Introduction to MATLAB

Operations on scalar variables

```matlab
global a
>> a=16
    a =
         16
Put attention to the response from the workspace
>> b=2
    b =
         2
>> a+b
    ans =
         18
Attributing the results to the newly defined variable ans
>> c=a+b
    c =
         18
>> c=a*b
    c =
         32
```

Complex Numbers

```matlab
>> c=1+2*i    (%or 2*j)
    c =
         1.0000+2.0000i
```

Vectors

Separating elements by comma or blanks:

```matlab
>> a=[16,b,3,13]
    a =
         16     2     3    13
Note that MATLAB defines vectors as row vectors by default!
>> a*5
    ans =
         80    10    15    65
>> b=[5 11 10 8]
    b =
         5    11    10    8
>> a+b
    ans =
         21    13    13    21
MATLAB treats the orientations properly! Algebraically, multiplying two row vectors is not possible:
>> a*b
    ??? Error using ==> *
    Inner matrix dimensions must agree.
```

Transpose (vectors are actually understood as matrices)

```matlab
>> a.'
    ans =
         16
         2
         3
```
>> a.'*b
   ans =
     80   176   160   128
     10    22    20    16
     15    33    30    24
     65   143   130   104

>> c = [1 i; -i 1]
c =
     1   0+1i
     0-1i   1
>> c.'
   ans =
     1   0-1i
     0+1i   1
>> c'
   ans =
     1   0+1i
     0-1i   1

Note that the .' command gives the transpose whereas the ` command gives the hermitian transpose of a matrix!

Scalar product
>> b*a.
   ans =
     236

Element by element operation – the dot operator
>> a.*b
   ans =
     80   22    30   104
>> a^2
   Error using ^
   Inputs must be a scalar and a square matrix.
   To compute elementwise POWER, use POWER (.^) instead.

Note the difference: a is treated as vector!
>> a.^2
   ans =
     256    4    9   169

Every element is squared now!

Access to single elements of the vector
>> a(1)+a(2)+a(3)+a(4)
   ans =
     34
>> a(5)
   ??? Index exceeds matrix dimensions.
>> a(6)=7
   a =
     16    2    3   13    0    7
>> c=[a,b]
c =
     16    2    3   13    0    7    5   11   10    8

Note that memory is allocated dynamically!
Special keyword end to address last element of an array

Deleting an element of a vector

\[
>> a(6) = []
\]

\[
a = \begin{bmatrix} 16 & 2 & 3 & 13 & 0 & 5 & 11 & 10 & 8 \end{bmatrix}
\]

[] itself is an empty vector

The colon (range) operator with vectors

\[
>> 1:10
\]

\[
ans = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{bmatrix}
\]

\[
>> 1:2:10
\]

\[
ans = \begin{bmatrix} 1 & 3 & 5 & 7 & 9 \end{bmatrix}
\]

\[
>> a = a(1:4)
\]

\[
a = \begin{bmatrix} 16 & 2 & 3 & 13 \end{bmatrix}
\]

\[
>> a(:)
\]

\[
ans = \begin{bmatrix} 16 & 2 & 3 & 13 \end{bmatrix}
\]

Note that a is put out as column vector here although it is a row vector!

\[
>> alpha = (-pi:pi/10:pi)
\]

\[
alpha = \begin{bmatrix}
-3.1416 & -2.8274 & -2.5133 & -2.1991 & -1.8850 & -1.5708 & -1.2566 & -0.9425 & -0.6283 & -0.3142 \\
0 & 0.3142 & 0.6283 & 0.9425 & 1.2566 & 1.5708 & 1.8850 & 2.1991 & 2.5133 & 2.8274 \\
2.1991 & 2.5133 & 2.8274 & 3.1416 \\
\end{bmatrix}
\]

Use of the colon operator together with ‘end’ to put out every fifth element of alpha:

\[
>> alpha(1:5:end)
\]

\[
ans = \begin{bmatrix} -3.1416 & -1.5708 & 0 & 1.5708 & 3.1416 \end{bmatrix}
\]

Suppressing the output on the screen – the semicolon operator

\[
>> alpha = (-pi:pi/10:pi);
\]

Mathematical functions

\[
>> \text{sin}(\pi/2)
\]

\[
ans = \begin{bmatrix} 1 \end{bmatrix}
\]

All built-in mathematical functions work on whole arrays! This is what makes MATLAB fast and easy to use in many cases:

\[
>> s = \text{sin}(alpha);
\]

\[
>> c = \text{cos}(alpha);
\]
Displaying results using the PLOT-function

```matlab
>> plot(s)
>> plot(alpha,s)
>> plot(s,c)
>> plot(s,c,'o')
>> plot(s,c,s*2,c*2)
>> plot(s,c,'o',s*2,c*2)
>> plot(s,c,'o',s*2,c*2,'m')
```

Retrieving former inputs using the CURSOR UP

```matlab
>> plot(s,c,'o',s*2,c*2,'--b')
```

What is the impact of --b? Modifying the plot properties to a large extent interactive in the window Mark at first with the arrow!

```matlab
>> axis equal
```

This will restore the natural aspect ratio in the plot

Matrices

```matlab
>> A=[1 2;3 4]
A =
    1     2
    3     4
>> A=[a; b; 9 7 6 12; 4 14 15 1]
A =
    16     2     3    13
    5    11    10     8
    9     7     6    12
    4    14    15     1
```

Accessing single elements of a matrix

*Sum of the first column*

```matlab
>> A(1,1)+A(2,1)+A(3,1)+A(4,1)
ans =
   34
```

*Sequential identification*

```matlab
>> A(1)+A(2)+A(3)+A(4)
ans =
   34
```

*Your own task: Sum of the second row*

```matlab
>> A(2,1)+A(2,2)+A(2,3)+A(2,4)
ans =
   34
```

*Your own task: Sum of the third column*

```matlab
>> A(1,3)+A(2,3)+A(3,3)+A(4,3)
ans =
   34
```

*Sum of all columns*

```matlab
>> sum(A)
ans =
   34   34   34   34
```

*Sum of all rows (transposing the matrix at first)*

```matlab
>> sum(A.)
ans =
   34   34   34   34
```

Task: Verify that A is a magic square - sum of the diagonals, by retrieving the former command:
>> sum(diag(A))
   ans =
       34

Task: Verify that A is a magic square - sum of the secondary diagonal (flipping the matrix left and right and first):
>> sum(diag(fliplr(A)))
   ans =
       34

The colon operator
Put out only the second column of a matrix:
>> A(:,2)
   ans =
       2
       11
       7
       14

... or the third row:
>> A(3,:)
   ans =
       9
       7
       6
       12

Exchanging the two central columns:
>> A=A(:,[1 3 2 4])
   A =
       16     3     2    13
       5    10    11     8
       9     6     7    12
       4    15    14     1

Removing the second column:
>> A(:,2)=[]
   A =
       16     2    13
       5    11     8
       9     7    12
       4    14     1

Removing the last two rows:
>> A(end-1:end,:) = []
   A =
       16     2    13
       5    11     8

Concatenation
Internally, all matrices and vectors are stored as arrays. With the colon operator, you restore this original form (note that this always is a column vector!):
>> A(:)
   ans =
       16
       5
       2
       11
       13
       8

It is also possible to address elements by its concatenated (array) index:
>> A(5)
   ans =
You can restore the matrix form from a linear array by using `reshape`:

```matlab
>> reshape([1 5 3 2], 2, 2)
ans =
 1   3
 5   2
```

**Linear algebra**

*Constructing a magic square using an built-in function*

```matlab
>> A=magic(4)
A =
 16   2   3  13
  5  11  10   8
  9   7   6  12
  4  14  15   1
```

*Sum of a square magic matrix with its transposed counterpart => symmetry*

```matlab
>> A+A.'
ans =
 32   7  12  17
  7  22  17  22
 12  17  12  27
 17  22  27   2
```

*Inverting a matrix*

```matlab
>> inv(A)
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND = 1.306145e-017.
an =
 1.0e+014 *
 0.9382   2.8147  -2.8147  -0.9382
 2.8147   8.4442  -8.4442  -2.8147
-2.8147  -8.4442   8.4442   2.8147
-0.9382  -2.8147   2.8147   0.9382
```

*Singular matrix, but is difficult for the program to unambiguously recognize the matrix to be singular due to rounding errors*

*Verification of the hypothesis by determining the determinant of the matrix*

```matlab
>> det(A)
an =
-1.4495e-12
```

What are the eigenvalues of a singular matrix?

```matlab
>> eig(A)
an =
 34.0000
  8.9443
-8.9443
  0.0000
```

*Correct! One of them is numerically close to zero. But why the magic 34 is appearing once again?*

*A unity vector belongs to one of the eigenvectors.*

*Verification:*

```matlab
>> eigv=ones(4,1)
eigv =
 1
 1
 1
 1
```
\[ A \times \text{eigv} \]

\[
\begin{array}{c}
\text{ans} = \\
34 \\
34 \\
34 \\
34 \\
\end{array}
\]

Exactly! The unit vector is the eigenvector to which the eigenvalue 34 is associated.

2D graphics with functions like `image`, `mesh` and `surf`

\[ A = \text{magic}(8) \]

\[
\begin{array}{c}
A = \\
\end{array}
\]

\[ \text{image}(A) \]

\[ \text{colorbar} \]

Displaying a quasi-continuous function

Generation of a general grid to operate on

\[ [X,Y] = \text{meshgrid}(-8:0.5:8,-8:0.5:8); \]

\[ R = \text{sqrt}(X.^2+Y.^2); \]

\[ Z = \text{sin}(R)./R; \]

Note that \( Z \) contains a singular value at \( Z(17,17) \). It is represented as NaN ("Not a Number") in MATLAB and occurs due to division by zero.

Correcting the singularity by adding the smallest possible number that can be represented by the computer (keyword `eps`):

\[ R = \text{sqrt}(X.^2+Y.^2)+\text{eps}; \]

\[ Z = \text{sin}(R)./R; \]

\[ \text{mesh}(X,Y,Z) \]

Making it more beautiful:

\[ \text{surf}(X,Y,Z) \]

\[ \text{surf}(X,Y,Z, 'FaceColor','red','EdgeColor','none') \]

\[ \text{camlight left} \]

\[ \text{lighting phong} \]

\[ \text{view}(-15,65) \]

Ah, an awesome gorgeous picture!
(But keep in mind science is not just about gorgeous pictures!)

Saving the results

`save` Saves all variables of the current workspace into the file `matlab.mat`

`load` Reads the file `matlab.mat`

Example:

\[ \text{save} \]

This saves all workspace variables to the default `matlab.mat` which can be restored by `load`:

\[ \text{clear} \]

\[ \text{load} \]

\[ \text{save('Filename','Variable','Format')} \]

Saves a certain variable into a file

Example: Write content of variable ‘a’ into file in ASCII format

\[ \text{save ('test.dat','a','-ASCII')} \]

Open the file in a text editor and check its content!

Please note that advanced methods of writing data exist like `dlmwrite`. 

Functions

Defining an own function

Generation of a new m-file in the m-file-editor with the following contents:

```matlab
function summe=addi(s1,s2)
% Calculating the sum of two numbers
summe=s1+s2;
end
```

1. row: Keyword (function)
   Outputs (summe)
   =
   Name of the function (addi)
   Input arguments (s1 and s2)

2. row: Comments which get displayed upon entering `help addi`.

3. row: Main body of the function. Here, the output argument has to be defined somewhere.

4. row: `keyword end` that corresponds to the keyword `function` and indicates its end.

Saving the m-file with the name `addi.m` in your current working directory.
And then try it out by entering:

```matlab
>> addi(1,2)
ans =
    3
```

Functions as arguments – the @ operator

If functions shall be applied on any other arbitrary functions, the latter have to be defined in an appropriate manner as input arguments. A ‘function handle’ is used for this purpose, which is defined as

```matlab
>>fhandle = @sin
```

The declaration of the function has to include the handle as an input argument.

Within the function, the input function is called by using the fhandle and invoking the function `feval` with ‘X’ as the input argument for the called function:

```matlab
function x = myplot(fhandle, X)
% myplot(fhandle, X) plots a function by its function handle
% in the range given by X
plot( X, feval(fhandle, X) )
end
```

Save the file and try it via:

```matlab
>> myplot(@sin,-pi:0.01:pi)
```

Controlling the flow of a program

Take a look on the introductory presentation for details on the different possibilities to control the flow of a program (if, while, for etc.)

The file includes as task that should be implemented by the students to program a bisection method.

Test the function bise in the file bise.m

```matlab
>> bise(@cos,1.0,2.0)
ans =
    1.57 ...
```