MSAN 502 - Homework 2

Andre Guimaraes Duarte

July 27, 2016

1

My previous version of the **eliminate** algorithm was built around a big `for` loop (*for every row in matrix A, perform elimination on next rows...*). While this works fine for p̈erfectc̈enarios, it was not versatile enough to bypass temporary failures. In order to account for them, and therefore allow permutations, I had to modify my code into a big `while` loop (*while we haven't reached the bottom of matrix A, perform elimination. If we can't, try permuation*). My `Eliminate.py` file is commented well enough (in my humble opinion) for someone to understand what every line is doing.

Some big changes comparing to the previous version is that my algorithm now accepts non-square matrices (although it is still not entirely useful for this week's assignment). $n \times n$ cases are handled fairly well in my tests. I have a few checks in place as well, to make sure that everything is set up correctly in order for my algorithm to work without hiccups (A and b must be instances of `np.array`, their dimensions must match, etc).

My code solves $Ax = b$ when there is a unique solution, even if permutations of rows are needed. Print statements (original matrix, reduced matrix, solution vector, etc) have been commented out in order to run the complexity problem below, but can be de-commented as needed.

My algorithm runs into a permanent failure if the **solvability check** fails. This means that, at one point in the algorithm, we reached a row where we have 0 on the left side of the equation and $c \neq 0$ on the right side, which is an impossibility. By calculating the number of pivots and comparing to the number of columns, I also have the number of special solutions.

If there are no special solutions, then it means there is only one solution. In the other case, there are infinitely many solutions. Again, print statements may have been commented out in order to run the complexity problem. In both of these cases, my algorithm runs to the end. If there is only one solution, backwards substitution finds the unique solution. I haven't yet integrated the scenario of infinitely many solutions (finding the particular and special solutions).

2

Time complexity of my implementation of this algorithm can be found in `complexity.py`. I run my algorithm ten times to solve $Ax = b$, where A is a $n \times n$ matrix with random noise for $n = 2^k$, where $k \in [2; 12]$. I save the average run time for each n.

Then, I use `complexity_plot.py` to plot $\log time$ as a function of $\log n$. The resulting plot is shown in figure 1.

We can see that there seems to be a linear correlation between $\log time$ and $\log n$. The slope of the most linear part (between $n = 128$ and $n = 4096$), the slope is $\frac{\Delta y}{\Delta x} = 2.05$. Since we have plotted a log-log curve of the time taken by the algorithm as a function of n, the slope is actually equal to the run time complexity. Therefore, empirically, the run time complexity of my implementation of the elimination algorithm is $O(n^2)$. This result is actually rather surprising, since we know that the complexity of the full *eliminate* algorithm is $O(n^3)$. To the best of my knowledge, I am performing the algorithm in its entirety, so there are four possibilities:

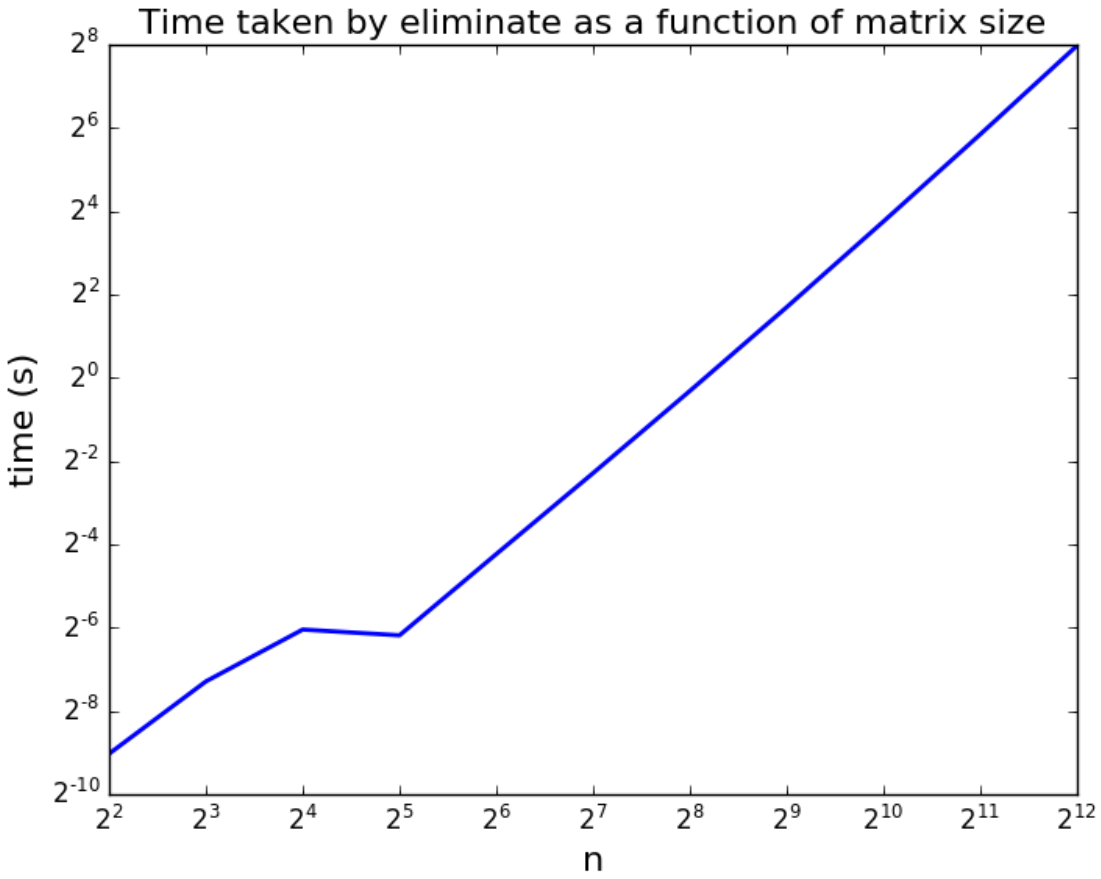

Figure 1: Time complexity of eliminate algorithm

- Either I didn't implement the elimination algorithm in its entirety (but I honestly don't think this is the case),
- or some functions that I am using are very well optimized in python, which brings the time complexity down,
- or there is a difference between computational complexity and run time complexity,
- or I should run this test for even bigger ns in order to reach the full complexity of the algorithm.

Either way, I am happy with the way my implementation is working. Now I only need to implement a function that finds the particular and special solutions, in order for the program to fail only when there are no solutions (the solvability check fails)!