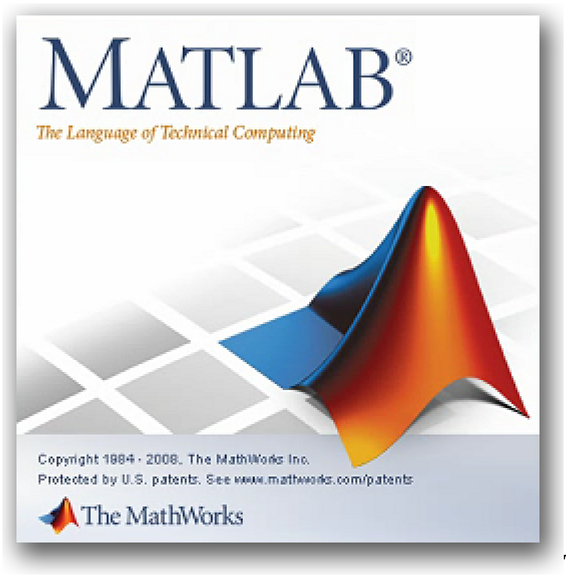
Introduction to Matlab

Table of Contents

Instructions

What is covered by this tutorial?


MATLAB®
The Language of Technical Computing
Copyright 1984 - 2008, The MathWorks Inc.
Protected by U.S. patents. See www.mathworks.com/patents
The MathWorks

This introductory course is intended to provide a practical introduction to working with the program **Matlab®** by **The MathWorks**, particularly focussing on aspects being relevant for data processing in **psychological research**. Matlab is often used in experiments applying psychophysical methods such as electro-encephalography (EEG), eye tracking, or registration of galvanic skin response (GSR). Put more generally, Matlab is very useful for the evaluation of large data sets that are often acquired automatically, as it is the case in logfile analysis or internet experiments as well.

Why Matlab?

Matlab is a programming language often used in psychological research. It is especially suitable for data analysis but can be applied for programming computer-controlled experiments as well. In contrast to other programming languages, a particular advantage of Matlab is that is works as an «interpreter»: commands that are typed in are processed immediately without having to compile the program beforehand; skipping this step leads to a faster development cycle. In that way, you can try out the syntax of a command in an uncomplicated manner and see whether they lead to the intended effect. Then the tested commands can be pasted into the final program file.

The scope of this course

Matlab is a very comprehensive software package, and most users only use a small part of it. Thanks to a very good integrated documentation and help system, it is relatively easy to learn additional commands once one has understood the basics.

In this sense, this course is confined to the **basic functionality of Matlab**, i.e. matrix-based data processing and visualisation.

The functionality of SimuLink (simulation program) will not be covered. Likewise, the functions of specialised tool boxes (neural network toolbox, wavelet toolbox etc.) cannot be covered; these are only rarely put to use in the course of psychological research. Rather, a solid basis should be established, starting from which it should be possible to autonomously acquire further knowledge and skills if needed.

Overview over the contents

- What can Matlab be used for?
- How to work with the user interface
- The basics: numbers, data types, operators, functions, etc.
- Programming of scripts and functions
- Program control structures: looping and branching
- Reading data from file, processing them, and writing them back to file
- Interaction with the user
- Statistical functions
- Graphics functions

All lesson contents can be practiced and applied in short practice assignments. In addition, self-test questions are available in order to assist the learning process by evaluating what you have learned.
Moreover, in the course of this tutorial, you will be able to develop a somewhat larger program project that is typical for applications in psychological research: importing and aggregation of data from several data sets, doing calculations with these data, saving the results in a format suitable for further processing (e.g. with SPSS or Statistica), and visualisation of the results.

Authoring

This learning resource was realised as a sub-project of the **edulap** project.

Roman von Wartburg, Ph.D.
Contents, didactical concept, implementation, translation, start page animation

Sarah Steinbacher, dipl. designer FH
Graphic design

Radka Wittmer, M.Ed.
Didactics counsel

Stephanie Schütze, Dipl. psych.
Usability evaluation

Joël Fisler
Implementation of graphic design

Support
This project was supported by several institutions:

Implementation and distribution
This course was implemented with **eLML**. The complete eLML source code is contained in the content packages that can be downloaded below:

ZIP archive of the HTML version
If you do not have permanent web access, you can unpack this archive to your hard disk and open locally with your web browser.

Content package in IMS/CP format
This version is intended for uploading to a learning management system such as OLAT, Moodle, or ILIAS.

Requirements

Previous knowledge

To profit from this Matlab course, previous knowledge as listed below will be helpful:

- **Basic experience in working with computers** is indispensable. If you are used to working with software packages such as SPSS, Microsoft Excel, or E-Prime, you should not experience serious problems to acquire the basics of Matlab as they are taught in this course.
- **Bachelor level methodological and statistical basic knowledge** in the field of psychology (or other social sciences) will help understand the practical examples. Mathematical knowledge exceeding this is not necessary.
- **Experience in computer programming** is very helpful, but not imperative.

Objectives

We tried to design this course in a way that experienced programmers as well as beginners can profit. However, to be realistic, the objectives and the level of demand will differ between these two groups.

- If you are experienced with other modern programming languages and thus know the way of thinking and the basic concepts of programming, you should be able to go through this course without significant difficulties. After that, you should be able to write Matlab routines from scratch. The basic concepts and structures of Matlab do not substantially differ from those of other modern (procedural) languages.
- For people with no or only rudimentary knowledge in programming, the course will be relatively demanding. We suppose that, after having studied this course, this group of learners will be able to understand and modify existing Matlab programs, and write simple routines based on the examples in the course.

Hardware and software requirements

As this course encompasses practical exercises as a constitutive part, it is indispensable to have the Matlab software package installed on the same computer. Matlab is available for Macintosh, Windows, various Linux distributions and Solaris (UNIX operating system, previously known as SunOS).

> Matlab is a commercial software and a license has to be purchased. For students, a license is offered by **Mathworks** for $99. Certain academic institutions (e.g. University of Bern) make Matlab available for free to staff and students. Please contact the IT support of your institution.

Most current computers that are not older than 5 to 8 years will have enough power to run Matlab. If you are in doubt, please consult the **system requirements** at the Mathworks website.
You will find installation instructions in the paragraph **Installation**.

Web browser requirements

In this course, certain contents will be displayed in **pop-up windows**. Therefore, make sure that your internet browser does not block pop-up windows for the website or learning management system on which this course runs. For a quick **test** click **here**. If you don't see a new window popping up after that, you will have to correct the preference settings of your internet browser accordingly.

Instructions

This tutorial is designed as a «hands-on» course: Everything that is being taught can and should be tried out immediately in the Matlab program running in parallel. We think that this is the way how knowledge of this kind is acquired the most efficiently.

All text areas suggesting concrete activity in Matlab are set in a green box and labelled with this icon.

All commands that have to be entered in Matlab's «command window» are displayed as shown below (the prompt >> must **not** be typed):

```
>> a = sin(2)
```

Explanations are printed in *italics* and are not to be entered:

>> 23.5 * pi   pi is an internally predefined value

Important information, key points and summaries are displayed in such a box and are marked with the corresponding icon.

Further icons

Specific text areas are labelled with other icons to point out their relevance:

Notes: This icon indicates additional, complementary information.

Help: Points to additional information in Matlab's help system.

Pop-up: If links labelled with this icon are clicked, a box or a new window open to display the information in.

Special notations

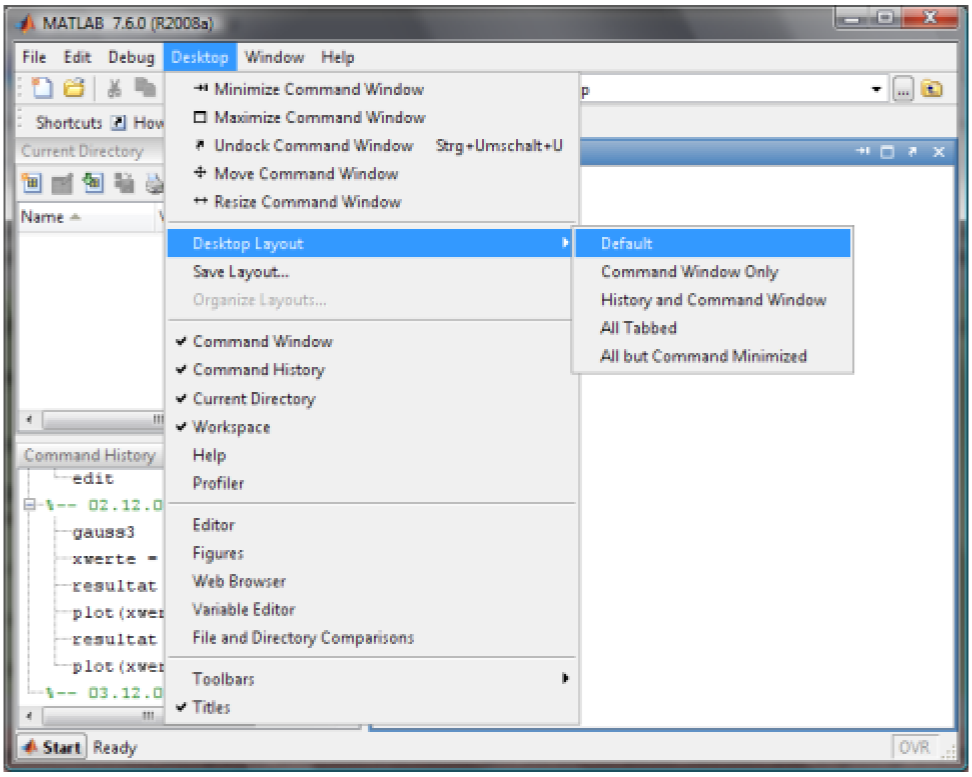
Matlab expressions such as commands, functions, and other parts of the program code are highlighted with the following text style:

`matlab_code`

Calls to Matlab's menu functions are formatted like this:

Desktop > Desktop Layout > Default

This, for example, indicates the operation:



Further features

In this course, two more functions are available to assist in your learning process. You can access them via the additional menu items below the main navigation of this tutorial (for the HTML version) or via the footer (content package version running in a learning management system such as OLAT).

- **PDF print version:** You can downnload the complete course as a PDF document. You can read this document even when you do not have internet access, or print parts of it.
- **Command list:** This list contains all commands used in the course. The list will open in a new, narrow window which you can conveniently place beside your other open windows for better visibility.

Preparations

This section describes the preparatory steps necessary for doing this course:

- Installation of Matlab on your computer
- Downloading and saving of the necessary materials such as example programs, data files, and sample solutions
- Configuration of Matlab

Installation of Matlab

Which version of Matlab do I need?

Version **7.2 / R2006a** or newer is recommended. It is possible to use older versions; however, certain details of the user interface may be different. Basically, working with Matlab is substantially more pleasant with the newer versions, especially due to useful extensions of the editor's functionality and improved help functions.

For Matlab, a license has to be obtained. A student license is available from **Mathworks** for $99. Certain academic institutions (e.g. University of Bern) make Matlab available for free to staff and students. Please contact the IT support of your institution.

If Matlab has not been installed on your computer yet, do it now. Follow the instructions provided by the installation program.

Important: Some versions of Matlab ask you to decide what exactly you want to install: the main application, the documentation, or both. Select **both**, as the documentation is an indispensable resource for working with this complex program package.

Downloading the necessary materials

To do this course, you need several data files such as code examples, sample solutions, or data files to test the programs you have written. Thus, you are asked to download these files to your computer now.

Please download the data files now, as described below:

Examples and sample solutions
You will find these program files in the ZIP archive file **examples.zip**. Please extract the contents to a new folder (= directory), e.g. `My Documents\matlab-course\m\` or `d:\m\`. It is useful to memorise or write down the name of this folder, because we will have to configure it in Matlab right away (cf. **here**).
Important: This folder will be your working folder for the programs you write.

Data
Some of the examples and exercise assignments need data files. You can download them as ZIP archive **data.zip**. Extract these files to another location, e.g. `My Documents\matlab-course\data\`.

Configuration of Matlab

Several settings are necessary or helpful to make Matlab «operational»:

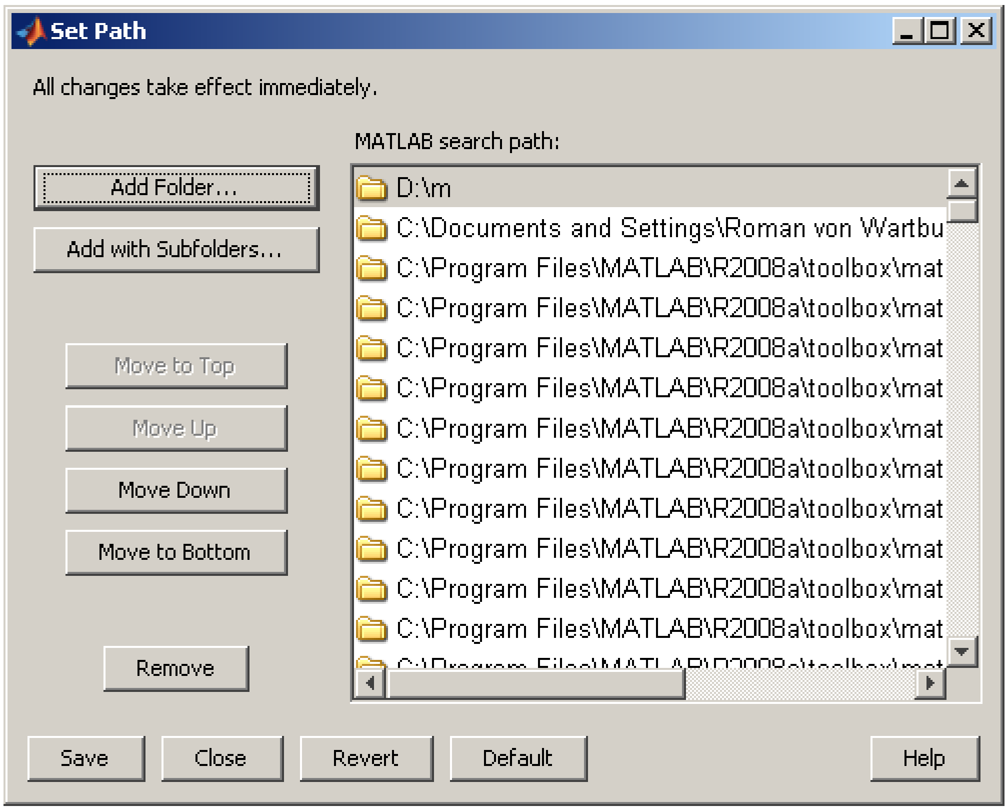
- setting the search path for your new programs
- configuration of the proxy server for internet access (if necessary)
- The tabulator size of the editor can be customised to suit one's taste.

Please follow these steps according to the instructions below. Afterwards you are ready to go!

Please check/configure the following settings:

Search path setting
Matlab has to be told where to look for programs. All paths for internal functions and toolboxes provided with the Matlab package have already been configured, but you have to let Matlab know the storage location of the programs you write yourself (the so-called «M-files»). In our case, this is the location where you have extracted the code examples and sample solutions from the downloaded ZIP archive (see section «Materials», **here**). Please proceed as indicated below:

- Select menu item **File** > **Set Path**
- Click the button **Add Folder...**
- In the file browser, look for the target folder and select it. Confirm by hitting **OK**
- Click **Save** and **Close**



All changes take effect immediately.

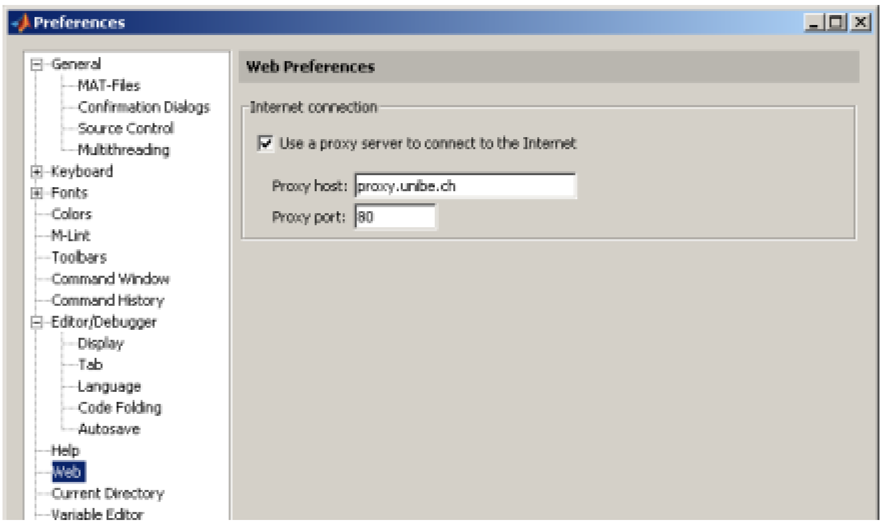
MATLAB search path:

```
D:\m
C:\Documents and Settings\Roman von Wartbu
C:\Program Files\MATLAB\R2008a\toolbox\mat
C:\Program Files\MATLAB\R2008a\toolbox\mat
C:\Program Files\MATLAB\R2008a\toolbox\mat
C:\Program Files\MATLAB\R2008a\toolbox\mat
C:\Program Files\MATLAB\R2008a\toolbox\mat
C:\Program Files\MATLAB\R2008a\toolbox\mat
C:\Program Files\MATLAB\R2008a\toolbox\mat
C:\Program Files\MATLAB\R2008a\toolbox\mat
C:\Program Files\MATLAB\R2008a\toolbox\mat
C:\Program Files\MATLAB\R2008a\toolbox\mat
```

Dialog box to configure the search path for M-files

Proxy server for internet access

For some functions of the help system, internet access is required. In case your computer is connected to a local area network using a so-called «proxy server», Matlab has to be informed of its address. If you don't know whether a proxy server is in use at all, or you don't know its address, contact your IT support. **Hint:** It is the same address as the one configured in your internet browser as proxy server.
In Matlab, you find this configuration option under the menu item **File** > **Preferences...** (see illustration below). If there is no proxy server, uncheck «Use a proxy server to connect to the Internet». Otherwise, configure its address as shown below. Instead of «proxy.unibe.ch / 80», the address and port number of your institution's proxy server have to be entered.



☑ Use a proxy server to connect to the Internet

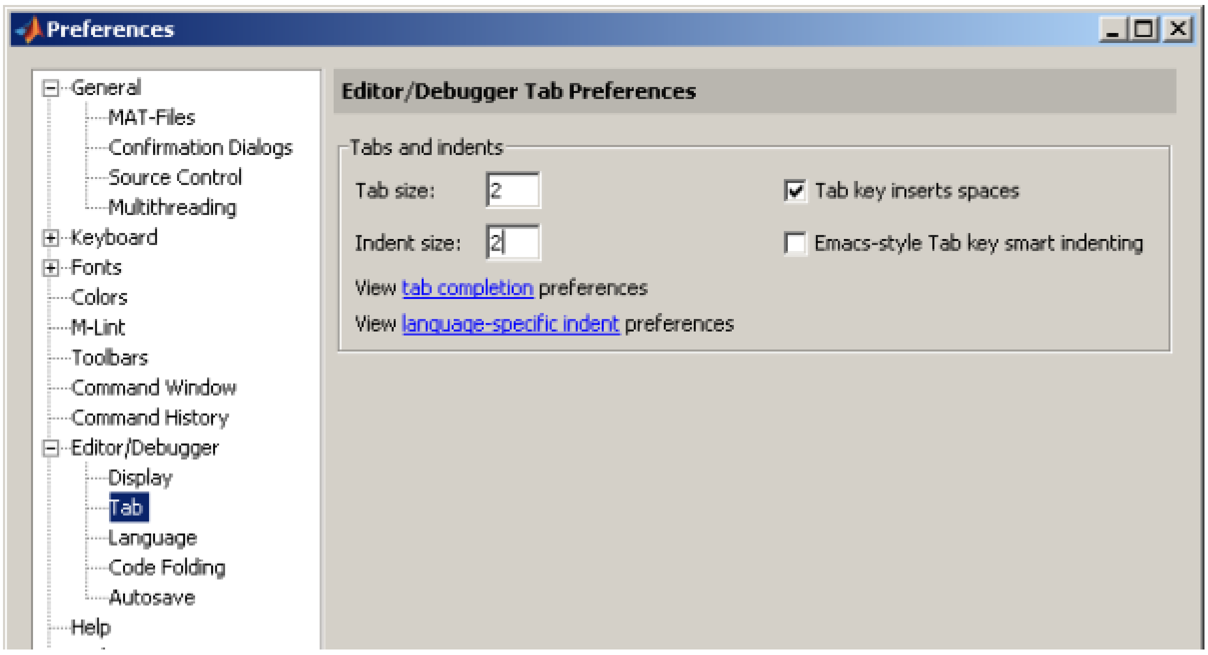
Proxy host: proxy.unibe.ch

Proxy port: 80

Proxy server settings for internet access

Tab setting for the editor

This option influences the display of program code in the editor, and leads to a more or less compact and clearly arranged representation. A value of 2 has proven optimal for most users, but it's a matter of taste (cf. Lesson 2, section **M-Files -> Editor/Debugger**) .
You can configure the tab setting like this: Select the menu item **File** > **Preferences...**, and then set the tab size according to the illustration below; it is advisable to set the same value for «Tab size» and «Indent size».



Editor/Debugger Tab Preferences

Tabs and indents

Tab size: 2

Indent size: 2

[✓] Tab key inserts spaces

[ ] Emacs-style Tab key smart indenting

View tab completion preferences

View language-specific indent preferences

Tabulator settings for the editor

Lesson 1: Matlab Basics

In this lesson you'll get to know the features and possibilities provided by Matlab, and you learn how to interact with Matlab by means of the user interface. Moreover, basic facts of data representation and manipulation are introduced.

Learning Objectives

- You get an overview of what you can do with Matlab
- You know the different parts of the user interface
- You can adapt the user interface to your needs
- You know how to perform calculations in command line mode
- You get to know Matlab's basic methods to represents data (scalars, vectors, matrices)
- You can manipulate these data types and apply mathematical functions and operations to them.

What Can You Use Matlab For?

Below, an overview of the possibilities of Matlab is given. In case you already know the potential of Matlab, you can skip these sections and continue with **The user interface**.
Matlab is short for **Matrix laboratory**. Matlab is a programming language and environment that makes use of specialised data types (*matrices* [1], in particular).

In what cases is it advantageous to use Matlab?
Small amounts of data that are acquired and evaluated only once can easily and efficiently be processed with spreadsheed software such as MS Excel or OpenOffice Calc. Afterwards they can be statistically analysed in statistics packages such as Statistica or SPSS. It is seldom worthwile to use Matlab for such tasks.
The use of Matlab is more appropriate in the following cases:

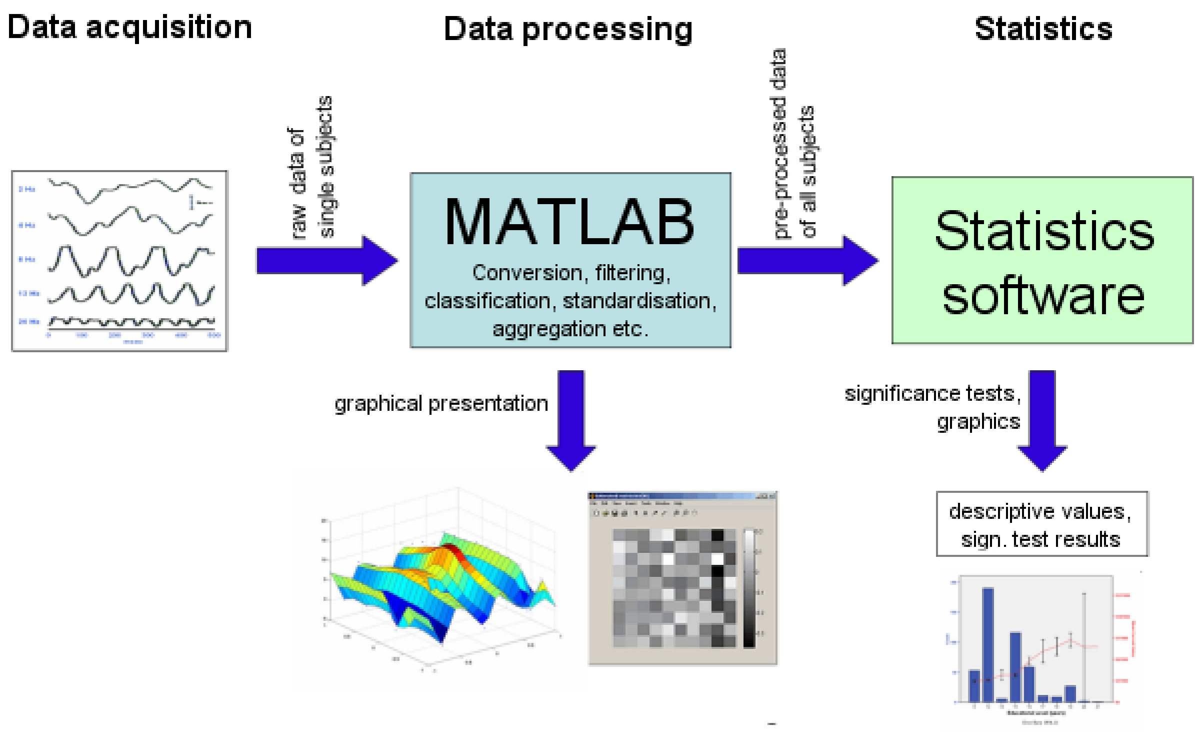
- If you have to evaluate large amounts of data that are acquired automatically by means of computers and other technical appliances. Examples are psychophysical experiments (*EEG* [2], *fMRI* [3], eye tracking, galvanic skin resistance measurements and the like), internet questionnaires, or log file analyses. In these applications, it is way too laborious to manually preprocess every single data set with a spreadsheet software and export it to a statistics software package. Thus, automating this process by means of a Matlab program will be profitable.
- Moreover, the application of programmed data evaluation routines is a more efficient way if the evaluation procedure has to be changed over and over again to 'fine-tune' it. Re-calculating the whole data set with the changed procedure is then done in a snap.
- Matlab offers a plethora of graphics functions, dedicated statistics functions and other interesting features which are not available in other software packages, or only in limited and unflexible implementations.
- Matlab allows for the programming of user-friendly interfaces for data evaluation programs that are repeatedly used. Thus, complicated data evaluation procedures can also be performed by collaborators who are less skilled with the computer.

Certainly, you can write data evaluation programs in other programming languages such as Visual Basic, C++, or Java, but Matlab is a language designed especially for processing, evaluating and graphical displaying of numerical data. A particular advantage of Matlab is that, contrary to most other languages, it can be used as an «interpreter»: you can enter single commands and have them executed immediately; in this way, you can quickly test how the syntax of the command has to be to yield the desired result. The thus verified commands can then transferred by «copy-and-paste» to your program files. And to execute a program, you do not need to pre-process (compile) it beforehand.

Note to programmers: In spite of this user-friendly interface, Matlab programs are executed almost as fast as program code written in compiling languages such as C++ or Pascal. Moreover, it is even possible to compile Matlab programs and run them «stand alone».

Typical data processing workflow
Below you see the schematic diagram of a typical project, showing which steps are often realised with Matlab.

[1] The most important way to represent data in Matlab. In the field of mathematics, a matrix is a table of numbers or other values. Matrices differ from ordinary tables in that they can be used for calculations. Usually, the expression «matrix» refers to two- or higher-dimensional matrices (cf. scalar, vector).


Data acquisition
Data processing
Statistics
raw data of single subjects
MATLAB
Conversion, filtering, classification, standardisation, aggregation etc.
pre-processed data of all subjects
Statistics software
graphical presentation
significance tests, graphics
descriptive values, sign. test results

Exemplary data processing workflow

Examples in Command Line Mode

Command line mode means that commands are directly entered in the «**command window**» on the «command line», i.e. just after the command prompt `>>`. Matlab will then execute them immediately, as opposed to **programmed mode**, i.e. a program is written first, and repeatedly executed later on (more on that subject see **Lesson 2**).

Now you can try out some of Matlab's features yourself. Start Matlab, enter the commands listed in the examples below in the «command window» right behind the command prompt `>>`, and see what happens. For now, you don't have to understand every command in detail; they will be explained later on in this course.

Example 1: The use of Matlab as a (somewhat oversized) «pocket calculator»

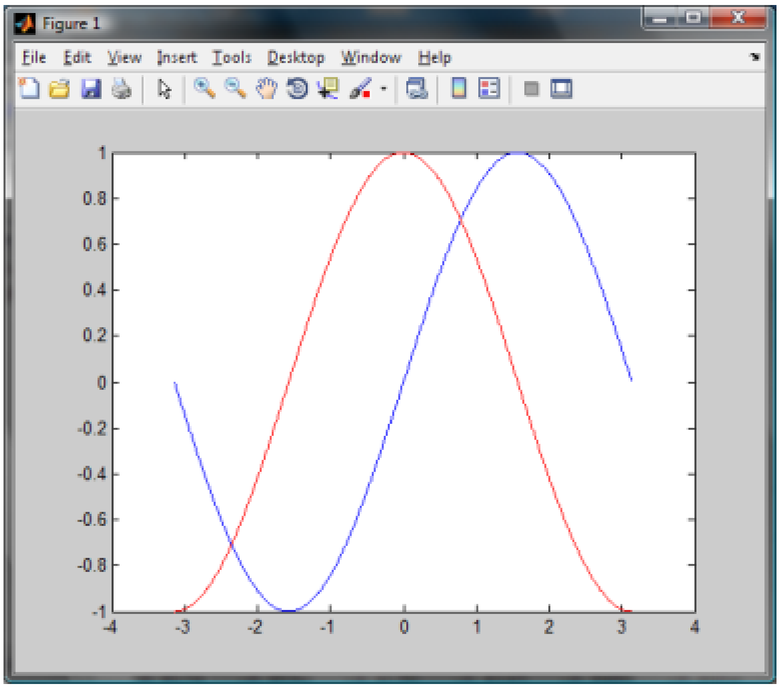
```
>> 27*5
>> 2^10
>> diameter=14.6
>> circumference=diameter*pi
>> area=diameter^2*pi/4
>> ans
```

Explanations: Simple calculations can be executed in the command line mode. If no variable name such as `diameter` is noted, the result will be saved in the default variable `ans` (answer). `pi` is an internally predefined variable that can be used at any time.

Example 2: Plotting a graph in command line mode

```
>> x = -pi:.01:pi
>> y = sin(x)
>> plot(x,y)
>> hold on
>> z = cos(x)
>> plot(x,z,'r')
```

Explanations: First, a sequence of values between -pi and pi with intervals of 0.01 is generated. Then, the sine values of these data are calculated. Subsequently, the results are plotted by means of the `plot` function. `hold on` «fixates» the graph so that it will not be deleted by the subsequent `plot` command. Additionally, the cosine values are calculated and plotted in red into the same graph.



Afterwards, close the graphics window: Either in the usual way windows are closed, or by entering `close` on the command line.

Example 3: Some basic statistics functions demonstrated

```
>> data = randn(1, 1000);
>> plot(data)
>> hist(data)
>> mean(data)
>> std(data)
```

Explanations: First, 1000 normally distributed random values are generated by means of the `randn` function (the semicolon at the end of the line suppresses the display of the generated values in the command window). By means of `plot`, these raw data are visualised in a new graphics window. `hist` draws a histogram showing the normal distribution shape of the data (you might have to get the graphics window to the foreground to see the result). `mean` and `std` calculate the mean and standard deviation values. As you can see now, `randn` generates values with a mean of 0 and a standard deviation of 1.

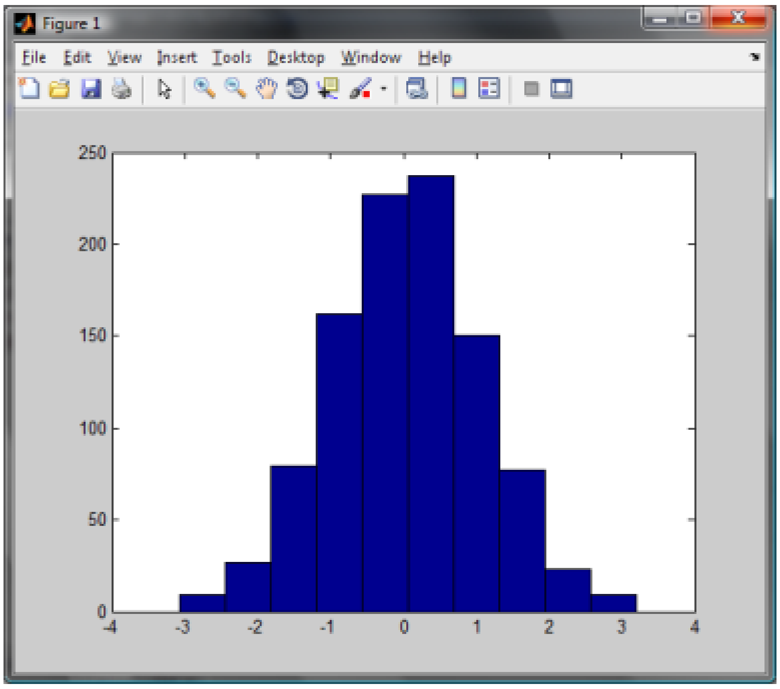
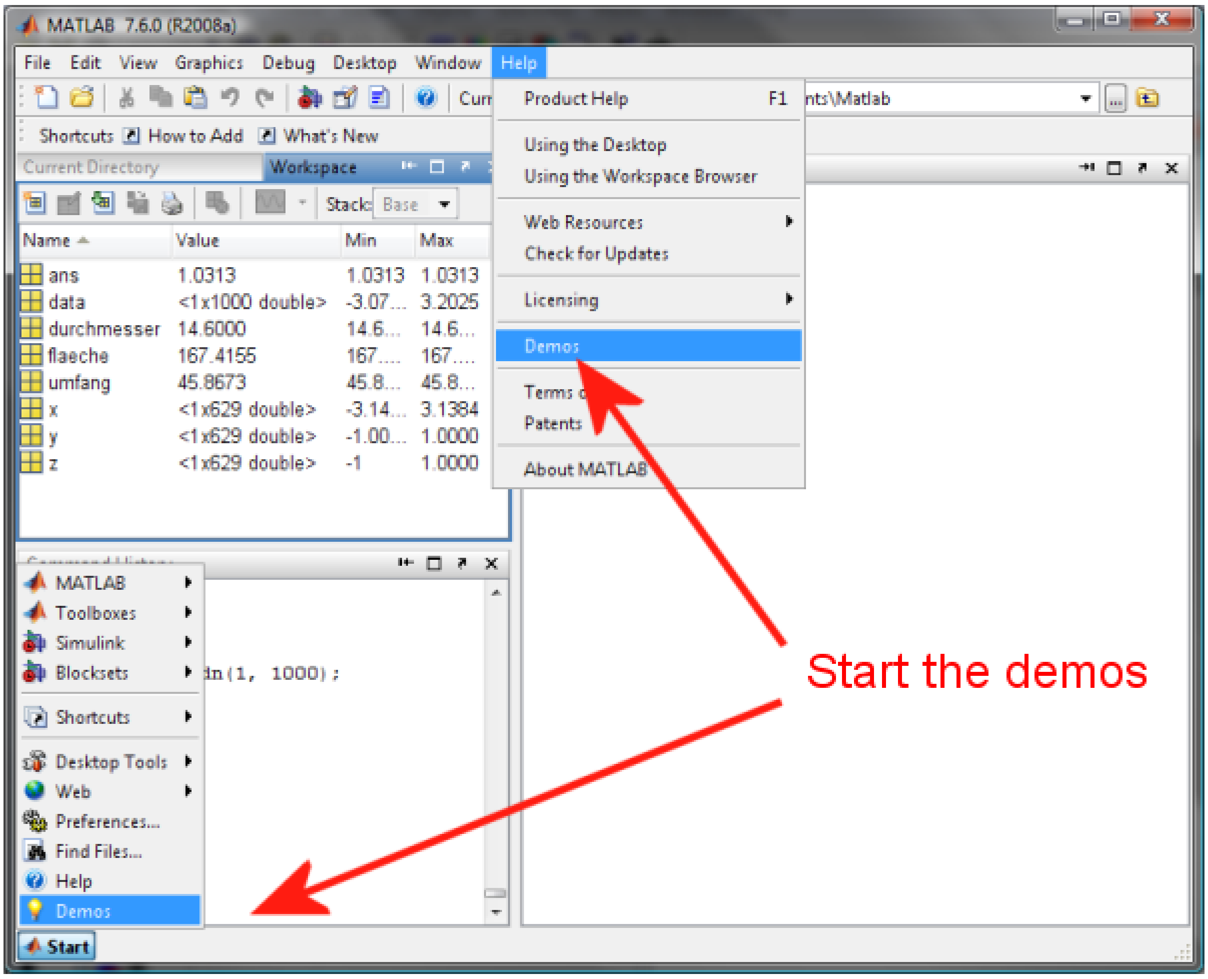


Figure 1
File Edit View Insert Tools Desktop Window Help
250
200
150
100
50
0
-4
-3
-2
-1
0
1
2
3
4

Demonstration Examples by Matlab

In Matlab's documentation, numerous impressive demos are availabe. You can run some of them in order to get a better picture of the power of Matlab.

Select the demos via Matlab's **Start Button** or via the **Help** menu.



Name	Value	Min	Max
ans	1.0313	1.0313	1.0313
data	<1x1000 double>	-3.07...	3.2025
durchmesser	14.6000	14.6...	14.6...
flaeche	167.4155	167....	167....
umfang	45.8673	45.8...	45.8...
x	<1x629 double>	-3.14...	3.1384
y	<1x629 double>	-1.00...	1.0000
z	<1x629 double>	-1	1.0000

```
dn(1, 1000);
```

Two ways to start the demos

Now, the Help system is opened. In the left window panel called help browser, open the directory structure «MATLAB» (click the «+» symbol). Now you can view some of the demos, e.g. in «Graphics». Recommended:

- 2-D Plots
- 3-D Plots
- Vibrating Logo
- Earth's Topography

After selecting a demo, you will have to click **Run in the Command Window** in the top right corner (and then repeatedly click **Next** or **Run this demo** to continue).

Example From Psychological Research

The following graphs exhibit some of the possibilities how to visualise eye movement data by means of Matlab programs (source: **Perception and Eye Movement Laboratory**, Neurology, University Hospital Bern).

To enlarge click on the images.

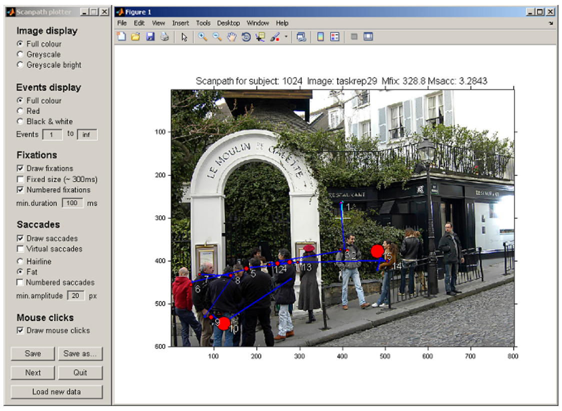

Scanpath plotter

Image display
- (●) Full colour
- ( ) Greyscale
- ( ) Greyscale bright

Events display
- (●) Full colour
- ( ) Red
- ( ) Black & white

Events 1 to inf

Fixations
- [✓] Draw fixations
- [ ] Fixed size (~ 300ms)
- [✓] Numbered fixations

min.duration 100 ms

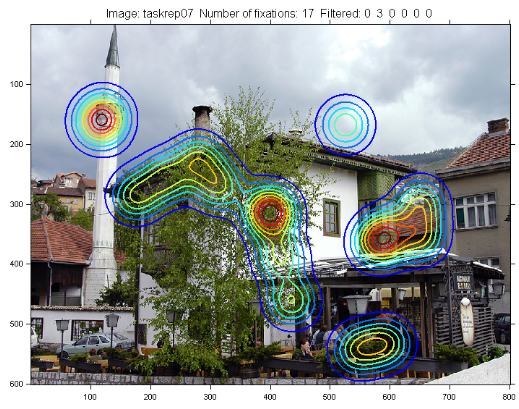
Saccades
- [✓] Draw saccades
- [ ] Virtual saccades
- ( ) Hairline
- (●) Fat
- [ ] Numbered saccades

min.amplitude 20 px

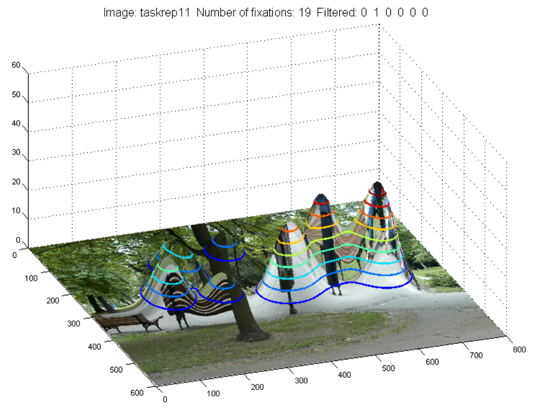
Mouse clicks
- [✓] Draw mouse clicks

Save
Save as...
Next
Quit
Load new data
Figure 1
File Edit View Insert Tools Desktop Window Help
Scanpath for subject: 1024 Image: taskrep29 Mfix: 328.8 Msacc: 3.2843

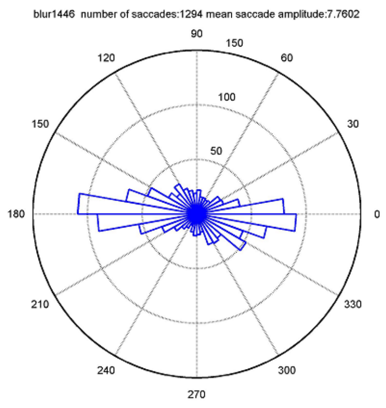
Scanpath plot: Program with graphical user interface

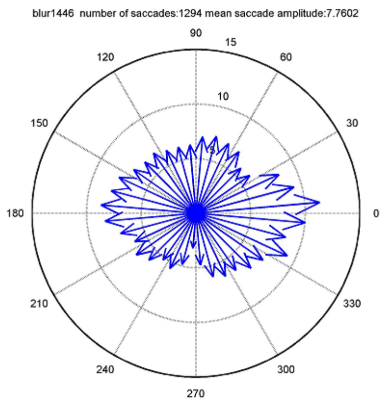


Density plot of fixations: 2D density plot



Density plot of fixations: 3D density plot





Plots of saccade amplitude and frequency per direction

Only pictures can be viewed in this version! For Flash, animations, movies etc. see online version. Only screenshots of animations will be displayed. [link]

Animated scanpath display

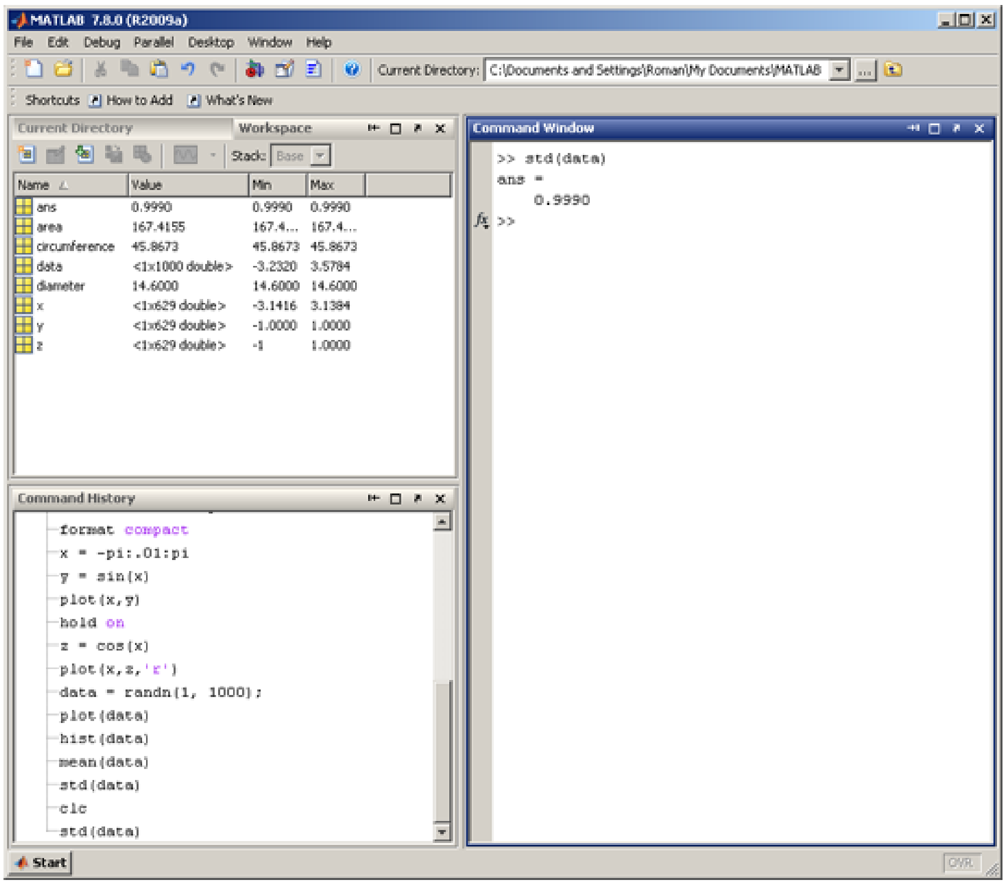
The User Interface

Overview

Matlab's user interface, or «desktop», is partitioned into different sub-windows. To know their functions is essential to be able to work with Matlab efficiently. The most important parts of the user interface will thus be explained in this section:

- the command window
- the workspace
- the current directory browser
- the command history

In addition, it is important to know what information is available through Matlab's help system.


MATLAB 7.8.0 (R2009a)
File Edit Debug Parallel Desktop Window Help
Current Directory: C:\Documents and Settings\Roman\My Documents\MATLAB
Shortcuts How to Add What's New
Current Directory
Workspace
Stack: Base
Command Window
Command History
Start
OVR

Name	Value	Min	Max
ans	0.9990	0.9990	0.9990
area	167.4155	167.4...	167.4...
circumference	45.8673	45.8673	45.8673
data	<1x1000 double>	-3.2320	3.5784
diameter	14.6000	14.6000	14.6000
x	<1x629 double>	-3.1416	3.1384
y	<1x629 double>	-1.0000	1.0000
z	<1x629 double>	-1	1.0000

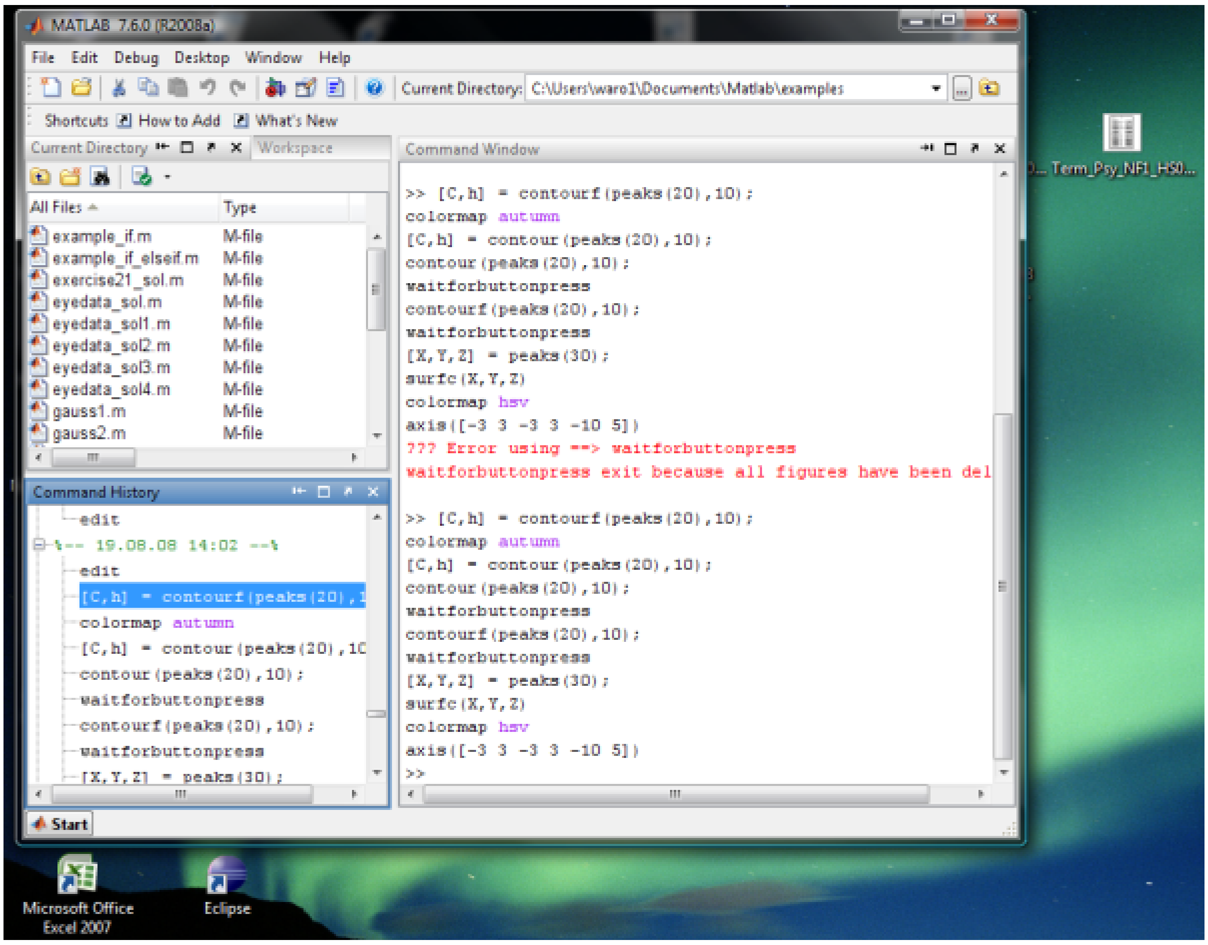
```
>> std(data)
ans =
     0.9990
>>
```

```
format compact
x = -pi:.01:pi
y = sin(x)
plot(x,y)
hold on
z = cos(x)
plot(x,z,'r')
data = randn(1, 1000);
plot(data)
hist(data)
mean(data)
std(data)
clc
std(data)
```

Matlab's user interface (desktop)

Desktop Display Options

The desktop is made up of different areas. The image below shows the default settings.


MATLAB 7.6.0 (R2008a)
File Edit Debug Desktop Window Help
Current Directory: C:\Users\waro1\Documents\Matlab\examples
Shortcuts How to Add What's New
Current Directory
Workspace
Command Window
Command History
Start
Microsoft Office Excel 2007
Eclipse
Term_Psy_NF1_HS0...

All Files	Type
example_if.m	M-file
example_if_elseif.m	M-file
exercise21_sol.m	M-file
eyedata_sol.m	M-file
eyedata_sol1.m	M-file
eyedata_sol2.m	M-file
eyedata_sol3.m	M-file
eyedata_sol4.m	M-file
gauss1.m	M-file
gauss2.m	M-file

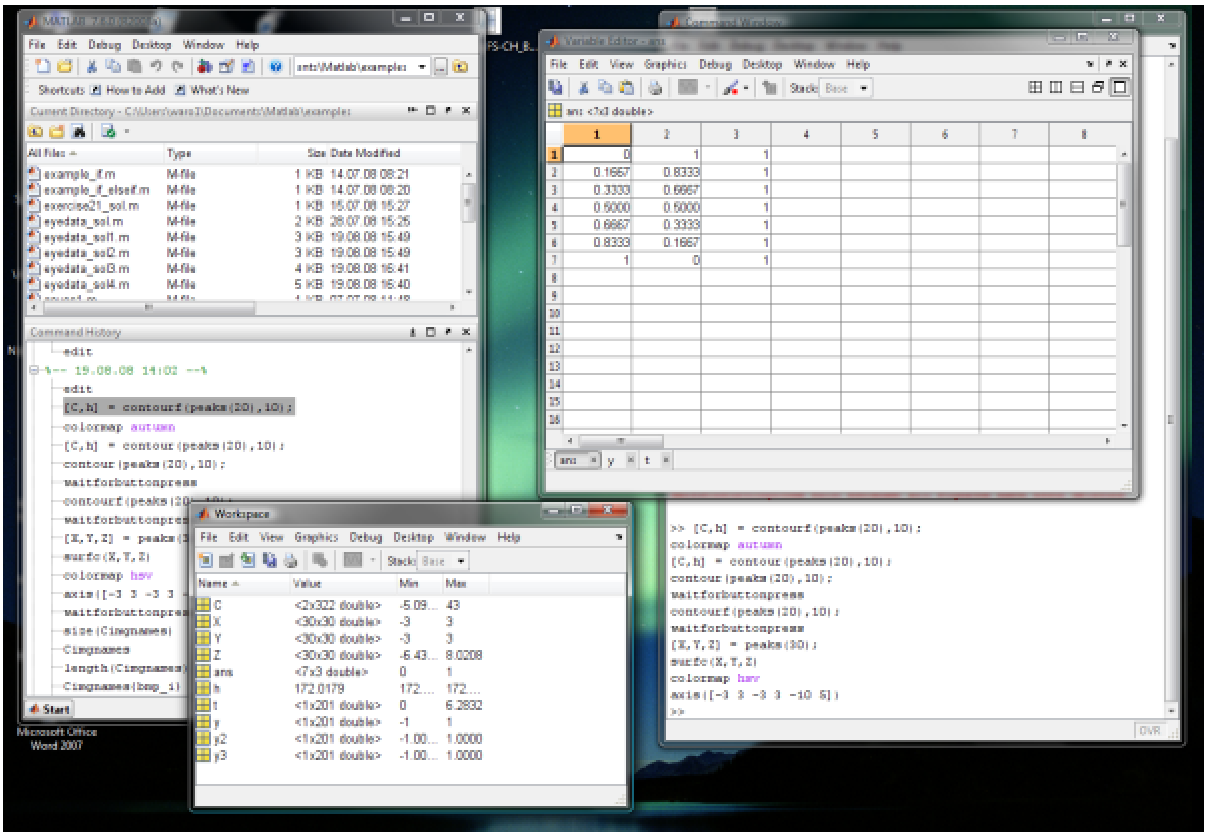
```
edit
%-- 19.08.08 14:02 --%
edit
[C,h] = contourf(peaks(20),1
colormap autumn
[C,h] = contour(peaks(20),10
contour(peaks(20),10);
waitforbuttonpress
contourf(peaks(20),10);
waitforbuttonpress
[X,Y,Z] = peaks(30);
```

```
>> [C,h] = contourf(peaks(20),10);
colormap autumn
[C,h] = contour(peaks(20),10);
contour(peaks(20),10);
waitforbuttonpress
contourf(peaks(20),10);
waitforbuttonpress
[X,Y,Z] = peaks(30);
surfc(X,Y,Z)
colormap hsv
axis([-3 3 -3 3 -10 5])
??? Error using ==> waitforbuttonpress
waitforbuttonpress exit because all figures have been del

>> [C,h] = contourf(peaks(20),10);
colormap autumn
[C,h] = contour(peaks(20),10);
contour(peaks(20),10);
waitforbuttonpress
contourf(peaks(20),10);
waitforbuttonpress
[X,Y,Z] = peaks(30);
surfc(X,Y,Z)
colormap hsv
axis([-3 3 -3 3 -10 5])
>>
```

Default settings of the desktop

The latest Matlab versions (approx. since version 2008a) have a somewhat different default layout. Thus, it is no mistake if your screen does not look exactly the same as shown in the figure.

Further user interface areas can be added or removed by means of the menu item **Desktop**. The available areas can be arranged in various ways, the size and position of window parts can be changed, sub-windows can be shown independently or docked to other parts, or you can shift the dividing lines between sub-windows etc. For these functions, use the buttons in the top right corner of a sub-window and the various options in the desktop menu.



Current Directory - C:\Users\wara3\Documents\Matlab\examples

All Files	Type	Size	Date Modified
example_if.m	M-file	1 KB	14.07.08 08:21
example_if_elseif.m	M-file	1 KB	14.07.08 08:20
exercise21_sol.m	M-file	1 KB	15.07.08 15:27
eyedata_sol.m	M-file	2 KB	28.07.08 15:26
eyedata_sol1.m	M-file	3 KB	19.08.08 15:49
eyedata_sol2.m	M-file	3 KB	19.08.08 15:49
eyedata_sol3.m	M-file	4 KB	19.08.08 16:41
eyedata_sol4.m	M-file	5 KB	19.08.08 16:40

Variable Editor - ans

ans <7x3 double>

	1	2	3	4	5	6	7	8
1	0	1	1					
2	0.1667	0.8333	1					
3	0.3333	0.6667	1					
4	0.5000	0.5000	1					
5	0.6667	0.3333	1					
6	0.8333	0.1667	1					
7	1	0	1					

Workspace

Name	Value	Min	Max
C	<2x322 double>	-5.09...	43
X	<30x30 double>	-3	3
Y	<30x30 double>	-3	3
Z	<30x30 double>	-6.43...	8.0208
ans	<7x3 double>	0	1
h	172.0179	172....	172....
t	<1x201 double>	0	6.2832
y	<1x201 double>	-1	1
y2	<1x201 double>	-1.00...	1.0000
y3	<1x201 double>	-1.00...	1.0000

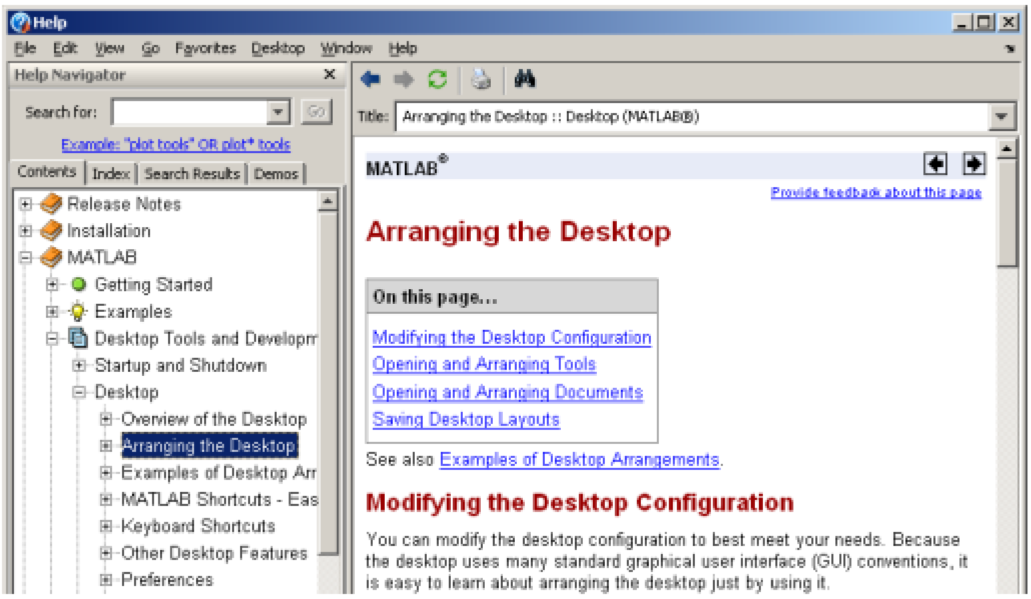
Command Window

```
>> [C,h] = contourf(peaks(20),10);
colormap autumn
[C,h] = contour(peaks(20),10);
contour(peaks(20),10);
waitforbuttonpress
contourf(peaks(20),10);
waitforbuttonpress
[X,Y,Z] = peaks(30);
surfc(X,Y,Z)
colormap hsv
axis([-3 3 -3 3 -10 5])
>>
```

File Edit Debug Desktop Window Help
Shortcuts How to Add What's New
Command History
Start

Example of a different desktop arrangement

If you want to know in more detail what options are available, you can find out in the help system. Open it via the menu **Help** > **Product Help**. The figure below shows how to find information on the desktop.



MATLAB®

Provide feedback about this page

Arranging the Desktop

On this page…

Modifying the Desktop Configuration
Opening and Arranging Tools
Opening and Arranging Documents
Saving Desktop Layouts

See also Examples of Desktop Arrangements.

Modifying the Desktop Configuration

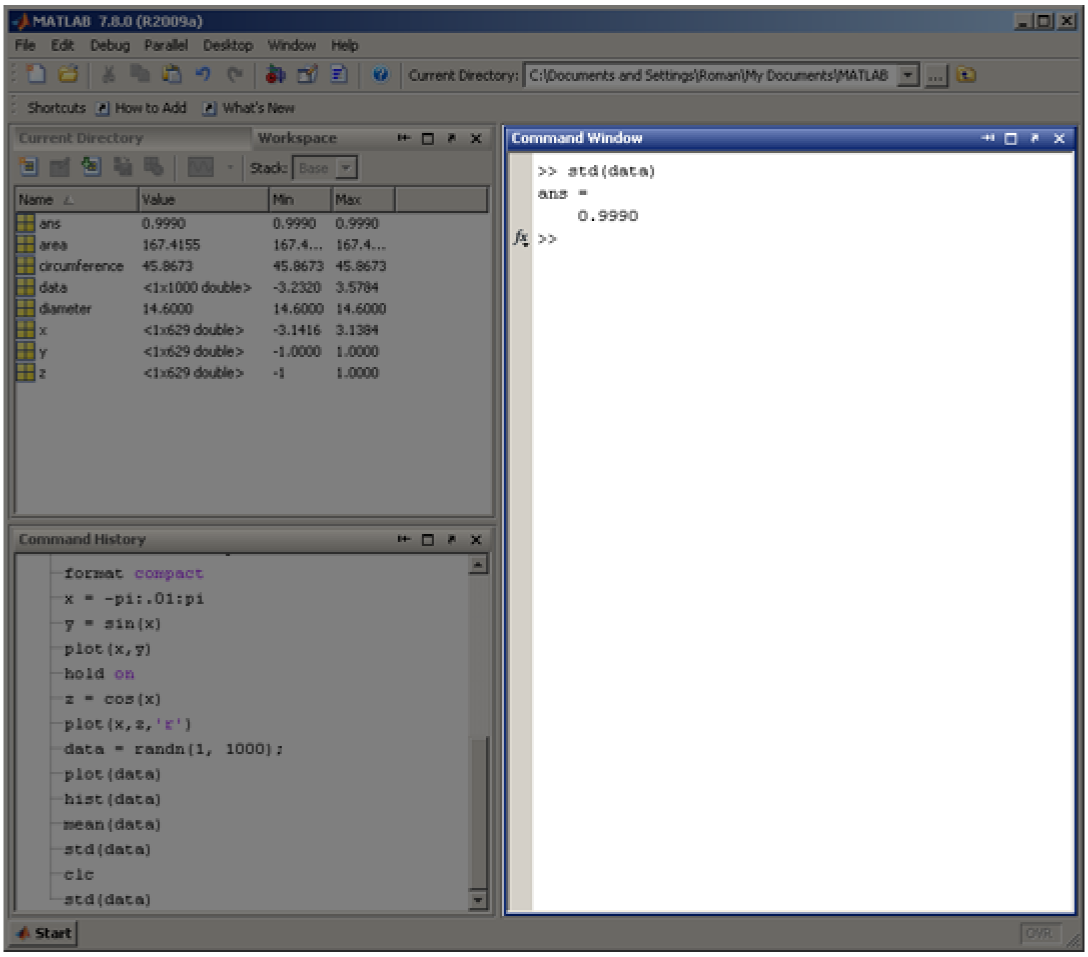
You can modify the desktop configuration to best meet your needs. Because the desktop uses many standard graphical user interface (GUI) conventions, it is easy to learn about arranging the desktop just by using it.

If your Matlab desktop does not correspond to the default layout, please restore the default now by selecting **Desktop** > **Desktop Layout** > **Default**

The «Command Window»

The «Command Window» area is the **communication channel** between you and Matlab. You have already used it to enter the **first examples**. It is used to enter commands and call programs, and the programs can output results and messages to this window. Warnings, error messages and the like also appear here.

Previously typed commands can be re-displayed by means of the <arrow up> und <arrow down> keys, in order to maybe change and re-run them.


MATLAB 7.8.0 (R2009a)
File Edit Debug Parallel Desktop Window Help
Current Directory: C:\Documents and Settings\Roman\My Documents\MATLAB
Shortcuts How to Add What's New
Current Directory
Workspace
Stack: Base
Command Window
Command History
Start
OVR

Name	Value	Min	Max
ans	0.9990	0.9990	0.9990
area	167.4155	167.4...	167.4...
circumference	45.8673	45.8673	45.8673
data	<1x1000 double>	-3.2320	3.5784
diameter	14.6000	14.6000	14.6000
x	<1x629 double>	-3.1416	3.1384
y	<1x629 double>	-1.0000	1.0000
z	<1x629 double>	-1	1.0000

```
format compact
x = -pi:.01:pi
y = sin(x)
plot(x,y)
hold on
z = cos(x)
plot(x,z,'r')
data = randn(1, 1000);
plot(data)
hist(data)
mean(data)
std(data)
clc
std(data)
```

```
>> std(data)
ans =
    0.9990
>>
```

Matlab's «Command Window»

Some command window settings

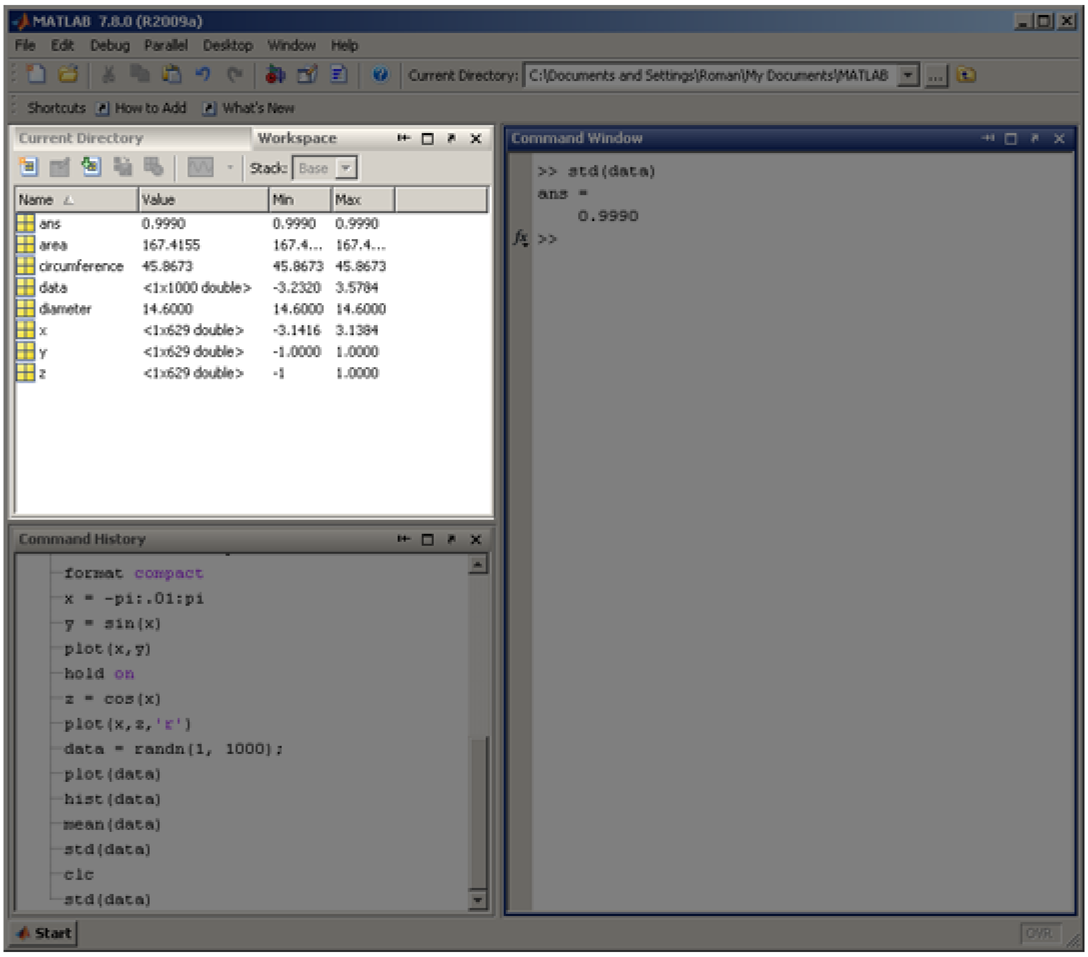
The following commands influence the way how data are shown in the command window:

>> pi	display PI
>> format compact	set the output format 'compact'
>> pi	now it is more compact indeed
>> home	go to the top left position, without clearing the contents of the command window
>> clc	completely clears the command window (clear command window)

Explanations: The command `format` changes the display of values in the command window in manyfold ways. `format compact` disables the output of superfluous empty lines. For further options please refer to the information in the help system (e.g. by typing `docsearch format` in the command window).

The «Workspace»

The workspace can be seen as Matlab's **working memory**. Here, all used variables are stored, and programs can access them directly. In the figure below you can see that the variables used in the demo examples are now displayed in the workspace. Here you can access the variables directly to display their values or to change them.



Name	Value	Min	Max
ans	0.9990	0.9990	0.9990
area	167.4155	167.4...	167.4...
circumference	45.8673	45.8673	45.8673
data	<1x1000 double>	-3.2320	3.5784
diameter	14.6000	14.6000	14.6000
x	<1x629 double>	-3.1416	3.1384
y	<1x629 double>	-1.0000	1.0000
z	<1x629 double>	-1	1.0000

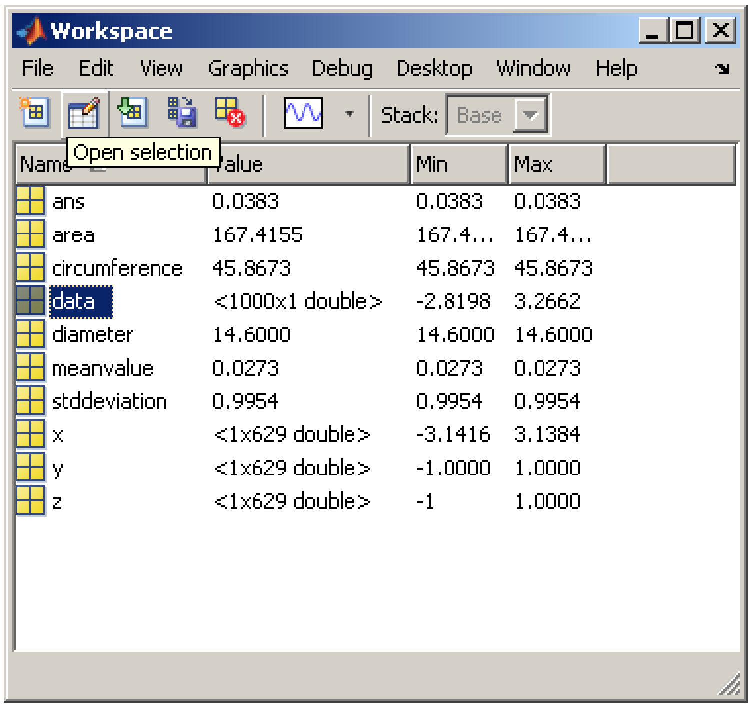
```
format compact
x = -pi:.01:pi
y = sin(x)
plot(x,y)
hold on
z = cos(x)
plot(x,z,'r')
data = randn(1, 1000);
plot(data)
hist(data)
mean(data)
std(data)
clc
std(data)
```

```
>> std(data)
ans =
     0.9990
>>
```

Matlab's "Workspace"

Tools in the Workspace

Select a variable in the workspace with the mouse, and have a look at the tools available in the toolbar of the workspace. The ones used most often are **Open** and the various **graphics tools**.

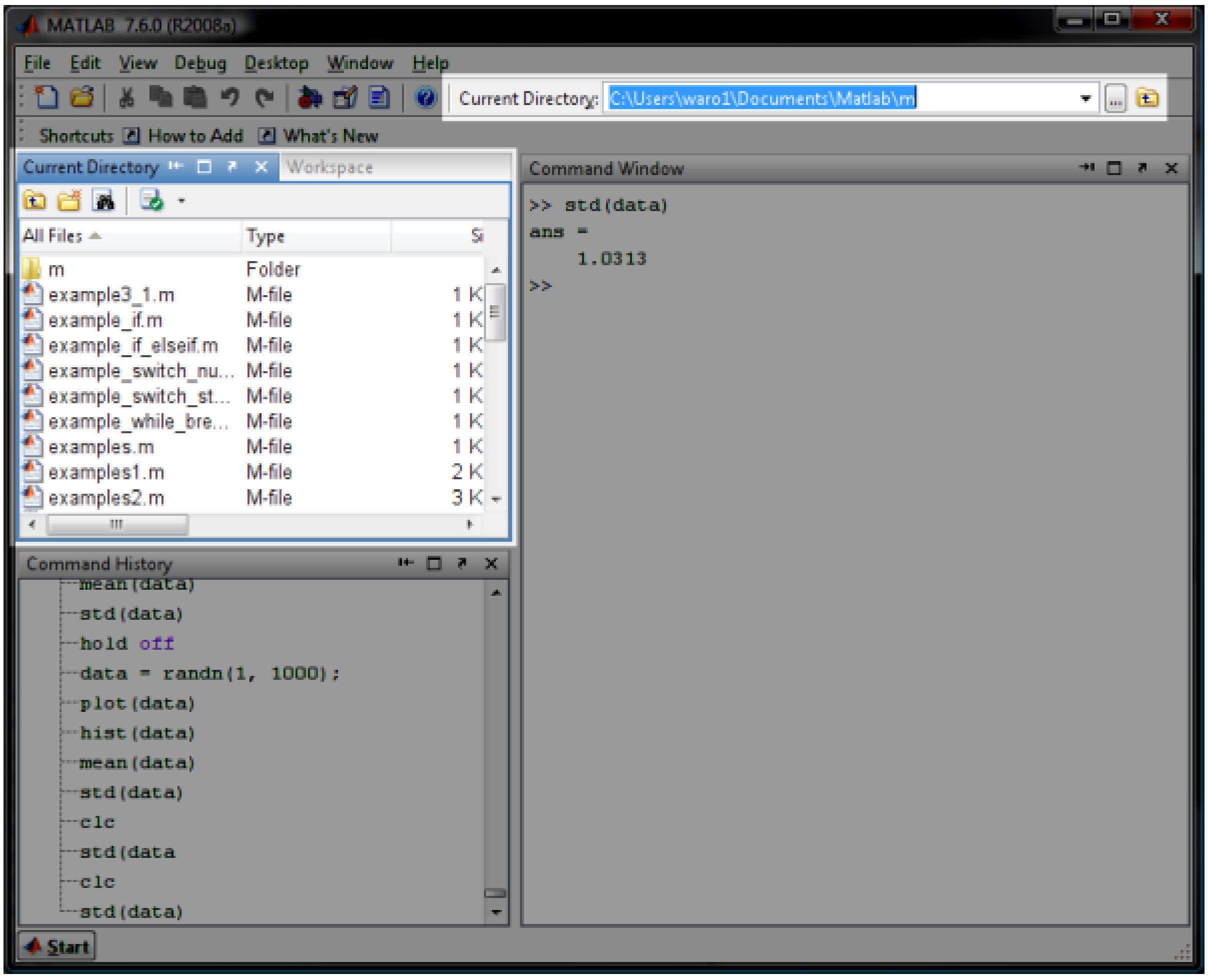


Name	Value	Min	Max
ans	0.0383	0.0383	0.0383
area	167.4155	167.4...	167.4...
circumference	45.8673	45.8673	45.8673
data	<1000x1 double>	-2.8198	3.2662
diameter	14.6000	14.6000	14.6000
meanvalue	0.0273	0.0273	0.0273
stddeviation	0.9954	0.9954	0.9954
x	<1x629 double>	-3.1416	3.1384
y	<1x629 double>	-1.0000	1.0000
z	<1x629 double>	-1	1.0000

The «Current Directory» Browser

This area is placed in the same sub-window as the workspace and can be accessed by means of the current directory tab. It is used to browse the computer's file system and thus select the **current directory for data loading and saving operations** as well as to select the **root directory of the editor**. The latter is indicated in the input field named «Current Directory» in the top right corner of the desktop. Alternatively, it is possible to select the current directory there.

Please do not confuse this with the «search path», by which Matlab has to be informed of where program files are stored, cf. section **Configuration of Matlab**.


MATLAB 7.6.0 (R2008a)
File Edit View Debug Desktop Window Help
Current Directory: C:\Users\waro1\Documents\Matlab\m
Shortcuts How to Add What's New
Current Directory
Workspace
Command Window
Command History
Start

All Files	Type	Si
m	Folder	
example3_1.m	M-file	1 K
example_if.m	M-file	1 K
example_if_elseif.m	M-file	1 K
example_switch_nu...	M-file	1 K
example_switch_st...	M-file	1 K
example_while_bre...	M-file	1 K
examples.m	M-file	1 K
examples1.m	M-file	2 K
examples2.m	M-file	3 K

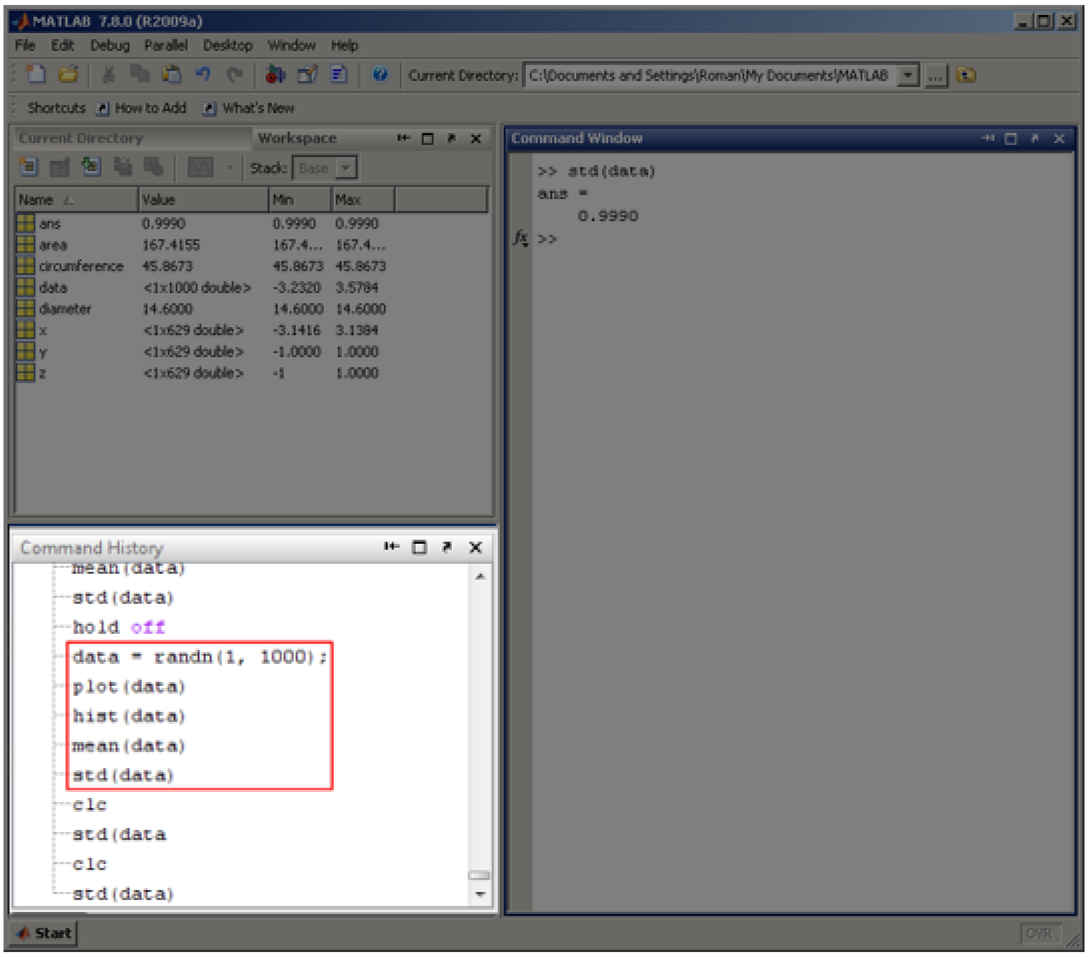
```
mean(data)
std(data)
hold off
data = randn(1, 1000);
plot(data)
hist(data)
mean(data)
std(data)
clc
std(data
clc
std(data)
```

```
>> std(data)
ans =
    1.0313
>>
```

Matlab's «Current Directory» window

The «Command History»

In this sub-window, the **History of your work with Matlab** is listed, i.e. the commands you have recently entered in the command window. This list can be searched, single commands can be re-run, or you can build a program (script) from several commands you have selected in the command history.


MATLAB 7.8.0 (R2009a)
File Edit Debug Parallel Desktop Window Help
Current Directory: C:\Documents and Settings\Roman\My Documents\MATLAB
Shortcuts How to Add What's New
Current Directory
Workspace
Stack: Base
Command Window
Command History
Start
OVR

Name	Value	Min	Max
ans	0.9990	0.9990	0.9990
area	167.4155	167.4...	167.4...
circumference	45.8673	45.8673	45.8673
data	<1x1000 double>	-3.2320	3.5784
diameter	14.6000	14.6000	14.6000
x	<1x629 double>	-3.1416	3.1384
y	<1x629 double>	-1.0000	1.0000
z	<1x629 double>	-1	1.0000

```
>> std(data)
ans =
    0.9990
>>
```

```
mean(data)
std(data)
hold off
data = randn(1, 1000);
plot(data)
hist(data)
mean(data)
std(data)
clc
std(data
clc
std(data)
```

Matlab's "Command History"

How to re-run previously entered commands

Find and select the commands highlighted in the figure above (e.g. by using shift-click or ctrl-click). These commands should still be listed there as you have entered them in the first examples of this lesson; if you have not done this, select some other commands listed there. Then select the menu item «Evaluate Selection» via the context menu (right mouse click on the selected items) or simply press F9. The selected commands are executed again.

Using the Help System

Matlab's help system is extremely well equipped and supersedes any printed manual. It is not only indispensable for beginners when learning Matlab, but also for advanced users to assist in the daily work. Apart from the documentation of the Matlab main program and additional tool boxes, it contains (as you have seen before) a wealth of examples, some introductory tutorials, and hyperlinks to online resources provided by Mathworks.

How to call the Help System

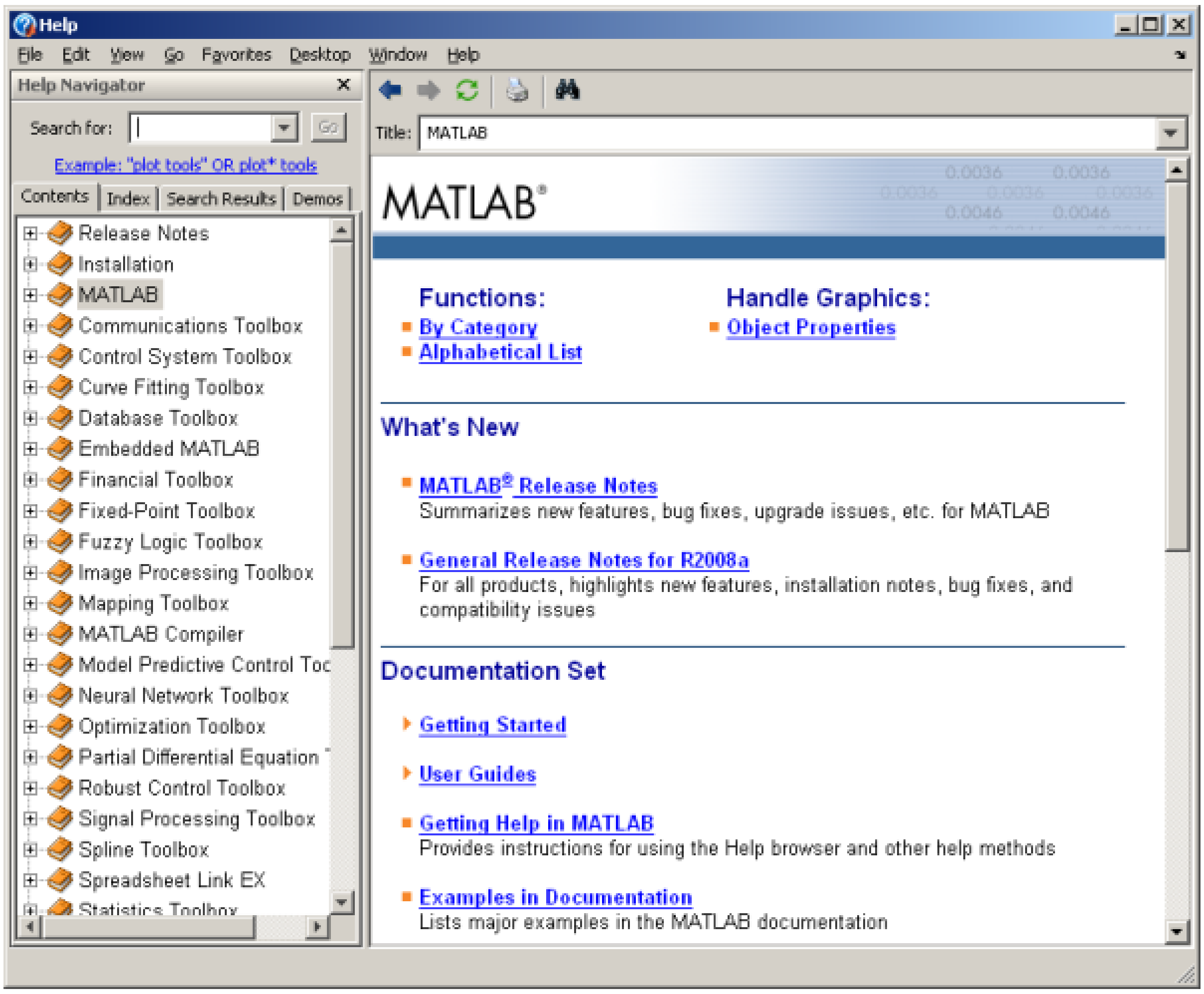
The help system can be called in different ways. The most important ones are listed below (please try them out immediately in Matlab):

- Via **Help > Product Help**. In so doing, the Help Browser/Help Navigator is opened (see figure below)
- By means of the **F1 key**. If you do this from within the Command Window, a reduced version will be opened, similar to other sub-windows of the desktop. In the bottom left corner you can find the link «Open Help Browser» allowing for switching to the full display.
- Help for specific functions: `doc function name`, e.g. `doc sin`. In this way, the help system entry for this function is immediately displayed.
- Similarly, you can select a command typed in the command window before and call help for it by means of the **context menu** (right click) **Help on Selection** (or by pressing **F1**).
- If you don't know exactly what entry you are looking for, you can enter the help system's search function by direct command, e.g. `docsearch operators`

Finding help contents

Matlab offers three ways how to find help for specific questions. They are available in the left part of the help system window, the **Help Navigator**.

1. **Contents**: The help items are shown as a logically structured table of contents.
2. **Index**: alphabetically ordered index of help topics
3. **Search function**: In the field **Search for:** you can search for specific key words. The results are then displayed in **Search Results**, the first found entry is also shown in the main window of the help system.


Help
File Edit View Go Favorites Desktop Window Help
Help Navigator
Search for:
Go
Example: "plot tools" OR plot* tools
Contents
Index
Search Results
Demos
Title: MATLAB

Release Notes
Installation
MATLAB
Communications Toolbox
Control System Toolbox
Curve Fitting Toolbox
Database Toolbox
Embedded MATLAB
Financial Toolbox
Fixed-Point Toolbox
Fuzzy Logic Toolbox
Image Processing Toolbox
Mapping Toolbox
MATLAB Compiler
Model Predictive Control Toc
Neural Network Toolbox
Optimization Toolbox
Partial Differential Equation
Robust Control Toolbox
Signal Processing Toolbox
Spline Toolbox
Spreadsheet Link EX
Statistics Toolbox

MATLAB®

Functions:
- By Category
- Alphabetical List

Handle Graphics:
- Object Properties

What's New

- MATLAB® Release Notes
  Summarizes new features, bug fixes, upgrade issues, etc. for MATLAB
- General Release Notes for R2008a
  For all products, highlights new features, installation notes, bug fixes, and compatibility issues

Documentation Set

- Getting Started
- User Guides
- Getting Help in MATLAB
  Provides instructions for using the Help browser and other help methods
- Examples in Documentation
  Lists major examples in the MATLAB documentation

Matlab's help system

Furthermore, immediately after starting the help system, several options are offered in the **main window**: Searching by categories, alphabetical list, searching for object properties, and hyperlinks to printable versions of the documentation (pdf) and external resources at the Mathworks homepage.
Similar to web browser functionnality, in the main window you can navigate back and forth, search, store favourites, or print.

How to copy example code from the help system

Code examples as listed in certain help topics can simply be transferred to the command window (or a self-written program) by means of standard copy-and-paste procedures. In the command window, you can then test the example.

- Type `plot3` in the Help Navigator and click **Go** (alternatively: enter `doc plot3` on the command line)
- scroll down and select the four program code lines after «Examples: Plot a three-dimensional helix»
- Select **Edit** > **Copy** or press **Ctrl-C**
- Switch to the command window, and select **Edit** > **Paste** or press **CTRL-V**
- Press the **Enter key** to execute the commands, and... WOW!

Summary

The most important contents of the section «The User Interface» are summarised below.

- The «Command Window» area is the communication channel between the user and the Matlab program. Here, commands can be entered, and programs can output results and messages. Warnings, error messages and the like are also shown here.
- In the Command Window, previously entered commands can be repeated with the keys <arrow up> and <arrow down>, and then changed and re-executed.
- The «Workspace» is the working memory of Matlab. It stores the used variables on which programs operate.
- The «Current Directory» browser is used to access the computer's file system. There, you can select the current directory for data storage as well as the root directory for the editor.
- In the window «Command History» you find the commands you have entered throughout the current session in the command window.
- Matlab's help system is an indispensable aid for beginners as well as experienced users. It makes a printed manual unnecessary.

Numbers and Variables in Matlab

As you have seen in section **Why Matlab?**, calculations, graphical displays etc. can be performed in a simple way in the command line mode, i.e. by directly typing commands in the **Command Window**.
In the next sections you can learn the basics that are necessary to work in the command line mode:

- How are numbers represented in Matlab?
- What are variables?
- The most important data type: the matrix
- How to work with matrices
- What operators and functions can be used for calculations?

These points not only pertain to the command line mode, but also to self-written programs, as you will learn in **Lesson 2: Programming in Matlab**
Using the command line mode allows for testing commands before including them in a program, which is useful in case you are not sure of their syntax or function.

Representation of Numbers

As usual in English, numbers with decimal places are written with a **period**. The **comma** is used to separate consecutive commands on the same line. The result of an operation (below it will only be entering a number) is assigned to the predefined variable `ans` (answer), unless it is explicitely assigned to a specific **variable**.

Please enter the commands listed below in the Command Window and try to understand what Matlab does with your inputs.

>> format compact	activate compact display
>> 23.736	correctly entered number
>> 784,400	this is interpreted as TWO numbers, separated by the comma
>> pi	i.e. you must not write large integer numbers with commas
	`pi` is a pre-defined constant, that can be used at any time

For numbers in **scientific notation**, the letter **e** is used to separate the exponent (to the base of 10). Examples:

>> 1.746e5	for 1.746 x 10^5
>> 584e-11	of course, negative exponents are allowed as well

Contrary to other programming languages, Matlab has representations for two special values: infinite (inf) and «not a number» (NaN). These values can be the result of calculations, such as division by zero, and can as such also be assigned to a variable.

Please try out the following commands that illustrate this.

>> 2/0	division by 0 yields `inf`
>> -5/0	minus infinite is also possible
>> 0/0	0 devided by 0 results in `NaN`
>> inf/inf	
>> 0*inf	

Variables

Data are stored in Matlab's working memory (i.e. the workspace) as **variables**.

A variable is a reserved «place» in computer memory that can be referenced with a unique name.
A variable can contain various kinds of data. This fact is expressed by saying that a variable is of a certain **data type** (or just «type»).

Examples for data types are simple numbers, matrices, character sequences (strings), structured data etc.
In the examples below we only use simple numbers. You will get to know further data types later on in this course.
Contrary to many other programming langauges, variables do not have to be pre-defined (i.e. to reserve the necessary memory space for them). Matlab will automatically reserve the memory space at the time of their first usage, that is when they are assigned a value for the first time.

Variable naming conventions
A variable is identified by a unique name. The name has to begin with a letter, after that it can contain further letters, numbers, or the «underscore» `_`.
Variable names are **case-sensitive**, i.e. capitalising is relevant.

In the examples below you can see what variable names are allowed, how variables can be filled with values, and how these values can be recalled again.

>> Speed = 63	the equal sign assigns a value to a variable
>> g65_fubar = 3	another valid name
>> Speed	recall the value
>> speed	this does not work when spelt lowercase
>> 6test = 2	a name not starting with a letter is invalid
>> _foo = 1	ditto
>> hi$there	the only special character allowed is the underscore _
>> pi = 11	It is even possible to redefine pre-defined constants like `pi` or function names
>> sin = 0	(here, the sine function `sin`), but has to be used with care (or better not at all)...

Loading and saving variables in the Workspace
Alle variables that have been used are visible in the **Workspace**. The contents of the Workspace can be saved to the hard disk with `save` (to the folder that is selected in the **Current Directory Browser** window) and loaded back from the hard disk by using `load`.
Single variables can be removed from the Workspace with the `clear` command.

Please try it out again by yourself:

>> save data	saves all variables to the file **data.mat**
>> clear Speed	now the variable Speed is no longer in the Workspace
>> clear	deletes **all** variables (note the Workspace window)
>> load data	loads back the variables from the file (see the Workspace window)

Saving and loading workspace variables can also be done by means of the menu buttons of the Workspace window. Single variables can directly be deleted in the Workspace (select the variable and hit the 'delete' key or use context menu > delete).

Summary

In the following, you find a brief summary of the most important points of the section «Numbers and Variables».

- Decimal places of a number are separated by a **period**.
- Scientific notation is spelt with `e`, for instance `2.7e6` for 2.7 x 10 ^{6}.
- In Matlab, operations such as division by zero, that are not allowed in other programming languages or applications, result in defined values such as `inf` for infinite or `NaN` für «not a number».
- A variable is referenced by a unique name. It has to begin with a letter and can contain more letters, numbers or the «underscore» _.
- Variable names are case-sensitive.

Data Representation 1: Scalars, Vectors, and Matrices

Matlab knows various ways to represente data. The most important one is the representation as **Matrix**, thus the name Matlab (**Mat**rix **Lab**oratory).
The most common form of a matrix is the two-dimensional matrix, comparable to a table. Matrices can only contain numerical values (exception: character strings, see section **Data Representation 1: Character Strings** in the next lesson.
A matrix is well suited to store data in a similar way as it is usually done when building a table - e.g. to quickly create a new data set, or to hold data that have been acquired by a technical system.
Example: A simple data set, containing the subject number in the first column, his/her sex (e.g. 1 for female, 2 for male) in the second, the age in the third, and the measured values of two dependent variables in the forth and fifth column.

1	1	25	0.8	56
2	2	33	0.65	65
3	2	26	0.97	45
4	1	26	0.78	50
5	1	29	0.77	50

A simple matrix with data from five subjects

According to the terminology used in linear algebra, some special cases of matrices in terms of their number of dimensions are referred to as:

- **Scalar**: «null-dimensional» matrix, i.e. only one element
- **Vector**: one-dimensional matrix
- **Matrix**: matrix with two or more dimensions. Sometimes, they are also called **arrays**.

You will learn more about these data types in the following sections.

Scalars

A scalar is a *matrix* with one row and one column, and thus only contains a single numerical value.

The notation with square brackets `[ ]` always designates a matrix, including scalars and vectors! (with one exception, see section **Programming of Functions**, Example 3).

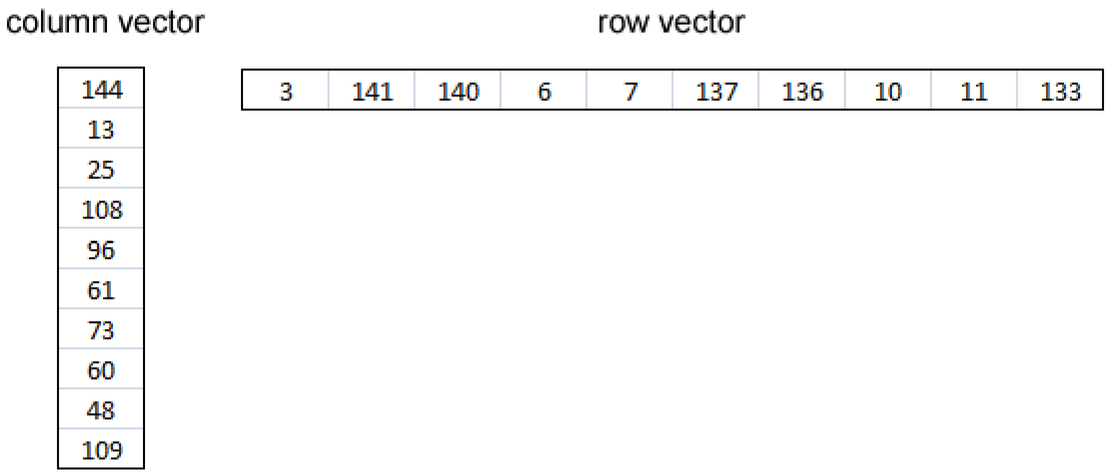
Please try this out yourself:

>> sc = 93	This is a scalar. Thus, the square brackets are not mandatory here.
>> sc2 = [13]	more general notation as 1x1 matrix
>> sc(1,1)	is identical to `sc` - the element in row 1, column 1
>> isscalar(sc)	determines whether `sc` is a scalar (1 = yes, 0 = no)

Vectors

A vector contains more than one numerical element; these elements are arranged side by side «in single file». Dependent on whether the elements are arranged vertically or horizontally, they are referred to as:

- **column vectors** (one column, any number of rows)
- **row vectors** (one row, any number of columns)


column vector
row vector

144
13
25
108
96
61
73
60
48
109

3	141	140	6	7	137	136	10	11	133

In other words, a vector is a *matrix* with only a single row or a single column.

Try to understand the examples below:

>> rvect = [1 4 8]	a row vector with three elements
>> rvect = [1,4,8]	the elements can be separated by **spaces or commas**
>> rvect(2)	read back the second element of the vector
>> rvect(0)	**Important**: contrary to other languages, there is no 0th element!
>> cvect = [12; 55; inf]	a column vector. The semicolon separates the rows
>> cvect(1:2)	returns the first to the second element of the vector
>> seq = 0:2:10	generates a vector with numbers in equal distance
>> seq2 = 20:30	if the middle value is omitted, the distance (increment) is 1
>> seq3 = pi:-.1:-pi	negative or fractional increments are allowed as well
>> seq2'	The apostrophe transposes a matrix.
>> isvector(seq2)	here: Converstion of a row into a column vector
>> length(seq3)	is this a vector?
	determines the length of a vector

Matrices

In the field of mathematics, a matrix is a table of numbers. The most common kind of matrix is the two-dimensional one (m rows, n columns), very similar to a table as you know it from applications such as MS Excel (but only containing numerical values). Matlab also supports matrices with more than two dimensions.

Demonstration example: The matrix in Dürer's «Melancholia»

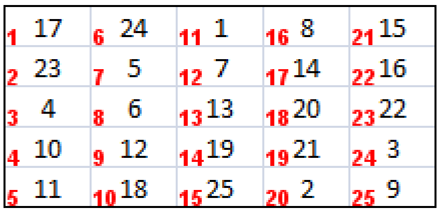
>> load durer	loads a pre-defined data set into the workspace (variable X)
>> image(X)	shows it as an image
>> colormap(map)	loads the correct colour palette. Can you see the matrix?
>> load detail	an other data set containing the image detail
>> image(X), colormap(map)	a magic square!
>> m4 = magic(4)	Matlab can do this, too (find the difference!)

Matrices can be built in various ways, and accessing the elements of a matrix is also possible in numerous ways, as you can see in the examples below.

Building matrices and accessing their contents and properties.

>> M3=[12 13 10;3 5 8;3 2 1]	a 3x3 matrix
>> M3(2, 3)	access single elements (row, col)
>> M3(:, 3)	the entire 3rd column; the : designates all elements
>> M4 = [1:4; 6:2:12]	the notation for number sequences is possible, too
>> M4(1, 2:end)	end designates the last element
>> M4(3, 4)	lies «beyond» the matrix
>> M4(3, 4) = 55	Matlab will automatically enlarge the matrix and fill with zeroes
>> M5 = M4(1:2, 2:3)	assign a part of the matrix to a new variable
>> M4(1:2, 2:3) = -1	assign a value to only part of the matrix
>> mag = magic(4)	the magic square again
>> mag = mag(:, [1 3 2 4])	convert it into Dürer's square by swapping column 2 and 3
>> M4(1, 1, 3) = 100	i.e. indexing with a vector
>> matrixsize = size(M4)	expand matrix M4 to a three-dimensional one
>> size(M4, 1)	yields the size of the matrix (as a vector!)
>> ndims(M4)	ask for the number of rows only (2 would be the columns)
>> numel(M4)	number of dimensions
>> emptymatrix = [];	number of elements
>> isempty(emptymatrix)	create an empty matrix
>> isempty(mag)	ask whether the matrix is indeed empty
>> p = pascal(8)	generate yet another matrix (Pascal's triangle)
>> p(3, :) = []	in this way, you can remove a row from a matrix
>> p(:, 6) = []	the same to delete a column

Linear indexing



1 17	6 24	11 1	16 8	21 15
2 23	7 5	12 7	17 14	22 16
3 4	8 6	13 13	18 20	23 22
4 10	9 12	14 19	19 21	24 3
5 11	10 18	15 25	20 2	25 9

Linear indexing allows for a different way of accessing the elements of a matrix. Instead of addressing them by row and column index, a matrix can be thought of as being serially numbered (first in columns, then in rows - see figure). In that way, with a single argument, you can address the x'th element of a matrix.

In the pictured matrix, the element `data(2, 4)` is thus identical with `data(17)`, containing the value 14.

Summary

Below, you again find the most important points in a brief summary. Additionally, relevant Matlab commands are listed.

Points to remember

- A scalar is a matrix with one row and one column, that is with only a single numerical value.
- A vector is a matrix with a single row (row vector) or a single column (column vector).
- The most common matrix is the two-dimensional matrix (m rows, n columns). Matlab also allows for higher-dimensional matrices, e.g. with m rows, n columns and p planes.
- When defining matrices and vectors, the elements have to be enclosed in square brackets `[ ]`. For scalars, the brackets are optional.
- The elements of matrices and vectors that belong to the same row are separated with a space or comma. The semicolon separates rows.
- Number sequences (= vectors) can be produced with the `:` - notation, e.g. `10:2:50` for all even numbers from 10 to 50.
- A solitary colon designates all elements of a row or column, e.g. `data(:, 12)` for all rows, but only column 12 of the matrix.
- When indexing elements of a matrix, the indices are enclosed in round brackets `( )`.
- The indices always run from 1 to n, the index 0 (the «zeroth» element) does not exist in Matlab.
- In two-dimensional matrices, the first index is the row, the second one is the column.
- Linear indexing: Instead of addressing elements by row and column index, a matrix can be seen as being serially numbered; thus you can address the x'th element of a matrix with a single argument.

Commands

- `isscalar, isvector`: check whether a variable is a scalar or a vector, respectively
- `length` determines the length of a vector
- `ndims` returns the dimensionality (number of dimensions) of a matrix
- `numel` calculates the total number of elements of a matrix
- `size` returns the size of a matrix
- `isempty` determines whether a matrix is empty

Matrix Manipulation

In this section, you learn various possibilities to manipulate *Matrices*:

- matrix concatenation
- matrix duplication
- creating special matrices
- matrix transformation

This section will be concluded with a first practical exercise.

Matrix Concatenation

Matrices can be concatenated with the normal `[ ]` -notation.

Please try to comprehend this by following the examples below:

>> a = magic(5);	the semicolon prevents the output of `a` in the Command Window
>> b = randn(5,3);	random numbers, 5 rows, 3 columns
>> c = [a b]	horizontal concatenation
>> d = [a; b]	vertical concatenation: does not work this way because the number of columns
>> b = randn(5,5);	is inequal
>> d = [a; b]	thus, we convert `b` into a 5x5 matrix
>> e = [a, d, a]	now it works
	you can also concatenate more than two matrices at once

The above examples clarify how the **semicolon after a statement** is used to suppress the output of generated values to the command window.

Besides the above demonstrated possibilities, Matlab provides dedicated functions for matrix concatenation.

This is again demonstrated by means of some examples:

>> horzcat(a, b)	horizontal concatenation
>> vertcat(a, b, a)	vertical concatenation; more than two matrices are possible

An overview of the basic matrix manipulation commands can be found in the help system, e.g. by entering `doc elmat` (**el**ementary **mat**rices and matrix manipulation.)

Matrix Duplication

The two most important matrix duplication functions are called `repmat` (**rep**etition of **mat**rices) and `cat` (con**cat**enation).

Please study the usage of these two commands by doing the manipulations below. The variables `a` and `b` should still be in your workspace, as you have used them in the previous examples.

>> m = magic(3);	matrix duplication. Here, duplicate twice vertically and three times horizontally
>> n = repmat(m,2,3)	general function: concatenate along dimension n (1st argument)
>> cat(1, a, b)	1 = vertically, 2 = horizontally, 3 = in «planes»
>> cat(2, a, b)	

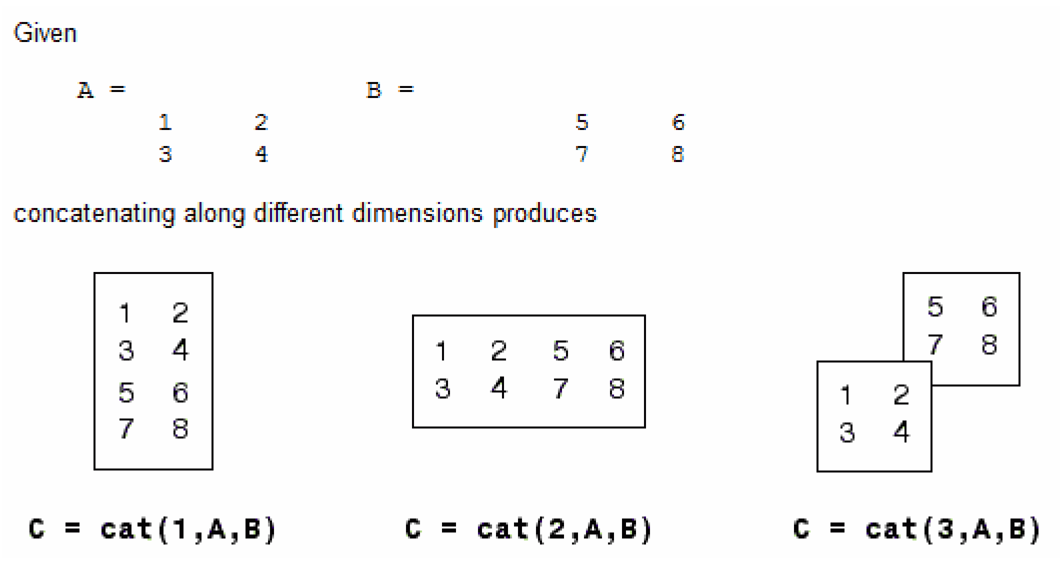


Illustration of the cat function

Creation of Special Matrices

Certain special matrices are needed again and again, thus Matlab provides commands to create such matrices. Some examples:

- Matrices filled with zeroes or ones
- Matrices with random numbers with different distributions. This can sometimes be helpful to test statistical functions or generate test data sets. Also see **Random Numbers** in Lesson 4
- Particular matrices for linear algebra: identity matrix, Pascal's triangle etc.

In the examples below, you can see the most important ones of these functions.

>> zeros(7, 3)	Matrix with zeroes
>> allones = ones(4)	Matrix with ones. If only one argument is given, a square matrix results
>> a(1:5, 1:10) = 3.5	To create a matrix with other values, there are several ways.
>> b = ones(7, 3) * 81	random numbers between 0 and 1
>> rand(1, 8)	normally distributed random numbers with a mean of 0
>> randn(2, 6)	and a standard deviation of 1
>> eye(5)	so-called identity matrix (only square matrix is possible), linear algebra
>> pascal(7)	Pascal's triangle matrix, a thing more for mathematicians
>> magic(8)	magic squares

Matrix Transformation

The most important transformations are:

- Mirroring (horizontally or vertically)
- Rotation
- Transposition (swap rows and columns)

Examples to try out:

```
>> a = [1 2 4 8 16; 0 1 2 3 4; -1 -6 0 1 6];
```

>> fliplr(a)	flip left-right: mirror aling the vertical axis
>> flipud(a)	flip up-down: mirror along the horizontal axis
>> rot90(a)	counter-clockwise rotation by 90 degress (default: once)
>> rot90(a, 3)	rotate by 90 degrees three times
>> a'	transpose matrix (exchange rows and columns)

Exercise 1: Create a Complex Data Matrix

You have now learned the basics of how to create matrices with various contents and how to manipulate them. For learning success it is central that you now try to use this knowledge yourself, by solving an example on your own.

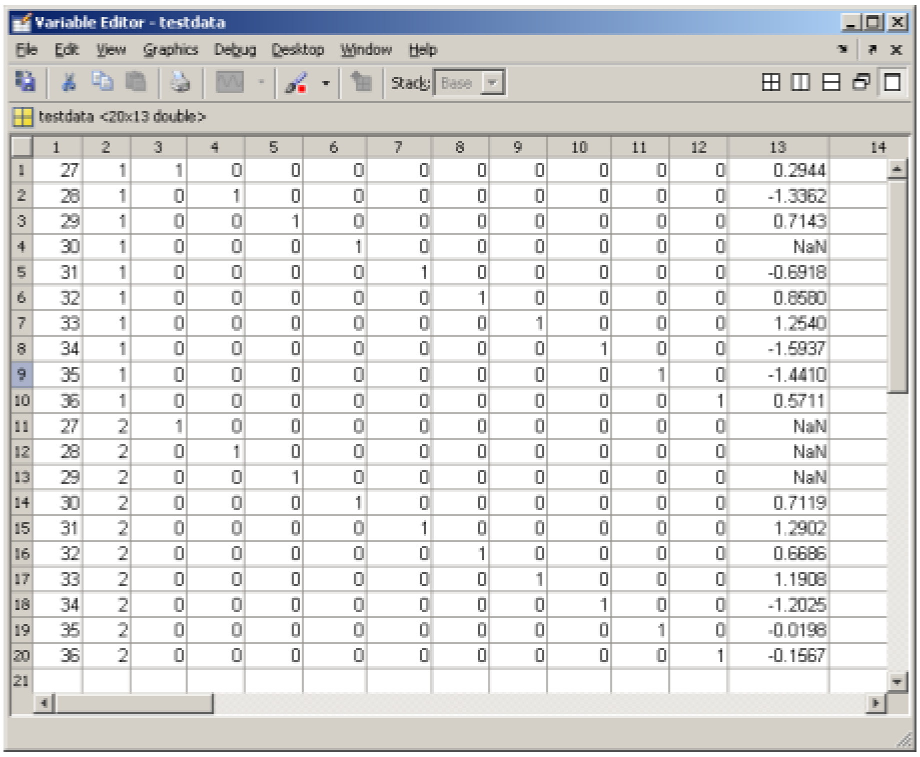
Task description

Sometimes, it is helpful to be able to create a test data set to test a data evaluation program before you have acquired «real» data. Try to do this now in the following exercise.
At the end of the exercise you will find a link to the sample solution that you can consult in case you need help. However, it is important to realise that there is more than one possible solution.

Please do not fall back on the sample solution too early, and try first to use the help system if necessary.

Create the matrix `testdata` in the workspace, containing hypothetical test data as pictured below. Try to accomplish this with as few commands as possible.

Explanation of the data:
Column 1: subject number
Column 2: 1st or 2nd run of the experiment
Columns 3-12: mark several experimental conditions
Column 13: normally distributed random numbers (they do not have to be identical with the pictures values).
Four values are labelled as "missing data" by the value `NaN`.
Tips for functions that can be used: eye, repmat, randn


Variable Editor - testdata
File Edit View Graphics Debug Desktop Window Help
Stack: Base

testdata <20x13 double>

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	27	1	1	0	0	0	0	0	0	0	0	0	0.2944	
2	28	1	0	1	0	0	0	0	0	0	0	0	-1.3362	
3	29	1	0	0	1	0	0	0	0	0	0	0	0.7143	
4	30	1	0	0	0	1	0	0	0	0	0	0	NaN	
5	31	1	0	0	0	0	1	0	0	0	0	0	-0.6918	
6	32	1	0	0	0	0	0	1	0	0	0	0	0.8580	
7	33	1	0	0	0	0	0	0	1	0	0	0	1.2540	
8	34	1	0	0	0	0	0	0	0	1	0	0	-1.5937	
9	35	1	0	0	0	0	0	0	0	0	1	0	-1.4410	
10	36	1	0	0	0	0	0	0	0	0	0	1	0.5711	
11	27	2	1	0	0	0	0	0	0	0	0	0	NaN	
12	28	2	0	1	0	0	0	0	0	0	0	0	NaN	
13	29	2	0	0	1	0	0	0	0	0	0	0	NaN	
14	30	2	0	0	0	1	0	0	0	0	0	0	0.7119	
15	31	2	0	0	0	0	1	0	0	0	0	0	1.2902	
16	32	2	0	0	0	0	0	1	0	0	0	0	0.6686	
17	33	2	0	0	0	0	0	0	1	0	0	0	1.1908	
18	34	2	0	0	0	0	0	0	0	1	0	0	-1.2025	
19	35	2	0	0	0	0	0	0	0	0	1	0	-0.0196	
20	36	2	0	0	0	0	0	0	0	0	0	1	-0.1567	
21														

Sample solution

```
>> testdata = (27:36)' % subject numbers upper half
>> testdata = [testdata ones(1, 10)' eye(10)] % group numbers of group 1 and diagonal
>> testdata = repmat(testdata, 2, 1) % duplicate the whole matrix vertically
>> testdata(11:20, 2) = 2 % group numbers of group 2
>> testdata(:, 13) = randn(20, 1) % random data
>> testdata([4, 11:13], 13) = NaN % add missing values
```

You can copy/paste all code lines to the Command Window.
Everything after the % sign are comments and are not interpreted by Matlab.

Summary

Points to remember

- A semicolon after a command suppresses the output of the created values in the Command Window.
- Matrices can be concatenated horizontally and vertically by means of the normal `[ ]`-notation. Instead of single numerical values, complete matrices are used, e.g. `d = [a; b]`. The to-be-concatenated matrices need to have the same dimensions, that is the equal number of rows when concatenating horizontally, and equal number of columns when concatenating vertically.

Commands

- `horzcat, vertcat`: horizontal or vertical concatenation of matrices
- `cat`: general command to concatenate matrices in rows, columns, planes, etc.
- `repmat`: duplicate a matrix
- `zeros, ones` create a matrix with zeroes or ones
- `rand, randn` generate matrices containing random numbers with equal or normal distribution.
- `fliplr, flipud`: mirrors a matrix left/right or up/down
- `rot90`: rotation by 90 degrees
- `data'`: transposition of matrix `data`, i.e. swapping rows and columns

Mathematical Operators and Functions

In this section, mathematical operators and functions are explained.
Mathematical **operators** are mathematical operation symbols such as:

- addition +, subtraction -, sign, etc.
- multiplication * and division /
- exponential functions such as square ^2
- In this context, the question of priorities of operations and the bracket rules are important as well.

Other mathematical manipulations are realised as **functions**, e.g.

- trigonometrical functions: sine, cosine, tangent, etc.
- descriptive functions such as minimum, maximum, mean values and the like
- basic mathematical functions such as: square root, logarithm, sum, rounding, etc.
- and many more...

The range of functions provided by Matlab is enormous, thus only the most important ones can be introduced in this course.
Later on, two exercises will be provided so that you can practise the use of functions and operators.

The Help System provides more information on all available functions. Call this by entering **elementary math** in the search field of the Help System.

Operators

The Help System provides a good overview over the mathematical operators and how to spell them. To display this information, you can either enter **arithmetic operators** in the Help Navigator, or you can type `docsearch 'arithmetic operators'` in the Command Window (the inverted commas are necessary because the search term includes a space).

When performing calculations, certain peculiarities of Matlab become relevant. As long as you exclusively work with *scalars* [4], everything works as you would expect from a pocket calculator. However, operations with *matrices* (and *vectors* [5]) follow other rules in some cases (linear algebra).

Please try to understand the following examples:

>> magic(3) + ones(3)	Addition is always performed element by element. The two matrices
>> a(1:3, 1:3) = 5;	have to be of the same size.
>> a * magic(3)	Multiplication of two matrices is NOT performed element by element
>> a .* magic(3)	(linear algebra, usually not important for applications in our field).
>> magic(3) * 5	To perform element-by-element multiplication, you can use the operator `.*`
	(array multiplication, element-by-element product)
	To multiply each element of a matrix with 5, you can just multiply it with a scalar
	(5 in this case). This method is called **«scalar expansion»**.

If you are unsure about the function of an operator, you can just test it in command line mode before inserting the statement into your program.

The sequence of how different operators are evaluated follows the general precedence mathematical rules (e.g. multiplication/division before addidion/subtraction); as usual, brackets are used if a different sequence is necessary.

You can find help on this topic by using the key word **operator precedence** in the Help System.

Functions

Matlab provides a plethora of pre-defined mathematical functions (e.g. sine, mean, square root, etc.)

In the Help System, you find the relevant information in **Contents** under **Function reference > Mathematics > Elementary math**

Examples for mathematical functions:

```
>> data = magic(10);      calculates the sum in columns
>> sum(data)              for the overall sum, two sum statements can be nested
>> sum(sum(data))         minimum and maximum per column
>> min(data)              standard deviation in columns
>> max(data)              functions such as the sine separately calculate the results for each element
>> std(data)              rounding function
>> sin(data)              another «function»: answer on all possible questions...
>> round(pi)
>> why
```

Exercise 2: Implementing a Mathematical Formula

Try to implement the formula of the *Gaussian bell shape* [6].

$$y = \frac{1}{s\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-m}{s}\right)^2}$$

m is the mean value or the expected (peak) value Erwartungswert (peak of the curve)
s is the standard deviation ("width" of the shape)
x is the independent value
e is the exponential function, in Matlab to be executed with `exp`. The square root function is called `sqrt`

>> m = 3; s = 5; x = 2;	definition of input values
>> y = ...	How is the formula to be written correctly? The result should be **0.0782**

Sample solution

```
>> m = 3; s = 5; x = 2;
>> y = 1/(s*sqrt(2*pi))*exp(-1/2*((x-m)/s)^2)
```

Exercise 3: Validation of a Magic Square

Check the correctness of a 7x7 magic square: The sum of all rows, columns, and the two diagonals have to be equal.

```
>> m = magic(7);
>> ...
```

Sample solution

```
>> m = magic(7);
>> sum(m) % sum of columns
>> sum(m') % transpose m to obtain row sums
>> sum(diag(m)) % calculate the sum of the main diagonal
>> sum(diag(fliplr(m))) % the other diagonal
```

Summary

Points to remember

- When evaluating consecutive mathematical operations, the sequence of how they are performed follows the general mathematical rules (e.g. multiplication/division before addition/subtraction). As usual, brackets are needed if a different sequence is necessary.
- You can find an overview of all operators by typing `docsearch 'arithmetic operators'`
- Addition and subtraction of whole matrices is performed element-by-element.
- Element-by-element mulitplication and division of whole matrices requires a special syntax: `a .* b` and `a ./ b`.
  Otherwise, Matlab will perform a matrix multiplication or division according to the rules of linear algebra, which are different (and which are seldom relevant for applications in psychology).
- Help System: You find the function overview in «Contents» **Function reference** > **Mathematics** > **Elementary math**.

Commands

- `sum`: calculates the sum
- `min, max`: minimum and maximum
- `std`: standard deviation
- `sin, cos, tan` etc.: trigonometric functions
- `round`: rounding

Self-test Questions

To test your understanding, please try to answer the questions below.

User interface

Which command is used to clear the command window?

❍ clear
❍ clc
❍ home
❍ clrcmd

What is NOT possible to do in the Workspace window?

❍ Remove variables from the Workspace
❍ Change the contents of variables
❍ Read data from external sources
❍ perform calculations

How can you select the folders in which Matlab searches for functions (M-files)?

❍ In the Current Directory browser
❍ with cd (change directory)
❍ By using the menu item File > Set Path

Help System

In the Help System, try to find the answer to the following question:

The function to draw a three-dimensional scatter plot is ___________ .

Matrices

Which statement will generate the matrix a shown below?

1	5	8	3
6	6	7	1
0	1	0	8

❍ `a = [ 1 5 8 3, 6 6 7 1, 0 1 0 8]`
❍ `a = [ 1 5 8 3; 6 6 7 1; 0 1 0 8]`
❍ `a = ( 1 5 8 3; 6 6 7 1; 0 1 0 8)`
❍ `a = [ 1 6 0; 5 6 1; 8 7 0; 3 1 8 ]`

Which command sequence leads to the pictured matrix being printed in the Command Window?

0	8	3	8	5
3	5	5	3	5

❍ `a=[0 8 3;3 5 5]; b=a'; b(1,:)=NaN; [a b]`
❍ `a=[0 8 3;3 5 5]; b=transpond(a); b(1,:)=[]; [a b]`

❍ `a=[0 8 3;3 5 5]; b=a'; b(1,:)=[]; [a b]`
❍ `a=[0 8 3;3 5 5]; b=a'; b=b(2:3,:); [a; b]`

Mathematical operations
Which code represents the shown formula?

$$y = 2\frac{e^{-\frac{t}{tau}}}{\sqrt{(x+1)(x-1)}}$$

❍ `y = 2*exp(-t/tau)/sqrt(x+1*x-1)`
❍ `y = 2*exp(-t/tau)/sqrt((x+1)*(x-1))`
❍ `y = 2(exp(-t/tau)/sqrt((x+1)*(x-1)))`
❍ `y = 2*e^(-t/tau)/sqrt((x+1)*(x-1))`

Glossary

EEG:

Electro-encephalography: registration of the electrical activity of the brain

fMRI:

functional magnetic resonance imaging

Gaussian bell shape:

density function of normally distributed data

matrix:

The most important way to represent data in Matlab. In the field of mathematics, a matrix is a table of numbers or other values. Matrices differ from ordinary tables in that they can be used for calculations.

Usually, the expression «matrix» refers to two- or higher-dimensional matrices (cf. scalar, vector).

scalar:

«null-dimensional» matrix, i.e. only one element

vector:

one-dimensional matrix, that is only one row or one column

Lesson 2: Programming in Matlab

Basically, a **program** is a sequence of statements that will be executed when running the program. The succession of executed statements is determined by **program control structures** such as loops or conditional branchings. This sequence of statements is stored in a file and can thus be called again at any time.

Learning Objectives

- You know the most important functions of the Editor.
- You understand the difference between scripts and functions.
- You can write simple scripts and functions.
- You know the most important loop and branching structures.
- You know how to use character strings.

M-Files: Scripts and Functions

All programs written in Matlab are stored as so-called **M-files** with the file name extension **.m**. This is also true for most functions that are included in Matlab's basic function library as well as additional toolboxes. In the following sections, you will learn how to write and execute such programs.
There are two kinds of program or M-files: **scripts** and **functions**.

Scripts
A script contains a sequence of statements that works in the same way as if you entered these statements one after the other in the Command Window. Accordingly, scripts can access all variables in the Workspace, and variables created by scripts are stored in the Workspace.

Functions
A function is characterised by the fact that (in most cases) values are passed to it to be processed, and the function returns the calculated results to the «caller» (i.e. to the statement entered in the Command Window, or another script or function). Thus, this is the same as - for instance - with standard functions, e.g. computation of a mean value with `a = mean(b)`. The data to be processed are passed to the function in matrix `b`, and the calculated mean value is returned in variable `a`.

The Editor/Debugger
As a prerequisite for being able to write scripts and functions, you have to get to know the proper tool, that is the Editor. It is designed to edit M-files. This will be explained in the next section.
At the same time, this program serves as the so-called «Debugger», that is the aid to find errors (*bugs*) in a program and fix them (referred to as *debugging*). This will be introduced later in Lesson 3, Section **Applications: External Files**.

The Editor/Debugger

To write programs, the built-in **Editor** is used. The Editor also serves as **Debugger**, i.e. the tool to support tracking down errors.

The Debugger will be demonstrated in Lesson 3 in the **Example: Data Import**.

The editor can be called in different ways:

- By typing `edit` or `edit function name` (or script name)
- by selecting a function or script name in the Command Window and choosing **open** via the context menu (right mouse click, or ctrl-click on a Mac with single-button mouse)
- or by doing the same in the Directory Browser.

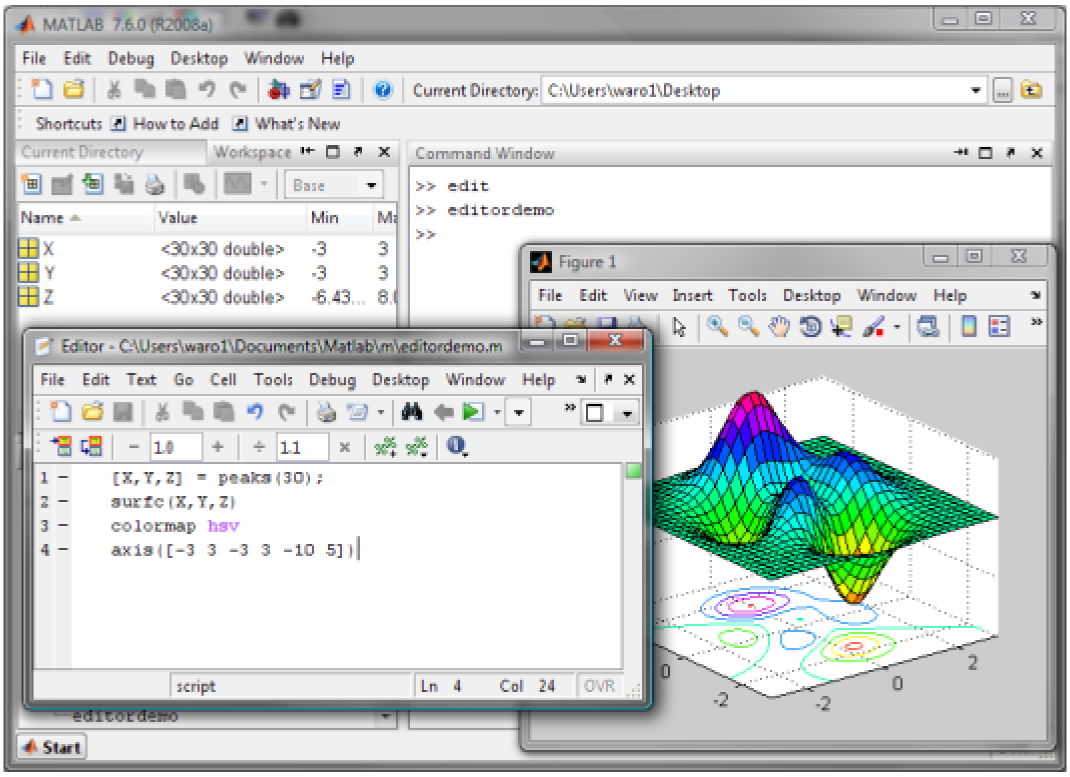
Please note: In some older Matlab versions, if you double click an M-file (that is, a Matlab program file with the file name extension .m) in the Windows file manager, the Matlab Editor is activated as well, but **without the Debugging features**.

Demonstration of the editor: Please perform the steps described below to see how programs are written in the editor.

- Start the editor by typing `edit` in the Command Window. While opening, the editor creates an empty M-file.
- Copy-and-paste the following statements to the Editor window.

```
[X,Y,Z] = peaks(30);
surfc(X,Y,Z)
colormap hsv
axis([-3 3 -3 3 -10 5])
```

- Save the program by selecting **File > Save as...**; name it **editordemo.m**. You should save it to the location you have defined earlier as the path for your M-files (see **Configuration of Matlab**).
- In the Command Window, type `editordemo`; now the program is executed.


MATLAB 7.6.0 (R2008a)
File Edit Debug Desktop Window Help
Shortcuts How to Add What's New
Current Directory Workspace
Base
Command Window
Figure 1
File Edit View Insert Tools Desktop Window Help
Editor - C:\Users\waro1\Documents\Matlab\m\editordemo.m
File Edit Text Go Cell Tools Debug Desktop Window Help
1.0
11
Start
0
-2
-2
0
2

Current Directory: C:\Users\waro1\Desktop

Name	Value	Min	Ma
X	<30x30 double>	-3	3
Y	<30x30 double>	-3	3
Z	<30x30 double>	-6.43...	8.(

```
>> edit
>> editordemo
>>
```

```
[X,Y,Z] = peaks(30);
surfc(X,Y,Z)
colormap hsv
axis([-3 3 -3 3 -10 5])
```

script   Ln 4   Col 24   OVR

That's how your screen should look like after this example

Script Programming

A script is a file containing a sequence of statements. When the script is called, Matlab executes these statements sequentially, in the same way as if you had «manually» entered these statements one after the other in the Command Window.

> Scripts can access all variables in the workspace, and accordingly, the variables generated by scripts are stored in the workspace.

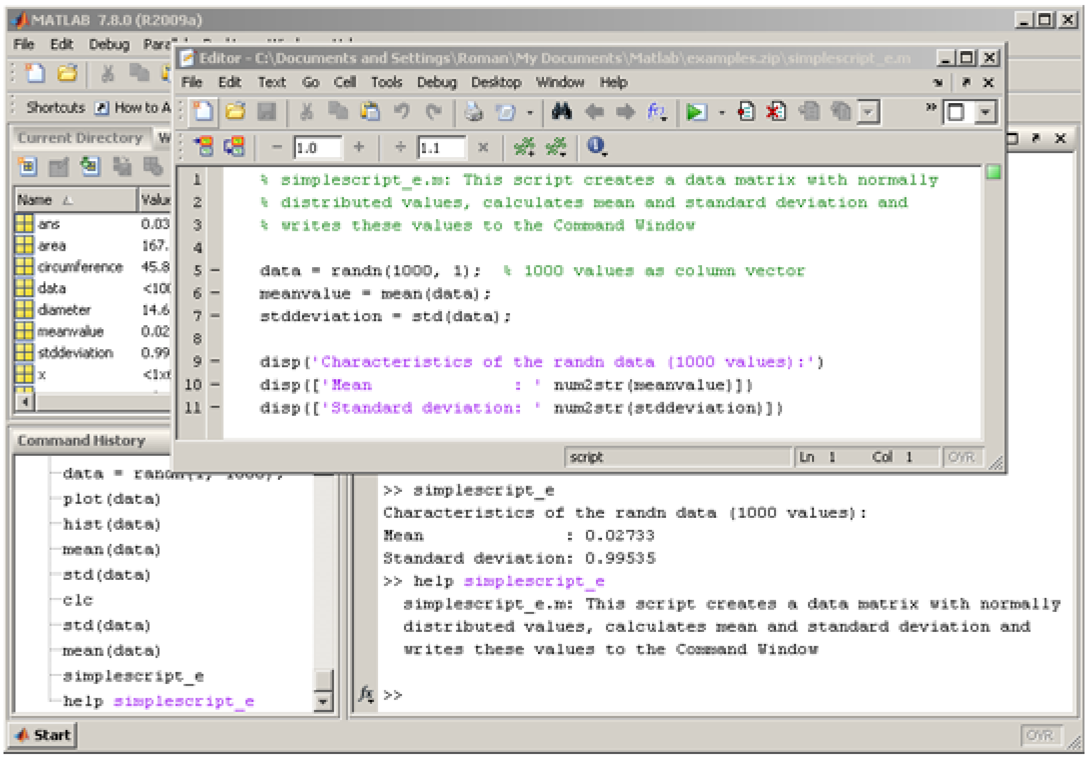
Structure of a Script

- **Program lines:** Usually, for better readability, only one statement is written on a single program line. A **semicolon** after the statement prevents that the calculated values are printed to the Command Window.
- However, it is possible to write more than one statement on a line. In this case, the statements are separated by a **comma**, or by a **semicolon** which separates the statements and prevents output to the Command Window at the same time.
- **Comments** are marked by a **percent sign %**. Everything on a line that follows this sign will not be interpreted by Matlab.
- **Header lines/Help text:** As shown in the figure below, a well-formed script begins with some comment lines describing the function of the script.
  These lines (up to the first empty line) are printed if you call the help information for this script by typing `help script name` in the Command Window.
- **Empty lines:** Any number of empty lines can be inserted in order to visually structure the program. They will not influence the function of the program at all.

Writing scripts
As demonstrated above, writing a script subsumes several steps:

- Starting the Editor: an empty file is opened.
- Writing the statements in the Editor window
- Saving the script as an M-file with a new name

Executing a script
A script is executed by typing the script name (i.e. the file name without the extension .m) in the Command Window, or by calling it from within another script or a function.



Editor - C:\Documents and Settings\Roman\My Documents\Matlab\examples.zip\simplescript_e.m

```matlab
% simplescript_e.m: This script creates a data matrix with normally
% distributed values, calculates mean and standard deviation and
% writes these values to the Command Window

data = randn(1000, 1);  % 1000 values as column vector
meanvalue = mean(data);
stddeviation = std(data);

disp('Characteristics of the randn data (1000 values):')
disp(['Mean              : ' num2str(meanvalue)])
disp(['Standard deviation: ' num2str(stddeviation)])
```

Name	Value
ans	0.03
area	167.
circumference	45.8
data	<10(
diameter	14.6
meanvalue	0.02
stddeviation	0.99
x	<1x(

Command History

```
data = randn(1, 1000);
plot(data)
hist(data)
mean(data)
std(data)
clc
std(data)
mean(data)
simplescript_e
help simplescript_e
```

```
>> simplescript_e
Characteristics of the randn data (1000 values):
Mean              : 0.02733
Standard deviation: 0.99535
>> help simplescript_e
  simplescript_e.m: This script creates a data matrix with normally
  distributed values, calculates mean and standard deviation and
  writes these values to the Command Window

>>
```

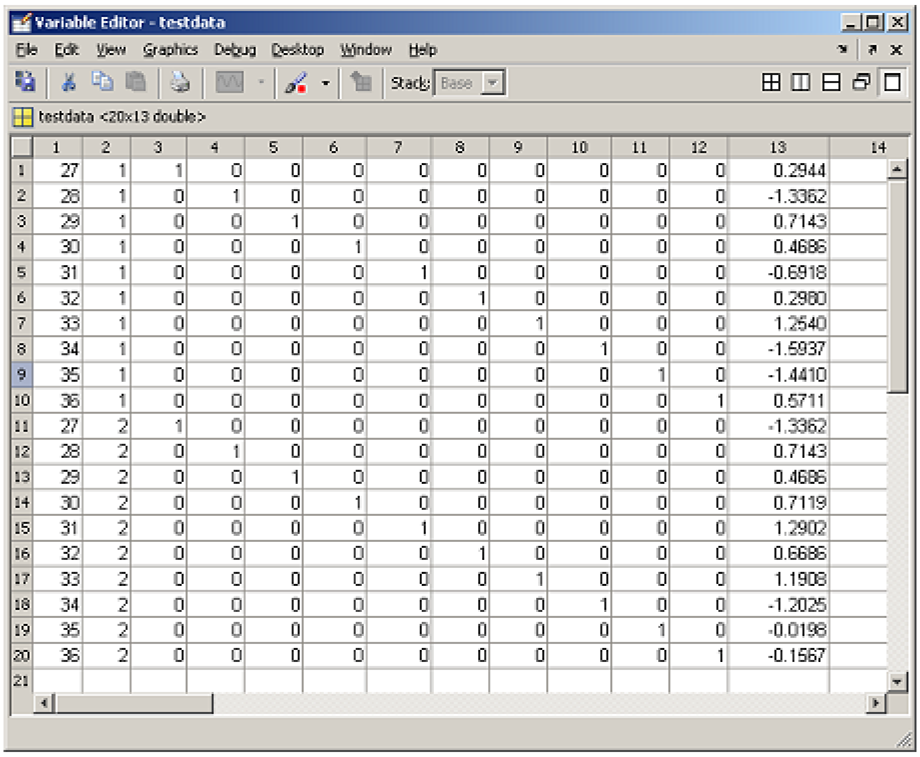
A simple script. In the Command Window, you can see the output when calling the script, and its help information.

Exercise 4: Programming a Script

The goal of this exercise is to write a *script* [1] containing the statments you used in **Exercise 1**. This script can then be executed repeatedly. You have seen how this is done in the section above.

Write a script that creates the data matrix from **Exercise 1** (but without the `NaN` values), calculates the mean and standard deviation of the data in column 13 per group, and prints these values to the Command Window.

Column 1: subject number
Column 2: 1st or 2nd run of the experiment
Column 3-12: mark the different experimental conditions
Column 13: normally distributed random data (they do not need to be identical with the values in the figure).


Variable Editor - testdata
File Edit View Graphics Debug Desktop Window Help
Stack: Base
testdata <20x13 double>

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	27	1	1	0	0	0	0	0	0	0	0	0	0.2944	
2	28	1	0	1	0	0	0	0	0	0	0	0	-1.3362	
3	29	1	0	0	1	0	0	0	0	0	0	0	0.7143	
4	30	1	0	0	0	1	0	0	0	0	0	0	0.4686	
5	31	1	0	0	0	0	1	0	0	0	0	0	-0.6918	
6	32	1	0	0	0	0	0	1	0	0	0	0	0.2980	
7	33	1	0	0	0	0	0	0	1	0	0	0	1.2540	
8	34	1	0	0	0	0	0	0	0	1	0	0	-1.5937	
9	35	1	0	0	0	0	0	0	0	0	1	0	-1.4410	
10	36	1	0	0	0	0	0	0	0	0	0	1	0.5711	
11	27	2	1	0	0	0	0	0	0	0	0	0	-1.3362	
12	28	2	0	1	0	0	0	0	0	0	0	0	0.7143	
13	29	2	0	0	1	0	0	0	0	0	0	0	0.4686	
14	30	2	0	0	0	1	0	0	0	0	0	0	0.7119	
15	31	2	0	0	0	0	1	0	0	0	0	0	1.2902	
16	32	2	0	0	0	0	0	1	0	0	0	0	0.6686	
17	33	2	0	0	0	0	0	0	1	0	0	0	1.1908	
18	34	2	0	0	0	0	0	0	0	1	0	0	-1.2025	
19	35	2	0	0	0	0	0	0	0	0	1	0	-0.0198	
20	36	2	0	0	0	0	0	0	0	0	0	1	-0.1567	
21														

Data matrix to be created

Save the script, for instance with the name **exercise4.m**, in your working directory (the one directory you created in **Section Preparations/Materials**). For the creation of the data matrix, you can either re-use your solution of Exercise 1, or copy the code from the box below.

Hints: To output data to the Command Window, you can just write the variable name as a statement (as in Command Line mode), omit the semicolon after a calculation statement, or you can use the function `disp(variable name)`. Other necessary functions: `mean, std`

```
% Exercise 4: This script creates the data matrix from exercise 1,
% calculates the mean and standard deviation per group, and
```

[1] A script is a sequence of statements that is stored in a file (M-file). These statements are processed in the same way as if they had been entered in the Command Window.

```
% outputs these values to the command window.
% generate the matrix
testdata = (27:36)';
testdata = [testdata ones(1, 10)' eye(10)];
testdata = repmat(testdata, 2, 1);
testdata(11:20, 2) = 2;
testdata(:,13) = randn(20,1);
% calculations and output
...
```

Sample solution (opens in a new window)

The solution can also be found in the M-file **exercise4_sol.m** (from the file examples.zip you downloaded before).

Function Programming

Function *M-files* [2] you have written yourself are applied in the same way as the functions provided by Matlab, e.g. the sine function `sin`. That is, they are called with their name (the file name of the M-file) and the necessary arguments, for example
y = sin(x)
A function is characterised by the fact that (in the majority of cases) values to be processed are passed to them, and that the function returns the results to the «caller» (i.e. the Command Window, or another script or function).
The main advantage of a function is its **reusability**. Tasks that have to be done in different applications only need to be written once. If there are changes, they only have to be made once in the code of the function, and the change becomes relevant for all calling programs at once.
Thus, functions allow for more efficient programming, and the programs themselves become more compact and thus better readable and easier to maintain.

As to their access to data (variables), functions are «encapsulated», i.e. they can only access those variables that have been passed to them in the function call. The Workspace as you know it is «invisible» to the function. While a function is being executed, it obtains a temporary, private Workspace containing the variables passed and to be returned, as well as temporary variables created by the function itself to perform the internal calculations. When the function has finished, this temporary Workspace is deleted.

Structure of a function
Formally, a function only differs from a script in its first line. It contains the so-called **function definition**. This definition specifies:

- the **function name**. It has to be identical with the file name under which the function is saved (i.e. the M-file name).
- the **input parameters**: They specify what values can be passed to the function.
- the **output (or return) parameters**: They specify the values in which the function returns the results.

Example 1: M-file gauss1.m
A simple function with one input and one output parameter.
Please open this function in the Editor. It should be available in your directory for M-files, and originates from the file **examples.zip**.

```
function y = gauss1(x)
% Calculates the value of the gaussian bell shape, with a fixed mean value
% and standard deviation
% another comment (will NOT be displayed when typing help gauss1)
s = 5; % sigma
m = 3; % mean value
y = 1/(s*sqrt(2*pi))*exp(-1/2*((x-m)/s)^2);
```

[2] A file with the extension .m, containing a sequence of Matlab statements. It can represent a script or a function.

Explanations

- For more information on the gaussian bell shape see **Exercise 2** in Lesson 1.
- Line 1: Here, the function is defined with its input and output paramters. `gauss1` is the function name and has to be identical to the M-file name. `x` stands for the value that is passed to the function. `y` is the variable that is used to return the calculated result to the caller.
- Line 2 and 3 contain a short function description as comments. It is strongly advised to add descriptions and explanations to all functions, in order to make sure that you know what the function does when looking at it half a year later, and that somebody else has a better chance to understand it. All comments up to the first blank line are displayed when `help gauss1` is typed in the Command Window.

Using the function:

```
>> result = gauss1(2)
```

Example 2: M-file gauss2.m
A function with several input parameters.

```
function y = gauss2(x, m, s)
% Calculates the value of the gaussian bell shape with variable mean value
% and standard deviation
y = 1/(s*sqrt(2*pi))*exp(-1/2*((x-m)/s)^2);
```

Explanations: Now, `m` and `s` are variable as well, and have to be passed to the function during the call.

Using the function:

```
>> result = gauss2(2, 3, 5)
```

Example 3: M-file stats.m
A function with more than one output parameter. They are specified in **square brackets**.

This is the only case in which the square brackets `[ ]` are not used to describe a matrix, but just a list of output parameters.

```
% Calculates statistical values for a row vector
%
% INPUT : M_in - input vector containing the data
% OUTPUT: meanval - mean value
% stddev - standard deviation
% minimum - minimal value
% maximum - maximal value
function [meanval, stddev, minimum, maximum] = stats(M_in)
% Remark: The function definition can be in the first line, but also be written
% after the help text lines.
meanval = mean(M_in);
stddef = std(M_in);
minimum = min(M_in);
```

```
maximum = max(M_in);
```

Using the function:

>> M_in = randn(1, 300)*100+100	create a row vector with random numbers
>> [m, s, mnm, mxm] = stats(M_in)	get back all 4 output parameters
>> stats(M_in) >> a = stats(M_in) >> [s1, s2, s3] = stats(M_in)	But it is also possible not to fetch all 4 return parameters - in this case, the variables are filled "from front to back".

Special Cases

Special case 1: Function without parameters

If a function is to be written that does not need any input values (e.g. a function that generates special random numbers), or does not return any results (but creates a graphical output, for instance), the function definitions look like this:

>> function numbers = randomx()	No input parameters: just leave the brackets empty
>> function [] = draw_graph(M_in)	No output parameters: two possibilities
>> function draw_graph(M_in)	

Special case 2: Variable number of parameters

Occasionally, it is necessary to have a function that does not strictly specify how many input or output parameters are used.

function string = strxcat(varargin)	variable number of input paramters
varargout = readsomenumbers(fid)	variable number of return parameters

The variables `varargin` (variable number of arguments IN) and `varargout` (variable number of arguments OUT) are **Cell arrays**, a special data type that can contain various kinds of data. You will learn more about *cell arrays* [3] in **Lesson 4, chapter Cell Arrays**.

As an example for a function with a variable number of input parameters, the function `strxcat` (also mentioned **here**) is presented:

```
function string = strxcat(varargin)
% A variable number of strings or numbers are concatenated into the output string.
%
% INPUT : n strings or numbers
% OUTPUT: string
string = [];
nbarg = length(varargin);
if nbarg == 0
return
end
for arg_i = 1:nbarg
substr = varargin{arg_i}(:)';
string = [string num2str(substr)];
end
```

Explanations: First, the number of *strings* [4] passed as input is determined with the `length` function and stored as `nbarg`.

Then, a loop is executed `nbarg` times, once for each input string. In the loop, the

[3] In every field of a Cell Array, different data types can be stored, e.g. matrices of different size.

[4] Character string. A vector or a matrix whose elements are interpreted as ASCII coded characters.

`arg_i`-th element is extracted, converted from number into string if necessary, and concatenated with the previous strings.

Summary

Points to remember

- A **script** contains a sequence of statements that are processed in the same way as if they had been entered one after the other in the Command Window.
- A script can be called from the Command Window by typing its file name (without .m). In the same way, a script can be called from within another script or function.
- Scripts can access all variables in the Workspace, and the variables created in the script are stored in the Workspace.
- A **function** differes from a script in that (in most cases) values are passed to the function to be processed, and the results can be passed back to the «caller».
- Contrary to scripts, functions are «encapsulated» with regard to data access, i.e. they cannot access the variables in the general Workspace, but only those variables that have been passed in the function call.
  During execution, a function obtains a temporary, private Workspace, containig the variables passed as input and return parameters as well as variables defined within the function itself.
- A semicolon at the end of a code line prevents the output of calculated or created values in the Command Window.

Commands

- `function [return parameters] = function name(variables for input parameters)`
  definition of a function with input and return parameters
- `disp()`: output of values to the Command Window

Program Control Structures: Loops

It is often necessary to repeat a sequence of statements. This is achieved by using **loops**. There are two kinds of loops:

- the `for`-loop: It is used if it is pre-determined how many times the statements have to be repeated.
- the `while`-loop: It allows for making the number of repetitions dependent on a comparison within the loop.

The For Loop

> If the number of necessary repetitions is known before the loop is started, the for loop is applied.

This is the schematic syntax of the for loop:

```
for indexvar = range
statement 1
...
statement n
end
```

`indexvar` - the so-called loop index. With every pass through the loop, it is set to the next value of `range`. It can be used like any other variable within the loop. Any valid variable name is allowed.
`range` - row vector or matrix. It defines which values are assigned to `indexvar`.

For loop examples

Please try to understand the way how for loops work by studying the examples below.

Simple for loop:

>> for index = 1:30	Matlab does not do anything until it hits `end`
>> facts = factor(index);	calculate the prime factors of the loop index
>> disp(facts)	display them
>> sumvalue = sum(facts);	further statements
>> end	with this, the loop is closed, and will be executed

Basically, for the loop index, all notations of vectors are allowed:

>> for i = pi:-.3:-pi, i, end	any sequence of numbers
>> for i = [1 4 7 11 13 17 3 5], i, end	enumeration
>> for i = rand(1,10) * 20, i, end	vector created by a function, but **only** row vectors

If a matrix is given as `range`, the loop index obtains a whole column of `range` at each pass. That is, the loop is executed as many times as there are columns in the matrix.

```
>> for i = magic(7), i, end
```

The While Loop

The while loop is applied when the number of passes is dependent on a comparison made each time the loop is executed, usually at the begin of the loop.

The syntax of the while loop in pseudo code:

```
while condition
statement 1
...
statement n
end
```

The loop is executed as long as the `condition` results in `true` (i.e. a value > 0). The condition is always evaluated at the begin of the loop. The first time the condition evaluates as `false` (value = 0), the loop is left. Program execution then continues with the first statement after the loop (i.e. the corresponding `end` statement). You will find detailed information on how to specify comparisons in the section **Comparison Operators and Logical Operations**.

A simple while loop:

```
>> index = 1;                execute the loop as long as index is smaller as or equal to 30
>> while index <= 30         print prime factors
>> disp(factor(index))       increment index
>> index = index + 1;
>> end
```

Exit Loops with the break Statement

It is possible to prematurely exit for- and while loops by means of the `break` statement, even if it has not «finished». That is, the `break` statement causes the program execution to continue after the `end` statement that closes the loop.
With this statement, for instance, a while loop can be written whose exit comparison is done at the end or anywhere within the loop instead of at the beginning. Or you can terminate a for loop before the loop index has reached its last value.

Example 1: for loop with premature exit

```
>> for index = 1:30                              already leave the loop when the index is 20
>> disp(sum(factor(index)))
>> if index == 20, break, end
>> end
```

Example 2: while loop with `break`

```
index = 1;
while true
disp(factor(index))
if index > 30
break
end
index = index + 1;
end
```

Explanations
`true` (or the value 1) in the `while` statement means: the comparison result is true, in this example: ALWAYS (counterpart: `false` or 0).

Exercise 5: Gaussian function for a vector

Rewrite the Gaussian function version 2 (M-file **gauss2.m**) by means of a loop in order to enable it to calculate the values for a row vector and return the results as a vector. Save the function as **gauss3.m**). Start out with these lines:

```
function Moutput = gauss3(Minput, m, s)
% Calculates the values of the gaussian bell shape for the row vector
% Minput, with the mean value m and standard deviation s
...
```

Afterwards call the function with values from -5 to 5 (with intervals of 0.1) and graphically display the values by means of `plot`

Sample solution (opens in a new window)

The solution can also be found in the M-file **gauss3_sol.m**

Vectorisation of Loops

Thanks to the data representation method as matrices, it is often possible to process complete data sets much more efficiently than as it would be feasible with loops.

Please have a look at the following example:

The task is to compute the sine of 1001 values from 0 to 10 and store it in row vector y. Using the means you know up to now, one possibility to solve this task is:

```
>> i = 0;
>> for t = 0:.01:10
>> i = i + 1;
>> y(i) = sin(t);
>> end
```

The «vectorised» version is simpler and executes faster:

```
>> t = 0:.01:10;
>> y = sin(t);
```

Or in an even more compact form:

```
>> y = sin(0:.01:10);
```

Summary

Points to remember

- The **for loop** is used whenever it is predetermined how many times a sequence of statements has to be executed.
- The **while loop** is applied when the number of reiterations has to be dependent on a condition.

Commands

- for index=1:n
  (some statements)
  end
  Repetition loop. Instead of 1:n, any vector can be used.
- while (condition)
  (some statements)
  end
  Loop with conditional exit. The statements are executed as long as the condition is fulfilled, i.e. results in true (value > 0). The comparison is always evaluated at the beginning of the loop.
- `break`: allows to exit for and while loops at any time.

Program Control Structures: Conditional Branching

In almost every program it is necessary to process some statements only under certain conditions. To realise this, so-called **conditional branchings** are used.
Matlab offers two constructs to achieve this:

- the **if-elseif-else** structure, and
- the **switch-case** structure.

The branching structure if-elseif-else

You can see how this structure works in the examples below.

Example 1: if-else structure with two cases

Load the M-file **example_if.m** into the Editor. You can execute the script by hitting the **F5** key in the Editor window, or by typing `example_if` in the Command window.

```
% Example 1 for if-else
a = rand(20) * 100 - 20; % generate random numbers
for i = 1:length(a)
if a(i) < 0
disp([num2str(a(i)) ': number is below zero'])
else
disp([num2str(a(i)) ': number is above zero'])
end
end
```

Explanations: The composed statement after `disp` displays the current entsprechende number together with the text. The function `num2str` will be explained later (see **here**).

Example 2: if-else structure with more than two cases

Again, you can load the appropriate M-file **example_if_elseif.m** into the Editor.

```
% Example 2 for if-elseif-else
a = rand(20) * 100 - 20;
for i = 1:length(a)
if a(i) < 0
disp([num2str(a(i)) ': number is below zero'])
elseif a(i) >= 0 && a(i) <= 20 % is the value >= 0 AND <= 20?
disp([num2str(a(i)) ': small number'])
else
disp([num2str(a(i)) ': large number'])
end
end
```

Explanations: Obviously, it is possible to realise any number of subsequent elseif comparisons.

The switch-case structure

If, in the course of a if-elseif-else structure, a great number of single comparisons with **the same variable** have to be done, the switch-case structure is the better solution:
The schematic syntax is:

```
switch indicator
case expr
statement, ..., statement
case {expr1, expr2, expr3, ...}
statement, ..., statement
otherwise
statement, ..., statement
end
```

`indicator` is the variable that contains the value on which the conditional branching is based. It can be a scalar or a *string*.
The `case` statement compares the `indicator` with `expr`. If they are the same, the following statements within the `case` block are executed. Afterwards, the program is continued after the `end` statement.
`expr` contains the comparison values (scalars or strings). Single values as well as *Cell Arrays* with several values can be used.
After `otherwise`, those statements follow that have to be executed in case none of the previous conditions have been met.

Example: Replacing a multiple if comparison by a switch-case structure
Please study the following implementation of a multiple comparison with if-elseif-else (the function `strcmp` compares two strings):

```
if strcmp(designator,'date')
% statements...
elseif strcmp(designator,'start')
% statements...
elseif strcmp(designator,'endtrial')
% statements...
elseif strcmp(designator,'fix') || strcmp(designator,'sacc') || strcmp(designator,'blink')
% statements...
else
% all remaining cases
end
```

The following implementation is more elegant and easier to understand:

```
switch designator
case 'date'
% statements...
case 'starttrial'
% statements...
case 'endtrial'
% statements...
case {'fix', 'sacc', 'blink'}
```

```
% statements...
otherwise
% all remaining cases
end
```

Comparison Operators and Logical Operations

In the two examples above, you have already seen various comparison operators such as «smaller than» oder «larger than or equal as».

Comparison Operators

Below, the basic comparisons are shown:

`A == B`	test for equality: **double** equal sign!
`A ~= B`	not to be mixed up with `=`, that would be a value assignment
`A < B` or `A > B`	test for inequality. The tilde symbol ~ always means a
`A <= B` or `A >= B`	logical inversion, that is, NOT
	test for «smaller» or «larger»
	test for «smaller or equal» or «larger or equal»

As the result, comparison operation either return the value `true` (internally pre-defined as 1, but any value > 0 is interpreted as "true") oder `false` (internally represented as 0, zero).

For an overview, see the Help System with the key word 'relational operators'.

Concatenation of Comparisons

It is possible to combine more than one comparison. To achieve this, two logical operators are available:

- `&&` as AND operation: The expression is only true if condition 1 **and** condition 2 are true.
- `||` as OR operation: Contrary to common language use, it means "A or B **or both**"!

Examples:

>> true	the pre-defined values
>> false	is x smaller than y?
>> x = 77; y = 12;	is x not equal to y?
>> x < y	&& and || are always evaluated last, thus not brackets are necessary
>> x ~= y	v now contains the truth value 'false' or 0
>> x <= 77 && y > 0	the ~ «inverts» the truth value (NOT function)
>> v = x == y	Comparisons of vectors and matrices are evaluated element-by-
>> ~v	element! This only works if both are of the same size.
>> a=1:4; b=[1 3 5 7];	
>> length(a)==4 || ~isempty(b)	
>> a == b	

Such comparisons mostly appear after an `if` statement, as shown in the **previous section**.

```
>> if (x < y || y ~= 0) && (v == true || numel(a) == 4)
>> disp('that is true')
>> else
>> disp('that is wrong')
>> end
```

By the way: If a program line becomes very long so that it does not fit into the editor window any more and thus becomes confusing, it can be split by using `...`:

```
if (x < y || y ~= 0) ...
&& (v == true || numel(a) == 4)
```

Exercise 6: Standardisation Function

Write a function named `standardize`, that z-standardises a maximally two-dimensional matrix of data (that is, it also correctly evaluates vectors, but no scalars, as at least two data points are necessary for these calculations).

The formula to be used is:

$$z = \frac{x - \mu}{\sigma}$$

x – value to be transformed
μ – mean value of the data
σ – standard deviation

The function is expected to return the transformed values in a matrix with the same size as the input matrix, as well as returning the mean value and the standard deviation. Use several for- or while loops to implement this (you might also try to figure out how a «vectorised» solution, i.e. without loops, would be realised).
Before the calculation, the matrix passed to the function has to be checked for validity:

- If it is empty or only contains only one element, the function has to return an empty matrix [ ], the calculated mean value and the standard deviation (which will then be 0 or NaN), as well as a warning text printed to the Command Window.
- If the matrix has more than two dimensions, the function has to exit and print an appropriate error message.

Tips for useful functions (you might have to look them up in the Help System):
`mean, std, reshape, isempty, ndims, size, numel, disp, error, return`

Sample solution (opens in a new window)

You can find the solution in the M-file **standardize_sol.m** as well; the «vectorised» solution is called **standardize_vect.m**

Summary

- Branching structure with multiple comparisons:
  if (condition 1)
  (statements)
  elseif (condition 2)
  (statements)
  else
  (statements)
  end
- Branching structure with case differentiation on a single variable:
  switch indicator
  case (value 1)
  (statements)
  case {value 2, value 3, value 4, ...}
  (statements)
  otherwise
  (statements)
  end

Data Representation 2: Character Strings

Up to now, we have only worked with matrices containing numbers. However, you will often have to process text information, that is «character strings», or short, «strings». This will be the case, for example, if you want to import and process data from an *ASCII* [5] files (e.g. tab-separated data), or if you wish to output results or usage hints for the user in clear text.
To this end, Matlab offers the option to store text information in matrices a well. In this case, every number in a matrix is interpreted as character coded in *ASCII*.

This way to work with text information is often applied, but it is not the best and not the only way to do this. However, it is introduced first for didactical reasons, as it is relatively easy to understand (cf. **Lesson 3: Cell Arrays**).

[5] American Standard Code for Information Interchange, the most common norm to represent alphanumerical characters.

String creation and concatenation

First you need to know how character strings (or briefly «strings») are created. Often, it is also necessary to convert numerical results into a string and chain it with another string.

Please try to follow the examples below to find out how this is done.

>> t1 = 'The quick brown fox jumps '	fixed texts are entered in inverted commas
>> t2 = 'over the lazy dog.'	conversion of a number into a string
>> n = num2str(23.28)	yields `true` if the matrix contains characters
>> ischar(t1)	conversion of a number that is stored in a string
>> m = str2num('3.1415926')	into a scalar
>> t3 = [t1 t2]	concatenation of text as with
>> t4 = strcat(t1, t2)	'normal' vectors
	another way. strcat = **str**ing con**cat**enation

If you want to store several strings in a matrix (that is, in rows), there is a complication: As you know, all rows of a matrix have to contain the same number of elements. Thus, Matlab will fill up rows with shorter texts with blank characters (the function to vertically concatenate strings is called `strvcat` = string vertical concatenation)!

>> z=['four';'five';'nine']	This works fine as all words have four characters
>> z=['four';'five';'nine';'seven']	This does not, as 'seven' is longer
>> z=strvcat('four','five','nine','seven')	correct statement
>> z1 = z(1, :)	address the first row
>> length(z1)	however, it is **five** characters long,
>> z2 = deblank(z1), length(z2)	a space character has been added
	`deblank` removes spaces at the
	end of a string

A helpful function: strxcat

If you want to concatenate several strings and numerical results, this quickly leads to complicated and confusing code. To avoid this, you can use a little function specifically written for that purpose (it is not included in the standard Matlab functions, you can find it with the downloaded examples).

The function `strxcat` is defined as:

```
function string = strxcat(varargin)
```

It concatenates any number of strings and numbers into a single string. Example:

```
>> sumvalue = fix(sum(rand(1, 3000)))/100
>> output = strxcat('The final amount is: ', sumvalue, ' Dollars')
```

`strxcat` is a simple example for a function with a variable number of input parameters (cf. Section **Special Cases**). If you are interested in how this is done, you can have a closer look at it in the editor:

```
edit strxcat
```

Examine and Compare Strings

Find specific contents in strings

Often you will have to process data in differing ways, dependent on the content of a string. Thus you need operations that allow for the examination of string contents.
Two easily confused functions are available: `strfind` and `findstr`

Again, they are best demonstrated with examples. The variables from the preceding examples of this lesson should still be in the workspace.

>> findstr(t3, 'fox')	Finds a string in an other one.
>> findstr(t3, 'cat')	result: starting position(s) of the found matches
>> pos = strfind(t3, ' ')	position of all space characters in pos

Explanations: `findstr` compares two strings and checks whether the shorter one is contained in the longer one.
`strfind` finds a pattern (the second argument) in a string (first argument). You may want to look up these differences in the Help System.

String comparison

To find out whether two strings are identical, you cannot simply use the comparison operators `==` or `~=` you already know; you have to use specific functions to that end.

The basic function is `strcmp` (**str**ing **comp**arison). This function exists in several variants `strcmpi, strncmp, strncmpi`, that not only allow for a simple equality or inequality test, but also for testing only a restricted number of characters, or ignoring uppercase/lowercase differences

To see these functions in action, please try out the following lines. The variables from the preceding examples of this lesson should still linger in the workspace.

>> strcmp(z2, 'one')	comparison
>> E = 'One';	wrong, because the comparison is case-sensitive
>> strcmp(z2, E)	if this is not wanted, use `strcmpi`
>> strcmpi(z2, E)	(i stands for case-**insensitive**)
>> strncmp(t2, 'over it', 4)	only compares the first n characters (this is also possible case-insensitive - `strncmpi`)

Summary

Points to remember

- (Character) Strings are matrices whose elements are interpreted as *ASCII*-coded characters.
- In contrast to variable names, literal texts are written in inverted commas: `a = 'Hello'`.
  Thus, `Hello` is not a variable, but a literal text.

Commands

- `length`: Determines the length of a string (same as with vectors)
- `str2num`: Transformation of a number stored in a string into a scalar
- `num2str`: Conversion of a number into a string
- `ischar`: returns true if the argument is a matrix containing characters
- `strcat`: string concatenation
- `strvcat`: vertical string concatenation
- `deblank`: removal of blank characters from the end of a string
- `findstr, strfind`: finds a string in another one
- `strcmp, strcmpi`: compare strings, with i: case-insensitive
- `strncmp, strncmpi`: compare the first n characters of strings, with i: case-insensitive

Self-test Questions

To test your understanding, please try to answer the questions below.

Programming

Which of the function definition(s) is/are correct? Multiple answers are possible.

- ❏ `function hello()`
- ❏ `function (ciphers, number) = generate(value, modus)`
- ❏ `function [subj, name, age] = getsubject(1, 6)`
- ❏ `function [] = startprocess(grunt)`

Which code lines fulfil the following goal: Output of the numbers 7 to 21 and their squared values in the Command Window? Several answers can be correct.

- ❏ `for i = 7:21, disp(i, i^2), end`
- ❏ `i=7;while i <22, disp(i, i^2), end`
- ❏ `i=6; while true, i=i+1, disp(i^2), if i==21, break, end, end`
- ❏ `for i = 7:28, disp(i, i^2), if i>20, break, end, end`

Which is the result of the statement:

`4 >= sqrt(9) && ~false`

- ❍ 3
- ❍ 0
- ❍ false
- ❍ 1

Which statement has to be inserted to get the result 1 (= true) in variable q?

```
m = 'To err is human, but to really foul things up requires a
 computer.';
q = 
```
__________ (m(11:15), 'Human')

Glossary

ASCII:

American Standard Code for Information Interchange, the most common norm to represent alphanumerical characters.

Cell array:

In every field of a Cell Array, different data types can be stored, e.g. matrices of different size.

Function:

A sequence of statements that is stored in a file (M-file), similar to a script. However, values can be passed to a function, and the function returns results. Functions cannot access the variables in the common Workspace.

M-file:

A file with the extension .m, containing a sequence of Matlab statements. It can represent a script or a function.

Script:

A script is a sequence of statements that is stored in a file (M-file). These statements are processed in the same way as if they had been entered in the Command Window.

String:

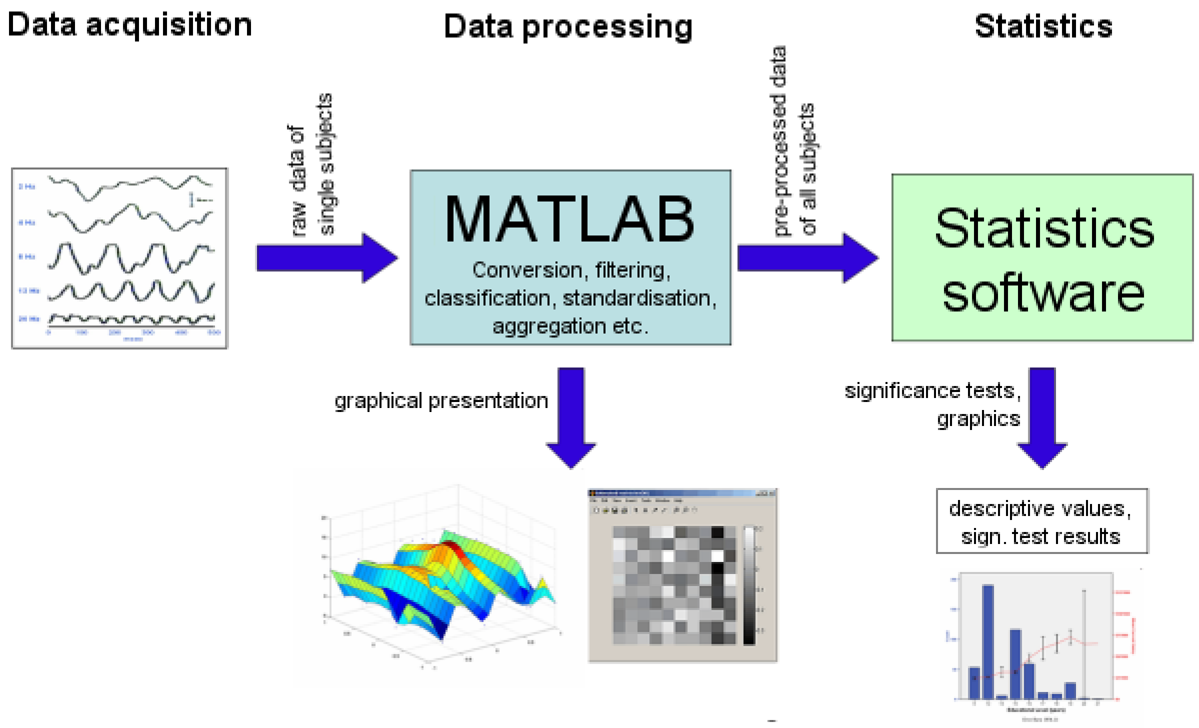
Character string. A vector or a matrix whose elements are interpreted as ASCII coded characters.

Lesson 3: Working with External Files

In this lesson you will learn important functions which are indispensable in psychological research applications.

- Typically, first the raw data files as they are created by other programs (E-Prime, Presentation) or technical systems (MRI, EEG, eye trackers etc.) are read into Matlab.
- These data have to be stored in a suitable way in Matlab, so that they can easily be processed afterwards (summarising, transformation, aggregation, calculus, statistics, plotting etc.)
- Often, the user can influence data processing by direct interaction with the program.
- Finally, the data have to be stored externally in a format that can be processed by other applications such as Excel, SPSS or Statistica.

In this and the following lesson, you can learn to practically realise these steps. To this end, you have the possibility to implement a larger exercise project that is subdivided into several phases.
Again, the schematic procedure of a typical project is shown:


Data acquisition
Data processing
Statistics
raw data of single subjects
MATLAB
Conversion, filtering, classification, standardisation, aggregation etc.
pre-processed data of all subjects
Statistics software
graphical presentation
significance tests, graphics
descriptive values, sign. test results

Learning Objectives

- You are able to import data into Matlab in various ways.
- You get to know further data types.
- You know the most important ways how a program can interact with the user.
- You can save the processed data to an external file.
- You understand how these steps are integrated in an exemplary application.

Data Import

If you want to process and visualise data, you first have to «get them into the program». Nowadays, most systems that are used to acquire psychophysical and behavioural data allow to store the data in simple text files. Matlab is very well equipped to import such data. Basically, there are two ways to do that:

- Comfortable features that can be used to import data «manually» from various standard applications. This option is useful to explore a single data set and graphically display the contents.
- Importing data by a Matlab program. This method is chosen if new sets of data in the same format have to be processed again and again - that is, the typical application in research.

Interactive Data Import

To quickly have a look at a data set, Matlab provides comfortable features. They let you import data from the system clipboard or from text, Excel, xml, or Lotus files without having to write a program for that purpose. Afterwards, the data can be processed and/or visualised.

Import by means of copy-and-paste

This is the most simple, but not the most versatile method: Create a matrix of appropriate size in the workspace, open this matrix in the «Variable Editor» (double click on the variable name in the Workspace), and transfer the data from applications such as Excel, SPSS or similar by means of *copy-and-paste* [1] or **Edit > Copy** and **Edit > Paste**).

Import from files - the «Import Wizard»

Import the file **exp1.xls** (it is contained in the zip file data.zip and should be found in your working directory), by selecting it in the Directory Browser, and choosing «Import data» via the context menu (right click), or via **File > Import Data...**

As you can see, the Import Wizard creates two data sets: a matrix with the numerical data, and a «cell array» (see **here**) containing the text data.

You can also call the Import Wizard from within a program: `uiimport` (also see `import`)

The keyword to find information on that topic in the Help System is **import wizard**.

Import from the system clipboard

You can import data from the system clipboard, i.e. data that has been saved beforehand from another application by copying it to the clipboard. The following options are available; they also call the Import Wizard.

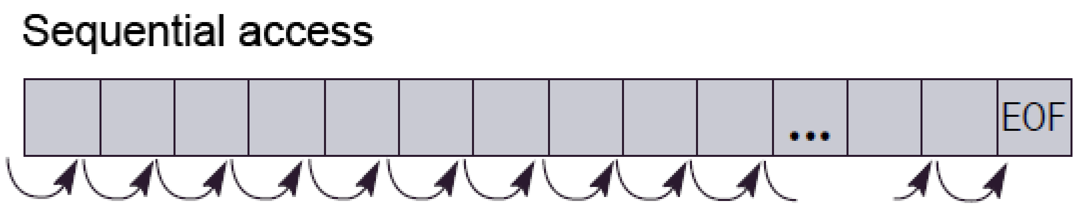
- by using **Edit** > **Paste to Workspace...**
- type `uiimport -pastespecial` in the Command Window
- or simply type `uiimport`, and select **Clipboard** at the item **Select Data Source**

[1] On PC/Windows: CTRL-C and CTRL-V. On Mac OS: cmd-C and cmd-V

Data Input from External Files: Programmed

How does a file work?

Basically in quite a simple way. A file can be seen as a serial sequence of data elements (single alphanumerial characters in the case of data in the *ASCII*[2] format), that starts with the first character, and ends with an end-of-file marker (EOF). When reading the file, a «reading pointer» is moved forward through the file - with every reading operation, it advances accordingly. The pointer could be moved backward as well, but usually this is neither necessary nor useful. When the pointer hits the EOF mark, the procedure is terminated.



Schematic depiction of seqential access to the elements of a file

Schematic procedure of file operations

1. To prepare a file for reading or writing, it first has to be «opened»: function `fopen`.
2. Subsequently, data can be read or written: functions `fgetl, fscanf, fprintf` etc.
3. Finally, the file has to be «closed» again: function `fclose`.

The necessary functions to work with files are best explained by means of an example. We first use the most basic commands - parts of the task could also be solved with more complex statements; to get a basic understanding of the procedure, however, the basic commands seem more suitable.

Basic functions for reading from external files

We assume that we want to work with *ASCII* data (e.g. tab-separated or comma-separated text files such as *.txt, *.tab, or *.csv). For dealing with binary data there are dedicated commands (`fread`).

Most of the described commands have more parameters than shown here - please refer to the Help System.

`fid = fopen(filename, mode)`

Opens a file.

If only a file name is given in parameter `filename`, the file is looked for in the current directory. Alternatively, a complete path can be given, e.g. `d:\daten\jan08\data.txt`

`fid` is the «identification number» of the opened files. It has to be stated in all subsequent operations accessing this file, to make sure that the to-be-accessed file is unequivocally referenced in the case of several open files. If the file cannot be opened, `fid` obtains the value -1.

`mode` is the operation mode; the most important options are `'r'` (read - open the file for reading only) and `'w'` (write - open the file for writing or creation).

`data = fgetl(fid)`

[2] American Standard Code for Information Interchange; the most common norm to represent alphanumerical characters.

Reads a complete line, that is up to the next line break (CR, LF). If the «reading pointer» is already in the middle of the line, only the rest of the line is read. The result of the operation is a *String* [3] (*vector* [4] with characters).

`data = fscanf(fid, format, n)`

Reads `n` data elements in a specified format which is given in `format`. For the format specifications and all other options see the following example and the information in the Help System.

`fclose(fid)`

Closes the file. Only then, other application can fully access it.

`feof(fid)`

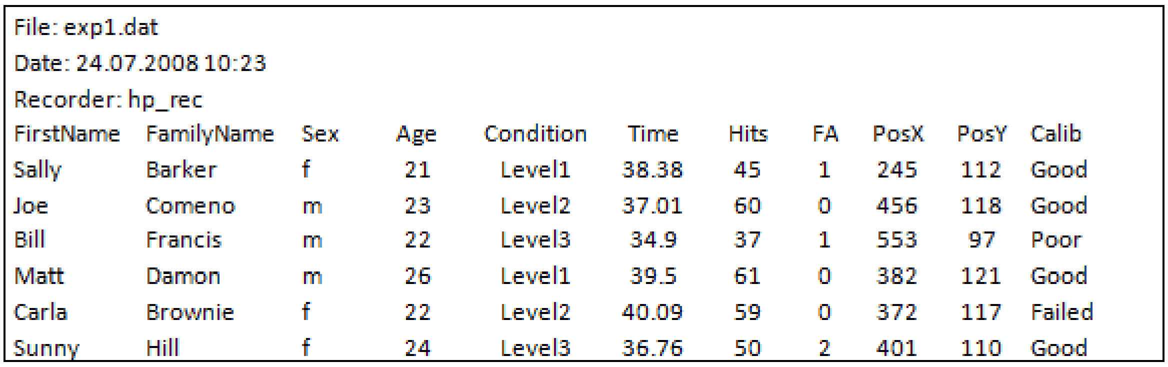
Checks whether the end of the file has been reached (eof = end of file); returns `true` if this is the case, otherwise `false`.

[3] Character string. A vector or a matrix whose elements are interpreted as ASCII coded characters.

[4] one-dimensional matrix, that is only one row or one column

Example: Data Import from an External File

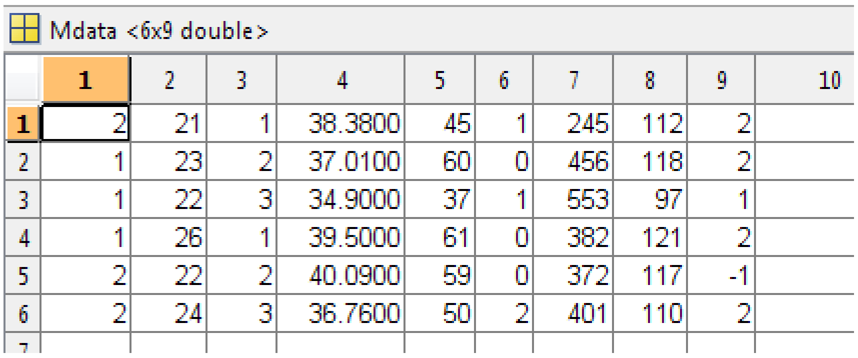
This example represents a typical function as it is used to import data from a text file. At the same time, you can see how the most important functions of the *Debugger* [5] are used.
The data file to be imported looks like this (also see file **exp1.dat**):



```
File: exp1.dat
Date: 24.07.2008 10:23
Recorder: hp_rec
FirstName  FamilyName  Sex   Age   Condition  Time   Hits  FA  PosX  PosY  Calib
Sally      Barker      f     21    Level1     38.38  45    1   245   112   Good
Joe        Comeno      m     23    Level2     37.01  60    0   456   118   Good
Bill       Francis     m     22    Level3     34.9   37    1   553   97    Poor
Matt       Damon       m     26    Level1     39.5   61    0   382   121   Good
Carla      Brownie     f     22    Level2     40.09  59    0   372   117   Failed
Sunny      Hill        f     24    Level3     36.76  50    2   401   110   Good
```

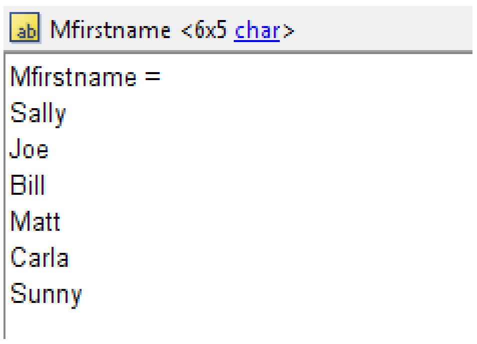
The function is defined as:
function [Mdata, Mfirstname, Mfamilyname] = load_data1('exp1.dat')
The data are returned by the function in the three variables `Mdata`, `Mfirstname` and `Mfamilyname`:

[5] Editor function to assist in finding and correcting errors in a program.



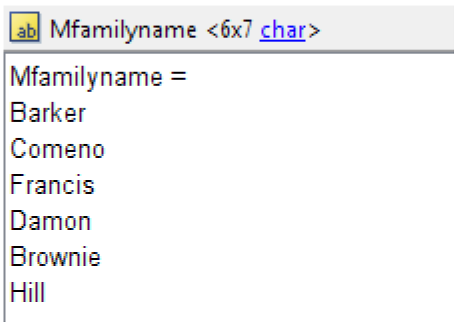
Mdata <6x9 double>

	1	2	3	4	5	6	7	8	9	10
1	2	21	1	38.3800	45	1	245	112	2	
2	1	23	2	37.0100	60	0	456	118	2	
3	1	22	3	34.9000	37	1	553	97	1	
4	1	26	1	39.5000	61	0	382	121	2	
5	2	22	2	40.0900	59	0	372	117	-1	
6	2	24	3	36.7600	50	2	401	110	2	
7										



Mfirstname <6x5 char>

```
Mfirstname =
Sally
Joe
Bill
Matt
Carla
Sunny
```



Mfamilyname <6x7 char>

```
Mfamilyname =
Barker
Comeno
Francis
Damon
Brownie
Hill
```

Exploring the function in the Debugger

Please watch the screencast below that demonstrates how you can examine a program in the Debugger and free it from errors. Afterwards, you can try this out yourself in Matlab.

Only pictures can be viewed in this version! For Flash, animations, movies etc. see online version. Only screenshots of animations will be displayed. [link]

Preview of the block structure of load_data1: The function first reads the information in the «file header», that is the first three lines. The table header is skipped, afterwards the data lines are read until the end of the file is reached. First names and surnames are extracted into two separate string matrices, the other data columns are converted to numerical values if necessary, and packed into another matrix.

Please try to do this yourself now. In this process, you will find additional hints on how the program works in detail.

- Make sure that your directory containing the example data is selected as the «current directory».

- Open the function **load_data1.m** in the Editor.
- Try to set a so-called «break point» on line 17 by clicking on the dash after the line number.
- Matlab will now inform you that you cannot set a break point because the program contains a syntax error (line 39). Try to correct it and save the function afterwards.
  Later Matlab versions will point out even earlier that there seems to be an error on that line.

Solution

A closing bracket at the end of line 39 is missing:
```
datestring = rdata(pos(1) + 1:length(rdata) );
```

- Now, the break point symbol has changed its colour to red (in most Matlab versions). That means that it is now active. If not, set it again.
- Call the function from the Command Window:
  ```
  >> [Mdata, Mfirstname, Mfamilyname] = load_data1('exp1.dat')
  ```
- Due to the break point, program execution is already stopped on line 17, that is, *before* executing this line.
- From now on, you can continue the program step-wise by selecting **Debug** > **Step** (or hit the F10 key), set a break point further down in the program and continue with **Debug** > **Continue** (or F5), or you can continue program execution up to the current cursor location by **Debug** > **Go Until Cursor**.
- You can examine the value of a variable by placing the mouse pointer over a variable name in the code. Alternatively, you can query the value of variables in the Command Window or in the Workspace.

Explanations of the program's function

- Lines 17-20: Load the file and return an error message in case it is not found. `fid` indicates this with the value -1.
- Lines 22-33: Read the 1st line, decode the file name and check that there is no conflict with the actual file name.
- Lines 35-40: Read the 2nd line and decode the date.
- Lines 42-43: "Over-reads" (skips) the recording system info and the table header line.
- Line 52: Start of the main loop that runs until EOF (end of file) is reached.
- Lines 53-60: Read a complete data line.
- Lines 62-81: The three data values registered as alphanumerical text (columns 3, 5, and 11) are converted into numerical data so that they can be stored in a single numerical matrix afterwards.
- Lines 84-86: The three return variables are filled with the data.

- On line 86, program execution will be stopped because Matlab found the following error:
  ```
  ??? Error using ==> horzcat
  CAT arguments dimensions are not consistent.
  ```
- To investigate the error in more detail, we make use of error breakpoints. Via **Debug** > **Stop if Errors/ Warnings...** we activate **Always stop if error** and run the function again. Now the program will stop at the error, and you can examine the involved variables (place the mouse pointer over the variable in the Editor, double click it in the Workspace, or type its name in the Command Window).

- Can you find the error? How can you correct it?

Solution

The problem lies on line 86:

```
Mdata = vertcat(Mdata, [gendern age condn data
 calibn]);
```

Remember that, when horizontally concatenating matrices, the number of rows of the matrices have to be identical. If you place the mouse pointer on the different variables in the square brackets of the `vertcat` statement, you will se that all except `data` are scalars, that is they only contain one row. `data`, however, has five data values arranged vertically, that is, five rows - this cannot work. We want to have the data values side by side on the same line. We can correct this by transposing `data`, that is by correcting this into `data'`.

- When you re-run the function now, it should work correctly and return the imported values.

Summary

- `fid = fopen(filename, mode)`: Opens a file
- `data = fgetl(fid)`: reads one line
- `data = fscanf(fid, format, n)`: Reads n data units in a certain format, specified in `format`
- `fclose(fid)`: closes a file
- `feof(fid)`: checks whether the end of the file has been reached (eof = end of file)

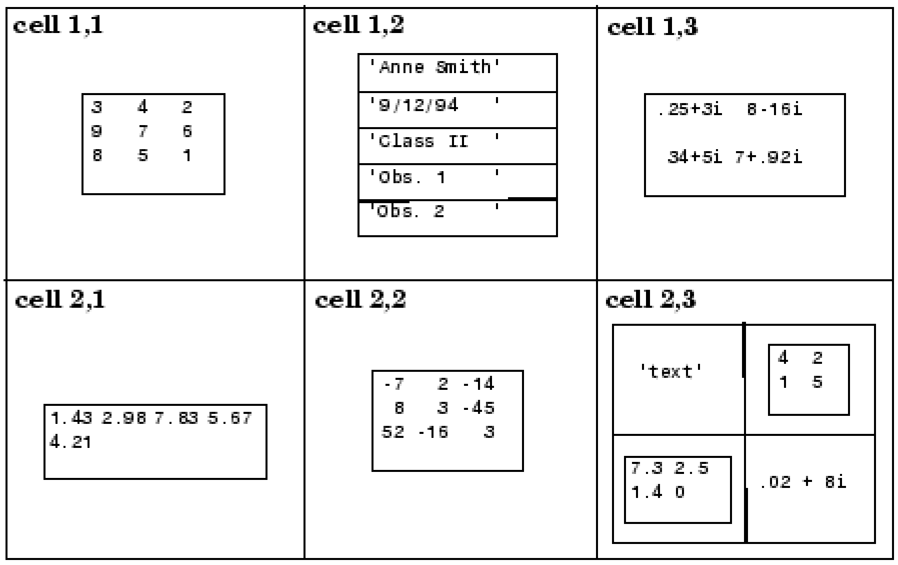
Data Representation 3: Cell Arrays

In this chapter you will get to know a different, comfortable way to represent data in Matlab: in so-called **cell arrays**.
While matrices only accept numbers and - with certain restrictions - *strings* as their contents, a cell array allows for storing differing data types in every one of its data fields (= cells), especially matrices of different size.

Data Representation as Cell Arrays

As you can see in the figure below, a cell array consists of several cells that are arranged in a matrix-like fashion. Each cell can contain different data entries:

- numerical matrices, i.e. *scalars* [6], *vectors*, or n-dimensional *matrices* [7] of any size (cell 1,1 / cell 1,3 / cell 2,1 / cell 2,2)
- matrices with *strings* (cell 1,2)
- further cell arrays (cell 2,3)
- *structure arrays* [8] (they are not depicted here and will be explained in section **Data Representation 4: Structure Arrays**).

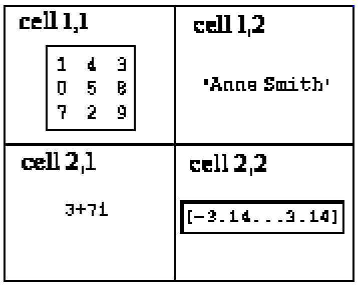


Schematic depiction of a cell array

Create and Reference Cell Arrays

Cell arrays can be recognised by the fact that their indices are not stated in round brackets as for matrices, but in curly brackets `{ }`.

The following examples show how cell arrays can be applied:



Creation of a cell array: Below you can see how the cell array shown above can be built. Please try to understand the function of these statements.

```
>> A = cell(2, 2)
```

This statement pre-defines an empty cell array wiht m x n cells. This is **optional**. If you do not do that, the cell array will be dynamically extended in the same way as matrices, as soon as additional cells are adjoined. As a matrix, a cell array always has to be «rectangular»; empty cells are added as necessary.

Now, the cells are filled with contents:

```
>> A{1,1} = [1 4 3; 0 5 8; 7 2 9];
>> A{1,2} = 'Anne Smith';
>> A{2,1} = 3+7i;
>> A{2,2} = -pi:pi/4:pi;
```

It is possible (but less clear) to define it all in one statement:

```
A = {[1 4 3; 0 5 8; 7 2 9], 'Anne Smith'; 3+7i, -pi:pi/4:pi};
```

Referencing and transforming

>> A	recall the complete cell array
>> A{1,1}	select cell 1,1
>> A{1,1}(2,3)	reference a single element of the matrix in cell 1,1
>> ischar(A{1,2})	this is indeed a string
>> cellplot(A)	visualisation of the structure
>> m = randn(60,50);	any matrix
>> b = num2cell(m);	convert a matrix into a cell array, one element per cell
>> c = mat2cell(m,[20 40],[25 25])	split matrix into sub-matrices per cell
>> c{2,1}	convert the cell array back into a matrix
>> d = cell2mat(c)	

Example for strings in cell arrays

Cell arrays are often used to store strings of different length without having to care for the problem of differing lengths. To do this, each string is stored in a separate cell. This method is supported by many Matlab functions.

An example:

```
>> texts = {'Hello world', 'The quick brown fox', num2str(pi), 'jumps over the lazy dog'}
>> texts{4}
>> length(texts{3})
```

Second example on data import from external files

The **previous example** will now be changed in a way that (1) the name information is returned as a cell array, and (2) the data are not read in a loop by using `fscanf`, but with a single call of the function `textscan`, which also returns the data in a cell aray.

Please have a look at the function **load_data2.m** by following the steps described below.

- Open the function **load_data2.m** in the Editor
- On line 47 you see how `textscan` is applied.
  ```
  Cdata = textscan(fid, '%s %s %s %f %s %f %f %f %f %f %s');
  ```
  This statement reads data into a cell array until the end of the file has been reached. The format of the data is defined in a format string (`%s` are strings, `%f` are floating point numbers). `Cdata{x}` will then contain all data from column x. Alphanumerical data are stored as a cell array in a cell, the numerical data as matrices.
- (regarding data processing, this function is intentionally left incomplete)
- on the lines 59-61, the data are re-configured according to the needs:
  ```
  Cfirstname = Cdata{1};
  Cfamilyname = Cdata{2};
  Mdata = [Cdata{4} Cdata{6:10}];
  ```

The function can be called by

```
[Mdata, Cfirstname, Cfamilyname] = load_data2('exp1.dat').
```

Summary

Points to remember

- A cell array consists of several cells that are arranged in a matrix-like way. Every cell can contain various kinds of data such as complete matrices, strings, furthre cell arrays or structure arrays.
- Cell arrays can be recognised by the fact that their indices are enclosed by curly brackets `{ }`, not round brackets as in matrices.

Commands

- `cellplot`: visualisation of the structure of a cell array
- `num2cell`: convert a matrix into a cell array, one element per cell
- `mat2cell`: divide matrix into sub-matrices per cell
- `cell2mat`: convert a cell array into a matrix
- `textscan`: read data from a file into a cell array

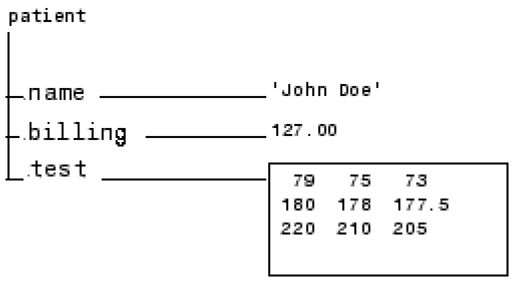
Data Representation 4: Structure Arrays

Another useful way to represent data are the so-called **structure arrays**. Every element of a structure array contains the same structure of various «data containers» that are each referenced by a unique name. Compared to *cell arrays* [9], it is more clearly visible what the contents of a referenced data field are.

[9] A data structure that consists of several cells arranged in a matrix. Every cell can contain various kinds of data such as complete matrices, strings, further cell arrays etc.

Data Representation in Structure Arrays

Every element of a structure array contains the same structure of distinctly named sub-elements, called **fields**. These fields can be seen as «data containers». In a similar way as cells of a *cell arrays*, they can hold all kinds of data items.
An element of a structure array could, for instance, contain a name as a string, a scalar for an amount due, and a matrix with medical test data.



Examples to try out:

>> screensize.x = 800	`screensize` only has two sub-elements
>> screensize.y = 600	they can be addressed like this
>> screensize.x	create an additional element;
>> screensize(2).x = 1024	it automatically assumes the same structure
>> screensize(2).y = 768	

Creation of the above example (patient data):
>> patient.name = 'John Doe';
>> patient.billing = 127.00;
>> patient.test = [79 75 73; 180 178 177.5; 220 210 205]

>> fieldnames(patient)	recall of the field names
>> lower(patient.name)	address an element and do a transformation
>> sum(patient.test(2, :))	reference the elements of the matrix
>> patient(3).name='Alan Johnson'	a new data set
>> patient(2)	thereby, patient(2) has been filled with empty fields

Application example

You could implement a function to import experiment data that stores all general data concerning the experiment trial in a structure array. In this way, you can avoid having to return the data in many individual variables.
That is, instead of:
function [Mdata, sex, age, eye, date, time] = loaddata(filename)
you could implement it in this way:
function [Mdata, expinfo] = loaddata(filename)
`expinfo` would consist of the fields:
expinfo.sex
expinfo.age
expinfo.eye
expinfo.date
expinfo.time

Summary

- Every element of a structure array contains the same structure of different «data containers» (fields) that are addressable with a name. Thus, contrary to cell arrays, it is clearly visible what a referenced data field contains.
- The fields are addressed by adding the field name to the variable name, separated by a period: `patient.name`

Your Data Processing Project

Now, you know all basic building blocks to enable you to start writing a data processing program of your own. The final goal is to have a program that imports data from files, processes them (aggregation, conversion, graphical display etc.) and stores the results in a file.
The project will be realised in several parts Px (P for project):

- Exercise P1: write the main program and the function to import the data
- Exercise P2: request user inputs
- Exercise P3: data processing and saving to an external file
- Exercise P4: computations
- Exercise P5: graphical visualisation of the data

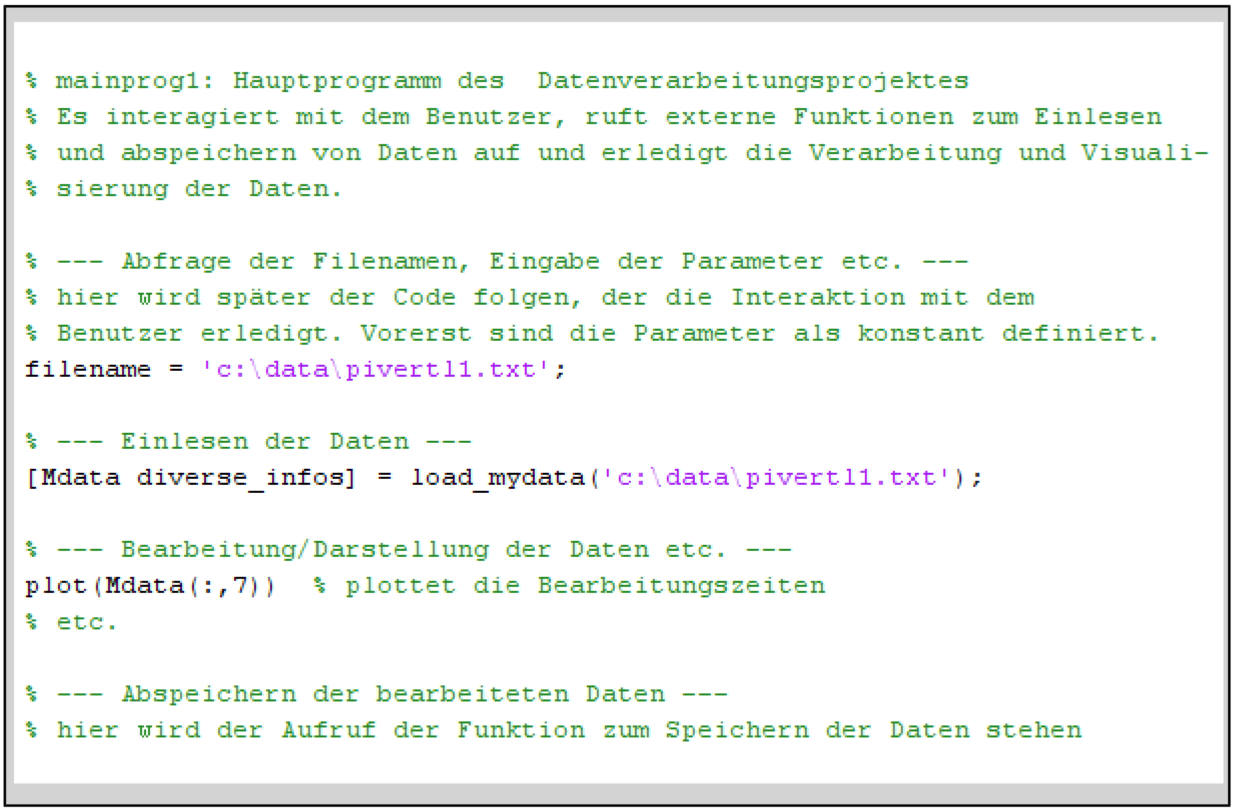
Data Base

Considering the data to be processed, there are several options for your project:

- You have data from your own research project that you want to process.
- If you do not have such a data set available, you can use the example project «**EyeData Processing**» provided by this course (it will be described later on).
  In this way, you can implement a kind of a template that might help you when you need to emplement a similar program later on.
- Alternatively, you can create a hypothetical practice data set (e.g. in Open Office Calc or MS Excel), similar to **exp1.dat**

Basic Structure

A data processing program usually consists of a main program (a script or a function) that controls the overall flow of the program - that is, it interacts with the user if necessary, calls functions to import or save data, processes the data, visualises them etc. Obviously, these latter parts can also be realised by calling other functions.
The following script may serve as a framework: **mainprog1.m**
You can again find this script with the example programs you have downloaded at the outset of this course.



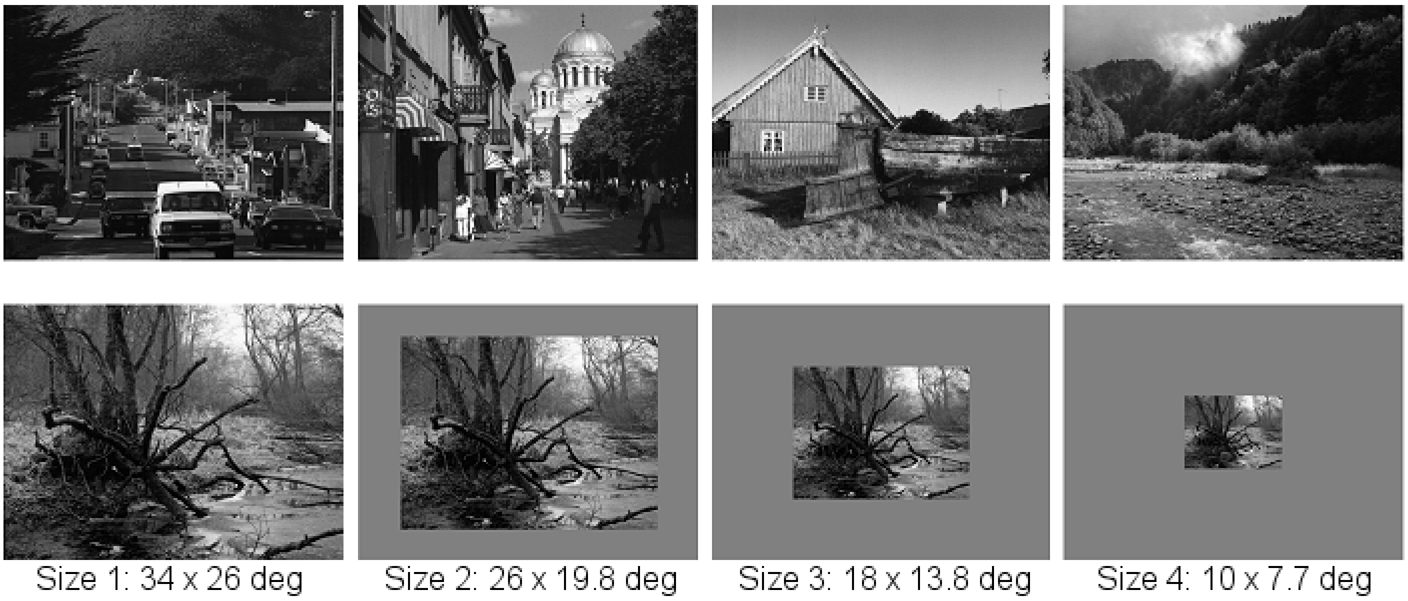
```
% mainprog1: Hauptprogramm des  Datenverarbeitungsprojektes
% Es interagiert mit dem Benutzer, ruft externe Funktionen zum Einlesen
% und abspeichern von Daten auf und erledigt die Verarbeitung und Visuali-
% sierung der Daten.

% --- Abfrage der Filenamen, Eingabe der Parameter etc. ---
% hier wird später der Code folgen, der die Interaktion mit dem
% Benutzer erledigt. Vorerst sind die Parameter als konstant definiert.
filename = 'c:\data\pivertl1.txt';

% --- Einlesen der Daten ---
[Mdata diverse_infos] = load_mydata('c:\data\pivertl1.txt');

% --- Bearbeitung/Darstellung der Daten etc. ---
plot(Mdata(:,7))  % plottet die Bearbeitungszeiten
% etc.

% --- Abspeichern der bearbeiteten Daten ---
% hier wird der Aufruf der Funktion zum Speichern der Daten stehen
```

If it is necessary for your application to pass parameters to the main program, it can later be converted into a function.

Data Base: EyeData Processing

Exercise in case you do not have your own data

The to-be-processed data originate from an eye tracking experiment. The subjects were shown natural scene images in different sizes on a video monitor (5.5 seconds for each image). They were asked to view them in preparation for a recognition test (which was not evaluated). During viewing, eye movements were recorded and registered as fixations.



Upper row: example images (shown in colour in the experiment). Lower row: the four image sizes

The goal of the exercise is to write a program that loads the data of several subjects and merges them into a single output file that can later be imported into other software packages such as SPSS for further processing. This requires additional data columns that contain the subject number and other information that is provided in the file header of the individual data files.
Moreover, the data has to be aggregated, i.e. calculation of mean values per subject, and saved in a way that there is only one row per subject. In this way, the data are prepared for statistical evaluations such as ANOVAs. Additionally, the program should contain features which graphically display the data (e.g. bar graphs of fixation duration per condition, or plots of the visuo-spatial distribution of fixations on the images).
The data files are contained in the zip file data.zip (you should have downloaded this file in the beginning) and are named vn**size**x**.fix** (vn are the initials of the subject, x designates the trial block 1-6).

Exercise P1: Data Import

Take the framework **mainprog1.m** for the main program outlined **above** and save it under a new name that fits your project (**eyedata.m** for the example task «EyeData Processing». The sample solution for this task can be found under the file name **eyedata_sol1.m**).

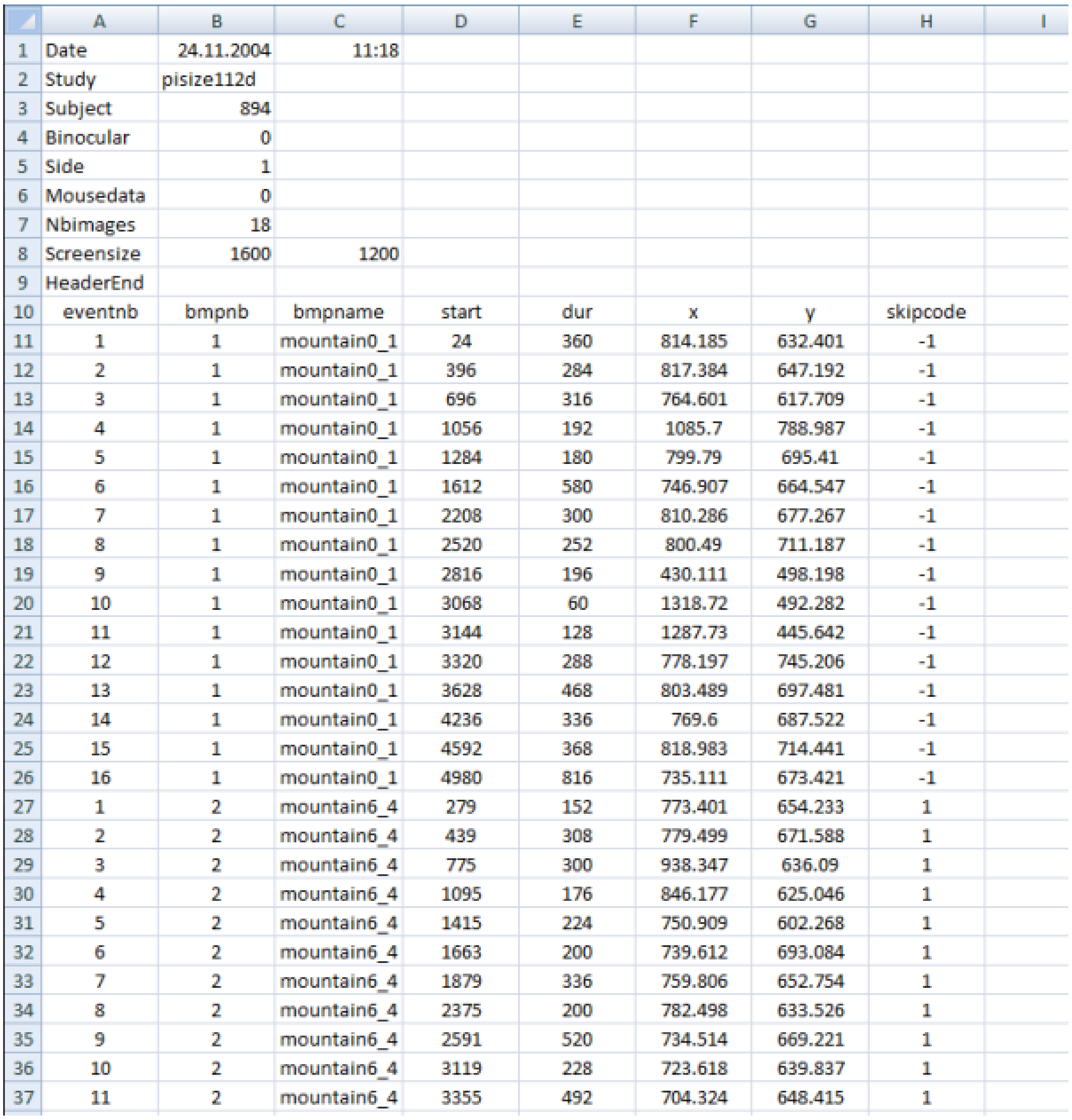
Write a new function to import your data from the external files. To do this, you can start from scratch, or you modify and extend **load_data1.m** used in an earlier example (it is recommended to save it with a new name as well).
If you have your own data set, please first consider what you want to do with your data. This will influence the way your function will have to return the imported data to the calling script. Relevant questions are:

- Which of the available data are needed for further processing?
- Are there certain data items that need to be recoded (typical case: conversion of experimental conditions coded in clear text into numerical codes)?
- Which data structures are best suited for further processing - single values, matrices, cell arrays, or structure arrays?

Guidelines for the «EyeData Processing» project
Try to implement a function **readfixdata.m** that reads a single data file and returns the data in suitable form, e.g. as

- a structure array with the fields study name, subject number, recording date, left/right eye, number of images, and screen resolution
- a cell array with the image names;
- and a matrix containing the eye movement data.

These suggestions refer to the sample solution, but - of course - there is not a single correct solution; it is just as well possible to pack everything into a cell array, for instance.
The data to process have the following stucture (**afsize1.fix**, tab-separated file):



	A	B	C	D	E	F	G	H	I
1	Date	24.11.2004	11:18						
2	Study	pisize112d							
3	Subject	894							
4	Binocular	0							
5	Side	1							
6	Mousedata	0							
7	Nbimages	18							
8	Screensize	1600	1200						
9	HeaderEnd								
10	eventnb	bmpnb	bmpname	start	dur	x	y	skipcode	
11	1	1	mountain0_1	24	360	814.185	632.401	-1	
12	2	1	mountain0_1	396	284	817.384	647.192	-1	
13	3	1	mountain0_1	696	316	764.601	617.709	-1	
14	4	1	mountain0_1	1056	192	1085.7	788.987	-1	
15	5	1	mountain0_1	1284	180	799.79	695.41	-1	
16	6	1	mountain0_1	1612	580	746.907	664.547	-1	
17	7	1	mountain0_1	2208	300	810.286	677.267	-1	
18	8	1	mountain0_1	2520	252	800.49	711.187	-1	
19	9	1	mountain0_1	2816	196	430.111	498.198	-1	
20	10	1	mountain0_1	3068	60	1318.72	492.282	-1	
21	11	1	mountain0_1	3144	128	1287.73	445.642	-1	
22	12	1	mountain0_1	3320	288	778.197	745.206	-1	
23	13	1	mountain0_1	3628	468	803.489	697.481	-1	
24	14	1	mountain0_1	4236	336	769.6	687.522	-1	
25	15	1	mountain0_1	4592	368	818.983	714.441	-1	
26	16	1	mountain0_1	4980	816	735.111	673.421	-1	
27	1	2	mountain6_4	279	152	773.401	654.233	1	
28	2	2	mountain6_4	439	308	779.499	671.588	1	
29	3	2	mountain6_4	775	300	938.347	636.09	1	
30	4	2	mountain6_4	1095	176	846.177	625.046	1	
31	5	2	mountain6_4	1415	224	750.909	602.268	1	
32	6	2	mountain6_4	1663	200	739.612	693.084	1	
33	7	2	mountain6_4	1879	336	759.806	652.754	1	
34	8	2	mountain6_4	2375	200	782.498	633.526	1	
35	9	2	mountain6_4	2591	520	734.514	669.221	1	
36	10	2	mountain6_4	3119	228	723.618	639.837	1	
37	11	2	mountain6_4	3355	492	704.324	648.415	1	

Most items in the file header are self-explaining, others are irrelevant for our project (Binocular, Mousedata). Side designates the eye from which the movement data has been recorded (1 = left, 2 = right).
The items in the main table (which continues until the end of the file) designate:

- eventnb - fixations, consecutively numbered within each trial
- bmpnb - image number = trial number
- bmpname - image name. The last character indicates the image size
- start - start time of a fixation relative to the start of the trial (in ms)
- dur - fixation duration [ms]
- x, y - coordinates of the fixatins on the image in pixels
- skipcode - marker for data filtering due to calibration problems. -1 = ok, all other codes designate problematic data.

Sample solution (opens in a new window)

The solution can also be found in the M-file **readfixdata_sol.m**.

User Interaction

Most larger applications require a certain amount of interaction with the user. Typical examples are:

- Selections of data files to be processed
- Specification of parameters such as limits, filter settings, selection of graphics options etc.
- Graphical inputs, e.g. definition of «regions of interests» (ROIs) or calibration points by means of mouse clicks
- Selection of the location and file format for saving data or graphics

Below, the most important interactive elements are listed. However, the implementation of integrated graphical user interfaces (GUIs) will not be discussed.

Cf. the information given in the Help System under **MATLAB > Function Reference > Creating Graphical User Interfaces**.

Selection of Files and Directories

Standard dialog boxes to select files or directory locations for loading and saving data as you know them from other applications are also available in Matlab.

All of them have a large number of configuration options, which cannot be discussed in detail here, thus only some typical application examples are shown. For further options, please consult the Help System.

Select a file: uigetfile

[FileName, PathName] = uigetfile('*.m', 'Select the M-file');

Opens the file selection dialog box. The function returns the file name and the path. If «Cancel» has been clicked, FileName = 0 results.

The first argument `'*.m'` is a filter, i.e. only files with this extension are shown. If all files should be visible, `'*.*'` can be entered. The second argument contains the text that will be shown in the window title.

The selection process begins at the current path as it is set in Matlab as the «Current Directory».

If you want the selection to start at a different place in the file system, you can specify it like this:

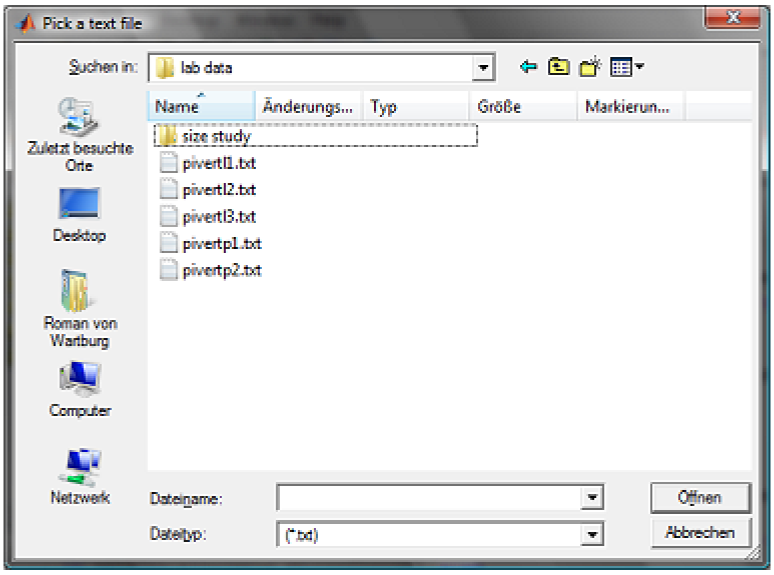
```
[FileName,PathName] = ...
uigetfile('d:\matlab\data\*.txt', 'Select data set');
```

If several file types should be available, you can specify this by means of a cell array. The desired file extensions are listed in the first column of the cell array:

```
[filename, pathname] = ...
uigetfile({'*.m'; '*.mdl'; '*.mat'; '*.*'}, 'File Selector');
```

Example for a file selection dialog with output of the result. Copy the code lines below into the Command Window (>> can also be copied - Matlab will ignore them).

```
>> [filename, pathname] = uigetfile('*.txt', 'Pick a text file');
>> if isequal(filename, 0)
>> disp('User selected "Cancel"')
>> else
>> disp(['User selected ', fullfile(pathname, filename)])
>> end
```



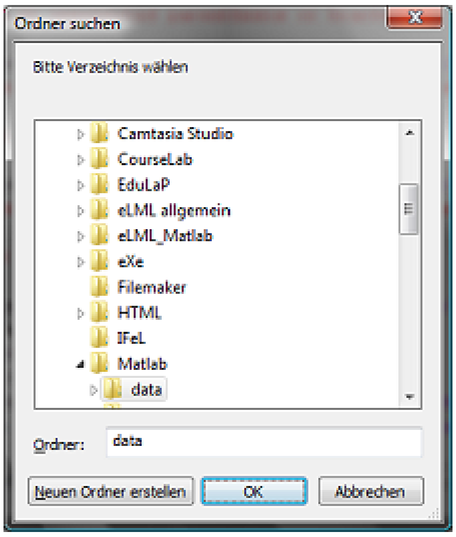
Resulting file selection window

Select a directory: uigetdir

directory_name = uigetdir('c:\data\', 'Please select the directory')

Opens a directory selection dialog box `directory_name` contains the selected path. If the user hits «Cancel», the value 0 is returned.

The first argument designates the path where the selection process starts out. The second argument holds the window title text.



Directory selection window

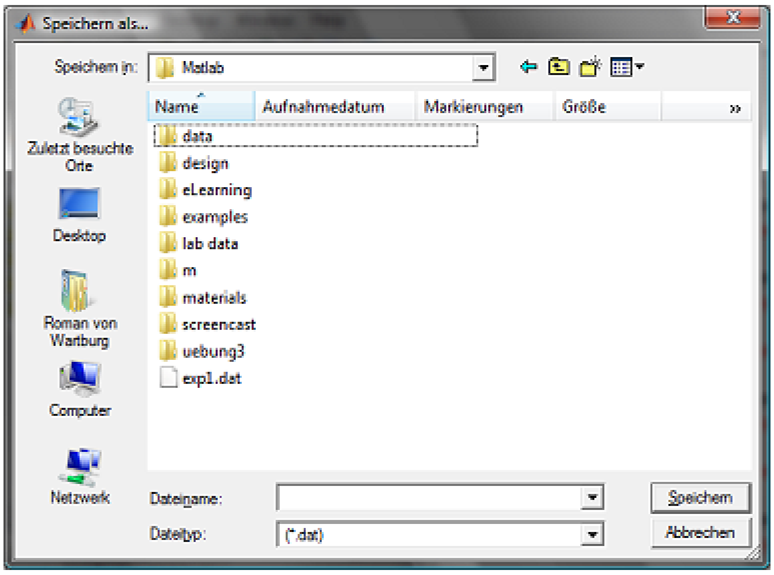
Select a directory/file name to save a file: uiputfile

[FileName, PathName] = uiputfile('*.dat', 'Save as...')

[FileName, PathName] = uiputfile('vp27data.dat', 'Save as...')

Creates a dialog box to select the location for a file saving operation. The function returns the selected file name and path. In case «Cancel» has been pressed, FileName = 0 results.

The first argument specifies the file type; in the second example, a complete file name is suggested. The second argument again specifies the window title text.



Window to select the location to save a file

Two useful functions to handle file and path names
Often, the queried file names have to be disassembled into their parts, or a complete path has to be assembled from parts. To aid this, there are helpful functions:

Please try it out for yourself.

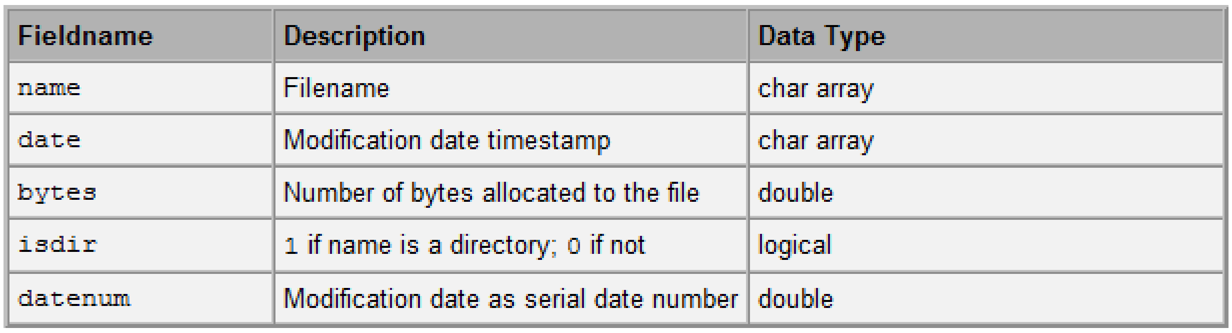
Decompose a complete path into its constituents:`fileparts`
>> [pathstr, name, ext] = fileparts('c:\data\new\myfile.txt')
And the converse: Generate a complete path from parts: `fullfile`
>> filepath = fullfile(pathstr, [name ext])
>> filepath = fullfile('C:', 'data', 'new', 'myfile.txt')

Sample Code: Select All Files in a Directory

Below you find the sample code for a function that might be helpful later when you implement a program. In data evaluation projects, it is often the case that you can simply store all to-be-evaluated data files in a certain folder. Thus it is helpful if you can simply select this folder in the evaluation program, instead of selecting single files.
The code shown below realises this: The user an select a directory, the program will read the contents of the directory and builds a list of all files with a specified extension (e.g. .tap for tab-separated ASCII files). Later on in the program, these files can be imported and processed in a loop.
You can simply overtake this code into your own program. You only have to modify the desired file extension. The file names are returned in the matrix `Mfiles`, the number of contained file names is stored in `nbfiles`.

```
pathname = uigetdir('', 'Select the           if the user hits 'cancel'...
directory');                                  ... terminate the program
if pathname == 0                              read all directory entries (*)
return                                        determine their number
end                                           pre-define the result matrix as empty matrix
Mdir = dir(pathname);                         loop over all directory entries
nbentries = size(Mdir, 1);                    is this NO directory entry?
Mfiles = [];                                  --> then it must be a file name
for entry_i = 1:nbentries                     extract the current file name
if Mdir(entry_i).isdir == false               exclude '.' and '..' (**)
filename = Mdir(entry_i).name;                determine the extension
if filename(1) ~= '.'                         compare with the desired extension
[p, n, ext] = fileparts(filename);            if matched, store the file name
if strcmpi(ext, '.tab')                       determine the number of found entries
Mfiles = strvcat(Mfiles, filename);
end
end
end
end
nbfiles = size(Mfiles, 1);
```

Weitere Erläuterungen
(*) The function `dir` returns the contents of a directory as a structure array with the following fields (for us, only the fields `.isdir` and `.name` are relevant):



Fieldname	Description	Data Type
`name`	Filename	char array
`date`	Modification date timestamp	char array
`bytes`	Number of bytes allocated to the file	double
`isdir`	`1` if name is a directory; `0` if not	logical
`datenum`	Modification date as serial date number	double

(**) `Mdir(...).name` always contains two special entries, that are needed for navigation within the file system: `'..'` (one directory level up) and `'.'` (current directory).
Typical output of `dir`:

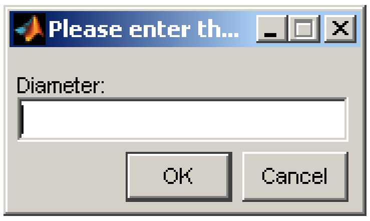
```
. aggr.dat jlsize3.fix rrsize4.fix slsize6.fix
.. bhsize1.fix jlsize4.fix rrsize5.fix tlsize1.fix
afsize1.fix bhsize2.fix jlsize5.fix rrsize6.fix tlsize2.fix
afsize2.fix bhsize3.fix jlsize6.fix slsize1.fix tlsize3.fix
afsize3.fix bhsize4.fix rawall.dat slsize2.fix tlsize4.fix
afsize4.fix bhsize5.fix rrsize1.fix slsize3.fix tlsize5.fix
afsize5.fix jlsize1.fix rrsize2.fix slsize4.fix tlsize6.fix
afsize6.fix jlsize2.fix rrsize3.fix slsize5.fix
```

Other User Input Features

Dialog box to input values: inputdlg

This function also has many configuration options; here, only exemplary cases can be shown. For further details please refer to the Help System.

```
>> answer = inputdlg('Diameter:', 'Please enter the value')
```

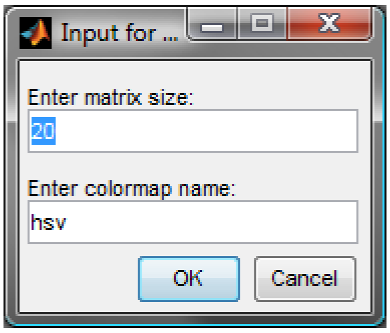


`answer` contains the entered value or `{}` (empty cell array) if «Cancel» has been clicked.

Please note: The entered value is returned as **String**! Matlab cannot know whether a number or a text is to be entered. If you want to proceed with a numerical value, you will have to convert the input into a number by means of `str2double` or `str2num`!

It is possible to query several values at once, and to define default values:

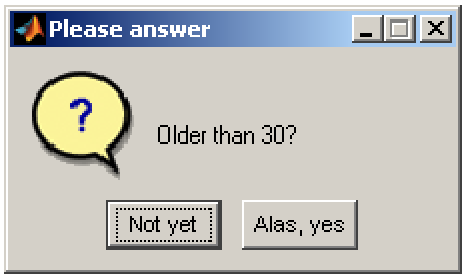
```
>> prompt = {'Enter matrix size:','Enter colormap name:'};
>> dlg_title = 'Input for peaks function';
>> num_lines = 1;
>> def = {'20','hsv'};
>> answer = inputdlg(prompt,dlg_title,num_lines,def)
```



The inputs are returned in a cell array.

Input request with buttons: questdlg

```
>> button = questdlg('Older than 30?', 'Please answer', ...
'Not yet', 'Alas, yes', 'Not yet')
```



Explanations: The last argument designates which button will be set as the «default», that is it will already be pre-selected (and can be confirmed with the enter key). This argument cannot be omitted.
`button` will return the clear text of the selected button. It is just as well possible to create more than two buttons.

You can find many more options in the Help System under the keyword «Predefined Dialog Boxes».

Graphical input of coordinates: ginput
With this function, coordinates can be entered in an existing graph.

Please have a look at this:

```
>> plot(peaks) % create any graphic
>> hold on
>> [x, y] = ginput(3)
>> plot(x, y, '.-r')
```

>In this example, three values are queried. By means of the return key, the input process can be terminated earlier. The coordinates are returned in the column vectors `x` and `y`.

Exercise P2: User Inputs

Integrate the necessary user inputs into your project's main program. Dependent on your actual goals, various inputs can be necessary:

- Selection of the file(s) to be processed, or the directory where the data are stored.
- Query any parameters that are necessary for data processing (i.e. filter settings)
- Selection of the file name and/or the storing location for saving the processed data.
- Ask the user whether certain evaluations or graphical outputs are desired.

If no file has been selected in the file selection process («cancel» has been pressed), the program should exit and return an appropriate message. Other erroneous inputs should either be made impossible, or lead to an error message.

For the EyeData project

- First, the user should be able to select a fixation file (that is, a file with the extension .fix).
- As a parameter, the user has to be asked whether s/he wants the data to be filtered based on the data in the column skipcode.
- Moreover, the user has to enter where and under which name the data (the merged raw data file and the aggregated data) are to be saved.

You can find the sample solution in the M-file **eyedata_sol2.m**

Summary

Summary of the most important commands:

- `uigetfile`: dialog box to select a file
- `uigetdir`: dialog box to select s folder (directory)
- `uiputfile`: dialog box to select the location to save a file
- `fileparts`: decomposes a complete path into its parts
- `fullfile`: creates a complete path from parts
- `inputdlg`: dialog box to query values
- `questdlg`: dialog box with buttons
- `ginput`: graphical input of coordinates
- `dir`: read directory entries from the file system

Writing Data to External Files

Basically, the procedure is very similar to reading from external files: First, the file is opened with `fopen`, then the data is written, and finally, the file has to be closed again with `fclose`.

Basic Functions to Write to External Files

Some of the commands described below have further parameters - please refer to the Help System if necessary.

`fid = fopen(filename, mode)`

Opens a file. Now, as `mode`, `'w'` (write - open a file for writing or creating) or `'a'` (append data to the existing file) has to be selected.

`fprintf(fid, format, M, ...)`

Writes the data contained in the matrix `M` (and any further matrices) to the file. The format of the written data is controlled by the `format` string.

There are extensive options how the data can be formatted by means of the format string. Some basic formats are shown in the example below. For the detailed format specifications and all further options please consult the Help System.

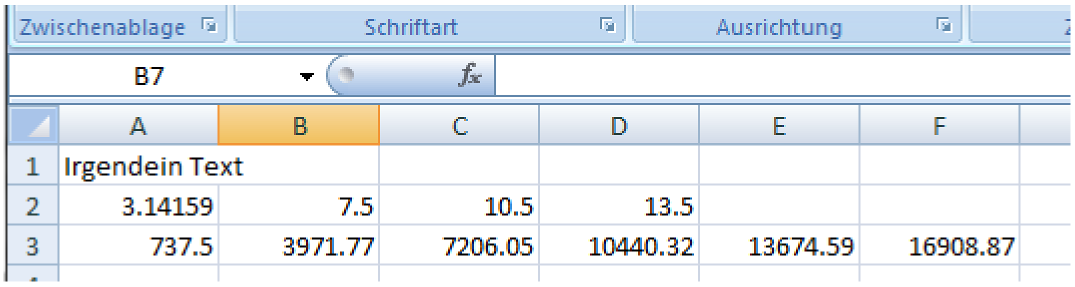
`fclose(fid)`

Closes the file. Only after closing, other applications can freely access the file.

Example: Writing to a file
Also see the M-file m-File **writedatafile.m**

```
% open the file in write mode
fid = fopen('C:\m\testdata.txt', 'w');
% Write text:
% %s designates a string, \n generates a line break
fprintf(fid, '%s\n', 'Any text');
% Schreiben von numerischen Daten:
% %g stands for fixed point notation, several values are written
% and separated by a tabulator (\t).
fprintf(fid, '%g\t', pi, [7.5:3:13.5]);
fprintf(fid, '\n'); % write the line break
% output with maximally 6 integer digits and 2 decimal digits
fprintf(fid, '%6.2f\t', [737.5:3234.273:20000]);
% close the file
fclose(fid);
```

This example generates the data file C:\m\testdata.txt (as viewed in MS Excel):



	A	B	C	D	E	F
1	Irgendein Text					
2	3.14159	7.5	10.5	13.5		
3	737.5	3971.77	7206.05	10440.32	13674.59	16908.87

You can find further file functions in the Help System, e.g. under: **MATLAB > Function Reference > File I/O > Text files**

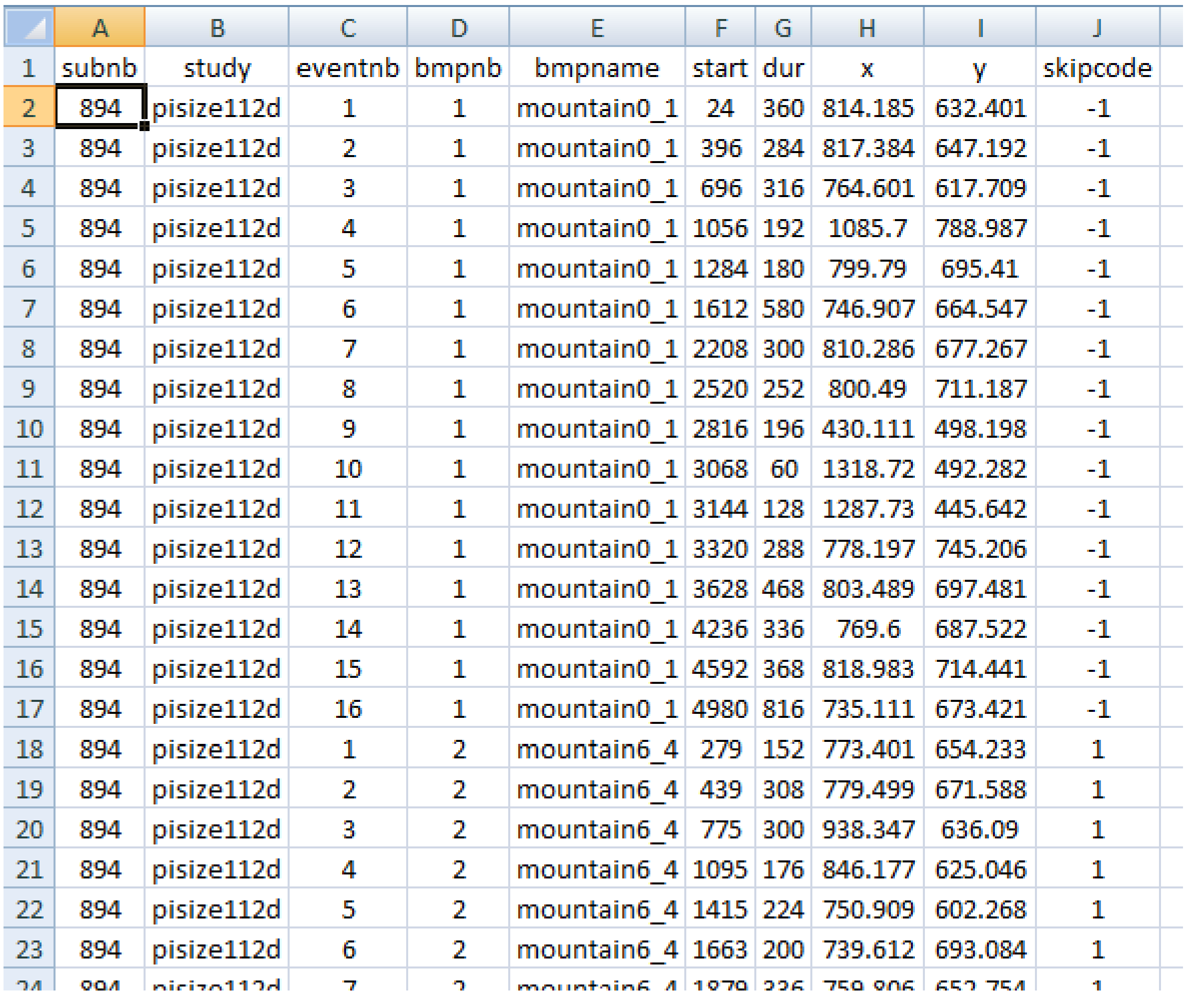
Exercise P3: Save Data

Write the program code that allows for saving the imported and processed dagta to an external file.

The intermediate data processing steps depend on your specific task. As a practice task, you can start with saving the unchanged data (or data with minor changes).

For the EyeData Processing project

- The fixation file selected by the user should be impoorted. Internally, the data have to be stored in a way that they can easily be re-called for saving.
- Afterwards, the program has to save the imported data to a file. According the image below, the header lines are not to be saved. Instead, two additional columns are added: Column 1 contains the subject number (Subject), column 2 contains the study name (Study).



	A	B	C	D	E	F	G	H	I	J
1	subnb	study	eventnb	bmpnb	bmpname	start	dur	x	y	skipcode
2	894	pisize112d	1	1	mountain0_1	24	360	814.185	632.401	-1
3	894	pisize112d	2	1	mountain0_1	396	284	817.384	647.192	-1
4	894	pisize112d	3	1	mountain0_1	696	316	764.601	617.709	-1
5	894	pisize112d	4	1	mountain0_1	1056	192	1085.7	788.987	-1
6	894	pisize112d	5	1	mountain0_1	1284	180	799.79	695.41	-1
7	894	pisize112d	6	1	mountain0_1	1612	580	746.907	664.547	-1
8	894	pisize112d	7	1	mountain0_1	2208	300	810.286	677.267	-1
9	894	pisize112d	8	1	mountain0_1	2520	252	800.49	711.187	-1
10	894	pisize112d	9	1	mountain0_1	2816	196	430.111	498.198	-1
11	894	pisize112d	10	1	mountain0_1	3068	60	1318.72	492.282	-1
12	894	pisize112d	11	1	mountain0_1	3144	128	1287.73	445.642	-1
13	894	pisize112d	12	1	mountain0_1	3320	288	778.197	745.206	-1
14	894	pisize112d	13	1	mountain0_1	3628	468	803.489	697.481	-1
15	894	pisize112d	14	1	mountain0_1	4236	336	769.6	687.522	-1
16	894	pisize112d	15	1	mountain0_1	4592	368	818.983	714.441	-1
17	894	pisize112d	16	1	mountain0_1	4980	816	735.111	673.421	-1
18	894	pisize112d	1	2	mountain6_4	279	152	773.401	654.233	1
19	894	pisize112d	2	2	mountain6_4	439	308	779.499	671.588	1
20	894	pisize112d	3	2	mountain6_4	775	300	938.347	636.09	1
21	894	pisize112d	4	2	mountain6_4	1095	176	846.177	625.046	1
22	894	pisize112d	5	2	mountain6_4	1415	224	750.909	602.268	1
23	894	pisize112d	6	2	mountain6_4	1663	200	739.612	693.084	1
24	894	pisize112d	7	2	mountain6_4	1879	336	759.806	652.754	1

The sample solution can be found in the m-File **eyedata_sol3.m**.

Summary

Summary of the most important points and commands:

Points to remember

- The procedure when writing data to a file is the almost the same as when reading: First, the file is opened by `fopen`, then data are written with `fprintf`, and finally, the file has to be closed by `fclose`.

Commands

- `fid = fopen(filename, mode)`: open a file
- `fprintf(fid, format, M)`: write the data in matrix M to the file, formatted according to the format string
- `fclose(fid)`: close a file

Glossary

ASCII:
American Standard Code for Information Interchange; the most common norm to represent alphanumerical characters.

cell array:
A data structure that consists of several cells arranged in a matrix. Every cell can contain various kinds of data such as complete matrices, strings, further cell arrays etc.

copy-and-paste:
On PC/Windows: CTRL-C and CTRL-V. On Mac OS: cmd-C and cmd-V

Debugger:
Editor function to assist in finding and correcting errors in a program.

matrix:
The most important way to represent data in Matlab. In the field of mathematics, a matrix is a table of numbers or other values. Matrices differ from ordinary tables in that they can be used for calculations.
Usually, the expression «matrix» refers to two- or higher-dimensional matrices (cf. scalar, vector).

scalar:
«null-dimensional» matrix, i.e. only one element

string:
Character string. A vector or a matrix whose elements are interpreted as ASCII coded characters.

structure array:
Every element of a structure array contains the same structure of various sub-elements that can be accessed with a unique name, called fields. These fields can be seen as «data containers» that can hold all kinds of contents.

vector:
one-dimensional matrix, that is only one row or one column

Lesson 4: Data Processing and Graphical Presentation

This lesson will show you how to further process data. On the one hand, it is about transformation, conversion, or aggregation of data, and computation of statistical characteristics. On the other hand, basic methods of data visualisation are introduced.

Learning Objectives

- You know how to selectively access data.
- You know the most important statistical functions.
- You know how to generate graphical figures and how to adapt them to your needs.
- You know the most important graphical functions and can apply them to your data.

Data Selection: Logical Indexing and the «find» Function

This section introduces important possibilities to selectively extract certain data from larger data fields. To this end, the following methods are available:

- logical indexing
- the `find` function
- a dedicated function `subM`

Afterwards, your data processing project will be complemented by data evaluation functions.

Logical Indexing

This data access method allows for addressing specific elements of a matrix based on logical comparisons. Put more simply, you can address all elements fulfilling a certain condition at once. For example:

```
>> a = magic(7);
>> a(a < 10) = 0
```

What has happened? All elements of `a` that are smaller than 10 are set to 0.

And how does it work? The comparison `a < 10` yields a matrix of the same size that contains the truth values of that comparison for every element. As you see now, comparison operators can also be applied to whole vectors or matrices! (cf. section **Comparison Operators** in Lesson 2). This resulting truth value matrix can be used as **logical index** to access the values in the data matrix.

Now try it out yourself:

>> a = magic(7);	result of the comparison
>> a < 10	extracts all elements that fulfill the condition
>> b = a(a > 20)	

It often happens that you need to extract data from a larger matrix according to certain criteria, e.g. the values from column 7 of all rows of a matrix `Mdata` whose column 3 contains a value larger than 0.5. An obvious solution for that problem is to write a loop and check every row by means of an `if` statement and, if the condition is met, extract the value from the desired column:

```
Mdata = rand(50, 7);
Mselect = [];
[nbrows, nbcols] = size(Mdata);
for row_i = 1:nbrows
if Mdata(row_i, 3) > .5
Mselect = vertcat(Mselect, Mdata(row_i, 7));
end
end
```

This can be realised with logical indexing in a much shorter and thus more elegant (and faster) manner:

```
>> Mdata = rand(50, 7);
>> Mselect = Mdata(Mdata(:,3)>.5, 7)
```

To clearly see how this works, we take a closer look at the «inner» expression:

```
>> Mdata(:,3)>.5
```

As you see, it yields a vector with the truth values of the comparison of all values in column 3 with the value 0.5. This vector can now be used as a logical index into a matrix. That is, all columns whose corresponding entry contains `true` (a numerical value of 1) are selected.

The find Function

The `find` function provides similar possibilities as the logical indexing discussed above. As a result, however, it does not return a matrix with logical indices, but a vector with the «linear indices» (cf. Lesson 1, section **Matrices**, paragraph «Linear Indexing» all the way down on the page).

Again, it is best to try it out:

```
>> a = magic(10);              find all elements of a that are > 50
>> selected = find(a > 50)     number of matching elements
>> length(selected)
```

This now results in a vector with the linear indices of the matching elements. It can be used for further operations:

```
>> b = a(selected)             extract the corresponding elements
>> a(selected) = inf           set those elements in a to inf
```

In most cases, `find` can be replaced by logical indexing. Usually, this makes sense as this is processed faster.

A Helpful Function: subM

You can find this useful function (which is not included in the standard function set of Matlab) with the example programs (zip file). It is defined like this:

```
function Mselected = subM(Min, col, min, max)
```

`subM` is a function that extracts all rows from a matrix `Min` whose value in column `col` lies within a defined range from `min` and `max` (inclusive).

```
>> M1 = magic(10);
>> column = 3; minvalue = 10; maxvalue = 30;
>> Mnew = subM(M1, column, minvalue, maxvalue)
```

Of course, it is possible to program this directly using **logical indexing**. However, this is less comfortable and results in a code that is more difficult to read :

```
>> Mnew = M1(M1(:, column) >= minvalue & M1(:, column) <= maxvalue, :)
```

Explanation: The results of the two comparisons are logically combined with `&` (AND operation). Let's look at this in detail again:

```
>> M1(:, column) >= minvalue
>> M1(:, column) <= maxvalue
>> M1(:, column) >= minvalue & M1(:, column) <= maxvalue
```

If you have a look at the function `subM` in the Editor (by typing `edit subM`), you see that - except for a plausibility check - it does not do anything else. Nevertheless, it can be helpful to write such often used operations as functions. This will later save time, and you get code that is easier to understand.

Summary

Points to remember

- Logical indexing: A data access method that is used to address at once all elements of a matrix that satisfy a certain condition.
  e.g. reading all elements from `a` that are smaller than 10: `b = a(a < 10)`

Commands

- `find`: Find elements in a matrix. As a result, the linear indices to the matching elements are returned.
- `subM`: A help function (not included in Matlab's library). Extracts all rows from a matrix in which the value in a certain column lies in a defined range.

Basic Statistical Functions

In this chapter, basic statistical functions included in the base version of Matlab are introduced. That is,

- functions to calculate statistical characteristics (descriptive statistics)
- calculation of correlation and covariance values
- functions to generate random data sets with defined distributions

Descriptive Statistics

Even the basic version of Matlab offers a set of mainly descriptive statistics functions. Further functions are included in the Statistics Toolbox for Matlab, which has to be licensed separately. Its functions are thus not discussed here.

In case you have access to the Statistics Toolbox, you can find all necessary information in the Help System under **Statistics Toolbox** (main category in the Contents part)

Minimum and maximum

The functions are called `min` and `max`. They can both be used in the same way:

Try it out:

>> a = magic(5);	calculates the minima in **columns**.
>> min(a)	calculates the minimum or maximum of a complete table
>> max(max(a))	define a second matrix
>> b=fix(rand(5)*25)	Compares `a` and `b` element by element
>> max(a, b)	and writes the maximum of each comparison in a matrix of the same size.
>> min(a, [], 2)	Of course, `a` and `b` have to be of the same size.
	In this way, you can specify the dimension of the calculation.
	1 are columns (default), 2 are rows, 3 are «planes», etc.

Mean, median, and modal values

There are three functions that have the same structure: `mean`, `median`, and `mode`.

Again, you best see for yourself:

>> m=fix(randn(200,6)*18+48);	matrix with normally distributed values
>> mean(m)	mean in **columns**
>> modeval=mode(m)	modal value, i.e. the most frequent value
>> [vals,freq]=mode(m)	in this way you can also get the numbers of values
>> median(m,2)	You can specify the dimension, too.

If you want to determine the value for a complete array, you first have to convert it into a column vector. A «logical solution» for this would be:
m1 = reshape(m, numel(m), 1);
But there's a «short cut»:

>> m1 = m(:)	`m(:)` means: just take ALL values of m
>> mode(m1)	or even more directly
>> mean(m(:))	

Variance and standard deviation

The two functions `var` and `std` are usually used in their most simple form (for details see the Help System). It will also calculate the values in columns.

Please study the examples below:

>> var(m)	variance
>> std(m)	standard deviation
>> std(m1)	or over the one-column array

Percentiles

The function `prctile(data, percvalues)` allows for computing percentiles in a comfortable way. The calculation also runs over columns of a matrix `data`. The parameter `percvalues` is a row vector that contains the desired percent values.

Percentiles calculation:

>> prctile(m, [5 50 95])	Calculates the 5%, 50%, and 95% percentile for every column.
>> prctile(m, [25 75], 2)	Every row of the result contains the three percentile values of a column. You can also specify the dimension for prctile.

Correlation and Covariance

The only inference statistics function of Matlab in its basic version is the calculation of correlations with `corrcoef`. The most important formats are:

`R = corrcoef(X)`

Yields a matrix R with the correlation coefficients. The function runs over a two-dimensional matrix, which is interpreted as follows: The **columns represent the dependent variables**, and the **rows represent the individual observations** (that is, the measured values).

`R = corrcoef(x, y)`

`x` and `y` are column vectors corresponding to two dependent variables. More precisely, if x and y are *not* column vectors, they will automatically be converted into column vectors. Obviously, the number of elements in both arrays need to be the same.

`[R, P, RLO, RUP] = corrcoef(X, 'alpha', .01)`

In this way, you obtain further values. `P` contains the significance values based on the specified `alpha` value. The parameter `alpha` can be omitted - the default value will then be .05 for 95% confidence interval.

`RLO` and `RUP` contain the lower and upper limits for a 95% confidence interval for every correlation coefficient.

As usual, you can find the details on further parameters in the Help System.

To illustrate this, an example from the Help System is shown. A data matrix which includes a correlation between column 4 and the other columns is used.

```
>> x = randn(30, 4);          uncorrelated random data
>> x(:, 4) = sum(x, 2);       now we generate a correlation
>> [r, p] = corrcoef(x)       calculate correlation coefficients and p values
>> [i, j] = find(p < 0.05);   find the significant correlations (columns, rows)
>> [i, j]                     display their columns and rows
```

Random Data with Defined Distributions

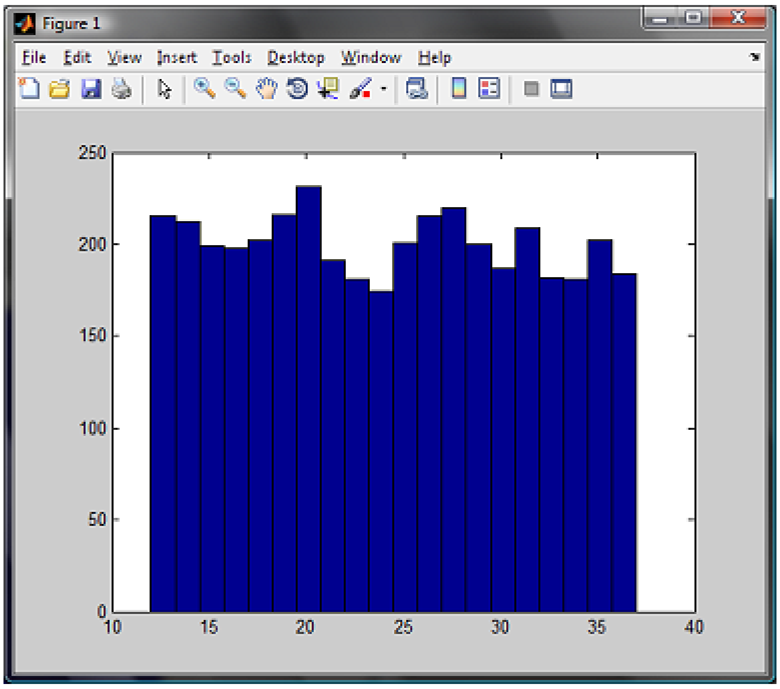
To test an evaluation program, it is often helpful to generate random data that have well-defined distributions.

Permutations
If you want to obtain a randomly ordered sequence of numbers in a certain range, you can use the function `randperm` (random permutation).
randperm(n)
This creates a randomly ordered sequence of the numbers 1 to `n` as a row vector, e.g. `[3 2 6 4 1 5]`. You can shift this sequence into any range by simply adding a constant value, or scale it by multiplying with a constant.

Equally distributed random numbers
The basic function to generate equally distributed random numbers is called `rand`. It generates numbers between 0 and 1. The syntax is:

```
a = rand(rows, cols)
```

This statement creates a matrix with the dimensions specified by `rows` and `cols`.
More often, random numbers in a specific range are needed. To generate data in the range from `LowerBound` to `UpperBound`, you can use this method:
data = rand(rows, cols) * (UpperBound - LowerBound) + LowerBound;



*Result of data = rand(4000, 1) * (37 - 12) + 12*

For most applications, this form suffices. In case you have to be sure that a different sequence is generated every time you call the function, you always want the same sequence, or have to know the random generator algorithm precisely, please consult the Help System on `rand`.

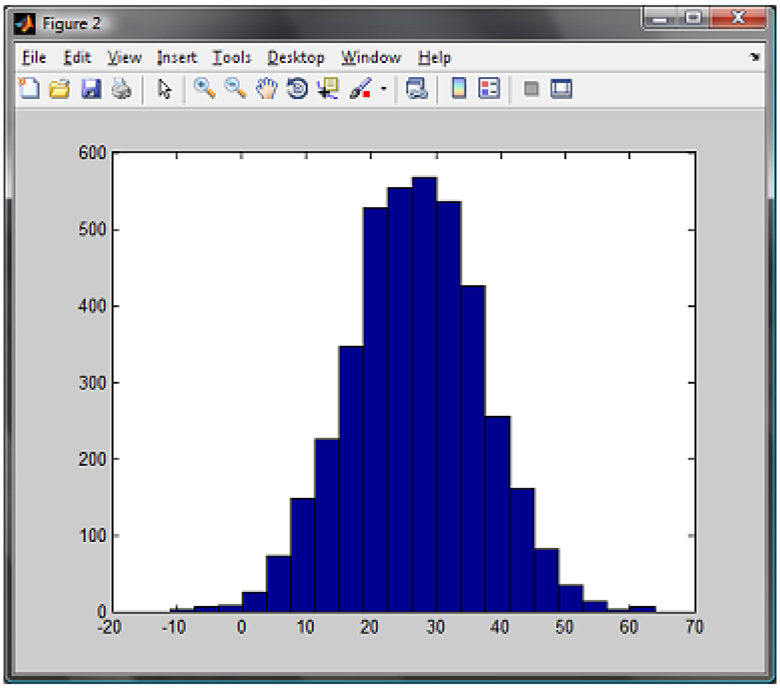
Normally distributed random data

With the function `randn` you obtain normally distributed random data with a mean of 0 and a standard deviation of 1.

data = randn(rows, cols)

To create data with any mean and standard deviation, proceed as shown below:

```
>> M = 27;                               define the mean
>> SD = 10;                              define the standard deviation
>> data = randn(4000, 1) * SD + M;       generate the data
>> mean(data)                            verify the mean
>> std(data)                             and the standard deviation
```

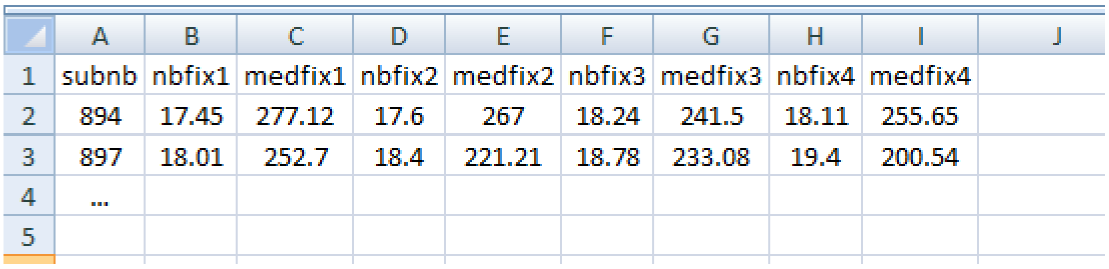


Exercise P4: Project Task - Data Processing

Implement the data processing parts in your project. How the data are processes depends on your specific task: maybe transformations, aggregation, calculation of average values or merging and regrouping of the data.

For the EyeData Processing Project

1. Extend the user interaction function for selecting the data: In a loop, several data files (i.e. .fix files) should be selectable (end this process when the user clicks "Cancel").
   Alternatively, you can also implement that the user can select a directory, from which all present .fix files are imported. You find the sample code for this in Lesson 3 **Sample Code: Select All Files in a Directory** (this is implemented in the sample solution for Exercise P5, **eyedata_sol5.m**).
2. In a loop, all files selected by the user are imported. The data have to be internally stored in a way that they can later be easily accessed for saving.
3. The program filters the data (if requested by the user) based on column **skipcode**. All data with -1 are ok, the others are to be deleted.
4. The filtered data of all subjects are to be saved in a single data file (= merged raw data file). The format remains the same as in Exercise P3, with the only difference that now there are the data sets of several subjects, one set after the other.
5. Per subject and image size, the following values are to be computed: (1) mean number of fixations per image, (2) the median of the fixation duration (as the distribution is skewed!) of all fixations that were made on all images of a certain size. Image size is defined by tha last character of the image name (1 to 4). These values are to be saved as repeated mesurements data (as shown in the image below) into the file **aggregated data**.



	A	B	C	D	E	F	G	H	I	J
1	subnb	nbfix1	medfix1	nbfix2	medfix2	nbfix3	medfix3	nbfix4	medfix4	
2	894	17.45	277.12	17.6	267	18.24	241.5	18.11	255.65	
3	897	18.01	252.7	18.4	221.21	18.78	233.08	19.4	200.54	
4	...									
5										

The sample solution can be found in the M-file **eyedata_sol4.m**

Summary

- `min, max`: minimum and maximum
- `mean, mode, median`: mean, modal, and median value
- `var, std`: variance and standard deviation
- `prctile`: calculation of percentiles
- `corrcoef`: correlation coefficients
- `randperm`: randomly ordered sequence of the numbers 1 to n
- `rand`: equally distributed random numbers between 0 and 1
- `randn`: normally distributed random numbers with mean 0 and standard deviation 1

-
  ```
  data = rand(rows, cols) * (UpperBound - LowerBound)
   + LowerBound;
  ```
  generating random numbers in a defined range

Graphics Basics: Figures and Axes

General information on graphics in Matlab

Matlab provides extensive possibilities to graphically display data. It would go beyond the scope of this course to give a comprehensive overview of all features. Instead of that, the basic mechanisms are introduced, together with a demonstration of those graphics that are most often used in our field.

- handling figure windows
- interactive modification of graphics
- axes
- universal plots
- bar graphs
- a short demo of other graphs
- annotation and extension of graphics

Creation and Manipulation of Figures

Every graphical output takes place in a special window, a so-called **figure graphics object**, in the following called **figure**. In the command line mode, the explicit creation of a figure is often unnecessary, as Matlab automatically creates a figure if none is available yet. However, default values (window size, background color, etc.) are applied which often do not suit the necessities of the task. Thus, in a program that uses graphical outputs, the figures are usually created explicitely and the needed properties are set accordingly.

To better understand how Matlab deals with graphical outputs, please have a look at the following exemples:

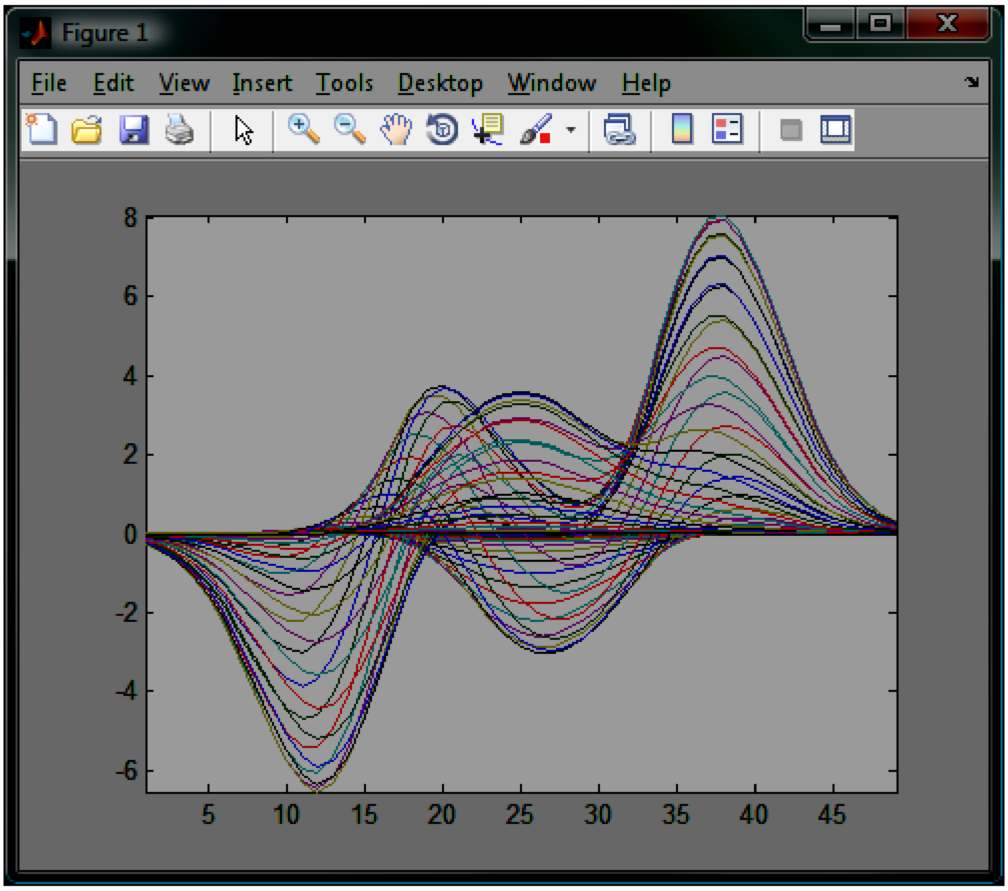
>> plot(sin(-pi:.1:pi))	Matlab automagically creates a figure with default settings
>> plot(cos(-pi:.1:pi))	the new graphical output replaces the previous one
>> hold on	fixates the graphics (the counterpart is `hold off`)
>> plot(sin(-pi:.1:pi),'r')	plot the sine again, now in red - this time, the previous curve is not overwritten

As you see, every graphical output is directed to the currently active graphics window (that is, the one in focus / the last used one). If you need an additional graphics window, you have to create an additional figure:

```
>> figure
>> bar(rand(10,4))
```

A new figure can also be created by using the menu item of the current figure **File > New > Figure**.
The display options of every graphics window can later be changed by using the built-in tools. In this way, you can modify a graph according to your needs and afterwards print or save in various file formats.

Exercise: Have a look at the plotting tools now. You might also want to consult the information provided in the Help System (`docsearch 'interactive plotting'`).



But what is if you want to plot to the first figure again? As long as you work in the command line mode, you can simply get the first window into the foreground «manually», and continue with your graphics commands:

>> bar(rand(10,1))	the plot goes to the first window again
>> close all	closes all open figures

If you want to use several figures from within a program, you proceed like this: Every figure is created separately and receives a so-called **figure handle**. It is used to repeatedly reference a certain figure later.

>> fig1=figure('Name','bar graphs');	the parameters determine the title
>> bar3(rand(10,3))	of the window (optional)
>> fig2=figure('Name','line graphs');	a second figure
>> plot(sin(-pi:.1:pi),'r')	change back to the 1st figure
>> figure(fig1)	e.g. switch off the grid
>> grid off	clear the whole figure (clear figure)
>> clf	

With functions such as `clf`, `close` etc. you can directly specify which figure you want to access without explicitely making it active, e.g. by `figure(fig1)`, `clf(fig2)`, or `close(fig1)`.

Figure Properties

Changig the properties of figures is done with **argument pairs**, as you have seen in the above example with the window title. The first argument of a pair indicates which property you want to change (e.g. 'Name'), the second argument specifies the value to be set.

For figures, a plethora of modifiable properties are available. You find the complete overview in the Help System, for instance under the keyword 'figure properties' (`docsearch 'figure properties'`). The most important ones are briefly described in the next section.
The properties can either be set directly when creating a figure:
>> fig2=figure('Name','line graphs');
Or you can modify them later by means of `set`. For this, you need the **handle** of the figure to be modified. You can either specify the handle directly (that is, `fig2`), or you can recall the handle of the currently active figure with `gcf` (get current figure). Example:
>> set(fig2, 'Name', 'Another name');
>> set(gcf, 'Name', 'Another name');

The most important figure properties
`'Name', 'titletext'`
Title of the figure window
`'Color', ColorSpec`
Background colour. For information on how to specify colours see the Help System: `doc colorspec`. Examples:

>> set(gcf,'Color','yellow')	`'yellow'` is one of the pre-defined colours
>> set(gcf,'Color','r')	`'r'` is short for 'red'
>> set(gcf,'Color',[.8 .8 .8])	RGB colour triple, results in light gray

`'Position', [left, bottom, width, height]`
Position and size of the window
`left, bottom`: distance in pixels from the left and bottom screen border
`width, height`: width and height of the window in pixels
Example:
>> fig=figure('Position', [100 100 800 600]);
A useful assisting function: How to find out the screen size in case you want to size the window relative to the screen size:
>> ss=get(0,'ScreenSize')
`'Resize', 'on' or 'off'`
Determines whether the user can manually change the window size.
`'Toolbar'`, `'none'` or `'auto'`
Determines whether the toolbar of the figure window is visible. 'auto' is the default value, that is, the window is visible.
`'Visible'`, `'on'` or `'off'`
With 'off', the window can be hidden, but it is still available for graphic outputs.

Axes

Actually, a graphics output is not directed into a figure, but into the **axes (or axes system)** contained therein. An axes is - in Matlab terminology - a system of two (x, y) or three (x, y, z) axes. This should not be confused with `axis`, with which you can set the properties of the actual axis (e.g. the scales, see below).

Axes, too, have a multitude of properties (`docsearch 'axes properties'`), but `'Color'` is probably the only one you will ever use.

All other axes properties can be changed by separate commands. For a demonstration, please close all open windows and create a new figure (simply use copy/paste):

```
>> close all
>> figure('Position', [100 100 800 600])
>> plot(peaks)
```

Now we can change the settings:

>> title('Peaks function')	title of the graph
>> grid off	switch off the grid
>> grid on	switch it on again
>> axis off	remove axis lines
>> axis on	change the scales: [xmin xmax ymin ymax]
>> axis([0 100 -20 20])	tightly fit the scales to the data
>> axis tight	set background colour. **gca** = get current axes
>> set(gca,'Color','blue')	

Further options of `axis`: see the Help System `doc axis`

>> box on	box(frame) around the graph
>> xlabel('Abscissa')	label the axes
>> ylabel('Ordinate')	

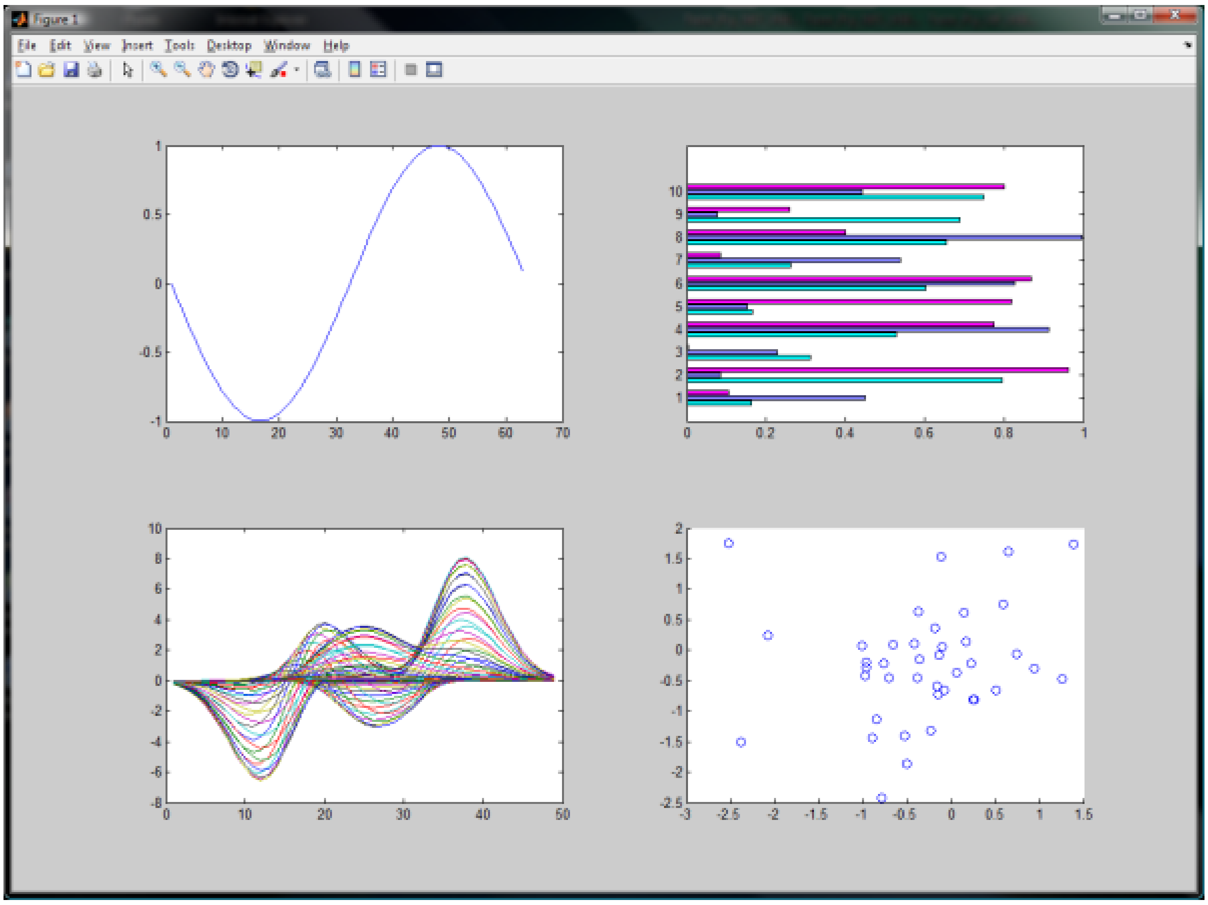
Another useful function is Eine weitere nützliche Funktion ist `waitforbuttonpress`: It pauses program execution until the user clicks into the graphics window or presses a key when the figure is active:

```
>> figure, plot(peaks), waitforbuttonpress, bar(peaks)
```

A different method for user interaction: the figure property `currentcharacter` (see Help System).

Multiple Axes

It is often useful to have a figure with several sub-plots (panels). You can create several axes within a figure by means of the function `subplot`.



Multiple axes

Please go through the following example:

>> close all, figure('Position', [100 100 800 600])

>> subplot(2,2,1)	creates 2x2 sub-plots and activates the first one
>> plot(sin(-pi:.1:pi))	plots into the first sub-plot
>> subplot(2,2,2)	select the second sub-plot
>> barh(rand(10,3))	horizontal bar graph
>> subplot(2,2,3)	the bottom right sub-plot
>> plot(peaks)	and the last one
>> subplot(2,2,4)	select the first one again
>> scatter(randn(1,40), randn(1,40))	only erases the contents of the current sub-plot (clear axes). `clf` would erase **all** axes from the figure.
>> subplot(2,2,1)	
>> cla	

Summary

Points to remember

- Every graphics output takes place in a dedicated figure window.
- More precisely, the graphics output is directed to an axes system contained in the figure window. There can be more than one axes in a figure window.
- Changing most figure properties is done with argument pairs. The first argument of a pair indicates which property you want to change, the second argument specifies the value to be set.

Commands: figure properties

- `figure`: create a figure window
- `clf`: clear the contents of a figure window
- `close, close all`: close figure(s)
- `set(gcf, 'property', parameter);`
  Change the properties of a figure window by means of argument pairs
  Some properties: `Name, Color, Position, Toolbar, Visible`
  `gcf` = get current figure handle
- `waitforbuttonpress`: Wait for key press or mouse click

Commands: axes

- `title`: set the title of an axes
- `xlabel, ylabel, zlabel`: label an axis
- `grid on, grid off`: show or hide the grid
- `axis on, axis off`: show or hide the axes
- `axis tight`: tightly fit scales to data
- `axis([xmin xmax ymin ymax])`: set scales
- `box on, box off`: show or hide the frame around the axes
- `hold on, hold off`: fixate graph, or release fixation
- `set(gca,'Color','blue')` :
  change the properties of an axes; `gca` = get current axes handle
- `subplot`: create or address sub-figures (= axes)
- `cla`: clear the current axes

Graphics Tools

Below, you will get to know some important graphics tools. Again: Matlab offers many options, and it is not possible here to explain all of them. As usual, you find everything in the Help System.

An overview over all graphics commands is given in the Help System under **Function reference > graphics > basic plots and graphs**

With the function `plot` as an example, the principle of most graphics commands are explained. After that, other frequently used graph types are briefly demonstrated.

The Plot function

The `plot` function creates two-dimensional line graphs. The data have to be given as column vecors. The data points are connected by a line in the sequence as they are found in the vectors.

In the following examples, you can see the two forms of the plot function:

Form 1: `plot(Y)`
If only one vector is given as argument, the data are interpreted as Y values and plotted against their index on the X axis.

```
>> ydata = sin(-pi:.1:pi);
>> plot(ydata)
```

Form 2: `plot(X, Y)`
If two vectors are given, they are interpreded as X and Y axis data.

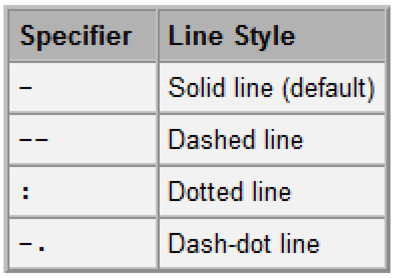
```
>> xdata = 20:39; ydata = rand(20, 1);
>> plot(xdata, ydata)
>> figure
>> plot(sin(-pi:.1:pi), cos(-pi:.1:pi))
```

LineSpec: set the line properties
plot(X1, Y1, LineSpec, ...)
The plot function accepts one argument that allows to set three important properties of the curve:

- **Line style specifier:** the style of the line (standard, dotted, dashed etc.)
- **Marker specifier:** the style of the data point markers (dot, circle, triangle, star, etc.)
- **Color specifier:** the colour of the curve

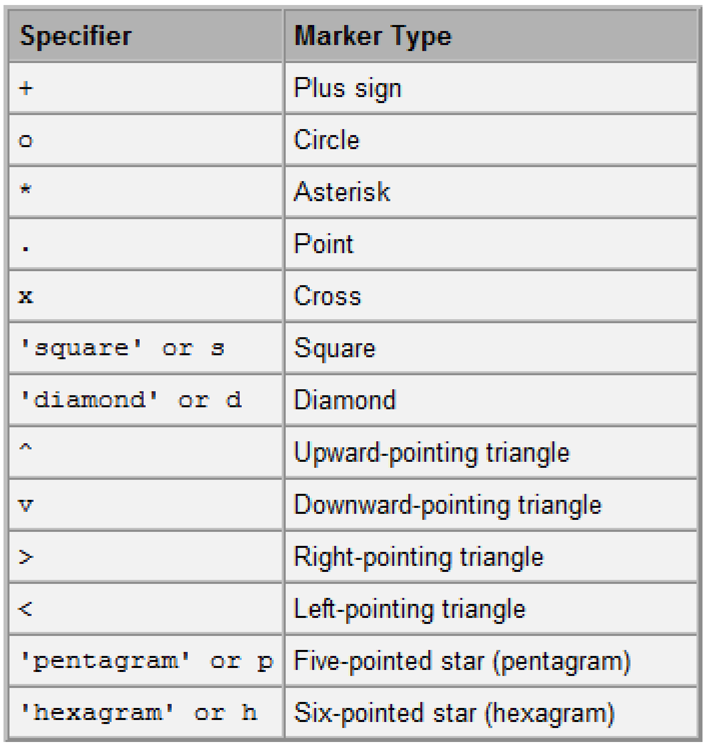
Example: To plot data with a red, dashed line and crosses as data markers, write:
plot(x, y, '--xr')

Line Style Specifiers



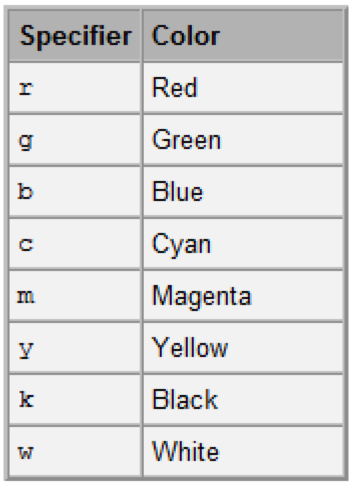
Specifier	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker Specifiers



Specifier	Marker Type
`+`	Plus sign
`o`	Circle
`*`	Asterisk
`.`	Point
`x`	Cross
`'square'` or `s`	Square
`'diamond'` or `d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`'pentagram'` or `p`	Five-pointed star (pentagram)
`'hexagram'` or `h`	Six-pointed star (hexagram)

Color Specifiers



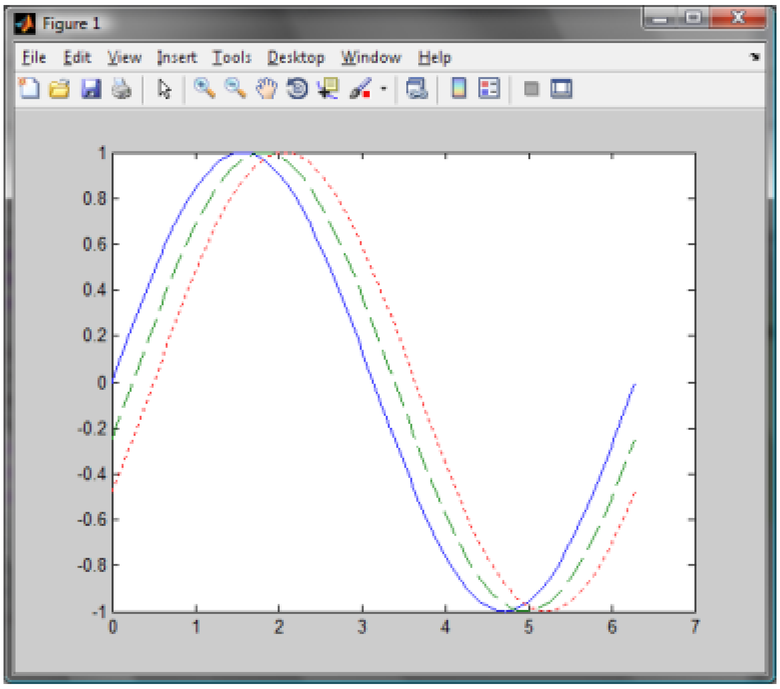
Specifier	Color
`r`	Red
`g`	Green
`b`	Blue
`c`	Cyan
`m`	Magenta
`y`	Yellow
`k`	Black
`w`	White

You can just as well draw plots without lines or without markers, by omitting the corresponding specifiers.

Also see the information in the Help System (`doc linespec`).

This example shows how several data sets can be plotted with a single plot statement:

```
>> t = 0:pi/100:2*pi;
>> y1 = sin(t);
>> y2 = sin(t-0.25);
>> y3 = sin(t-0.5);
>> plot(t, y1, '-', t, y2, '--', t, y3, ':')
```



More properties

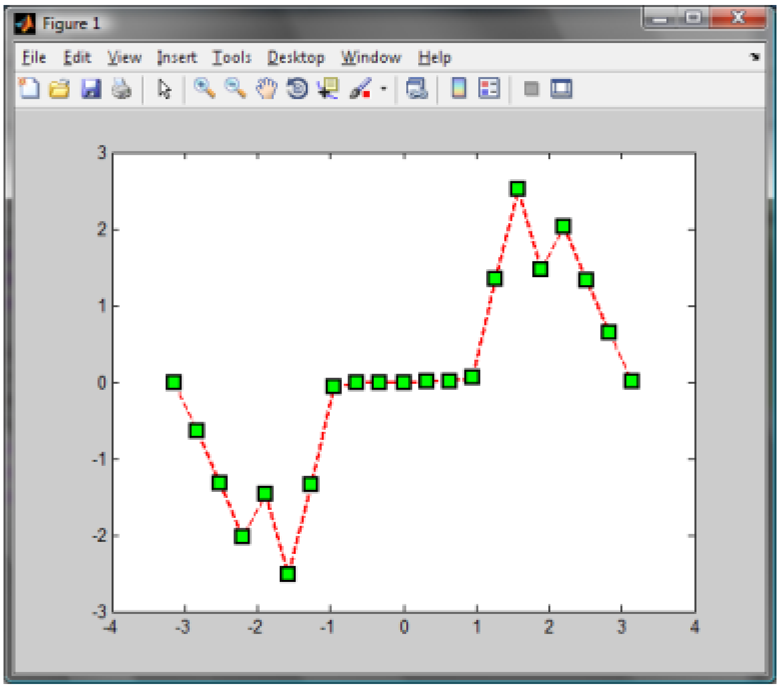
Other properties of lines and markers can again be influenced by means of **pairwise arguments**:
plot(..., 'PropertyName', PropertyValue,...)

- MarkerFaceColor - fill colour of markers
- MarkerEdgeColor - edge colour of markers
- MarkerSize - size of the markers
- LineWidth - width of the line
- Color - line colour

Example:

```
>> clf
>> x = -pi:pi/10:pi;
>> y = tan(sin(x)) - sin(tan(x));
>> plot(x, y, '--rs', 'LineWidth', 2,...
'MarkerEdgeColor', 'k', ...
'MarkerFaceColor', 'g', ...
```

'MarkerSize', 10)



Other options can be found with `doc plot`.

Bar Graphs

Another often used type of graph is the bar graph. The two-dimensional form with vertical bars is called `bar`, `barh` plots horizontal bars, and `bar3` and `bar3h` do the same, but with three-dimensional bars.

Examples for **two-dimensional bar graphs:**

>> figure	one data column
>> data=rand(10,3)*20+5;	different bar width (the default is 0.8)
>> bar(data(:,1))	re-label x axis
>> bar(data(:,1),0.5)	several sets of data
>> bar(10:10:100,data(:,1))	
>> bar(data)	

Examples for **3D bar graphs** in various variants. Copy all these lines at once into the Command Window.

```
Y = cool(7);
subplot(2,3,1)
bar3(Y,'detached')
title('Detached')
subplot(2,3,2)
bar3(Y,0.25,'detached')
title('Width = 0.25')
subplot(2,3,3)
bar3(Y,'grouped')
title('Grouped')
subplot(2,3,4)
bar3(Y,0.5,'grouped')
title('Width = 0.5')
subplot(2,3,5)
bar3(Y,'stacked')
title('Stacked')
subplot(2,3,6)
bar3(Y,0.3,'stacked')
title('Width = 0.3')
colormap([1 0 0;0 1 0;0 0 1])
```

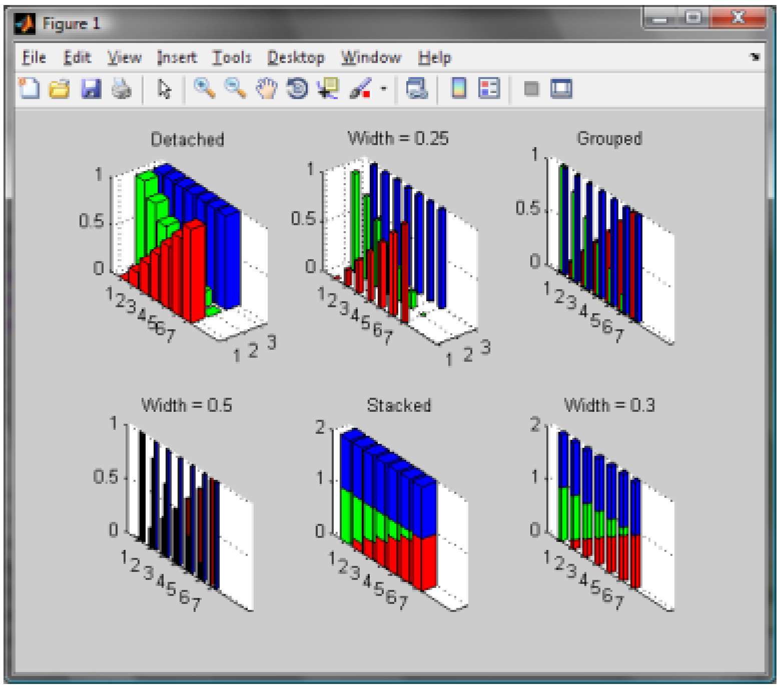


Figure 1
File Edit View Insert Tools Desktop Window Help
Detached
Width = 0.25
Grouped
Width = 0.5
Stacked
Width = 0.3

Brief Demonstration of Other Functions

Below, a script is used to illustrate some more of Matlab's interesting graphics options.

Please go to the Matlab Editor, load the script **graph_examples.m** and run it by hitting the **F5 key**. With a key press or mouse click into the figure window (function `waitforbuttonpress`) you can continue to the next graph.

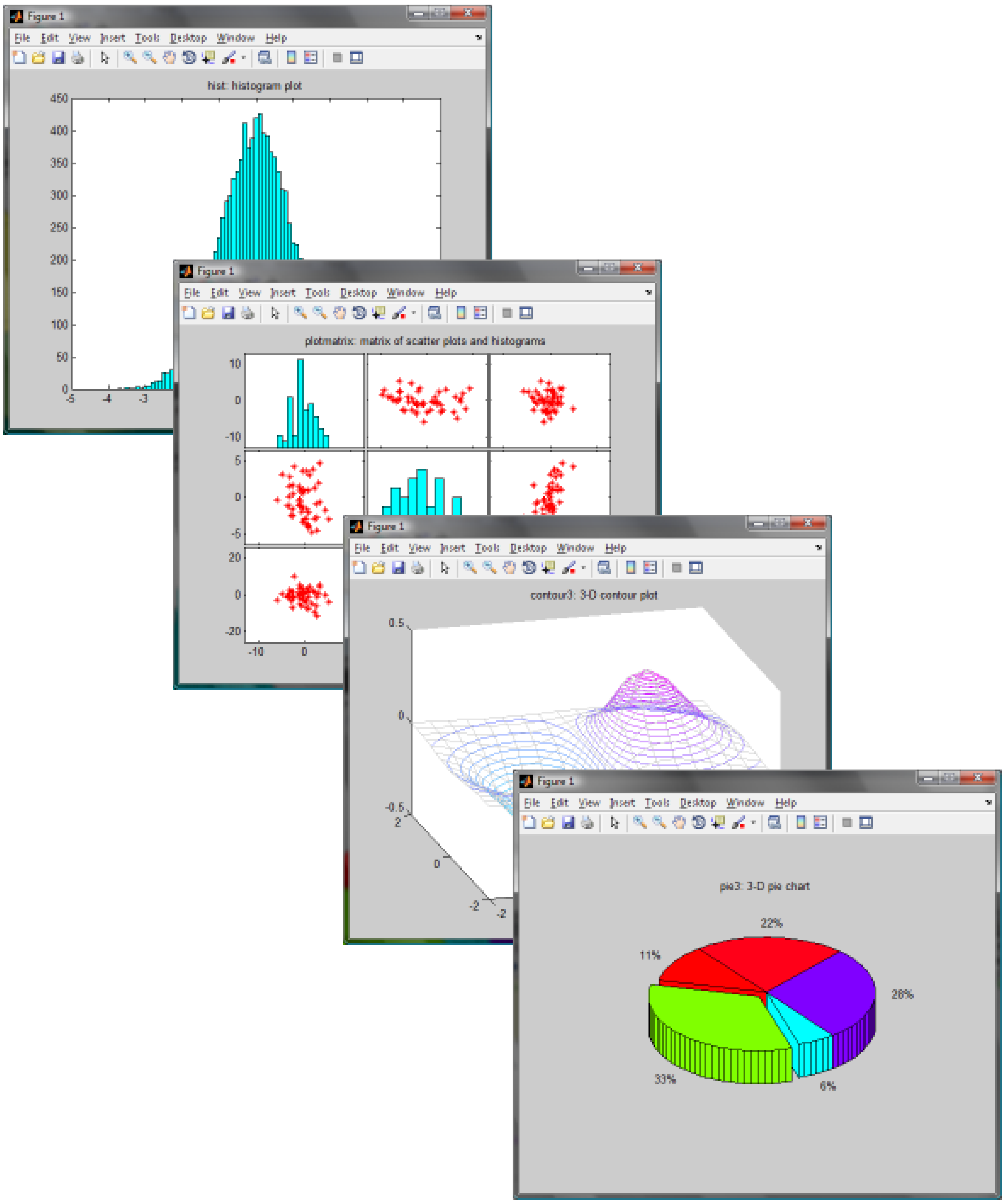


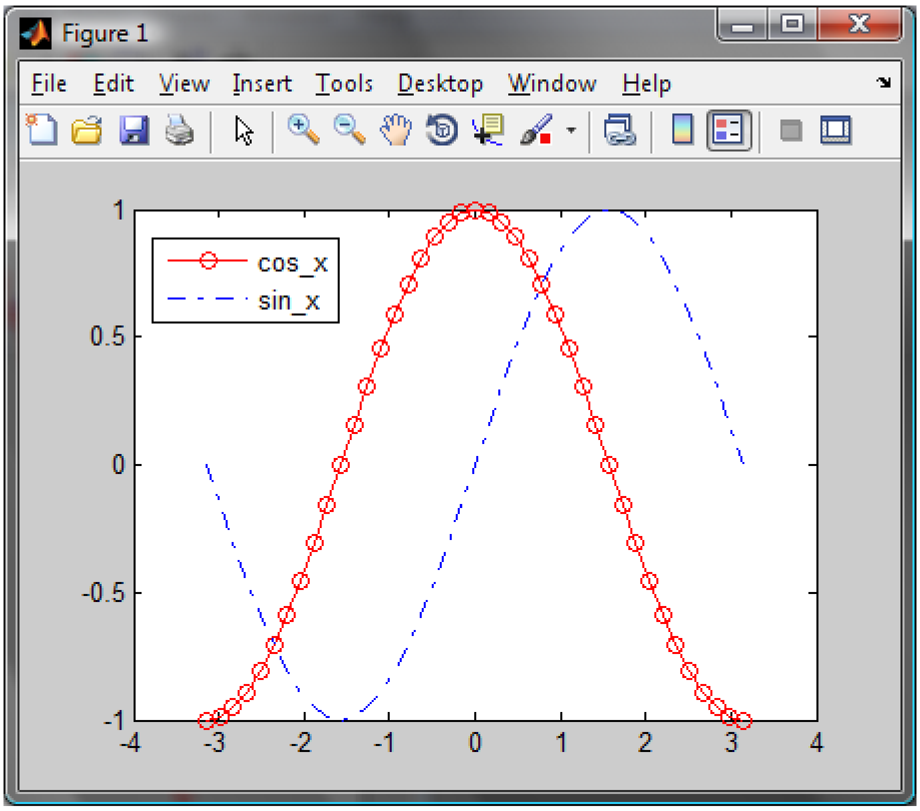
Figure 1
File Edit View Insert Tools Desktop Window Help
hist: histogram plot
plotmatrix: matrix of scatter plots and histograms
contour3: 3-D contour plot
pie3: 3-D pie chart
22%
11%
28%
33%
6%

Annotate and Extend Graphics

Usually it makes sense to complement graphics with additional captions, arrows, lines, or legends. Below you get an outline of the most important functions.

Legend
The function `legend` adds a legend to a graph. The most simple form of the statement is:
h = legend('string1', 'string2', ...)
h is the «handle» of the legend, the strings are the text that describe the different curves. Example:

```
>> x = -pi:pi/20:pi;
>> plot(x, cos(x), '-ro', x, sin(x), '-.b')
>> h = legend('cos_x', 'sin_x', 2);
>> set(h, 'Interpreter', 'none')
```



Explanation: The last line relates to the text interpreter used by Matlab. By default, it is TeX - with the consequence that - for instance - an underscore _ in the text is interpreted as a modifier that shows the following character in subscript. As this is not desired here, we disable the text interpreter. For further details, please refer to the Help System.

A nice feature: In the graphics window, you can grab the legend with the mouse and move it around freely, in case it hides relevant parts of the graph. Afterwards, you can save the graph.

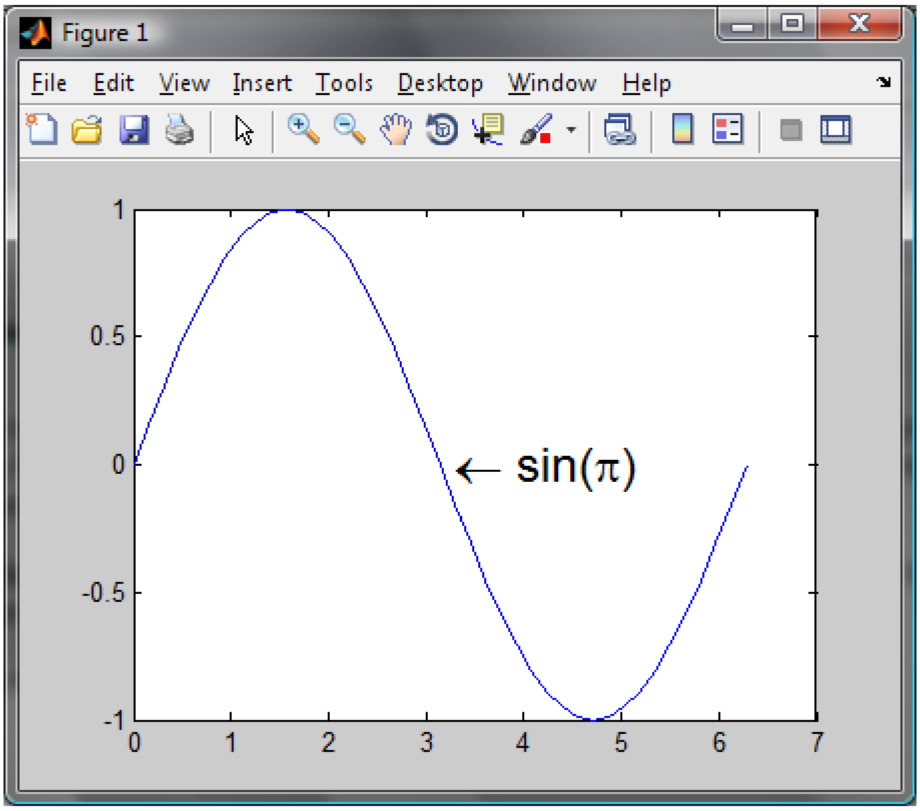
The `legend` function has innumerable further options, cf. the Help System.

Annotate graphs with text

By means of the function `text`, you can place text anywhere in the current figure. The coordinates for text placement correspond to the data coordinates (i.e. as the scales indicate).

Below you can see how the TeX interpreter can be used to display a formula. With an additional parameter, the text size is set.

```
>> plot(0:pi/20:2*pi, sin(0:pi/20:2*pi))
>> text(pi, 0, ' \leftarrow sin(\pi)', 'FontSize', 18)
```



Again, it has to be pointed to the Help System, as a complete description of all text options would go beyond the scope of this chapter.

Geometrical Elements

There are two elements that can be used to directly draw into a figure. Again, the coordinates are the same as the axes coordinates.

Lines

The `line` function serves to draw lines into the figure. The basic form of the function for two-dimensional applications is:

line(X, Y, 'PropertyName', propertyvalue,...)

X and Y are vectors specifying two or more coordinates that will be joined by straight lines. The properties are again given as argument pairs. Example:

```
>> line([0 1 .5], [0 1 .8], 'LineStyle', '--', 'Color', 'blue', 'LineWidth', 5)
```

Rectangles and ellipses

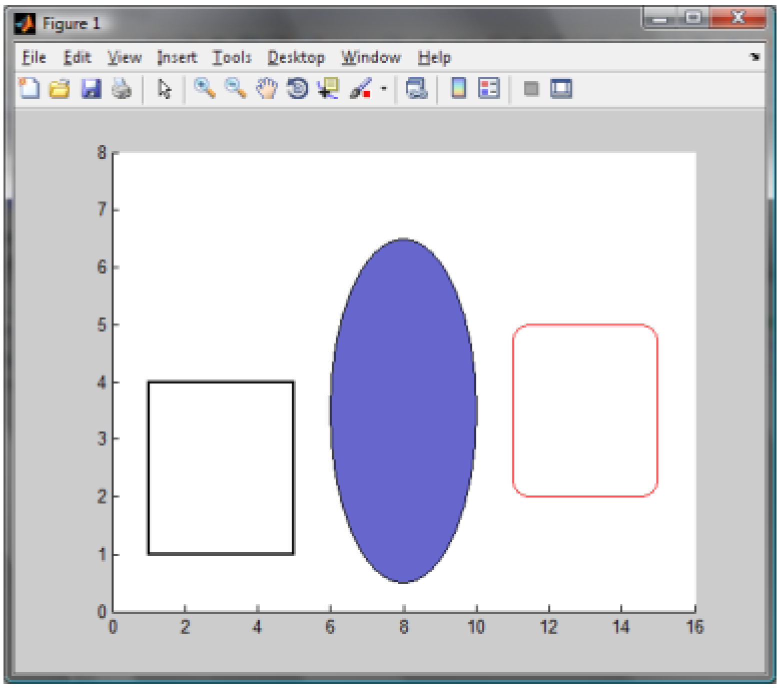
With the function `rectangle`, rectangles (also with rounded corners) and ellipses can be constructed. These shapes can be filled with colour.

rectangle('Position', [x,y,w,h], 'PropertyName', 'PropertyValue', ...)

x and y are the coordinates of the bottom left corner, w and h are width and height of the rectangle. With the property **Curvature** the corners can be rounded.

Some examples:

```
>> figure, axis([0 16 0 8])
>> rectangle('Position', [1 1 4 3], 'LineWidth', 2) % default:schwarz
>> rectangle('Position', [11 2 4 3], 'Curvature', .2, 'EdgeColor', 'red')
>> rectangle('Position', [6 .5 4 6], 'Curvature', [1 1], 'FaceColor', [.4 .4 .8])
```



Annotations

Annotations are another possibility to add texts and geometrical elements to figures. As with the above introduced functions, annotations can be used to draw text, lines, rectangles, and ellipses, plus the additional element arrows/text arrows.

Annotations differ from `text/line/rectangle` in that the coordinates for placement are not axis coordinates, but «normalised» coordinates, i.e. they lie between 0 and 1 for the whole **figure** (not the axes). 0,0 is the bottom left corner, 1,1 is the top right one.
Annotations are drawn on a layer of their own. It will not be changed even when the graph's scales are changed or deleted.

All annotations are applied with the same function `annotation` and the necessary parameters. The basic form is:

annotation(type, parameters)

The following types of annotations exist. x and y are the normalised coordinates, w and h are width and height (also normalised).

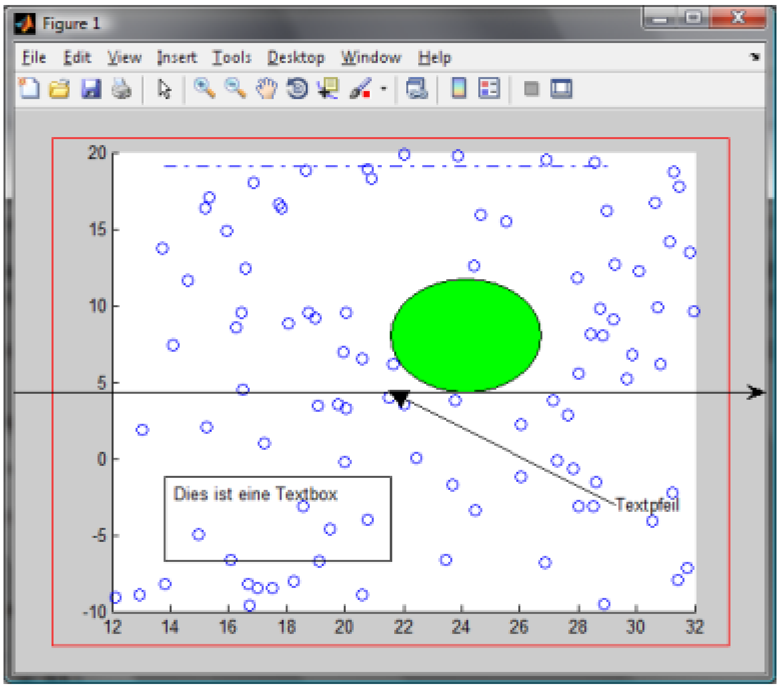
annotation('line', x, y)

annotation('arrow', x, y)
annotation('doublearrow', x, y)
annotation('textarrow', x, y)
annotation('textbox', [x y w h])
annotation('ellipse', [x y w h])
annotation('rectangle', [x y w h])

Examples to try out:

```
>> figure, scatter(rand(100, 1) * 20 + 12, rand(100, 1) * 30 - 10)
>> annotation('rectangle', [.05 .05 .9 .9], 'EdgeColor', 'r');
>> annotation('line', [.2 .8], [.9 .9], 'LineStyle', '-.', 'color', 'b');
>> annotation('ellipse', [.5 .5 .2 .2], 'FaceColor', 'g');
>> annotation('arrow', [0 1], [.5 .5]);
>> annotation('textarrow',[.8 .5],[.3 .5],'String','Text arrow','HeadStyle','plain');
>> annotation('textbox', [.2 .2 .3 .15 ], 'String', 'This is a text box');
```

Remark: `annotation` always returns a handle, that's why a semicolon is necessary at the end of the line to avoid its output to the Command Window.



Exercise P5: Project Task - Graphical Data Display

Implement in your project the graphical output of the data. Corresponding to your project's needs, line graphs, bar graphs etc. can be necessary. Often it makes sense to automatically save the graphs to the hard disk drive (to a fixed location, or a location specified by the user).

For the EyeData Processing Project

- Create a bar graph that displays fixation duration per condition (= image size), over all subjects; use the median values you have already computed. Add a legend and error bars with the standard deviation.
- Draw a scatter plot of all fixations for all image sizes. With `rectangle`, mark the image area as shown below.
  The corner coordinates are: (startx, starty, stopx, stopy)
  Size 1: full size 1600 x 1200
  Size 2: 196, 148, 1402, 1052
  Size 3: 385, 289, 1213, 909
  Size 4: 571, 429, 1028, 771
- Store the graphs to the hard disk drive, either to a fixed location with a pre-defined file name, or to a location and with a file name selected by the user.

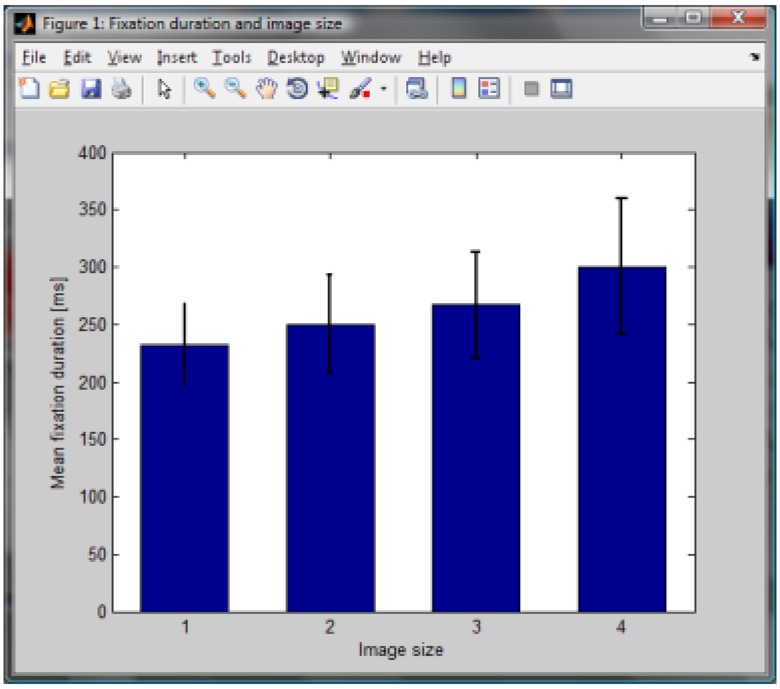
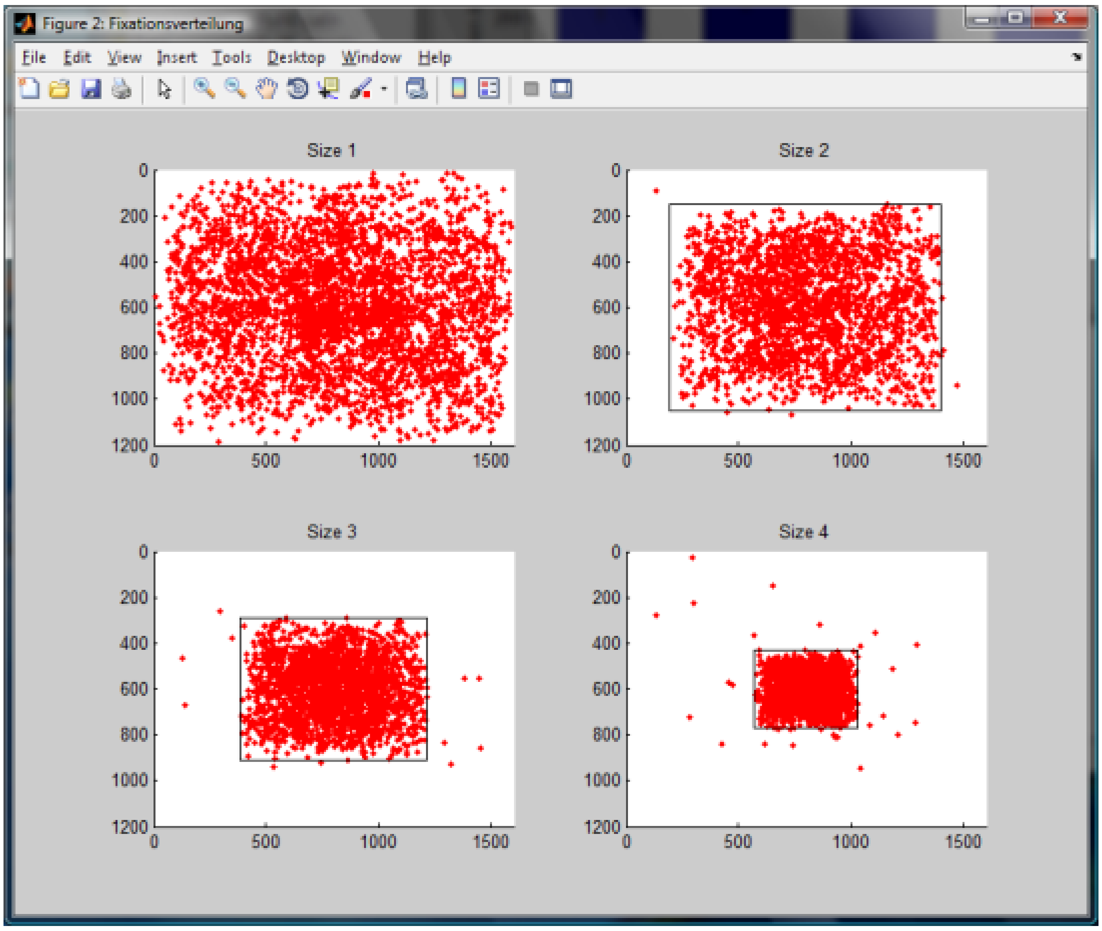


Figure 1: Fixation duration and image size
File Edit View Insert Tools Desktop Window Help
Mean fixation duration [ms]
Image size
400
350
300
250
200
150
100
50
0
1
2
3
4

That's how the bar graph should look like



Example for a scatter plot of fixation data per image size

You can find the sample solution in the M-file **eyedata_sol5.m**

Summary

Summary of the most important commands:

- `plot`: plot data
- `bar, barh, bar3, bar3h`: bar graphs; normal, horizontal, and 3D, 3D horizontal.
- `legend`: show the legend
- `text, line, rectangle`: draw texts, lines, or rectangles (and ellipses) in the current axes of a figure
- `annotation`: draw graphical elements in relative (normalised) coordinates of the whole figure, e.g. `annotation('rectangle', [.05 .05 .9 .9], 'EdgeColor', 'r');`

Final Remarks

You have now reached the end of this course. We hope that it has helped you to start with Matlab. We are fully aware that learning a new programming language is a very demanding undertaking - especially if you are new to programming.

Only pictures can be viewed in this version! For Flash, animations, movies etc. see online version. Only screenshots of animations will be displayed. [link]

Thus, we are very much interested in improving this course. If you have any comments, suggestions, or if you have found errors, please let us know!

If you would like to support our initiative to build this course, which is an open educational resource, and are a native speaker of English: The course is in need of proofreading, as it was translated by a non-native speaker of English (as you sure have noticed).
If you are willing to proofread a part (however small) of this course, please contact **me**.
Thank you very much!