CSA E0 226: Linear Algebra and Probability (24^{th} September 2018)

Assignment 1

Submitted by:- YASH KHANNA [MTech (Res.) CSA / 04-04-00-10-22-18-1-15563]

Problem I - Bonus Question

If it is not possible to prepare the potion (i.e. no possible solution) because of limitation in ingredient quantity, can you tell the maximum amount of potion that can be prepared out of the given ingredients?

Solution: Given a matrix A (Percentage of k ingredients distributed among n persons) of size $n \times k$, we are to find a vector b of size $n \times 1$ which denotes the maximum potion we can make. Also we are given the maximum amount of each ingredient available in the vector m. The maximum amount of potion we can make is when we use all the ingredients, i.e. given by Am, this is because for any m' such that $m' < m$ (where intuitively this means we use lesser quantity of the ingredients and formally meaning that all vector entries $m'_i < m_i \; \forall i \in [n]$), $\Rightarrow Am' < Am$, so we can always increase the amount of potion by using m, this works out because all the vector and matrix entries are positive reals.

Problem II - Second Part

You will have to report the time taken for the process to complete. Compare it with the standard library API available in Python and mention the difference. Can you find a way to improve the efficiency of your algorithm in terms of time taken?

Solution: We have used *NumPy* as the library to compare the performance of the algorithms. We generate random entries in a square matrix and take the average over 50 iterations for different sizes. We use both the algorithms and note down the difference. The below table shows the data. All the time units are shown in seconds,

Matrix-Size($n \times n$)	Our-Algorithm	NumPy-Algorithm	Difference
5	0.00024772	0.00011978	0.00012794
10	0.00104496	0.00012846	0.0009165
100	0.51307468	0.00214492	0.51092976
500	54.545607	0.172603	54.373004
1000	443.1451105	2.615269	440.5298415

To improve the efficiency of our algorithm, we can do several improvements,

1. If the inverse does not exist, our algorithm should fail fast.
2. To find the pivot element in each row, we can do some book keeping instead of iterating over the whole array and to make it 1 we can divide the only diagonal entry at the end just by choosing a different set of constants.
3. We can parallelize (almost) the set of operations on the identity matrix (right half) instead of doing it in the standard augmented way which takes twice the amount of time (number of columns doubles) and this really adds up for large matrices.

Problem II - Third Part
Consider you are given the same three operations, but this time instead of row operations you are allowed to do column operations. Can you find an inverse? Explain why or why not.

Solution: Let us assume we have a matrix A of size $n \times n$ and it is invertible. Then there exists a some elementary matrices $E_1, E_2, ..., E_k$ such that $E_1E_2...E_kA = I$. Taking the transpose both sides, $A^T E_k^T E_{k-1}^T ... E_1^T = I$. We will show that doing a column operation is same as post-multiplying a matrix by a transpose of an elementary matrix.

Example 1:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} d & e & f \\ a & b & c \\ g & h & i \end{bmatrix} (Swaps\ Rows\ 1\ and\ 2)$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} b & a & c \\ e & d & f \\ h & g & i \end{bmatrix} (Swaps\ Columns\ 1\ and\ 2)$$

Example 2:

$$\begin{bmatrix} 1 & \lambda & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} a+\lambda d & b+\lambda e & c+\lambda f \\ d & e & f \\ g & h & i \end{bmatrix} (Adds\ \lambda\ times\ Row\ 2\ to\ Row\ 1)$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ \lambda & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a+\lambda b & b & c \\ d+\lambda e & e & f \\ g+\lambda h & h & i \end{bmatrix} (Adds\ \lambda\ times\ Column\ 2\ to\ Column\ 1)$$

Example 3:

$$\begin{bmatrix} \lambda & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} \lambda a & \lambda b & \lambda c \\ d & e & f \\ g & h & i \end{bmatrix} (Multiplies\ \lambda\ to\ Row\ 1)$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} \lambda & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \lambda a & b & c \\ \lambda d & e & f \\ \lambda g & h & i \end{bmatrix} (Multiplies\ \lambda\ to\ Column\ 1)$$

The above three examples show we can perform the elementary row operations on columns by post multiplying with the transpose. Thus to find the inverse using column operations, just take the transpose of A and post-multiply with the matrices $E_k^T E_{k-1}^T ... E_1^T$ in this order to get I. Hence if A is invertible we can find an inverse using either row operations or column operations.

Problem II - Bonus Question
Say you have only two out of the three operations at your disposal. Given a matrix, is it possible to get identity matrix? Also, what can you say about the complexity of the algorithm?

Solution: Consider a matrix A of size $m \times n$, The three elementary row operations are:

1. $R_i \leftarrow cR_i$ for some $c \in F$ and for all i
2. $R_i \leftarrow R_i + cR_j$ for some $c \in F$ and for all i, j where $i \neq j$
3. $R_i \Longleftrightarrow R_j$ for all i, j where $i \neq j$

We show that 1 and 2 can be used to implement operation 3 (for $i \neq j$) (i.e. swap can be implemented from the other two);

1. $R_i \leftarrow R_i + R_j$ $(R_i = R_i + R_j, R_j = R_j)$
2. $R_j \leftarrow -R_j$ $(R_i = R_i + R_j, R_j = -R_j)$
3. $R_j \leftarrow R_j + R_i$ $(R_i = R_i + R_j, R_j = R_i)$
4. $R_i \leftarrow R_i - R_j$ $(R_i = R_j, R_j = R_i)$

We can prove that 2 and 3 can be used to implement operation 1 (for $i \neq j$ and $c \neq 0$) if and only if the rows are linearly dependent;

Proof (IF) : Consider a matrix A of size $m \times n$, where we have to transform (w.l.o.g) R_1 to λR_1 where $\lambda \neq 0$ and $\lambda \neq 1$ (nothing to do in this case) and the rows are linearly dependent, i.e. $R_m = \sum_{i=0}^{m-1} \alpha_i R_i$ where all $\alpha_i's$ are not zero. Then we do the following operations to first create a zero row in the m^{th} row,
1) $R_m \leftarrow R_m - \alpha_1 R_1$
2) $R_m \leftarrow R_m - \alpha_2 R_2$
$\vdots$
m-1) $R_m \leftarrow R_m - \alpha_{m-1} R_{m-1}$
It is easy to see that after these operations we get 0 in the m^{th} row. Now we do the following set of operations.
m) $R_m \leftarrow R_m + (\lambda - 1)R_1$
m+1) $R_1 \leftarrow R_1 + R_m$
m+2) $R_m \leftarrow R_m - (\lambda - 1)/\lambda R_1$
Hence we got the desired transformation. Now to recover R_m to its original form, we can use the inverse of the first $m - 1$ steps as follows with a slight modification.
m+3) $R_m \leftarrow R_m + (\alpha_1/\lambda)R_1$
m+4) $R_m \leftarrow R_m + \alpha_2 R_2$
m+5) $R_m \leftarrow R_m + \alpha_3 R_3$
$\vdots$
2m+1) $R_m \leftarrow R_m + \alpha_{m-1} R_{m-1}$

Proof (ONLY IF) : Assuming that rows of the A are linearly independent, then let's say there is some set of operations which transforms R_1 to λR_1 by using 2 and 3 only. This implies at the end of our steps, all rows are intact as is and the first row has become a linear combination with some of the rows (this is what operation 2 does). This is only possible when,
$R'_1 = \sum_{i=0}^{m} \alpha_i R_i = \lambda R_1$ (for not all zero $\alpha'_i s$)
$\sum_{i=0}^{m} \beta_i R_i = 0$ (for not all zero $\beta'_i s$)
But this implies the rows are linearly dependent. This contradicts our assumption.

This proof also shows the trivial case when $m = 1$ (linearly independent) that we cannot implement this operation.

It is easy to see that operation 2 cannot be implemented from 1 and 3. Intuitively, this is because we don't even have addition as a field operation to perform in this case.

COMPLEXITY: We will show an upper bound on the number of steps taken using addition and multiplication of two field entries as an elementary operation. Consider a matrix A of size $n \times n$ to be converted to I_n. Let B be $[A|I_n]$ the augmented matrix of size $n \times 2n$. Each iteration of the algorithm, takes a row and finds the pivot column ($\mathcal{O}(n)$) and makes it 1 by dividing the row by a suitable constant ($\mathcal{O}(n)$) and then we do $n-1$ row operations to make that a pivot column, each such operation takes $\mathcal{O}(2n)$ time, thus this adds to $\mathcal{O}(2n * (n-1))$ for each row. Summing up, this amounts to $\mathcal{O}(2n(n-1)(2n))$ $= \mathcal{O}(n^3)$. Also note that we perform atmost n swaps in the whole algorithm and each swap takes $\mathcal{O}(n)$ time. Thus this contributes as $\mathcal{O}(n^2)$ time. Thus the final complexity is $\mathcal{O}(n^3) + \mathcal{O}(n^2) = \mathcal{O}(n^3)$ time.
