Matlab for PhD students – Advanced Topics 2 – WS 2010/11 Error handling

Tests

- Testing a program to make sure that it produces correct results usually takes at least as long as writing the program. A (non-trivial) program hardly ever works correctly for all possible cases straightaway. It requires thorough testing and debugging.
- <u>Prior to testing, use the Matlab editor to look for potential errors:</u>
    - If the code in the Matlab editor looks different from what you expect there is probably a syntax error. E.g.:
        - A string (text) must be displayed in purple. If it is red, you probably forgot the second ‘ to end the text.
        - When key words like `while`, `for` or `if` are underlined in red, the corresponding `end` is missing.
        - When the indentation looks weird, first try to mark the piece of code and click on `Text -> smart indent` (or press `ctrl+i`). If this does not help, look for a forgotten `end.`
        - If you try to start a new code cell with %%, but the editor does not show a new line dividing the codes, there is a syntax error in the previous piece of code – very often a forgotten `end.`
    - The Matlab editor shows you potential errors or pieces of code that could cause unwanted effects with yellow marks and red lines below the suspicious commands.
        - When you move the mouse pointer over the yellow mark, a pop-up box tells you about the potential problem. When you click into the box, you will get more information. Right click (or Ctrl-click) opens a pop-up menu to give you suggestions how to deal with the potential error.
        - Very often, Matlab suggests a replacement. When you press `Alt-Return` (or click the appropriate button) the editor replaces the suspicious code with the suggested one.
        - Matlab considers it to be good style to suppress all command line outputs from within functions and scripts by using ; and will mark all lines without semicolon. If you want to have the command line output, just ignore the red line or click with a right click (or Ctrl-click) on the marked = and choose “suppress this instance” from the pop-up menu.
        - Matlab also marks variables that are defined but not further used in the program. Consider if you really need them or if you can get rid of the command to keep your program as simple and clear as possible and to avoid using more memory capacity than necessary.
- <u>Steps to test a program</u>:
    - It helps to test programs by looking only at one sub-problem at a time. The Matlab editor cell mode (see below) provides a convenient way to do so.
    - Does the program run at all? (-> get rid of all red error messages, make sure the program terminates)
    - Test at least one dummy example (-> easy to know the correct results)

- Test some typical examples (-> data for which you know the correct results or at least the range in which the results should be. If you do not know the exact results, test some examples for which you e.g. know which result has to be the biggest one).
- Test some unusual examples (-> data in a range that could in principle be reached but is not typical, e.g. data with missing values (NaN)).
- Test if the program works correctly for vectors and matrices if both are meant to be possible input arguments (-> try different matrix dimensions). If your program is only meant to work for specific input dimensions, try out if you included proper error messages or warnings if the inputs do not meet the specifications.
- Test the time performance of your program (-> is it fast enough for real-world tasks? You can improve the performance by using the profiler, which will be covered later) .
- Try to use inappropriate input arguments (-> wrong argument data types, wrong number of inputs etc). Add helpful error messages.
- Try to be the most stupid or nasty user you can think of (-> try to misunderstand all user interaction, react inappropriately).
- Give your program to somebody else for testing (-> tell him or her to be nasty. Since people think differently, he or she will probably find some weaknesses you were not aware of).

Debugging
If a program does not produce the expected results, you will have to find out if the error is in the algorithm or in the implementation. Therefore, you need to follow the program step by step and compare the results of the program with your expectations. You can do that by hand (which is often the fastest way in simple cases) or with the debugger provided by the Matlab editor (which is more efficient in more complicated cases).

- <u>Debugging by hand:</u>
    - Include some <u>text outputs</u> to follow the progress of the program. Usually it is sufficient to delete some semicolons. If you want to have nicer text outputs you need to include some `disp` or `sprintf` commands.
    - Add <u>`pause`</u> after steps you want to evaluate.
    - A "clean" way to have text outputs for test purposes, but to easily get rid of them for real-world application is to introduce a logical variable and to <u>make text outputs conditional</u>, depending of the status of the variable. E.g. `testcase=true; a=b+c; if testcase a end` (This works in all programming languages, even if they do not provide a debugger)
    - <u>Try-catch-block:</u> A try-catch block provides user-controlled error-trapping capabilities. In the `try`-part, all commands are executed. If no Matlab `error` is generated, the program continues after the `end` statement. If an `error` occurs, the program does not terminate with red lines but executes the commands in the `catch`-part, making it possible to react to errors in a more specific way (with more specific error messages or maybe with fixing the reason for the error).
      Syntax:

```
try
   commands1
```

```
catch
   commands2   % are executed only if commands1 issue a
               % Matlab error
end
```

- Using the Debugger:
  - Set Breakpoints: Breakpoints are steps in the program at which the program stops for inspection of the current status of the workspace. Breakpoints are introduced by clicking on the "-" between the line number and the desired line in the Matlab editor. A red dot appears, showing that a breakpoint is set. If you make changes to the program, the dot will turn grey until you save the file.
  - Clear Breakpoints: To delete a breakpoint click again on the red (or grey) dot. If you want to clear all breakpoints, click on the symbol with the red cross in the editor.
  - Start the program: If you are working on a function with input arguments, you need to start it from the command window (otherwise Matlab would not know the input arguments). If no input arguments are needed, you can also click on the green arrow symbol in the editor. The program will run until the first breakpoint is reached and stop there. In the command window you will see `K>>` to indicate that Matlab is in debugging mode.
  - Inspect variables: In debugging mode, you have access to all current variables (also for functions):
    - You see the list of current variables in the workspace window
    - You can access these variables by double clicking in the workspace window to open the array editor. (You can even change the value in the array editor, but you should usually restrain from doing so if you want to do your operations reproducibly).
    - Or you can type the name of the variable in the command window and you will get the value displayed as text message.
    - If you place the tip of your mouse pointer on a variable name in the editor window, a small window will pop up, showing type and value of the variable. (This of course only works for variables introduced in program lines, which were already processed.)
  - Continue the program: The current program line is marked with a green arrow in the editor. To continue, you have the following options, depicted as symbols in the editor window:
    - Continue (green arrow): continue the program until the next breakpoint or the end of the file is reached.
    - Step (little blue arrow down): process one line of program code and stop again. The displayed variables will change their values according to the operations performed.
    - Step in (little blue arrow to next page): step into (sub-) function or (sub-) script (if the next line is no call of a function or script, it does the same as step).
    - Step out (little blue arrow up): go back from a (sub-) function or script to the program from where it was called.
    - Exit debug mode (crossed out blue arrow): continue the program to the end without stopping at breakpoints.

- Even more advanced tools for debugging:
  - Parse for syntax errors and inefficiencies: The function `mlint` parses M-files for syntax errors and other possible problems and inefficiencies. E.g. `mlint` will point out variables that are defined but never used, statements that are unreachable etc. It also makes suggestions to improve execution time. Example: `mlint myfunction`
    `mlint` can also be used to check all M-files in a directory, by simply calling it with the directory name instead of the name of a specific M-file.
  - Code analyzer report: Matlab provides a tool to check for potential errors for all files in the current folder. Click on the toothed wheel symbol in the current folder toolbar. You will get a list of actions, including “reports”. Click on “code analyzer report”. You will get a list of all programs in the current folder with the potential errors occurring in those programs. If you click on the line number in the list, the Matlab editor opens the file and jumps to the suspicious line. If you change your code, you can click “rerun” in the code analyzer report to see a new version of the list.
  - Check for file dependencies: The function `depfun` parses an m-file for all file dependencies and points out any files needed for the execution of the m-file and not found in the current search path (see below).
  - Dependency report: Click on the toothed wheel symbol in the current folder toolbar. And choose “reports -> dependency report”. You will get a list of all programs in the current folder with the called functions (“children”) and calling functions in the current directory (“parents”), identifying missing programs.

Common sources of errors

- Unexpected user inputs
  - Almost all programs depend in some form on user inputs – input parameters to functions, keyboard or mouse inputs or data files provided by the user. Be aware that users (even you yourself in a couple of months) usually do not know by heart, which types of inputs the program expects and which will cause chaos.
  - Therefore make sure to test each user input if it meets the specification of your program. This is usually done with an if-statement that causes
    - the program to continue as planned if the input meets the specification
    - a warning message telling the user about potential problems, but the program continues anyway. (But if the user reads the message, he might decide to interrupt the program with `ctrl-c`). E.g.
      ```
      if a==0
       warning(‘division by zero! (a==0)’)
      end
      c=b/a;
      ```
    - an error message and the end of the program if it does not make sense at all to continue the program with the data provided by the user. E.g.
      ```
      if ~isnumeric(a)
       error(‘a must be a number’)
      end
      ```

    - Common tests for user inputs are:
        - Data types: Use commands like `isnumeric, islogical` or `ischar.`
        - Matrix dimensions: If your program requires specific matrix dimensions, e.g. a vector, a square matrix or a matrix of dimensions 10000x3 you should test if the data provided by the user has these dimensions, before you start your computations. If the data does not meet the specifications, issue appropriate warnings or error messages. E.g.
```
[rows,cols]=size(a);
if rows~=cols
 warning('a should be a square matrix!')
end
```
        - Parameter range: Very often, certain values provided by the user only make sense if they are in a certain range. E.g. they have to be negative, or they need to be in the range between 100 and 200, or the need to be integer values (but not necessarily integer variables). E.g. (test for a single integer value between 100 and 200)
```
if length(a)~=1
 error('a must be a scalar value!')
end
if a<100 | a>200
 warning('a should be between 100 and 200')
end
if int32(a)~=a
 warning('a should be an integer value')
end
```
- Case distinctions
    - Case distinctions are needed in almost every program to control the sequence of steps performed and to test for potential problems.
    - Add error messages or warnings for cases which could cause problems (e.g. it is a good idea to test for division by 0 or to check if the values of certain parameters stay in the plausible range during computations.)
    - When you use complicated case distinctions, depending on more than one variable, it can be complicated to see if all potentially possible combinations of cases are considered in the program. It is good style to take care of all "wanted" cases in if-elseif (or nested if-) constructions and to reserve the else-case for all unwanted (and potentially unforeseen) cases, for which your program should issue a warning or stop with an error.

- Random numbers
    - Random numbers generated by a computer are "pseudorandom". Pseudorandom sequences typically exhibit statistical randomness while being generated by an entirely deterministic causal process. Since a computer works deterministically - even though you sometimes might get a different impression ;-) pseudorandom sequences are much easier to produce than real randomness.
    - Pseudorandom numbers are generated with complicated algorithms and depend on a "seed" (some number that determines the initialization of the sequence). In Matlab, this seed depends on the running time of the Matlab session and is re-set every time a Matlab session is started.

(In contrast, in most other languages the seed correlates with date and time). This feature makes it easy to reproduce the same sequence of random numbers. On the other hand, it can be very annoying if you want to produce different sets of random numbers but accidentally produce the same sequence over and over again… If you want to get different random numbers for different Matlab sessions (e.g., when you do simulations on a cluster with several nodes working almost parallel), you can set the seed in your program either manually to different numbers or according to the current time of day using the command:

```
RandStream.setDefaultStream ...
(RandStream('mt19937ar','seed',sum(100*clock)));
```

(see `help rand`)

  - If you need to re-use the same sequence of random numbers, the safest way is to generate them once and save them to a file ("frozen noise")
  - If real random numbers are needed, a computer needs to use a non-deterministic source of randomness (e.g. user keystrokes).
  - When using random numbers in Matlab, make sure that you use the right type for your problem:

```
N=randn(n,m)        % generates a n-by-m matrix N with
                    % normally distributed numbers with
                    % mean 0 and standard deviation 1.
U=rand(n,m)         % generates a n-by-m matrix E with
                    % uniformly distributed numbers
                    % between 0 and 1.
I=randi(rmax,n,m)   % generates a n-by-m matrix I with
                    % uniformly distributed random integer
                    % numbers between 1 and rmax. All
                    % numbers in I are independent of each
                    % other.
P=randperm(rmax)    % generates a vector P with random
                    % permutation of the integers 1:rmax
                    % (each number occurring only once in
                    % random order)
```

  - Keep in mind that you usually need quite a big data sample to represent a certain distribution with your sample. If you need to use only a limited number of random numbers e.g. for an experiment that cannot last several days, it makes sense to look at the sample of random numbers (e.g. with `hist`) and maybe produce a new sequence if the distribution does not look the way you want it to look.

- Directories or file names
  - Some platitudes….
    - When Matlab suddenly seems not know your program any more, your current directory is probably not the directory in which you saved your program.
    - Make sure that function file names match the function name used in the function header to avoid confusion (if they do not match, you have to use the file name to call the function).
    - When your program does not change its behavior, even though you changed the code, make sure you saved it under the correct name in the correct directory.
    - If you want to start your Matlab sessions generally in a different

directory than Matlab does with the installation on your computer, you can define the directory in the script startup.m e.g. by the command

```
cd /Users/jutta/Matlab/Matlab_PhD10/
```

The script startup.m can contain any commands you want to be executed when a new Matlab session is started. It needs to be saved to the default startup directory.

- It often helps to use relative paths for file names. E.g.

```
load subdir1/subsubdir5/myfile
% loads data file myfile.mat from the directory
% with the name subsubdir5, which is contained
% in the directory subdir1, which is a
% subdirectory of the current working directory.
save ../newdata v
% saves variable v to data file newdata.mat in
% the directory one level higher than the
% current directory.
```

- Some helpful commands for file handling:

```
exist('filename.mat','file')       % Returns 2 (!) if file exists
                                   % in current directory (or in
                                   % directory specified by path
                                   % in name) and 0 if not
whos –file filename.mat            % displays variables contained
                                   % in file in command window
w=whos(' –file','filename.mat')    % returns struct array with
                                   % fields name, size, bytes,
                                   % class of variables contained
                                   % in file
delete filename.ext                % deletes file on disk
copyfile(oldname, dirname)         % copy file oldname to
                                   % directory
copyfile(oldname, newname)         % copy file oldname to newname
cd directory                       % changes the current
                                   % working directory
```

- The Matlab search path
  - The search path is the subset of all the folders in the file system, which Matlab searches for programs when interpreting program code. The search path specifies the order of directories Matlab considers in its search for programs.
  - Sequence in which Matlab interprets code: If the word `stupidname` occurs in a script or function, Matlab performs the following sequence of tests:
    1. Is `stupidname` a variable name defined in the current workspace?
    2. If it is not a variable name, is it a subfunction in the file in which `stupidname` occurs?
    3. If not, is it a private function to the file?
    4. If not, is it the name of a built-in function?
    5. If not, is it the name of a function or script in the current directory?
    6. If not, is it the name of a function or script in the first directory listed in the search path?

7. If not, Matlab visits the second, third,… until last directory listed in the search path.
8. If none of these directories contains a program with the name, Matlab will issue the error `‘??? Undefined function or variable 'stupidname'.’`

- By default, the Matlab search path consists of the folders in which Matlab and its toolboxes are installed on your computer.
- Note that if two files with the same name are somewhere in the search path, only the one which is visited first will be executed.
- The path browser: you can open the path browser by clicking on `File -> set path.` In this graphical user interface, you can see the Matlab search path with the order in which folders are searched. You can add or remove folders and change the order of folders by clicking on the appropriate buttons.
- Commands to change the search path:

```
matlabpath    % displays the Matlab search path for
              % your computer in the command window
p=path          % writes the current Matlab search
                % path to string variable p
addpath(‘folder1’,’folder2’,…) % adds the specified
              % folders to the top of the search
              % path. Use the full path name for
              % each folder.
rmpath('folderName') % removes the specified
              % folder from the search path.
path(path,'newpath') % adds the newpath folder to
              % the end of the search path.
              % If newpath is already on the
              % search path, then path(path,
              % 'newpath') moves newpath to the
              % end of the search path.
```

Using the cell mode in the editor

- Cells are usually short blocks of program code belonging together to solve a sub-task.
- Cells are marked by %% in the line before the program lines start.
- %% can be followed by a comment. These comments are displayed in bold, a very convenient way to have titles for programming blocks.
- You can run individual cells without the rest of the program by clicking on the cell (the current cell is marked in yellow) and clicking on the symbol “arrow right on yellow block”
- This is also a convenient way to debug the code for a sub-problem
- To run a cell and to continue running afterwards, click on the symbol “arrow down from yellow block”
- To run the entire program, click on the symbol “red arrow down” (or the regular green arrow symbol in the bar above)
- In Matlab 2008, the display of cells can be compressed in the editor to just the title line by clicking on a little minus symbol left of the title line. This gives a nice overview over the structure of the program.
- When you use the cell mode in the editor, you can use `File -> publish` from the menu to generate a well-structured html or pdf document, showing the program code, the computation results and the resulting figures.

Create your own toolbox

- If you consider your set of scripts and functions to be (more or less) complete, tested and ready to use, you can place them in a new subdirectory e.g. MyToolbox in the toolbox directory to permanently include them into your Matlab search path.
- Write two additional files containing only Matlab comments:
    - `Contents.m` lists all m-files contained in the toolbox (with input and output arguments). The first line in the `Contents.m` file should specify the name of the toolbox, the second line the toolbox version and date.
      ```
      helpwin MyToolbox   % displays the file Contents.m in
                          % the Matlab help window.
      ```
    - `Readme.m` describes late-breaking changes or descriptions of known bugs or undocumented features.
      ```
      whatsnew MyToolbox  % display the file Readme.m in
                          % the help window.
      ```
- For your toolbox directory, it makes sense to use the reports provided by Matlab (click the toothed wheel in the current folder toolbar). In addition to the code analyzer and the dependencies report discussed today, you will find the help report, helping you to write good help texts, and the contents report to see if your contents.m file is up-to-date.

Homework

- Add a sub-structure to one of your programs by using the Matlab editor cell mode.
- Use the debugger to step through one of your programs (ideally a program with sub-functions) and inspect how the variables change their values.
- Test if your program is foolproof. Add appropriate warnings or error messages to prevent wrong input arguments from causing chaos.