# A0B17MTB – Matlab

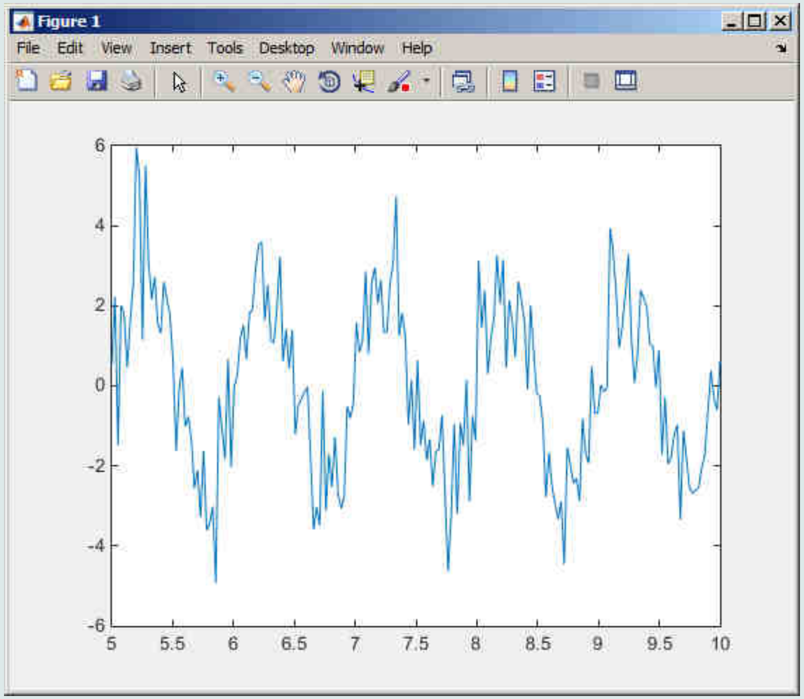
# Part #4

Miloslav Čapek
miloslav.capek@fel.cvut.cz

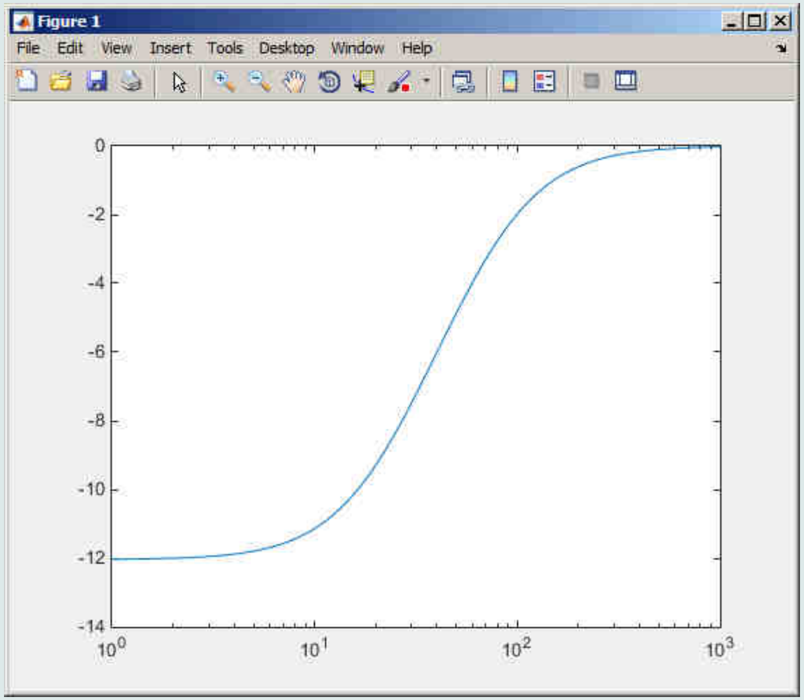
Filip Kozák, Viktor Adler, Pavel Valtr

Department of Electromagnetic Field
B2-626, Prague

# Solution to exercise #3 from last lecture



# Solution to exercise #5 from last lecture

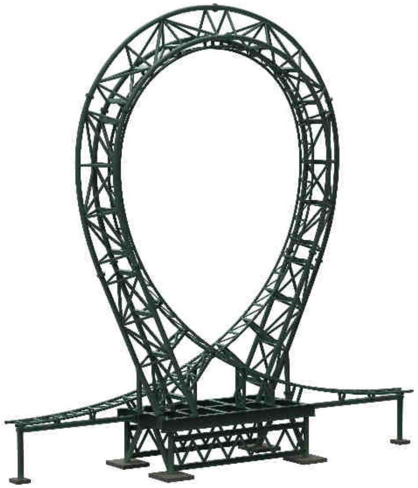


# Learning how to …

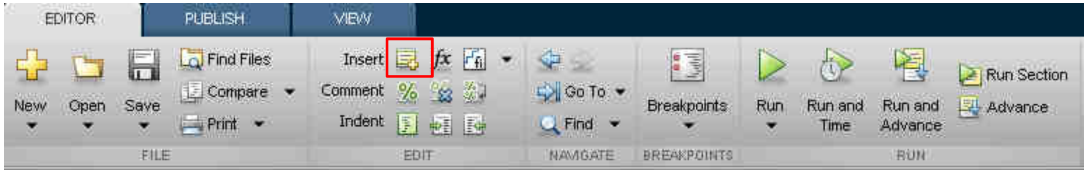
**Relational and logical operators**

**Cycles**

**Program branching #1**



# Cell mode in Matlab Editor




- cells enable to separate the code into smaller logically compact parts
  - separator: **%%**
  - the separation is visual only, but it is possible to execute a single cell - shortcut CTRL+ENTER
- in the older versions of Matlab, it is usually necessary to activate the cell mode

# Cell mode in Matlab Editor

240 s

- split previous script (`loanRepayment.m`) into separate parts
  - use the (cell) separator **%%**


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

# Data in scripts

- scripts can use data that has appeared in Workspace
- variables remain in the Workspace even after the calculation is finished
- operations on data in scripts are performed in the base Workspace

# Naming conventions of scripts and functions

- names of scripts and functions
  - max. number of characters is 63 (additional characters are ignored)
  - naming restrictions similar to variable names apply
  - choose names describing what the particular function calculates
  - avoid existing names as the new script is called instead of an existing built-in function (overloading can occur)
- more information:
  - `http://www.mathworks.com/matlabcentral/fileexchange/2529-matlab-programming-style-guidelines`
- in the case you want to apply vector functions row-wise
  - check whether the function enables calculation in the other dimension (`max`)
  - transpose your matrix
  - some of the functions work both column-wise and row-wise (`sort` × `sortrows`)

# startup.m script

- script `startup.m`
  - always executed at Matlab start-up
  - it is possible to put your predefined constants and other operations to be executed (loaded) at Matlab start-up
- location (use `>> which startup`):
  - `...\Matlab\R201Xx\toolbox\local\startup.m`
- change of base folder after Matlab start-up :

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

```
Workspace is changing to:

d:\Data\Matlab

February 25, 2014  3:36:03.347 PM
Keep on working...
>>
```

# `matlabrc.m` script

- executed at Matlab start-up (or manually executed: `>> matlabrc`)
- contains some basic definitions, e.g.
  - figure size, set-up of some graphic elements
  - sets Matlab path (see later)
  - and others
- in the case of a multi-license it is possible to insert a message in the script that will be displayed to all users at the start-up
- location (use `>> which matlabrc`):
  - `...\Matlab\R201Xx\toolbox\local\matlabrc.m`

- last of all, `startup.m` is called (if existing)

- `matlabrc.m` is to be modified only in the case of absolute urgency!

# Relational operators

- to inquire, to compare, whether 'something' is greater than, lesser than, equal to etc.
- the result of the comparison is always either
  - positive (`true`), logical one „1"
  - negative (`false`), logical zero „0"

| | |
|---|---|
| `>` | greater than |
| `>=` | greater than or equal to |
| `<` | lesser than |
| `<=` | lesser than or equal to |
| `==` | equal to |
| `~=` | not equal to |

- all relational operators are vector-wise
  - it is possible to compare as well vectors vs. vectors, matrices vs. matrices, …
- often in combination with logical operators (see later)
  - more relational operators applied to a combination of expressions


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

# Relational operators

300 s

- having the vector $\mathbf{G} = \begin{pmatrix} \dfrac{\pi}{2} & \pi & \dfrac{3}{2}\pi & 2\pi \end{pmatrix}$, find elements of $\mathbf{G}$ that are
  - greater than $\pi$
  - lesser or equal to $\pi$
  - not equal to $\pi$
- try similar operations for $\mathbf{H} = \mathbf{G}^{\mathrm{T}}$ as well
- try to use relational operators in the case of a matrix and scalar as well
- find out whether $\mathbf{V} \geq \mathbf{U}$:

$$\mathbf{V} = \begin{pmatrix} -\pi & \pi & 1 & 0 \end{pmatrix}$$

$$\mathbf{U} = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}$$


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

# Relational operators

200 s

- find out results of following relations
  - try to interpret the results

```
>> 2 > 1 & 0 % ???
```

```
>> r = 1/2;
>> 0 < r < 1 % ???
```

```
>> (1 > A) <= true
```


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

# Logical operators

- to enquire, to find out, whether particular condition is fulfilled
- the result is always either
  - positive (`true`), logical one „1“
  - negative (`false`), logical zero „0“

| | |
|---|---|
| & | `and` |
| \| | `or` |
| ~ | `not` |
| | `xor` |
| | `all` |
| | `any` |

- `all`, `any` is used to convert logical array into a scalar

- Matlab interprets any numerical value except `0` as `true`
- all logical operators are vector-wise
  - it is possible to compare as well vectors vs. vectors, matrices vs. matrices, …

- functions `is*` extend possibilities of logical enquiring
  - we see later


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

# Logical operators – application

- assume a vector of 10 random numbers ranging from -10 to 10

```
>> a = 20*rand(10, 1) - 10
```

  - following command returns `true` for elements fulfilling the condition:

```
>> a < -5 % relation operator
```

  - following command returns values of those elements fulfilling the condition (logical indexing):

```
>> a(a < -5)
```

  - following command puts value of -5 to the position of elements fulfilling the condition :

```
>> a(a < -5) = -5
```

  - following command sets value of the elements in the range from -5 to 5 equal to zero (opposite to tresholding):

```
>> a(a > -5 & a < 5) = 0
```

  - tresholding function (values below -5 sets equal to -5, values above 5 sets equal to 5):

```
>> a(a < -5 | a > 5) = sign(a(a < -5 | a > 5))*5
```

# Logical operators

420 s

- determine which of the elements of the vector $\mathbf{A} = \begin{pmatrix} \frac{\pi}{2} & \pi & \frac{3}{2}\pi & 2\pi \end{pmatrix}$

  - are equal to π <u>or</u> are equal to 2π
    - pay attention to the type of the result (= logical values `true` / `false`)

  - are greater than π/2 <u>and at the same time</u> are not equal 2π

  - elements from the previous condition add to matrix A

# Logical operators: &&, ||

- in the case we need to compare scalar values only then "short-circuited" evaluation can be used

- evaluation keeps on going till a point where it makes no sense to continue
  - i.e. when evaluating

```
>> clear; clc;
>> a = true;
>> b = false;
>> a && b && c && d
```

… no problems with undefined variables `c`, `d`, because the evaluation is terminated earlier

- however:
  - terminated with error …

```
>> clear; clc;
>> a = true;
>> b = true;
>> a && b && c && d
```

# Logical operators

150 s

- create a row vector in the interval from 1 to 20 with step of 3
  - create a the vector filled with elements from the previous vector that are greater than 10 <u>and at the same time</u> smaller than 16; use logical operators

# Logical operators

240 s

- create matrix `A = magic(3)` and find out using functions `all` and `any`
  - in which columns all elements are greater than 2
  - in which rows at least one element is greater than or equal to 8
  - whether the matrix A contains positive numbers only

$$\mathbf{A} = \begin{pmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{pmatrix}$$

$$\text{any}\begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}, \quad \text{all}\begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}, \quad \text{any}\left(\text{all}\begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}\right) = \text{any}\begin{pmatrix} 0 & 1 & 0 \end{pmatrix} = 1$$


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

# Logical operators

240 s

- find out the result of following operation and interpret it

```
>> ~(~[1 2 0 -2 0])
```

- test whether variable *b* is not equal to zero and then test whether at the same time $a$ / $b$ > 3
  - following operation tests whether both conditions are fulfilled while avoiding division by zero!


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

# Matrix indexation using own values

300 s

- create matrix `A`

```
>> N = 4;
>> A = magic(N)
```

```
A =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

- first think about what will be the result of the following operation and only then carry it out

```
>> B = A(A)
```

  - does the result correspond to what you expected?
  - can you explain why the result looks the way it looks?
  - notice the interesting mathematical properties of the matrix `A` and `B`
  - are you able to estimate the evolution?, `C = B(B)`

- try similar process for `N = 3` or `N = 5`

# Program branching – loops

- repeating certain operation multiple-times, one of the basic programming techniques
- There are 2 types of cycles in Matlabu:
  - `for` – the most used one, number of repetitions is known in advance
  - `while` – condition is known ensuring cycle (dis)continuation as long as it remains true
- essential programing principles to be observed:
  - memory allocation (matrix-related) of sufficient size /see later.../
  - cycles should be properly terminated /see later.../
  - To ensure terminating condition with `while` cycle /see later.../
- frequently is possible to modify the array (1D → 2D, 2D → 3D using function `repmat` and carry out a matrix-wise operation, under certain conditions the vectorized code is faster and more understandable, possibility of utilization of GPU)
- we always ask the question: <u>is a cycle really necessary?</u>



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

# for loop

- `for` loop is applied to <u>known number of repetitions</u> of a group of commands

```
for m = expression
    commands
end
```

- `expression` is a vector / matrix; columns of this vector / matrix are successively assigned to `m` / `n`

```
for n = 1:4
    n
end
```

```
for m = magic(4)
    m
end
```

- frequently, `expression` is generated using `linspace` or using „`:`“, with the help of `length`, `size`, etc.

- instead of `m` it is possible to use more relevant names like `mPoints`, `mRows`, `mSymbols`, …
  - for clarity, it is suitable to use e.g. `mXX` pro rows and `nXX` for columns

# Loops #1

400 s

- create a script to calculate factorial $N$!
  - use a cycle, verify your result using Matlab `factorial` function

```
>> factorial(N)
```

  - can you come up with other solutions? (e.g. using vectorising…)
  - compare all possibilities for decimal input $N$ as well



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

# Memory allocation

- allocation can prevent perpetual increase of the size of a variable
  - Code Analyser (M-Lint) will notify you about the possibility of allocation by underlining the matrix's name
  - whenever you know the size of a variable, allocate!
    - sometimes, it pays off to allocate even when the final size is not known - then the worst-case scenario size of a matrix is allocated and then the size of the matrix is reduced
    - allocate the variables of the largest size first, then the smaller ones
- example:
  - try…

```
%% WITHOUT allocation
tic;
for m = 1:1e7
    A(m) = m + m;
end
toc;
% computed in 0.45s
```

```
%% WITH allocation
tic;
A = zeros(1,1e7);
for m = 1:1e7
    A(m) = m + m;
end
toc;
% computed in 0.06s
```



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

# while loop

- keeps on executing commands contained in the body of the cycle (`commands`) depending on a logical condition

```
while condition
    commands
end
```

- keeps on executing commands as long as all elements of the expression (`condition` can be a multidimensional matrix) are non-zero
  - the condition is converted to a relational expression, i.e. till all elements are `true`
  - logical and relational operators are often used for condition testing
- if `condition` is not a scalar, it can be reduced using functions `any` or `all`


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

# Typical application of loops

```
%% script generates N experiments with M throws with a die
close all; clear all; clc;

Mthrows   = 1e3;
Ntimes    = 1e2;
Results   = NaN(Mthrows, Ntimes);
for mThrow = 1:Mthrows % however, can be even further vectorized!
    Results(mThrow, :) = round(rand(1, Ntimes)); % vectorized
end
```

```
%% script finds out the number of lines in a file
fileName = 'sin.m';
fid = fopen(fileName, 'r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    count = count + 1;
end
disp(['lines:' num2str(count)])
fclose(fid);
```

# Loops #2

360 s

- calculate the sum of integers from 1 to 100 using `while` cycle
  - apply any approach to solve the task, but use `while` cycle
  - are you able to come up with another solution (using a Matlab function and without cycle)?


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

# `while` cycle – infinite loop

- pay attention to conditions in `while` cycle that are always fulfilled ⇒ danger of infinite loop
  - mostly, not always however(!!) it is a semantic error

- trivial, but good example of a code…

```
while 1 == 1
    disp('ok');
end
```

```
while true
    disp('ok');
end
```

… that „never“ ends (shortcut to terminate: CTRL+C)


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

# Interchange of an index an complex unit

- be careful not to confuse complex unit (`i`, `j`) for cycle index
  - try to avoid using `i` and `j` as an index
    - overloading can occur (applies generally, e.g. `>> sum = 2` overloads the `sum` function)

- find out the difference in the following pieces of code:

```
A = 0;
for i = 1:10
    A = A + 1i;
end
```

```
A = 0;
for i = 1:10
    A = A + i;
end
```

```
A = 0;
for i = 1:10
    A = A + j;
end
```

- all the commands, in principle, can be written as one line

```
A = 0; for i = 1:10, A = A + 1i; end,
```

  - usually less understandable, not even suitable from the point of view of the speed of the code


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

# Nested loops, loop combining

- quite frequently there is a need for nested loops
  - consider vectorising instead
  - consider loop type

- loop nesting usually rapidly increases computational demands

```
%% script generates N experiments with M throws with a die
close all; clear all; clc;

Mthrows  = 1e3;
Ntimes   = 1e2;
Results  = NaN(Mthrows, Ntimes);
for mThrow = 1:Mthrows
    for nExperiment = 1:Ntimes % not vectorized (30 times slower!!)
        Results(mThrow, nExperiment) = round(rand(1));
    end
end
```



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

# Loops #3

600 s

- fill in the matrix using loops $\mathbf{A}(m,n) = \frac{mn}{4} + \frac{m}{2n}$
- consider $m \in \{1,\ldots,100\}$, $n \in \{1,\ldots,20\}$, allocate matrix first
- create a new script

- to plot the matrix **A** use for instance the function `pcolor()`


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

# Loops #4

600 s

- in the previous task the loops can be avoided entirely by using vectorising
  - it is possible to use `meshgrid` function to prepare the matrices needed

  - `meshgrid` can be used for 3D arrays as well!!



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

# Loops #5

600 s

- visualize current distribution of a dipole antenna described as

$$\mathrm{I}(x,t)=\mathrm{I}_0(x)\mathrm{e}^{-\mathrm{j}\omega_0 t},\quad \mathrm{I}_0(x)=\cos(x),\quad \omega_0=2\pi$$

- in the interval $t\in(0,4\pi),\quad x\in\left(-\frac{\pi}{2},\frac{\pi}{2}\right)$ choose N = 101

for visualization inside the loop use following piece of code:

```
% ... your code
    figure(1);
    plot(x,real(I));
    axis([x(1) x(end) -1 1]);
    pause(0.1);
% ... your code
```

# Loops #6

600 s

- try to write moving average code applied to following function

$$f(x) = \sin^2(x)\cos(x) + 0.1r(x),$$

where $r(x)$ is represented by function of uniform distribution (`rand()`)

  - use following parameters

```
clear; clc;
signalSize = 1e3;
x  = linspace(0, 4*pi, signalSize);
f  = sin(x).^2.*cos(x) + 0.1*rand(1, signalSize);
windowSize = 50;
% your code ...
```

  - and then plot:

```
plot(x, f, x, my_averaged);
```

  - try to make the code more efficient


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

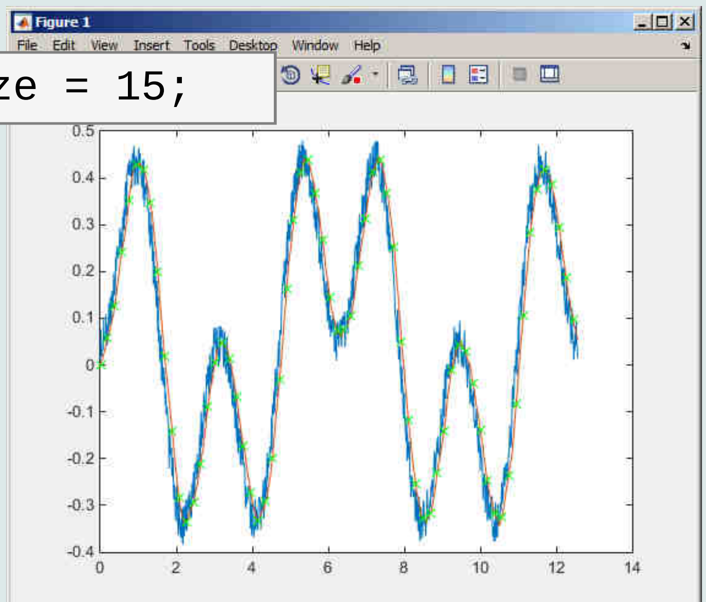
# Loops #7

600 s

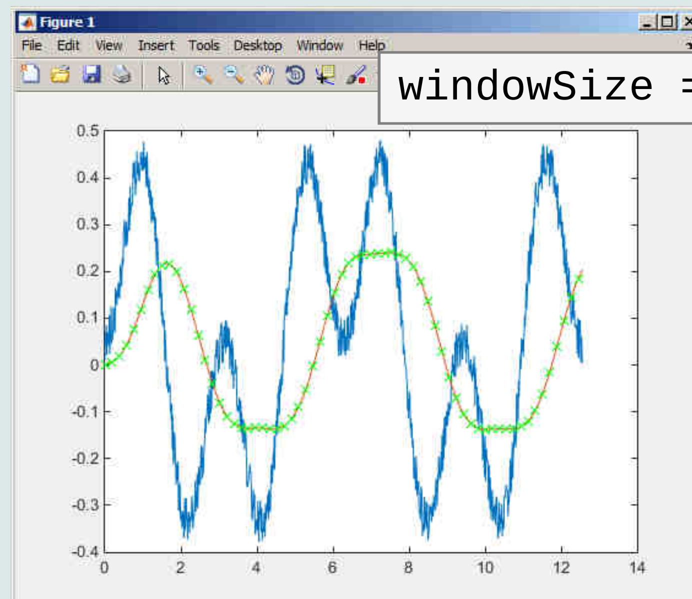
- for comparison it is possible to use Matlab built-in function `filter`

- check how the result is influenced by parameter `windowSize`

`windowSize = 15;`




`windowSize = 150;`





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

# break, continue

- function `break` enables to terminate execution of the loop

```
% another code ...
for k = 1:length(A)
    if A(k) > threshold
        break;
    end
    % another code ...
end
```

if (true)

- function `continue` passes control to next iteration of the loop

```
% another code ...
for k = 1:length(A)
    if A(k) > threshold
        continue;
    end
    % another code ...
end
```

if (true)

# Loops vs. vectorizing #1

- since Matlab 6.5 there are two powerful hidden tools available
  - *Just-In-Time accelerator* (JIT accelerator)
  - *Real-Time Type Analysis* (RTTA)

- JIT enables partial compilation of code segments
  - precompiled loops are even faster than vectorizing
  - following rules have to be observed with respect to loops:
    - scalar index to be used with `for` loop
    - only built-in functions are called inside the body of `for` loop
    - the loop operates with scalar values only

- RTTA assumes the same data types as during the previous course of the code - significant speed up for standartized calculations
  - when measuring speed of the code, it is necessary to carry out so called warm-up (first run the code 2 or 3 times)


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

# Loops vs. vectorizing #2

- the motivation for introduction of JIT was to catch up with 3. generation languages
  - when fully utilized, JIT's computation time is comparable to that of C or Fortran
- highest efficiency (the highest speedup) in particular
  - when loops operate with scalar data
  - when no user-defined functions are called (i.e. only build-in functions are called)
  - when each line of the loop uses JIT
- as the result, some parts of the code don't have to vectorised (or should not even be!)
- the whole topic is more complex (and simplified here)
  - for more details see `JIT_accel_Matlab.pdf` at the webpage of this course


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

# Loops vs. vectorizing #3

- previous statement will be verified using a simple code - filling a band matrix
- conditions for using JIT are fulfilled …
  - working with scalars only, calling built-in functions only
- filling up the matrix using `for` loops is faster!
  - try it yourself…

```
clear; clc;
N   = 5e3;
mat = NaN(N, N);
tic,
for n1=1:N
    for n2=1:N
        mat(n1, n2)=0;
    end
end
for n1 = 1:N
    mat(n1, n1)=1;
end
for n1 = 1:(N-1)
    mat(n1, n1+1)=2;
end
for n1 = 2:N
    mat(n1, n1-1)=3;
end
toc,
% computed in 0.52s
(2015b)
```

```
clear; clc;
N   = 5e3;

tic,
mat = diag(ones(N, 1)) + ...
    2*diag(ones(N-1, 1), 1) + ...
    3*diag(ones(N-1, 1), -1);
toc,
% computed in 0.18s (2015b)
```

# Program branching

- if it is needed to branch program (execute certain part of code depending on whether a condition is fulfilled), there are two basic ways:
  - `if` – `elseif` – `else` – `end`
  - `switch` – `case` – `otherwise` – `end`

```
if condition
    commands
elseif condition
    commands
elseif condition
    commands
else
    commands
end
```

```
switch variable
 case value1
    commands
 case {value2a, value2b, ...}
    commands
 case ...
    commands
 otherwise
    commands
end
```


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

# `if` vs. `switch`

| `if-elseif-else-end` | `switch-otherwise-end` |
|---|---|
| it is possible to create very complex structure (&& / \|\|) | simple choice of many options |
| `strcmp` is used to compare strings of various lengths | test strings directly |
| test equality / inequality | test equality only |
| great deal of logical expressions is needed in the case of testing many options | enables to easily test one of many options using `{}` |


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

# Program branching – `if` / `else` / `elseif`

- the most probable option should immediately follow the `if` statement
- only the `if` part is obligatory
- the `else` part is carried out only in the case where other conditions are not fulfilled

- if a M×N matrix is part of the condition, the condition is fulfilled only in the case it is fulfilled for each element of the matrix

- the condition may contain calling a function etc.
- `if` conditions may be nested

```
c = randi(1e2);
if mod(c, 2)
    disp('c is odd');
elseif c > 10
    disp('even, >10');
end
```



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

# Program branching – `if` / `else` / `elseif`

400 s

- generate random numbers

```
r = 2*rand(8, 1)-1;
```

- save the numbers in matrices `Neq` and `Pos` depending on whether each number is negative or positive; use `for` cycle, `if-else` statement and indexing for storing values of `r`

  - pay attention to grownth in size of mtrices `Pos` and `Neq` – how to solve the problem?

- can you come up with a more elegant solution? (for cycle is not always necessary)

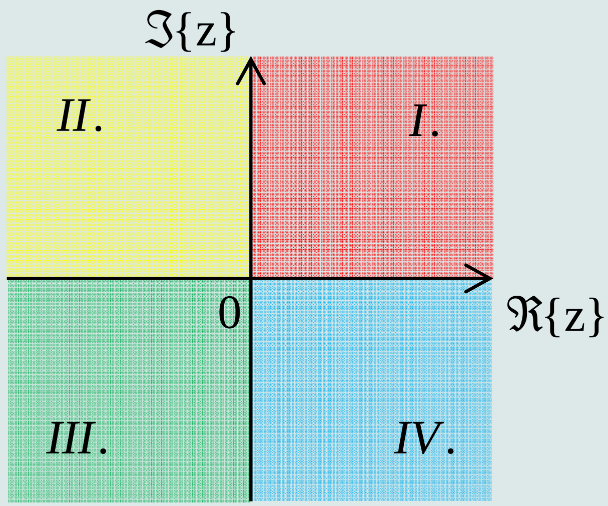


Department of Electromagnetic Field, CTU FEE, `miloslav.capek@fel.cvut.cz`

# Program branching – `if` / `else` / `elseif`

500 s

- write a script generating a complex number and determining to what quadrant the complex number belongs to



# Discussed functions

| | | |
|---|---|---|
| `edit` | open Matlab Editor | ● |
| `disp, pause` | display result in command line, pauses code execution | ● |
| `num2str` | conversion from datatype `numeric` to `char` | ● |
| `for-end, while-end` | loop | ● |
| `factorial` | calculate factorial | |
| `break, continue` | terminates loop execution, passes control to loop's next iteration | |
| `and, or, not, xor` | functions overloading logical operators | |
| `all, any` | evaluation of logical arrays („all of", „at least one of") | |
| `sign` | signum function | |
| `if-elseif-else-end` | branching statement | ● |

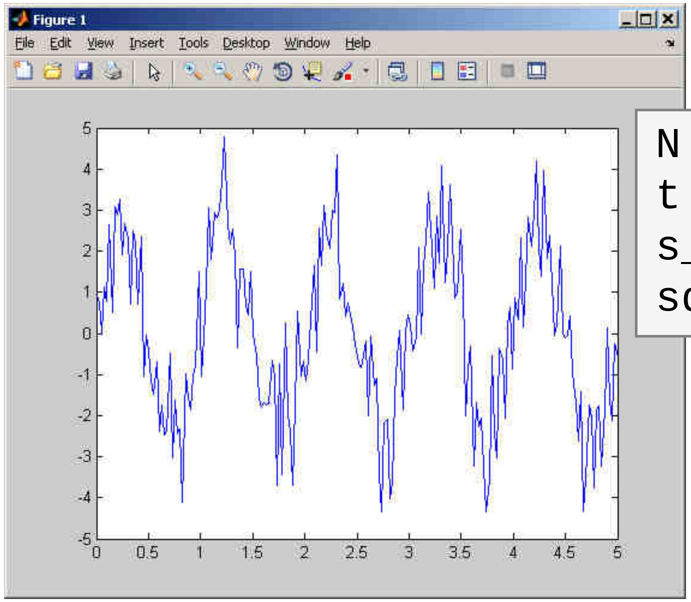

Department of Electromagnetic Field, CTU FEE, miloslav.capek@fel.cvut.cz

# Exercise #1

360 s

- recall the signal from lecture 3
  - try again to limit the signal by values $s_{\text{min}}$ a $s_{\text{max}}$
  - use relational operators (`>` / `<`) and logical indexing (`s(a>b) = c`) instead of functions `max`, `min`
    - solve the task item-by-item

$$s_{\text{p}}(t) = \begin{cases} s_{\text{min}} \Leftrightarrow s(t) < s_{\text{min}} \\ s_{\text{max}} \Leftrightarrow s(t) > s_{\text{max}} \\ s(t) \ldots \text{ jinak} \end{cases} \qquad s_{\text{min}} = -\frac{9}{10} \qquad s_{\text{max}} = \frac{\pi}{2}$$

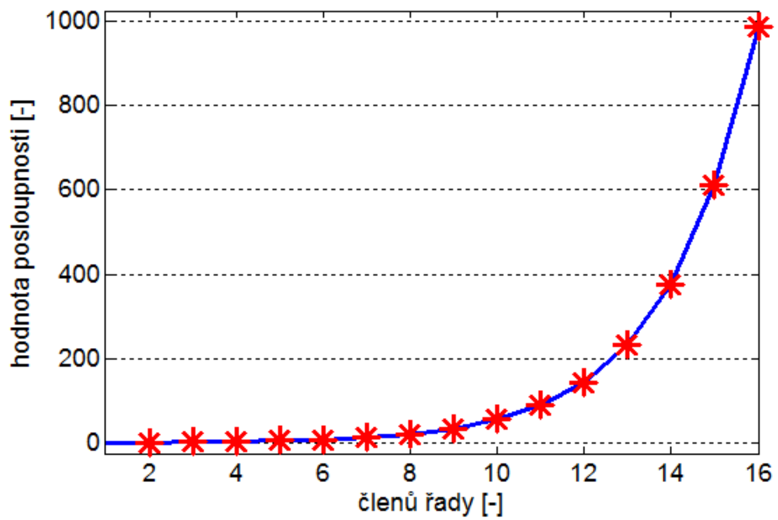



```
N = 5; V = 40;
t = linspace(0, N, N*V);
s_t = randn(1, N*V) + ...
sqrt(2*pi)*sin(2*pi*t);
```

# Exercise #2

600 s ↑

- draft a script to calculate values of Fibonacci sequence up to certain value `limit`
  - have you come across this sequence already?
  - if not, find its definition
  - implementation:
    - what kind of loop you use (if any)?
    - what matrices / vectors do you allocate?
  - plot the resulting series using function `plot`

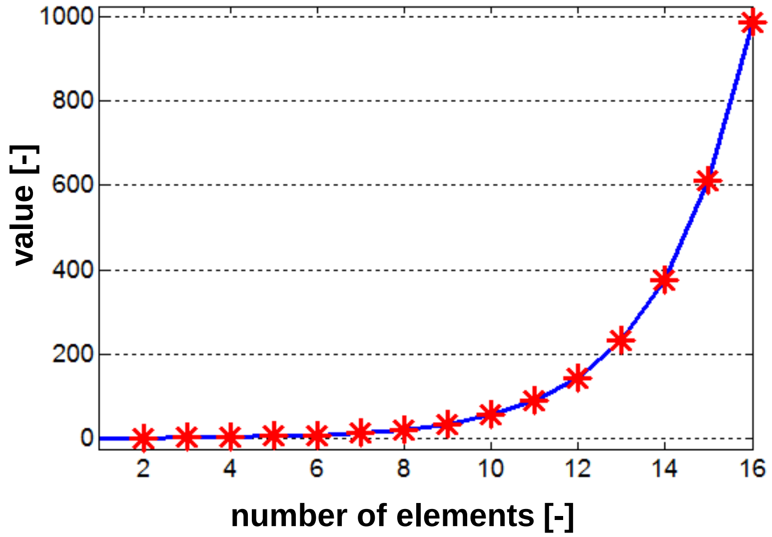


# Exercise #3

240 s

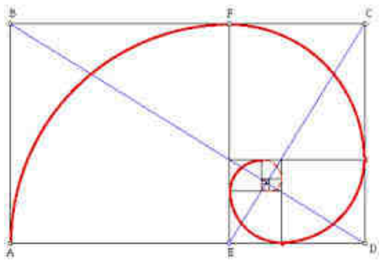
- rate of reproduction of rabbits:

```
%% fibonacci sequence
f = [0 1]; % first two members
n = 1;     % index for series generation
limit = 1000;
while f(n) + f(n+1) < limit
    f(n+2) = f(n) + f(n+1);
    n = n + 1;
end
plot(f);
```




- try to find out the relation of the series to the value of golden ratio
- try to calculate it:

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618033\ldots$$



# Exercise #4

300 s

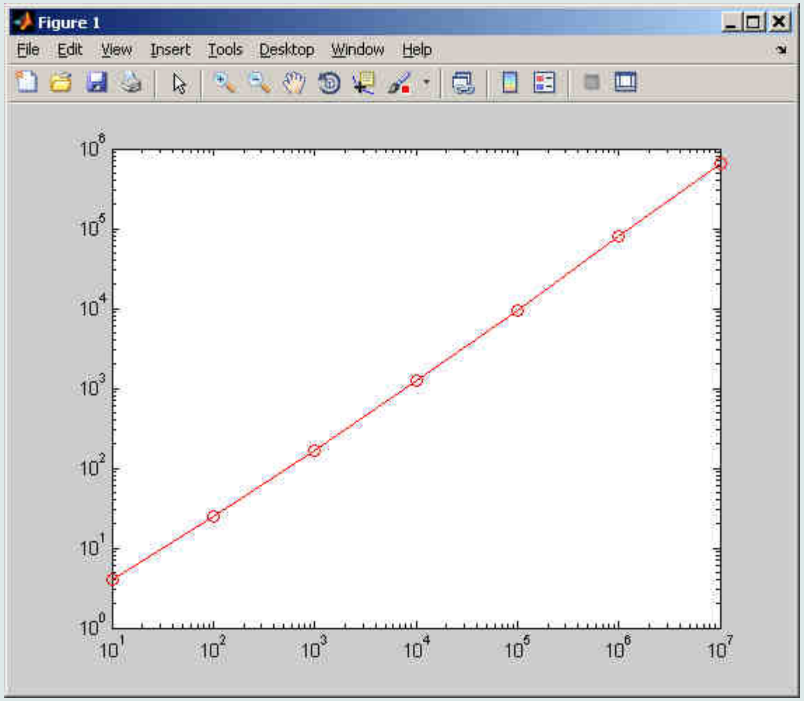
- consider following matrix: $\mathbf{A} = \begin{pmatrix} 1 & 1 & 2 \\ 2 & 3 & 5 \end{pmatrix}$
- write a condition testing whether all elements of **A** are positive and at the same time all elements of the first row are integers
  - if the condition is fulfilled display the result using `disp`
- compare with
  - what is the difference?


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

# Exercise #5

600 s

- try to determine the density of prime numbers
  - examine the function `primes` generating prime numbers
  - for the orders $10^1$ – $10^7$ determine the primes density (i.e. the number of primes up to 10, to 100, …, to $10^7$)
  - outline the dependence using `plot`
  - use logarithmic scale (function `loglog`)
    - how does the plot change?



# Exercise #6

- did you use loop?
- is it advantageous (necessary) to use a loop?
- do you allocate matrices?
- what does, in your view, have the dominant impact on computation time?

# Exercise #7

- the script can be further speeded-up
  - function `primes` is costly and can be run just once:

- would you be able to speed-up the script even more?



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

# Thank you!

ver. 4.2 (25/10/2015)

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



Document created as part of A0B17MTB course.