1 Representation of Signals I: Introduction to the Software Environment

1.1 Overview

Many exercises of these supplements are computer-based. With MATLAB, a widely used mathematical interpreter language for vector arithmetic and linear algebra, you are able to simulate and solve typical signal processing problems as they arise in the speech and audio context. Special *toolboxes* like the "Signal Processing Toolbox" as part of this software make it ideally suitable, both for learning and research.

This first exercise introduces the simulation language within a basic discrete-time signal processing context and goes on to the illustration of some important fundamentals on the representation of statistical signals. The analysis of real-world speech and audio signals will follow in the next exercise. Appendix A provides a brief introduction to the computer environment and MATLAB.

The supplement material can be found in the directory `./SHARED_FILES/spsa/`. For each supplement, copy the subdirectory `./SHARED_FILES/spsa/ExerciseX/` into your home directory before editing the files in Matlab.

1.2 Basic Signals

We consider the following signals[1]:

$$\begin{aligned}
v_1[k] &= \delta[k-4]; & k &= 0:10\\
v_2[k] &= \sin[2\pi k/32]; & k &= -32:32\\
v_3[k] &= \cos[\pi k/\sqrt{2}]; & k &= 0:127\\
v_4[k] &= \exp[\mathrm{j}\ 2\pi k/16]; & k &= -32:32
\end{aligned}$$

where

$$\delta[k] = \begin{cases} 1 & \text{for} \quad k=0 \\ 0 & \text{else} \end{cases} \tag{1}$$

Create the basic signals $v_1[k]$ to $v_3[k]$ using the description in the appendix A.2 ("Introduction to Matlab") and plot the signals. $\boxed{\text{M}}$

Example:

```
k1  = 0:10;
v1  = (k1 == 4);
stem(k1,v1);
```

[1]Parts of this section are taken from a former version of *Praktikum Digitale Signalverarbeitung, University of Erlangen-Nuremberg.*

For the complex sequence $v_4[k]$, determine the real part and the imaginary part using `real` and `imag` and plot them in different colors into the same figure (use `hold on` and `hold off`).

Example:

```
stem(k, real(x), 'r');   % plot real part of x in red
hold on
stem(k, imag(x), 'b');   % plot imaginary part of x in blue
hold off
```

Are the sequences $v_2[k]$ to $v_4[k]$ periodical? If so, how long is the respective period? T

v_2:	☐ yes	☐ no	▷ ______________________
v_3:	☐ yes	☐ no	▷ ______________________
v_4:	☐ yes	☐ no	▷ ______________________

1.3 Functions: Center Clipper

In the following, we consider a simple example for defining a new MATLAB function, see A.3. Our new function should transform a signal $x[k]$ to $y[k]$ using the memoryless system shown in *Fig.* 1, where η shall denote a parameter to the function.

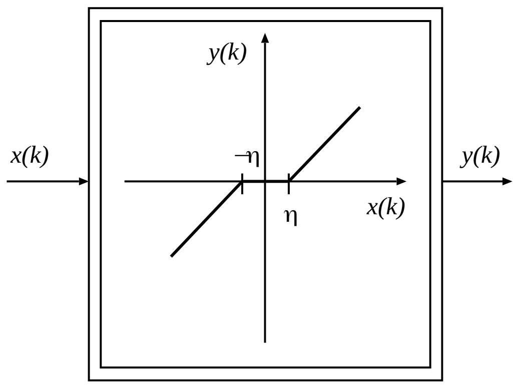


Fig. 1: Center clipper

Such a nonlinearity is a useful building block in several speech processing applications as it allows for a simple suppression of noise (*'center clipping'*).

Open the file `cclipper.m` and add the missing lines in the code. $\boxed{\text{M}}$

```
function y = cclipper(x,eta)
...
for k = ...
    if x(k) > ...
       y(k) = ...
    else if x(k) < ...
               y(k) = ...
           else
               y(k) = ...
           end
    end
end
```

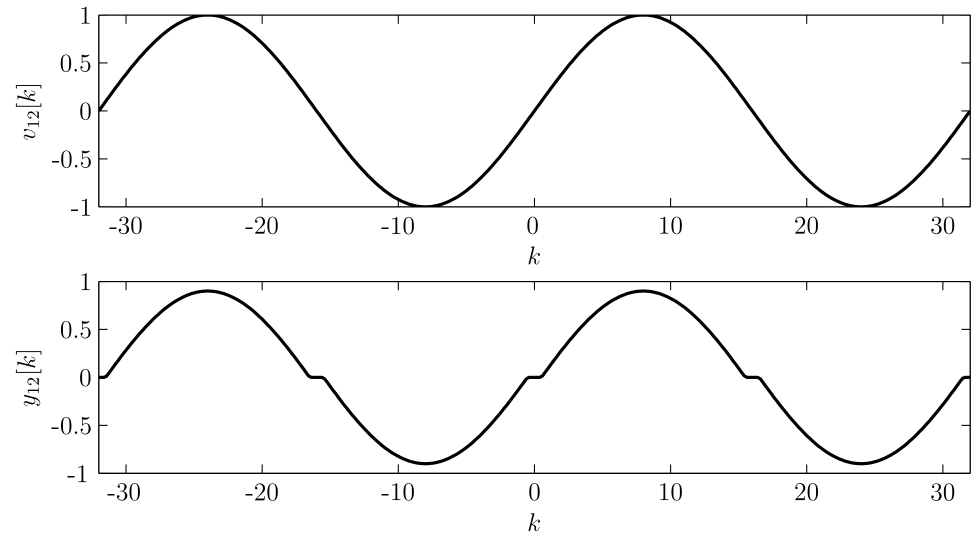
Then, by invoking, e.g.,

```
y12 = cclipper(v12,0.1);
```

where

$$v_{12}[k] = \sin[2\pi k/32], \qquad k = -32 : 0.2 : 32$$

all input values with magnitude smaller than 0.1 are suppressed. For comparison, the input and output curves should be shown in one figure using the commands `subplot`, followed by `plot` for each sequence.



1.4 Representation of Stochastic Signals

Speech and audio signals have to be modeled as stochastic processes (i.e., ensembles of random sequences), as the observer (usually) cannot describe speech and audio signals analytically. In this section, we recall important fundamentals on the representation of stochastic signals and show how to handle them with MATLAB. Moreover, this section is intended to gain some feeling for the problems associated with the estimation of stochastic quantities. In order to be able to compare the estimated quantities with their ideal values, we work with white noise signals in this exercise (speech and audio signals will be studied in the next exercises).

1.4.1 Random Signals, Moments, Densities

a, The following problems should be solved for preparation (not in Matlab): $\boxed{\text{T}}$

- Sketch the probability density function (pdf) of a uniformly distributed stationary process which does not have zero mean.

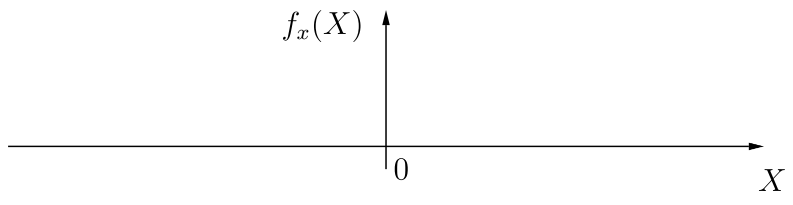


Fig. 2: pdf of a uniform distribution

- Sketch the pdf of a normally distributed stationary process which does not have zero mean. Mark the linear mean and the standard deviation (square root of the variance).

Fig. 3: pdf of a normal distribution

- What is the difference between the pdf and a histogram? Which one can be measured in reality?

  ▷ ______________________________

  ▷ ______________________________

  ▷ ______________________________

b, – Create, using the function `rand`, two random sequences x_1 and x_2 (row vectors) of lengths 1000 and 10000, respectively, and show sequence x_1 on the screen using `plot`. M

– Determine the linear means m_{x_1}, m_{x_2} and the variances $\sigma^2_{x_1}$, $\sigma^2_{x_2}$ using the MATLAB commands `mean` and `var`.

– Using the function `hist`, plot the histograms on 100 points (`help hist`).

– Which pdf is approximated by the random number generator `rand` ?

▷ ______________________________

– Towards which values do the linear mean and variance converge for infinitely long sequences?

▷ ______________________________

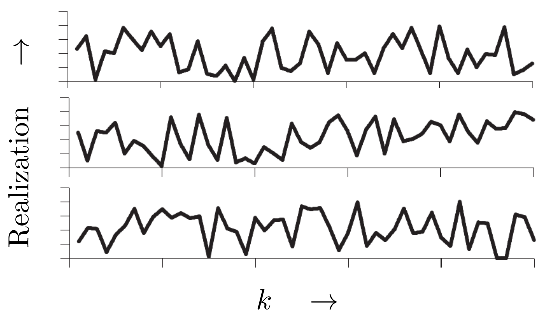
c, Now, use the function `randn` and create two random sequences x_3 and x_4 (row vectors), again of lengths 1000 and 10000, respectively. Compare the signal waveforms of x_1 and x_3. Determine the linear mean, the variance and the estimated pdf for x_3 and x_4. What distribution is approximated by the random number generator `randn`? M

▷ ______________________________

1.4.2 Auto- and Cross-Correlation, Power Spectral Density

Recall some basics of statistical signal theory and answer the following questions:

a, What do the terms stationarity and ergodicity mean? Use the following illustration to explain how these properties are related (k is the time index). T



▷ ______________________________
▷ ______________________________
▷ ______________________________
▷ ______________________________

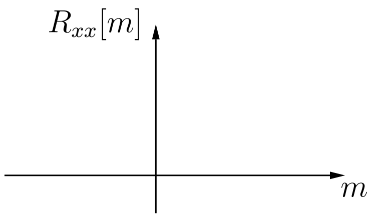
b, Consider a discrete-time white noise signal. Sketch the shape of the autocorrelation function (ACF)

$$R_{xx}[m] = \mathcal{E}\left\{x[k]x[k+m]\right\} \tag{2}$$

and the power spectral density (PSD)

$$S_{xx}(\mathrm{e}^{\mathrm{j}\Omega}) = \sum_{m=-\infty}^{\infty} R_{xx}[m]\mathrm{e}^{-\mathrm{j}\Omega m} = \lim_{M\to\infty} \frac{1}{2M+1}\left|X_N(\mathrm{e}^{\mathrm{j}\Omega})\right|^2 \tag{3}$$

in this particular case (X_N denotes the Fourier transform of a segment with length $N = 2M+1$).



Mark the linear mean and the variance on the ACF sketch.

1.4.3 Filtering of Noise Signals

a, For filtering of signals, MATLAB provides the function `filter` for linear time-invariant systems with fixed coefficients which we invoke with the two sets of coefficients $\boxed{\text{T}}$

Set 1 : $\mathbf{b} = [\,0.5\;\;0.5\,]$ $\quad\mathbf{c} = [\,1\;\;0\,]$

Set 2 : $\mathbf{b} = [\,0.1\;\;0.1\;\;0.1\;\;0.1\;\;0.1\;\;0.1\;\;0.1\;\;0.1\;\;0.1\;\;0.1\,]$
$\mathbf{c} = [\,1\;\;0\;\;0\;\;0\;\;0\;\;0\;\;0\;\;0\;\;0\;\;0\,]$

The elements of $\mathbf{b} = [\;b_0\;\;b_1\;\;...\;\;b_m\;]$ denote the coefficients of the numerator of the corresponding transfer function, and $\mathbf{c} = [\;c_0\;\;c_1\;\;...\;\;c_m\;]$ contain the coefficients of the denominator in the z domain:

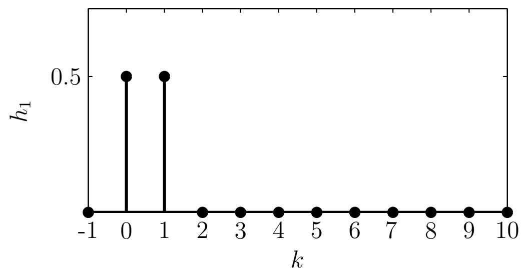
$$H(z) = \frac{B(z)}{C(z)} = \frac{\sum\limits_{i=0}^{m} b_{m-i}z^i}{\sum\limits_{i=0}^{m} c_{m-i}z^i} = \frac{b_0 + b_1z^{-1} + b_2z^{-2} + \cdots + b_mz^{-m}}{c_0 + c_1z^{-1} + c_2z^{-2} + \cdots + c_mz^{-m}} \tag{4}$$

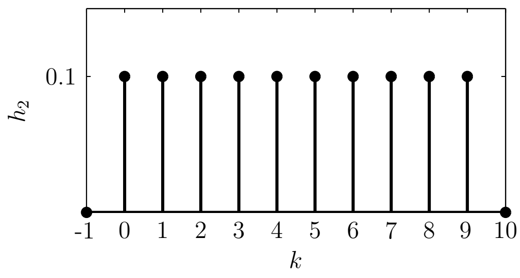
What particular kind of filter model (AR, MA, ARMA) is described by the above defined Set 1 and Set 2 of coefficients?

▷ ____

Do the filters have a *finite impulse response (FIR)* or an *infinite impulse response (IIR)*?

▷ ______________________________





b, The resulting 1st and 9th order lowpass filters perform arithmetic averaging of two and ten subsequent signal samples. $\boxed{\text{M}}$

– First, apply each filter (use `filter`$(\mathbf{b}, \mathbf{c}, x)$ in MATLAB) to the signals $x = x_2$ (first, subtract the mean) and $x = x_4$. The resulting output signals should be denoted y_{21}, y_{22}, and y_{41}, y_{42}, respectively.

– Estimate the probability density functions of the output sequences (using `hist`). Plot the pdf estimates derived from x_2 and x_4 in one figure (using `subplot`). How do the pdfs change with increasing averaging of the signal?

▷ ______________________________

– Calculate

  ∗ the autocorrelation sequences (using `xcorr`) and
  ∗ the corresponding PSDs (using `pwelch`)

for the sequences x_2, x_4, y_{21}, y_{22}, y_{41}, y_{42}, and plot them (`subplot` for comparisons on the screen). How does the filtering affect the autocorrelation sequences and the power spectral densities?

▷ ______________________________
▷ ______________________________

A Matlab-Based Exercises

A.1 Environment for Matlab-Based Exercises

A.1.1 Login

The exercises are based on software running under Linux/Unix operating systems. Please note that the computers in the Telecommunications Laboratory must not be rebooted or switched off. To login, just switch on the monitor and type `spsaX` (for "Signal Processing for Speech and Audio". `X` stands for the number of your group) at the prompt. Each group will obtain a separate password.

A.1.2 Accessing Matlab

After logging in, one can start Matlab from the Unix shell with the command `matlab`. Then, the graphical matlab desktop will appear in a separate window. (It is also possible to run Matlab in text mode by `matlab -nodesktop` which is often much faster, but a little less comfortable. In this case, you will find it convenient to keep both Matlab and your local text editor (e.g. `emacs` or `kedit`) active in separate windows.)

A.2 A Short Introduction to Matlab

Matlab works with essentially only one kind of object: real or complex valued matrices. Scalar values are represented as *matrices* with dimension 1×1, row vectors and column vectors are also treated as matrices where one dimension is set to one. It is important to note that the indexes of matrix components always start with **one**, not with zero.

Operations, such as addition of vectors or multiplication of vectors or matrices (with compatible dimensions) are denoted compactly as `a + b` and `c * d`, respectively. The Operations are carried out in the usual order as known from mathematics. Moreover, the use of parentheses is allowed.

In the following, *signals* are represented by row vectors, with the signal samples as vector elements. *Signal names* consist of one letter, followed by up to 19 alphanumerical symbols. Matlab is case-sensitive in the name of commands, functions and variables. For example `input` is not the same as `Input` or `INPUT`.

We now create our first signal with Matlab by typing

```
y = [ 1 2 3.3 2 0 0.5 0 0 -1.7 ];
```

on the command line. Without the semicolon ';' all elements would be printed out on the screen (therefore, the semicolon is usually very important when working with long signals).

With

`plot(y)` or `stem(y)`,

we obtain graphical representations on the screen. `Plot` carries out a linear interpolation, while `stem` draws discrete lines. For a detailed description of the commands, we refer to several tutorials and the MATLAB handbooks. Furthermore, there is an online help for every command or operator by typing `help Name`.

A.2.1 Basic Operators for Creating Signals

The creation of vectors through direct input of the samples has been already presented above. Using the operators described in the following, many interesting signals can be created very easily:

• With the **increment operator** '`:`' in `k = 0:3;` a row vector with elements from 0 to 3 with step size 1 is created: `k = [ 0 1 2 3 ];`
`k = -2:8;` results in `k = [ -2 -1 0 1 2 3 4 5 6 7 8 ]`. In
`k =` x_0`:`x_1`:`x_2`;` the increment x_1 is used, which can be also negative: `k = 3:-0.3:1;` creates a vector where the values of the elements are decreasing: `k = [ 3 2.7 2.4 2.1 1.8 1.5 1.2 ];`

• For creating signals by **functions** (e.g. ones, sin, exp), first the argument, i.e., index sequence `k` has to be created (e.g. using the ':' operator). Applying a function to such an argument sequence results in a vector (or matrix in general) of the same dimension, where the function has been applied element-wise.

Example for `k = 0:3;` :

`v = sin(k)` yields the sine of the input vector.
`v = [ sin(0) sin(1) sin(2) sin(3) ]` and therefore:
`v = [ 0 0.8415 0.9093 0.1411 ]` .

`v = ones(size(k))` yields an output signal having the same length as the input signal, where all values are equal to one: `v = [ 1 1 1 1 ];`

`v = zeros(size(k))` is correspondingly `v = [ 0 0 0 0 ];`

Some important functions which can be applied to complex valued vectors as well, are e.g. (`help` shows many more functions!),

`sin`	sine	`cos`	cosine	`tan`	tangent
`exp`	exponential function	`log`	nat. logarithm	`sqrt`	square root
`atan`	inverse tangent	`ones`	matrix of ones	`zeros`	matrix of zeros
`real`	real part	`imag`	imaginary part	`abs`	absolute value

Remark: The functions `ones` and `zeros` create an $n \times n$ matrix, if they are invoked with one argument n. If `ones` shall be applied to a vector `v` (which may have length one), the commands `ones(1,length(v))` or `m = size(v); ones(m(1),m(2))` can be used.

• Element-wise access to *single signal samples* or *signal intervals* is possible. In this context it is important that the numbering of the indexes starts with *one* and not with zero.

`v = zeros(1,4); v(2) = 1;` creates a row vector containing 4 zeros and sets the second element equal 1: `v = [ 0 1 0 0 ];` . With this method one can create, e.g., a shifted unit impulse $\delta(k-\kappa)$.

`v = 3:7; v(3:4) = [ 0 1 ];` first creates `v = [ 3 4 5 6 7 ];` and replaces the elements v(3) to v(4) by the corresponding elements on the right side with the result `v = [ 3 4 0 1 7 ];`

`v = zeros(1,10); v(2:3:10) = ones(size(2:3:10));` first creates a row vector with 10 zeros and replaces then the elements 2, 5 and 8 by ones.

• Often, the concatenation of vectors or matrices **out of sub-vectors or sub-matrices** is needed. E.g. in the case of a row vector `v` put together by two other row vectors `v1` and `v2`, the two components are set in brackets: `v = [ v1 v2 ];`.

Example:

`x = 1:3; v = [ ones(1,3) zeros(1,3) 7 x ];` yields:
`v = [ 1 1 1 0 0 0 7 1 2 3 ];`

• **Elimination of vector elements** is done by the "empty" vector. With `v(2) = [];` the 2. element in `v` can be removed, with `v(1:3) = [];` the elements 1 to 3. This allows also shortening of vectors.

• **Complex valued signals** can essentially be processed in exactly the same way as the real valued signals. The imaginary unit j is predefined as `i` and `j`. However, if there is a variable of the same name in the MATLAB workspace, it has the priority and `i` or `j` denotes a "normal" name. Then, either the variable has to be deleted (e.g. `clear j`), or j need to be calculated by `sqrt(-1)`.

Example: `k = 0:100; v = exp(j*.3*pi*k)` yields $v = e^{j0.3\pi k}$ for $k = 0(1)100$. π is predefined as a constant `pi`.

• Besides the row vectors, we can create and process column vectors and matrices as well. In the input, each semicolon '`;`' begins a new row in the matrix.

`x = [ 1; 2; 3 ]` yields a column vector and
`y = [ 1 2 3; 4 5 6; 7 8 9]` a 3×3 matrix:

$$\mathtt{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}; \quad \text{and} \quad \mathtt{y} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix};$$

With `.'` matrices are **transposed** (i.e., row vectors are converted into column vectors). With `'` (i.e., without dot), the Hermite operation is applied, i.e., the

conversion to the conjugate transpose.
Since `x` and `y` are real in the above example, `.'` and `'` yield the same result. With `x1 = x.'; y1 = y'` one obtains:

$$\texttt{x1 = [1 2 3];} \quad \text{and} \quad \texttt{y1} = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix};$$

• With the command `whos` one gets a list of all current variables in the Matlab workspace (memory) including information about type and dimension of the variables and the remaining free memory. Variables that are not needed anymore can be deleted with `clear <var1> <var2> ...`.

A.2.2 Operators

As known from other more traditional programming languages, using the operators `+`, `-`, `*` (multiplication), `/` (division) and `^` (matrix power) arithmetic expressions can be built. The operators have the usual priorities, hierarchies by parentheses are also allowed.

• Note that for vector and matrix operation one has to take care of compatible dimensions of the operands: For addition and subtraction, both operands either must have the same dimension or one of the operands must be a scalar value. By multiplication of vectors or matrices a matrix multiplication is always meant.

Example:

`[ 1 2 3 ] * [ 1; 2; 3 ]` calculates the product of a row vector and a column vector, i.e., the inner product (result `= 14`).

• In the context of vectors and matrices, the **division** has a special meaning: with '/' and '\' there are **two different** operators for the division. Thereby, `x = a \ b` yields a solution of `a * x = b` and `x = b / a` yields a solution of `x * a = b`. Formally we have: `a\b` $\hat{=}$ `inv(a) * b` and `b/a` $\hat{=}$ `b * inv(a)`. However, using the division operator, the linear system is solved directly, i.e., without explicit matrix inversion. This yields more accurate results and is less computationally complex.

Example: With

$$\texttt{A} = \begin{bmatrix} 1 & 2 \\ 3 & 5 \end{bmatrix}; \quad \text{and} \quad \texttt{b} = \begin{bmatrix} 1 \\ 4 \end{bmatrix};$$

we obtain

$$\texttt{A} \setminus \texttt{b} = \begin{bmatrix} 3 \\ -1 \end{bmatrix}; \quad \text{and} \quad \texttt{(b') / A = [7 -2];}$$

• The **component-wise** multiplication, division or power of two vectors or matrices is done by the operators `.*`, `./` and `.^` .

Examples:

`x = [ 1 2 3 ] .* [ 1 2 3 ];` yields `x = [ 1 4 9 ];`

`x = [ 1 2 3 ] ./ [ 1 2 3 ];` yields `x = [ 1 1 1 ];`

`x = [ 1 2 3 ] .^ [ 1 2 3 ];` yields `x = [ 1 4 27 ];`

• **Comparisons** are carried out by the operators `>`, `<`, `>=`, `<=`, `==` (equal) and `~=` (not equal). They can be only applied to variables with compatible dimension. The resulting vector or matrix contains ones, where the condition is satisfied, otherwise there are zero elements.

The relational operators allow elegant realizations of impulse, step and rect sequences.

Example: Impulse sequence

`k = 0:5; v = k == 1;` yields `v = [ 0 1 0 0 0 0 ];`

The result of the *element-wise* comparison `k == 2` is handed over to the variable `v`. Parentheses are not necessary due to the higher priority of the relational operator.

Example: Rectangular sequence

`k = 0:5; v = (k >= 1) - (k >= 4);` yields `v = [ 0 1 1 1 0 0 ];`

A.3 MATLAB Functions

In order to avoid subsequent input of similar or equal sequences of MATLAB commands, there are two possibilities to save programs: script files and function files. MATLAB source codes from script files are executed just as commands directly typed in by the keyboard. In contrast, function files define new MATLAB commands.

Script files Using the text editor, a file with extension `'.m'` , e.g. `prog.m` , is created, which contains the MATLAB source code:

```
k = 0:100;
v = sin(2*pi*k/10);
stem(v)
```

The content of this file can now be executed by typing the file name without extension `'.m'`, i.e.,

```
prog
```

Note that all variables created during the execution remain in the workspace after finishing the program. Also, within the script file access to the variables in the workspace is possible.

Function files Function files represent an extension of the script files. Using an additional line with the keyword `function` and a description of the parameter interface at the beginning of the file one obtains a MATLAB function.

Example:

The MATLAB function `reverse` consists of the following commands in one file `reverse.m`:

```
function y = reverse(v)
% y = reverse(v)  time-reversal of signal vector:
%       returns output vector y in reversed order
%       of vector v, with exception of v(1).
y = v([ 1 (length(v):-1:2) ]);    % that's all!
```

The input of `reverse([1 1 1 0 0])` yields then: `[1 0 0 1 1]`.

Remarks:

- The first line contains the keyword `function` and the input and output parameters.
- After a %-sign, the following text until the end of the line is treated a comment and ignored during the execution of the function.
- Comments in the first lines of the function file are used as description for the `help` function. `help reverse` will show the following description:

```
y = reverse(v)  time-reversal of signal vector:
     returns output vector y in reversed order
     of vector v, with exception of v(1).
```

- New variables defined within the function file are *local variables*, i.e., they are only known inside the function and are deleted when returning to the calling program. Access to variables of the main program is not possible, unless they are given to the function as parameters[9].
- Several input and output parameter are allowed. The first line of a function file `fun2.m` with the input parameter `a`, `b` and `c`, as well as the output paramets `x` and `y` reads:

```
function [x,y] = fun2(a,b,c)
```

[9]Exception: Global variables, see the MATLAB handbook.