A0B17MTB – Matlab

Part #7

Miloslav Čapek
miloslav.capek@fel.cvut.cz

Filip Kozák, Viktor Adler, Pavel Valtr

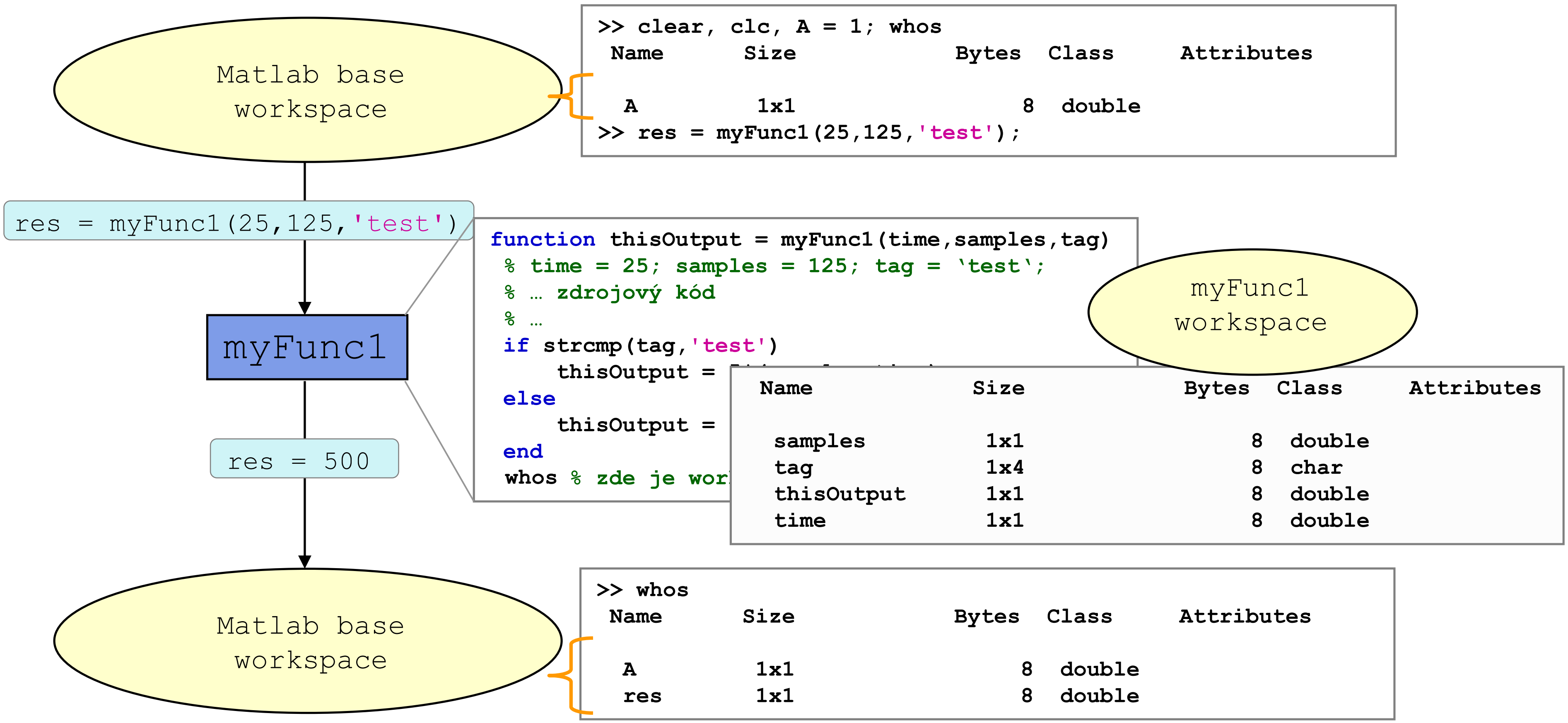
Department of Electromagnetic Field
B2-626, Prague

Learning how to …

Functions #2

Workspace of a function

- each function has its own workspace



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

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
  - 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 (monitored by Matlab, <500)
- 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

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
```


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)

if nargin == 1
    % do something
elseif nargin == 2
    % do something
else
    disp('Bad inputs!');
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
subfunction	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 subfunction, sees the WS of the function one level up
handle	Function reference (`mySinX = @sin`)
anonymous	similar to handle functions (`myGoniomFcn = @(x) sin(x)+cos(x)`)
OOP	specific access (constructor, `set` a `get` methods and other)

- 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 subfunctions is not important (logical connection!)
- help of local functions is not accessible using `help`

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

Subfunctions

- subfunctions can be launched by main functions only
  - 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)
  - subfunctions "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
```

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
```

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 subfunctions, and they can be called by all functions plased 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]

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

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
```

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)
```

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 used in function's header, `varargin` returns negative value

```
function plot_data(varargin)

nargin
celldisp(varargin)

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


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

Variable number of input parameters

500 s

- create a new function with header in the form :

```
function plot_data(data, varargin)
```

where variables in `varargin` are always in pairs (property and its value)

- verify first that the variable `data` contains numerical data (`is*` function)
- if it is the case, plot the data, if not, display an error message and terminate the function

```
hndl = plot(data);
```

- then, step by step (until `varargin` is empty) assign values to corresponding properties:

```
set(hndl, property, value);
```

- for properties see >> `doc line` or following page

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

Variable number of input parameters

- selected properties that can be used for function testing:
  - think about how to modify the function to detect incorrect properties (non-existing / wrongly entered / with wrong value)?

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

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

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
```

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

Output parameter `varargout`

180 s

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

```
function varargout = fibonacciFcn2(in1, in2)
% documentation is here!!!
if nargin == 0
   limit = 1e3;
   f     = [0, 1];
elseif nargin == 1
   limit = in1;
   f     = [0, 1];
elseif nargin == 2
   limit = in1;
   f     = in2;
end

n = 1;      % index for series generation
while f(n+1) < limit
   f(n+2) = f(n) + f(n+1);
   n = n + 1;
end
f(end) = [];
varargout{1} = f;
if nargout == 2
   varargout{2} = plot(f);
end
```

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)]);
```

Recursion

- Matlab supports recursion (function can call itself)
  - the number of recursion is limited by 500 by default
  - the number of recursions can be changed, or get current setting:

```
>> get(0, 'RecursionLimit')
```

  - recursion is part of some useful algorithms (e.g. Adaptive Simpsons Method of integration)

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…

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

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

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

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
```

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:

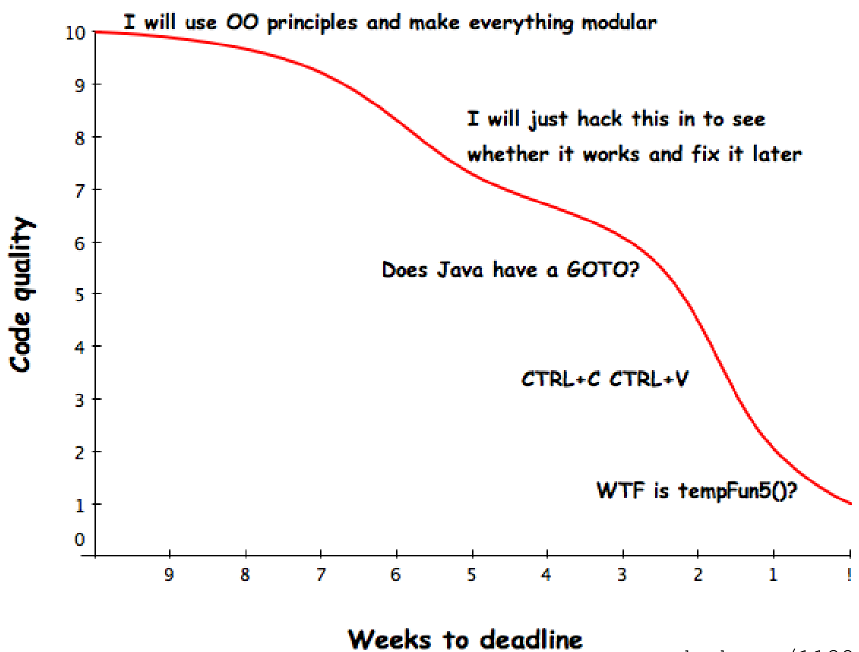
```
function output = myFunc1(par1, par2, par3)

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


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

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/

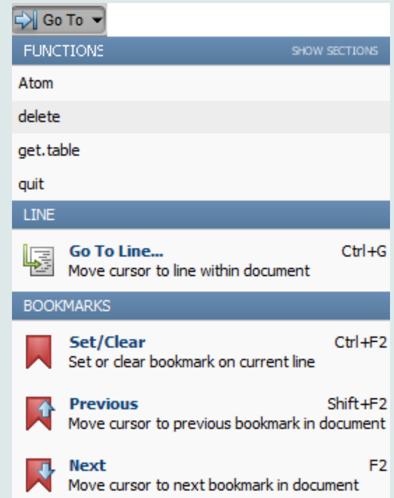
Selected advices for well arranged code

- 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)

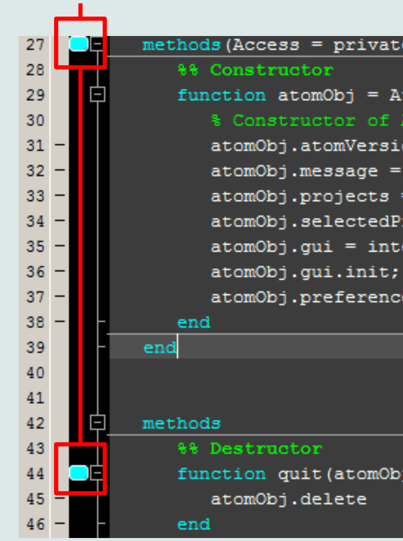
- further information:
  - Martin: Clear Code (Prentice Hall)
  - McConnell: Code Complete 2 (Microsoft Press)
  - Johnson: The Elements of Matlab Style (Cambridge Press)
  - Altman: Accelerating Matlab Performance (CRC)

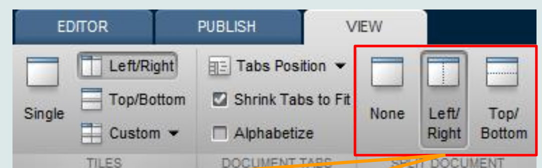
Useful tools for long functions

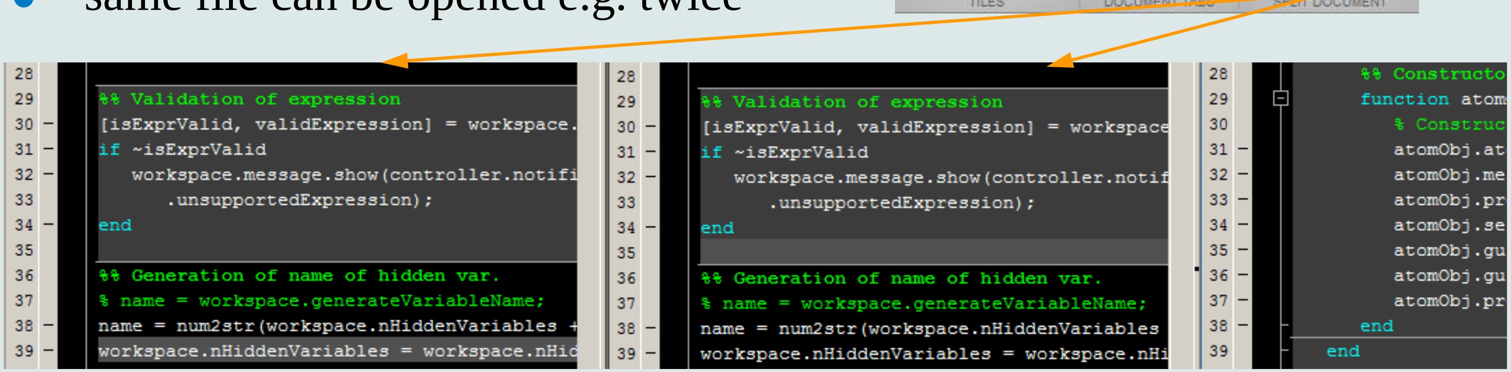
- bookmarks
  - CTRL+F2 (add / remove bookmark)
  - F2 (next bookmark)
  - SHIFT+F2 (previous bookmark)
- Go to...
  - CTRL+G (go to line)
- long files can be split
  - same file can be opened e.g. twice



bookmarks







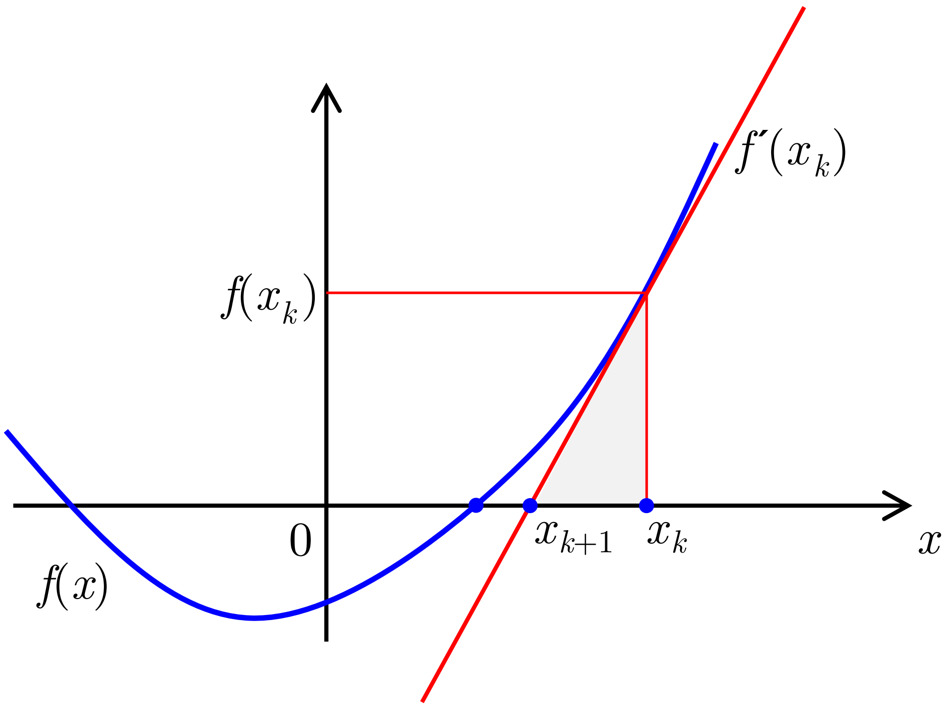
Discussed functions

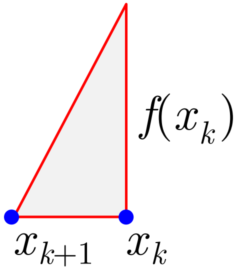
`@`	handle, anonymous function
`varargin, varargout`	variable number of input / output variables
`evalin, assignin`	evaluation of a command / assignment in another workspace
`inputname`	names of input variables in parent's workspace

Exercise #1

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

- Newton's method:





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

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

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

Exercise #2

- 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 #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 :

```
function fx = optim_fcn(x)

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

```
clear all; close all; clc;

% enter variables
% xk, xk1, err, k

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

Exercise #5

- 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?

Cvičení #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

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

  - solve the problem using handle function

  - using anonymous function

Thank you!

ver. 4.2 (28/09/2015)
Miloslav Čapek, Pavel Valtr
miloslav.capek@fel.cvut.cz

Document created as part of A0B17MTB course.