CS 370 - A4

Bilal Khan
bilal2vec@gmail.com

November 28, 2023

Contents

1 1

1.1 a

By hand, use Gaussian elimination with row pivoting to find the $PA = LU$ factorization for the following matrix

$$A = \begin{bmatrix} 2 & 3 & 0 \\ -2 & -7 & 8 \\ 4 & -4 & 23 \end{bmatrix}$$

Swap first and last row

$$A = \begin{bmatrix} 4 & -4 & 23 \\ -2 & -7 & 8 \\ 2 & 3 & 0 \end{bmatrix} P = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Performing $R_2 + 1/2R_1$ and $R_3 - 1/2R_1$

$$A = \begin{bmatrix} 4 & -4 & 23 \\ 0 & -9 & 19.5 \\ 0 & 5 & -11.5 \end{bmatrix}$$

Performing $R_3 + 5/9R_2$

$$A = \begin{bmatrix} 4 & 0 & 23 \\ 0 & -9 & 19.5 \\ 0 & 0 & -2/3 \end{bmatrix}$$

This gives us the upper triangular matrix U, and the elements in the unit lower triangular matrix L are given by the operations we performed on the rows of A: $R_2 + 1/2R_1$ and $R_3 - 1/2R_1$ and $R_3 + 5/9R_2$, and we invert the sign of the coefficients to get the elements of L:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/2 & -5/9 & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} 4 & -4 & 23 \\ 0 & -9 & 19.5 \\ 0 & 0 & -2/3 \end{bmatrix}$$

$$P = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

1.2 b

By hand, use the factorization you found above to solve for x in the linear system $Ax = b$ where

$$b = \begin{bmatrix} -3 \\ 15 \\ 30 \end{bmatrix}$$

Since we have a permutation matrix, we will solve the system $PAx = Pb = b'$ instead. We have

$$b' = \begin{bmatrix} 30 \\ 15 \\ -3 \end{bmatrix}$$

We now forward solve $Lz = b'$ to get z:

$$\begin{aligned} z_0 &= b'_0 = 30 \\ z_1 &= b'_1 - L_{1,0}z_0 = 15 - (-1/2)30 = 30 \\ z_2 &= b'_2 - L_{2,0}z_0 - L_{2,1}z_1 = -3 - (1/2)30 - (-5/9)30 = -4/3 \end{aligned}$$

We now backward solve $Ux = z$ to get x:

$$x_0 = (z_0 - U_{0,1}x_1 - U_{0,2}x_2)/U_{0,0} = (30 - (-4)(1) - (23)(2))/4 = -3$$
$$x_1 = (z_1 - U_{1,2}x_2)/U_{1,1} = (30 - (19.5) * 2)/(-9) = 1$$
$$x_2 = z_2/U_{2,2} = (-4/3)/(-2/3) = 2$$

2 2

Suppose a square $N \times N$ nonsingular matrix A has already been factored as $A = LU$, where L is unit-diagonal and lower-trianguale and U is upper triangular. Consider the system $A^2x = b$ where b is a given vector and x is an unknown vector.

2.1 a

Show how to use the L and U factors of A to solve the linear system efficiently.

Since $A = LU$, we have that $A^2 = LULU = LU^2$. We can solve $LU^2x = b$ efficiently. First we compute U^2. Next, we forward solve $Lz = b$ to get z. Finally we backward solve $U^2x = z$ using our precomputed U^2 to get x. This is more efficient because it takes less flops to compute matmul two upper triangular matrices (U^2) than it does to compute matmul two general matrices A^2. The rest of the operations are the same. Since U is upper triangular, it has only zeros below the diagonal, $L_{i,j} = 0$ if $i > j$, and performing any multiplications against these zero elements doesnt change the output. So when we compute U^2, we can skip the multiplies where one of the operands are zero and just add the non-zero operand to the output. This keeps the number of adds in the matmul the same, but reduces the number of multiplies by half.

2.2 b

The initial LU-factorization of A takes $2N^3/3 + O(n^2)$ flops. The forward and backward solves take a total of $2n^2 + O(n)$ time. Normally, computing a matmul U^2 for a $N \times N$ upper-triangular matrix takes $N^2(2N - 1)$ flops. This is because the output of the matmul is a $N \times N$ matrix, and to compute each element of the output, we element-wise multiply a N-element vector with another N-element vector, then add up the results. This takes N flops for the multiplies, and $N - 1$ flops for the $N - 1$ additions between pairwise elements the multiplied vector $N + N - 1 = 2N - 1$, for a total of $N^2(2N - 1)$ flops.

However, since U is upper triangular, we can skip multiplies where one of the operands are zero and just add the non-zero element to the resulting sum. The number and location of the non-zero elements for each output element in U^2 differs (we're element-wise multiplying a vector that starts with some number of zeros by a vector that ends with some other number of zeros, both numbers depending on the i, j location of the resulting element in the output), but on average there will be $N/2$ multiplies required for each output element, and the same $N - 1$ additions. So the total number of flops in an efficient matmul is $N^2(N/2 + N - 1) = N^2(3N/2 - 1)$.

Note that this is an *exact* calculation of the number of flops required, even though not every output element will require $N/2$ multiplies, but as a short proof to show this is correct you can see that, due to the symmetric upper-triangular structure of the matrix U, for every output element in the matrix that requires $x < N/2$ element-wise multiplies due to there being x non-zero elements in one of the vector operands, there is another unique output element that requires $N/2 - x$ element-wise multiplies due to there being $N/2 - x$ non-zero elements in one of its vector operands, and we

can pair up these elements in a one-to-one correspondence to show that there are $N^2/2$ pairs of output elements that each require a total of $N/2$ multiplies each.

This comes out to $2N^3/3 + 2N^2 + N^2(3N/2 - 1) + O(n^2)$ flops.

2.3 c

Using the naive matmul implementation described in part b, we would first compute A^2 then compute its LU-factors and solve the system. This would take $2N^3/3 + O(n^2)$ flops for the initial LU-factorization, $N^2(2N - 1)$ flops for the matmul, and $2n^2 + O(n)$ flops for the forward and backward solves, for a total of $2N^3/3 + N^2(2N - 1) + 2n^2 + O(n^2)$ flops, which is greater than the efficient method.

3 3

Let A be an $N \times N$ matrix and you have already computed the L, U factors of A. i.e. $A = LU$ where L is unit-diagonal and lower-triangular and U is upper triangular. Later on, some elements of A were found to be wrong. The new matrix can be written as $A + pt^T$ where p and q are $N \times 1$ vectors It is known that

$$(A + pq^T)^{-1} = A^{-1} - A^{-1}p(1 + q^T A^{-1}p)^{-1}q^T A^{-1}$$

3.1 a

Consider the linear system $(A + pq^T)x = b$ Using the above formular write down the formula for computing the solution x.

$$\begin{aligned}
(A + pq^T)x &= b \\
(A + pq^T)^{-1}(A + pq^T)x &= (A + pq^T)^{-1}b \\
x &= (A + pq^T)^{-1}b \\
&= (A^{-1} - A^{-1}p(1 + q^T A^{-1}p)^{-1}q^T A^{-1})b
\end{aligned}$$

3.2 b

Describe an efficient algorithm to compute x based on the formula in part a. You should use the L and U factors of A. If you compute the L and U factors of $A+pq^T$, you will recieive 0 marks for this aprt. Also, do not compute A^{-1} explicitly. You will also receive 0 marks if you do. Explain where and how many (if any) vector dot products, matrix-vector multiplies, forward and backward solves are needed. Perform a complexity analysis. The flop counts for your efficient algroithm should be given by $4N^2 + O(N)$, in which you should ignore the flops for the LU factorization of A.

$$\begin{aligned}
x &= (A^{-1} - A^{-1}p(1 + q^T A^{-1}p)^{-1}q^T A^{-1})b \\
x &= A^{-1}b - A^{-1}p(1 + q^T A^{-1}p)^{-1}q^T A^{-1}b
\end{aligned}$$

We can decompose this into smaller parts to solve individually:

Let $z = A^{-1}b$, or equivalently, $Az = b$, we can solve this using the LU-factorization of A: $Ly = b$ and $Uz = y$.

Let $w = A^{-1}p$, or equivalently, $Aw = p$, we can solve this using the LU-factorization of A: $Lv = p$ and $Uw = v$.

$$\begin{aligned} z &= A^{-1}b \\ w &= A^{-1}p \\ x &= A^{-1}b - A^{-1}p(1 + q^T A^{-1}p)^{-1}q^T A^{-1}b \\ &= z - w(1 + q^T w)^{-1}q^T z \end{aligned}$$

The product $q^T w$ results in a scalar, and the product $q^T z$ results in a scalar, so the scalar inverse followed by scalar product $(1 + q^T w)^{-1}q^T z$ results in a scalar, and its product with w and substraction from z results in a vector.

The LU factorization of A is given so the forwards+backwards solves to get z and w each take $2N^2 + O(n)$ flops, the dot products $q^T w$ and $q^T z$ each take $N + N - 1 = 2N - 1$ flops, the scalar addition $1 + q^T w$ takes 1 flop, its scalar inverse takes another 1 flop, and the scalar product between it and $q^T z$ takes one flop. The product between the resulting scalar and w takes N flops, and the final substraction from z takes N flops.

The total number of flops is then:

$$\begin{aligned} 2 \times (2N^2 + O(N)) + 2 \times (2N - 1) + 1 + 1 + 1 + N + N \\ 4N^2 + 4N - 2 + 1 + 1 + 1 + 2N + O(N) \\ 4N^2 + 6N + 1 + O(N) \\ 4N^2 + O(N) \end{aligned}$$

4 4

4.1 a

Write a python function $PageRank(G, alpha)$ which determines the pagerank for a network using the basic PageRank algorithm described in class. The inputs are the adjacency matrix G given as a standard 2D numpy array, representing the directed graph of the network, and the scalar weight alpha. (As usual the adjacency matrix is defined such as $G(i, j) = 1$ if there is a link from page j pointing to page i otherwise $G(i, j) = 0$. You do not need to remove any self-links that may be present in the input graph G.) The function should output a tuple (p, it) consisting of the vector p of pagerank scores, and integer it representing the number of steps of pagerank required for the computation to finish. Use a tolerance value of 10^{-8}. For this part, you can implement the basic PageRank method in a naive/straightforward fashion; for example, explicitly forming the (dense) Google matrix is acceptable for this part.

4.2 b

The following figure shows a small directed graph representing a set of 14 web pages. Write Jupyter/Python code that constructs the adjacency matrix G and computes the pagerank vector for

the web shown above using your PageRank function with $\alpha = 0.9$. Print the vector of final pagerank scores (in ascending order of the node numbers). List the indices of the pages in descending order of importance (from most to least), according to your results. Use the matplotlib spy command to graph the sparsity (nonzero) pattern of the adjacency matrix G. Use the matplotlib bar command to plot the pagerank scores as a bar graph. Title your graphs appropriately.

4.3 c

In order to efficiently handle larger network graphs, write a new function PageRankSparse(Gcsr, alpha). This new function should take in an adjacency matrix Gcsr which is our usual adjacency matrix but stored in an underlying sparse matrix representation known as compressed sparse row, or CSR. You may wish to refer to

https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html. (Fortunately though, SciPy largely hides the underlying internal data structure details, so we essentially do not need to be con- cerned with them.) Your new function should also accept the same parameter alpha as before, and likewise return a tuple [p, it] consisting of the pagerank score vector p and iteration count it. This function must efficiently compute the update for the pagerank vector by taking advan- tage of the sparsity of Gcsr and the inherent structure of the Google matrix, as discussed in class. Avoid creating any full (i.e., dense) matrices and do not explicitly form the Google matrix.

The main PageRank iteration should consist of only a single outer loop structure (i.e., do not implement matrix or vector multiplication manually). Make use of SciPy's built in sparse matrix multiplication functionality. A few tips: Sparse matrix multiplication can be performed in Python using the dot function. That is, if A is a CSR matrix and b is a (standard, dense) numpy vector, you can compute their matrix-vector product as A.dot(b). dot can also be used for computing dot products between two vectors, a.dot(b) or sparse matrix-matrix products A.dot(B). The sum function may be useful for computing column sums of G, i.e., outdegrees of nodes. You can convert a dense matrix G to a CSR matrix using Gcsr = sparse.csr matrix(G). This may be useful for debugging your sparse method, by comparing against the results of your earlier dense implementation. Refer to online Numpy/SciPy documentation for further details of their sparse matrix support.

4.4 d

Write Python code to apply your efficient pagerank implementation to the in- ternet graph data in the file bbc.mat provided with the assignment, again using $\alpha = 0.9$. This data represents a network of 500 webpages, originally generated starting from www.bbc.co.uk. The data consists of a sparse matrix G and a list of urls U. These two pieces of data can be loaded as follows:

```
import scipy.io
 data = scipy.io.loadmat('bbc.mat')
 Gcsr = data['G']
 Gcsr = Gcsr.transpose() #data uses the reverse adjacency matrix convention.
 U = data['U']}
```

Once again, use the spy command to graph the sparsity (nonzero) pattern of this adjacency matrix, titling the graph appropriately. Using the results of running your efficient page rank routine on this data, your code should print the top 20 urls in descending order of importance. You may find the function argsort useful to determine the indexing that would sort the pagerank vector.