Matlab for PhD students – Advanced Topics 1– WS 2010/11

Data flow and control flow

Software development:
When you start to work on a (non-trivial) programming task, you should take the time to perform the following steps. Even though it might at first seem faster to just start generating code, it will save you a lot of time and trouble to follow the classical course of software development

1. Specify the task
    - A specification defines the problem that a program is supposed to solve.
    - For the specification it does not matter, how the program solves the task, but only what the task is.
    - Specifications can be written in every day language. Some people use specifications later as help-texts for their programs.
2. Define data sources and data sinks
    - Define which data the program will process.
    - Where do these data come from? (E.g. a data file, a different program or user input)
    - Define which outputs the program will generate. (E.g. data files, graphics, outputs to different programs)
    - See later section "data flow"
3. Divide the problem into sub-problems
    - Identify the sub-problems of your task. Usually, the building blocks of a big problem are easy to generate, if the task is divided into sufficiently small pieces. (This concept is called "divide and conquer"). There may be several layers of sub-division until you reach the sub-problems that are easy to program.
4. Develop the algorithms for each sub-problem
    - An algorithm is the sequence of steps used to solve a problem. Develop an algorithm for each of the sub-problems you identified.
    - It is a good idea to think about the algorithm independently of the programming language you will use. Depending on your personal preferences, you could use natural language, flow charts (see below) or pseudo code (see below) to put your algorithm down on paper.
    - Very often it helps to develop algorithms by picking a simple example and writing each step that should happen on a piece of paper.
5. Define data flow and control flow between sub-problems
    - Think about the main structure of your program by combining the sub-problems:
        - In which sequence should the sub-problems be solved?
        - Which sub-problems depend on each of the sub-problems?
        - Which data should be transferred between the sub-problems?
    - At this step, you should also decide how to structure your data. Is it useful to combine variables to more complex data types? (Will be discussed later in the course.)
    - For each sub-problem you should decide if you should use a script, a function or a sub function (see below)

    - Flow charts (see below) are particularly useful to structure the sequence – and the hierarchy – of sub-problems.
    - The result of this step is the complete algorithm for the entire task, including all sub-problems.
6. Implement the program
    - After you have specified the complete algorithm in a language-independent way, you can start to produce Matlab code.
    - First decide for which of the sub-problems the variables should be visible in the main workspace and for which they should be hidden from the user
        - If variables should be hidden, use functions or sub-functions.
        - If variables should be visible, use scripts.
        - It is good style to encapsulate sub-problems into (sub-) functions and to write scripts only to call functions for specific cases e.g. of specific parameter settings.
    - First write code for sub-problems and test it separately, then put the parts together.
7. Make the program safe to use
    - Remember to document your program while you write it. Comments will make it much easier for you when you later have to re-visit your code.
    - Make sure to test all user inputs (input arguments to functions, user keyboard or mouse inputs, maybe also contents of data files) if they meet the specifications of your program. Common sources of hard to detect errors are unforeseen data types, matrix sizes and parameter ranges as inputs.
    - Tests for correct user inputs are usually done before any code is performed. If the user input does not meet the specification, issue adequate error or warning messages.
8. Test, test, test!
    - Of course, you should make sure that your program does not stop with red error lines (unless you issued them on purpose) when you call it.
    - However, this is only the very first step of testing! You also have to make sure that your program produces correct results for all kinds of different inputs. (see script Matalb_adv2_errorhandling)
    - If you detect errors, you need to go back either to step 6 (if the error is due to a problem in the implementation – which is usually the easiest problem to fix), to step 5 (if the communication between sub-problems does not work correctly), to step 4 (if the algorithm of one of the sub-problems is not correct) or even to step 3 (e.g. if you forgot to take care of a specific sub-problem that rarely occurs.)
    - Usually, the process of testing and revising the program takes longer than the implementation itself!
9. Re-visit your comments
    - When the program is written and tested thoroughly you should take the time to go through your entire program with all functions and sub functions again to make sure that your comments are up-to-date and understandable not only for yourself but also for others who want to use your code.
    - In particular, make sure that the help texts are helpful – they should at least specify the input and output arguments and shortly define the purpose of the program.

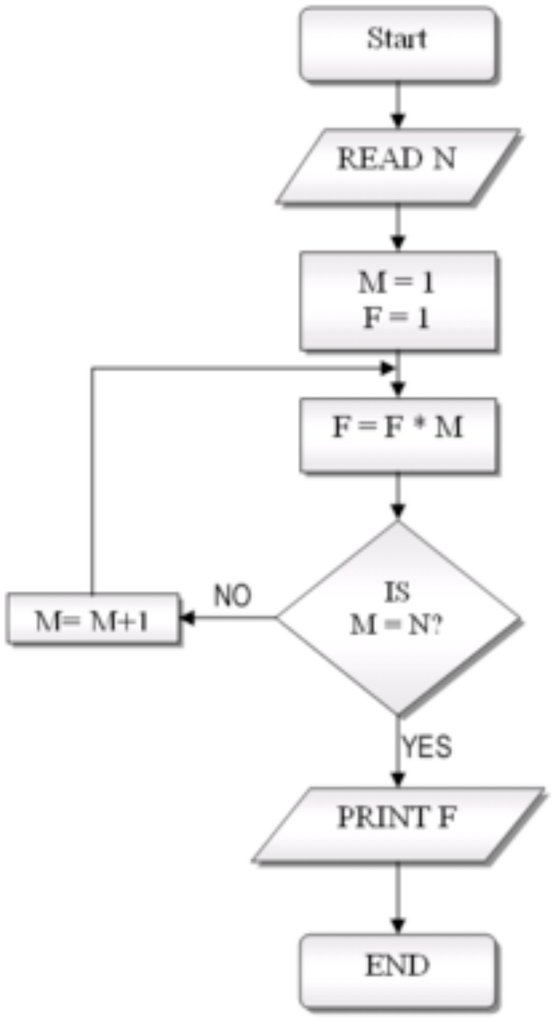
How to structure a big problem:

- Control flow:
    - Most important principle "divide and conquer"
    - First divide your big task (coarsely) into smaller sub-tasks
    - For each of the sub-tasks define the input and output data. Depending on the data flow you can decide to use one big program or to encapsulate sub-tasks into functions (or into separate scripts).
    - If a sub-task could also be used in other contexts you should write a separate function with well-defined inputs and outputs to solve it.
    - Divide each of your sub-tasks into even smaller units. To give your code a clear structure, these blocks can become cells of program code in Matlab (see script Matlab_adv2_errorhandling).
    - It is a good idea to first write a separate file for each sub-task you want to program, in which you define the task and its inputs and outputs in the help text. The sequence of sub-sub-tasks leading to the solution of the sub-task can be defined as cell titles.
    - Which control elements (e.g. loops, case differentiation) are needed to solve the sub-sub-tasks?
    - Try to avoid spaghetti code, aim for lasagna code!
- Pseudocode:
    - Pseudocode is an informal way to define algorithms without having to think about syntax and other language-specific issues.
    - Pseudocode lists the sequence of steps needed to solve a problem by using a mixture of natural language and programming-style writing, e.g.

```
While z is not 0
    Ask the user to type in a new value for z
End
Display value of z
```

    - There are several definitions of pseudocodes, but they are (on purpose) not standardized. Otherwise pseudocode would be a programming language itself that people would need to learn. So feel free to define your own pseudocode!
- Flowchart:
    - A graphical tool to structure control flow. Each program step is represented by a symbol. Arrows show the direction of the control flow.
    - Flow charts are usually drawn from top to bottom. In loops, the control flow re-connects to earlier steps.
    - Symbols:
        - Arrow: Direction of control flow, connecting program steps
        - Circle, oval or rounded edge rectangle: start and end symbols
        - Rectangle: Generic processing step (e.g. x=x+1)
        - Rectangle with double-struck vertical edges: subroutines, which are explained in a separate flowchart
        - Parallelogram: Input / Output (e.g. save a file or get keyboard input)
        - Rhombus (diamond): conditional or decision (usually a true / false test). Two arrows leave the rhombus, pointing to the subsequent program steps, depending on the condition. The arrows should always be labeled.

        - Arrow pointing to another arrow or arrows joining with a blob: junction of control flow (different processing steps are followed by the same next step.)
    - Example (from Wikipedia): Flowchart to calculate N! (the factorial of N)



    - see http://en.wikipedia.org/wiki/Flow_chart

- Data flow:
    - Which types of data should be processed?
    - Where do the data come from? Possible data sources are:
        - Matlab data files
        - Other types of data files (data import)
        - Measurement hardware
        - User inputs
        - Function input parameters
    - Where do the data go? Possible data sinks are:
        - Matlab data files
        - Other types of data files (data export)
        - Hardware control
        - Graphics
        - Sound
        - Text output
        - Function output parameters

    - If you have structured your algorithm into sub-parts, think about which part needs which data and how the data should be transferred between the parts.
- A graphical tool to structure data flow is the data flow diagram, in which data flow is depicted by labeled arrows between functions (usually shown as circles or rectangles). However, there are several graphical notations for data flow http://en.wikipedia.org/wiki/Data_flow_diagram
- Data flow diagrams should not be confused with flow charts! In flow charts, arrows depict the order of programming steps, while in data flow diagrams the order of steps is unimportant and the arrows show the direction of data import and export between functions.

Control elements in programs

- If-else-end construction: Executes a group of commands depending on a logical value. Very often, the logical value is the result of a relational test. More than two alternatives how to continue the program can be given by using elseif statements. (If you feel you want to review if-elseif-else constructions, you should take a look at the course script IntroductionMatlab2 and the corresponding demo program if_demo.m)

- Switch construction: If several different cases should be considered, it is often clearer and more convenient to use a switch construction instead of using several elseif statements or nested if constructions. Switch constructions allow at most one of the command groups to be executed. Switch constructions are particularly convenient if you are comparing strings (to compare two strings you usually need the function `out=strcmp(string1,string2)`, `string1==string2` does not work).

- For-Loop: for-loops are (usually) count loops. They are used when a command block needs to be repeated N times and the number N is known beforehand. Moreover, they are very convenient to use when an algorithm should be run for several specified values of a parameter. (for-loops were topic of day 3 of the introductory part of the course. See script IntroductionMatlab3 and corresponding demo programs for more details.)
- Good to know:
    - Even though a vector (e.g `t=1:5`) is used in the for-statement, in each repetition of the commands the variable has a scalar value corresponding to the N-th element. E.g. in the first repetition `t==1`, in the second `t==2,`..., in the last `t==5`.
    - The most common vector used in for-loops are used for counting (e.g. `i=1:10` or `even_num=2:2:22`), but it is also possible to pre-define vectors (e.g. v=[78; 9; -0.5] and use their values sequentially during the repetitions)
    - You should not change the value of the counter variable in the body of the loop. The value will be overwritten anyway when the next repetition starts.

- While-loops: while-loops are the more general case to repeat a block of statements than for-loops. The program body of while-loops is repeated as long as a certain condition is true. You do not need to know beforehand, how often the commands will be repeated. While-loops were topic of day 3 of the introductory part of the course. See script IntroductionMatlab3 and corresponding demo programs for more details.)

- Good to know:
    - A while-loop only works if the variable used in the condition is defined beforehand
    - The variable used in the condition must be changed (at least under certain conditions) in the body of the while loop. Otherwise, the loop will never terminate.
    - In case you produced an infinite loop, you can interrupt the program with ctrl-c.
    - You can always use while-loops instead of for-loops. See program factorial_demo.m (from the course homepage) for comparison. (However, for-loops are often easier to program and will always terminate.)

Scripts and functions

- Scripts and functions were topic of day 2 of the introductory part of the course. See script IntroductionMatlab2 for more details.

- Scripts
    - Sequence of commands, equal to typing the sequence in the command window
    - Variables in a script are present in the workspace
    - All workspace variables can be used in a script
    - Workspace variables can be changed and cleared in a script
    - Scripts are saved as .m files (you have to give them a name)
    - To call a script, the name of the file is used (type the name in the command window or use it as a command in a script or function).
    - Convenient way to make a sequence of commands reproducible.
    - The most common use for scripts is to call a sequence of functions for a specific data example, e.g. a specific parameter set.

- Functions
    - Sequence of commands, using a separate workspace
    - Workspace variables, which should be used in the function, have to be given to the function explicitly with input arguments, all other workspace variables are not visible within the function
    - Function variables, which should be used in the workspace after the function is executed, have to be given to the workspace explicitly by output arguments
    - The first line in the function file, the function-declaration line, defines the number and names of input and output arguments (all combinations are possible). It must contain the key word `function` and the function name

```
function blabla                % function without input and
                               % output
function blabla (in1)          % function with one input
                               % argument
function out1=blabla           % function with one output
                               % argument
```

```
function out1=blabla (in1)    % function with one in- and
                              % one output
function [out1,out2]=blabla (in1,in2,in3)  % function with 3
                              % in- and 2 outputs
```

- When saving the program, Matlab will suggest the function name as standard file name (in this case `blabla.m`). It is good style to use the same name for the file and the function-declaration line of the function (otherwise it can get very confusing.)
- To call a function, you need to type the filename followed by as many input arguments as you have specified in the function-declaration line of the file. (If you have used a different file name than the function name in the function-declaration line, you need to use the file name.) You also should give as many output arguments as you have specified, otherwise this information will get lost. E.g. call your function with `[a,b]=blabla(A,B,27.5);` (with A and B variables you have defined before)
- It does not make sense to define input arguments in the function-declaration line if they are not used in the function commands! It also does not make sense to define output arguments that do not get assigned a value during the function execution!
- Functions that could be useful in more than one context should be written as general as possible and documented well.
- Functions are the best way to solve a task with defined input and output data, without interfering with workspace variables.

• Sub functions
  - If you have specified a sub-task with defined input and/ or output arguments, which is used more than once in your algorithm, you should encapsulate this sub-task into a function.
  - However, if you are sure that you will not need this specific sub-task in any other programming project, but only as part of the function you are currently working on, you can define a sub-function.
  - Sub-functions are placed in the same file as the main function after the end statement of the main function program code.
  - Sub-functions can be called from anywhere in the main function, but they cannot be called from outside the m-file.
  - Sub-functions have their own workspace and communicate with the parent function only via the input and output arguments.
  - Sub-functions begin with a standard function statement line and follow all rules applying to functions.
  - If you use sub functions you need to end each function with the keyword `end`. (This keyword can be used but is usually not mandatory to mark the end of a function.)
  - It is good style to start the sub-function name with local_ to remind the user that it is a local function. To display the help text of a sub-function, you can use `helpwin parent_function/local_sub_function`
  - Syntax:

```
function out=parent_function(in)
% parent_function description

% code in parent function, calling local_sub_function
```

```
end % of parent_function

function nout=local_sub_function(nin)
%local_sub_function description

%code in local_sub_function working on separate workspace

end % of local_sub_function
```

- Nested functions:
    - Nested functions are functions, which are fully contained in the definition of another function (the “parent” function.
    - Nested functions have their own workspace and can have specified input and output arguments.
    - In addition to their own workspace, nested functions also have access to the workspace and with all variables of the parent function. A nested function can overwrite values of variables of the parent function!
    - Nested functions cannot be called separately from other functions, scripts or the command window. They can only be called from within the same file.
    - Nested functions can be called from anywhere in the parent function code, even though they are usually defined at the end of the parent function.
    - If there are several nested functions, they can all call each other, if they are on the same level.
    - There can be several layers of nested functions. A nested function can only call nested functions on the same or a higher level (see help for details: Types of functions -> nested functions)
    - Nested functions are a way to avoid global variables, but they usually make the code more difficult to read and to debug. Usually, there is no need to use them – I would discourage their use! (Nested functions are allowed in version 7.0 and up.)
    - Syntax:

```
function out=parent_function(in)
% parent_function description

% code in parent function, calling nested_function with an
% appropriate input argument.

   function nout=nested_function(nin)
   %nested_function description

   %code in nested_function can access all variables in
   %parent_function in addition to explicit input argument
   %nin)

   end % of nested_function
end % of parent_function
```

- Global variables:
    - Usually, encapsulation is a very useful and powerful concept, making sure that functions are not able to interfere with the base workspace.
    - However, in some cases it might be more convenient to have a certain variable available in all (or many) functions of a program package (e.g.

your own toolbox), without the need to pass them as input argument to each function (and to take care of the correct sequence of input arguments etc.). In this case a global variable can be used.

  - However, the use of global variables is strongly discouraged!!! In my opinion, they only make sense for constants, which are never changed by any of the functions they are used by.
  - It is good style to use names with all capital letters for global variables.
  - A global variable has to be defined in a workspace and declared global in this workspace and all functions called by this script or function, which should have access to the global variable.
  - Syntax:

```
global MYCONSTANT
```

Homework:

- Draw a control flow chart of your own program. If you have a complicated program, you will need to draw several charts on different levels of abstraction, showing the main problems at the top-level and the corresponding sub-problems.
- Draw a data flow diagram for your program.