A0B17MTB – Matlab

Part #7

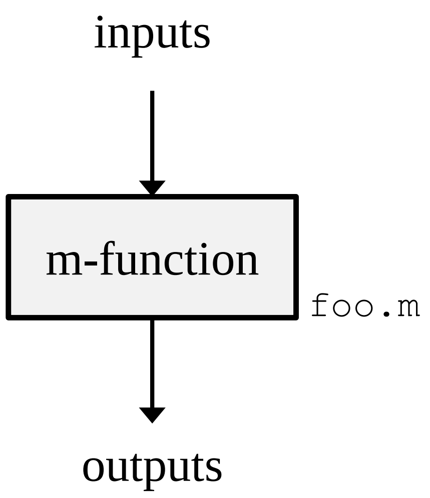
Miloslav Čapek
miloslav.capek@fel.cvut.cz

Viktor Adler, Pavel Valtr, Filip Kozák

Department of Electromagnetic Field
B2-634, Prague

Learning how to …

Functions



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 or from other function (in both cases the function has to be accessible)
- each function has its own work space created upon the function's call and terminated with the last line of the function

Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

Function types by origin

- built-in functions
  - not accessible for editing by the user, available for execution
  - optimized and stored in core
  - usually frequently used (elementary) functions
- Matlab library functions ([`toolbox`] directory)
  - subject-grouped functions
  - some of them are available for editing (not recommended!)
- user-created functions
  - fully accessible and editable, functionality not guaranteed
  - mandatory parts: function header
  - recommended parts of the function: function description, characterization of inputs and outputs, date of last editing, function version, comments

Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

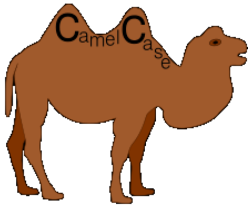
Function header

- ~~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` can't be identical to any of its parameters' name
- `functionName` is usually typed as `lowerCamelCase` or using underscore character (`my_function`)



Function header – examples

```
function functA
%FUNCTA - unusual, but possible, without input and output
```

```
function functB(parIn1)
%FUNCTB - e.g. function with GUI output, print etc.
```

```
function parOut1 = functC
%FUNCTC - data preparation, pseudorandom data etc.
```

```
function parOut1 = functD(parIn1)
%FUNCTD - „proper“ function
```

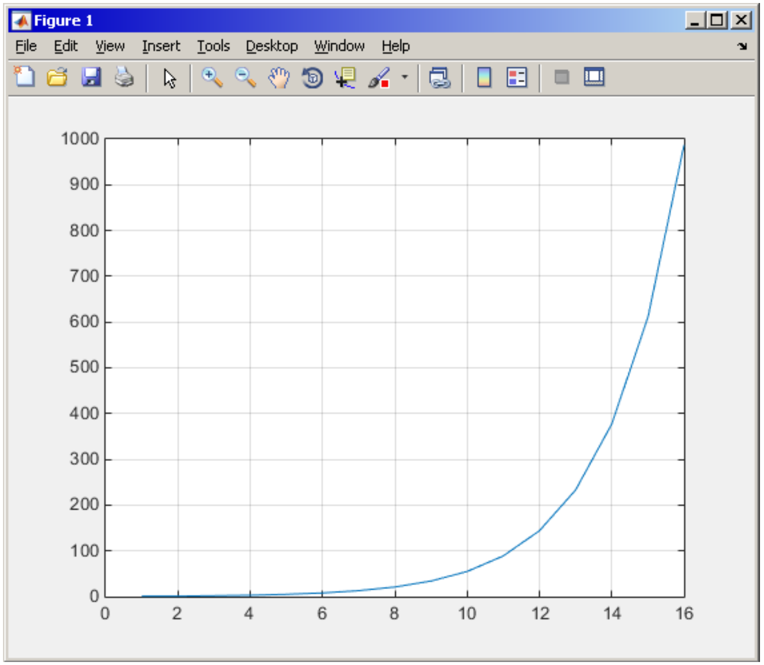
```
function parOut1 = functE(parIn1, parIn2)
%FUNCTE - proper function
```

```
function [parOut1, parOut2] = functF(parIn1, parIn2)
%FUNCTF - 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 <u>sequentially</u>
  - input parameter: `limit`
  - output variable: Fibonacci series `f`
  - <u>drawbacks:</u>
    - input is not treated (any input can be entered)
    - matrix `f` is not allocated, i.e. matrix keeps growing (slow)

Simple example of a function

- any function in Matlab can be called with less input parameters than stated in the header
- any function in Matlab can be called with less output parameters than stated in the header
  - for instance, consider following function:

```
function [parOut1, parOut2, parOut3] = functG(parIn1, parIn2, parIn3)
%FUNCTG - 3 inputs, 3 outputs
```

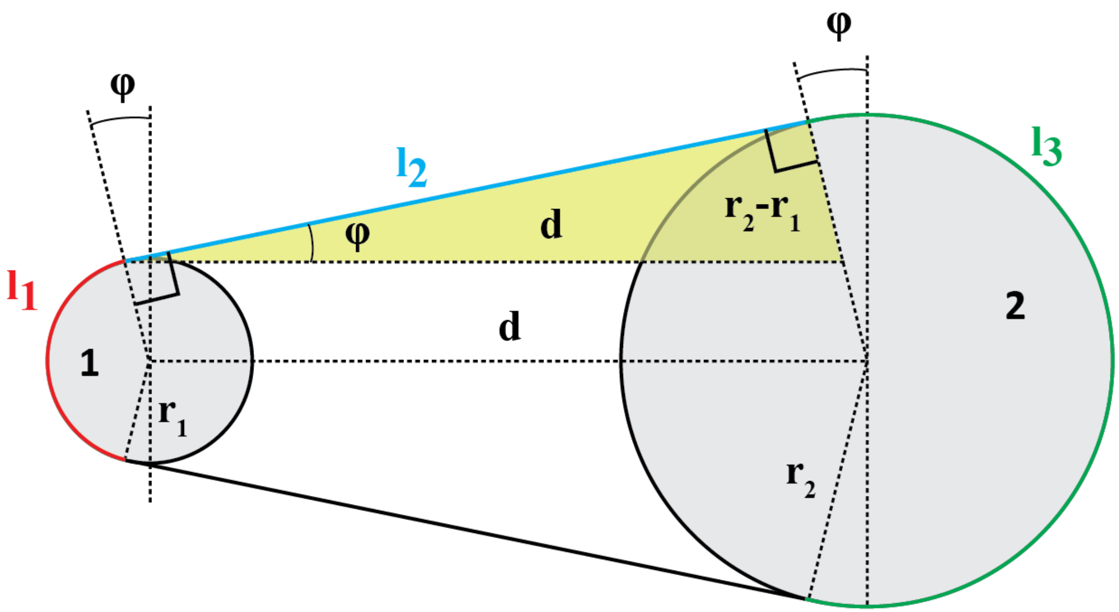
  - all following calling syntaxes are correct

```
>> [par01, par02]          = functG(pIn1, pIn2, pIn3)
>> [par01, par02, par03] = functG(pIn1)
>> functG(pIn1,pIn2,pIn3)
>> [par01, par02, par03] = functG(pIn1, pIn2, pIn3)
>> [par01, ~, par03] = functG(pIn1, [], pIn3)
>> [~, ~, par03] = functG(pIn1, [], [])
>> functG inputStr1 inputStr2
```

Simple example of a function

100 s

- propose a function to calculate length of a belt between two wheels
  - diameters of both wheels are known as well as their distance (= 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`



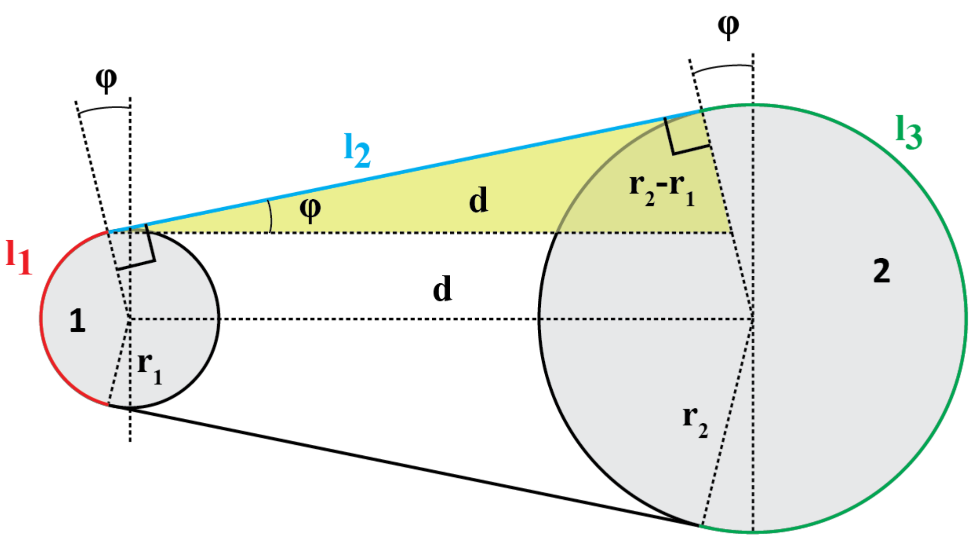
Simple example of a function

500 s ↑

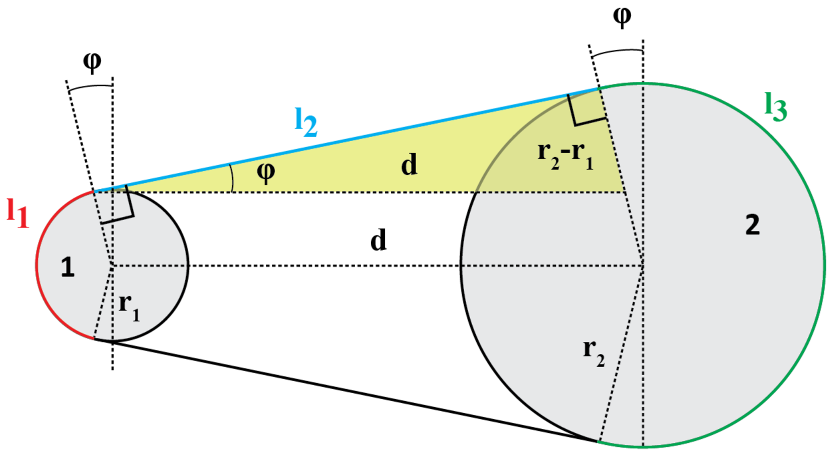
- total length is
- known diameters → recalculate to radiuses
- l_2 to be determined using Pythagorean theorem :

- Analogically for φ:
- and finally :

- verify your results using



Simple example of a function



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));
```

```
function pdetool(action, flag)
%PDETOOL PDE Toolbox graphical user interface (GUI).
%   PDETOOL provides the graphical user ...
```

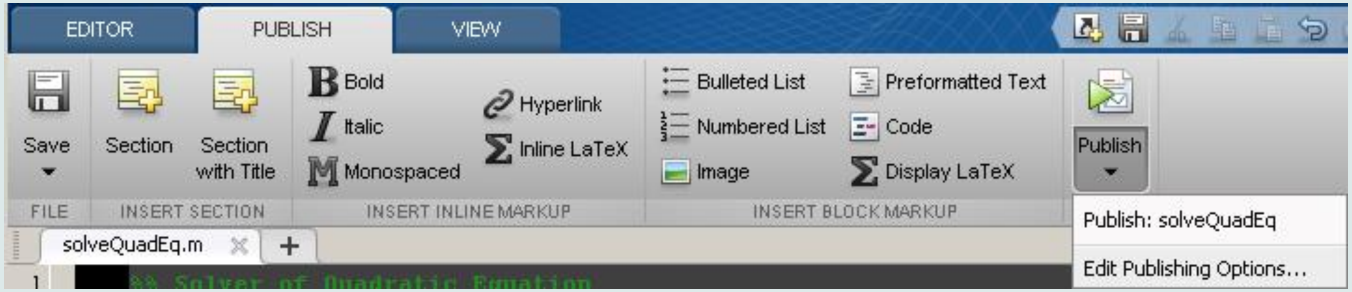
```
DO COMMENT!
% Comments significantly improve
% transparency of functions' code !!!
```

Function documentation – example

```
function Z = impFcn(f,MeshStruct,Zprecision)
%% impFcn: Calculates the impedance matrix
% -solver-
%
%  Syntax:
%    Z = impFcn(f,MeshStruct,Zprecision)
%
% impFcn version history:
%    ver. 1.0a
%    ver. 1.0b (8.8.2011)
%         default option (if nargin == 2) is Zprecision = true
%
%    Last update: 8.8.2013
%
% Notes:
% A)  (contains rwg3.m): Calculates the impedance matrix (includes infinite
%                        groud plane)
% B)
%   RHO_P(3,9,edgTotal)
%   RHO_M(3,9,edgTotal)
%
%   Temporary variables:
%   RP(3,9,EdgesTotal)
%
% C)  See: [1] Sergey N. Makarov: Antenna and EM Modeling with MATLAB
%     Copyright 2002 AEMM. Revision 2002/03/05 and ČVUT-FEL 2007-2010
%
% D)  This function is used by preTCM software!
%
% Author(s): Sergey N. Makarov, Copyright 2002 AEMM. Revision 2002/03/05
%            Miloslav Čapek, capekm6@fel.cvut.cz, 2010-2013
%
% See also impBsxFcn, impGndFcn, preTCM, prepTCMinput, TCM_RUN_solver
```

Function `publish`

- 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` - example

```
%% 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/matlab>.
```

publish →

Solver of Quadratic Equation

Function **solveQuadEq** solves quadratic equation.

Contents

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:

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

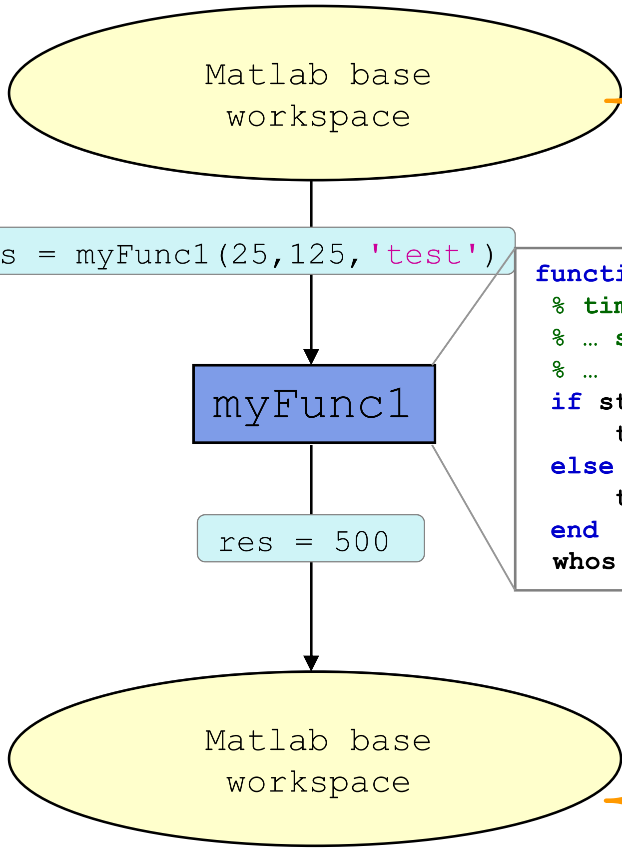
Matlab code:

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

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

Workspace of a function

- each function has its own workspace




```
>> clear, clc, A = 1; whos
 Name       Size                Bytes  Class     Attributes

  A          1x1                    8  double
>> res = myFunc1(25,125,'test');
```

```
function thisOutput = myFunc1(time,samples,tag)
 % time = 25; samples = 125; tag = 'test';
 % … source sode
 % …
 if strcmp(tag,'test')
        thisOutput =
 else
        thisOutput =
 end
 whos % workspace
```

myFunc1 workspace

```
 Name                Size                Bytes  Class     Attributes

  samples             1x1                     8  double
  tag                 1x4                     8  char
  thisOutput          1x1                     8  double
  time                1x1                     8  double
```

```
>> whos
 Name       Size                Bytes  Class     Attributes

  A          1x1                    8  double
  res        1x1                    8  double
```

Data space of a function #1

- on a function being called, input variables are not copied into workspace of the function, just their values are made accessible for the function (*copy-on-write technique*)
  - if an input variable is modified by the function, however, it is copied to the function's work space
  - with respect to memory saving and calculation speed-up it is advantageous to take corresponding elements out of a large array first and modify them rather than to modify the array directly and therefore evoke its copying in the function's workspace
- if the same variable is used as an input and output parameter it is immediately copied to the function's workspace
  - (provided that the input is modified in the script, otherwise the input and output variable is a reference to the same data)

Data space of a function #2

- all principles of programming covered at earlier stages of the course (operator overloading, data type conversion, memory allocation, indexing, etc.) apply to Matlab functions
  - in the case of overloading a built-in function, `builtin` is still applicable

- in the case of recursive function calling, own work space is created for each calling
  - pay attention to excessive increase of work spaces

- sharing of variables by multiple work spaces
  → global variables
  - by careful with how you use them (utilization of global variables is not recommended in general) and they are usually avoidable

Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

Function execution

- when is function terminated?
  - Matlab interpreter reaches 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
```


Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

Number of input and output variables

- number of input and output variables is specified by functions `nargin` a `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

500 s

- 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 call 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

```
hndl = plot(f);
```

Number of input and output variables

Syntactical types of functions

Function type	Description
main	header on the first line, above principles apply, the only one in the m-file visible from outside
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
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 (`myGoniomFcn = @(x) sin(x)+cos(x)`)
OOP	class 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 accessible using `help`

Local functions

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

```
function x = model_ITUR901(p,f)
% main function body
% ...
% ...
end

function y = calc_parTheta(q)
% function body
end
```

Local functions

- local functions launched by script (new from R2016b)
  - functions have to be at the end of file
  - all these functions have to be terminated with keyword `end`
  - local functions "see" each other and have their own workspaces

```
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
```

Nested functions

- nested functions are placed inside other functions
  - it 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`-`else`-`elseif` / `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
```


Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

Nested functions: calling

- apart from its workspace, nested functions can also access workspaces of all functions it is nested in

- nested function can be called from:
  - its parent function
  - nested function on the same level of nesting
  - function nested in it

- it is possible to create handle to a nested function
  - see later

```
function x = A(p)
    function y = B(q)
     ...
        function z = C(t)
         ...
        end
    end
...
    function u = D(r)
     ...
        function v = E(s)
         ...
        end
     ...
    end
...
end
```

Private functions

- they are basically the local functions, and they can be called by all functions placed in the root folder
- reside in subfolder `[private]` of the main function

- private functions can be accessed only by functions placed in the folder immediately above that private subfolder
  - `[private]` is often used with larger applications or in the case where limited visibility of files inside the folder is desired

```
…\TCMapp\
  private\
     eigFcn.m
     impFcn.m
     rwgFcn.m
  parTCM.m
  preTCM.m
  postTCM.m
```

these functions can be called by `parTCM`, `preTCM` and `postTCM` only

`parTCM` 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 in a usual way
- typically, handle references are used as input parameters of functions

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

```
>> whos
 Name      Size            Bytes  Class              Attributes

  ans       1x1                8  double
  fS        1x1               32  function_handle
```


Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

Anonymous functions

- 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

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

- `>> doc Anonymous Functions`

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


Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

Anonymous functions – Example

500 s

- create anonymous function $\mathbf{A}(p)=\left[A_1(p) \quad A_2(p) \quad A_3(p)\right]$ 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 $p=[0,2\pi]$

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

Taylor series – script

600 s

- expand exponential function using Taylor series:
  - in this case it is in fact McLaurin series (expansion about 0)

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

  - 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 script, enter :
    x (function argument)
    N (order of the series)

Taylor series – function

600 s

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

- implement as a function
  - choose appropriate name for the function
  - input parameters of the function are `x` and `N`
  - Output parameters are values `f1`, `f2` and `err`
  - add appropriate comment to the function (H1 line, inputs, outputs)

  - test the function

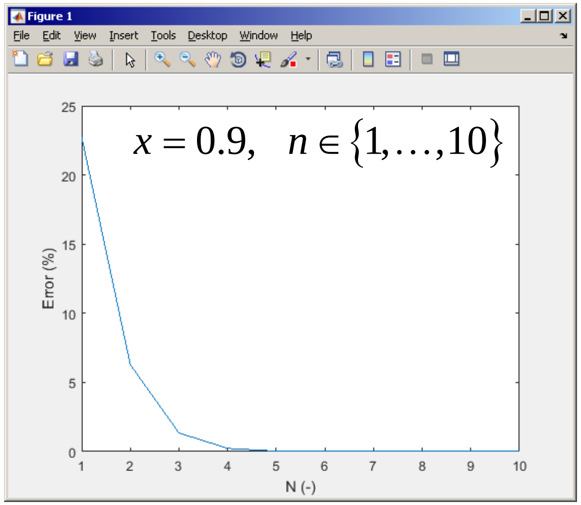
Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

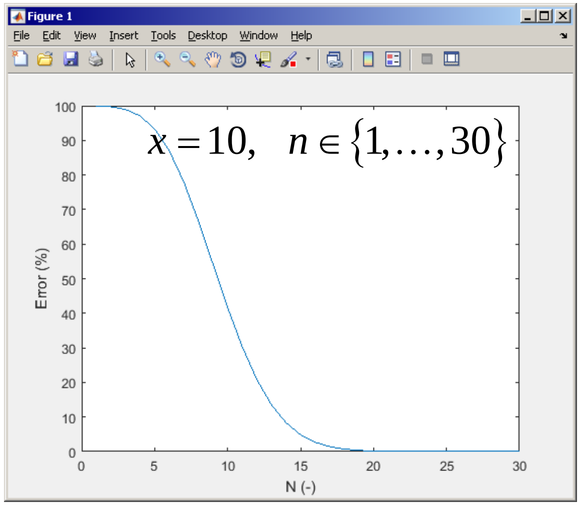
Taylor series – calling function

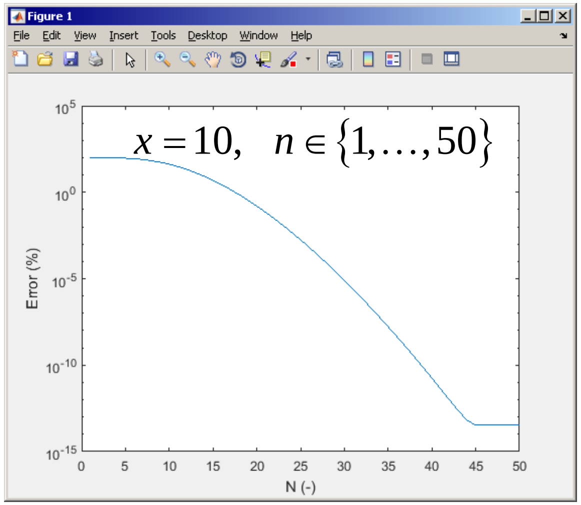
600 s

- 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)

Taylor series – results







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. GUI (functions like `stem`, `surf` etc. include `varargin`)
- variable `varargin` is always of type `cell`, even when it contains just a 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};
% ...
end
```


Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

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})
>> [extrema, 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`

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

property	value
Color	[R G B]
LineWidth	0.1 – …
Marker	'o', '*', 'x', …
MarkerSize	0.1 – …
and others …	

```
function plot_data(data, varargin)
%% documentation should be here!

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
```

`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 sporadically

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

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


Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

Output parameter `varargout`

180 s

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

Expression evaluation in another WS

- function `evalin` („evaluate in“) can be used to evaluate an expression in a workspace that is different from the workspace where the expression exists
- apart 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;
>> vysl = 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)
- ver. R2014b and older:
  - the number of recursion 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
```

- ver. R2015b and newer: recursion calling works until stack memory is not full
  - every calling creates new function's workspace!

Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

Number of recursion steps

360 s ↑

- write a simple function that is able to call itself; 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
  - think over in what situations the recursion is necessary…

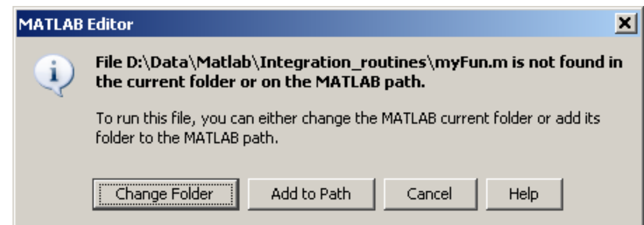
```
...
...
...
...
...
...
```

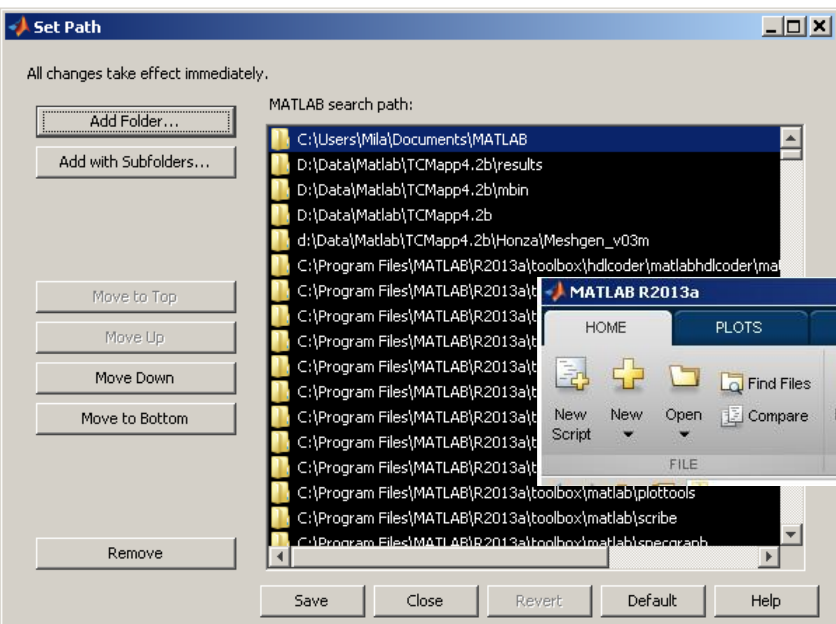
```
>> test_function(0)
```

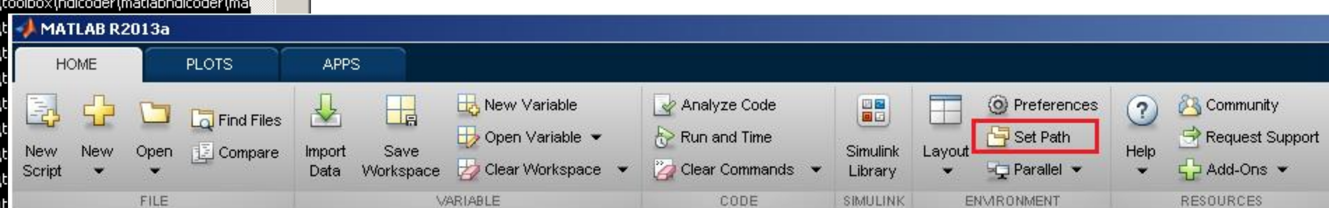

Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

Matlab `path`

- list of directories seen by Matlab : `>> path`
- for more see `>> doc path`
- `addpath`: adds folder to path
- `rmpath`: removes folder from path







Calling a function – order

- how Matlab searches for a function (simplified):
  - it is a variable
  - function imported using `import`
  - nested or local function inside given function
  - private function
  - function (method) of a given class or constructor of the class
  - function in given folder
  - 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`

Function vs. Command Syntax

- In Matlab exist two basic syntaxes how to call a function:

```
>> grid on     % Command syntax
>> % vs.
>> grid('on') % Function syntax
```

```
>> disp 'Hello Word!'   % Command syntax
>> % vs.
>> disp('Hello Word!') % Function syntax
```

- Command syntax
  - all inputs are taken as characters
  - outputs can't be assigned
  - input containing spaces has to be closed in single quotation marks

```
>> a = 1; b = 2;
>> plus a b % = 97 + 98
ans =
   195
>> p = plus a b % error
>> p = plus(a, b);
```


Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

Class `inputParser` #1

- 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` enables to insert 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` #2

- 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
```


Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

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 2x3, 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 following function call :

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

  - and then inside the function:

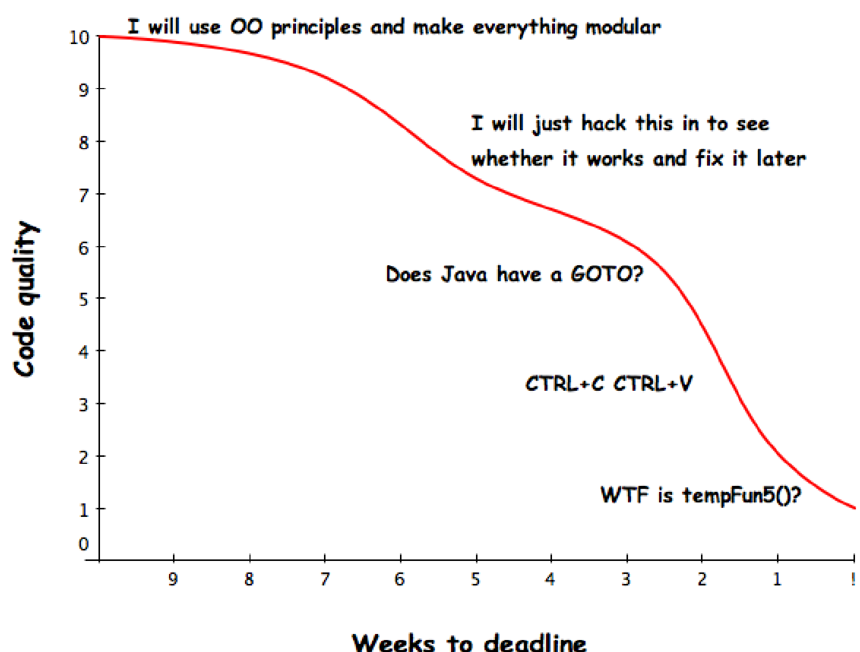
```
function output = myFunc1(par1, par2, par3)

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

Function creation – advices

- viewpoint of efficiency – the more often a function is used, the better its implementation should be
  - code scaling
  - it is appropriate to verify input parameters
  - it is appropriate to allocate provisional output parameters
  - debugging
  - optimization of function time

- principle of code fragmentation – in the ideal case each function should solve just one thing; each problem should be solved just once



xkcd.com/1132/

Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

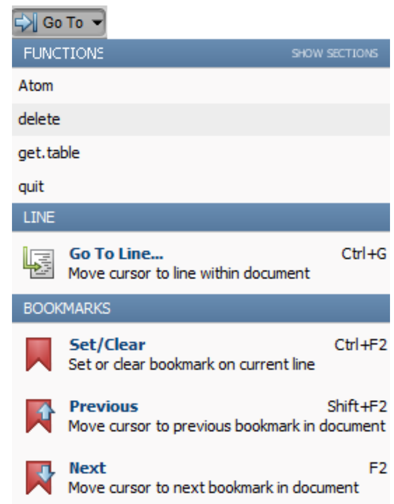
Selected advices for well arranged code

- ideally just one degree of abstraction
- code duplicity prevention
- function and methods should
  - solve one problem only, but properly
  - be easily and immediately understandable
  - be as short as possible
  - have the least possible number of input variables (< 3)

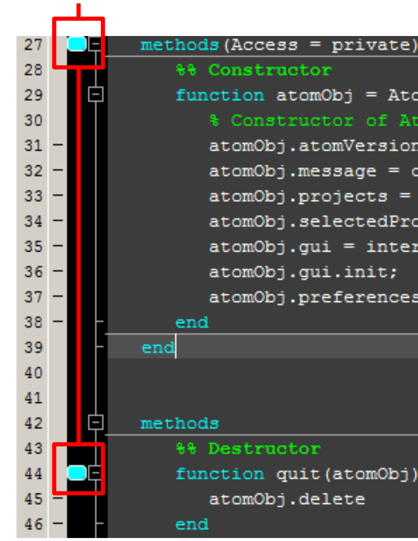
- further information:
  - Martin: Clear Code (Prentice Hall)
  - McConnell: Code Complete 2 (Microsoft Press)
  - Johnson: The Elements of Matlab Style (Cambridge Press)
  - Altman: Accelerating Matlab Performance (CRC)

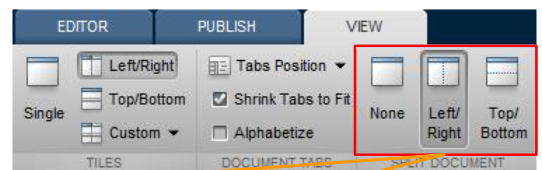
Useful tools for long functions

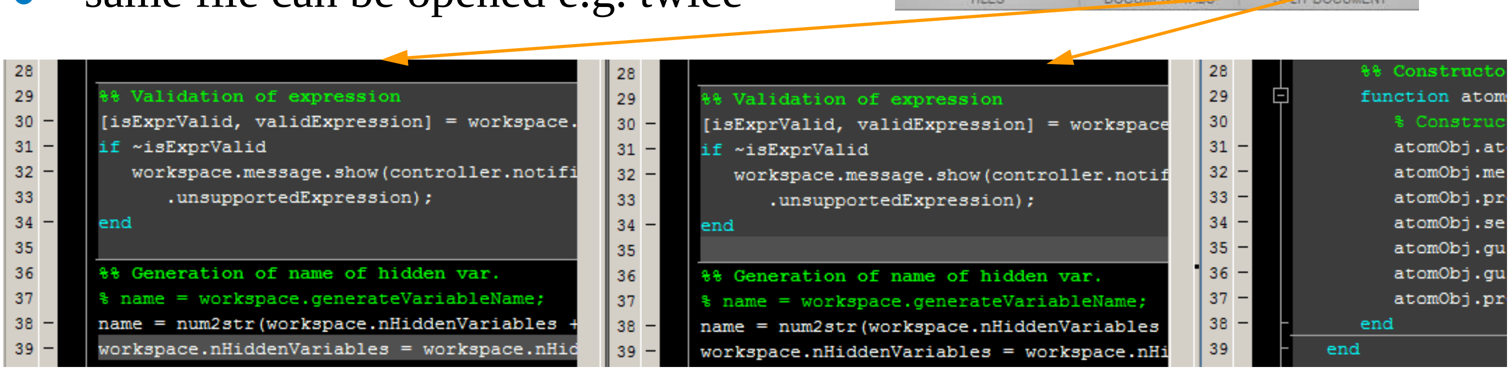
- bookmarks
  - CTRL+F2 (add / remove bookmark)
  - F2 (next bookmark)
  - SHIFT+F2 (previous bookmark)
- Go to...
  - CTRL+G (go to line)
- long files can be split
  - same file can be opened e.g. twice



bookmarks







Discussed functions

`function`	key word to create Matlab function
`@`	handle, anonymous function
`varargin, varargout`	variable number of input / output variables
`evalin, assignin`	evaluation of a command / assignment in another workspace
`inputname`	names of input variables in parent's workspace

Exercise #1 - notes

- find the unknown x in equation $f(x) = 0$ using Newton's method

- typical implementation steps:
  (1) mathematical model
    - seize the problem, its formal solution

  (2) pseudocode
    - layout of consistent and efficient code

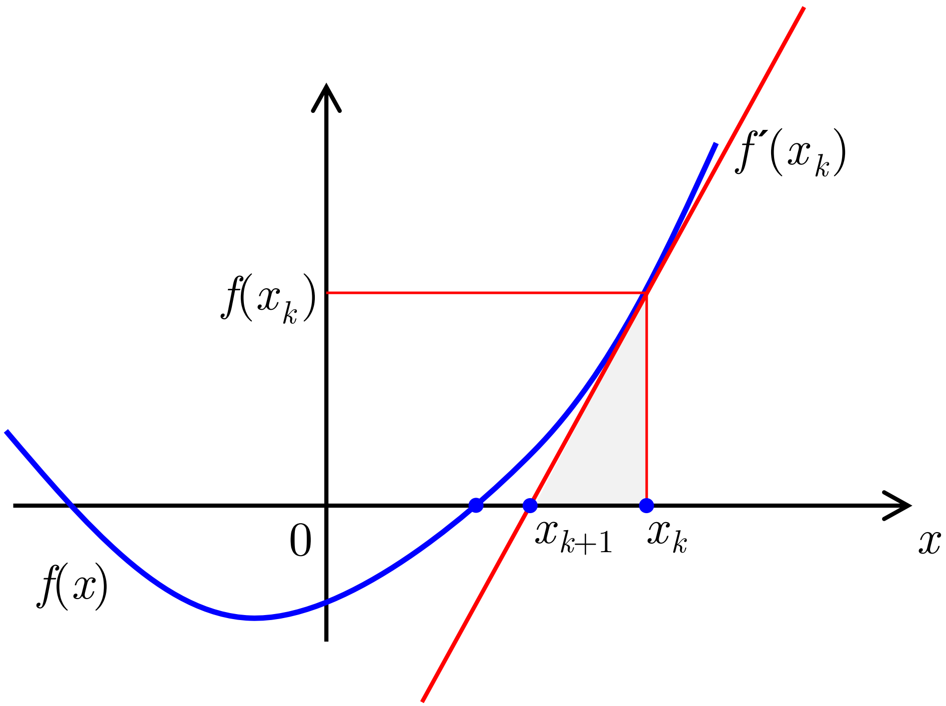
  (3) Matlab code
    - transformation into Matlab's syntax

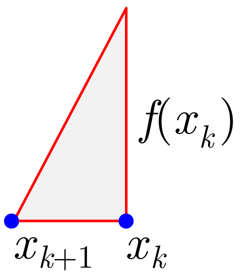
  (4) testing
    - usually using a problem with known (analytical) solution
    - try other examples...

Exercise #2

- find the unknown x in equation of type $f(x) = 0$
  - use Newton's method

- Newton's method:





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

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

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

Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

Exercise #3

- find the unknown x in equation $f(x)=0$ using Newton's method
- pseudocode draft:

(1) until $|(x_k - x_{k-1})/x_k| \geq err$ and simultaneously $k < 20$ do:

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

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

(4) $k = k + 1$

- pay attention to correct condition of the (`while`) cycle
- create a new function to evaluate $f(x_k)$, $f'(x_k)$
- use following numerical difference scheme to calculate $f'(x_k)$:

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

Exercise #4

600 s

- find the unknown x in equation $f(x)=0$ using Newton's method
  - implement the above method in Matlab to find the unknown x in $x^3+x-3=0$
  - the method comes in the form of a script calling following function :

```
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
```

```
function fx = optim_fcn(x)

fx = x^3 + x - 3;
```

Exercise #5

```
function fx = optim_fcn(x)

fx = x^3 + x - 3;
```

- 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*?

Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

Exercise #6

600 s

- using `integral` function calculate integral of current $Q = \int I(t)dt$ in the interval $t \in \langle 0{,}1 \rangle$s. The current has following time dependency, where $f = 50$ Hz

$$I\left(t\right) = 10\cos\left(2\pi f t\right) + 5\cos\left(4\pi f t\right)$$

  - solve the problem using handle function

  - using anonymous function

Thank you!

ver. 8.1 (12/11/2017)

Miloslav Čapek, Pavel Valtr
miloslav.capek@fel.cvut.cz

Document created as part of A0B17MTB course.