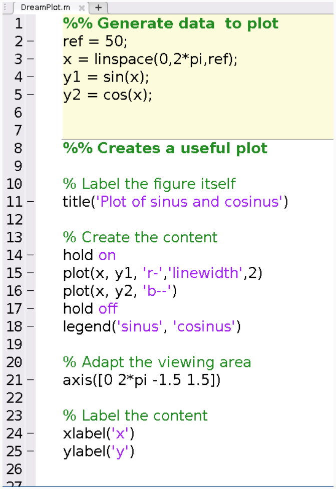
Legend
: *When several plots are in one figure, it makes clear which plot corresponds to which vector or function*

Axis labels
: *To know the used scaling, and the variable spanned on each axis*

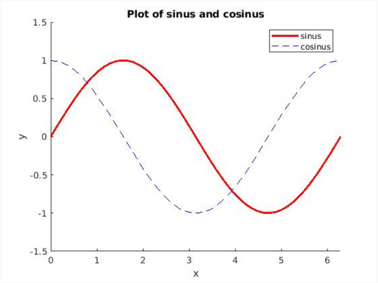
Axis bounds
: *Crops the figure to the range of values given for each axis*

The attributes of a useful figure

A dream plot



```
%% Generate data  to plot
ref = 50;
x = linspace(0,2*pi,ref);
y1 = sin(x);
y2 = cos(x);


%% Creates a useful plot

% Label the figure itself
title('Plot of sinus and cosinus')

% Create the content
hold on
plot(x, y1, 'r-','linewidth',2)
plot(x, y2, 'b--')
hold off
legend('sinus', 'cosinus')

% Adapt the viewing area
axis([0 2*pi -1.5 1.5])

% Label the content
xlabel('x')
ylabel('y')
```



Multiple figures

Defining multiple figures

When the visualization is desired in different figures, one can use the command `figure()` to draw the plot in a new figure

```
>> figure()           % Opens a new figure,
>> plot(x,y1)         % automatically indexed
```

It is also possible to select a particular figure and edit it later on

```
>> figure(1)            % Opens or jumps to the figure 1
>> plot(x,y1)
>> figure(2)            % Opens or jumps to the figure 2
>> plot(x,y2)
```

Multiple figures

Defining sub-figures

Sub-figures are used when the visualisation is wished in one single figure but should show distinct plotting spaces

`subplot(a,b,n)`: Creates and selects a sub-figure environment
`n` is the place where the plot is collocated
`a,b` are the dimension of the collocating matrix

```
>> figure(3)          % Create or jumps to the figure 3
>> subplot(1,2,1)     % Creates a 1x2 collocating matrix
>> plot(x,y1)         % and plots y1 at the position 1
>> subplot(1,2,2)     % Selects the second position
>> plot(x,y2)         % and plots y2
```

Figure management

Saving figures

Saving a created figure can be done either through a graphical interaction or by script instructions

- By scripting, the main commands are `savefig` and `saveas`

  ```
  >> fig = figure()   % Points to the desired figure
  >> savefig('my filename0')
  >> savefig(fig,'my filename1')
  >> saveas(fig, 'my filename2')
  ```

  To export the figure in a specific format, use an optional argument in the command `saveas`

  ```
  >> saveas(fig,'my filename2','png')  % png, eps,...
  ```

- Graphically, go to File → Save As and select the format

Figures management

Clearing figures

All the options to clear or close figures act on the main figures, regardless whether the figure contains sub-figures or not

- Clear the content of the figures, without closing them

```
>> clf
```

- Close the last figure

```
>> close
```

- Close all the figures

```
>> close all
```

Note: Always clear figures *after* having saved them, otherwise you will save a blank figure

3D Plots

1. Create a 3D line by entering

```
>> plot3(x,y1,y2)
```

2. Create a 3D Surface by using the commands `meshgrid` and `surf`:

```
>> [xx, yy]=meshgrid(x,x);
>> surf(xx,yy,sin(xx).*cos(yy))
```

3. What happens if we type `*` instead of `.*`?

3D Plots

1. Create a 3D line by entering

```
>> plot3(x,y1,y2)
```

2. Create a 3D Surface by using the commands `meshgrid` and `surf`:

```
>> [xx, yy]=meshgrid(x,x);
>> surf(xx,yy,sin(xx).*cos(yy))
```

3. What happens if we type `*` instead of `.*`?

→ The command `meshgrid` creates a grid of (x, y) coordinates upon two given scalar vectors

→ The command `surf` creates surface by interpolating given values at the corresponding grid points

→ We get a matrix product, which is not the desired result

Particular typologies of plots

Various of other types of plots are available *(see the Matlab® documentation)*. As a brief insight, the tools commonly used for statistics and 2D data visualisation are respectively

- Bar graphs

```
>> bar(rand(1,10))
```

- Pie chart

```
>> pie(rand(1,5))
>> pie3(rand(1,5))
```

- Scatter plot

```
>> scatter(y1, y2)
```

- Mesh visualisation

```
>> mesh(peaks)
```

- Contour visualisation

```
>> contour(peaks)
```

- Color fill

```
>> pcolor(peaks)
```

Note: The plotting functions have options, described in the help

Animation

Create your first animation!

1. Put together multiple plots and store the corresponding frames in an array

```
clear M
x=linspace(0,2*pi,100); y=sin(x);
n=120;
for k=1:n
      plot(x,y*sin(pi* k/n))
      axis([0,2*pi,-1,1])
      M(k)=getframe;
end
```

2. Launch your movie with `movie(M,2)`

Animation

Save your first animation

1. Save your first animation with `VideoWriter`

```
writerObj = VideoWriter('myvid.avi');
open(writerObj);

for k=1:n
     plot(x,y*sin(pi* k/n)
     axis([0,2*pi,-1,1])
     writeVideo(writerObj, getframe)
end

close(writerObj)
```

Visualisation

Best practice

- Provide a complete description of your graphics, with a title, plots and axes labels, ...
- Use markers and line properties so that your plots can be also differentiated when printed in black and white
- Always save the figures *before* clearing them
- Always clear your figures when switching exercise

Exercises

Exercises: Visualisation

Exercise

1. Plot in one figure the functions
   $f(x) = e^{x/10}\sin(2\pi x)$ and $g(x) = log(3 + x)\cos(4\pi x)$
   on the interval $[0, 1]$. The plot be such that:

   a) f is plotted in red colour and dashed lines
   b) g is in blue and it is alternating dots and dashes
   c) It should contain a title "Cute functions"
   d) The x axis ranges from 0 to 1 and is labelled "Time"
   e) The y axis ranges from -2 to 2 and is labelled "Money"
   f) Specify a legend: f relates to "Marc" and g to "John"

2. Save the plot as `"my_first_functions.fig"` and close the figure (using the functions we learned).

Exercises: Visualisation

Exercise

3. Given the serie $\sum_{k=1}^{\infty} \frac{1}{2^k} = 1$
   a) Plot the partial sums $A_n = \sum_{k=1}^{n} \frac{1}{2^k}$ with respect to n and an horizontal line at the hight of the limit 1
   b) Plot the error $e_n = |1 - \sum_{k=1}^{n} \frac{1}{2^k}|$ with respect to n by choosing the more appropriate norm

4. Plot the first 4 (ν=0,1,2,3) Bessel-functions of 1st and 2nd typology. Hint: use the `besselj, bessely` functions
   a) Plot all the functions of the first type in a same figure, and all the functions of the second type in an other one
   b) The $\nu = 1, 2$ functions must be put in a 1×2 array of plots, with corresponding descriptions and labels
   c) Change the x-axis to be $[0.2, 20]$ and the y-axis $[-1, 1]$

Exercises: Visualisation

Exercise

5. Check `help plot`. Create then the plots of:
    a) $f = x^2 - 0.5$ on $[-1, 1]$ with red dashed lines
    b) $f = sin(2 * pi * x)$, where the function values have to be displayed as little black circles at the points $x = n/10$.
    c) the functions $f = sin(s * x)$, $s = 1, 2, 3, 4$ with different colors and a legend

Exercises: Visualisation

Exercise

6. a) Plot the first eigenmodes of a quadratic cymbal with the length of its side 1

   b) Create a video of an overlay of two eigenmodes. Hint: The eigen-oscillations of the trumpet are given through the formula:

$$sin(m \cdot \pi \cdot x)\ \ sin(n \cdot \pi \cdot y)\ \ sin(c\sqrt{m^2+n^2}t+\varphi)$$

Further data types

Types of data

In Matlab®, the nature of any data is given by its *typology*

Logical values *(0 or 1)*	`logical`
Floating-point	`single`, `double` numbers
Integers	`int8`, `int16`, `int32`, `int64`
Unsigned integers	`uint8`, `uint16`, `uint32`, `uint64`
Character *(16 bit)*	`char`
Sequence of characters	`string`
Varying data type	`cell`
Matrix *(named row and columns)*	`table`
Data structure with fields	`struct`
Pointer to a function	`function_handle`
Other data types *(not seen here)*	`object`, `map`, `graphics`
	`handle`, `time series`, `...`

Typology of data

To check the default data type assigned by Matlab® to your variable, use the command `class`

```
>> a= 1==0;
>> class(a)
>> b=rand(5);
>> class(b)
```

```
>> a= [1.5,2.4,3.0];
>> class(a)
>> b=[1,2,3];
>> class(b)
```

To change the type of any variable, convert it with the function named as the wished type (see the above table for a list)

```
>> a=1.5; b=int8(a)   % Converts a float to an integer
>> b, class(b)        % The float has been floored
>> char(1:1000)       % Converts to a vector of chars
```

Types of data

Combination of different data typologies

Combining data of different typologies through operators may be possible. Matlab® converts the data into the type it thinks the most suitable. However, unexpected result can occur

```
>> class(b*2.7)
>> class(b*single(2.7))
```

```
>> 2*"2"
>> 2*'2'
```

Similarly, a data conversion may involve several typology changes, leading to hidden typologies combination that may also yield errors

```
>> a="3"; b=int8(a)   % Leads to an error
>> int8('3')          % Does not map to 3
>> int8('2.3')        % Gives out an array
```

Types of data

Combination of different data typologies

Combining data of different typologies through operators may be possible. Matlab® converts the data into the type it thinks the most suitable. However, unexpected result can occur

```
>> class(b*2.7)
>> class(b*single(2.7))
```

```
>> 2*"2"
>> 2*'2'
```

Similarly, a data conversion may involve several typology changes, leading to hidden typologies combination that may also yield errors

```
>> a="3"; b=int8(a)   % Leads to an error
>> int8('3')          % Does not map to 3
>> int8('2.3')        % Gives out an array
```

Characters and strings

Characters

A way to define variables containing non-numeric data is to use characters of type `char`

```
>> c2 = 'q', c1 = '@'
```

A sequence of characters is a character array *(formerly character string)*, which is a $1 \times n$ matrix of type `char`

```
>> s='hello'
>> class(s)
```

```
>> size(s)
>> s(2)
```

The concatenation of character strings is done as for vectors by

```
>> s2 = [s ' world']
>> s2(1)
```

```
>> s=['hallo', 'world'];
>> s(1), s(2)
```

Characters and strings

Strings

A string is a container for text, defined by double quotes `"`

```
>> s = "hello"
>> s
>> s(1)
```

A string array is defined as follows

```
>> s = ["hello", "world"];
>> s(1)
>> s(2)
```

Characters and strings

Conversion between characters’ array and string

To convert a string into a character array or an array of characters into a string, use the functions named as the desired type

```
>> z = char(s)
```

```
>> t = string(z)
```

Conversion between strings numbers

To convert a number into a string and *vis-versa*, there exists in-built conversion function

```
>> num2str(1.234)
>> str2double('1.234')
```

A default format is used in the string creation, while the conversion to a number is done up to machine precision

Strings' formatting

Informing the user of the results leads to create meaningful sentences or export structured data using *string formatting*

```
>> sprintf('The number is %f or %d.\n', 1.234, 5)
>> sprintf('The number is \t %.1f.', 1.234)
>> sprintf('The number is %f. ', [1,2,3,4])
```

The function `sprintf` also allows you to concatenate sentences. To replace a part of a sentence, use the function `strrep`

```
>> sprintf('This is %d concatenated %s', "string", 1)
>> strrep('A Test', 'Test', 'Toast')
>> strrep('A Test', "Test", 'Toast')
```

A small aside: date and time

Date representation

Operations on date and time are made easy in Matlab® with predefined variables storing the time in a `double`, representing the amount of previous days since the 0. January 0

```
>> now
>> today
```

```
>> now-today
>> date
```

The vector of doubles containing the year/month/day/ hour/minute/seconds is given by `datevec`

```
>> datevec(now)
```

A small aside: date and time

Date conversion to string

A conversion from the date's double representation to a human readable string, is given by the commands `datestr`.
The string is automatically formatted

```
>> datestr(now)
>> datestr(today)
>> datestr(0)
```

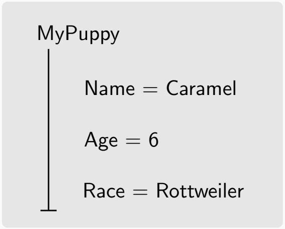
The reverse operation is given by the command `datenum`

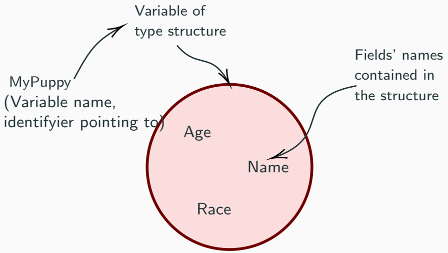
```
>> datenum('2-Mar-1973')
>> datenum('2.3.1973 09:47', 'dd.mm.yyyy HH:MM')
```

Structures

Structures' definition

A *structure* is a type of variable that contains different fields, useful to group values that characterise a single entity




Variable of type structure
MyPuppy (Variable name, identifyier pointing to)
Fields' names contained in the structure
Age
Name
Race

A structure with initial valued fields is defined with the command `struct`. Defining a structure on the fly is also possible

```
>> S1 = struct("Field1","Value1","Field2","Value2")
```

Structures

Structures' syntax

The fields can be set and grabbed through `( structname).(fieldname)`

```
>> calib = struct                 % Decalres the structure
>> calib.focal_length = 1.2       % New field + given value
>> calib.date = now               % Fields of any type
>> calib.visual.color = "Red"     % Nested structure
>> fieldnames(calib)
```

As usual, the size of the structure's array is automatically adjusted

```
>> calib(3).focal_length=1.5
>> size(calib)
```

Containers' Map

Maps' definition and value's access

A map is an object from the package containers that associates values to unique keys, comparable to dict in python

```
>> % Create a Map object that contains rainfall data
>> keySet = {'Jan','Feb','Mar','Apr'};
>> valueSet = [327.2 368.2 197.6 178.4];
>> M = containers.Map(keySet,valueSet)
```

Retrieving the stored data is done through specific methods

```
>> M.keys()
>> M.length()
>> M.values()
```

How much was the rainfall in March?

Containers' Map

Maps' definition and value's access

A `map` is an object from the package `containers` that associates values to unique keys, comparable to `dict` in python

```
>> % Create a Map object that contains rainfall data
>> keySet = {'Jan','Feb','Mar','Apr'};
>> valueSet = [327.2 368.2 197.6 178.4];
>> M = containers.Map(keySet,valueSet)
```

Retrieving the stored data is done through specific methods

```
>> M.keys()
>> M.length()
>> M.values()
```

How much was the rainfall in March?

$\rightarrow$ `>> M('Mar')`

Cell arrays

Cell arrays' definition

Cells arrays are similar to matrices of type `cell` whose fields can contain any data typology (comparable to `list` is python)

```
>> s={ [1 2 3], 'example'; 2, 1==0 }
>> class(s), size(s)
```

Extracting parts of a cell array is done as usual, but the obtained data type is `cell`. Accessing the content of a cell itself is done with curly brackets `{}`

```
>> a=s(2,1)
>> class(a) % Is a cell
```

```
>> a=s{2,1}
>> class(a)
```

Cell arrays

Cell arrays' specificities

Using a comma-separated list in the definition of a cell array is equivalent to separate each field with a comma

```
>> s2={1,2,[3,4]}
>> [s2{:}]
```

Accessing several parts of cell array is done as for any array, but returns two distinct quantities

```
>> s{1, 1:2}
```

What outputs from?

```
>> class(s{1,1:2})
```

Cell arrays

Cell arrays' specificities

Using a comma-separated list in the definition of a cell array is equivalent to separate each field with a comma

```
>> s2={1,2,[3,4]}
>> [s2{:}]
```

Accessing several parts of cell array is done as for any array, but returns two distinct quantities

```
>> s{1, 1:2}
```

What outputs from?

```
>> class(s{1,1:2})
```

→ Error: no constructor

Exercises: Further data types

Exercise

1. Display in the prompt a string of the form:

   The value of x^2 at 2 is 4.
   The value of x^2 at 2.1 is 4.41.

   for x ranging from 1 to 3.
   Hint: a new line is created with `\n`

2. How many days lasted the first world war (28-Jul-1914, 11-Nov-1918)?

3. What does `mat2cell` do?

Exercises: Further data types

Exercise

4. Guess the type and value of the following expressions. Check it then in the prompt.

(c)
```
>> int8(255) + 1;
>> int32(int8(255) + 1) * 5;
```

(d)
```
>> logical(255) + 1;
>> int64(double(int64(2)^63)-1) - int64(2)^63
```

Functions’ inputs and outputs

Variable functions' arguments

Keywords controlling the function's arguments

Allowing a function to be called with a various number of arguments requires the use of the keywords `varargin` and `varargout` in the function's header

```
function varargout = FlexibleFunction(varargin)
    ....
end
```

If we pass N arguments to a function and retrieve M outputs, `varargin` and `varargout` are $1 \times N$ and $1 \times M$ **cell arrays**, respectively

- The numbers of arguments are thus not known *a-priori*.
- The keywords `nargin`, `nargout` contain a negative value

Variable functions' arguments

Variable number of inputs

Define and test the following function on various number and types of input arguments, all stored in the cell array `varargin`

```
function [ ninp ] = varargin_function(varargin)
   % This function displays all the inputs

   ninp=size(varargin,2);

   for i=1:ninp
       disp(varargin{i})
   end
end
```

Variable functions' arguments

Variable number of outputs

Define and test the following function on various number and types of outputs, returned in the cell array `varargout`

```
function [varargout] = varargout_function()
    % This returns as many outputs as called
    for i=1:nargout
      if mod(i,2)==0
        varargout{i}=i^2;
      else
        varargout{i}=char(100+i);
      end
    end
end
```

Variable functions' arguments

Input strings' specificity

If a function takes strings as inputs, those commands are equivalent

```
>> varargin_function a b c
>> varargin_function 'a' 'b' 'c'
>> varargin_function('a' ,'b', 'c')
```

This behaviour will break on any other kind of input data

1. The following is a mistake... why?

```
>> sin pi
```

2. Does this work?

```
>> disp hello
>> disp hello world
```

Variable functions' arguments

Input strings' specificity

If a function takes strings as inputs, those commands are equivalent

```
>> varargin_function a b c
>> varargin_function 'a' 'b' 'c'
>> varargin_function('a' ,'b', 'c')
```

This behaviour will break on any other kind of input data

1. The following is a mistake... why?

```
>> sin pi
```

2. Does this work?

```
>> disp hello
>> disp hello world
```

→ `pi` is understood as a char, incompatible with the function `sin`

Function as an argument

Anonymous function as an argument

If a function takes strings as inputs, those commands are equivalent

```
% Calling the function
>> g = @(t) cos(t)+2
>> x = 2; y = 4;
>> w = Func(g,x,y)
```

```
% Function's definition
function z = Func(f,x,y)
    z = f(x) + y
end
```

It is also possible to pass Matlab®'s predefined functions and use anonymous functions

```
>> g = @(f, t) cos(t)+2*f(t);
>> x = 2;
>> w = g(@sin,x)
```

Function as an argument

In-file function as an argument

It is also possible a self-defined function which is stored in an external file as an argument. However, one has to point to the function's identifier by using the symbol `@`

```
% Calling the function
>> x=2; y=4;
>> w=Func(@L1_power8,x,y)
```

```
% Function's definition
function z = Func(f,x,y)
    z = f(x) + y
end
```

Note: Be careful to the *path* where the function is stored, Matlab® may not find it automatically

Data input and output

External data management

Save and load binary data (1/2)

The most convenient way to save or read data when that is not used outside Matlab® is to create a *binary file*

```
>> save          % Saves the workspace in the matlab.mat
>> clear all
>> load          % Loads a workspace from matlab.mat
```

Save a given variable in a specific file by specifying arguments

```
>> A=rand(3); B=rand(4);
>> save myfile.mat A B
>> clear
>> load myfile
>> who
```

```
>> A=rand(3); B=rand(4);
>> save("myfile","A","B")
>> clear
>> load("myfile")
>> who
```

Save and load binary data (2/2)

When the name of the variables contained in the file are known *a-priori*, it is possible to load it specifically

```
>> load myfile.mat A       % Loads the variable A only
>> who
```

When loading the file content into a variable, its type is `struct` and its fields are the names of the variables stored in the file

```
>> saved = load('myfile.mat');
>> saved                              % Look in this structure
>> A_new = saved.A
>> B_new = saved.B
```

Note: save in a text file (no formatting) with `save myfile.txt -ascii`

External data management

Save formatted files

If the data has to be exported in a format or typology that is human-readable, this has to be with the command `fprintf`

```
>> fid = fopen('myfile2.txt', 'w')
>> fprintf(fid, 'Date; Mesurement\n')
>> fprintf(fid, '%s; %f\n', datestr(now), 1.245)
>> fclose(fid)
```

- The command `fprintf` has the same structure as `sprintf` in the creation of formatted sentences to be written in the file
- The choice of file extension (.txt, .dat, .csv, .any , ...) is free. Be careful however not to use extensions misleading for Maltab® (.fig, .mat) or external software (.tiff, .odt, ...)

External data management

Load formatted files

A structured way to read data from files having a predefined format is to use `textscan`. It returns a cell array, the cells' elements corresponding to the columns (see the help for more options)

```
>> fid=fopen('ListOfThings.txt');

>> % Get the 1-line header of the file
>> HDR = textscan(fid,'%s %s',1, 'delimiter',';');

>> % Get all the following rows having a same layout
>> DATA = textscan(fid,'%s %f', 'delimiter',';');
>> meastimes = datevec(DATA{1},'dd.mm.yyyy HH:MM');
>> mesval = DATA{2};

>> fclose(fid);
```

External data management

Save and load automatically CSV formatted files

When a file is simply formatted as Comma Separated Values, it is possible to save and read data automatically from it with `csvread` and `csvwrite` (see the help for more options)

```
>> A=rand(4)
>> csvwrite('file.csv',A)
```

```
>> clear all
>> B=csvread('file.csv')
```

Separation signs as comma, semicolon or blank are recognized. Specific separators can be manually defined as a 2nd argument

Further means of saving and loading external data

There are many more ways to save and read data

- Text files and human-readable (ascii) formats can be dealt with `fread`, `fwrite`, `dlmread`, `dlmwrite`, `importdata`, ...
- Specific routines for database files or spreadsheets are also available `xlsread`, `xlswrite`, ...

Each command has its own coding format. Check them in the help

```
>> help importdata
```

Note: The Matlab GUI also contains some information about how to import and export data to generate *Import-scripts* (Import Data Button)

Data input and output

Best practice

- When the data is wished to be used within Matlab® only, prefer using binary format
- Choose an formatting rule where you loose the less information as possible when exporting the data in a human-readable format
- Pay attention to give a meaningful file extension that is not yet commonly used when exporting data in your specific format

Exercises

Exercises: data input and output

Exercise

1. Write a function `mysum`, which takes as many inputs as desired. The output is given by partial sums:

   $$out(k) = \sum_{i=1}^{\texttt{nargin}} input(i), \quad \forall k = \{\texttt{nargin}{+}1, \texttt{nargout}\}$$

   if `nargin < nargout`, and

   $$out(k) = \sum_{i=1}^{k} input(i), \quad \forall k = \{1, \ldots, \texttt{nargin}\}$$

   otherwise

2. Enhance the function `curry`, in way to compute the composition of as many functions as the user wants.

   $$f_1(f_2(f_3(f_4 \ldots (f_n(x)) \ldots)))$$

Exercises: data input and output

Exercise

3. a) Write a script, which imports the real weather data from the file *wetter.txt*.

   b) Plot the data in a readable way (for example a different histogram for temperature, humidity and pressure).

Common uses and performance

Common uses

As seen in the introduction, Matlab® is commonly used for

- Algorithm development
- Modeling, simulation, and prototyping
- Data analysis, exploration, and visualization
- ...

Some functions and operators have then been designed specifically to ease the resolution of problems frequently arising in those fields

Common uses

Data analysis

When dealing with large sets of data, computing the mean-value, standard deviation and variance can be done with

`mean`, `stdvar`, `cov`

If the data is grouped, the function `grpstats` computes the summary statistics of each group in a matrix, dataset array or table

```
>> data = rand(50,1);
>> groups = randi(3,50,1);
>> m = grpstats(data, groups, 'mean')
>> ugrps = unique(groups)
```

Common uses

Linear system of equations

A recurrent problem in scientific computing is

Given A an invertible $n \times n$ matrix, b an $n \times 1$ vector, solve the system $Ax = b$

Matlab® offers two ways to solve it

```
>> x=inv(A)*b     % Depreciated, can fail
>> x=A\b          % Optimised, handles overdetermination
```

In practice, there is no exact solution for overdetermined systems of equation (A is a $m \times n$ matrix, $m > n$, b is a $m \times 1$ vector)

$\rightarrow$ Not computing an exact solution, the operator `\` minimizes the error `norm(A*x-b)`

Performance

Complexity

The *complexity* of an algorithm is a measure that estimates the number of single operations (+, -, *, /) required to end its execution

Let $f, g : \mathbb{N} \to \mathbb{R}$ be functions on natural numbers. Then it is said $f = \mathcal{O}(g)$ if there exists $c \in \mathbb{R}$, s.t.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < c$$

- The number of FLoating point Operations Per Second that a computer can do is given in *flops*
- Given the algorithm's complexity and the size of the initial data, the running time is known *a-priori*

Examples:

Search the biggest element of a list of length n: $\mathcal{O}(n)$
Naive multiplication of two $n \times n$ matrices: $\mathcal{O}(n^3)$

Computational efficiency measurement

It is possible to time the execution of blocks of code to compare the running efficiency of two coding possibilities or identify the slowest part of a code with the keywords `tic` and `toc`

```
>> A=rand(2000);
>> tic;              % Starts the timer
>> svd(A);           % Instructions to time
>> toc               % Stops the timer and
>>                   % returns the elapsed time
```

It is possible to start multiple timers

```
>> T1=tic;
>> % wait
>> T2=tic;
```

```
>> elapsed1=toc(T1)
>> elapsed2=toc(T2)
```

Performance

Coding habits and performance killing

Try in the prompt the following instructions, that in the end create the same vector

```
>> tic; x=[1:100000]; toc
>> tic; x=zeros(1,100); for
>> k=1:100000, x(k)=k; end, toc
```

What do you observe? What could be the reason(s)?

Coding habits and performance killing

Try in the prompt the following instructions, that in the end create the same vector

```
>> tic; x=[1:100000]; toc
>> tic; x=zeros(1,100); for
>> k=1:100000, x(k)=k; end, toc
```

What do you observe? What could be the reason(s)?

→ Observation: the speed changes by a factor 200!

→ Issue: the variable x changes at each iteration its size

→ Issue: Matlab® is an interpreted language: slow loops

Performance

Best practice

- Prefer algorithm having low complexity
- Use vectorisation whenever possible
- Try to avoid loops
- Do not increase the variables' dimension inside loops

Symbolic computations

Symbolic Math Toolbox

Definition of symbolic variables

Though **usually not advised**, *symbolic computations* are possible. Symbolic variables should be declared as such before their first use

```
>> syms x y
>> class(x)
```

Computations are then possible with symbols, *i.e.* with no specific value assigned to the *e.g.* `x` ot `y`

```
>> cc=sqrt(x+y)
>> class(cc)
>> f(x) = sqrt(x)
```

Symbolic Math Toolbox

Symbolic representation of numbers

A variable's value can be assigned by its symbolic representation

```
>> a=sym('123312312312312.123213123123213123123123')
>> sym('10/3')
>> pp=sym('pi'), sin(pp)
```

The precision of the arithmetic evaluation of a symbolic expression is up to Matlab®'s interpretation unless explicitely specified

```
>> vpa(pp)
>> vpa(pp,500)
```

When representing a number, a sym variable can be turned double

```
>> double(pp)
```

```
>> double(cc)
```

Symbolic Math Toolbox

Symbolic representation of matrices

A matrix can also be represented symbolically

```
>> A=sym('A',[2,2])
```

Algebraic quantities are then computed symbolically, returning their expression in function of the symbolic matrix's coefficients

```
>> inv(A)
>> eig(A)
```

Symbolic Math Toolbox

Using symbolic expressions (1/2)

When working with symbolic variables acting as symbols and not represented numbers, one can:

- replace or substitute values in an expression
  defining a symbolic variables representing a number

```
>> a=subs(sin(x),x,3*pi/2)
```

- simplify expressions

```
>> a=sin(x)^2+cos(x)^2
>> simplify(a)
```

- perform algebraic operations

```
>> expand((1+x)^10)
>> factor(x^10-1)
```

Symbolic Math Toolbox

Using symbolic expressions (2/2)

Many analysis operations that can be used on symbolic expressions are already implemented in the toolbox

```
>> limit(x*sin(x),x,0)
>> diff(x^5,x,2)
>> int(x^5,x)
>> int(1/x,x,1,2)
>> symsum(k^2,k,0,10)
>> symprod(1-1/k^2,k,2,Inf)
```

Note: Those functions take different kind of arguments, retrieve their details in the help

Symbolic Math Toolbox

Solving symbolic equations

Solve algebraic equation by writing naturally their expression after having declared the variable to solve for as a symbolic one

```
>> solve(x^3+2*x==1)
>> [a,b]=solve(x+y==1,...
>>               x-y==-1)
```

```
>> x = sym('1/2')
>> solve(x == 4)
```

Differential equations can be solved in a similar manner

```
>> syms y(t)
>> dsolve(diff(y,2)==-y,y(0)==0)
```

Symbolic Math Toolbox

Solving symbolic equations

Solve algebraic equation by writing naturally their expression after having declared the variable to solve for as a symbolic one

```
>> solve(x^3+2*x==1)
>> [a,b]=solve(x+y==1,...
>>                x-y==-1)
```

```
>> x = sym('1/2')
>> solve(x == 4)
```

$\rightarrow$ Error: `x` had a value

Differential equations can be solved in a similar manner

```
>> syms y(t)
>> dsolve(diff(y,2)==-y,y(0)==0)
```

Symbolic Math Toolbox

Symbolic visualisation

Plotting with the symbolic library is similar to the classical Matlab® framework

```
>> ezplot(sin(x), [0,3*pi])
>> syms y
>> ezsurf(sin(x)*cos(y),[0,3*pi,0,2*pi])
```

It is particularly useful for anonymous functions

```
>> ezplot(@(s) sin(s),[0,2*pi])
>> ezplot(@cos,[0,2*pi])
```

Exercises

Exercises: Performance

Exercise

1. Write a program which measures (approximately) the complexity of a matrix-multiplication

   Hint: $f(n) \approx Cn^k \Rightarrow \frac{f(n_1)}{f(n_2)} \approx \left(\frac{n_1}{n_2}\right)^k$

2. Write a program which measures the complexity of the operation `svd`.

Exercises: Symbolic Math Toolbox

Exercise

3. Compute the 57-th number after the comma of the Euler number
4. Compute the integrals of
   a) $sin(3x)\,cos(5x)$
   b) $x\,ln(x)$ on $[2,5]$
5. Compute the limit for $x \to 0$ of $sin(x)\,(1 - cos(2x))/x^3$
6. Compute the limit for the series $\sum_{i=1}^{\infty} \frac{(-1)^{n+1}}{n}$

Thank you and enjoy programming!