# CS 370 Winter 2025: Assignment 4

Instructors: Leili Rafiee Sevyeri (email: leili.rafiee.sevyeri@uwaterloo.ca)
Eugene Zima (email: ezima@uwaterloo.ca)

**Due Wednesday, April 2nd, at 5:00pm EST**

Submit **all** components of your solutions (written/analytical work, code/scripts, figures, plots, output, etc.) to Crowdmark in PDF form for each question (multiple pages are allowed). Note that you may resubmit as often as necessary until the due date — only the final submission will be marked.

You must *also* separately submit **a single zip file** containing any and all code/scripts you write to the Assignment 4 DropBox on LEARN, in runnable format (and if necessary, any data needed to reproduce your results).

You have a number of options of how to prepare your solutions. You can typeset your solutions in a word-processing application (MS Word, LaTeX, etc.), you can write on a tablet computer, or you can write on paper and take photos. **It is your responsibility to ensure that your submission is sufficiently legible (and not corrupted upon upload). This is particularly important if you are submitting photos of handwritten pages.** TAs have the right to take marks off for illegible answers.

Note that your solutions will be judged not only for correctness but also for the quality of your presentation and explanations.

1. **(10 marks)**

   (a) (6 marks)
   By hand, use Gaussian elimination with row pivoting (i.e., partial pivoting) to find the $PA = LU$ factorization for the following matrix

$$A = \begin{bmatrix} 1 & 7 & 3 \\ 1/2 & 9/4 & -1/4 \\ 1/3 & 22/3 & 0 \end{bmatrix}$$

   where $L$ is unit lower triangular, $U$ is upper triangular, and $P$ is a permutation matrix. (Pivoting should be applied whenever the diagonal entry is not the largest magnitude value on or below the diagonal, i.e., not just when the diagonal is zero or near zero.)

   (b) (4 marks)
   By hand, use the factorization you found above to solve for $x$ in the linear system $Ax = b$ where

$$b = \begin{bmatrix} 0 \\ 4 \\ 8 \end{bmatrix}.$$

2. **(6 marks)** Suppose a square $n \times n$ nonsingular matrix A has *already* been factored as

$$PA = LU \tag{1}$$

where $P$ is a permutation matrix, $L$ is unit lower triangular, and $U$ is upper triangular. That is, assume you are given the matrices that make up this factorization. Suppose you wish to solve the linear system

$$A^T x = v \tag{2}$$

where $v$ is a given vector and $x$ is an unknown vector to be found.

(a) (3 marks) Describe an efficient algorithm for solving this linear system for $x$, using the factorization provided above.

(b) (3 marks) Determine the FLOP count of your algorithm. Do not include the FLOPs for constructing the original factorization. (Note that we consider both permutation [i.e., applying $P$] and matrix transposition to require 0 flops, since each can be achieved via appropriate indexing).

3. **(12 marks)** Let $A$ be an $n$-by-$n$ nonsingular matrix, and let $b$ and $c$ be column vectors of length $n$.

Suppose you are asked to compute the matrix $W = bc^T A^{-1}$.

(a) (6 marks) One possible algorithm is the following:

(1) Compute $V = bc^T$.
Since $W = VA^{-1}$ we have $WA = V$ and therefore $A^T W^T = V^T$. Thus $W$ can be computed as follows:

(2) Compute a factorization $A^T = LU$ (for simplicity, we assume no pivoting is needed).

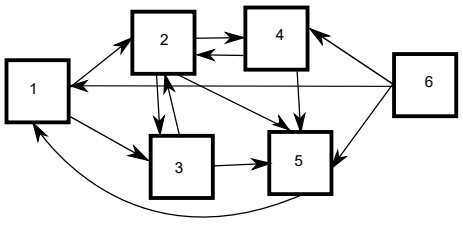
(3) For $i = 1, \ldots, n$ compute the $i^{\text{th}}$ row of $W$ by triangular solves $Lz = V_i^T$ and $UW_i^T = z$ where $V_i$ denotes the $i^{th}$ row of $V$.

Show that this algorithm requires $\frac{8}{3}n^3 + O(n^2)$ flops.

(b) (6 marks) Suggest a more efficient algorithm for computing $W = bc^T A^{-1}$ which requires only $\frac{2}{3}n^3 + O(n^2)$ flops and show that this is the case. (Hint: Consider reordering how the computation is done.)

4. **(8 marks)**

Construct, by hand, the Google matrix needed to perform the PageRank algorithm for the following small web graph, using $\alpha = 1/2$. Then, compute the probability vector for the web surfer (i.e., probability of being on each page) after taking **a single step** of "surfing" according to this matrix, assuming that, rather than uniform initial probability, the surfer begins initially on page 1, 2, 4, or 6, with a 1/4 probability for each and with zero probability of being on pages 3 or 5.




5. (**20 marks**) Implementing Google PageRank

Prepare the solution to the following tasks in a Jupyter Notebook A4Q5.ipynb.

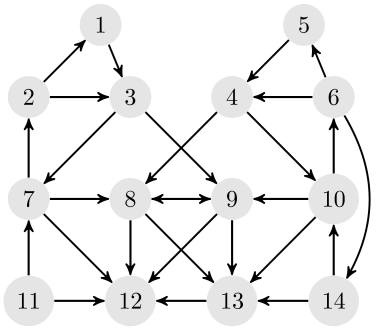
(a) (5 marks) Write a Python function

`PageRankDense(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 that $G(i, j) = 1$ 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 should implement the *basic* PageRank method in a direct/straightforward fashion; that is, by explicitly forming the dense Google matrix $M$.

(b) (5 marks)

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.

(c) (6 marks)

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 concerned 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 advantage 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.

(d) (4 marks)

Write Python code to apply your efficient pagerank implementation to the internet 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 url's 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.