Matlab for PhD students – Basics 2 – WS 2010/11

Functions, data types, if statements

Functions

- Definition
  - Like a script, a function is a program consisting of programming steps that are processed in sequential order.
  - Like a script, a Matlab function is a file with ending `.m` that is started by the filename.
  - In contrast to a script, typing in the commands to the command window would usually not have exactly the same effect as calling the function.
  - The main difference between scripts and functions is that each function has its private temporary workspace that is separated from the main workspace of the command window.
  - The function workspace is generated when the function is started. It contains the variables, which are generated during the processing of the program. After processing the last line of the program, Matlab deletes the temporary function workspace, while the main workspace of the command window persists.
  - The command window workspace and the function workspace can exchange specified variables via input and output arguments:
  - Input arguments are variables, which are transferred from the main workspace into the function workspace. They can be used within the function as private variables of the function. If the value of an input variable is changed in the function, the value in the command window workspace nevertheless remains the same.
  - Output arguments are variables that are transferred from the private function workspace to the command window workspace. They are the only data (except for data saved to a file) that is available after the program has terminated.
- Syntax
  - General: Functions are defined by the key word `function` in the first line of the program (the "header"), followed by the name of the program that is also used as file name. Functions are called by their file name.
  - Function without input- and output arguments

    First line in program text: `function testfunction1`
    Filename: `testfunction1.m`
    Call from command window: `testfunction1`
  - Function with one input argument

    The variable names of input arguments are put in parentheses behind the function name both in the first line of the program and in the function call.
    Variables can have different names in the function workspace and the command window workspace.

    First line in program text: `function testfunction2(in1)`
    Filename: `testfunction2.m`
    Function call:

```
testfunction2(in1)
     % variable in1 must be defined in
```

```
     % command window workspace.
testfunction2(M)
     % variable M  must be defined in
     % command window workspace.
     testfunction2([27 15])
```

- Function with several input arguments

  Input arguments are separated by comma in the parentheses behind the function name.

  First line in program text: `function testfunction3(in1,in2)`
  Filename: `testfunction3.m`
  Function call:

```
testfunction3(in1,in2)
     % variable in1 and in2 must be
     % defined in command window
     % workspace.
testfunction2(M,7)
     % variable M from command window
     % work space will be called in1
     % in function workspace. in2 in
     % function workspace has value 7.
```

- Function with one output argument

  Output arguments are defined like in variable definitions as `outputname=functionname` both in the file header and the function call.

  First line in program text: `function out1=testfunction4`
  Filename: `testfunction4.m`
  Function call:

```
out1=testfunction4
A=testfunction4
     % writes content of variable out1
     % in function workspace to
     % variable A in command window
     % workspace.
testfunction4
     % if a function is called without
     % specifying the name of the
     % output variable, the output is
     % written to ans
```

- Function with several output arguments

  When a function has more than one output argument, they are written in square brackets and separated by comma.

  First line in program text: `function [o1,o2]=testfunction5`
  Filename: `testfunction5.m`
  Function call:

```
[a,b]=testfunction5
     % regular function call,
     % transferring all output
     % variables o1 and o2 to the
     % workspace via the new workspace
     % variables a and b.
a=testfunction5
     % if a function is called with
     % fewer output arguments than
     % possible, only the first
```

```
                    % variables will be set, the
                    % variables occurring later in
                    % the list of output arguments in
                    % the function header will not be
                    % transferred into the workspace.
               testfunction5
                    % if a function is called without
                    % specifying the name of the
                    % output variable, the first
                    % output is written to ans
```

- Function with input and output arguments
  Very often, input and output arguments are combined:
  First line in program text:
  `function [o1,o2]=testfunction6(in1,in2,in3)`
  Filename: `testfunction6.m`
  Function call: `[a,b]=testfunction6(M,N,L)`

- A build-in function with variable number of input and output arguments

```
hist(v)          % If hist is called with only one input and
                 % without output argument, the range of
                 % values in vector v is divided into 10
                 % equally spaced bins and the number of
                 % occurrences of these classes is displayed
                 % graphically.
hist(v,nbins)    % The range of values in vector v is
                 % divided into nbins (positive integer
                 % number) equally spaced bins and the
                 % number of occurrences of these classes is
                 % displayed graphically.
hist(v,centers)% If the second input argument is a vector,
                 % the values in vector v are divided into
                 % classes centered on the values given
                 % in vector centers.
H=hist(v)        % If hist is called with an output argument
                 % it gives back the numbers of occurrences
                 % of the classes as a vector and does not
                 % display them graphically. Function calls
                with
                 % Output arguments can be combined with one
                 % or two input arguments.
[H,xout]=hist(v)    % If hist is called with two output
                 % arguments, it gives back the vector H of
                 % numbers of occurrences as first output
                 % and the vector xout of class centers as
                 % second output. Function calls with
                 % Output arguments can be combined with one
                 % or two input arguments.
```

- Good style
  - Program names:
    - Programs should have names describing their purpose. You will not be amused to find 7 years old files called function1,

function2, … function27 and other persons who want to use your programs will be really annoyed.
        - For program names, the same rules as for variable names apply. They need to consist of letters, digits and underscores and have to start with a letter. Keep away from space characters and Umlaute (it might work ok sometimes on one computer, but it is very unsafe).
        - For functions, the function name given in the program header and the program name should be the same. (In case they are not, you need to call the function by the filename.)
    - Comments and help text
        - The first lines after the function header are reserved for the help text, which is displayed in the command window in response to typing in e.g. `help myfunction`. In the help text, each line starts with %.
        - Help texts of functions should describe the purpose of the function and list their input and output arguments. It can also be helpful to give an example of the usage.
        - In the main body of the function, it is good style to write a comment for each line that is not immediately obvious.
        - The editor cell mode helps to structure the program into sub-problems.
    - Test for correct input arguments
        - If a user tries to call your function with too few input arguments, Matlab will stop to execute the function with an error when the function reaches the point at which the input variable is needed for a calculation.
        - A Matlab error will also occur, if it is impossible for Matlab to calculate a step in the program with a given input argument.
        - However, there are many possibilities that Matlab will execute the function, even though an input argument does not make sense for the program (e.g. calculate with the ASCII number of a letter, see below)
        - Therefore it is good style to test for all function inputs if they are useful in the context of the function, before starting the calculations themselves.
        - In particular test if the input arguments have the correct types.
        - And test if they are in the correct range or if they are e.g. too small or too big numbers.

Well-structured programs

When you write a program for a non-trivial problem, it is very useful to spend some time on thinking about the structure of the program before you start to write code – this step can save a lot of time on the long run!

- Divide and conquer principle
    - Break down your problem into sub-problems until each of them is small enough to be solved easily.
    - Write a separate piece of code for each of these small sub-problems.
    - Label these pieces of code in a meaningful way. A very good way is to write separate programs (scripts or functions) for each of them. Another possibility is to use the Matlab editor cell mode (see below).

    - You will find that many of these problems are of a general type, occurring in more than one context. If you implement your algorithms for these problems as general as possible, you will be able to re-use them over and over again without having to invent and implement them again.
    - A big advantage of re-using your programs is that you only have to make changes in one place if you find an error in your code. If you copy your code into all the programs doing the same thing, you have to find all these occurrences in several places to fix an error.
- Calling scripts and functions from a program
    - Scripts and functions can call other scripts or functions, like it would be done from the command line.
    - Therefore, it is very easy to re-use programs for problems that occur in several contexts.
    - The workspace rules are the same as for script and function calls from the command line:
    - A function always has its private workspace and input and output variables are used to transfer variables between the workspaces of called function and calling command line / script / function.
    - A script always shares the workspace of the calling command line / script / function.
- Script or function?
    - To decide if you want to write a script or a function to solve a specific problem, you need to think about the workspace:
    - If you want the user to see all variables used in your program and if you are sure that it is ok to potentially overwrite variables of the same name, you should write a script.
    - Otherwise, you should write a function with well-defined input and output parameters.
    - It is good style to write functions for all sub-problems and to write scripts calling these functions for specific problems with specific parameters, which should be seen by the user.
- Matlab editor cell mode
    - In the Matlab editor, you can add a sub-structure to your programs by using the cell mode.
    - A program cell is a piece of code for a specific sub-task.
    - Borders between cells are defined by a comment line starting with `%%`. It is good style to give the cell a name, which can be used later for quick reference of the program structure.

Numerical data types

- **Double**
    - Definition: double precision floating point numbers are the standard data type in Matlab. If you use a variable for numeric data without specifying the data type explicitly, you will get a double precision floating point number.
    - Creation: E.g. `a=9; b=0.007` or `c=zeros(5,6)`
    - Symbol in workspace window: box of four small boxes
    - Special numbers:
        - `eps` Smallest possible difference between two numbers

(double precision numbers: `eps = 2^(-52)`)
    - `inf` Infinity (e.g. division by 0)
    - `nan` Not-a-number (undefined numerical results, e.g. missing data)
  - Formatted display: With format you can set the display format for number output. After the command `format('type')` all floating point numbers will be displayed according to the type specified:

```
short     % fixed point 4 digits after decimal point
          % e.g. 3.1416
long      % fixed point 14 digits after decimal point
          % e.g. 3.14159265358979
short e   % floating point 4 digits, e.g. 3.1416e+000
long e    % floating point 14 digits, e.g.
          % 3.141592653589793e+000
```

  - Measurement of continuous variables: For the measurement of continuous variables (e.g. the membrane potential of a cell, the blood pressure of a patient…) it is important to keep in mind that a computer needs to sample the values at certain time points and only in certain steps. Weird things can happen, if the sample rate is too low. Take a look at the script `aliasing_effect.m` (on the homepage).

- **Integer**
  - Definition: Integers are natural numbers, 0 and the negatives of the natural numbers.
  - In Matlab, it is necessary to specify the number of bits used to store the numbers and to decide if you want to use signed or unsigned integers:
    - uint8 (unsigned 8-bit integer, range 0 to 255)
    - int8 (signed 8-bit integer, range -128 to 127)
    - uint16 (unsigned 16-bit integer, range 0 to 65535)
    - int16 (signed 16-bit integer, range -32768 to 32767)
    - uint32 (unsigned 32-bit integer, range 0 to 2^{32})
    - int32 (signed 32-bit integer, range -2^{31} to $2^{31}-1$)
    - uint64 (unsigned 64-bit integer, range 0 to 2^{64})
    - int64 (signed 64-bit integer, range -2^{63} to $2^{63}-1$)
  - The 8 and 16 bit integers make sense on systems with little storage capacity. On modern computers, however, they do not play such an important role as they did in the past.
  - Symbol in workspace window: box of four small boxes (like floats)
  - Creation: in most cases you have to cast results into the desired data type, e.g. `a=uint16(7866567); b=int8(-7)`. Specifically for the creation of a matrix consisting only of zeros or ones, you can use a specific option, e.g. `c=ones(5,8,'int32');` `d=zeros(17,1,'uint8')`
  - Usage: Mathematical operations are defined for integer types up to 32 bit in the same way as for double precision floating point numbers. (Maybe in Matlab 2008 also 64 bit.) However, only integers of the same type can be combined (exception: combination of a scalar double precision value with an integer matrix). If necessary, the results will be rounded to the nearest integer values. If a calculation result exceeds the range of the variable type, the result will be set to the minimum or maximum value of the data type.

- The most common use of integer number is indexing, because indices of vectors and matrices must be positive natural numbers.
- Caution! Once a variable is specified as integer, it will remain integer even if floating point numbers are included. The floating point number will be rounded to the nearest integer value. E.g. `a=zeros(2,3,'uint8'); a(1,2)=pi` will give `a=[0 3 0; 0 0 0]`.

Character

- Definition: Characters are letters, digits and special symbols. In ASCII code each character has a specific number. Text is organized in Matlab as array of characters. These arrays can have several lines (rows), but all of them must have the same length.
- Symbol in workspace window: box with 'ab'
- Creation: Items surrounded by ' ' are considered to be strings of data type char (character). E.g. `t='This is a text'.`
- Type conversion: Numbers and characters can be converted into each other

```
A=int2str(M)  % converts integer array to character array
B=num2str(M)  % converts floating point array to
              % character array
d=double(s)   % converts character array in array of
              % ASCII values
f=int8(s)     % converts character array in array of
              % ASCII values (similar for other int
              % types)
t=sprintf('text %g', x)   % most flexible way to create a
                          % character array from
                          % different types of variables. Can
                          % also be used to generate command
                          % line output for users.
                          % For options and related functions
                          % see help!
```

- Caution! You can calculate with char arrays without meaning to… e.g. `c='a'; d=c+1` will result in double variable `d=98` (= ASCII value 97 + 1)

- Manipulation: Character arrays can be manipulated in the same way as numeric arrays with concatenation by `[ ],` a new line by `;` and the usual indexing, `size` function etc. Moreover, there are special commands:

```
strcat(s1,s2)      % concatenates two or more strings
                   % horizontally
strvcat(s1,s2)     % concatenates two or more strings
                   % vertically. If the strings are not
                   % equal in length, the shorter ones
                   % are padded with blanks
a=lower(s1)        % converts all upper-case letters in
                   % s1 to lower case.
b=upper(s1)        % converts all lower-case letters in
                   % s1 to upper case.
```

- Comparison: Unfortunately, comparison of char arrays with `s1==s2` sometimes works, but most often it does not. The save commands are

```
strcmp(s1,s2)      % compares if s1 and s2 are equal
```

```
strncmp(s1,s2,n)   % compares if first n characters of s1
                   % and s2 are equal
strcmpi(s1,s2)     % compares if s1 and s2 are equal,
                   % ignoring case (Groß-/
                   % Kleinschreibung)
strncmpi(s1,s2,n)  % compares if first n characters of s1
                   % and s2 are equal, ignoring case
findstr(s1,s2)     % find shorter string within longer
                   % string
strmatch(s1,s2)    % find indices of matching strings
```

There are also some specific tests for strings:

```
ischar(s)          % test if s is a string array
isletter(s)        % test if s consists of letters of the
                   % alphabet
isspace(s)         % test if s consists of white-space
                   % characters
```

Searching in strings is also possible with regular expressions. If you want to use this powerful tool, please refer to the help or a good book!

- Command line outputs: Strings are also the way to communicate with the user via the command window. In addition to command line outputs in response to any command that is issued without ; there are some specialized commands for command window output:

```
disp('hello world')              % displays the string in the
                                 % command window
sprintf('result: %g', m)         % displays the string in the
                                 % command window and fills in
                                 % the value of variable m into
                                 % the space kept open by %g
warning('are you sure?')         % displays the string in the
                                 % command window as a warning

error('this is impossible!')     % displays the string in the
                                 % command window in red, issues
                                 % a beep and interrupts the
                                 % program.
```

- Command line inputs: You can read in keyboard inputs from the user

```
a=input('Your name: ','s')       % reads in a string typed in
                                 % by the user into the
                                 % variable a
b=input('Your age in years: ')   % reads in a number (or a
                                 % matlab expression) into
                                 % variable b
```

Logicals

- Definition: Logical variables are arrays of type logical, consisting of only two values 'false'=0 and 'true'~=0 (usually 1)
- Most common use of logical variables:
  - Using logicals in conditional statements (like if, while)
  - Logical indexing
- Symbol in workspace window: box with checkmark
- Creation: by explicit type definition logical, or (very often) by using logical operations and relational operators, e.g. `a=logical(1); b=logical(false); c=(9>7);`

- Relational operators: comparison of values, having arrays of logical values as outputs: `<, <=, >, >=, ==, ~=` (Caution: == can be tricky to use. If values differ by a very small amount due to limited calculation precision the result will be `false`!)
- Logical operators: Way to combine or negate relational expressions (e.g. `a=b&c`)
    - `&` element-by-element AND for arrays
    - | element-by-element OR for arrays
    - ~ element-by-element NOT for arrays
    - `&&` scalar AND with short-circuit
    - || scalar OR with short-circuit
- Examples for relational and logical functions:

```
z=xor(x,y)          % element-by-element exclusive OR for arrays
a=any(x)            % true if any element of x is nonzero. In
                    % Matrix: for each column
aa=all(x)           % true if all elements of x are nonzero. In
                    % Matrix: for each column
e=isempty(x)        % true for empty matrix
n=isnan(x)          % true for not-a-number, returns a logical for
                    % each element of a vector or matrix
isinf(x)            % true for infinity, returns a logical for
                    % each element of a vector or matrix
f=isfinite(x)       % true for all numerical values and false for
                    % NaN and inf, returns a logical for each
                    % element of a vector or matrix
s=issorted(x)       % true if x is sorted in ascending order
k=iskeyword(x)      % true if x is a reserved keyword
vn=isvarname(x)     % true for valid variable names
```

Casting

- Idea: Casting is the conversion of a variable from one data type to a different one.
- Syntax: In general, variables can be changed to a specific type by using the type name as function name. Moreover, there are two general functions for casting, which differ in their results concerning the conversion of a variable into a data type using less disc space.

```
x=logical(1)            % Converts the number 1 (which would
                        % usually be a double) into a logical.
C=char(77)              % Gives the letter with ASCII number 77,
                        % returns C='M'
D=double('9')           % Gives the ASCII number of digit '9',
                        % returns D=57
I=int64(x)              % Converts a variable x of any numeric
                        % data type, a character or a logical to
                        % a variable I of type int64.
F=uint8(1000)           % Values that are too big to fit in a
                        % data type are truncated to the biggest
                        % possible value. Here, F=255.
Y=cast(X,type)          % Converts a variable X to the
                        % data type specified by type.
                        % type is a string specifying the
                        % numeric data type, e.g. 'double',
                        % 'uint32' or 'char'.
                        % cast truncates values that are too big
                        % to fit in a data type. E.g.
```

```
                        % y=cast(1000,uint8) returns y=255.
```

Tests for types and equality

- Tests for types: Particularly in functions it is very important to test if the user has called the function with input arguments of adequate types. Otherwise weird errors can occur, if the user e.g. tries to calculate the square of 'hello world'…

```
isscalar(x)         % true for a scalar
isvector(x)         % true for vector
isfloat(x)          % true for array of floating point numbers
isnumeric(x)        % true for numeric array
ischar(x)           % true for character string array
islogical(x)        % true for array of logicals
isstruct(x)         % true for structure
iscell(x)           % true for cell array
isobject(x)         % true for objects
isa(x,'type')       % true if x is of type 'type'
```

- Tests for equality: The logical test `a==b` only works reliably for numerical data types and when two variables of the same dimensions are compared. Other data should be compared with `isequal`.

```
A=(n==8)            % A is true (1) if n is equal to 8, otherwise A
                    % is false (0). If n is a matrix or a vector,
                    % each element is compared with 8 and A is a
                    % logical matrix with the same dimensions as n
A=n==8              % Returns the same result as above. The
                    % parentheses are not necessary but make it
                    % easier to read the statement
B=([8 9]==[9 8])    % For vectors and matrices of the same size, ==
                    % compares element-by-element and returns a
                    % logical value for each element
tf=isequal(v,[9 7]) % Test if two or more arrays are equal.
                    % Returns only one logical value independent
                    % of the dimensions of the input arguments.
                    % Save way to compare also non-numerical data
                    % types like char arrays, structures and cell
                    % arrays.
```

if-Statements

With if statements the control flow of a program is split. Depending on certain conditions, computations are performed or not.
The Matlab editor formats programs with if statements in a way that conditional parts of the program are indented for easier overview of the total control flow.

- if-statements with one condition
  - Idea: Certain computations should only be performed if a certain condition is true.
  - Syntax:

```
if condition
   commands
end
```

  - Example:

```
a=input('your number: ')            % user input
if a>0                              % condition
    b=a^2;                          % b will only be set if
                                    % condition a>0 is true.
```

```
    disp('your  number is positive.')      % output on command
                                           % line depends on user input.
end                                        % key word to end
                                           % conditional part
c=-2*a;                                    % commands after key word ‚end'
                                           % will be performed
                                           % independently of the
                                           % condition.
```

- if-else-statements
  - Idea: Certain computations should only be performed if a certain condition is true. If the condition is false, certain different computations should be performed.
  - Syntax:

```
if condition
  commands
else
  commands2
end
```

  - Example:

```
a=input('your number: ')                   % user input
if a>0                                     % condition
    b=a^2;                                 % b will only be set if the
                                           % condition a>0 is true.
    disp('your  number is positive.')      % output on command line
                                           % depends on user input.
else                                       % key word to start the other
                                           % block of commands
    disp('your number is not positive.')   % output on command line
                                           % depends on user input.
end                                        % key word to end conditional
                                           % part
c=-2*a;                                    % commands after key word ‚end'
                                           % will be performed
                                           % independently of the
                                           % condition.
```

elseif-statements with more than one condition

- Idea: If there are more than two possibilities for a condition, additional blocks of statements are performed depending on conditions in `elseif` statements.
- There can be several `elseif` statements. However, it is good style to also have an `else`-statement to make sure that each possibility is covered.
- Matlab looks for the first condition that is true and performs the corresponding commands. Subsequent elseif conditions that happen to also be true will not be executed!
- Syntax:

```
if condition
  commands
elseif condition2
  commands2
else
  commands3
end
```

- Example:

```
a=input('your number: ')                % user input
if a>0                                  % condition
    b=a^2;                              % b will only be set if the
                                        % condition a>0 is true.
    disp('your number is positive.')     % output on command line
                                        % depends on user input.
elseif a==0                             % condition2
    disp('your number is 0. ')          % output on command line
                                        % depends on user input.
else                                    % key word to start the other
                                        % block of commands
    d=a/2;                              % d will only be set if a<0
    disp('your number is negative.')     % output on command line
                                        % depends on user input.
end                                     % key word to end conditional
                                        % part
c=-2*a;                                 % commands after key word ‚end'
                                        % will be performed
                                        % independently of the
                                        % condition.
```

- Nested if-statements
  - Idea: If certain blocks of commands depend on combinations of conditions, you can use if-statements in if-statements.
  - There can be several `elseif` statements for the inner and outer if-statements.
  - The Matlab editor uses several levels of indentation to make the program easier to read.
  - Syntax:

```
if condition                    % outer condition
   commands
   if condition2                % inner condition
        commands2
  elseif condition3
        commands3
   else
        commands4
   end                          % end of inner statement
else
   commands5                    % else- or elseif- cases can
                                % also contain nested if-
                                % statements.
end
```

  - Example:

```
a=input('your number: ')                % user input
if isnumeric(a)                         % outer condition
    if a>0                              % inner condition
        b=a^2;                          % b will only be set if both
                                        % conditions are true.
        disp('your number is positive.') % output on command line
                                        % depends on user input.
    else                                % else for inner if condition
        disp('your number is not positive.') % output on command
                                        % line depends on user input.
    end                                 % key word to end inner if
```

```
                                        % statement
    c=-2*a;                             % commands after inner ‚end'
                                        % will be performed
                                        % independently of the inner
                                        % condition, but only if the
                                        % outer condition is true.
else                                    % else for outer condition
    warning('This was not a number!') % warning will be issued if
                                        % outer condition is false
end                                     % end of outer if statement
v=zeros(1,10);                          % Following statements are
                                        % computed independent of
                                        % conditions.
```

- Combined conditions
  - Idea: The other way to let certain blocks of commands depend on combinations of conditions, is to use combined conditions in if-statements, combining the conditions with logical operators.
  - There can be several elseif-statements.
  - Since combined conditions can become quite confusing, it is important to think about all possible combinations of conditions that can occur and to have an else-case collecting all other (potentially unwanted) combinations of conditions.
  - Test your programs with combined conditions thoroughly!
  - Again, it is important to keep in mind that only the first block of commands for which all conditions are true will be performed!
  - Syntax:

```
if condition1 && condition2
  commands
elseif condition1 && condition3
  commands2
elseif condition2 && condition3
   commands3
else
  commands4
end
commands5
```

  - Example:

```
a=input('your number: ')                % user input
if isnumeric(a) && a>0                  % combination of conditions
    b=a^2;                              % b will only be set if the
                                        % condition a>0 is true.
    disp('your number is positive.')    % output on command line
                                        % depends on user input.
    c=-2*a;                             % this command should be
                                        % performed in any case when a
                                        % is numeric.
elseif isnumeric(a)                     % one of the conditions in if
                                        % is true (a is a number), the
                                        % other is not (a<=0).
    disp('your number is not positive.')    % output on command line
                                        % depends on user input.
    c=-2*a;                             % this command should be
                                        % performed in any case when a
```

```
                                        % is numeric.
else                                    % all cases when a is not
                                        % numeric
    warning('This was not a number!')    % warning will be issued if
                                         % isnumeric(a) is false
end                                     % end of if statement
v=zeros(1,10);                          % Following statements are
                                        % independent of conditions.
```

- Switch
  - Idea: Switch statements are the alternative to elseif-statements with many different cases. They are usually used for characters or arrays of characters (text), because they expect expressions.
  - Syntax:

```
switch switch_expression
   case case_expression1
        commands1
   case case_expression2
        commands2
   otherwise
        commands3
end
commands4
```

  - Example:

```
a=input('enter letter A, B, or C:','s')   % user input of
                                        % character
a=upper(a);                             % make sure that a contains
                                        % upper case letters
switch a                                % different possibilities for a
    case 'A'                            % first case
        disp('your letter is A')
        c=27;
    case {'B','C'}                      % cases can be combined by
                                        % using {}, meaning "B or C"
        disp('your letter is B or C')
        c=108;
    otherwise                           % all other cases
        warning('you did not enter A, B, or C!')
end
```

Statistical significance test

- Significance tests are not included in the most basic Matlab version. They are part of the Matlab Statistics Toolbox (which Peter Harmand will show you later in the course). Today, you will only use the functions for the student t-test as an example how to determine statistical significance with Matlab.
- One-sample t-test: The function ttest tests the significance of the expectation value of a normally distributed data sample. The null hypothesis is that a sample of independent normally distributed data was drawn from a distribution with mean μ_0 and unknown variance. It is tested if the empirical determined mean μ "is equal to" μ_0 in the statistically expected range depending on the significance level. Standard significance level is 5%. The function ttest gives back a logical value h. h = 1 indicates a rejection of the null hypothesis at the given significance level (meaning that the data does not come from a normal

distribution with mean μ_0). h = 0 indicates a failure to reject the null hypothesis at the given significance level (meaning that the data could come from a normal distribution with mean μ_0).

```
h=ttest(x,mu_0)       % Tests the null hypothesis that data in
                      % the vector x are a random sample from a
                      % normal distribution with mean mu_0 and
                      % unknown variance at 5% significance
                      % level.
h=ttest(x,mu_0,alpha) % Test at significance level alpha (e.g.
                      % alpha=0.01).
```

- Two-sample t-test: The function ttest2 performs a t-test of the null hypothesis that data in the vectors x and y are independent random samples from normal distributions with equal means and equal but unknown variances, against the alternative that the means are not equal. The result of the test is returned in h. h = 1 indicates a rejection of the null hypothesis at a given significance level (default: 5%, meaning that x and y come from different distributions). h = 0 indicates a failure to reject the null hypothesis at the given significance level (meaning that x and y could come from the same distribution). x and y need not be vectors of the same length.

```
h=ttest2(x,y)         % Tests the null hypothesis that data in
                      % vectors x and y are random samples from
                      % the same normal distribution at 5%
                      % significance level.
h=ttest2(x,y,alpha)   % Test at significance level alpha (e.g.
                      % alpha=0.01).
```

Homework

Exercises marked with * are optional.

1. **Exercises to practice functions and if statements**
   a. * Write a function that gets a matrix as input and gives back a vector of the four “corners” of the matrix
   b. Write a function that gets a square matrix as input and gives back the elements on the diagonal as a vector.
   c. Add to your function of the last exercise the test, if the matrix is really a square matrix of numbers and issue a warning if it is not.
   d. Write a function that gets a matrix and a number N as input arguments and plots the N-the row of the matrix.
   e. * Write a function that gets a matrix as input argument and plots the first, the medium and the last column of the matrix by calling the function of the last exercise.
   f. Write a function that gets some user input and saves it to a file. The name of this file should be generated automatically by combining the fixed string ‘userinput’ with the current date obtained by the function `date` (see help).
   g. * Write a program to tell people the name of their responsible official at the employment center. The program should ask the user in a friendly way to type in their name and tell them the name of the person in charge depending on the first letter of their last name:
      A-D: Frau Schmidt
      E-H: Herr Klein
      I-L: Frau Gross
      M-P: Frau Mueller
      Q-T: Herr Maier
      U-Z: Herr Dimpfelmoser

2. Exercises for data analysis

a) Load the data set `katzen.mat` from the course home page. This data set describes the cats in an animal shelter. The vector `alter` gives their age (in years), the vector `weibchen` their sex (`true` means female, `false` means male). The matrix `farben` tells the colors of each of the 23 animals' fur:
   1. Column: black
   2. Column: red
   3. Column: white
   4. Column: tabby

   The order of the animals is in all variables the same.
   Find out:
   1. How many cats are younger than 2 years?
   2. How many female cats are younger than 2 years?
   3. How many cats are tabby?
   4. * How many cats are only tabby without having also other colors?
   5. * Are the black-white-red cats males or females?
   6. * How many cats have only one color (excluding tabby cats?)
   7. * How many male black cats are older than one year?

b) On the course homepage, you will find the two matrices `stimulus.mat` and `antwort.mat`. They were recorded with 1000 bins/second with our electrophysiology setup and an electronic cell model. `stimulus.mat` is the current stimulus (nA) which was applied to the cell model. `antwort.mat` is the recorded voltage response (mV) of the cell model. Write a program to
   - Plot the stimulus and the response in two sub-plots corresponding to each other. Label the axes, displaying the time as seconds.
   - Plot in a new figure the histogram of the recorded response values (it makes sense to use more than 10 classes). What do you see?
   - Calculate the mean and the standard deviation of the responses before the stimulus changes.
   - Determine the time when the cell model responds to the stimulus by using a threshold: The response starts, when a data point differs by more than two standard deviations from the average response before stimulation.
   - Calculate the time difference between the stimulus change and the response onset.
   - * Think about a useful criterion to determine the offset of the response and calculate the time between stimulus offset and response offset.

   Write your program as a function with useful input and output parameters.

c) Write a function which gets a positive number N and the option 'normal' or 'uniform' as input
   - The function generates N random numbers according to the option specified by the user.
   - It plots the distribution of the random numbers as a histogram
   - It gives back the median, the mean and the standard deviation of the generated random numbers.

d) Load the data set `pellets.mat`. They contain the data of food pellets eaten by a two populations of 30 mice, counted for 30 days. The values for an individual mouse are written in one row, the columns represent the different days.
   - Write a program to calculate mean and standard deviations of the eaten pellets once across animals and once across days and display them

graphically with `errorbar(x,m,s)` (with mean m and standard deviation s which will be shown in relation to vector x, which in this case is the number of the mouse or the day) .

- The mouse food dealer claims that a mouse eats on average 32 food pellets per day. Test this claim for your population on the 5% significance level.
- Compare two mice with similar appetite, e.g. mouse 3 and mouse 26. Are their numbers of eaten pellets significantly different? How about mice 25 and 28? Or 1 and 8?