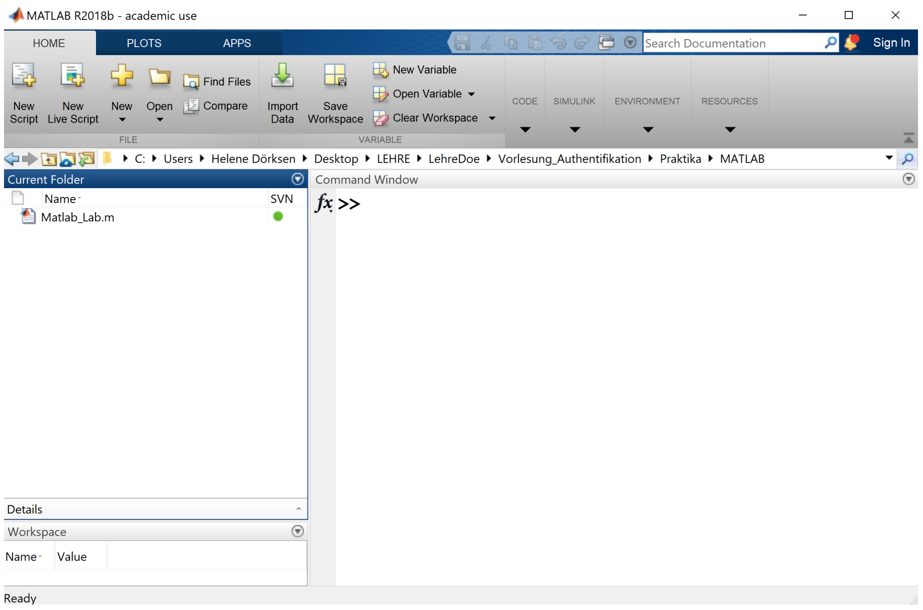
Technische Hochschule Ostwestfalen-Lippe
Fachbereich Elektrotechnik und Technische Informatik
Prof. Dr. Helene Dörksen

MATLAB Introduction

1. **User Interface**

   - **Command Window:** The commands and variable assignments are entered here. Test $2 + 3$ or $a = 2, b = 3, a + b$ or *help ans*, each with <return>.
   - **Current Folder:** This is the current directory.
   - **Workspace:** Here you can see the currently used variables and parameters.



2. **Basic Arithmetic**
   The basic arithmetic operations include addition $+$, subtraction $-$, multiplication $*$, division $/$ and power generation $\hat{}$. Variables do not have to be declared separately. They are created by assigning a value, and their values can be transferred to other variables.

   By default, the result of a command is output directly. This can be suppressed by terminating the command with a semicolon. If the result of a command is not saved in a variable, it is automatically written to the variable *ans*.

Several commands can be written on a line separated by commas (with output of the results) or semicolons (without output of the results). Only the last result is saved in *ans*.

Try it yourself:

$>> c = 1$

$>> x = 3 * c - 4;$

$>> x$

$>> r = x\hat{\ }3$

$>> r + c$

Complex numbers are entered by <real part> + <imaginary part> i, e.g. $2 + 0.5\,i$. For this reason, the imaginary unit i should not be overwritten by a variable, since otherwise there may be problems with complex numbers. MATLAB automatically uses complex numbers when necessary. For example, $\sqrt{-1}$ would be correctly interpreted as a complex number. In many other programming languages, this call would generate an error message.

3. **Math Functions**
Examples for functions are the sine function *sin*, cosine function *cos*, exponential function *exp*, logarithmic function *log*, root function *sqrt*. These functions are applied to numbers with parentheses. Try it yourself:

$>> sin(pi/2)$

$>> a = sin(pi/4);$

$>> a$

$>> sqrt(2)$

Other functions are polynomial functions or composites of functions. Such functions can be easily explained as so-called *anonymous functions*. In the following example we explain such an anonymous function f and apply it to different numbers:

$>> f = @(x)x\hat{\ }2 + sin(x\hat{\ }2 + pi/2)$

$>> f(0)$

$>> f(pi)$

We can also define functions in several variables in this way, such as a function g in the variables x and y:

$>> g = @(x, y)x\hat{\ }2 - y\hat{\ }2$

$>> g(1, 2)$

>> $g(2,1)$

MATLAB distinguishes between upper and lower case. The command $sin(pi/2)$ will be interpreted correctly, but entering $Sin(pi/2)$ returns an error message.

MATLAB provides a lot of functions. Try:

>> *help elfun*

4. **Useful little things**

   - *clc*: this clears the Command Window.
   - *clear*: this deletes all variables in the workspace. Of course, individual variables can also be deleted. For example, *clear a* deletes the variable *a*.
   - The key combination Ctrl + C aborts MATLAB (mostly) if you e.g. have created an endless loop.
   - *help*: this calls the help, e.g. *help sin* or *help clear*.
   - If *help* is not sufficient, use *doc*, e.g. *doc sin*.
   - Number formats in MATLAB: With $format$ the number format can be changed:
     - $format$ *short*, e.g. 0.3212.
     - $format$ *long*, e.g 0.321234276512387.
     - $format$ *rat*, e.g. $a = 1457/536$.

5. **Generate vectors or matrices**

   We understand column vectors as single-column matrices and row vectors as single-line matrices. So we can present any vector by a matrix. The zero matrix, the unit matrix and ones matrix can be created using the functions $zeros$, *eye* and *ones*. All have the same syntax. For example, $zeros(m,n)$ or $zeros([m,n])$ creates an $m \times n$ zero matrix, while $zeros(n)$ generates an $n \times n$ zero matrix.

   >> $zeros(2)$

   >> $ones(2,3)$

   >> $eye(3,2)$

   Rand is used to create matrices with pseudorandom numbers as entries. The syntax is the same as for *eye*. Without an argument, the function returns a single random number.

   >> $rand$

   >> $rand(3)$

   >> $rand(2,3)$

The *diag* function can be used to create diagonal matrices. For a vector x, $diag(x)$ creates a diagonal matrix with the diagonal x.

$$>> diag([1\ 2\ 3])$$

$diag(x,k)$ places x above the main diagonal for $k > 0$, below the main diagonal for $k < 0$ ($k = 0$ denotes the main diagonal).

$$>> diag([1\ 2], 1)$$
$$>> diag([3\ 4], -2)$$

Matrices can be created explicitly using the square bracket notation. For example, a 3×3 matrix with the first 9 prime numbers can be created with the following command:

```
>> A = [2 3 5
        7 11 13
        17 19 23]
```

The end of a line can be specified using a semicolon instead of a line break. On shorter variant of the last example is:

$$>> A = [2\ 3\ 5; 7\ 11\ 13; 17\ 19\ 23]$$

Individual elements within a line can be separated by a space or a comma. It should be noted that when specifying the sign for the individual entries no space between the sign and the element may be left. MATLAB interprets the sign otherwise as a plus or minus operator.

$$>> v = [-1\ 2\ \ -3\ 4]$$
$$>> w = [-1, 2, -3, 4]$$
$$>> x = [-1\ 2\ \ -\ \ 3\ 4]$$

MATLAB has functions to create very special matrices. E.g. quadratic $n \times n$ matrices that consist only of the numbers $1, \dots, n^2$, with the equal both row and column sums as well as the sum of the diagonals elements, are called *magic* matrices.

$$>> magic(3)$$

Over fifty more special and famous matrices can be created using the *gallery* command.

6. **Colon operator ' : '**

   The colon operator is one of the most important operators in MATLAB and is used in many cases. With its help, special line vectors can be generated, e.g. used for indexing in *for*-loops or when plotting. Starting from a number, a unit is added and stored in the vector until a predetermined end has been reached or exceeded. The general syntax is:

$$< \mathit{Start} >:< \mathit{End} > \text{ or } < \mathit{Start} >:< \mathit{Increment} >:< \mathit{End} > .$$

$>> j = 1 : 5$

$>> X = 1.2 : 0.2 : 2$

$>> X = 1 : -0.3 : 0$

The function *linspace* is related to the colon operator. As input, it requires the number of points to be created instead of the distance in addition to the start and end. *linspace (a, b, n)* creates n points of equal distance between a and b. The default value for n is 100.

$>> linspace(-1, 1, 9)$

$>> linspace(1, 100)$

7. **Indexing**

Fields are indexed in MATLAB using parentheses and start at index 1. For matrices, the first index stands for the row number, the second for the column number of the element.

$>> A = rand(2, 2)$

$>> A(1, 1)$

$>> A(2, 1)$

$>> A(0, 0)$

Analyse results of the following example:

$>> A = [1, 2, 3; 4, 5, 6; 7, 8, 9]$

$>> A([1, 2], 3)$

$>> A([2, 3], [2, 3])$

The colon operator is used particularly frequently in these cases. The submatrix consisting of the intersection of the rows p to q and the columns r to s is returned with $A(p : q, r : s)$. A special case is a single colon, which selects all rows or columns, i.e. $A(:, j)$ means the j-th column, and $A(i, :)$ the i-th row of A. The keyword *end* stands for the last index in the specified dimension, i.e. $A(end, :)$ is the last line of A.

$>> A(1 : 3, 2 : end)$

$>> A(1, :)$

$>> A(:, end)$

MATLAB stores all fields internally as a column vector, e.g., matrices column by column from the first to the last column. This representation can be accessed by specifying only one index, this is often called *linear indexing* in the literature.

$>> A(:)$

$>> A(4)$

8. **Operators**

To transpose a field, MATLAB provides the operator ' :

$>> A = [1, 2; 1, 2]$

$>> A'$

MATLAB supports computing with vectors and matrices. In this way, two fields of the same dimensions can simply be added by + and subtracted by − :

$>> x = 1 : 3;$

$>> y = 2 : 4;$

$>> x + y$

$>> eye(2) - ones(2)$

MATLAB interprets the multiplication operator ' ∗ ' as a matrix product or as a multiplication by a scalar. In the former, the number of columns in the first argument must equal the number of rows in the second argument. There is also the elementary multiplication operator ' . ∗ ' :

$>> x = 1 : 3;$

$>> y = 2 : 4;$

$>> x * y$

$>> x * y'$

$>> x. * y$

$>> A = [1, 2, 3; 4, 5, 6; 7, 8, 9];$

$>> A * y$

$>> 2 * A$

9. **Some basic functions**

length is the length of a vector, for a matrix the larger of the two dimensions is returned.
size returns the dimensions of a matrix.
numel returns the number of all elements in a matrix.

$>>B = [1, 2; 2, 3; 4, 5]$

$>> length(B)$

$>> size(B)$

$>> numel(B)$

Another examples are:

$>> x = 1 : 4$

$>> sum(x)$

$>> max(x)$

$>> diff(x)$

10. **Matrix manipulations**

    There are several commands for manipulating matrices. The *reshape* function changes the dimensions of a matrix: $reshape(A, m, n)$ creates an $m \times n$ matrix, the elements of which are extracted columnwise from A.

    $>> A = [1\ 4\ 9; 16\ 25\ 36], B = reshape(A, 3, 2)$

    The notation [ ] stands for an empty 0×0 matrix. If you assign the value [ ] to a row or column in a matrix, it is deleted from the matrix.

    $>> A = magic(3)$

    $>> A(2, :) = []$

    $>> x = []$

    $>> x = [x, 1 : 3]$

    Additional elements can also easily be added to an existing vector by assigning a value to an index that is longer than the length of the vector. The vector is then automatically extended to the corresponding index and filled with zeros.

    $>> x(6) = 9$

11. **Relational operators**

    The relational operators are:
    == equal
    ∼= unequal
    < less as
    <= less or equal as
    > bigger as
    >= bigger or equal as

    $>> A = [1\ 2; 3\ 4]; B = 2 * ones(2);$

$>> A == B$

$>> A > 2$

To test whether two matrices A and B are equal, the expression $isequal(A, B)$ can be used. The *isequal* function is one of the many useful logical functions whose name begins with *is*. For a list of all these functions, call MATLAB *doc is*.

12. **Logical operators**

The logical operators are:

&	logical *and*
|	logical *or*
∼	logical *not*
xor	logical exclusive *or*
all	is *true* if considered propertiy holds for all elements
any	is *true* if considered property holds for at least one element

$>> x = [-1\ 1\ 1]; y = [1\ 2\ \ -3];$

$>> x > 0\ \&\ y > 0$

$>> x > 0\ |\ y > 0$

$>> xor(x > 0, y > 0)$

$>> any(x > 0)$

$>> all(x > 0)$

13. **Flow control**

MATLAB has four structures for flow control: the *if*-query, the *for*-loop, the *while*-loop and the *switch*-command.

$>> x = rand(1);$

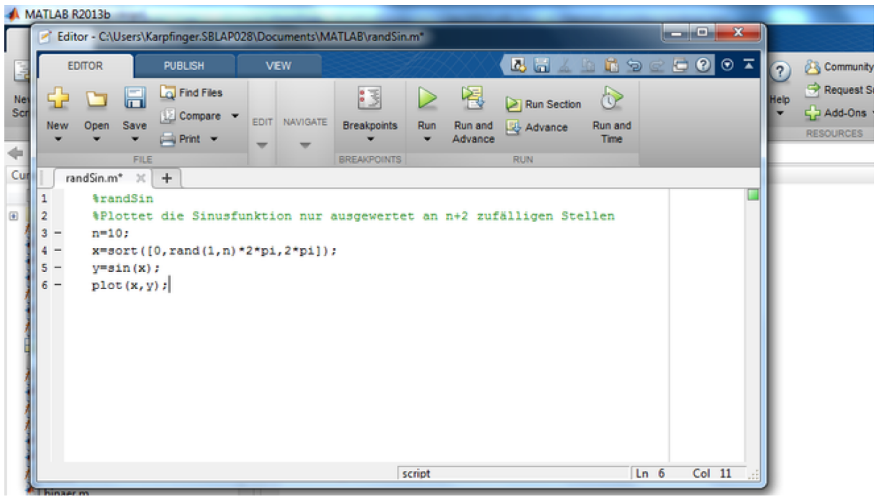
$>> if\ x > 0,\ disp(sqrt(x)); end$

$>> s = 0;$

$>> for\ i = 1 : 25,\ s = s + 1/i; end,\ s$

$>> for\ x = [pi/6\ pi/4\ pi/3],\ disp([x, sin(x)]), end$

```
>> x = 1;
   while x > 0
       xmin = x;
       x = x/2;
   end
   xmin
```



```
%randSin
%Plottet die Sinusfunktion nur ausgewertet an n+2 zufälligen Stellen
n=10;
x=sort([0,rand(1,n)*2*pi,2*pi]);
y=sin(x);
plot(x,y);
```

14. **M-files**

Many useful calculations can be carried out using the MATLAB command line. Nevertheless, sooner or later you will have to write M-files. These are the counterpart to programs, functions, subroutines and procedures in other programming languages. If you combine a sequence of commands into an M-file, there are various options, such as:

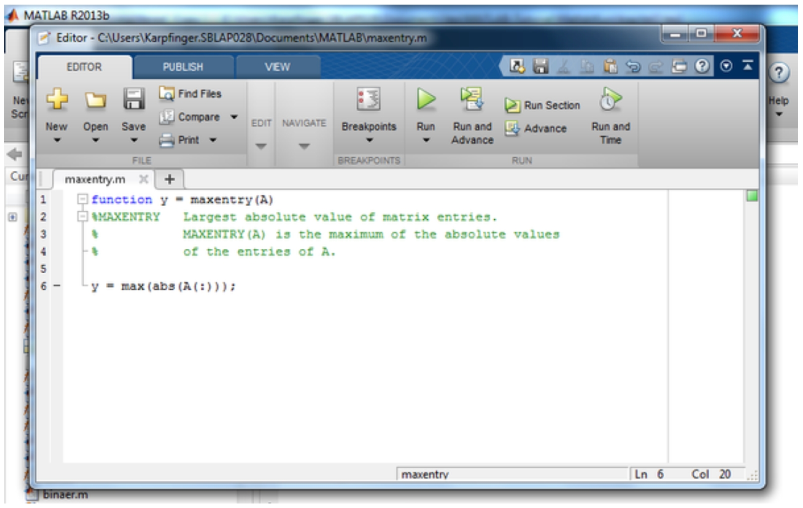
- experiment with an algorithm by editing a file instead of typing a long list of commands over and over again;
- create lasting evidence for a numerical experiment;
- build up useful functions that can be used again later;
- exchange M-files with other colleagues.

There is a large number of useful M-files written by different users. An M-file is a text file with the file extension ' .m ' that contains MATLAB commands. There are two kinds:

- *Scriptfiles* (or command files) have no input or output arguments and operate on variables defined in the script itself or in the workspace;
- *Function* files contain a *function* headline, they accept input arguments and return output arguments, their internal variables are operate locally (unless they have been declared globally).

A *script* collects a series of commands that are to be used repeatedly or will be used in the future.

An example of a script *randSin.m* is presented above. The first two lines of the script start with the % symbol and are therefore comments. As soon as MATLAB encounters %, it ignores the rest of the line. This allows the



```
function y = maxentry(A)
%MAXENTRY   Largest absolute value of matrix entries.
%           MAXENTRY(A) is the maximum of the absolute values
%           of the entries of A.

y = max(abs(A(:)));
```

insertion of text that makes the script easier for people to understand. Assuming this script was saved under the name randSin.m, entering randSin on the command line is equivalent to entering the all individual lines of the script.

Self-written function files expand the scope of the MATLAB possibilities regarding its own functions like *sin, eye, size* etc.

The function example *maxentry* presents several features. The first line begins with the keyword *function*, followed by the output argument y and the ' = '-sign. To the right of ' = ' comes the function name *maxentry*, followed by the input argument A in brackets. In general, there can be any number of input and output arguments. The function name must be the same as the name of the M-file, i.e. in this case the file must be called *maxentry*.

The *maxentry* function is to run like any other MATLAB function:

$$>> maxentry(1:10)$$

$$>> maxentry(magic(4))$$

15. **Graphics**

In MATLAB it is relatively easy to create different types of graphics. The appearance can be designed very individually. For example, you can adapt colors, axis scales and labels to your own ideas. There are basically two options for modifying graphics, either directly via the command line or alternatively interactively with the mouse in the existing graphic.

The simplest way to create two-dimensional plots is to couple two vectors with the same dimensions. If you have given two such vectors (x and y), the command $plot(x, y)$ recognizes the respective $x(i)$ and $y(i)$ as belonging together and generates a corresponding plot.

```
>> x = [1.5 2.2 3.1 4.6 5.7 6.3 9.4];
>> y = [2.3 3.9 4.3 7.2 4.5 3.8 1.1];
>> plot(x,y)
>> close
```

The appearance can be modified with additional information. A more general call to the function *plot* has the form *plot*($x, y,$ ‘*string*‘), whereby the *string* consists of up to three specifications (color, knot, line type).

```
>> plot(x,y,'m:')
>> close
```

A typical call has the form *plot*($x, y,$ ‘mx : ‘), the same result can be obtained with *plot*($x, y,$ ‘m : x‘), where you can see that the order of the information is irrelevant.

It is also possible to place more than one graph in the coordinate system:

```
>> a = 1:.1:10;
>> b = 1./a;
>> plot(x,y,'rp--',a,b,'mx:')
>> close

>> plot(x,y,'LineWidth',2)
>> xlabel('x')
>> ylabel('y')
>> title('One example')
>> grid on
>> close
```

The command $subplot(m, n, p)$ divides the figure window into m rows and n columns. p is a scalar and denotes the active field.

```
>> subplot(2,2,1), fplot('exp(sqrt(x)*sin(12*x))',[0 2*pi])
>> subplot(2,2,2), fplot('sin(round(x))',[0 10],'--')
>> subplot(2,2,3), fplot('cos(30*x)/x',[0.01 1 -15 20],'-.')
>> subplot(2,2,4), fplot('[sin(x),cos(2*x),1/(1+x)]',[0 5*pi -1.5 1.5])
>> close
```

Here we used a new function: *fplot*. MATLAB evaluates the function at appropriate positions to obtain a suitable graph. Of course, other parameters can also be passed to *fplot*. More about this can be found under *doc fplot* - just like any other function. An irregular division of the figure field is also possible:

```
>> x = linspace(0, 15, 100);
>> subplot(2, 2, 1), plot(x, sin(x))
>> subplot(2, 2, 2), plot(x, round(x))
>> subplot(2, 2, 3 : 4), plot(x, sin(round(x)))
>> close
```

As with the plot command in two dimensions, there is also the command *plot3* for plotting curves in 3-D. Basically, it works the same way as 2-D.

```
>> t = −5 : .005 : 5;
>> x = (1 + t.^2). ∗ sin(20 ∗ t);
>> y = (1 + t.^2). ∗ cos(20 ∗ t);
>> z = t;
>> plot3(x, y, z)
>> grid on
>> close
```