Exercises - general remarks

- This document contains programming tasks for the course. They are designed for Matlab Versions 8.4 (Release 2014b) and up. There are some differences between versions – they are notable, but mostly not critical, and will in some instances be pointed to. Demonstrations/instructions will be done with Release 2018a.
- m-file names containing solutions to the exercises start with an 's' and follow the numbering of the exercises (e.g. solution to exercise 12 in unit 5 would be named 's0512.m'). If more than one solution is given, the additional ones are labeled 's0512b.m', 's0512c.m', etc. Functions created as part of the solution are located in the *func* directory.
- If data need to be loaded for the exercise, the file name(s) are given *in italics* in the text or at the bottom (in which case they are in parentheses).
- For a part of the exercises help in the form of hints to useful/needed functions is given, either in the text body, or in square brackets [] at the bottom, or both.
- Fixed-width font is used for function and variable names like `repmat` or `d`.
- The majority of the exercises build up on each other. Useful/needed functions are listed in the first exercise they are needed, and in the following one or two; after that it is assumed that you are aware of their existence.
- If the rationale behind an exercise is not clear at first, it may become so after taking a look at the solution(s). A substantial number of these contain comprehensive comments, explanations as to the usefulness of the code produced, background information, and the odd hint towards material not covered during the course.
- Characterization of exercises: exercises marked with an asterisk (*) may be – depending on your experience with Matlab or programming in general – tricky. Exercises marked by ⅄ (a pyramidal cell) have a more or less direct bearing on neurophysiology. Finally, exercises marked with a ∑ are designed to introduce or reinforce aspects of Matlab which are useful for mathematics. They are as a rule NOT mathematically demanding!
- If you find inconsistencies, ambiguities, errors etc. in the exercises and/or solutions, please let me know

Abbreviations, synonyms

1. Matlab jargon and terms used in signal processing

scalar	a single number, e.g. `6`
array	a container holding more than one element, e.g. `[4 99]`. It is the elementary data type in Matlab. ***Although the term suggests only one dimension, anything termed an 'array' in Matlab may be 2-, 3- or higher dimensional***.
matrix	usually a 2-dimensional array, but often ***array and matrix are used interchangeably***
vector	a 1-dimensional array like `[4 99 3]`
index	the position of an element within an array: in the array `[4 66 7]` the element `66` is at position 2, that is, it has index 2. In a 2-dimensional array, we have a row and a column index. Row and column indexes of element `66` in the array `[4 66` / ` 7  1]` are 1 and 2, respectively. In higher dimensions conventions apply accordingly.
sampling	

frequency	the rate at which a signal was sampled (and digitized). For example, audio data often have a sampling frequency of 44100 Hertz, corresponding to 44100 data points per second.
sampling interval	the time interval between two sampled points (=the inverse of sampling frequency). In the example above the sampling interval is 1/44100 s = ca. 22.7 μs.

Sources of information about Matlab

Internet

The website of The MathWorks is the primary source of up-to-date information:
http://www.mathworks.com/

Annoying as it may be to have yet another user account at some site, *I recommend you to register with TheMathworks* because otherwise you will not be able to interact with their website in a fruitful way. For example, you won't be able to download material such as the highly recommended manuals accompanying Matlab. For registered users, they are freely available and provide extensive information. Particularly the Matlab Primer (formerly known as 'Getting Started with Matlab') is a very good starting point. Some third-party documents helpful for the course are in \mcourse\doc.

If you are a complete novice in Matlab, you may want to check out Matlab **Onramp** at the Matlab Academy, a self-paced online course worth ca. 2 hours which takes you through some basics of Matlab. Of course, there is a great overlap between Onramp and Introduction to Matlab, but if you anticipate that programming is not entirely your cup of tea having the opportunity to practice at your own pace is certainly a good idea. There is also a host of other basic and specialized courses on offer.
Link to Matlab Academy:
https://matlabacademy.mathworks.com/

The file exchange and newsgroup site may be useful if you are looking for a solution to a problem someone else is likely to have solved already
http://www.mathworks.com/matlabcentral/

You may also sign up for the Matlab Digest, a newsletter sent out via mail on a bimonthly basis, or find the webinars on various topics useful.

Books

The Mathworks site
http://www.mathworks.com/support/books/

lists quite a few of the ever-growing volume of books available on the subject.

I have a few recommendations for books on Matlab (based on personal experience):

- Hanselman DC, Littlefield B. Mastering Matlab, Prentice Hall, 2012, or the previous edition from 2005 (available in the library of the Graduate School). A comprehensive overview and reference. The book also has a web site with example files.

- Higham DJ, Higham NJ. Matlab Guide, 3rd edition, Philadelphia: SIAM; 2017. Succinct, elegant; also material for the more mathematically oriented user.
- Johnson RK. The Elements of Matlab Style, Cambridge University Press, 2011. A collection of numerous helpful and insightful guidelines and standards for producing intelligible and well-behaved code. The open-access document on which this book is based is \mcourse\doc\MatlabProgramStyleGuide.pdf
- Altman, Y. Accelerating MATLAB Performance: 1001 tips to speed up MATLAB programs, Chapman and Hall/CRC, 2014. Definitely intended for advanced and dedicated users, this is a trove of tips to...well, see title

Units 1a & b

Topics: generation, concatenation, indexing of arrays & simple arithmetic computations

1.0
Enter five arbitrary numbers between 1 and 10 into an array. Imagine that these numbers represented the cap diameters of some edible mushrooms you collected, expressed in inches. Convert the heights to centimeters (1 inch = 2.54 cm). After doing this, imagine you realized that the second mushroom had not been measured correctly, but that 0.3 inches are missing. Correct the list of cap diameters for this.

1.1
Clear the workspace and load data file *d01.mat* via `load d01`. From array `a1`, generate array `b1` consisting of the 1st, 3rd, 5th... column. Do this in two different ways.
(d01.mat)
`[clear,load]`

1.2
From `a1`, generate several other arrays according to the following rules:
`b2`: 2 copies of `a1`, horizontally concatenated
`b3`: 2 copies of `a1`, vertically concatenated
`b4`: the last, the second and the first row of `a1`, in this order
`b5`: 50 copies of the third column of `a1`, vertically concatenated
(d01.mat)
`[repmat]`

1.3
Generate an 80 by 80 array `stripes` in which the first 10 columns consist of zeroes (=elements of value 0), the next 10 columns consist of ones, the next 10 columns again of zeroes and so on. Verify your result using `spy`. This function opens up a plot, which represents the array, and places a marker (a dot) on the plot for each non-zero element at the element's position within the array. Next, multiply stripes by 15.4 and re-`spy` it. Does the appearance of the plot change? What if you add 2 to all even (2nd, 4th, 6th,..) columns?
`[zeros,ones,repmat,spy]`

1.4
Generate a matrix which – viewed by `spy` – has a checkerboard pattern (a regular checkerboard has 8 by 8 fields). Do this using `stripes` generated above.
`[rot90]`

1.5
In array `a1`, delete the upper right 'corner'. Here, we define a 'corner' as a 2 by 2 subset of the whole array:

				*	*
				*	*

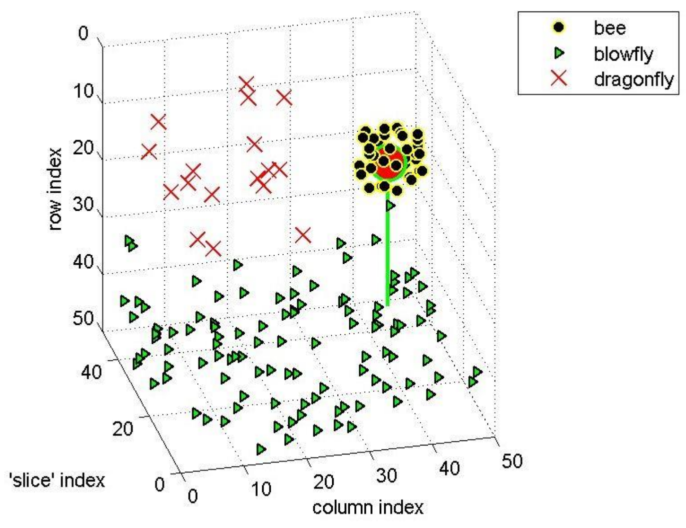
Why does it not work in the suggested way? How would you 'get rid of' the four upper right corner values in the array?
`[size]`
(d01.mat)

1.6 (∑)
Take a look at `a2` in *d01.mat*. Exchange the lower left and lower right corners of this array (a corner now is a **10 by 10** subset of the whole array). Then rotate the array by 90 degrees clockwise and inspect the result. Repeat this procedure several times. Does a stable matrix pattern emerge?
Note: there are several ways to code the exchange of the corners. If you feel comfortable with the Matlab style of indexing arrays, try accomplishing this in one statement, without generating a new matrix. If not, you may find the use of temporary variables and the function `size` helpful. At any rate, accessing portions of arrays is one of the most elementary things to accomplish in Matlab.

1.7 (∑)
Generate a multiplication table of all integers up to 10, that is, a 10 by 10 matrix containing the products of all possible combinations of the integers from 1 to 10. Again, there is more than one way of doing it. If you are familiar with linear algebra use just a row and a column vector, otherwise remember that arrays may be rotated. In either case, knowing the difference between the operators '`.*`' and '`*`' will most likely help.

1.8 (*)
Welcome to an insect party. Load variable `buzz` from *d04.mat* and look at the scene using `buzzplot(buzz)`. The bees busily hover around the only flower in sight (back in the far right). The blowflies are of course all over the place in the lower strata, whereas the dragonflies take up the airspace on the upper left, anxious not to get too close to the bees. Make sure you grasp the spatial arrangement of the partygoers by rotating the plot along the various axes (in the figure's toolbar there is a button which puts the mouse pointer into 3D rotating mode). Now, your task: mix the party up! More precisely, manipulate `buzz` in the following ways:
i) The blowflies shall occupy the upper strata (row index 1-20) and the dragonflies shall hover in the lower left area (row index 31-50), while the bees as a group remain where they are (the individuals may change their position). In other words, the scene created by you shall be a mirror image of the original scene, the mirror plane being the horizontal plane.
ii) The whole insect scene except the flower shall rotate 90 degrees clockwise around a vertical axis. That is, the different insect groups remain in their strata (row indexes), but they swirl around. The bees move to the front right, the dragonflies to the back, sparing the upper right

corner, and the blowflies as a group don't change (again, the individuals' positions will most likely change).



An entomologist's rendition of a 3D array

iii) Now, really mix it up. No group shall be as coherent anymore. This is an open task, there is no single 'right' solution, no single right way of doing it. Experiment!

Notes:
a) This is a task involving manipulations of a 3D variable, `buzz`. `buzz` has 50 rows, 50 columns and 50 'slices'. `buzzplot` is a 3D version of the `spy` function, visualizing the content of 3D arrays in a special way. It interprets the row, column and slice index of each element as that element's position in space. Elements of value `1` code for bees; `2` corresponds to blowflies and `3` to dragonflies. Any other value is ignored. Realize that the axis pointing upwards (labelled 'row index' in the plot) is inverted: values increase as you move downwards. This is because in Matlab the row index increases as you 'go down' the columns (take a look at the output of spy and you will see the same thing).
b) The 'flower' is not contained in `buzz`. It is absolutely stationary, no matter what manipulations you perform on `buzz`. It is meant to give you some visual grounding, a fixed point of reference when rotating the plot.
c) Some of the operators/functions you know already, like the transpose operator (`'`), `rot90`, `fliplr` etc. either work only in 2D or work only in a restricted way on 3D and higher-dimensional variables. Among the functions useful for rearranging arrays with three and more dimensions are `permute, ipermute, flip, shiftdim, reshape` and `circshift`.
d) Admittedly, this task is a little concocted. However, some real tasks (e.g. importing binary data from other applications and efficiently using them within Matlab) require a good deal of rearranging arrays in more than two dimensions. Think of imaging data or multi-channel, multi-sweep electrophysiological recordings.
(d04.mat)

1.9 (*)
Imagine yourself in a deep valley. Whenever you shout anything, you will hear your own voice and then, with a delay, the echo, which is basically the same acoustic signal but less loud, and maybe even multiple echoes to follow. Try to simulate an echo effect with the following ingredients:

cpf.wav or *cow.wav*	- wav files (in directory 'data')
`audioplayer, play, playblocking`	- Matlab functions dealing with audio files

Notes:
a) The wav files can be loaded with

```
    [y,fs]=audioread('d:\mcourse\data\cpf.wav');
    % cow file accordingly
```

which gives you column vector `y` representing the sound, and `fs`, the sampling frequency. The values in `y` are bounded by `-1` and `1`.
b) In developing a solution you will most likely play around extensively with several parameters, e.g. the time delay between the original sound and the echo, and the attenuation of the echo. Try to program in such a way that experimenting with parameters is made as easy as possible. For example, if you decide that the echo shall be attenuated by a factor of `0.4`, generate a variable (e.g. `att`), assigning `0.4` to it in a prominent place within the script (at the top, for example) and use this variable instead of typing `0.4` in the computations to follow. This will potentially reduce typing work, it makes complex scripts much more readable and, most importantly, prevents errors stemming from inconsistent assignment of values to variables.
`[round,max,min]`

Unit 1c

Topics: logical data type, logical indexing, relational operators
*So far, accessing portions of arrays had been explicit and static: in e.g. 1.6 the indexes into the elements defined as 'corners' were set arbitrarily and did not depend on the values of the elements. However, particularly in data analysis tasks there are numerous situations in which the treatment of an array depends on its contents. For example, in an array representing the volume of milk (in liters) produced by a typical US Midwest cow each day we would not accept numbers larger than 10 because it is unrealistic that a cow produces more than 10 liters of milk per day (we would put the blame on the milking robot). In this situation, if our goal were the computation of the mean value (liters/day), the treatment of the array would consist of 1. locating elements the value of which is >10, 2. eliminating these elements, and 3. computation of the mean value. The following exercises are primarily concerned with variations of step 1; this type of computation is termed **logical indexing**. Its practical relevance in data analysis in Matlab cannot be overemphasized*

1.10 (*)
Consider

```
a=[6 7 8];
x=[1 0 1];
```

Generate

```
y=logical(x);
```

x and y look pretty much alike.
Explain why, nevertheless,

```
b=a(x);
```

results in an error (which should not be too much of a surprise by now) whereas

```
b=a(y);
```

is fine.
Hints: inspect the data type of `x` and `y` (`whos`) and read the output of `help logical`

1.11
Consider

```
      a=[6 7 8];
      x=[1 1 1 1];
      y=logical(x);
```

Explain why now

```
      b=a(x);
```

is fine while

```
      b=a(y);
```

results in an error.

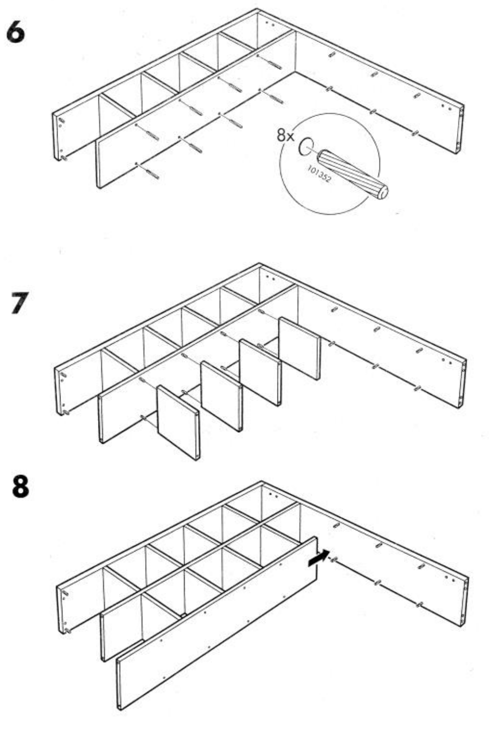
1.12 (*)
Try to find examples of row vectors `x` which change *neither type nor value* when subjected to the following expressions (one at a time)

```
      x=x(x);                    % expression 1
      x=x(logical(x));           % expression 2
      x=x(~x);                   % expression 3
      x=logical(x);              % expression 4
      x=~x;                      % expression 5
```

For the sake of typing economy, `x` should have 4 elements, no more.
How many examples can you find for each expression?
Notes:
a) the '`~`' is an operator for the logical data type, it is termed negation. `~1` is `0` and `~0` is `1`. Negation as well as other 'relational operators' will be explained upon typing `help relop`.
b) in older Matlab versions you will probably get warning messages like

```
    Warning: Values other than 0 or 1
converted to logical 1
```

when trying various solutions for expressions 2-5. This is fine because this is exactly what has to happen (unless we restrict solutions to arrays just containing ones and zeroes).



1.13
Solve 1.4 with the help of logical operator `xor`.

1.14
In Matlab, there is a handful of weird but useful numbers like `NaN` (not a number) and `Inf` (infinity).

An IKEArray

These may be identified via functions whose names start with '`is...`', for example `isnan`. Make yourself familiar with `isnan` and `isinf`; particularly the first will be needed frequently from now on. `isempty` is also quite important. For a complete list of `is...` functions open the help window and see what is listed under `is*`.
Then, take a look at array `milk1` which represents the volume of milk (in liters) produced per day by a particular cow over a period of 100 days. Time runs down the column. Find out on how many days
- the animal escaped and did not show up again so the volume was undefined (`NaN` entries)
- the volume was suspicious in other respects

`[sum]`
(d01.mat)

1.15
`find`, used in conjunction with the `is....` -functions or relational operators, is a very useful function for locating specific numbers within arrays. For example,

```
      x=[1 NaN 33 NaN];
      find(isnan(x));
```

returns `[2 4]`, namely the positions of the `NaN`s within array `x`.
Solve the task above using `find`.
(d01.mat)

1.16 (*)
`milk2` is similar to `milk1`, except that it has more rows and lists the daily yield of milk from three cows (accordingly, `milk2` has three columns). Find out which cow (=column)
- was most reliable (=had the fewest days with zero, NaN or other unsatisfactory output)
- had the largest mean daily output
- escaped for more than one day in a row (=had `NaN` as output on those days; there is only one cow that does that).

Furthermore, can you think of a way of identifying days of outstanding performance for each animal?
Note: The higher dimensionality of `milk2` versus `milk1` (2d versus 1d) poses a certain challenge at this point. Two remarks in connection with this:
a) Remember, once again, that Matlab by default operates along columns, not along rows. Read e.g. what `help sum` puts out in this regard. Play around with small toy matrices to make sure you understand the difference between e.g. `sum(milk2)` and `sum(milk2,2)`
b) If you use `find` to determine e.g. cow escape times, understand why a statement like

```
      daysEscaped=find(isnan(milk2));
```

will not be of much help, in contrast to

```
      [daysEscaped,cowID]=find(isnan(milk2));
```

or

```
      daysEscaped_cow1=find(isnan(milk2(:,1)));
```

(d01.mat)
`[min,find,diff,length or size,sum,mean,std]`

Unit 2

Topics: flow control (language elements testing for conditions & accomplishing repetition of tasks), preallocation, error handling

2.1

Write a piece of code which does the following:
a) determines whether an array `d` exists in the workspace
b) if it does not exist, issues an error message informing you about this; if it exists, determines whether `d` is empty
c) if `d` is empty issues a warning stating this fact; if `d` is not empty, determines the size of `d`, assigning the number of rows, columns and 'slices' to variables r,c and s, respectively;
d) if `d` is truly three-dimensional (`s>=2`) informs you about this fact; otherwise wishes you a nice day.
Test your piece of code with various examples of `d`.
`[if,exist,isempty,error,warning,disp,size]`

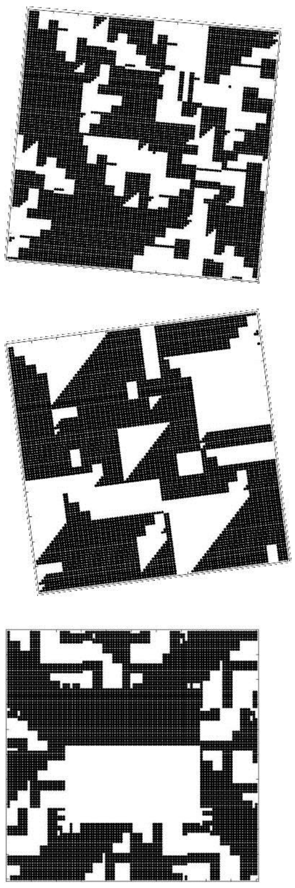
2.2

Write a script which acomplishes the following
- loads file *d01.mat*
- does some trivial calculation on `a1` (e.g. adds `1` to all elements of `a1`)
- tries (`try`) to save data in file *temp.mat* on the CD drive or a non-existent drive on your computer
- if that fails (which it will), does not produce an error, but saves the data in the current working directory of your machine and informs you about the mishap.
`[try,warning]`

2.3

Repeat 1.7 by calculating the products of the integers with nested `for` loops. The result should be the same, that is, a square array with all values properly ordered must be generated in the process. Once this is done, place the command `tic` in the line before the first `for` and place the command `toc` in the line directly below the matching `end`. This will time the process and automatically put out the time required for the computations on the screen. Next, set the number of integers to `N` (see notes below), run the code *repeatedly* and see how long each run takes to execute. Then, time one of the original solutions found for 1.7 with the same number of integers (again, run repeatedly). Which code is faster?
Notes:
a) Do not clear the resulting variables in between your test runs and, for now, ignore the fact that the first run of the `for` loop may take substantially longer than the following ones. You will soon (2.4) see why.
b) Set the number of integers to a value which keeps the computer busy for a second or so solving this task. On a modern desktop PC `N=8000` is a good starting point.
c) understanding this and the following exercise is important if your interest lies in producing fast code in Matlab.

2.4
Include into the script produced in 2.3 another nested loop. In fact, this loop should be an exact replica, including the `tic toc` commands, of the loop which is already in the script. Now, make sure that the array produced in the first of the loops does not exist before it is executed (in other words, clear the variable if it exists in the workspace). For the second loop, the contrary must be the case: make sure the array *exists in a pre-fabricated form, either as the result of the first loop or as a properly sized matrix of zeroes (the elements of which will be replaced by the results of the computations)*. Execute both loops repeatedly. Which one is faster? Can you explain the differences in speed?


Flavors of 2.7

2.5 (*)
Imagine a tiny particle moving randomly in one dimension (along a single axis). Simulate the movement of the particle by assigning to it new positions in a step by step fashion. Details:
- the distance which the particle covers in each step should be drawn from a normal distribution (which `randn` provides you with)
- the particle starts its path at coordinate `0`
Execute as many steps as needed for the particle to fulfil a certain condition (Q: Which of the loop constructs in Matlab, `for` or `while`, lends itself to tasks like this?). Incorporate two conditions between which you can easily `switch`:
1) the particle must reach a coordinate at least `lim1` away from the origin, `lim1=5`.
2) the summed path of the particle exceeds a minimum distance `lim2=30` (regardless of where it is at that moment).
Note: **this script will serve as the basis of a more elaborate program with which we will, in the units to follow, explore different data types, plots, etc. Stick to annotation b) of exercise 1.9 if you can – you will benefit from it. Also, include comments at least at critical positions so you (or someone else who reads your code) know what it is about.**
`[abs]`

The following exercises do not add new aspects related to flow control. However, if you appreciate seeing complex behavior in systems governed by simple rules you may enjoy the output of 2.7; also, practising never harms.

2.6 (*)
Write a piece of code which tests whether a particular row array `x` with four elements fulfils expression 1 in 1.12. Most likely you will need a `try catch` statement (why?). Once this is done, try to incorporate the other expressions in 1.12 in your code using a `switch` statement. If you want to display a message informing you about the outcome of the test together with the elements in x, something like

```
      disp(['x = ' int2str(x) ' fulfils x=x(x)']);
```

will do.
`[try,if,all,switch,islogical,disp,int2str]`

2.7
Embed the code you generated for1.6 into a `for` loop. As the last line before the `end` add the command `drawnow`, otherwise the plot will not be updated after each run of the loop. After convincing yourself that everything works fine, make the definition of corners (originally 10 by 10 elements) dynamic - for example, make the corners larger or smaller after each run, let one or both of the corners 'climb' up and down again. This adds quite some flavor to the game - try it!
Note: `mod`, the modulo function, may come in handy for this task.
(d01.mat)
`[mod]`

2.8
Type `edit why`. Up opens a file in which the `switch` construct is used throughout. Understand it, use it, give it a personal touch, if you like.

Unit 3

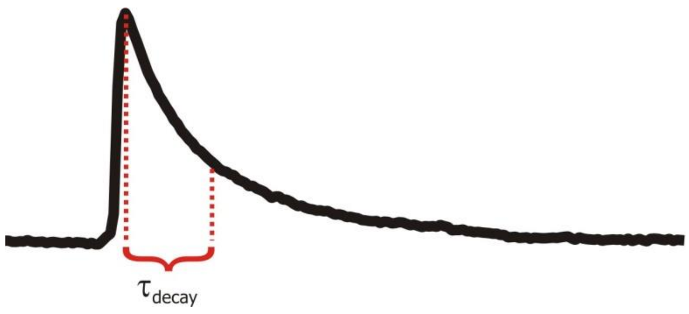
Topic: graphics

3.1
Include some graphics animation in the script produced for 2.5: after each position-computing step plot the position of the particle. Use `drawnow` to ensure that the figure is properly updated and `pause` code execution for 50 ms after each step (because otherwise things may move too fast for us to notice). Label the x-axis.
Note: *please work on exercise 3.5 to learn about optimizing animations in terms of speed!*
`[plot,xlabel,drawnow,axis,pause]`

3.2 (*)
Program a silly gambling game. There should be a figure with three big circles, one of which is associated with a big prize (think of the circles as three doors to chambers holding either the prize or nothing. The association of the prize with one of the doors should change randomly from run to tun). Each time the program runs you have one try. Use the mouse to click on one of the doors; if you got the right one, it should light up in green, otherwise 'GAME OVER' should appear in big letters across the figure.
Notes:
a) each plot has a property `'markersize'` which determines the size of its markers (=the symbols used for the plot), and `'markerfacecolor'` determining their fill color
b) solve the task using `ginput`. It is a function which provides you with the coordinates of the mouse pointer on the plot upon a click with the mouse button(s)
`[plot,set,ginput,text]`

3.3 (∑)
Load image file *littletown.jpg* (in the mcourse\data directory) using function `imread`. Let's call the resulting variable `d`. Visualize `d` (the picture) using function `image` and issue the command `axis image`. Then, take a close look at the structure of `d`. It has many rows and columns and three 'slices', or layers. While the rows and columns in the variable correspond to the rows and columns of pixels in the picture, the different layers code for the intensity of each pixel in red, green and blue, in this order. Try to create a somewhat surrealistic version of the picture in a new figure by exchanging the red and green layers of `d`. Next, depict the moustached dog (or is it a cat? Only the artist knows...). Do this in two ways. First, by zooming in on the animal in the first figure window. Use either the `axis` command or, preferably, the `set(gca,'xlim'...` syntax. Second, by creating a subset of `d` and visualizing it in a new figure window. Finally, try to overpaint the animal's nose (which we assume to be square) in the most pure and intense red possible. Hint: maximal intensity in each of the colors corresponds to a value of `255`, minimal intensity to `0`. If you are curious, try e.g. `histogram(double(d(:)),0:255)` for a quick overview of the distribution of intensities.
`[imread,image,set(gca,'ylim')]`

3.4 ()
Inhibitory postsynaptic potentials (IPSCs) arise from the release of inhibitory neurotransmitter at synapses. Their kinetics can be expressed in two numbers: the rise time and the decay time. We define rise time as the time needed for the current to shoot up from 10 to 90% of its peak amplitude; decay time is the time during which the current decays to ~37% of the peak amplitude (see illustration).



What goes up must decay (exponentially, for example)

IPSC.mat contains these kinetic data from a large number of IPSCs recorded from pyramidal cells in mouse hippocampal area CA1 (data courtesy of Matthew I. Banks, University of Wisconsin, Madison, USA). Your task is to employ some Matlab graphics functions to visualize the relationship between rise time and decay time of the IPSCs, from simple to more sophisticated.
Description of variables in *IPSC.mat*:

`tRise`	column array of the rise times
`tDecay`	matching column array of the decay times
`binRise`	bins for rise times
`binDecay`	bins for decay times
`h2`	2-dimensional frequency distribution of the IPSCs according to rise time (rows) and decay time (columns) using the bins above

Note: decay and rise times have been log-transformed. This is why the variables also contain negative numbers.

Your tasks:

- plot histograms of the rise times and decay times using function `histogram` (and/or `histcounts, bar`). You may either use the predefined bins (`binRise, binDecay`) or generate them yourselves.

- plot decay time versus rise time in a 'scatter plot' (using either the `plot` or `scatter` function)
- generate a surface plot of the 2-dimensional frequency distribution using `surf` and the predefined bins (`binRise, binDecay`).
- use `contour` and `bar3` for alternative 3-dimensional depictions of the data
- In all plots, make sure all axes are properly labeled and scaled.

`[histogram, histcounts, bar, plot, scatter, surf, bar3, linspace, xlabel, ylabel]`
(IPSC.mat)

3.5
Repeat 3.1 (including some graphics animation in the script produced for 2.5), but now make sure that within the loop only the bare minimum of graphics operations is performed. Specifically, plot the initial position of the particle before the loop, obtaining a handle to the marker representing the particle, e.g. via `ph=plot(....` Also, set axis labels etc. before the loop. In the loop, change the marker's position using the `set(ph,'XData',…` syntax. Notice any difference in speed compared to 3.1? See also exercise 7.2.

3.6
The standard appearance of plots in Matlab is a little boring. More importantly, the default marker sizes, line widths, font sizes etc. are a compromise - while OK for single standard portrait plots they may appear quite oversized as soon as figures get crowded with more than just a few subplots. Write a script which, after having been called once, changes the standard appearance of plots in a specific way, e.g. by producing small markers and fonts, etc., such that 20 subplots in a figure to be printed on a standard page (A4 or USletter size) in landscape orientation look OK. Test your code with any kind of data.
`[set(groot,'default'),get(groot,'default')]`

Unit 4

Topics: specific data types (character arrays, structures, cell arrays, tables)

4.1
The command
`h=help('datestr')`
generates variable `h`, a character array containing the help text of function `datestr`. In previous versions of Matlab there was a typographical error in it – instead of 'Tuesday' it said 'Tueday'. Alas, the makers of Matlab have found and eliminated it. So, instead of correcting an error we will now mischievously re-introduce one. Locate the passage. More precisely, find out in which line

Entf
Ende
⚠WICHTIGER HINWEIS
Für den sicheren und angenehmen Umgang mit dem Gerät, lesen Sie bitte das Handbuch für sicheres und angenehmes Arbeiten.
DP
H1, hardware style

of the help text 'Tuesday' occurrs with the help of `regexp(h,'\n')`. Replace 'Tuesday' in variable `h` by e.g. 'Tomato' using `strrep`.
Note: `h` contains 'line breaks', that is, special characters which cause a line break if you display the contents of `h` on the screen. `regexp(h,'\n')` will provide you with the position of these line breaks within `h`.
`[datestr,strfind,regexp,strrep]`

4.2
Generate a structure `people` with fields `name`, `shoe_size`, and `favourite_color`. First, fill in the fields with your own data. Then, ask your nearest neighbours for the same details of their personality and accomodate these data into `people` (this will necessitate expanding the single-element structure to a structure array). Then, write some code which puts out the names of people whose shoe size is within a certain range. Can you accomplish this in a single command?

4.3
You know by now about the benefits of preallocating large variables. Repeat exercise 2.4, placing the numbers into a cell array instead of an array. As in 2.4, do this with and without preallocation of the cell array. Is it worth preallocating a cell array? Is usage of a cell array in this particular task of any help?

4.4
Ernie and Bert have an ongoing dispute as to the frequency of vowels in the English language. While Ernie contends that 'e' is the most frequent vowel in any language with Latin roots, Bert is convinced that 'a' appears at least as often as 'e'. See who is right by counting the occurrences of letters 'a' and 'e' in an assortment of limericks.
Notes:
a)
```
fid=fopen('c:\mcourse\data\limerick.txt');
limericks=textscan(fid,'%c','commentstyle','matlab');
fclose(fid);
```

will read the text file except for the comments in the upper four lines and put out variable `limericks`, which contains a single-column char array containing the whole text. By contrast,

```
...
limericks=textscan(fid,'%s','delimiter','\n',...
'commentstyle','matlab');
...
```

will result in `limericks` containing a cell array with one element per line of text. Which way of reading the data is better for the task, which is best for displaying the poetry on screen in a readable format?
b) The contents of *limerick.txt* are the result of a contest initiated by the American Physical Society. They can be can be found in the March 1997 issue of APS News (Vol. 6, issue 3), or at https://www.aps.org/publications/apsnews/features/limericks/
`[textscan,findstr,lower,length]`

4.5 ()

Structure `spx` in *d03.mat* contains data from a real electrophysiological experiment in which spikes had been recorded from several electrodes in the somatosensory cortex of a rat. Two of the numerous fields of `spx` are interesting for this task, `name` and `data`. `name` holds the names of the recorded channels, `data` contains the corresponding time stamp lists (of spikes, unit is milliseconds) in the same channel order as in `name`.
- find out of which data type `name` and `data` are
- try to generate an inter-spike-interval histogram of the channel termed 001a
`[diff,histogram,histcounts,bar]`

4.6
Generate the shoe data base (exercise 4.2) using data type `table`.

Unit 5

Topics: file I/O, data types continued
This unit should be of interest to those who plan on routinely importing and exporting data in/out of Matlab

5.1
Load variable `milk2` from *d01.mat*. Then, write it back on disk in the following formats (a new file for each format):
- Matlab (*.mat)
- ascii, 8 digits
- ascii, 16 digits
Which format takes the least disk space? What is the (approximate) number of bytes per data point needed to store the data in the specific format? Can you think of a more economic way of storing them? The first lines of output produced by `help datatypes` may help.
`[load,save,int8,uint8]`

5.2 ()
98n22014.atf is a 'real' data file in ascii format. Open it using any text editor (the Matlab editor is fine as well) and inspect its contents. It contains a header (lines on top in which some experimental parameters are specified) and the data in columns, including a comment column (the last column). The first column contains time stamps from an electrophysiological recording (one channel). Write a routine which extracts the time stamps from this file (and others in the same format).
Note: function `textscan`, which is most suitable for this task, has an overwhelming number of input options. The corresponding passage in `demo08.m` may give you some orientation.

5.3 (*)
Modify function `i16load.m` (in func directory) such that, instead of 2-byte integers, 4-byte floating point numbers will be read (name this new function `f32load.m`). Next, read data in file *lfp.f32* with this function and take a look at the resulting variable's dimension (let's call it `d`). `d` originally resulted from a multi-channel recording, the data from individual channels placed in columns. It had been written into *lfp.f32* using fwrite, without any prior reshaping.
Your tasks:

- Find out how many channels there are (`d` represents 3 seconds of recording; the sampling frequency was 2000 Hz)
- Change `f32load` such that the data will be returned as a matrix with one channel per column (there is more than one way to do this).
- Assume `d` were much larger and you frequently needed to access excerpts of it. What is the problem with the current structure of data `d` in *lfp.f32* ? Can you think of a better way of saving them?

Note: solutions to this task can be found in the *func* directory
`[i16load]`

Unit 6

Topics: functions vs. scripts, global variables

6.1

Add a member to the family of `is...` functions. For example `isschnapszahl`, a function returning logical 1 if the input variable given to it is a number of the sort 777 or 11111 (an integer with repetitions of one and the same digit), and logical zero otherwise.
`[num2str or sprintf,findstr,deblank]`

6.2

Rewrite exercises 1.7 and 2.7 as functions. Do the same with a variant of exercise 2.5/4.6: a single particle shall move in one, two or three dimensions (again, as in exercise 4.6, implement only one loop termination condition). Think about which variables are best specified as input and output parameters. Consider a variable number of input/output arguments. Do not forget to add - right below the function name - at least some rudimentary information as to what the functions do.
`[varargin/varargout,nargin/nargout]`

6.3 (,∑)

Neurons have ionic gradients across their membranes (due to the activity of ion pumps). The Nernst equation can be used to compute the equilibrium potential of an ion, that is, the potential (voltage) between the interior and exterior of the cell at which no net ionic flow occurs. Example: in the cerebrospinal fluid of mammals potassium has a concentration of ca. 3 mmol/L, whereas inside neurons it is ca. 120 mmol/L. If there were no potential difference between the inside and outside, potassium ions would rush through potassium channels (provided they are open) from the inside to the outside (because there are many more ions inside than outside). Now imagine we apply (e.g. via an intracellular electrode) a negative potential inside the cell. The potassium ions, which have a positive charge, are attracted to the imposed negative charge, and their net outflow will decrease. At a certain point, this transmembrane *electric* gradient will compensate the *chemical* gradient; although individual ions still slip through the channels from either side of the membrane, the net flow is zero. This equilibrium potential can be computed via

$$E_{ion} = \frac{RT}{zF} \ln \frac{[ion]_{outside}}{[ion]_{inside}}$$

where R=8.3144 $J*K^{-1}*mol^{-1}$, T the absolute temperature in Kelvin, z the charge of the ion (+1 in the case of potassium) and F= 96485 $C*mol^{-1}$. $[ion]_{outside}$ and $[ion]_{inside}$ are the concentrations of the ion outside and inside the cell, respectively, in arbitrary but equal units. Your task: write a function which computes the equilibrium potential of any ion given T, z, and ionic concentrations (as input arguments). The equilibrium potential in millivolts shall be the output argument of the function. In the above example, at 25 °C the equilibrium potential of potassium is -94.78 mV.

6.4 (∑)
Write a function which plots a damped cosine wave over a range of input (abscissa) values. The formula is $y=\cos(x)/(1+ax^2)$. Input arguments of the function shall be

i) the range of abscissa values over which to plot the function (in radians: as you may remember, 2π, which is ca. 6.28, is one period)
ii) the step size (increments) of the abscissa values
iii) the value of parameter a
iv) the color and line style of the plot

For example, your function should be able to plot the damped cosine wave as a magenta solid line over the interval [-10 10] at increments of 0.01, with a damping factor a of 0.1.

6.5 (*)
Consider a function with a single input variable. Imagine you generate a variable in the base workspace and call the function, giving it as input the variable just created. Question: when the function is called, will Matlab by default generate a copy of this variable in the computer's memory (because it 'expects' that this variable will be changed in the function anyways)? This is of importance in situations where this variable is so large that an additional copy of it would slow down the computer or even not fit into its memory. Investigate this, conducting a crude test: write a simple function with one input argument (or a set of different variants of this function), generate a large array and call the function with the array as the input.Try the following conditions:

i) function accesses elements of the input array but does not modify them
ii) function accesses and modifies elements of the input array (changes the value of at least one element)
iii) function accesses and modifies elements of a *global* array of the same size.

In either condition, observe how much of system memory is taken up before and after calling the function(s) (on Windows machines the task manager informs you about this). If a copy of the variable is generated, the memory demanded by Matlab should increase substantially, otherwise it will not. What is the outcome? What are the implications for program development?
Notes:

a) The memory taken up by the test variable should be substantial, maybe a third of the computer's RAM, otherwise you may not get clear results. On the other hand, too large a variable may cause 'out of memory' errors right away. At any rate, it is recommended to shut Matlab down and re-start it before running the tests because Matlab fragments memory the more the longer it runs (and the `pack` command does not really rectify this)
b) Depending on a number of conditions, memory demand may either be tonically increased or only show a short 'spike' – you will probably have to set break points at strategic positions in your funtions to step through the code.
c) **Once again, this is an exercise the outcome of which hints at ways of efficient programming.**

Unit 7

Topics: memory issues, development tools (m-lint, dependency report, profiler), parallel computing

7.1
Run the profiler on `s0205.m`, a solution to exercise 2.5. You can start the profiler by typing `profile viewer` in the command window. Improve the code in `s0205` (=make it faster) on the basis of the profiler's results. Then, run the profiler on `s0205b.m`, the 'semi-vectorized' solution. Would it be worth changing the code in this solution in the same way as in `s0205.m`? Note: working with the profiler, you'll have to change the parameters to values that keep your computer busy for a few seconds to get meaningful results. Also, ignore the results of the first run – Matlab will possibly need to load some functions from hard disk in the first run (which will then become memory-resident and therefore not cause a delay anymore when called).

7.2
Using the profiler, convince yourself that `s0305.m`, the solution to exercise 3.5, is in fact on average faster than `s0301.m`: delete (or comment) the line `pause(0.05)` in both files and set identical stop conditions in both files which keep the computer busy for a few seconds.

7.3 (*)
Rewrite the code which generates surrogate data for one of the data files in demo10.m using function `parfevalOnAll`, ('parallel function evaluation on all workers') and compare execution time to the standard sequential code. 40 instances of surrogate data shall be generated and fetched. Hints: You'll have to know how many workers (=CPU cores) are available on your machine (see corresponding section in demo10.m). Also, in order to be able to measure execution time you have to use function `wait`, which will block the Matlab command window until all tasks run by `parfevalOnAll` are finished. This counteracts the purpose of `parfevalOnAll` (which similar to the `batch` command runs in the background, not blocking the Matlab client), but is necessary for timing. Finally, the size of variables generated in this task is substantial (several gigabytes), so reduce the number of instances computed or the size of the raw data if your computer needs to use swap space on the hard disk or you run into Matlab memory issues.
When you run the code on a computer with two or more cores, does `parfevalOnAll` speed up data generation compared to sequential computation? How about the memory (RAM) required in the different versions of the code?
`[fetchOutputs,wait]`
(\data\proj_timeSeriesAnalysis\wrat04_enf0004.mat)

Unit 8

Topic: fitting functions to data

8.1 (,*)
Variable ipsc in the data file listed below contains a small number of GABAergic inhibitory postsynaptic currents (IPSCs) recorded from a neocortical neuron in vitro at 35° C. The sampling interval is 100 µs in this recording, and the data traces represent the current in an interval of [-10 80] ms around the point of detection. First, get an overview of the data traces by plotting them. Pick an IPSC (or the average of all, if you prefer that) and subtract its base line (that is, the average value of data points before the steeply descending flank). Invert the

IPSC so that we see a positive peak and a decaying current (think shark fin). Then, fit a single exponential *to its decaying phase* using the curve fitting tool (evoke it by typing cftool at the command line or by clicking on its symbol in the APPS pane). The function shall be of the form y=a*exp(-t/tau). In the pane on the top, center, click on 'Custom Equation' and enter the formula. Which value does the decay time constant tau attain in milliseconds? (Hint: GABAergic IPSCs of the sort mentioned above have decay time constants of roughly 10-40 ms). Next, extend the formula to two exponentials (with decay time constants tau1 and tau2). Which one fits the data better, the single or the double exponential? Finally, *export* the code into an m-file (File - Generate Code). Modify this copy to fit any of the other IPSCs (or all of them in a loop).
`[cftool, fit]`
(\data\IPSCRawData.mat)

Unit 9

Topics: graphical user interfaces (GUIs)

9.1
A GUI is, in a rather broad sense, a graphics object which reacts to the user's actions (like pressing a mouse button or hitting a key). According to this view, even the simplest plot is a GUI because it can react to a mouse click. Run

```
      ph=plot(1,'o');
      get(ph)
```

and try to locate field `ButtonDownFcn` in the output of the get command. `ButtonDownFcn` is short for "Button press callback function". In other words, it is a property of the graphics object which determines the action(s) Matlab should take when a mouse button is pressed while the pointer is above the object. So, via

```
      set(ph,'ButtonDownFcn','disp(''bzzz'')');
```

each time you click on the lone data point in the graph the string `bzzz` will appear in the command window. More precisely, the string specified via the `set` command will be executed in the base workspace.
Your task: rewrite exercise 3.2 using this feature instead of `ginput`. Note that most likely the behavior of the solution produced here will not match that of exercise 3.2 in all fine detail (which is fine). For example, in `s0302.m` clicking near one of the doors will 'open' it whereas here the mouse pointer will definitely have to be within 5 pixels of the marker's confines for anything to happen.

9.2
Modify the previous task. The callbacks shall not be strings containing the commands but rather strings consisting of calls of functions which accomplish the job. An example of what is meant by this: `'disp(''hello'');'` is a string consisting of a call of function `disp` which accomplishes the job of greeting you at the command line. You will have to create these functions based on the solution to the previous exercise.
Hint: you may have the idea to solve the task by creating a 'main' function with subfunctions (to be called in the strings). Don't try – most likely you won't succeed. Leave the main code as a script and generate separate function files for the functions to be called. You will see why.

9.3 (*)
Produce a variant of 9.2. The callbacks shall now be handles to functions based on those created in 9.2.
Note: remember that any function whose handle shall be used in a callback of a graphics object **must** have two 'default' input arguments (plus as many others as needed).