Matlab for PhD students – Advanced Topics 4 – WS 2010/11

"Speeding up"

General consideration:

- Memory vs. time: Optimally, you want to write code, which is efficient, both for the usage of memory and CPU time. However, there is a tradeoff. For example, it usually is better not to perform the same complicated computation twice. But if the result is so big that it cannot be kept in memory, it may be the better way to do so. It depends on the amount of data you are processing and on the computer system you are using if you should try to optimize memory or processing time in the first place.
- Do not overburden your computer: In general, if you want to perform very time or memory consuming computations, it is better not to run other programs in parallel on the same computer.
- Optimize your code as last step of implementation: Premature optimization can increase code complexity unnecessarily without providing a real gain in performance. Your first implementation should be as simple as possible. Then, if speed is an issue, use profiling to identify bottlenecks.
- Matlab is an interpreted language: Matlab interprets programs each time they are started. This feature makes Matlab easy and convenient to use (you just start your program as you write it), but it requires additional time, e.g. for checking the syntax every time a program is started. In contrast, programs written in other languages like C, python or java have first to be translated into machine code. This compiled version of the program can then be started and is executed directly by the CPU without the need of an additional interpreter program. In Matlab, exceptions of the rule that all code is interpreted are built-in functions and MEX-files and compiled standalone applications, which were written in Matlab.

Memory management:

- How Matlab uses memory:
  - It is possible that Matlab runs out of memory.
  - Matlab allocates memory for variables as they are created and for M-file functions as they are used.
  - When variables are copied, at first both names point to the same block of data. Only when one of the variables is changed, Matlab duplicates the data in the workspace and modifies one of the data blocks.
  - Similarly, the content of function input arguments is only copied to the function workspace if its content is changed. Exception: if input and output arguments have the same name, Matlab copies the variable immediately.
  - The `clear` command frees up memory. I would strongly recommend the frequent use of this command ;-)
  - It is possible that memory becomes fragmented over time, in particular if you increase the size of existing arrays (e.g. in loops). For garbage collection use the `pack` command. It saves all workspace variables to disk, clears everything and reloads the variables.

- Tips to save memory:
  - Load only as much data as you need from files. If you load mat-files containing several variables, you can selectively load only the variables you need with `load(filename, variables)`. To find out which variables of which size are contained in the file you can use the command `whos` with the option '`-file`' (The syntax will become clear when you know structures):

```
S=whos('-file','testdata')      % returns a structure
                                % with 9 fields, in which
                                % the properties of each
                                % variable in the file
                                % testdata.mat are listed
                                % (including name and
                                % size)
n3=S(3).name                    % name of the third
                                % variable in the file
s1=S(1).size                    % size of the first
                                % variable in the file
```

  - Do not define variables or function arguments if you do not need them.
  - Pre-allocate sufficient memory for your variables by initializing them with the command `zeros` or `ones` before you fill them in loops. This will help to minimize fragmentation.
  - Clear variables as soon as they are no longer in use.
  - If you have to process very large amounts of data, it may be necessary to divide them into smaller pieces, store them to the hard disk and process them only piecewise.
  - Use sparse matrices where possible.

- Sparse matrices:
  - Idea: Sparse matrices are a special data type to reduce memory requirements. The use of sparse matrices makes sense for big matrices (e.g. `max(size(A))>100`) which contain a high percentage of zero elements. In sparse matrices only the nonzero elements are stored together with two sets of indices identifying the row and column position of these elements.
  - Symbol in workspace window: box with three diagonal lines
  - Working with sparse matrices: All Matlab array manipulations and most operations work equally well and with equal syntax on sparse and on full matrices (e.g. `e=s(5,9)`). Operations on sparse matrices in general produce sparse matrices. If full and sparse matrices are combined by operations, a sparse matrix will be generated unless it becomes too densely populated.
  - Generation: Sparse matrices are generated by the Matlab function sparse. E.g.

```
A=[0 0 0; 0 9.5 0; 1 0 0]; S1=sparse(A)
```

    Results in:

```
S1 =
   (3,1)        1.0000
   (2,2)        9.5000
```

  - Conversion: The Matlab function `full` converts a sparse matrix to a conventional full matrix. E.g. `F=full(S1)` gives

```
F =
      0            0            0
```

```
         0            9.5000       0
         1.0000       0            0
```

- Test function if a matrix is sparse: `issparse(s)`
- Some useful functions for sparse matrices:

```
find       % find indices of nonzero elements
           % (like in full matrices)
nonzeros   % values of nonzero matrix elements
           % (like in full matrices)
speye      % creates sparse identity matrix
spfun      % apply function to nonzero elements
spones     % replaces nonzeros with ones
sprand(S)  % has the same sparsity structure as Matrix S,
           % but uniformly distributed random entries
sprandn(S) % sparse normally distributed random matrix
spy        % visualize sparsity pattern
           % (useful for raster plot!)
```

- Caution! Sparse matrices are limited to two dimensions!

Time management:

- Initialize variables: It takes time to allocate memory. Therefore, initialize variables before using them in loops to save time (but you may not want to allocate more memory than potentially needed if memory usage is critical).
  Example: It is much faster to use

```
vec=zeros(1,1000);
for z=1:1000
     vec(z)=myfunction(z);
end
```

  than to use

```
vec=[ ];
for z=1:1000
     vec=[vec myfunction(z)];
end
```

  But if the function `myfunction` could be applied to a vector it would of course be best to use

```
z=1:1000;
vec=myfunction(z);
```

  See script `random_numbers_tictoc.m` (download from course homepage, included in updown.zip) for comparison of different ways to generate a big matrix.

- Vectorized solutions: In general, it is always better to use vectorized solutions than to use loops. Matlab is optimized to solve matrix problems quite efficiently. Vectorized solutions are the “Matlab way” of programming! You will see the difference when running the script `updown_performance.m` (download from course homepage, included in updown.zip, see below)

- JIT-Accelerator: Matlab provides the so-called JIT-Accelerator for for-loops, making them (almost) as fast as vectorized code. The for-loop has to fulfill these requirements:
  - The loop contains only the data types logical, character string, double-precision and less than 64-bit integer.
  - Arrays are three-dimensional or smaller .
  - All variables used in the loop are defined prior to loop execution.

- For all variables used in the loop the memory needs to be pre-allocated and maintain constant size and data type for all loop iterations.
- Loop indices are scalar quantities, such as index z in z=1:27.
- Only built-in Matlab functions (see below) are called within the loop.
- Conditional statements (if-then-else or switch-case) involve scalar comparisons.
- All lines within the block contain no more than one assignment statement.

- Built-in functions:
  - Functions that are frequently used are often implemented as executable files in Matlab. They were compiled by Mathworks to produce fast-running machine code that does not need to be interpreted for every function call. These functions are called built-ins.
  - To identify built-in functions, you can use the `exist` function. It returns the number 5 if it is applied to the name of a built-in function, e.g.

    ```
    exist reshape
    ans =
         5
    ```

    or:

    ```
    if exist('reshape','builtin')==5
         disp('Built-in function!')
    end
    ```

    (Some other outputs of `exist` are: `0`: name does not exist, `1`: name is a variable in the workspace, `2`: name is a file on the search path (so also regular functions will return `2`), `7`: name is a folder; see help for more information)
  - You can read the code of "regular" Matlab functions by using the `type` command (e.g. `type factorial`). However, for built-in functions (e.g. `type reshape`), Matlab will only display that it is a built-in function (because Mathworks wants to sell licenses and it would be too easy just to copy all programs…).
  - With the command `which` you can see the path of the program code file (e.g. `which factorial`), so you can open, read and copy the program. For built-in functions, you will also find a program, but it consists mainly of the help text and only calls the executable built-in function at the very end.

- Tips to save CPU time:
  - Use vectorized solutions or JIT-acceleration if possible!
  - Avoid changing the size of variables. Pre-allocate all memory you need for a variable.
  - Avoid unnecessary steps of computations. Make sure that all steps in loops have to be performed in all iterations of the loop. E.g. if a variable is set to a fixed value, you should do so before the loop starts and not over and over again.
  - Function calls take time (e.g. because a new workspace has to be generated and closed again). Sometimes it is faster to re-code a simple function in your program than to call the function. (But for the sake of clarity of your code you should only avoid the function if it turns out to be really critical for the performance of your program!)
  - The `find` function is slower than the use of logical indexing.

- Built-in functions are usually faster than other functions.
- [ ] consumes time, ( ) does not. [ ] is slower than the `cat` command to concatenate arrays.
- Loading and saving data to hard disk is time consuming and should be limited to the necessary amount.
- The most time consuming aspect of many Matlab programs are outputs: drawing (and opening!) figures and writing outputs to the command line considerably slows down the program.

Tools to improve the performance:

- Time watch: `tic` starts a time watch, `toc` reads the time elapsed since the last `tic` command and displays it on the screen. The result depends e.g. on the time used for opening figures or displaying text between the two commands. However, the CPU time also can increase when the computer is busy with other programs.
  Example: `tic; plot(rand(50,5)); toc`
  (See script `random_numbers_tictoc.m` from the course homepage for an extensive example.)

- CPU time: The function `cputime` returns the amount of CPU time in seconds that Matlab has used since the current session was started.
  Example: `t0=cputime; plot(rand(50,5)); t1=cputime-t0`

- Debugging tools: The debugging command `mlint` and the `code analyzer report` (see script of lecture 2) both give tips to improve the performance of your programs concerning CPU time and memory usage.

- Profiler: Matlab provides a specific tool with graphical user interface to optimize the execution of M-file functions, the profiler.
  - The profiler identifies which lines consume the greatest amount of time relative to the rest of the code.
  - Very often, it is not obvious which lines are the most time-consuming, because many commands call other functions, which in turn call functions etc. If one of the called functions consumes a lot of time, the profiler will show that to you.
  - You can start the `profiler window` either from the Matlab desktop (click desktop -> profiler) or use the `profile` command to start profiling and the command `profile viewer` to open the profiler window.
  - Example:

```
clear all                          % make sure that no memory is
                                   % used
profile on;                        % starts to record CPU time
for i=1:100                        % it is a good idea to run your
                                   % program several times to
                                   % average out effects of other
                                   % tasks your computer performs
                                   % additionally.
      out=myfunction(in);          % your function
end
profile viewer                     % opens the profiler GUI or
                                   % refreshes the window if it
                                   % is already open.
```

- If you click in the profiler window on the name of program you are interested in, the profiler shows the lines where the most time was spent and additionally displays the `mlint` results.
- The profiler is also a great debugging tool, because it also works for programs terminating with an error message (and program interrupted with Ctrl-c). So you can keep track of what was going on prior to the error.

- Optimize your code with the profiler:
  - Start the profiler
  - Run your program (if possible several times)
  - Look at the profile summary report in the profiler window, click on the name of the program you want to optimize
  - Look at the detail report to see which lines consume most time.
  - Click on the links to get to the lines in the Matlab editor.
  - Try to optimize the most offending lines
  - Use `clear all`
  - Run your program again by calling it from the "run this code" line in the profiler window to see if the performance increased.
  - You might want to keep a copy of your original profiler window for comparison.

Matlab and C

- MEX-files: You can call your own C or Fortran subroutines from Matlab as if they were built-in functions. Matlab callable C and Fortran programs are referred to as MEX-files. MEX-files are dynamically linked subroutines that the Matlab interpreter can automatically load and execute. Applications of MEX-files include:
  - Large pre-existing C and Fortran programs can be called from Matlab without having to be rewritten as M-files.
  - Bottleneck computations that do not run fast enough in Matlab (usually computations requiring complicated loops) can be recoded in C or Fortran for efficiency

  MEX-files are platform specific and also have platform specific extensions, e.g. mexmac (mac), maxw32 (32 bit windows) and mexglx (32 bit linux). It is not possible to use a MEX-file on a different platform than on the one it was compiled on.
  You can call a MEX-file in Matlab exactly like an M-file.
  To consider which compiler to use and to learn how to build MEX-files please refer to the help page "Building MEX-files".

- Matlab Engine: The Matlab-engine allows the use of M-files from within C. So you can use specific functions in a faster environment.

- Matlab Compiler: Matlab provides a compiler to create standalone applications and shared libraries for C and C++. The Matlab compiler is not part of the standard Matlab, but needs to be paid separately like a toolbox. The compiler works with all Matlab functions and most toolboxes. The compiler comes with a graphical user interface to build and package components for deployment on a different computer.

Example for speeding up Matlab code and using the profiler

- Task: Up-Down Sequence: Let `N` be a positive integer. If `N` is even, divide it by 2. But if it is odd, multiply it by 3 and add 1. Repeat until `N` becomes 1.
- This algorithm converts always to 1, but it takes different numbers of iterations to converge. We calculate the numbers of iterations required to achieve convergence for each elements of a vector `Nums`.

- Example programs: Example programs are on the course homepage in `updown.zip`. They are modified from the book "Mastering Matlab 7". You will find several versions of the algorithms described below. The script updown_performance.m calls all of these versions with the profiler to compare the performance of the different versions of the algorithm.

- First approach: Algorithm based on a for-loop: `updown2.m` (see file for full version, only the main algorithm is shown here, without test for correct function input etc…)

```
function Counts=updown2(Nums)
% parts are missing here...
%% calculate lengths of up-down sequences for each vector element
for z=1:length(Nums)
   N = Nums(z);     % number to test in this iteration of for loop
   count = 0;       % while loop iteration count
    while N>1
       if rem(N,2)==0    % even numbers
          N = N/2;       % are divided by 2
          count = count+1;
       else              % odd numbers
          N = 3*N+1;     % are multiplied by 3 and 1 is added
          count = count+1;
       end
    end
 Counts(z) = count;      % attach current count of iterations to
                         % the list

end
```

  This algorithm is used in the functions `updown2, updown2f` (calling `updown1`) to show the performance difference between using function calls versus including the code into the main program. On my computer there is not much difference between the performances of these versions.

- Optimization of algorithm: This algorithm can be made faster by using the fact that for an odd number `N` the computation `N = 3*N+1;` always yields an even number `N`. Therefore, the case for odd numbers can be replaced by

```
      else          % odd numbers: do next two steps together
           N = (3*N+1)/2;
           count = count+2;
      end
```

  This algorithm is used in the functions `updown2a`, `updown2af` (calling `updown1a`), `updown2nestedfunc` and `updown2subfunc` to show the performance difference between using function calls, calls of sub functions and calls of nested functions versus including the code into the main program. Again, there is not much difference between all of these versions on my computer. However, all of these versions show a clear reduction in processing

time compared to the first approach.

- Optimization of implementation: Due to the command `Counts(z) = count;` the variable `Counts` changes size in each iteration of the for loop. Matlab should be able to use JIT acceleration for the for-loop when all variables are pre-allocated. Therefore, the following code cell is added to the program prior to the for-loop:

```
%% pre-define variables:
Counts = zeros(size(Nums)); % preallocate array. This should make
                            % a big difference for speed!
N = Nums(1);                % predefine N data type and dimension
count = 0;                  % predefine count data type and
                            % dimension
```

  This algorithm is used in the function `updown2a_predef`. According to "Mastering Matlab 7" this addition should make a big difference in performance. However, on my computer it does not…

- Vectorized solution: The more elegant way to solve this task in Matlab is (as usual) to use a vectorized solution instead of the for-loop. However, it is not trivial to deal with the entire vector `Nums` instead of its individual elements, because different numbers of iterations are needed for each vector element. The idea is to use logical indexing to extract a vector of elements that are larger than one and odd and a vector of elements that are larger than one and even and to process all of these elements in one step:

```
%% pre-define variables:
N = Nums;                  % duplicate numbers
Counts = zeros(size(N)); % preallocate array
not1 = N>1;                % True for numbers greater than one

%% lengths of up-down sequences for all vector elements together

while any(not1)                % true if any element of vector not1
                               % is a nonzero element
   odd = rem(N,2)~=0;          % Vector with 1 for odd values and 0
                               % for even
   odd_not1 = odd & not1;      % Vector with 1 for odd values
                               % greater than one
   even_not1 = ~odd & not1;    % Vector with 1 for even values
                               % greater than one
   N(even_not1) = N(even_not1)/2;           % Process evens
   Counts(even_not1) = Counts(even_not1)+1;
   N(odd_not1) = (3*N(odd_not1)+1)/2;       % Process odds, next
                               % two steps together like in
                               % updown2a
   Counts(odd_not1) = Counts(odd_not1)+2;
   not1 = N>1;                  % Find remaining numbers
end
```

  This algorithm is implemented in the function `updown_vector`. The improvement in processing speed compared to the for-loop algorithm is striking. On my computer it only takes ~1/60 of the CPU time!

- Optimized vectorized solution: Profiling shows that the line
  `odd = rem(N,2)~=0;`
  takes a lot of the total CPU time used by function `updown_vector`. The reason is the call of the function `rem`, even though this is a built-in function.

"Mastering Matlab 7" suggests the replacement of the line by

```
odd = (N-2*fix(N/2))~=0;
```

This algorithm is implemented in the function `updown_vector_opt`, which indeed shows a small additional improvement (on my computer 20%) compared to `updown_vector`.

Homework:

- Test the processing times of the different examples of `updown` and the generation of random numbers on your own computer.
- Use the profiler to optimize the code of your own project.