

# Lecture 2: Vectors & Matrices

BE0B17MTB – Matlab

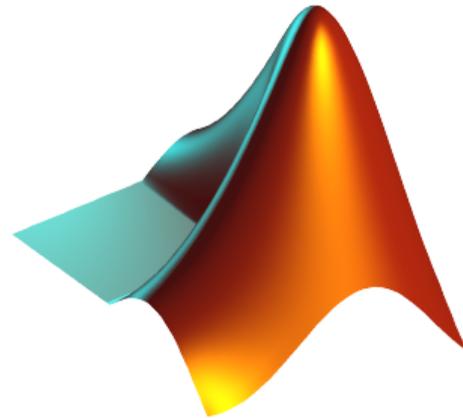
Miloslav Čapek, Viktor Adler, Pavel Valtr, Michal Mašek, and Vít Losenický

Department of Electromagnetic Field  
Czech Technical University in Prague  
Czech Republic  
[matlab@elmag.org](mailto:matlab@elmag.org)

February 26, 2020  
Summer semester 2019/20



1. MATLAB Editor
2. Matrix Creation
3. Operations with Matrices
4. Exercises





# MATLAB Editor

- ▶ It is often required to evaluate certain sequence of commands repeatedly  $\Rightarrow$  utilization of MATLAB scripts (plain ASCII coding).
- ▶ The best option is to use MATLAB Editor,
  - ▶ which can be opened using the following command:

```
>> edit
```

- ▶ A script is a sequence of statements what we have been up to now typing in the command line.
  - ▶ All the statements are executed one by one upon the launch of the script.
  - ▶ The script operates over MATLAB base workspace data.
  - ▶ Scripts are suitable for quick analysis and solving problems involving multiple statements.
- ▶ There are specific naming conventions for scripts (and also for functions as we will see later).

# MATLAB Editor, R2019



```

1
2
3
4 % why(N) provides the N-th answer.
5 % Please embellish or modify this function to suit your own tastes.
6
7 % Copyright 1984-2014 The MathWorks, Inc.
8
9
10 if nargin > 0
11     dftit = rng(m,'Uniform');
12 end
13 switch randi(10)
14     case 1
15         a = special_case;
16     case {2, 3, 4}
17         a = phrase;
18     otherwise
19         a = sentence;
20 end
21 a{1} = upper(a{1});
22 disp(a);
23 if nargin > 0
24     rng(dftit);
25 end
26
27
28
29 function a = special_case
30 switch randi(10)
31     case 1
32         a = 'why not?';
33     case 2
34         a = 'don''t ask!';
35     case 3
36         a = 'it''s your karma.';
37     case 4
38         a = 'stupid question!';
39     case 5
40         a = 'how should I know?';
41     case 6
42         a = 'can you rephrase that?';
43     case 7
44         a = 'it should be obvious.';
45     case 8
46         a = 'the devil made me do it.';
47     case 9
48         a = 'the computer did it.';
49     case 10
50         a = 'the customer is always right.';

```



# Script Execution, m-files

- ▶ To execute script:
  - ▶ F5 function key in Matlab Editor,
  - ▶ Current folder → select script → context menu → Run,
  - ▶ Current folder → select script → F9,
  - ▶ from the command line:

```
>> script_name
```

- ▶ Scripts are stored as so called m-files, `.m`
- ▶ **Caution:** If you have Mathematica installed, the `.m` files may be launched by Mathematica.



# Data in Scripts

- ▶ Scripts can use data located in Workspace.
- ▶ Variables remain in the Workspace even after the calculation is finished.
- ▶ Operations on data in scripts are performed in the base Workspace.
- ▶ MATLAB carries out commands **sequentially**.



# Useful Functions for Script Generation I.

- ▶ Function `disp` displays value of a variable in Command Window.
  - ▶ Without displaying variable's name and the equation sign “=”.
  - ▶ Can be combined with a text (more on that later).
  - ▶ Often it is advantageous to use more complicated but robust function `sprintf`.

```
a = 2^13 - 1;  
b = [8*a 16*a];  
b
```

```
a = 2^13 - 1;  
b = [8*a 16*a];  
disp(b);
```



# Useful Functions for Script Generation II.

- ▶ Function `input` is used to enter variables.
  - ▶ If the function is terminated unexpectedly, the input request is repeated

```
A = input('Enter parameter A: ');
```

- ▶ It is possible to enter strings as well:

```
str = input('Enter String str', 's');
```



# Script Commenting

## ▶ MAKE COMMENTS!!

- ▶ Important/complicated parts of code.
- ▶ Description of functionality, ideas, change of implementation.

## ▶ Typical single-line comment:

```
% create matrix, sum all members  
matX = [1, 2, 3, 4, 5];  
sumX = sum(matX); % sum of matrix
```

## ▶ Multiple-line comment:

```
%{  
This is a multiple-line comment.  
Mostly, it is more appropriate to use more  
single-line comments.  
%}
```

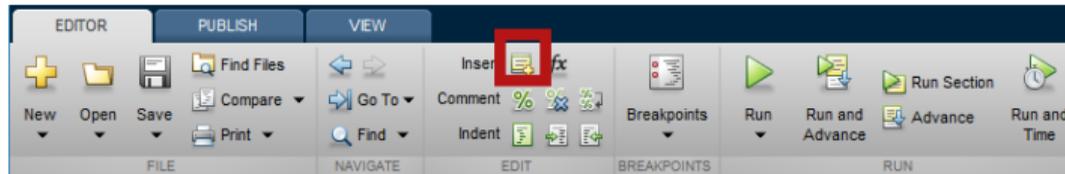
## ▶ Cell mode enables to separate script into more blocks.

```
matX = [1, 2, 3, 4, 5];  
%% CELL mode (must be enabled in Editor)  
sumX = sum(matX);
```



# Cell Mode in MATLAB Editor

- ▶ Cells enable to separate the code into smaller, logically compacted parts.
  - ▶ Separator `%%`.
  - ▶ The separation is visual only, but it is possible to execute a single cell – shortcut **CTRL+ENTER**.





# Entering Matrices Using “:” I.

- ▶ Large vectors and matrices with regularly increasing elements can be typed in using colon operator.

- ▶ **a** is the smallest element (“from”), **x** is increment, **b** is the largest element (“to”)

```
A = a:x:b
```

```
>> A = 1:4:13
A =
    1    5    9   13
```

- ▶ **b** doesn't have to be element of the series.
  - ▶ Last element  $N \cdot x$  then follows the inequality:

$$|a + N \cdot x| \leq |b|$$

```
>> A = 1:4:10
A =
    1    5    9
```

- ▶ If **x** is omitted, the increment is set equal to 1.

```
A = a:x:b
```

```
>> A = 3:8
A =
    3    4    5    6    7    8
```



# Entering Matrices Using “:” II.

- ▶ Using the colon operator “:” create:
  - ▶ Following vectors

$$\mathbf{u} = [1 \ 3 \ \dots \ 99]$$

$$\mathbf{v} = [25 \ 20 \ \dots \ -5]^T$$

- ▶ Matrix

- ▶ Caution, the third column can't be created using colon operator “:” only,

$$\mathbf{T} = \begin{bmatrix} -4 & 1 & \frac{\pi}{2} \\ -5 & 2 & \frac{\pi}{4} \\ -6 & 3 & \frac{\pi}{6} \end{bmatrix}$$

but can be created using “:” and dot operator “.” (*we will see later*).





# Entering Matrices Using linspace, logspace I.

- ▶ Colon operator defines vector with **evenly spaced** points.
- ▶ In the case when **fixed number of elements** of a vector is required, use `linspace`:

```
A = linspace(a, b, N);
```

```
>> A = linspace(0, 2, 5)
A =
    0    0.5000    1.0000    1.5000    2.0000
```

- ▶ When the N parameter is left out, the vector with 100 elements is generated:

```
A = linspace(a, b);
```

- ▶ The function `logspace` works analogically, except that logarithmic scale is used

```
A = logspace(a, b, N);
```



## Entering Matrices Using `linspace`, `logspace` II.

- ▶ Create a vector of 100 evenly spaced points in the interval  $[-1.15, 75.4]$
- ▶ Create a vector of 201 evenly spaced points in the interval  $[-100, 100]$  sorted in descending order.
- ▶ Create a vector with spacing of  $-10$  in the interval  $[-100, 100]$  sorted in descending order.
  - ▶ try both options using `linspace` and colon `:`



# Entering Matrices Using Functions I.

- ▶ Special types of matrices of given sizes are needed quite often.
  - ▶ MATLAB offers a number of functions to serve the purpose.
- ▶ Example: matrix filled with zeros
  - ▶ Will be used frequently.

```
zeros(m)           % matrix of size [m x m]
zeros(m, n)        % matrix of size [m x n]
zeros(m, n, p, ..) % matrix of size [m x n x p x ..]
zeros([m,n])       % matrix of size [m x n]

B = zeros(m, 'single') % matrix of size [m x n], of type 'single'

% see documentation for other options
```



## Entering Matrices Using Functions II.

- ▶ Following useful functions analogical to the `zeros` function are available

---

<code>ones</code>	matrix filled with ones
<code>eye</code>	identity matrix
<code>NaN, Inf</code>	matrix filled with NaN, matrix filled with Inf
<code>magic</code>	matrix suitable for MATLAB experiments, notice its properties
<code>rand, randn, randi</code>	matrix filled with random numbers coming from uniform and normal distribution, matrix filled with uniformly distributed random integers
<code>randperm</code>	returns vector containing random permutation of numbers
<code>diag</code>	creates diagonal matrix or returns diagonal of a matrix
<code>blkdiag</code>	construct block diagonal matrix from input arguments
<code>cat</code>	groups several matrices into one
<code>true, false</code>	creates a matrix of logical ones and zeros

---

- ▶ For further functions see MATLAB → Mathematics → Elementary Mathematics → Constants and Test Matrices.



# Entering Matrices Using Functions III.

- ▶ Create following matrices
  - ▶ use MATLAB functions
  - ▶ begin with matrices you find easy to cope with.

$$\mathbf{M}_1 = \begin{bmatrix} \text{NaN} & \text{NaN} \\ \text{NaN} & \text{NaN} \end{bmatrix}$$

$$\mathbf{M}_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}$$

$$\mathbf{M}_3 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & -5 \end{bmatrix}$$

$$\mathbf{M}_4 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$





## Entering Matrices Using Functions IV.

- ▶ Try to create an empty three-dimensional array of type `double`.
- ▶ Can you find another option?
  - ▶ `empty` is hidden (but public) method of all non-abstract classes in MATLAB.





# Dealing with Sparse Matrices

- ▶ MATLAB provides support for working with sparse matrices.
  - ▶ Most of the elements of sparse matrices are zeros and it pays off to store them in a more efficient manner.

- ▶ To create a sparse matrix **S** out of matrix **A**:

```
S = sparse(A)
```

- ▶ Conversion of a sparse matrix to a full matrix:

```
B = full(S)
```

- ▶ In the case of need see Help for other functions.



# Entering Matrices

- ▶ Quite often, there are several options how to create a given matrix.
  - ▶ It is possible to use an **output of one function as an input of another** function in MATLAB:

- ▶ Consider:

- ▶ clarity,
- ▶ simplicity,
- ▶ speed,
- ▶ convention.

```
plot(diag(randn(10, 1), 1))
```

- ▶ E.g. band matrix with “1” on main diagonal and with “2” and “3” on secondary diagonals.

```
N = 10;
diag(ones(N, 1)) + diag(2 * ones(N - 1, 1), 1) + diag(3 * ones(N - 1, 1), -1)
```

- ▶ Can be done using **for** cycle as well (see later in semester), might be faster ...
- ▶ Some other idea?



# Transpose and Matrix Conjugate

- ▶ Pay attention to situations where the matrix is complex,  $\mathbf{A} \in \mathbb{C}^{M \times N}$ .
- ▶ There are two operations:

transpose	$\mathbf{A}^T = [A_{ij}]^T = [A_{ji}]$	transpose(A) <- don't use	A.'
transpose + conjugate	$\mathbf{A}^H = [A_{ij}]^H = [\mathbf{A}^*]^T$	ctranspose(A) <- don't use	A'

```
>> A = magic(2) + 1j * magic(2)'
```

```
>> A.'
ans =
    1.0000 + 1.0000i    4.0000 + 3.0000i
    3.0000 + 4.0000i    2.0000 + 2.0000i
```

```
>> A'
ans =
    1.0000 - 1.0000i    4.0000 - 3.0000i
    3.0000 - 4.0000i    2.0000 - 2.0000i
```



# Matrix Operations I.

- ▶ There are other useful functions apart from transpose (`transpose`) and matrix diagonal (`diag`):

```
P = magic(4)
```

- ▶ upper triangular matrix,

```
U = triu(P)
```

- ▶ lower triangular matrix,

```
L = tril(P)
```

- ▶ a matrix can be modified taking into account secondary diagonals as well

```
V = triu(P, -1)
```

$$\mathbf{P} = \begin{bmatrix} 16 & 2 & 3 & 13 \\ 5 & 11 & 10 & 8 \\ 9 & 7 & 6 & 12 \\ 4 & 14 & 15 & 1 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} 16 & 2 & 3 & 13 \\ 0 & 11 & 10 & 8 \\ 0 & 0 & 6 & 12 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{L} = \begin{bmatrix} 16 & 0 & 0 & 0 \\ 5 & 11 & 0 & 0 \\ 9 & 7 & 6 & 0 \\ 4 & 14 & 15 & 1 \end{bmatrix}$$

$$\mathbf{V} = \begin{bmatrix} 16 & 2 & 3 & 13 \\ 5 & 11 & 10 & 8 \\ 0 & 7 & 6 & 12 \\ 0 & 0 & 15 & 1 \end{bmatrix}$$



# Matrix Operations II.

- ▶ Function `repmat` is used to copy (part of) a matrix.

```
B = repmat(A, m, n)
```

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \end{bmatrix}$$

```
B = repmat(A, 1, 2)
C = repmat(A, [2 1])
```

$$\mathbf{B} = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{11} & A_{12} & A_{13} \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{11} & A_{12} & A_{13} \end{bmatrix}$$

- ▶ `repmat` is a very fast function.
  - ▶ Comparison of execution time of creation a  $10^4 \times 10^4$  matrix filled with pi (HW, SW and MATLAB version dependent):

```
X = ones(1e4) % computed in 0.71s
Y = repmat(1, 1e4, 1e4) % computed in 0.4s, BUT... don't use it
```

- ▶ It is for you to consider the way of matrix creation...



# Matrix Operations III.

- ▶ Function `reshape` is used to rearrange a matrix

```
B = reshape(A, m, n)
```

- ▶ e.g.

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

```
C = reshape(A, [4, 1])
D = reshape(A, 1, 4)
D = reshape(A, [], 4)
```

$$\mathbf{C} = \begin{bmatrix} A_{11} \\ A_{21} \\ A_{12} \\ A_{22} \end{bmatrix}$$

$$\mathbf{D} = [ A_{11} \quad A_{21} \quad A_{12} \quad A_{22} ]$$



# Matrix Operations IV.

- ▶ Following functions are used to swap the order of

- ▶ columns: `fliplr`,

```
B = fliplr(A)
```

- ▶ rows: `flipud`,

```
C = flipud(A)
```

- ▶ row-wise or column-wise: `flip`.

```
B = flip(A, 1)
C = flip(A, 2)
```

- ▶ The same result is obtained using indexing (*later in the course*).

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} A_{13} & A_{12} & A_{11} \\ A_{23} & A_{22} & A_{21} \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} A_{21} & A_{22} & A_{23} \\ A_{11} & A_{12} & A_{13} \end{bmatrix}$$



# Matrix Operations V.

- ▶ Circular shift is also available.
  - ▶ Can be carried out along an arbitrary dimension (row-wise/column-wise).
  - ▶ Can be carried out in both directions (back/forth).
- ▶ Consider the difference between `flip` and `circshift`.

```
B = circshift(A, -2)
C = circshift(A, [-2 1])
```

$$\mathbf{B} = \begin{bmatrix} A_{31} & A_{32} & A_{33} \\ A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} A_{33} & A_{31} & A_{32} \\ A_{13} & A_{11} & A_{12} \\ A_{23} & A_{21} & A_{22} \end{bmatrix}$$



# Matrix Operations VI.

- Convert matrix  $\mathbf{A}$  into the form of matrices  $\mathbf{A}_1$  to  $\mathbf{A}_4$ .

$$\mathbf{A} = [1 \ \pi; \exp(1) \ -1i]$$

$$\mathbf{A} = \begin{bmatrix} 1 & \pi \\ e & -i \end{bmatrix}$$

- Use `repmat`, `reshape`, `triu`, `tril` and `conj`.

$$\mathbf{A}_1 = \begin{bmatrix} 1 & \pi & 1 & \pi & 1 & \pi \\ e & -i & e & -i & e & -i \end{bmatrix}$$

$$\mathbf{A}_2 = [1 \ \pi \ e \ -i]$$

$$\mathbf{A}_3 = \begin{bmatrix} 1 & \pi \\ e & +i \\ 1 & \pi \\ e & +i \\ 1 & \pi \\ e & +i \end{bmatrix}$$

$$\mathbf{A}_4 = \begin{bmatrix} 1 & \pi & 0 & 0 & 0 & 0 \\ e & -i & e & 0 & 0 & 0 \\ 0 & \pi & 1 & \pi & 0 & 0 \\ 0 & 0 & e & -i & e & 0 \\ 0 & 0 & 0 & \pi & 1 & \pi \\ 0 & 0 & 0 & 0 & e & -i \end{bmatrix}$$

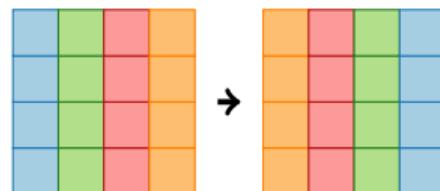


# Matrix Operations VII.

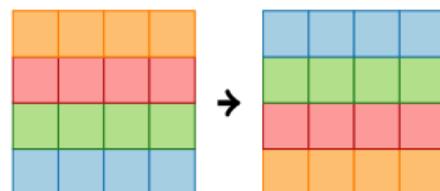
- ▶ Create the following matrix (use advanced techniques)

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 & 1 & 2 & 3 \\ 0 & 2 & 4 & 0 & 2 & 4 \\ 0 & 0 & 5 & 0 & 0 & 5 \end{bmatrix}$$

- ▶ Create matrix **B** by swapping columns in matrix **A**.



- ▶ Create matrix **C** by swapping rows in matrix **B**.





# Matrix Operations VIII.

- ▶ Compare and interpret following commands.

```
x = (1:5).';           % entering vector
x = repmat(x, [1 10]); % 1. option
X = x(:, ones(10, 1)); % 2. option
```

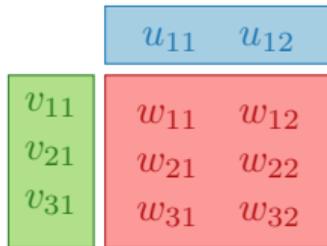
- ▶ repmat is powerful, but not always the most time-efficient function.



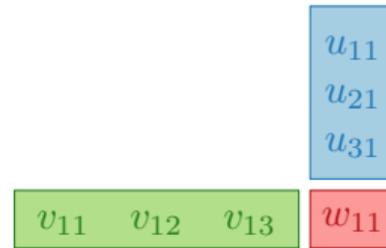
# Vector and Matrix Operations

- ▶ Remember that matrix multiplication is not commutative, i.e.  $\mathbf{AB} \neq \mathbf{BA}$ .
- ▶ Remember that vector  $\times$  vector product results in

$$\mathbf{v}_{M,1} \mathbf{u}_{1,N} = \mathbf{w}_{M,N}$$



$$\mathbf{v}_{1,M} \mathbf{u}_{M,1} = \mathbf{w}_{1,1}$$



...pay attention to the dimensions of matrices!



# Element-by-element Vector Product

- ▶ It is possible to multiply arrays of the same size in the element-by-element manner in MATLAB.
  - ▶ Result of the operation is an array.
  - ▶ Size of all arrays are the same, e.g. in the case of  $1 \times 3$  vectors:

$$\mathbf{a} = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}$$

```
>> a*b
```

 $a_1 \quad a_2 \quad a_3$ 
 $*$ 
 $b_1 \quad b_2 \quad b_3$ 
 $\rightarrow$ 

Error using \*

(Inner matrix dimensions must agree.)

```
>> a.*b
```

 $a_1 \quad a_2 \quad a_3$ 
 $.*$ 
 $b_1 \quad b_2 \quad b_3$ 
 $\rightarrow$ 
 $a_1b_1 \quad a_2b_2 \quad a_3b_3 = [a_i b_i]$



# Element-by-element Matrix Product

- ▶ If element-by-element multiplication of two matrices of the same size is needed, use the `.*` operator.
  - ▶ It is so called *Hadamard product*/*element-wise product*/*Schur product*:  $\mathbf{A} \circ \mathbf{B}$ .
  - ▶ These two cases of multiplication are distinguished:

`>> A*B`

$$\begin{array}{|c|c|} \hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline \end{array} * \begin{array}{|c|c|} \hline B_{11} & B_{12} \\ \hline B_{21} & B_{22} \\ \hline \end{array} \rightarrow \begin{array}{|c|c|} \hline A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ \hline A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \\ \hline \end{array}$$

`>> A.*B`

$$\begin{array}{|c|c|} \hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline \end{array} .* \begin{array}{|c|c|} \hline B_{11} & B_{12} \\ \hline B_{21} & B_{22} \\ \hline \end{array} \rightarrow \begin{array}{|c|c|} \hline A_{11}B_{11} & A_{12}B_{12} \\ \hline A_{21}B_{21} & A_{22}B_{22} \\ \hline \end{array}$$



# Compatible Array Size

- ▶ Since MATLAB version R2016b most two-input (binary) operators support arrays that have *compatible sizes*.
  - ▶ Variables have compatible sizes if their sizes are either the same or one of them is 1 (for all dimensions).
- ▶ Examples:
  - ▶  $\circ$  represents arbitrary two-input element-wise operator (+, -, .\*, ./, &, <, ==, ...).

$$\begin{array}{ccc}
 [2 \times 2] & [2 \times 2] & [2 \times 2] \\
 \begin{array}{|c|c|} \hline & \\ \hline & \\ \hline \end{array} \circ \begin{array}{|c|c|} \hline & \\ \hline & \\ \hline \end{array} = \begin{array}{|c|c|} \hline & \\ \hline & \\ \hline \end{array} &
 \begin{array}{ccc}
 [2 \times 2] & [2 \times 1] & [2 \times 2] \\
 \begin{array}{|c|c|} \hline & \\ \hline & \\ \hline \end{array} \circ \begin{array}{|c|} \hline \\ \hline \\ \hline \end{array} = \begin{array}{|c|c|} \hline & \\ \hline & \\ \hline \end{array} &
 \begin{array}{ccc}
 [2 \times 2] & [1 \times 1] & [2 \times 2] \\
 \begin{array}{|c|c|} \hline & \\ \hline & \\ \hline \end{array} \circ \begin{array}{|c|} \hline \\ \hline \end{array} = \begin{array}{|c|c|} \hline & \\ \hline & \\ \hline \end{array}
 \end{array}
 \end{array}$$

$$\begin{array}{ccc}
 [3 \times 1] & [1 \times 2] & [3 \times 2] \\
 \begin{array}{|c|} \hline \\ \hline \\ \hline \end{array} \circ \begin{array}{|c|c|} \hline & \\ \hline & \\ \hline \end{array} = \begin{array}{|c|c|} \hline & \\ \hline & \\ \hline & \\ \hline \end{array} &
 \begin{array}{ccc}
 [4 \times 3 \times 1] & [1 \times 3 \times 3] & [4 \times 3 \times 3] \\
 \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \circ \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline & & \\ \hline \end{array}
 \end{array}$$

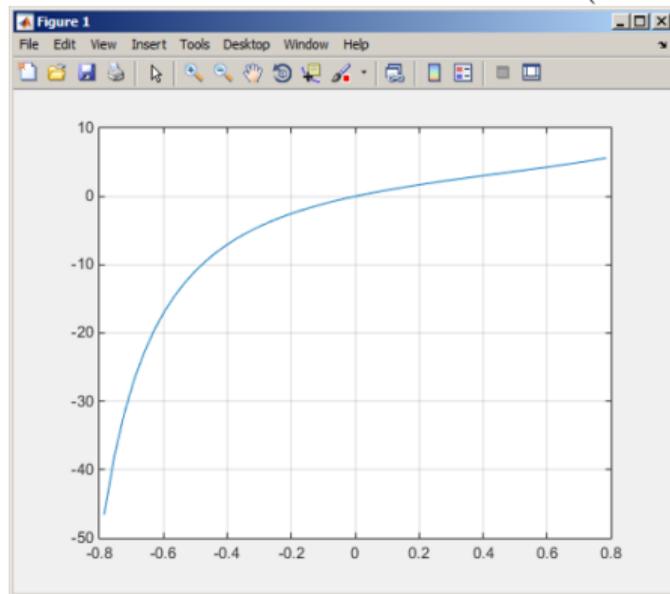


# Element-wise Operations I.

- ▶ Elements-wise operations can be applied to vectors as well in MATLAB. Element-wise operations can be usefully combined with vector functions.
- ▶ It is possible, quite often, to eliminate 1 or even 2 for-loops!!!
- ▶ These operations are exceptionally efficient → allow use of so called **vectorization** (see later).

$$f(x) = \frac{10}{(x+1)} \tan(x), \quad x \in \left[-\frac{\pi}{4}, \frac{\pi}{4}\right]$$

```
x = -pi/4:pi/100:pi/4;
fx = 10 ./ (1 + x) .* tan(x);
plot(x, fx)
grid on
```





## Element-wise Operations II.

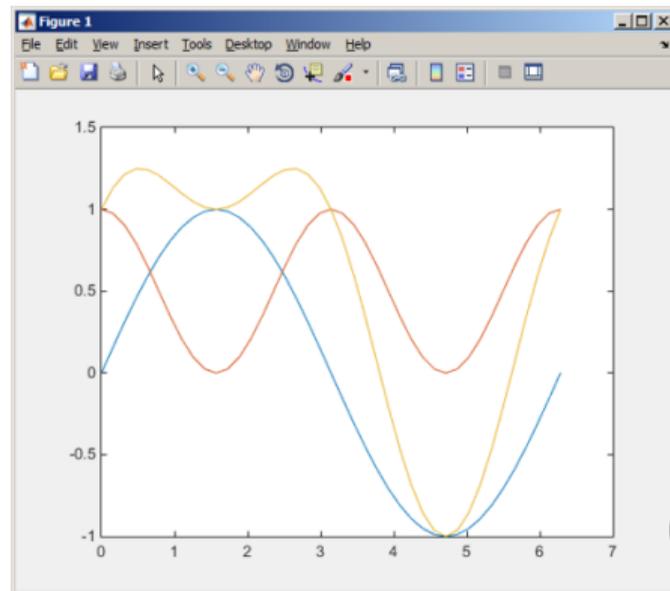
- ▶ Evaluate functions of the variable  $x \in [0, 2\pi]$ :
- ▶ Evaluate the functions in evenly spaced points of the interval, the spacing is  $\Delta x = \pi/20$ .
- ▶ For verification use:
 

```
plot(x, f1, x, f2, x, f3)
```

$$f_1(x) = \sin(x)$$

$$f_2(x) = \cos^2(x)$$

$$f_3(x) = f_1(x) + f_2(x)$$





## Element-wise Operations III.

- ▶ Depict graphically following functional dependency in the interval  $x \in [0, 5\pi]$ .
- ▶ Plot the result using the following function:

$$f_4(x) = \frac{-\cos(3x)}{\cos(x) \sin\left(x - \frac{\pi}{5}\right) - \pi}$$

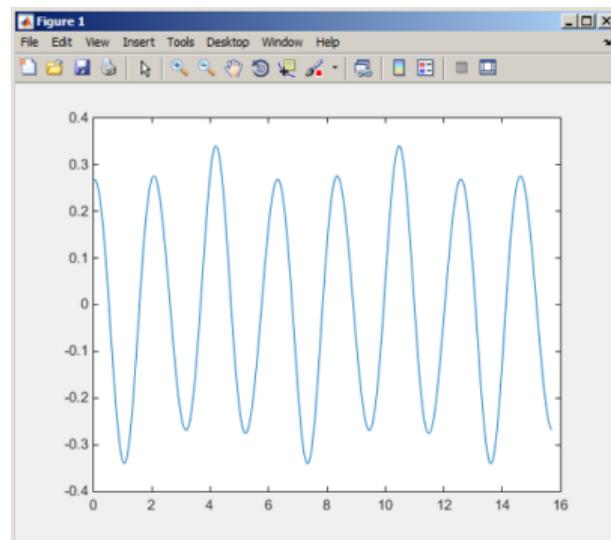
```
plot(x, f4)
```

- ▶ Explain the difference in the way of multiplication of matrices of the same size.

```
>> A*B
```

```
>> A.*B
```

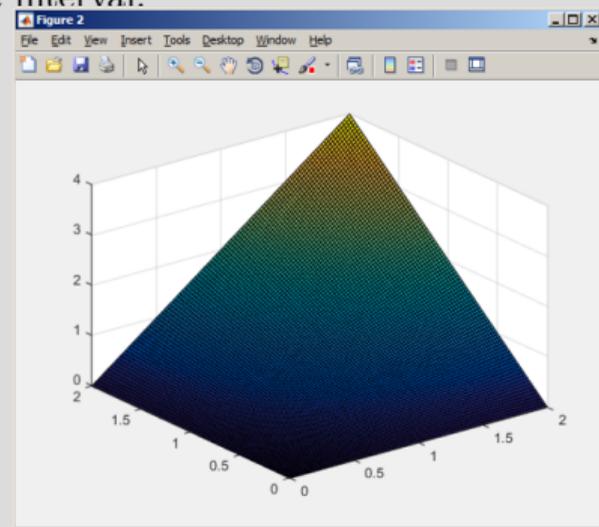
```
>> A' .*B
```





# Element-wise operation IV.

- ▶ Evaluate the function  $f(x, y) = xy$ ,  $x, y \in [0, 2]$ , use 101 evenly spaced points in both  $x$  and  $y$ .
- ▶ The evaluation can be carried out either using vectors, matrix element-wise vectorization or using two for loops.
  - ▶ Plot the result using `surf(x, y, f)`.
  - ▶ When ready, try also  $f(x, y) = x^{0.5}y^2$  on the same interval.

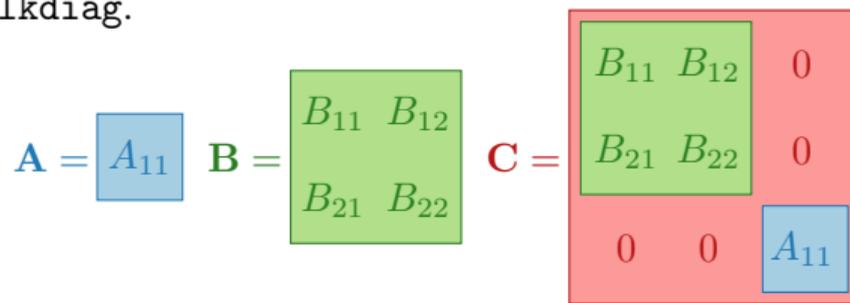




# Matrix Operations

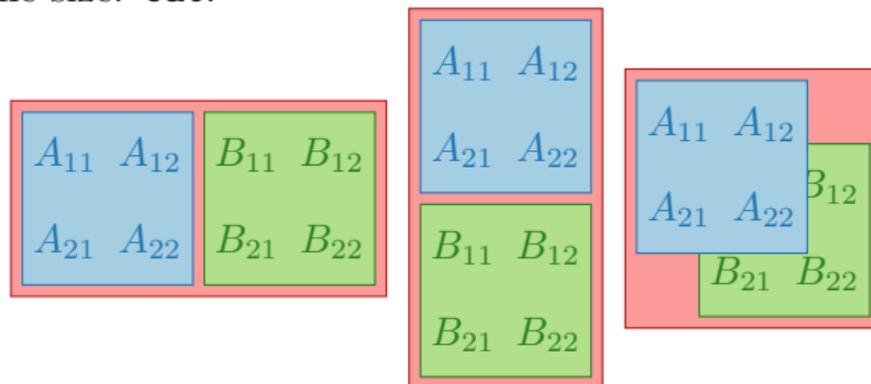
- ▶ Construct block diagonal matrix: `blkdiag`.

```
A = 1;
B = [2 3; -4 -5];
C = blkdiag(B, A);
```



- ▶ Arranging two matrices of the same size: `cat`.

```
C = cat(2, A, B)
C = cat(1, A, B)
C = cat(3, A, B)
```





# Size of Matrices and Other Structures I.

- ▶ It is often needed to know sizes of matrices and arrays.
- ▶ Function `size` returns vector giving the size of a matrix/array.

```
A = randn(3, 5);
d = size(A) % d = [3 5]
```

- ▶ Function `length` returns largest dimension of an array.

```
length(A) = max(size(A))
```

```
A = randn(3, 5, 8);
e = length(A) % e = 8
```

- ▶ Function `ndims` returns number of dimensions of a matrix/array.

```
ndims(A) = length(size(A))
```

```
m = ndims(A) % m = 3
```

- ▶ Function `numel` returns number of elements of a matrix/array.

```
numel{A} = prod(size(A))
```

```
n = numel(A) % n = 120
```



## Size of Matrices and Other Structures II.

- ▶ Create an arbitrary 3D array.
  - ▶ You can make use of the following commands:

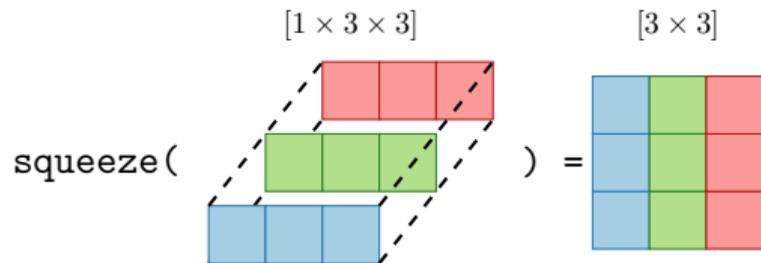
```
A = rand(2 + randi(10), 3 + randi(5));  
A = cat(3, A, flipud(fliplr(A)))
```

- ▶ And now:
  - ▶ Find out the size of A.
  - ▶ Find out the number of elements of A.
  - ▶ Find out the number of elements of A in the “longest” dimension.
  - ▶ Find out the number of dimensions of A.



# Squeeze

- ▶ Function `squeeze` removes dimension of an array with length 1.
  - ▶ If the input is scalar, vector or array without any dimension of the length 1, the output is identical to the input.



$$\text{squeeze}(\text{[square]}) = \text{[square]}$$

$$\text{squeeze}(\text{[vertical vector]}) = \text{[vertical vector]}$$

$$\text{squeeze}(\text{[2x2 grid]}) = \text{[2x2 grid]}$$



# Function gallery

- ▶ Function enabling to create a vast set of matrices that can be used for MATLAB code testing.
- ▶ Most of the matrices are special-purpose.
  - ▶ Function `gallery` offers significant coding time reduction for advanced MATLAB users.
- ▶ See: `help gallery` or `doc gallery`
- ▶ Try for instance:

```
gallery('pei', 5, 4)  
gallery('leslie', 10)  
gallery('clement', 8)
```

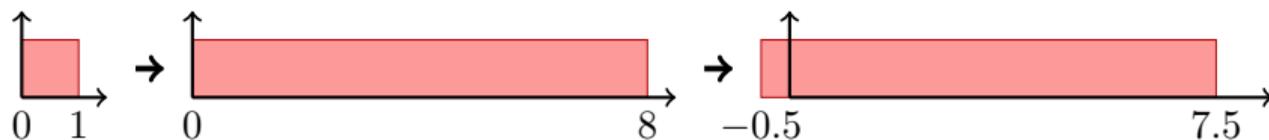
# Exercises



## Exercise I.

- ▶ Create matrix  $\mathbf{M}$  of size  $\text{size}(\mathbf{M}) = [3 \ 4 \ 2]$  containing random numbers coming from uniform distribution on the interval  $[-0.5, 7.5]$ .

$$I(x) = (I_{\max} - I_{\min}) \text{rand}(\dots) + I_{\min}$$





## Exercise II.

- Consider the operation  $\mathbf{a1}^{\wedge}\mathbf{a2}$ . Is this operation applicable to the following cases?

$\mathbf{a1}$ – matrix	$\mathbf{a2}$ – scalar
$\mathbf{a1}$ – matrix	$\mathbf{a2}$ – matrix
$\mathbf{a1}$ – matrix	$\mathbf{a2}$ – vector
$\mathbf{a1}$ – scalar	$\mathbf{a2}$ – scalar
$\mathbf{a1}$ – scalar	$\mathbf{a2}$ – matrix
$\mathbf{a1}, \mathbf{a2}$ – matrix	$\mathbf{a1}.\wedge\mathbf{a2}$

You can always create the matrices  $\mathbf{a1}$ ,  $\mathbf{a2}$  and make a test ...



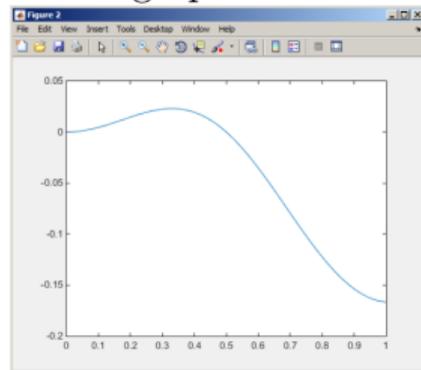


## Exercise III.

- ▶ Make corrections to the following piece of code to get values of the function  $f(x)$  for 200 points on the interval  $[0, 1]$ :

$$f(x) = \frac{x^2 \cos(\pi x)}{(x^3 + 1)(x + 2)}$$

- ▶ Find out the value of the function for  $x = 1$  by direct accessing the vector.
- ▶ What is the value of the function for  $x = 2$ ?
- ▶ To check, plot the graph of the function  $f(x)$ .





## Exercise IV.

- ▶ Create a random matrix  $\mathbf{M}$  of size  $N \times N$  containing only 0 and 1 elements.
- ▶ Compute the percentage of 0 elements in matrix.
- ▶ Compute number of 1 elements on the matrix main diagonal.





## Exercise V.a

- ▶ A proton, carrying a charge of  $Q = 1.602 \cdot 10^{-19}$  C with a mass of  $m = 1.673 \cdot 10^{-31}$  kg enters a homogeneous magnetic and electric field in the direction of the  $z$  axis in the way that the proton follows a helical path; the initial velocity of the proton is  $v_0 = 1 \cdot 10^7$  ms<sup>-1</sup>. The intensity of the magnetic field is  $B = 0.1$  T, the intensity of the electric field is  $E = 1 \cdot 10^5$  Vm<sup>-1</sup>
- ▶ Velocity of the proton among the  $z$  axis is  $v = \frac{QE}{m}t + v_0$ ,
  - ▶ where  $t$  is time, traveled distance along the  $z$  axis is  $z = \frac{1}{2} \frac{QE}{m}t^2 + v_0t$ ,
  - ▶ radius of the helix is  $r = \frac{vm}{BQ}$ ,
  - ▶ frequency of orbiting the helix is  $f = \frac{v}{2\pi r}$ ,
  - ▶ the  $x$  and  $y$  coordinates of the proton are  $x = r \cos(2\pi ft)$ ,  $y = r \sin(2\pi ft)$ .

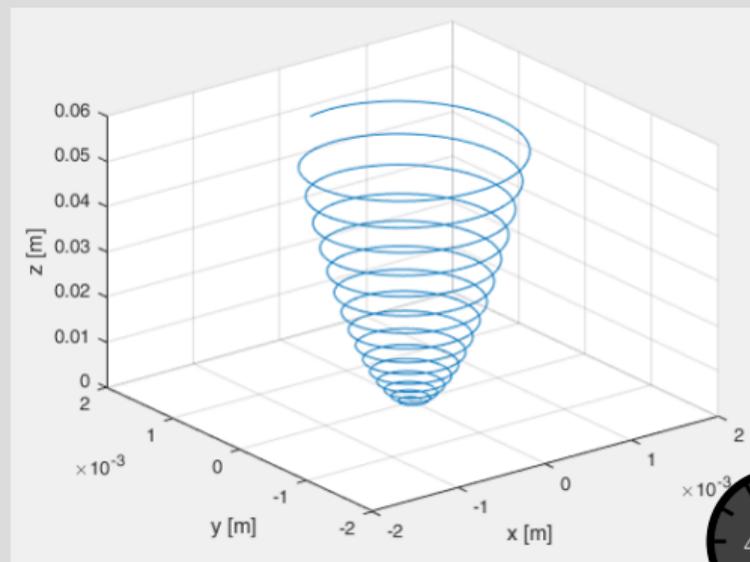


## Exercise V.b

- Plot the path of the proton in space in the time interval from 0 ns to 1 ns in 1001 points using function `comet3(x, y, z)`.

```
clear; clc; close all;
```

```
comet3(x, y, z);
```



# Questions?

BE0B17MTB – Matlab  
matlab@elmag.org

February 26, 2020  
Summer semester 2019/20

---

This document has been created as a part of BE0B17MTB course.  
Apart from educational purposes at CTU in Prague, this document may be reproduced, stored, or transmitted only with the prior permission of the authors.

Acknowledgement: Filip Kozak.