Lecture 5: Functions, Debugging

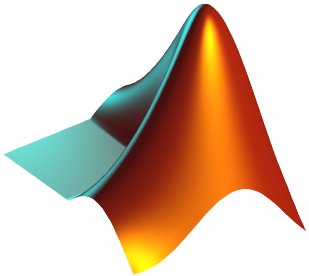
B0B17MTB – Matlab

Miloslav Čapek, Viktor Adler, Pavel Valtr, Michal Mašek, and Vít Losenický

Department of Electromagnetic Field
Czech Technical University in Prague
Czech Republic
matlab@elmag.org

October 21, 2019
Winter semester 2019/20

Outline



Functions in MATLAB

- More efficient, more transparent, and faster than scripts.
- Defined input and output, comments → function header is necessary.
- Can be called from Command Window, script, or from other function (in all cases the function has to be accessible).
- Each function has its own Workspace created upon the function's call and terminated with the last line of the function.
- All principles of programming covered at earlier stages of the course (memory allocation, data type conversion, indexing, etc.) apply also to MATLAB functions.
  - In case of overloading a built-in function, *e.g.*, defining your own variable/function `sum`, `builtin` function is applicable.

Function Header

- ~~Has to be the first line of a standalone file!~~ MATLAB 2017a+.
- Square brackets `[]` for one output parameter are not mandatory.
- Function header has the following syntax.

```
function [out1, out2, ..] = functionName(in1, in2, ..)
```

keyword — function's output parameters — function's name — function's input parameters

  - `functionName` has to follow the same rules as a variable's name.
  - `functionName` cannot be identical to any of its parameters' name.
  - `functionName` is usually typed as `lowerCamelCase` or using underscore character (`my_function`).

Simple Example of a Function

- Any function in MATLAB can be called with less input parameters or less output parameters than stated in the header.
  - For instance, consider following function:

```
function [out1, out2, out3] = funcG(in1, in2, in3)
```

  - All following calling syntaxes are correct:

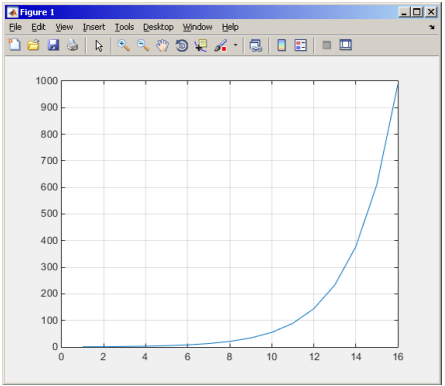
```
[out1, out2] = funcG(in1)
funcG(in1, in2, in3)
[out1] = funcG(in1, in2, in3)
[~, ~, out3] = funcG(in1, in2)
```

  - Another header definitions:

```
function funcA       % unusual, but possible, without input and output
function funcB(in1, in2)    % e.g. function with GUI output, print, etc.
function out1 = funcC       % data preparation, pseudorandom data, etc.
function out1 = funcD(in1)      % "proper" function
function [out1, out2] = funcE(in1, in2)     % proper function with more
    parameters
```

Calling MATLAB Function

```
>> f = fibonacci(1000); %
     calling from command prompt
>> plot(f); grid on;
```




```
function f = fibonacci(limit)
%% Fibonacci sequence
f = [1 1]; pos = 1;
while f(pos) + f(pos + 1) < limit
    f(pos + 2) = f(pos) + f(pos + 1);
    pos = pos + 1;
end
end
```

- MATLAB carries out commands sequentially
  - Input parameter: `limit`
  - output parameter: `f`
  - drawbacks:
    - Input is not validated (any input can be entered).
    - Matrix `f` is not allocated, *i.e.*matrix keeps growing (slow)

Comments Inside a Function

Function help displayed upon: `>> help myFcn1`.

1st line (so called H1 line), this line is searched for by `lookfor`. Usually contains function's name in capital characters and a brief description of the purpose of the function.

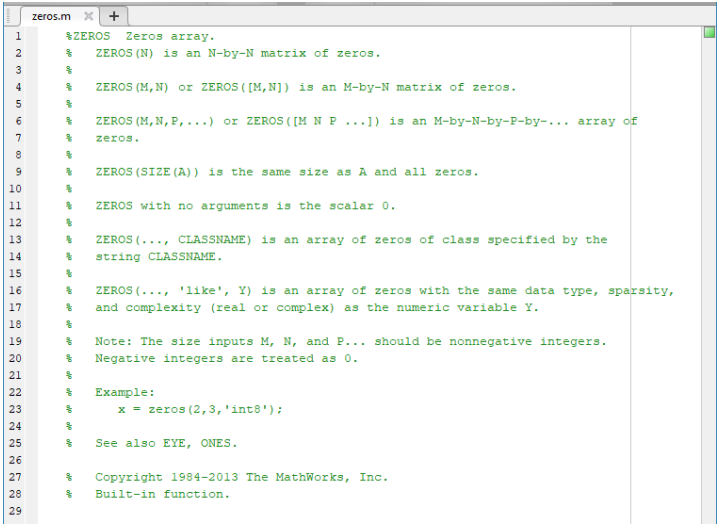
```
function [dataOut, idx] = myFcn1(dataIn, method)
%MYFCN1: Calculates...
% syntax, description of input, output,
% expamples of function s call, author,
    version
% other similar functions, other parts of help

matX = dataIn(:, 1);
sumX = sum(matX); % sumation
%% displaying the result:
disp(num2str(sumX));
```

DO COMMENT!!

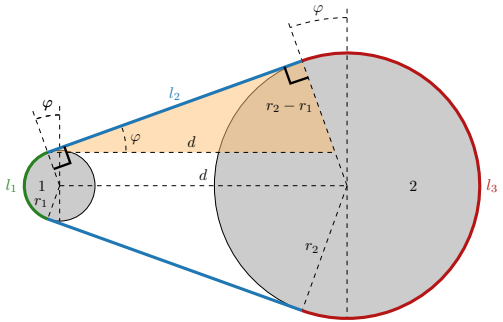
Comments significantly improve transparency of functions' code.

Function Documentation: Example



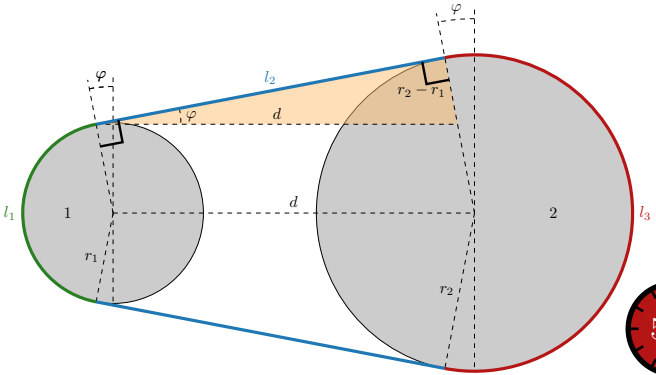
Simple Example of a Function I.

- Propose a function to calculate length of a belt between two wheels.
  - Diameters of both wheels r_1, r_2, are known as well as their distance d (= function's inputs).
  - Sketch a draft, analyze the situation and find out what you need to calculate.
  - Test the function for some scenarios and verify results.
  - Comment the function, apply commands `doc`, `lookfor`, `help`, `type`.



Simple Example of a Function II.

- Total length is:
- Known diameters → recalculate to radii:
- l_2 to be determined using Pythagorean theorem:
- Analogically for φ:
- Finally, the arches:
  - Verify your results:
    $d_1 = 2, d_2 = 2, d = 5$
    $L = 2\pi + 2 \cdot 5 \approx 16.2832.$



Simple Example of a Function III.

Workspace of a Function

- Each function has its own workspace.

```
clear
d1 = 2;
d2 = 2;
d = 5;
L1 = band_wheel(d1, d2, d);
d = 10;
L2 = band_wheel(d1, d2, d);
```

Base Workspace

A	NaN	double	1x1
t	1	double	1x1
x	[1 2]	double	1x2

Data Space of a Function

- When calling a function:
  - Input variables are not copied into workspace of the function, they are just made accessible for the function (*call-by-reference*, *copy-on-write technique*) unless they are modified.
  - If an input variable is modified inside the function, its value is copied into the function's workspace.
  - If an input variable is also used as an output variable, it is immediately copied.
  - Beware of using large arrays as input parameters to the functions which are then modifying them. With respect to memory demands and calculation speed-up, it is preferred to take the necessary elements out of the array and take care of them separately.
- In the case of the recursive calling of a function, the function's workspace is created for each call.
  - Pay attention to the excessive increase in the number of workspaces.
- Sharing variables between multiple workspaces (global variables) is generally not recommended and can be avoided in most cases.

Function execution

- When is function terminated?
  - MATLAB interpreter reaches the last line.
  - Interpreter comes across the keyword `return`.
  - Interpreter encounters an error (can be evoked by `error` as well).
  - On pressing CTRL+C.

```
function res = myFcn2(matrixIn)

if isempty(matrixIn)
    error('matrixInCannotBeEmpty');
end
normMat = matrixIn - max(max(matrixIn));

if matrixIn == 5
    res = 20;
    return
end
end
```

Number of Input and Output Variables I.

- Number of input and output variables is specified by functions **nargin** and **nargout**.
- These functions enable to design the function header in a way to enable variable number of input/output parameters.

```
function [out1, out2] = myFcn3(in1, in2)
nArgsIn = nargin;
if nArgsIn == 1
    % do something
elseif nArgsIn == 2
    % do something
else
    error('Bad inputs!');
end
% computation of out1
if nargout == 2
    % computation of out2
end
end
```

Number of Input and Output Variables III.

Modify the function `fibonacci.m` to enable variable input/output parameters:

- It is possible to call the function without input parameters.
  - The series is generated in the way that the last element is less than 1000.
- It is possible to call the function with one input parameter `in1`.
  - The series is generated in the way that the last element is less than `in1`.
- It is possible to call the function with two input parameters `in1`, `in2`.
  - The series is generated in the way that the last element is less than `in1` and at the same time the first 2 elements of the series are given by vector `in2`.
- It is possible to all the function without output parameters or with one output parameter.
  - The generated series is returned.
- It is possible to call the function with two output parameters.
  - The generated series is returned together with an object of class `Line`, which is plotted in a graph: `hLine = plot(f);`.

Number of Input and Output Variables IV.

Syntactical Types of Functions

Function type	Description
main	The only one in the m-file visible from outside, above principles apply.
local	All functions in the same file except the main function, accessed by the main function, has its own workspace, can be placed into `[private]` folder to preserve the private access, function in script file (2016b+).
nested	The function is placed inside the main function or local function, sees the WS of all superior functions.
handle	Function reference (`mySinX = @sin`).
anonymous	Similar to handle functions (`myGFcn = @(x)sin(x)+ cos(x)`).
OOP	Cass methods with specific access, static methods.

- Any function in MATLAB can launch a script which is then evaluated in the workspace of the function that launched it, not in the base workspace of MATLAB (as usual).
- The order of local functions is not important (logical connection!).
- Help of local functions is not accesible using `help`.

Local Functions I.

- Local functions launched by a main function. They
  - can (should) be terminated with the keyword `end`,
  - are used for repeated tasks inside the main function (helps to simplify the problem by decomposing it into simple parts),
  - "see" each other and have their own workspace,
  - are often used to process graphical elements' events (callbacks) when developing GUI.

```
function PRx = getRxPower(R, PTx, GAnt, freq)
% main function body
FSL = computeFSL(R, freq); % free-space loss
PRx = PTx + 2*GAnt - FSL; % received power
end

function FSL = computeFSL(R, freq)
% local function body
c0 = 3e8;
lambda = c0./freq;
FSL = 20*log10(4*pi*R./lambda);
end
```

Local Functions II.

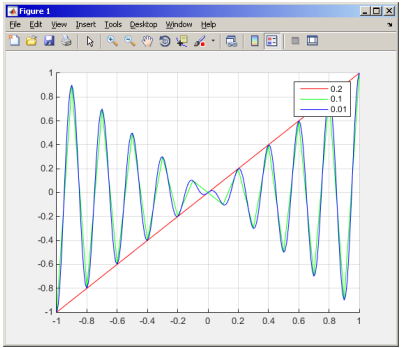
- Local functions launched by a script (new from R2016b). They
  - have to be at the end of file,
  - have to be terminated with the keyword `end`,
  - "see" each other and have their own workspace,
  - are not accessible outside the script file.

```
clear;
% start of script
r = 0.5:5; % radii of circles
areaOfCirles = computeArea(r);

function A = computeArea(r)
% local function in script
A = pi*r.^2;
end
```

Local Functions III.

- Plot function $f(x) = x \sin\left(\frac{\pi}{2}(1 + 20x)\right)$ in the interval $x \in [-1, 1]$ with step 0.2, 0.1 and 0.01.
- Compare the results!



Nested Functions

- Nested functions are placed inside other functions.
  - In enables us to use workspace of the parent function and to efficiently work with (usually small) workspace of the nested function.
  - Functions can not be placed inside conditional/loop control statements (`if-elseif-else`/`switch-case`/`for`/`while`/`try-catch`).

```
function x = A(p)
% single
% nested function
...
    function y = B(q)
     ...
    end
...
end
```

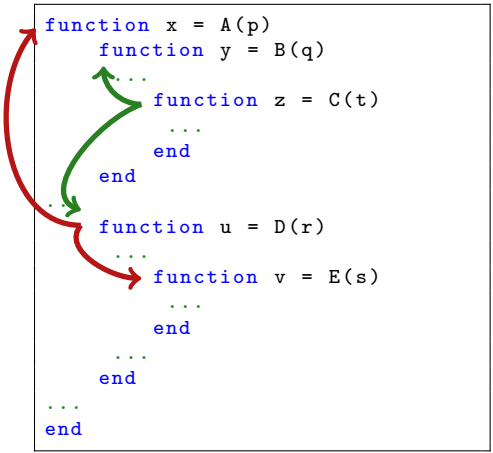
```
function x = A(p)
% more
% nested functions
...
    function y = B(q)
     ...
    end

    function z = C(r)
     ...
    end
...
end
```

```
function x = A(p)
% multiple
% nested function
...
    function y = B(q)
     ...
       function z = C(r)
        ...
       end
     ...
    end
...
end
```

Nested Functions: Calling

- Apart from its workspace, nested functions can also access workspaces of all functions it is nested in.
- Nested functions can be called from:
  - its parent function,
  - nested functions on the same level of nesting,
  - function nested in it.
- It is possible to create handle to a nested function.
  - See later.

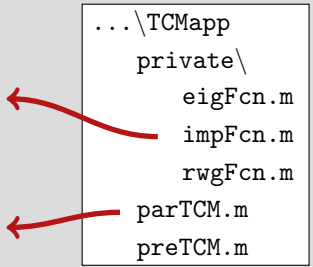


Private Functions

- They are basically the local functions, and they can be called by all functions placed in the root folder.
- Reside in sub-folder `[private]` of the main function.
- Private functions can be accessed only by functions placed in the folder immediately above that private sub-folder.
  - `[private]` is often used with larger applications or in the case where limited visibility of files inside the folder is desired.

These functions can be called by `parTCM.m` and `preTCM.m` only.

`parTCM.m` calls functions in `[private]`.



Handle Functions

- It is not a function as such.
- Handle = reference to a given function.
  - Properties of a handle reference enable to call a function that is otherwise not visible.
  - Reference to a handle (here `fS`) can be treated is a usual way.
- Typically, handle references are used as input parameters of functions.

```
>> fS = @sin; % handle creation
>> fS(pi/2)
ans =
     1
```

Anonymous functions I.

- Anonymous functions make it possible to create handle reference to a function that is not defined as a standalone file.
  - The function has to be defined as one executable expression.

```
>> sqr = @(x) x.^2; % create anonymous function (handle)
>> res = sqr(5);    % x ~ 5, res = 5^2 = 25;
```

  - Anonymous function can have more input parameters.

```
>> A = 4; B = 3; % parameters A,B have to be defined
>> sumAxBy = @(x, y) (A*x + B*y); % function definition
>> res2 = sumAxBy(5,7);    % x = 5, y = 7
% res2 = 4*5+3*7 = 20+21 = 41
```

  - Anonymous function stores variables required as well as prescription.
- `>> doc Anonymous Functions`

```
>> Fcn = @(hndl, arg) (hndl(arg))
>> res = Fcn(@sin, pi)
```

```
>> A = 4;
>> multAx = @(x) A*x;
>> clear A
>> res3 = multAx(2);
% res3 = 4*2 = 8
```

Anonymous functions II.

- Create anonymous function $\mathbf{A}(p) = [A_1(p) \quad A_2(p) \quad A_3(p)]$ so that

$$A_1(p) = \cos^2(p),$$
$$A_2(p) = \sin(p) + \cos(p),$$
$$A_3(p) = 1.$$

- Calculate and display its components for range $p = [0, 2\pi]$.

- Check the function $\mathbf{A}(p)$ with MATLAB built-in function functions, *i.e.*, `functions(A)`.

Functions: Advanced Techniques

- In the case the number of input or output parameters is not known one can use `varargin` and `varargout`.
  - Function header has to be modified.
  - Input/output variables have to be obtained from `varargin`/`varargout`.

```
function [parOut1, parOut2] = funcA(varargin)
%% variable number of input parameters
```

```
function varargout = funcB(parIn1, parIn2)
%% variable number of output parameters
```

```
function varargout = funcC(varargin)
%% variable number of input and output parameters
```

```
function [parOut1, varargout] = funcC(parIn1, varargin)
%% variable number of input and output parameters
```

`varargin` Function

- Typical usage: functions with many optional parameters/attributes.
  - *e.g.*, visualisation (functions like `stem`, `surf` etc. include `varargin`)
- Variable `varargin` is always of type `cell` (*see later*), even when it contains just single item.
- Function `nargin` in the body of a function returns the number of input parameters upon the function's call.
- Function `nargin(fx)` returns number of input parameters in function's header.
  - When `varargin` is used in function's header, returns negative value.

```
function plot_data(varargin)
nargin
celldisp(varargin)

par1 = varargin{1};
par2 = varargin{2};
```

`varargout` Function

- Variable number of output variables.
- Principle analogical to `varargin` function.
  - Bear in mind that function's output variables are of type `cell`.
- Used occasionally.

```
function [s, varargout] = sizeout(x)
nout = max(nargout, 1) - 1;
s = size(x);
for k = 1:nout
   varargout{k} = s(k);
end
```

```
>>  [s, rows, cols] = sizeout(rand(4, 5, 2))
% s = [4 5 2], rows = 4, cols = 5
```

Advanced Anonymous Functions

- Inline conditional:

```
>> iif = @(varargin) varargin{2*find([varargin{1:2:end}], 1, 'first')}();
```

  - Usage:

```
>> min10 = @(x) iif(any(isnan(x)), 'Don''t use NaNs', ...
                    sum(x) > 10, 'This is ok', ...
                    sum(x) < 10, 'Sum is low')
>> min10([1 10])  % ans = 'This is ok'
>> min10([1 nan]) % ans = 'Don't use NaNs'
```

- Map:

```
>> map = @(val, fcns) cellfun(@(f) f(val{:}), fcns);
```

  - Usage:

```
>> x = [3 4 1 6 2];
>> values = map({x}, {@min, @sum, @prod})
>> [extreme, indices] = map({x}, {@min, @max})
```

Variable Number of Input Parameters

- Input arguments are usually in pairs.
- Example of setting of several parameters to `line` object.
- For all properties see `>> doc line`.

Property	Value
`Color`	[RGB]
`LineWidth`	0.1 - ...
`Marker`	'o', '*', 'x', ...
`MarkerSize`	0.1 - ...
...	

```
>> plot_data(magic(3),...
    'Color',[.4 .5 .6],'LineWidth',2);
>> plot_data(sin(0:0.1:5*pi),...
    'Marker','*','LineWidth',3);
```

```
function plot_data(data, varargin)
if isnumeric(data) && ~isempty(data)
    hndl = plot(data);
else
    fprintf(2, ['Input variable ''data''',
        ...
         'is not a numerical variable.']);
    return;
end

while length(varargin) > 1
    set(hndl, varargin{1}, varargin{2});
    varargin(1:2) = [];
end
end
```

Output Parameter `varargout`

- Modify the function `fibonacciFcn.m` so that it has only one output parameter `varargout` and its functionality was preserved.

Expression Evaluated in Another Workspace

- Function `evalin` ("evaluate in") can be used to evaluate an expression in a workspace that is different from the workspace where the expression exists.
- part from current workspace, other workspaces can be used as well
  - `'base'`: base workspace of MATLAB.
  - `'caller'`: workspace of parent function (from which the function was called).
- Can not be used recursively.

```
>> clear; clc;
>> A = 5;
>> res = eval_in
% res = 12.7976
```

```
function out = eval_in
%% no input parameters (A isn t known here)

k   = rand(1,1);
out = evalin('base', ['pi*A*', num2str(k)]);
end
```

Recursion

- MATLAB supports recursion (function can call itself).
  - Recursion is part of some useful algorithms (*e.g.*Adaptive Simpsons Method of integration).
- MATLAB ver. R2014b and older:
  - The number of recursions is limited by 500 by default.
  - The number of recursions can be changed, or get current setting:

```
>> set(0, 'RecursionLimit', 200)
>> get(0, 'RecursionLimit')
% ans = 200
```

- MATLAB ver. R2015a and newer: recursion calling works until context stack is not full.
  - Every calling creates new function's workspace!
- Any recursive algorithm can be expressed as an iterative algorithm[1].

[1] https://en.wikipedia.org/wiki/The_Art_of_Computer_Programming
https://en.wikipedia.org/wiki/Church%E2%80%93Turing_thesis

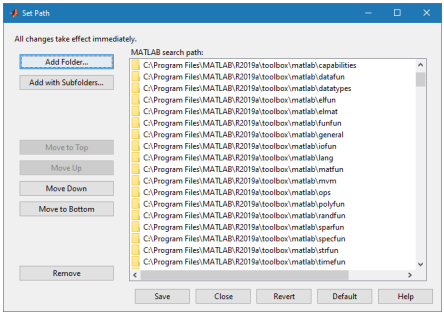
Number of Recursion Steps

- Write a simple function that is able to call itselg; input parameter is `rek = 0` which is increased by 1 with each recursive step.
  - Display the increase of the value of `rek`.
  - At what number does the increase stop.
  - Thin over in what situations the recursion is necessary ...

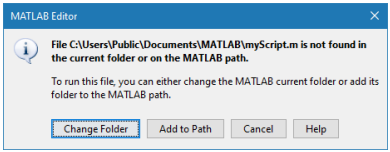
MATLAB Path

- List of directories seen by MATLAB:

```
>> path
```



- For more see `>> doc path`.
- `addpath`: adds folder to path.
- `rmpath`: removes folder from path



Namespace

- Namespace is a simple way how to create a package.
- Useful for organizing functions into logical blocks.

- Create a folder starting with "+".
- Place selected functions inside.

```
...\+foo
    bar.m
    baz.m
```

- Any function is accessed using namespace prefix.

```
foo.bar();
foo.baz();
```

Order of Function Calling

- How MATLAB searches for a function (simplified):
  1. It is a variable.
  2. Function imported using `import`.
  3. Nested or local function inside given function.
  4. Private function.
  5. Function (method) of a given class or constructor of the class.
  6. Function in given folder.
  7. Function anywhere within reach of MATLAB (`path`).
- inside a given folder is the priority of various suffixes as follows:
  - built-in functions,
  - `mex` functions,
  - `p`-files,
  - `m`-files.
- `doc Function Precedence Order`

Class `inputParser` I.

- Enables to easily test input parameters of a function.
- It is especially useful to create functions with many input parameters with pairs `'parameter', value`.
  - Very typical for graphical functions.

```
>> x = -20:0.1:20;
>> fx = sin(x)./x;
>> plot(x, fx, 'LineWidth', 3, 'Color', [0.3 0.3 1], 'Marker', 'd',...
 'MarkerSize', 10, 'LineStyle', ':')
```

- Method `addParameter` enablesto inser optional parameter.
  - Initial value of the parameter has to be set.
  - The function for validity testing is not required.
- Method `addRequired` defines name of mandatory parameter.
  - On function call it always has to be entered at the right place.

Class `inputParser` II.

- Following function plots a circle or a square of defined size, color and line width.

```
function drawGeom(dimension, shape, varargin)
p = inputParser; % instance of inputParser
p.CaseSensitive = false; % parameters are not case sensitive
defaultColor = 'b'; defaultWidth = 1;
expectedShapes = {'circle', 'rectangle'};
validationShapeFcn = @(x) any(ismember(expectedShapes, x));
p.addRequired('dimension', @isnumeric); % required parameter
p.addRequired('shape', validationShapeFcn); % required parameter
p.addParameter('color', defaultColor, @ischar); % optional parameter
p.addParameter('linewidth', defaultWidth, @isnumeric) % optional parameter
p.parse(dimension, shape, varargin{:}); % parse input parameters

switch shape
   case 'circle'
      figure;
      rho = 0:0.01:2*pi;
      plot(dimension*cos(rho), dimension*sin(rho), ...
         p.Results.color, 'LineWidth', p.Results.linewidth);
      axis equal;
   case 'rectangle'
      figure;
      plot([0 dimension dimension 0 0], ...
         [0 0 dimension dimension 0], p.Results.color, ...
         'LineWidth', p.Results.linewidth)
      axis equal;
end
```

Function `validateattributes`

- Checks correctness of inserted parameter with respect to various criteria.
  - It is often used in relation with class `inputParser`.
  - Check whether matrix is of size 2×3, is of class `double` and contains positive integers only:

```
A = [1 2 3;4 5 6];
validateattributes(A, {'double'}, {'size',[2 3]})
validateattributes(A, {'double'}, {'integer'})
validateattributes(A, {'double'}, {'positive'})
```

  - It is possible to use notation where all tested classes and attributes are in one cell:

```
B = eye(3)*2;
validateattributes(B, {'double', 'single', 'int64'},...
   {'size',[3 3], 'diag', 'even'})
```

- For complete list of options `>> doc validateattributes`.

Original Names of Input Variables

- Function `inputname` makes it possible to determine names of input parameters ahead of function call.
  - Consider folowing function call:

```
>> y = myFunc1(xdot, time, sqrt(25));
```

  - And then inside the function:

```
function output = myFunc1(par1, par2, par3)

% ...
p1str = inputname(1);     % p1str = 'xdot';
p2str = inputname(2);     % p2str = 'time';
P3str = inputname(3);     % p3str = '';
% ...
```

Function and m-file Dependence

- Identify all the files and functions required for sharing your code.
- Function `matlab.codetools.requiredFilesAndProducts`
  - returns user files and products necessary for evaluation of a function/script,
  - does not return which are part of required products.
- *e.g.*, dependencies of Homework1 checker

```
[fList, plist] = matlab.codetools.requiredFilesAndProducts('homework1')
fList =
  1x5 cell array
    {'C:\Homework1\homework1.m'}
    {'C:\Homework1\problem1A.m'}
    {'C:\Homework1\problem1B.m'}
    {'C:\Homework1\problem1C.m'}
    {'C:\Homework1\problem1D.m'}
plist =
  struct with fields:
              Name: 'MATLAB'
           Version: '9.4'
     ProductNumber: 1
           Certain: 1
```

Function `why`

- It is a must to try that one!
  - Try `help why`.
  - Try to find out how many answers exist.

Script `startup.m`

- Script **`startup.m`**:
  - Is always executed at MATLAB start-up.
  - It is possible to put your predefined constant and other operations to be executed(loaded) at MATLAB start-up.
- Location (use **`which startup`**):
  - `...\Matlab\R20XXx\toolbox\local\startup.m`
- Change of base folder after MATLAB start-up:

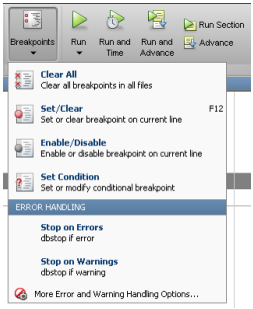
```
%% script startup.m in ..\Matlab\Rxxx\toolbox\local\
clc;
disp('Workspace is changed to: ');
cd('d:\Data\Matlab');
cd
disp(datestr(now, 'mmmm dd, yyyy HH:MM:SS.FFF AM'))
```

Debugging I.

- *Bug* → *debugging.*
- We distinguish:
  - Semantic errors ("logical" or "algorithmic" errors).
    - Usually difficult to identify.
  - Syntax errors ("grammatical" errors).
    - Pay attention to the content of error messages – it makes error elimination easier.
  - Unexpected events (see later).
    - For example, a problem with writing to open file, not enough space on disk etc.
  - Rounding errors (everything is calculated as it should be but the result is wrong anyway).
    - It is necessary to analyze the algorithm in advance, to determine the dynamics of calculation etc.
- Software debugging and testing is an integral part of software development.
  - Later we will discuss the possibilities of code acceleration using MATLAB profile.

Debugging II.

- We first focus on semantic and syntax errors in scripts.
  - We always test the program using test-case where the result is known.
- Possible techniques:
  - Using functions `who`, `whos`, `keyboard`, `disp`.
  - Using debugging tools in MATLAB editor (illustration).
    - Using MATLAB built-in debugging functions.



MATLAB Functions

`dbclear`	Clear breakpoints
`dbcont`	Resume execution
`dbdown`	Reverse workspace shift performed by `dbup`, while in debug mode
`dbquit`	Quit debug mode
`dbstack`	Function call stack
`dbstatus`	List all breakpoints
`dbstep`	Execute one or more lines from current breakpoint
`dbstop`	Set breakpoints for debugging
`dbtype`	List text file with line numbers
`dbup`	Shift current workspace to workspace of caller, while in debug mode
`checkcode`	Check MATLAB code files for possible problems
`keyboard`	Input from keyboard
`mlintrpt`	Run `checkcode` for file or folder, reporting results in browser

Useful Functions for Script Generation

- The function `keyboard` stops execution of the code and gives control to the keyboard.
  - The function is widely used for code debugging as it stops code execution at the point where any doubts about the code functionality exist

```
K>>
```

  - `keyboard` status is indicated by `K>>` (K appears before the prompt).
  - The keyboard mode is terminated by `dbcont` or press F5 (Continue).
- Function `pause` halts code execution.
  - `pause(x)` halts code execution for x seconds.

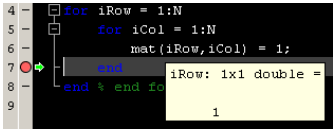
```
% code; code; code
pause;
```

- See also: `echo`, `waitforbuttonpress`.
  - Special purpose functions.

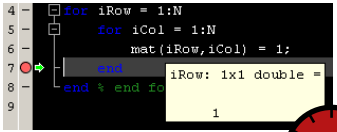
Debugging I.

- For the following piece of code:

```
clear; clc;
N    = 5e2;
mat  = nan(N,N);
for  iRow = 1:N
     for  iCol = 1:N
          mat(iRow,iCol) = 1;
     end % end for
end % end for
```



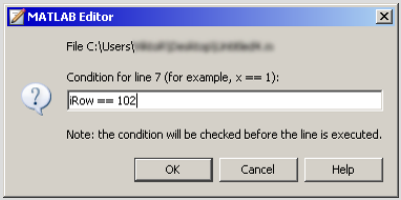
- Use MATLAB editor to:
  - set breakpoint (click on dash next to the line number),
  - run the script (F5),
  - check the status of variables (`keyboard` mode or hover over variable's name),
  - keep on tracing the script.
    - Find difference between Continue and Step (F10)?

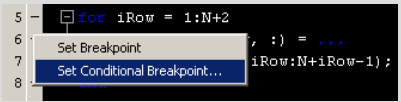


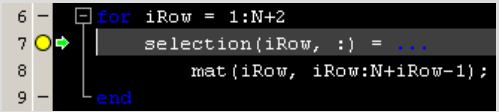
Advanced Debugging

- Conditional breakpoints:
  - Serve to suspend the execution of code when a condition is fulfilled.
    - Sometimes, the set up of the correct condition is not an easy task ...
  - Easier to find errors in loops.
    - Code execution can be suspended in a particular loop.
  - The condition may be arbitrary evaluable logical expression.

```
% code with an error
clear; clc;
N   = 100;
mat = magic(2*N);
selection = zeros(N, N);
for iRow = 1:N+2
    selection(iRow, :) = ...
        mat(iRow, iRow:N+iRow-1);
end
```







Selected Hints for Code Readability I.

- Use indention of loop's body, indention of code inside conditions(TAB).
  - Size of indention can be adjusted in **preferences** (usually 3 or 4 spaces).
- Use "positive" conditions.
  - *i.e.*use **isBigger** or **isSmaller**, not **isNotBigger** (can be confusing).
- Complex expressions with logical and relation operators should be evaluated separately → higher readability of code.
  - Compare:

```
if (val>lowLim)&(val<upLim)&~ismember(val,valArray)
    % do something
end
```

```
isValid = (val > lowLim) & (val < upLim);
isNew   = ~ismember(val, valArray);
if isValid & isNew
    % do something
end
```

Selected Hints for Code Readability II.

- Code can be separated with a blank line to improve clarity.
- Use two lines for separation of blocks of code.
  - Alternatively use cells or commented lines `% ----------`, etc.
- Consider to use of spaces to separate operators (`=`, `&`, —).
  - To improve code readability:

```
(val>lowLim)&(val<upLim)&~ismember(val,valArray)
```

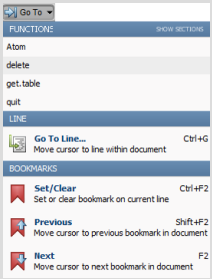
  - vs.

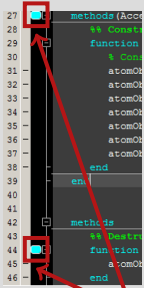
```
(val > lowLim) & (val < upLim) & ~ismember(val, valArray)
```

- In the case of nesting use comments placed after `end`.

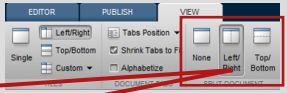
Useful Tools for Long Functions

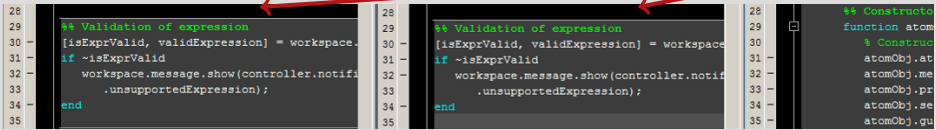
- Bookmarks:
  - CTRL+F2 (add/remove bookmark),
  - F2 (next bookmark),
  - SHIFT+F2 (previous bookmark).
- Go to ...
  - CTRL+G (go to line).
- Long files can be split.
  - Same file can be opened *e.g.*twice.





bookmarks

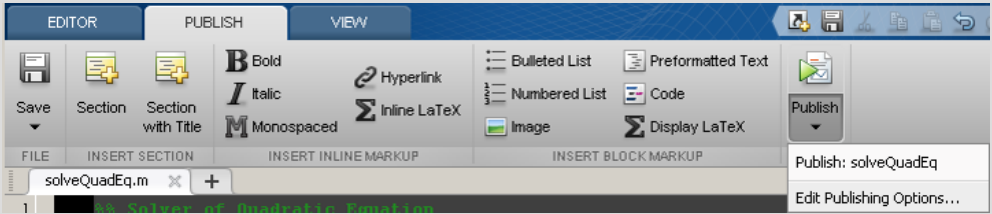




User Scripts and Functions

Function `publish` I.

- Serves to create script, function or class documentation.
- Provides several output formats (html, doc, ppt, LaTeX, ...).
- Help creation (`>> doc my_fun`) directly in the code comments!
  - Provides wide scale of formatting properties(titles, numbered lists, equations, graphics insertion, references, ...).
- Enables to insert print screens into documentation.
  - Documented code is implicitly launched on publishing.
- Supports documentation creation directly from editor menu:



Function `publish` II.

```
%% Solver of Quadratic Equation
% Function *solveQuadEq* solves quadratic equation.
%% Theory
% A quadratic equation is any equation having the form
% $ax^2+bx+c=0$
% where |x| represents an unknown, and |a|, |b|, and |c|
% represent known numbers such that |a| is not equal to 0.
%% Head of function
% All input arguments are mandatory!
function x = solveQuadEq(a, b, c)
%%
% Input arguments are:
%%
% * |a| - _qudratic coefficient_
% * |b| - _linear coefficient_
% * |c| - _free term_
%% Discriminant computation
% Gives us information about the nature of roots.
D = b^2 - 4*a*c;
%% Roots computation
% The quadratic formula for the roots of the general
% quadratic equation:
%
% $$x_{1,2} = \frac{ - b \pm \sqrt D }{2a}.$$
%
% Matlab code:
%%
x(1) = (-b + sqrt(D))/(2*a);
x(2) = (-b - sqrt(D))/(2*a);
%%
% For more information visit <http://elmag.org>.
```

publish

Solver of Quadratic Equation

Function **solveQuadEq** solves quadratic equation.

Contents

- Theory
- Head of function
- Discriminant computation
- Roots computation

Theory

A quadratic equation is any equation having the form $ax^2 + bx + c = 0$ where x represents an unknown, and a, b, and c represent known numbers such that a is not equal to 0.

Head of function

All input arguments are mandatory!

```
function x = solveQuadEq(a, b, c)
```

Input arguments are:

- a - *qudratic coefficient*
- b - *linear coefficient*
- c - *free term*

Discriminant computation

Gives us information about the nature of roots.

```
D = b^2 - 4*a*c;
```

Roots computation

The quadratic formula for the roots of the general quadratic equation:

$$x_{1,2} = \frac{-b \pm \sqrt{D}}{2a}.$$

Matlab code:

```
x(1) = (-b + sqrt(D))/(2*a);
x(2) = (-b - sqrt(D))/(2*a);
```

For more information visit http://elmag.org/matlab.

Exercises

Exercise I.

- Using `integral` function calculate integral of a function $G = \int g(x)\,\mathrm{d}t$ in the interval $t \in [0, 1]\,\mathrm{s}$. The function has following time dependency, where $f = 50\mathrm{Hz}$.

$$g(t) = 10\cos(2\pi f t) + 5\cos(4\pi f t)$$

- Solve the problem using handle function.

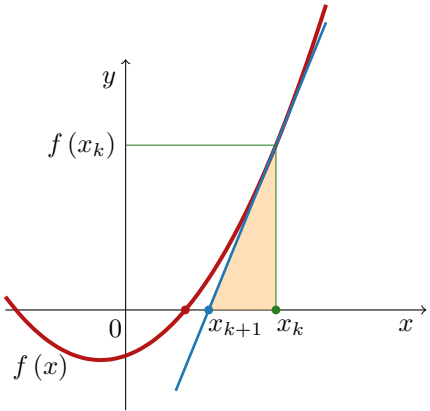
- Solve the problem using anonymous function.

Exercise II.a

- Find the unknown x in equation $f(x) = 0$ using Newton's method.
- Typical implementation steps:
  - Mathematical model.
    - Size the problem, its formal solution.
  - Pseudocode.
    - Layout of consistent and efficient code.
  - MATLAB code.
    - Transformation into MATLAB's syntax.
  - Testing.
    - Usually using a problem with known (analytic) solution.
    - Try other examples ...

Exercise II.b

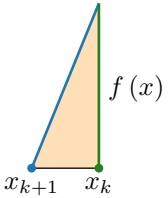
- Find the unknown x in equation of type $f(x) = 0$.
  - Use Newton's method.
- Newton's method:



$$f'(x) = \frac{\Delta f}{\Delta x} \approx \frac{\mathrm{d}f}{\mathrm{d}x}$$

$$f'(x) = \frac{\Delta f}{\Delta x} = \frac{f(x_k - 0)}{x_k - x_{k+1}}$$

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$



Exercise II.c

- Find the unknown x in equation of type $f(x) = 0$.
- Pseudocode:

Algorithm 1: Pseudocode of an implementation of the Newton's Method

while $|(x_{k+1} - x_k)/x_k| \geq error$ and simultaneously $k < 20$ **do**

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

display $[k \quad x_{k+1} \quad f(x_{k+1})]$

$k = k + 1$

end while

- Pay attention to correct condition of the (`while`) cycle.
- Create a new function to evaluate $f(x)$ and $f'(x)$.
- Use following numerical difference scheme to calculate $f'(x)$:

$$f'(x) \approx \Delta f = \frac{f(x_k + \Delta) - f(x_k - \Delta)}{2\Delta}.$$

Exercise II.d

- Find the unknown x in equation of type $f(x) = 0$.
  - Implement the above metnod in MATLAB to find the unknown x in $x^3 + x - 3 = 0$.
  - This method comes in the form of a script calling following function:

```
function fx = optim_fcn(x)
fx = x^3 + x - 3;
end
```

```
clear; close all; clc;
% enter variables
% xk, xk1, err, k, delta

while cond1 and_simultaneously cond2
    % get xk from xk1
    % calculate f(xk)
    % calculate df(xk)
    % calculate xk1
    % display results
    % increase value of k
end
```

Exercise II.e

- What are the limitations of Newton's method.
  - In relation with existence of multiple roots.
- Is it possible to apply the method to complex values of x?

Exercise III.

- Modify Newton's method in the way that the polynomial is entered in the form of a handle function.
  - Verify the code by finding roots of a following polynomials: $x - 2 = 0, \quad x^2 = 1$.
  - Verify the result using function `roots`.

Exercise IV.a

- Expand exponential function using Taylor series:
  - In this case is is in fact Maclaurin series (expansion about 0).

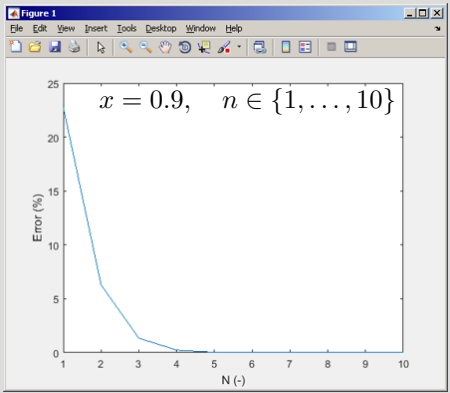
$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \dots$$

    - Compare with result obtained using `exp(x)`.
    - Find out the deviation in % (what is the base, *i.e.*, 100%?).
    - Find out the order of expansion for deviation to be lower than 1%.
- Implement the code as a function.
  - Choose the appropriate name for the function.
  - Input parameters of the functions are: `x` (function argument) and `N` (order of the series).
  - Output parameters are values: `f1` (result of the series), `f2` (result of `exp(x)`) and `err` (deviation in %).
  - Add appropriate comment to the function (H1 line, inputs, outputs).
  - Test the function!

Exercise IV.b

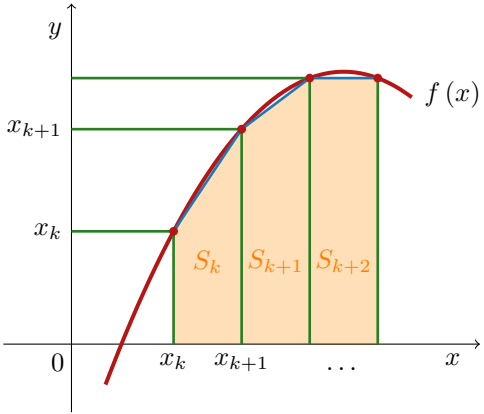
Exercise IV.c

- Create a script to call the above function (with various N).
  - Find out accuracy of the approximation for $x = 0.9, \quad n \in \{1, \ldots, 10\}$.
  - Plot the resulting progress of the accuracy (error as a function of n).

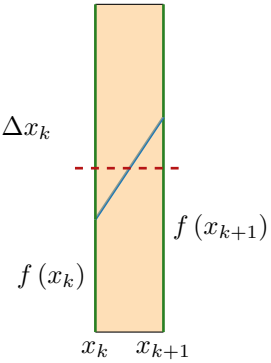


Example V.a

- Write a function that approximates definite integral by trapezoidal rule.
- Trapezoidal rule:



$$\int_a^b f(x)\,\mathrm{d}x \approx \approx \sum_{n=1}^{N} \frac{f(x_k) + f(x_{k+1})}{2} \Delta x_k$$



Example V.b

- Implement a function that approximates definite integral of a function given by handle function.
  - Choose the appropriate name for the function.
  - Input parameters are `f` (handle function), `a` (lower limit), `b` (upper limit) and `N` (number of divisions).
  - Output parameter is `I` (value of the integral).
  - Test the function.
- Compare the results with function `integral` for: $f(x) = \sqrt{2^x + 1}; \quad x \in [0, 5]$.

Questions?

B0B17MTB – Matlab
matlab@elmag.org

October 21, 2019
Winter semester 2019/20

Acknowledgement: Filip Kozak.