MSAN 502 - Homework 1

Andre Guimaraes Duarte

July 20, 2016

Problem 1

The python file corresponding to this part of the homework is `HW1_1.py`.

A

We have:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 \\ 41 & 24 & 43 & 44 & 45 & 46 & 47 & 48 & 49 & 50 \\ 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 & 59 & 60 \\ 61 & 62 & 63 & 64 & 65 & 66 & 67 & 68 & 69 & 70 \\ 71 & 72 & 73 & 74 & 75 & 76 & 77 & 78 & 79 & 80 \\ 81 & 82 & 83 & 84 & 85 & 86 & 87 & 88 & 89 & 90 \\ 91 & 92 & 93 & 94 & 95 & 96 & 97 & 98 & 99 & 100 \end{bmatrix}, v = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{bmatrix}, \text{ and } b = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

We use `linalg.solve` from the package `numpy` in order to solve $Ax = b$ and $Ax = v$. The solutions given by this function are:

$$Ax = b \Leftrightarrow x = \begin{bmatrix} -0.08789062 \\ -0.09179688 \\ 0.03125 \\ -0.02734375 \\ 0.08203125 \\ -0.0234375 \\ 0.05273438 \\ 0.00195312 \\ -0.0078125 \\ 0.0703125 \end{bmatrix}, \text{ and } Ax = v \Leftrightarrow x = \begin{bmatrix} -0.003125 \\ 0.00546875 \\ -0.05664062 \\ -0.02734375 \\ 0.16601562 \\ -0.07226562 \\ -0.01171875 \\ 0.00390625 \\ 0.03125 \\ 0.06445312 \end{bmatrix}$$

We can easily verify that these solutions, `x1` and `x2` in my python file, are indeed correct by using the following code:

```
Ax1 = np.dot(A, x1) # == b
Ax2 = np.dot(A, x2) # == v
```

Indeed, we get that $Ax1 = b$ and $Ax2 = v$. I used the `numpy` function `isclose()` instead of the simple equivalence comparison (`==`) because of floating point approximations. We do get the desired result though.

B

The above matrix A is singular, meaning that, by using the elimination method to solve the system, we would have hit a failure at some point of the algorithm (actually fairly early, after the second row operation in fact). But `linalg.solve` is s̈martënough to overcome this problem.

In order to solve this problem ourselves, one solution is to create a second 10x10 matrix $R(\epsilon)$ whose entries are indepentent and identically distributed random variables $X_{i,j} \sim U(-\epsilon, \epsilon)$ that we add to our original matrix A. We then solve the problem by using the new matrix $A + R(\epsilon)$.

In python, I created a function `randomMatrix(e)` that generates a 10x10 matrix as described above, where ϵ is set by the user. It is then easy to create different matrices according to the values of ϵ that we want.

```
def randomMatrix(e):
  # Create matrix R (10x10) filled with zeros
  R = np.zeros(100).reshape(10, 10)
  # For every line
  for i in range(len(R)):
    # For every column
    for j in range(len(R[i])):
      # Replace R[i,j] with a random variable from a uniform(-epsilon, epsilon)
      R[i][j] = random.uniform(-e, e)

  # Return the matrix
  return R

# Create matrix R (10x10) with random noise
e = 0.01
R = randomMatrix(e)
```

Using this new matrix $A + R(\epsilon)$, for various values of $\epsilon > 0$ ranging from 0.01 to 1, we can compute the two earlier systems ($Ax = b$ and $Ax = v$), and we do get the same results as before.

C

This noise idea to answer $Ax = b$ and $Ax = v$ is interesting. By adding even a little bit of random noise to our singular matrix, we make it non-singular, and therefore (almost certainly) solvable. By changing a little bit the values of A, we don't seem to change the solution x by much. In fact, since A is a fairly large matrix, we can expect the overall impact of random uniform noise (centered around 0!) in **all** values of A to almost cancel itself out in the solution. This is due to the fact that the uniform distribution we chose (from which we sample random variables) is centered around 0. The noise on a line tends to average around 0, meaning that the impact on the solution will be negligible.

Problem 2

The python files corresponding to this part of the project are `Eliminate.py` and `HW1_2.py`.

A

The python file `Eliminate.py` contains the algorithm that I implemented for elimination. I added a few checks in the beginning to make sure that the matrices are indeed matrices, that they have correct dimensions, etc ... My algorithm kicks out if it encounters a temporary or permanent failure. It prints an error message to explain what happened, and then exits the program.

Running my algorithm to solve the previous problems ($Ax = b$ and $Ax = v$) causes it to hit a permanent failure, leading to no or infinitely many solutions to the problems. This is because the matrix A, as seen previously, is singular: at one point (in the beginning of the algorithm), the third row is equal to two times the second row. After that round of row operations, the entire third row will contain only zeros. This causes the failure in the algorithm.

B

I decided to create a new file for this question, in order to keep `Eliminate.py` clean. The code for this question is in `HW1_2.py`.

It is interesting to compare the performance of my implementation of the elimination algorithm to `linalg.solve`. To do so, I created a loop that creates a matrix $R(0.5)$ and uses my algorithm and `linalg.solve` to find a solution to $(A + R(0.5))x = b$ for $n = 100$, 1000, 5000, and 10000 and measured the time taken by each algorithm, then took the mean of all 10 runs. I commented out all `print` statements as to not slow it down. The results are shown in the table 1 below.

n	Mean time taken by my algorithm (s)	Mean time taken by `linalg.solve` (s)
100	0.4493	0.0026
1000	13.3276	0.1080
5000*	435.1310	6.1805
10000**	2249.7651	47.1378

Table 1: Summary of the time taken by my implementation of the elimination algorithm and `linalg.solve` to solve $Ax = b$, for 10 runs of the algorithms (*: only 5 runs; **: only 1 run).

We can clearly see that `linalg.solve` is clearly better optimized to tackle problems of this type. While my implementation is acceptable with n up to about 1000, it seriously starts to struggle with larger data sets, whereas `linalg.solve` has no problems. It probably does not use the elimination algorithm in its full extent, but uses other methods to achieve faster results. However, it is nice to see that my implementation works, even for pretty large data sets.