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

![Graph Image]
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

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

- 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 (\( \text{sort} \times \text{sortrows} \))
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:

```matlab
%% 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“

<table>
<thead>
<tr>
<th>Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>lesser than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>lesser than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>equal to</td>
</tr>
<tr>
<td>~=</td>
<td>not equal to</td>
</tr>
</tbody>
</table>

- 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
Relational operators

- having the vector \( \mathbf{G} = \left( \frac{\pi}{2}, \pi, \frac{3}{2}\pi, 2\pi \right) \), find elements of \( \mathbf{G} \) that are
  - greater than \( \pi \)
  - lesser or equal to \( \pi \)
  - not equal to \( \pi \)

- try similar operations for \( \mathbf{H} = \mathbf{G}^T \) as well

- try to use relational operators in the case of a matrix and scalar as well

- find out whether \( \mathbf{V} \geq \mathbf{U} \):
  \[
  \mathbf{V} = (-\pi, \pi, 1, 0) \\
  \mathbf{U} = (1, 1, 1, 1)
  \]
Relational operators

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

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

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

```
>> (1 > A) <= true
```
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“

- 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
Logical operators – application

- assume a vector of 10 random numbers ranging from -10 to 10
  \[
  \gg \ a = 20*\text{rand}(10, \ 1) - 10
  \]

- following command returns `true` for elements fulfilling the condition:
  \[
  \gg \ a < -5 \ % \ \text{relation operator}
  \]

- following command returns values of those elements fulfilling the condition (logical indexing):
  \[
  \gg \ a(a < -5)
  \]

- following command puts value of -5 to the position of elements fulfilling the condition:
  \[
  \gg \ a(a < -5) = -5
  \]

- following command sets value of the elements in the range from -5 to 5 equal to zero (opposite to thresholding):
  \[
  \gg \ a(a > -5 \ \& \ a < 5) = 0
  \]

- thresholding function (values below -5 sets equal to -5, values above 5 sets equal to 5):
  \[
  \gg \ a(a < -5 \ | \ a > 5) = \text{sign}(a(a < -5 \ | \ a > 5))*5
  \]
Logical operators

- 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 \( \pi \) or are equal to \( 2\pi \)
  - pay attention to the type of the result (= logical values true / false)

- are greater than \( \frac{\pi}{2} \) and at the same time are not equal \( 2\pi \)

- elements from the previous condition add to matrix \( \mathbf{A} \)
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

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

- create matrix $A = \text{magic}(3)$ 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

\[
A = \begin{bmatrix}
8 & 1 & 6 \\
3 & 5 & 7 \\
4 & 9 & 2
\end{bmatrix}
\]

\[
\text{any} \begin{bmatrix}
0 & 1 & 1 \\
1 & 1 & 0 \\
0 & 1 & 1
\end{bmatrix} = (1 \ 1 \ 1), \quad \text{all} \begin{bmatrix}
0 & 1 & 1 \\
1 & 1 & 0 \\
0 & 1 & 1
\end{bmatrix} = (0 \ 1 \ 0), \quad \text{any} \left(\text{all} \begin{bmatrix}
0 & 1 & 1 \\
1 & 1 & 0 \\
0 & 1 & 1
\end{bmatrix}\right) = \text{any}(0 \ 1 \ 0) = 1
\]
Logical operators

- find out the result of following operation and interpret it

\[
\gg \sim(\sim[1\ 2\ 0\ -2\ 0])
\]

- 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!
Matrix indexation using own values

- create matrix A

![Matrix A]

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

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

- 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

```matlab
>> B = A(A)
```
Program branching – loops

- repeating certain operation multiple-times, one of the basic programming techniques
- There are 2 types of cycles in Matlab:
  - 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 programming 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: is a cycle really necessary?
for loop

• for loop is applied to known number of repetitions 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

- 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
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…

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

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

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

```plaintext
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
Typical application of loops

```matlab
%% 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

- 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)?
**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…

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

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

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

  
  \[
  \begin{array}{l}
  A = 0; \\
  \text{for } i = 1:10 \\
  A = A + 1i; \\
  \text{end}
  \end{array}
  \quad
  \begin{array}{l}
  A = 0; \\
  \text{for } i = 1:10 \\
  A = A + i; \\
  \text{end}
  \end{array}
  \quad
  \begin{array}{l}
  A = 0; \\
  \text{for } i = 1:10 \\
  A = A + j; \\
  \text{end}
  \end{array}
  \]

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

  \[
  A = 0; \text{ for } i = 1:10, A = A + 1i; \text{ end},
  \]

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

```matlab
%% 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
```
Loops #3

- fill in the matrix using loops

\[ A(m,n) = \frac{mn}{4} + \frac{m}{2n} \]

- 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 \( \text{pcolor}() \)
Loops #4

- 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!!
Loops #5

- visualize current distribution of a dipole antenna described as
  \[ I(x, t) = I_0(x) e^{-j \omega_0 t}, \quad I_0(x) = \cos(x), \quad \omega_0 = 2\pi \]
- in the interval \( t \in (0, 4\pi) \), \( x \in \left(-\frac{\pi}{2}, \frac{\pi}{2}\right) \) choose \( N = 101 \)

for visualization inside the loop use following piece of code:

```matlab
% ... your code
figure(1);
plot(x, real(I));
axis([x(1) x(end) -1 1]);
pause(0.1);
% ... your code
```
Loops #6

- 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 (\( \text{rand}() \))

- use following parameters

```matlab
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:

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

- try to make the code more efficient
Loops #7

- for comparison it is possible to use Matlab built-in function `filter` 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;
```
break, continue

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

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

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

```matlab
% another code ... 
for k = 1:length(A)
    if A(k) > threshold
        continue;
    end
% another code ...
end
```
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)
Loops vs. vectorizing #2

- The motivation for introduction of JIT was to catch up with 3rd 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 built-in functions are called)
  - When each line of the loop uses JIT
- As the result, some parts of the code don't have to be vectorized (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
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…

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

tic,
mat = NaN(N, N);
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 = 2:N
    mat(n1, n1-1)=3;
end
toc,
% computed in 0.52s (2015b)

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 0.18s (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`

```plaintext
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
```
**if VS. switch**

<table>
<thead>
<tr>
<th>if-elseif-else-end</th>
<th>switch-otherwise-end</th>
</tr>
</thead>
<tbody>
<tr>
<td>it is possible to create very complex structure (&amp; &amp;</td>
<td>simple choice of many options</td>
</tr>
<tr>
<td>(&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td><code>strcmp</code> is used to compare strings of various lengths</td>
<td>test strings directly</td>
</tr>
<tr>
<td>test equality / inequality</td>
<td>test equality only</td>
</tr>
<tr>
<td>great deal of logical expressions is needed in the case of testing many options</td>
<td>enables to easily test one of many options using <code>{}</code></td>
</tr>
</tbody>
</table>
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');
elif c > 10
    disp('even, >10');
end
```
Program branching – if / else / elseif

- generate random numbers
  \[ r = 2 \times \text{rand}(8, 1) - 1; \]

- save the numbers in matrices \( \text{Neq} \) and \( \text{Pos} \) 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 growth in size of matrices \( \text{Pos} \) and \( \text{Neq} \) – how to solve the problem?

- can you come up with a more elegant solution? (for cycle is not always necessary)
Program branching – *if / else / elseif*

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

![Program flow diagram](image)

Program flow

25.10.2015 12:00
### Discussed functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>edit</em></td>
<td>open Matlab Editor</td>
</tr>
<tr>
<td><em>disp, pause</em></td>
<td>display result in command line, pauses code execution</td>
</tr>
<tr>
<td><em>num2str</em></td>
<td>conversion from datatype numeric to char</td>
</tr>
<tr>
<td><em>for-end, while-end</em></td>
<td>loop</td>
</tr>
<tr>
<td><em>factorial</em></td>
<td>calculate factorial</td>
</tr>
<tr>
<td><em>break, continue</em></td>
<td>terminates loop execution, passes control to loop's next iteration</td>
</tr>
<tr>
<td><em>and, or, not, xor</em></td>
<td>functions overloading logical operators</td>
</tr>
<tr>
<td><em>all, any</em></td>
<td>evaluation of logical arrays („all of“, „at least one of“)</td>
</tr>
<tr>
<td><em>sign</em></td>
<td>signum function</td>
</tr>
<tr>
<td><em>if-elseif-else-end</em></td>
<td>branching statement</td>
</tr>
</tbody>
</table>
Exercise #1

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

\[
s_p(t) = \begin{cases} 
  s_{\min} \iff s(t) < s_{\min} & s_{\min} = -\frac{9}{10} \\
  s_{\max} \iff s(t) > s_{\max} & s_{\max} = \frac{\pi}{2} \\
  s(t) \ldots \text{jinak} 
\end{cases}
\]

\begin{verbatim}
N = 5; V = 40;
t = linspace(0, N, N*V);
s_t = randn(1, N*V) + ... + sqrt(2*pi)*sin(2*pi*t);
\end{verbatim}
Exercise #2

- 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

- rate of reproduction of rabbits:

```matlab
%% 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:

\[
\phi = \frac{1 + \sqrt{5}}{2} \approx 1.618033\ldots
\]
Exercise #4

- consider following matrix: \( A = \begin{pmatrix} 1 & 1 & 2 \\ 2 & 3 & 5 \end{pmatrix} \)

- 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?
Exercise #5

- 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?
Thank you!

ver. 4.2 (25/10/2015)
Miloslav Čapek, Pavel Valtr
miloslav.capek@fel.cvut.cz
Pavel.Valtr@fel.cvut.cz

Apart from educational purposes at CTU, this document may be reproduced, stored or transmitted only with the prior permission of the authors. Document created as part of A0B17MTB course.