Matlab for PhD students – Advanced Topics 5 – WS 2010/11

"Advanced data types"

General consideration

- Basic data types:
  - Standard data type in Matlab are double precision floating point numbers. If you use a variable for numeric data without specifying the data type explicitly, you will get a double precision floating point number.
  - Other basic data types in Matlab are integers, logicals and characters.
  - Casting is the way to convert a variable of one data type into a different data type.
- Idea of complex data types:
  - Create a variable consisting of several sub-variables, which can have different types. A structure or cell array is a container for other data types.
  - The contents of a cell or structure must be addressed to perform mathematical operations.
- Comparison of structures and cell arrays:
  - Structures and cell arrays are very similar, except for the way their contents are addressed:
  - A structure can consist of several fields, which have specific names.
  - A cell array can consist of several cells, which are numbered like in a matrix.
- Advantages of complex data types:
  - Properties of one "item" which belong together will stay together.
  - Only one variable needs to be transferred between functions.
  - If the structure or cell array is changed (e.g. by adding a new field or cell), the new version will be present in all functions which have the structure or cell array as input argument.
  - Structures and cell arrays can contain all data types including structures and cell arrays.
  - Structure: Field names make it easy to remember what is inside the structure.
  - Cell array: Cell numbers can be used for indexing like you know it from matrices. This makes it easy to search within the cell array through all the cells.
- Disadvantages:
  - Mathematical functions are not defined for structures and cell arrays, because they can contain different data types (their contents must be addressed to use mathematical functions).
  - Functions like *find* and *sort* also do not work on structures and cell arrays.
  - Structure: It is not possible to use loops to address all fields (because they are not sorted numerically, but are addressed by their names). You have to remember the names of the fields (or to look them up) to address them.
  - Cell array: You have to remember which cell number contains which property or data, because the cells do not have names, but numbers.

Basic data types: See script IntroductionMatlab2

- Numerical data types
  - Double
  - Integer
- Character and strings
- Logicals
- Tests for types and equality
- Casting of types

Arrays: Vectors and matrices

- Basics: See script IntroductionMatlab1 for
  - Nomenclature
  - Generation of vectors and matrices in Matlab
  - Indexing
  - Determine the size of vectors and matrices
  - Concatenation
  - Transposing
  - Reshaping
  - Mathematical operations
- Sparse matrices:
  - See script Matlab_adv3_speeding_up
- Logical indexing:
  - Idea: Logical indexing is a convenient and fast way to extract elements of a vector or matrix, for which a condition is true.
  - Usage: When a matrix (or vector) of type logical is used to index a matrix (or vector) of the same dimensions, only those elements of the matrix are extracted, for which the corresponding elements of the logical matrix are `true`.
  - Example (see `logical_indexing_demo.m` included in `struct_cells.zip` at the course homepage):

```
%% Generate a matrix
a=[8 9 -5; 0 -1 -10; 9 -9 0]

%% extract positive values with logical indexing:
b=a>0          % generate a logical matrix with 1 for
               % elements >0
c=a(b)         % generate a vector of positive elements of a

% Of course, the same result could be obtained without using
% a new variable:
d=a(a>0)              % d==c
```

  - Logical indexing vs find: The same results can be obtained by using the `find` command, but logical indexing is faster.

```
%% alternative solution using find:
ind=find(a>0)% vector of indices of positive elements of a
f=a(ind)     % apply indices to a to extract a vector of
             % positive elements
```

Cell arrays

- Idea: A cell array is very similar to a matrix with elements consisting of several different data types. Cell arrays can have any number of dimensions.
- Example to visualize a cell array: Imagine the inventory of laboratory equipment in the university. It contains a large number of very different items (e.g. chemicals, glass ware, gloves, paper, pencils…). The size and amount of each of these different items can differ considerably. Each of these different items has a number, which you need to know to get one or several of these items.
- Creation (see `celldemo.m` in `struct_cells.zip` at the homepage): There are several ways to generate a cell:
  - Command `cell`:

```
D = cell(3,2)          % creates a cell array D with 3-by-2
                       % cells containing empty matrices.
```

  - Conversions from matrix to cell: If you want to convert a matrix into a cell array you can use the function `num2cell`.

```
c=num2cell(m);    % creates a cell array c with the same
                  % dimensions as matrix m. Each matrix
                  % element is written to a separate
                  % cell.
```

  - Creating cell arrays by assigning values to cells: A cell array can also be generated without specific commands by assigning a value to a cell of a cell array with a new name (like a matrix can be generated by assigning a value to a matrix element). (Both types of cell array indexing can be used to create cell arrays, see below.
- Indexing: There are two ways of addressing cell arrays, each of which has its advantages and disadvantages (and the use of each one of these also seems to be a matter of personal taste).
  - Content addressing: Individual cells of a cell array are addressed by curly braces. A new cell array is generated by assigning a value to a cell of a cell array with a new name.

```
D{1,3}=8;              % assigns value 8 to the cell at
                       % position 1,3 of cell array D
A{1}=[1 5 3];          % creates a cell array A with one
                       % cell containing the vector
A{2}='A B C D E';      % adds a second cell to cell Array A
                       % containing the string
A{3}=true;             % adds a third cell containing the
                       % logical value
```

    - When cell arrays have more than one dimension, the data types of each cell can still be different. In particular the data types do not have to be the same for all cells of one row or of one column. E.g. you can continue to define `A` by writing

```
A{2,2}=9      % your cell array A now has dimensions 2x3
              % with cell {3} becoming cell {1,3} etc,
              % and cells {2,1} and {2,3} being empty
              % matrices [ ].
```

    - You can combine cell indexing with matrix indexing inside of cells by

placing the index behind the curly braces. E.g.

```
z=A{1,1}(2)  % results in z=5, because the second element of
             % the vector contained in A{1,1} is 5.
```

- You can use the : notation to address several cells. E.g.

```
[a,b]=A{:,1} % results in a=[1 2 3], b= []
```

- If you want to address several cells in an order that cannot be constructed by the : notation, you can use the function `deal` to assign the contents of individual cells to individual output arguments. E.g.

```
[c,d]=deal(A{1,3},A{2,1})     % results in c=true and d=[]
```

- Cell indexing: There is also an alternative way to index cells. In cell indexing, the curly braces are written around the right-hand side of the equal sign and cell indices are written with parentheses like for matrices. Again, a cell array can be generated by assigning values to certain cells of a previously non-existing variable. (In contrast to content addressing, I was not able to find out how to address individual elements of a matrix contained in a cell in this notation – however there might be a way…)

```
A(1,3)={9}        % assign value 9 to cell at position (1,3)
E={m, 'bla', true, D}      % convenient way to combine
                           % variables of different types in
                           % one cell array
t1=E(3)          % Returns a cell array
t2=E{3}          % Returns the contents (logical value 1)
```

- Symbol in work space window: box with curly braces
- Display cell arrays:
  - The command `celldisp(A)` displays the contents of all cells in the command window.
  - Statements without semicolon only display the contents of a cell array if it is short, otherwise only the data type and size of the cells will be displayed.
  - In the array editor, some cells look confusing. By double clicking on a cell you can see the contents of the cell separately in the usual way.
- Cell array manipulation:
  - The commands `reshape` and `repmat` work for cell arrays like for matrices.
  - `cellfun('functionxy',A)` applies a function `functionxy` to all cells of `A`. (This does not work for all functions, but only for specific ones like e.g. `length` and `isreal`, for more information see help!)
- Important examples of cell arrays:
  - Cell arrays are used in Matlab to generate functions with variable numbers of input and output arguments (see below).
  - Cell arrays of strings provide the opportunity to combine character strings of different lengths.
- Good to know:
  - If you assign a cell to an existing variable that is not a cell, Matlab will stop and report an error.

Structures

- Idea:
  - Structures consist of fields.
  - The user gives a name to each field, describing its content.

- Each field has a specific data type.
- Instances of structures with the same field names can be combined in a matrix-like way.

- Example to visualize a structure: Imagine a backpack filled with books, pens, a moneybag, a memory stick… If you want to go from one room to the next, you only have to grab the backpack, without having to think about taking every individual item with you. However, there is no specific order of the things inside of the backpack.
- Creation:
  - The command `struct` combines variables of different types into a structure variable.

```
stimulus=struct('binduration',0.001,'stimname','flashes1')
% creates a structure stimulus with fields binduration and
% stimname which are filled with the corresponding values.
```

  - Alternatively, a structure is generated when one of its fields is addressed (see below).
- Indexing and working with structures (see script `structuredemo.m` in `struct_cells.zip` at the homepage):
  - Individual fields in structures are addressed by their names, which are connected by a dot to the name of the structure. E.g.

```
tdata.vec=[0.1 8.8];        % generate a structure tdata
                            % with field vec and assign vector
                            % [0.1 8.8] to this field.
tdata.text= 'blabla';       % add a field with name text to
                            % structure tdata
```

  - Individual elements of matrices contained in structures can be addressed directly by usual matrix indexing with the indices written behind the field name.

```
V1=tdata.vec(2);      % returns V1=8.8
```

  - Structures can contain several instances which all have the same field names. These instances are arranged like a matrix. Fields with the same name can have different types. To address individual instances, their indices are written between the structure name and the dot.

```
tdata(2).text=true;         % generates a 1-by-2 array of
                            % structure tdata and assigns the
                            % logical value to field text of
                            % the second instance (even though
                            % it is a string in the first
                            % instance).
M=tdata(2).vec              % returns M=[] because it was not
                            % filled yet.
```

- Symbol in work space window: box with branching lines
- Special functions for structures:

```
fieldnames(s)         % returns names of fields in s
isfield(s,'field')    % true if there is a field 'field' in s
                      % (field names must be given as strings)
isstruct(s)           % true if s is a struct
s=rmfield(s,'field')% removes field 'field' from s
s=orderfields(s)      % sorts fields in alphabetic order (by
                      % default, fields are sorted by the order
                      % of their creation.)
v=getfield(s,'field')   % returns the contents of the
                      % specified field of a one-dimensional
                      % structure. If s is one-dimensional the
```

```
                    % command is equivalent to v=s.field
                    % See help for usage for multi-dimensional
                    % structures.
s=setfield(s,'field',v) % assigns value of v to the specified
                    % field of a one-dimensional structure.
                    % If s is one-dimensional the command is
                    % equivalent to s.field=v See help for
                    % usage for multi-dimensional structures.
```

Functions with variable numbers of input and output arguments

- Idea: The lists of input and output arguments can get very long (in particular if you do not use complex variables…). Matlab provides a way to make input and output arguments optional, so the user only needs to specify them if he / she needs them. The control flow of a Matlab function can depend on the number of input or output arguments specified by the user.

- Functions with variable number of input arguments:
  - Calling a function with too few input arguments causes an error when Matlab tries to use a variable in the program code that is expected to be defined as input argument.
  - However, many of the functions provided by Matlab can be used with a variable number of input elements (e.g. `plot(x), plot(x,y), plot(x,y,'rx-'),…`).
  - You can write functions with optional input arguments yourself.
- Creation of functions with variable numbers of input arguments:
  - The keyword `varargin` as last input argument in the function-declaration line generates a function with optional input arguments.
  - Matlab then generates a cell array with the name `varargin` (this is not a usual variable name, it has to have this name!), whose nth cell contains the nth argument, starting from where `varargin` appears.
  - In the program, you can access e.g. the 3rd optional argument with `varargin{3}`.
  - The function `nargin` determines the number of input arguments specified by the user in the current function call (including non-optional and optional arguments).
  - Syntax example:

```
function a=testfunction(in1,varargin)
%Help text, explanation, comments
if nargin>1 % if there is at least one optional input argument
   disp(varargin{1}) % the first optional argument is displayed
   a=0;
else
   a=27;
end
```

This function could be called (without producing an error) e.g. as

```
>> v=testfunction(27,25)
>> v=testfunction('blabla',true,[8 9 7])
>> v=testfunction([8 9 7])
```

- Functions with variable number of output arguments:
  - You can call any function with less output arguments than are specified in the function declaration line. (E.g. if you call a function without

specifying output variables in the function call and without semicolon at the end, only the value of the first output argument will be stored in `ans`.)

- The control flow of functions can depend on the number of output arguments. E.g. the `hist` function will display a figure if you call `hist(x)` but will not display a figure but give back the histogram values if you call `a=hist(x)`.

- Creation of functions with variable numbers of input arguments:
  - The keyword `varargout` as last output argument in the function-declaration line generates a function with optional output arguments.
  - Matlab then generates a cell array with the name `varargout` (this is not a usual variable name, it has to have this name!), whose nth cell contains the nth argument, starting from where `varargout` appears.
  - In the program, you can access e.g. the 3rd optional argument with `varargout{3}`.
  - The function `nargout` determines the number of input arguments specified by the user in the current function call (including non-optional and optional arguments).
  - Syntax example:

```
function [a,varargout]=testfunction2(in1)
%Help text, explanation, comments
if nargout>1 %if there is at least one optional output argument
     for n=1:nargout-1
          varargout{n}=in1/100;      % assign a value to all
                                     % optional arguments
     end
   a=0;
else
   a=27;
end
```

This function could be called (without producing an error) e.g. as

```
>> v=testfunction2(27)
>> [v,w]=testfunction2(27)
>> [v,w,x]=testfunction2(27)
```

- Alternative way to produce functions with variable numbers of input and output arguments:
  - The functions `nargin` and `nargout` can be used in any function, no matter if they use `varargin` and `varargout` or not.
  - Therefore, an alternative way to write functions with variable numbers of input and output arguments is to:
    - Specify all potentially useful arguments as individually named input or output arguments.
    - Sort these arguments by relevance. The argument that is least probable to be used (e.g. a rarely changed value for experimental specifications like the sample rate) should be placed as last argument.
    - Start the functions with an if-elseif construction depending on the values of `nargin` and / or `nargout`, in which you assign default values to those parameters that were not specified by the user.
  - However, the more elegant way – which is also used by functions provided by Mathworks – is to use `varargin` and `varargout`.

Homework:

- Think about using structures or cell arrays for your data. Would that help you to keep a good overview?
- Take a look at your functions. Would it make sense to make certain input or output arguments optional?