Overview

The pedagogical purpose of this project is to

1. give you practice implementing a numerical linear algebra algorithm, so that you better understand the algorithm through practice as well as theory, and
2. demonstrate how numerical linear algebra and multivariable calculus are useful in solving computational problems.

This project consists of three parts. The first part is worth 55 points, the second part is worth 25 points, and the third part is worth 20 points. Extra points will be considered for quality of discussion for up to 15 points.

You may work in teams of **up to three** persons in the same section (C1 with C1 only, G2 with G2 only, etc.). Code for your project should be submitted in either Java, Python, or MATLAB (or its free version FreeMat). (Choose only one of these languages to use for the whole project—do not mix and match.) Your final deliverable should be submitted in a single `.zip` archive file. The archive file should be uploaded to the Dropbox on T-Square of *one* of your team members by **11:59 p.m.** on **November 24**. The file should contain:

- A `team_members.txt` file listing the name of each person in the project team.
- All source files for each part of the project you submit.
- A `.doc`, `.docx`, or `.pdf` file for each part of the project you submit, containing the written component of the project.
- A `readme` file for each part of the project you submit, explaining how to execute that part of the project.
- For Java or Python users, do not submit a project that has to be run on an IDE. The graders should be able to at least initialize your project on a command line interface, either Linux or Windows.

While you may use built-in or external libraries of classes and objects for matrix and linear algebra (e.g. `numpy`), you may only use built-in/library functions and methods that do the following:

- Create/instantiate/initialize vectors and matrices
- Add and subtract vectors and matrices
- Multiply a scalar with a vector or matrix
- Take the dot product of two vectors
- Transpose a matrix or vector

You must program any other linear algebra functionality you wish to use yourself, including (but not limited to) procedures which

- Find the multiplication of two matrices
- Find the LU-factorization of a matrix
- Find the QR-factorization of a matrix
- Find the solution of triangular systems with backward (forward) substitution
- Find the determinant of a matrix
- Find the trace of a matrix
- Find the eigenvalues and eigenvectors of a matrix
- Rotate, reflect, or project a vector

Using built-in or external library functions or using code copied from an external source (e.g. the Internet) for these operations will result in significant penalties on the relevant portion of the project. Moreover, submitting copied code *without proper credit due* will be considered plagiarism (see below).

Academic Honor Code Warning

You may not share code outside of your team. Code that appears improperly shared will be submitted to the Office of Student Integrity for investigation and possible sanctions. Additionally, code that appears copied from an external source and does not have a proper citation will be considered plagiarism and will also be referred to the Office of Student Integrity. **Disciplinary sanctions could include a grade of 0 on the project, an additional loss of a letter grade in the class, official notation of academic dishonesty on your transcript, and possible suspension or expulsion from the Institute.**

1 The Symmetric Pascal Matrix

The symmetric Pascal matrix P is a square matrix with entries being the 1s, $P_{11} = P_{1n} = P_{m1} = 1$

$$P_{mn} = \begin{pmatrix} (m-1)+(n-1) \\ (m-1) \end{pmatrix} = \frac{((m-1)+(n-1))!}{(m-1)!(n-1)!}, \text{ where } 0! = 1 and n! = 1 \cdot 2 \cdots n.$$

(http://en.wikipedia.org/wiki/Pascal_matrix.)
For instance, the 5×5 Pascal matrix is

$$P = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 6 & 10 & 15 \\ 1 & 4 & 10 & 20 & 35 \\ 1 & 5 & 15 & 35 & 70 \end{pmatrix}.$$

The objective of this part of the project is to solve the linear system $P\overline{\mathbf{x}} = \overline{\mathbf{b}}$, for different dimensions n, $n = 2, \ldots, 12$, with different solution methods. This exercise will lead you to a comparison of some of the numerical methods covered in class.

To calculate the error, use the maximum norm:

$$||\overline{\mathbf{x}}||_\infty = \max_{1 \leqslant i \leqslant n} |x_i|, \qquad ||A||_\infty = \max_{1 \leqslant i,j \leqslant n} |A_{ij}|.$$

Your job is to

(a) Implement the LU decomposition method for a $n \times n$ matrix A. Name the procedure `lu_fact`. Your program should return the matrices L and U, and the error $\|LU - A\|_\infty$.

(b) Implement two versions of a procedure to compute a QR-factorization of a $n \times n$ matrix A. The first version should use Householder reflections. The second version should use Givens rotations. Name the procedures `qr_fact_househ` and `qr_fact_givens`. In each version, your program should return the matrices Q, R, and the error $\|QR - A\|_\infty$.

(c) Implement the procedures to obtain the solution to a system $A\overline{\mathbf{x}} = \overline{\mathbf{b}}$, for a $n \times n$ matrix A and a $n \times 1$ vector $\overline{\mathbf{b}}$, using the LU and $QR-$factorizations. Name your programs `solve_lu_b`, `solve_qr_b`. IMPORTANT: the use of an inverse matrix function should be avoided, your program should use backward or forward substitution; using an inverse matrix defects the purpose of these methods (Why?).

(d) For each $n = 2, 3, ..., 12$, solve the system $P\overline{\mathbf{x}} = \overline{\mathbf{b}}$, where $\overline{\mathbf{b}} = (1, 1/2, ..., 1/n)^t$. Using the LU and $QR-$factorizations and solution methods described in (a), (b) and (c). For each n, your program should output the solution $\overline{\mathbf{x}}_{sol}$, and the error $\|LU - P\|_\infty$ or $\|QR - P\|_\infty$, and $\|P\overline{\mathbf{x}}_{sol} - \overline{\mathbf{b}}\|_\infty$. The output should be easily readable on the screen or a text file.

(e) Summarize your findings by plotting the errors obtained as a function of n, for each of the methods. The plot can be done using your own code, Excel, or any graphing program. The plots should be included in the written component of this part of the project.

(f) Answer the following questions in the associated written component for this part of the project:

(i) Why is it justified to use the LU or $QR-$factorizations as opposed of calculating an inverse martix?

(ii) What is the benefit of using LU or $QR-$factorizations in this way? (Your answer should consider the benefit in terms of conditioning error.)

Added note: A test case for your code. For $n = 4$

$$P = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 3 & 6 & 10 \\ 1 & 4 & 10 & 20 \end{pmatrix}, \overline{\mathbf{b}} = \begin{pmatrix} 1 \\ 1/2 \\ 1/3 \\ 1/4 \end{pmatrix}$$

LU:

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 2 & 1 & 0 \\ 1 & 3 & 3 & 1 \end{pmatrix}, U = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

$\overline{\mathbf{x}}_{sol} = (2.08333, -1.91667, 1.08333, -0.25)^t$.

QR by Householder method:

$$Q = \begin{pmatrix} -0.5 & 0.67082 & -0.5 & -0.223607 \\ -0.5 & 0.223607 & 0.5 & 0.67082 \\ -0.5 & -0.223607 & 0.5 & -0.67082 \\ -0.5 & -0.67082 & -0.5 & 0.223607 \end{pmatrix}, R = \begin{pmatrix} -2 & -5 & -10 & -17.5 \\ 0 & -2.23607 & -6.7082 & -14.0872 \\ 0 & 0 & -1 & -3.5 \\ 0 & 0 & 0 & 0.223607 \end{pmatrix}.$$

$\overline{\mathbf{x}}_{sol} = (2.08333, -1.91667, 1.08333, -0.25)^t$.

QR by Givens method:

$$Q = \begin{pmatrix} 0.5 & -0.67082 & 0.5 & -0.223607 \\ 0.5 & -0.223607 & -0.5 & 0.67082 \\ 0.5 & 0.223607 & -0.5 & -0.67082 \\ 0.5 & 0.67082 & 0.5 & 0.223607 \end{pmatrix}, R = \begin{pmatrix} 2 & 5 & 10 & 17.5 \\ 0 & 2.23607 & 6.7082 & 14.0872 \\ 0 & 0 & 1 & 3.5 \\ 0 & 0 & 0 & 0.223607 \end{pmatrix}.$$

$\overline{\mathbf{x}}_{sol} = (2.08333, -1.91667, 1.08333, -0.25)^t$.

2 Convergence of the iterative methods

In this part, you will use both Jacobi and Gauss-Seidel method to obtain the approximate solution to a system $A\overline{\mathbf{x}} = \overline{\mathbf{b}}$, for 3×3 symmetric matrix $A = \begin{pmatrix} 1 & 1/2 & 1/3 \\ 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \end{pmatrix}$ and 3×1 vector $\overline{\mathbf{b}} = \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \end{pmatrix}$.

In GaussSeidel method, you may use this formula for the inverse of lower triangle $S = \begin{pmatrix} a & 0 & 0 \\ b & c & 0 \\ d & e & f \end{pmatrix}$, the inverse S^{-1} is $\begin{pmatrix} 1/a & 0 & 0 \\ -b/ac & 1/c & 0 \\ (-cd+be)/acf & -e/cf & 1/f \end{pmatrix}$, for $a, c, f \neq 0$.

Your job is to

(a) Implement a procedure named `jacobi_iter` that uses the Jacobi iterative method to approximate the solution to the system $A\overline{\mathbf{x}} = \overline{\mathbf{b}}$. Also, name `gs_iter` for Gauss-Seidel iterative method. The inputs to the procedure should be

- a 3×1 vector $\overline{\mathbf{x}}_0$ with floating-point real numbers as entries,
- a tolerance parameter ε (a positive floating-point real number) that determines when the approximation is close enough, and

- a positive integer M giving the maximum number of times to iterate the method before quitting.

The outputs should be the approximate solution $\overline{\mathbf{x}}_N$ obtained by the iterative method as well as the number of iterations N needed to obtain that approximation. If the procedure iterates M and has not attained an answer with sufficient accuracy, the output should instead be a value representing a failure (for example, `None` in Python).

(b) Randomly generate at least 100 3×1 initial vectors $\overline{\mathbf{x}}_0$. The entries of the vectors should be floating-point real numbers uniformly distributed in the interval ~~$[-10, 10]$~~ $[-1, 1]$. For each $\overline{\mathbf{x}}_0$

- Record $\overline{\mathbf{x}}_0$.
- Use both `jacobi_iter` and `gs_iter` to find the approximation solution of $A\overline{\mathbf{x}} = \overline{\mathbf{b}}$ within a tolerance of 5 decimal places ($\|\overline{\mathbf{x}}_n - \overline{\mathbf{x}}_{n-1}\|_\infty \leq \varepsilon = 0.00005$). Use a maximum of $M = 100$ iterations before quitting in failure.
- Record $\overline{\mathbf{x}}_N$ and number of iterations N in both methods.

(c) Average these 100 $\overline{\mathbf{x}}_N$ from (b) to get an approximation solution $\overline{\mathbf{x}}_{approx}$. Compute the error of this approximation to the exact solution $\overline{\mathbf{x}}_{exact} = \begin{pmatrix} \text{~~0.3~~ } 9/190 \\ \text{~~-2.4~~ } 28/475 \\ \text{~~3~~ } 33/475 \end{pmatrix}$, that is $\|\overline{\mathbf{x}}_{approx} - \overline{\mathbf{x}}_{exact}\|_\infty$, for both iterative methods. Average the ratio of number of iteration steps N between Jacobi and Gauss-Seidel method, that is the average of the ratio $N_{Jacobi}/N_{Gauss-Seidel}$.

(d) Using your results from (b), plot a colored scatterplot. The x-axis is the initial error $\|\overline{\mathbf{x}}_0 - \overline{\mathbf{x}}_{exact}\|_\infty$, and y-axis is the iterations N associated with this initial data $\overline{\mathbf{x}}_0$. Use black scatters for Jacobi results, and blue for Gauss-Seidel results.

(e) In the written component for this part, interpret your results in part (c) and graphs from part (d). Especially explain why the graphs look the way they do.

---

3 Convergence of the Power Method

In this part you'll investigate the convergence of the power method for calculating eigenvalues and eigenvectors of randomly generated 2×2 matrices.

Your job is to

(a) Implement a procedure named `power_method` that uses the power method to approximate an eigenvalue and associated eigenvector of an $n \times n$ matrix. The inputs to the procedure should be

- an $n \times n$ matrix A with floating-point real numbers as entries (the algorithm should work for $n \geq 2$),
- a vector $\mathbf{v}$ of n floating-point real numbers that serves as the initial guess for an eigenvector of A,
- a tolerance parameter ε (a positive floating-point real number) that determines when the approximation is close enough, and
- a positive integer N giving the maximum number of times to iterate the power method before quitting.

The outputs should be the approximate eigenvalue and eigenvector obtained by the power method as well as the number of iterations needed to obtain that approximation. If the procedure iterates N times and has not attained an answer with sufficient accuracy, the output should instead be a value representing a failure (for example, `None` in Python).

(b) Write a program that generates at least 1000 2×2 matrices. The entries of the matrices should be floating-point real numbers uniformly distributed in the interval $[-2, 2]$. For each matrix A,

- Compute A^{-1} by $\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{ad-bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$. (In the unlikely event A^{-1} doesn't exist, throw out A, generate a new random 2×2 matrix, and restart with this newly generated matrix.)
- Use `power_method` to find the largest eigenvalue of A in absolute value within an accuracy of 5 decimal places ($\varepsilon = 0.00005$). Use a maximum of $N = 100$ iterations before quitting in failure.
- Use `power_method` on A^{-1} to find the smallest eigenvalue of A in absolute value within an accuracy of 5 decimal places. Use a maximum of $N = 100$ iterations before quitting in failure.
- Record the trace and determinant of A, and also record the number of iterations needed for the runs of `power_method` on A and on A^{-1}.

(c) Using your results from (b), plot two color-coded scatterplots. The x-axis of the plot should be the determinant, the y-axis should be the trace. Plot the determinant and trace of each matrix from (b) as a point, and color the point based on the number of iterations needed in `power_method` for that matrix. The second scatterplot should do the same, except coloring the point based on the number of iterations needed in `power_method` for the inverse of each matrix.

(d) In the written component for this part, interpret your graphs from part (c). Especially explain why the graphs look the way they do.