Scientific Software
Exercise session 6: Debugging, timing and profiling in Matlab

Wouter Baert, Emil Loevbak & Pieter Appeltans

Leuven, November 12, 2020

Introduction

In this session we will work with Matlab. Matlab is designed to easily write code for numerical experimentations. However, it is often necessary to optimise this code using the profiler and the Code Analyser. This session aims for you to develop an understanding of the following concepts:

- Debugging in Matlab
- Using the profiler in Matlab
- Writing efficient code in Matlab

As this is a graded session, you should work **individually**. Submit your most optimised version of `mandelbrot.m` and all your revisions `mandelbrot_i.m` as a zip with name `lastname_firstname_studentnumber.zip` on the Toledo exam platform by the end of the session at 13h. Give a short explanation for each of the following questions in comments at the bottom of `mandelbrot.m`. Also mention which version of Matlab you used in the comments. **After each question save your code as `mandelbrot_i.m` with i the question number, but continue working in `mandelbrot.m`.**

In this session, we will examine the Mandelbrot set. To find an approximation for this set in a certain region in the complex plane, we will check for each point c on a grid in this region how long it takes before the sequence

$$z_r = z_{r-1}^2 + c, \quad \text{met } z_0 = c \quad \text{voor } r = 1, 2, \ldots$$

produces elements whose modulus is bigger than 2. By converting the critical index $\tilde{r}$ to color code for each grid point, one obtains the well-known Mandelbrot figures.

Download the following files from Toledo: `mandelbrot.m`, `mandelbrot_99.p` and `mandelbrot_driver.m`. `mandelbrot.m` is used to determine the critical indices $\tilde{r}$. `mandelbrot_driver.m` shows the resulting Mandelbrot set (left) and the error (right), which is calculated using `mandelbrot_99.p`.

Matlab debugger

Start Matlab on your local machine.
Remark: If you are on a Linux machine, you may wish to change the keyboard shortcuts from the default Emacs bindings to the Windows defaults in HOME ← ENVIRONMENT - Preferences ← Shortcuts - Active settings.
Hint: You can use `help functionname` in the Command Window to get more information about a function.

To start with, we will use the graphic debugger of Matlab to resolve a bug in `mandelbrot.m`.

- You can place a breakpoint at a certain line by left clicking on the dash in the breakpoint alley. Upon reaching the breakpoint, Matlab will stop the execution of your code and you can use the command line to check intermediate results. You can for example inspect the value of variables in the current workspace. **Question 1:** What is the value of `mandelbrot_fun` after executing line 17 in `mandelbrot_driver.m`?

- The buttons in the Editor menu, at the top of the screen, allow you to control the debugger. You can resume the execution of your code with `Continue`. With `step` and `step in` you can step through your code. **Question 2:** What is the difference between `step` and `step in`? Test this near the call to `mandelbrot` in line 22? With `Step out` Matlab will continue executing your code till the end of the function or script in which you currently are.

- Now find the bug in `mandelbrot`. The bug is somewhere around the assignment `R_tilde(m,n) = r` on line 32. Put a beakpoint on this line. Remark that this is not really practical because you stop for each grid point. To avoid this, we'll use a conditional breakpoint (right click on your breakpoint to make it conditional). Make sure Matlab only stops if `m==1` and `n==1` and try to resolve the bug. Make sure $\tilde{r}$ really is the *first* index for which the modulus of z_r is bigger than 2. **Question 3:** What was the bug and how did you solve it?
  Hint: If you are unable to find the bug, you can request a hint by sending an email to `pieter.appeltans@kuleuven.be` with HINT as the subject line. If you are still unable to figure it out you can send an email to the same address with subject BUG to get an explicit solution

Matlab profiler

Using the profiler in Matlab, it is easy to check what the most expensive code fragments are, qua number of calls or qua execution time. These fragments offer the greatest opportunity to reduce the execution time (Amdahl's law). You can start the profiler using the Run and Time (profiler) button in the EDITOR menu or using HOME → CODE - Analyze Code (Code Analyzer).

Using the information provided by the profiler, we are going to optimize `mandelbrot.m` in several steps. Don't forget to save your code after each step as `mandelbrot_i.m`. Use the driver to check whether the result is still correct! A correct result is far more important than a fast program.

- Execute the `mandelbrot_driver.m` using the profiler and go through the provided report. **Question 4:** What is the difference between *self-time* and *total-time*? What is the most computationally intensive function? Do you look at *self-time* or *total-time* to find this?

- Click on `mandelbrot` in the profile report. Look for the *code-coverage*. **Question 5:** Is there code that is not executed? If yes, why can these lines be removed here?

- **Question 6:** Have a look at the warnings the Code Analyzer gives about `Z`,`C` and `R_tilde`. Fix this. Is there a difference in execution time? How do explain you this?

- **Question 7:** Look into the tip the Code Analyzer gives in connection to the complex numbers. Does this save time? Is there another reason why this modification is benificial?

- **Question 8:** Look in the profiler for lines of code that require a lot of time. What can you say about the test `if r == 0`? In what way could you avoid this?

- **Question 9:** Does the order in which the matrices elements are accessed, matter? To this end, exchange the two innermost loops and look at the difference in time. What do you notice.

- **Question 10:** The profiler probably shows that your solution for the bug in question 3 is an expensive line. Indeed, after a few iterations, there are only a limited number of grid points still active. Determine the success rate of this test using the profiler output. Use `find` to determine the indices m and n for which this test will succeed. Then iterate over these indices instead of over the full matrix `Z`.
  Example: `[M,N] = find(A>0.4)` returns two vectors, `M` and `N`, with respectively the row- and column indices of the elements in A that are larger than 0.4.

- **Question 11:** Even multiple dimensional arrays can be index using so-called *linear indices* in Matlab. This indexation corresponds with the linearised way in which these matrices are stored in memory. Make a revision in which you use linear indices instead of m and n. Does this change the execution time?
  Example: use `I = find(A>0.4)` to find the linear indices of the elements of A that are larger than 0.4.

- **Question 12:** Can you eliminate the loop over these linear indices using vectorisation? Be careful when assigning the values of `R_tilde`. Use a second `find` and an extra indirection to avoid reintroducing the bug mentioned before!
- **Question 13:** And now the Code Analyzer comes to the rescue again with a tip about avoiding the second `find`. Do you notice a big difference doing this?
- **Question 14:** In the initialisation of the variables you can avoid explicit loops using `linspace` and `meshgrid`. Do you see how?
- **Question 15** Do you still see possibilities for optimisations? Briefly try some things. (Tip: retrieving an element from an array requires a bit more work than reading out a variable).