CECS 229 Programming Assignment #7

Due Date:

Sunday, 12/10 @ 11:59 PM

Instructions:

1. In `helpers.py`, copy-paste implementation for the following functions from your `pa6.py`:
   - `gram_schmidt()`
   - `_ref()`
   - `rank()`
2. In `structures.py`, copy-paste the implementation of the missing `Matrix` methods from `pa5.py`.

3. Complete the programming problems in the file named `pa7.py`. You may test your implementation on your Repl.it workspace by running `main.py`.

4. When you are satisfied with your implementation,
   - Submit your Repl.it workspace
   - Download the file `pa7.py` and submit it to the appropriate CodePost auto-grader folder.

Objectives:

1. Apply the QR-factorization of a matrix A to solve the system of equations $A\overrightarrow{x} = \overrightarrow{b}$.
2. Create a function that computes the determinant of an $n \times n$ `Matrix` object.
3. Use the Python built-in function `numpy.linalg.eig()` to find the eigenvalues and eigenvectors of a matrix.
4. Create a function that computes the singular value decomposition of a `Matrix` object.

Notes:

Unless otherwise stated in the FIXME comment, you may not change the outline of the algorithm provided and you may not use any built-in functions that perform the entire algorithm or replaces a part of the algorithm, unless otherwise stated.

Problem 1:

Background:

In this problem, you will implement a solver for the system of linear equations $A\overrightarrow{x} = \overrightarrow{b}$ where

- A is an $n \times n$ matrix whose columns are linearly independent

- $\vec{x} \in \mathbb{R}^n$
- $\vec{b} \in \mathbb{R}^n$

To implement the solver, you must apply the following theorem:

THM | QR-Factorization

If $A \in \mathbb{F}_{m \times n}$ matrix with linearly independent columns $\vec{a_1}, \vec{a_2}, \ldots \vec{a_n}$, then there exists,

1. an $m \times n$ matrix Q whose columns $\vec{u_1}, \vec{u_2}, \ldots, \vec{u_n}$ are orthonormal, and
2. an $n \times n$ matrix R that is upper triangular and whose entries are defined by,

$$r_{ij} = \begin{cases} \langle \vec{u_i}, \vec{a_j} \rangle & \text{for } i \leq j \\ 0 & \text{for } i > j \end{cases}$$

such that $A = QR$. This referred to as the QR factorization (or decomposition) of matrix A.

To find matrices Q and R from the QR Factorization Theorem, we apply Gram-Schimdt process to the columns of A. Then,

- the columns of Q will be the orthonormal vectors $\vec{u_1}, \vec{u_2}, \ldots, \vec{u_n}$ returned by the Gram Schimdt process, and
- the entries r_{ij} of R will be computed using each column $\vec{u_i}$ as defined in the theorem.

Your Task:

Assuming $A \in \mathbb{R}_{n \times n}$ is a `Matrix` object, and $\vec{b} \in \mathbb{R}^n$ is a `Vec` object, finish the implementation of the function `qr_solve(A, b)` which uses the QR-factorization of A to compute and return the solution to the system $A\vec{x} = \vec{b}$.

- INPUT:
  - `A` : `Matrix` object
  - `b` : `Vec` object
- OUTPUT:
  - `Vec` object representing the solution to the system $A\vec{x} = \vec{b}$.

HINT:

If $A = QR$, then $A\vec{x} = \vec{b}$ becomes $QR\vec{x} = \vec{b}$. What happens if we multiply both sides of the equation by the transpose of Q? i.e., What does $Q^t Q R \vec{x} = Q^t \vec{b}$ simplify to?

```
In [ ]:  def qr_solve(A : Matrix, b: Vec):
             """
             Solves the system of equations Ax = b by using the
             QR factorization of Matrix A
             :param A: Matrix of coefficients of the system
             :param b: Vec of constants
             :return:  Vec solution to the system
```

```python
    """
    # Constructing U
    # U should be the set of orthonormal vectors returned
    # by applying Gram-Schmidt Process to the columns of A
    U = None # FIXME: Replace with the appropriate line
    n = len(U)

    # Constructing Q
    # Q should be the matrix whose columns are the elements
    # of the vector in set U
    Q = Matrix([[None for j in range(n)] for i in range(n)])
    for j in range(n):
        pass  # FIXME: Replace with the appropriate line


    # Constructing R
    R = Matrix([[0 for j in range(n)] for i in range(n)])
    for j in range(n):
        for i in range(n):
            if i <= j:
                pass # FIXME: Replace with the appropriate line

    # Constructing the solution vector x
    b_star = Q.transpose() * b
    x = [None for i in range(n)]
    # FIXME: find the components of the solution vector
    #        and replace them into elements of x
    return Vec(x)
```

Problem 2:

Implement the helper function `_submatrix(A, i, j)` which creates and returns the sub-matrix that results from omitting row i-th row and j column of `A`.

- INPUT:
    - `A`: `Matrix` object representing an $m \times n$ matrix
    - `i`: int index of a row of `A` satisfying $1 \leq i \leq m$
    - `j`: int index of a column of `A` satisfying $1 \leq j \leq n$
- OUTPUT:
    - `Matrix` object of the sub-matrix

```python
In [ ]:  def _submatrix(A : Matrix, i : int, j: int):
         """
         constructs the sub-matrix of an mxn Matrix A that
         results from omitting the i-th row and j-th column;
         i and j satisfy that 0 <= i <= m, and 0 <= j <= n
         :param A: Matrix object
         :param i: int index of row to omit
         :param j: int index of column to omit
         :return: Matrix object representing the sub-matrix
         """

         m, n = A.dim()
         pass # FIXME: Implement this function
```

Problem 3:

Finish the implementation of the function `determinant(A)` which computes the determinant of $n \times n$ matrix `A`.

- INPUT:
    - `A`: `Matrix` object
- OUTPUT:
    - the determinant as a `float` value

```python
In [ ]:  def determinant(A: Matrix):
             """
             computes the determinant of square Matrix A;
             Raises ValueError if A is not a square matrix.
             :param A: Matrix object
             :return: float value of determinant
             """
             m, n = A.dim()
             if m != n:
                 raise ValueError(f"Determinant is not defined for Matrix with dimension {m}x{n
             if n == 1:
                 return None # FIXME: Return the correct value
             elif n == 2:
                 return None # FIXME: Return the correct value
             else:
                 d = 0
                 # FIXME: Update d so that it holds the determinant
                 #        of the matrix.  HINT: You should apply a
                 #        recursive call to determinant()
                 return d
```

Problem 4:

Implement the function `eigen_wrapper(A)` which uses Python built-in function `numpy.linalg.eig()` to create a dictionary with eigenvalues of `Matrix` `A` as keys, and their corresponding list of eigenvectors as values.

- INPUT:
    - `A`: `Matrix` object
- OUTPUT:
    - Python dictionary

```python
In [ ]:  def eigen_wrapper(A: Matrix):
             """
             uses numpy.linalg.eig() to create a dictionary with
             eigenvalues of Matrix A as keys, and their corresponding
             list of eigenvectors as values.
             :param A: Matrix object
             :return: Python dictionary
```

```
    """
    pass # FIXME: Implement this function
```

Problem 5:

Finish the implementation of function `svd(A)` so that it returns the singular value decomposition of A. Recall that the singular value decomposition of a matrix $A \in \mathbb{R}_{m \times n}$ consists of matrices,

- $\Sigma \in \mathbb{R}_{m \times n}$: a diagonal matrix whose main diagonal hold the singular values of $A^T A$ in decreasing order,

$$\begin{bmatrix} \sigma_1 & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & \sigma_r & 0 \\ 0 & \cdots & \cdots & \cdots & 0 \end{bmatrix}$$

i.e., $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r$ where $r = $ number of eigenvalues of $A^T A$

- $V \in \mathbb{R}_{n \times n}$: the matrix whose columns are the eigenvectors of $A^T A$. The order of the columns corresponds to the order of the singular values, i.e. the first column is the eigenvector corresponding to the largest singular value $sigma_1$, the second column is the eigenvector correspoind to $sigma_2$, etc.

- $U \in \mathbb{R}_{m \times m}$: the matrix whose columns are given by $\vec{u_j} = \frac{1}{\sigma_j} A \vec{v_j}$

- INPUT:

  - `A` : `Matrix` object
- OUTPUT:

  - `tuple` with Matrix objects `(U, Sigma, V)`

```
In [ ]:  def svd(A: Matrix):
         """
         computes the singular value decomposition of Matrix A;
         returns Matrix objects U, Sigma, and V such that
             1. V is the Matrix whose columns are eigenvectors of A.transpose() * A
             2. Sigma is a diagonal Matrix of singular values of A.transpose() * A
                appearing in descending order along the main diagonal
             3. U is the Matrix whose j-th column uj satisfies A * vj = sigma_j * uj
                where sigma_j is the j-th singular value in decreasing order and vj is
                the j-th column vector of V
             4. A = U * Sigma * V.transpose()
         :param A: Matrix object
         :return: tuple with Matrix objects; (U, Sigma, V)
         """
         m, n = A.dim()
```

```python
        aTa = A.transpose() * A
        eigen = eigen_wrapper(aTa)

        # Constructing V
        # V should be the mxm matrix whose columns
        # are the eigenvectors of matrix A.transpose() * A
        eigenvectors = list(eigen.values())
        V = Matrix([[None for j in range(n)] for i in range(n)])
        for j in range(1, n + 1):
            pass # FIXME: Replace this with the line that will
                 #        correctly build the entries of V

        # Constructing Sigma
        # Sigma should be the mxn matrix of singular values.
        singular_values = None # FIXME: Replace this so that singular_values
                               #        holds a list of singular values of A
                               #        in decreasing order
        Sigma = Matrix([[0 for j in range(n)] for i in range(m)])
        for i in range(1, m + 1):
            pass # FIXME: Replace this with the line that will correctly
                 #        build the entries of Sigma

        # Constructing U
        # U should be the matrix whose j-th column is given by
        # A * vj / sj where vj is the j-th eigenvector of A.transpose() * A
        # and sj is the corresponding j-th singular value
        U = Matrix([[None for j in range(m)] for i in range(m)])
        for j in range(1, m + 1):
            pass # FIXME: Replace this with the line that will
                 #        correctly build the entries of U
        return (U, Sigma, V)
```