16-8111

Shrinivas Ramasubramanian
shrinivr@andrew.cmu.edu
Assignment 1

September 2024

Question 1. Implement the P A = LDU decomposition algorithm discussed in class. Do so yourself (in other words, do not merely use predefined Gaussian elimination code in MatLab or Python). Simplifications: (i) You may assume that the matrix A is square and invertible. (ii) Do not worry about column interchanges, just row interchanges. Demonstrate (in your pdf) that your implementation works properly, on some examples.

Solution We implement a Python class to perform **Gaussian Elimination** on a matrix, decomposing it into permutation (P), lower triangular (L), diagonal (D), and upper triangular (U) matrices.

- **Initialization:** The matrix `A` is deep-copied and converted to `np.float64`. Matrices P, L, D, and U are initialized.
- **Permutations:** If a pivot is zero, rows are swapped in `A` and `P` to maintain numerical stability.
- **Lower Triangularization:** Elements below the diagonal are eliminated via row operations stored in `L`, followed by adding the identity matrix to finalize `L`.
- **Diagonal and Upper Matrices:** The diagonal and upper triangular matrices are extracted from the transformed `A`, with `U` normalized to have ones on the diagonal.
- **Final Decomposition:** The class returns matrices `L`, `D`, `U`, and `P`.

```python
import numpy as np
import copy

class GaussianElimination:
    """
    A class to perform Gaussian Elimination on a given matrix A, decomposing it into
    permutation (P), lower triangular (L), diagonal (D), and upper triangular (U) matrices.
    """

    def __init__(self, A):
        """
        Initialize the GaussianElimination class with matrix A.

        Parameters:
        -----------
        A : np.ndarray
            The matrix to be decomposed.
        """
        self.A = A.astype(np.float64)  # Ensure matrix A is of dtype np.float64
        self.A_ = copy.deepcopy(self.A)  # Create a deep copy of A to avoid modifying the original matrix
        self.initialize_matrices()

    def initialize_matrices(self):
        """
        Initialize matrices P, L, D, and U based on the dimension of matrix A.
        """
        self.dim = self.A.shape[0]

        # Initialize permutation matrix P as an identity matrix
        self.P = np.eye(self.dim, dtype=np.float64)

```

```python
        # Initialize L as a zero matrix and U, D as zero matrices
        self.L = np.zeros((self.dim, self.dim), dtype=np.float64)
        self.D = np.zeros((self.dim, self.dim), dtype=np.float64)
        self.U = np.zeros((self.dim, self.dim), dtype=np.float64)

    def first_non_zero_index(self, idx):
        """
        Find the index of the first non-zero element in the sub-array starting from index `idx`.

        Parameters:
        -----------
        idx : int
            The starting index for the search in the sub-array.

        Returns:
        --------
        int
            The index of the first non-zero element in the sub-array.
        """
        column = self.A_[idx:, idx]
        non_zero_indices = np.nonzero(column)[0]

        return non_zero_indices[0] + idx

    def permutation_matrix_colswap(self, col1, col2):
        """
        Swap two columns in the permutation matrix P.

        Parameters:
        -----------
        col1 : int
            The first column to swap.
        col2 : int
            The second column to swap.
        """
        self.P[:, [col1, col2]] = self.P[:, [col2, col1]]

    def swap_rows(self, matrix, row1, row2):
        """
        Swap two rows in a given matrix.

        Parameters:
        -----------
        matrix : np.ndarray
            The matrix in which rows will be swapped.
        row1 : int
            The first row to swap.
        row2 : int
            The second row to swap.

        Returns:
        --------
        np.ndarray
            The matrix with the specified rows swapped.
```

```python
        """
        matrix[[row1, row2], :] = matrix[[row2, row1], :]
        return matrix

    def lower_triangularization(self):
        """
        Perform lower triangularization of matrix A.

        This method transforms matrix A into a lower triangular form by applying
        a series of row operations, and updates matrix L accordingly.
        """
        for i in range(self.dim):
            # Handle zero pivot by row swapping
            if self.A_[i, i] == 0:
                interchange_idx = self.first_non_zero_index(i)
                self.permutation_matrix_colswap(i, interchange_idx)  # Swap columns in permutation matrix
                self.A_ = self.swap_rows(self.A_, i, interchange_idx)  # Swap rows in A_
                self.L = self.swap_rows(self.L, i, interchange_idx)  # Swap rows in L

            # Perform row operations to create zeros below the pivot
            for j in range(i + 1, self.dim):
                multiplier = self.A_[j, i] / self.A_[i, i]
                self.L[j, i] = multiplier
                self.A_[j, :] -= multiplier * self.A_[i, :]

        # Add identity matrix to L to finalize lower triangular form
        self.L += np.eye(self.dim)

    def extract_diagonal_and_upper(self):
        """
        Extract the diagonal (D) and upper triangular (U) matrices from A_.
        """
        for i in range(self.dim):
            self.D[i, i] = self.A_[i, i]  # Extract diagonal elements
            self.U[i, :] = self.A_[i, :]  # Extract upper triangular part

        # Normalize U to ensure ones on the diagonal
        for i in range(self.dim):
            if self.D[i, i] != 0:
                self.U[i, :] /= self.D[i, i]

    def decompose(self):
        """
        Perform Gaussian Elimination to decompose matrix A into L, D, U, and P.

        Returns:
        --------
        tuple of np.ndarray
            A tuple containing the lower triangular matrix (L),
            diagonal matrix (D), upper triangular matrix (U), and permutation
```

```python
                    matrix (P).
        """
        self.lower_triangularization()
        self.extract_diagonal_and_upper()
        return self.L, self.D, self.U, self.P
```

Listing 1: Gaussian Elimination Class in Python

Here is an example of the class in use for a sample 3×3 matrix A:

```python
>>> A = np.array([[1,3,2], [-2,-6,1], [2,5,7]], dtype=np.float64)
>>> GE = GaussianElimination(A)
>>> L,D,U,P = GE.decompose()
>>> print(L)
array([[ 1.,  0.,  0.],
       [ 2.,  1.,  0.],
       [-2.,  0.,  1.]])
>>> print(D)
array([[ 1.,  0.,  0.],
       [ 0., -1.,  0.],
       [ 0.,  0.,  5.]])
>>> print(U)
array([[ 1.,  3.,  2.],
       [-0.,  1., -3.],
       [ 0.,  0.,  1.]])
>>> print(P)
array([[1., 0., 0.],
       [0., 0., 1.],
       [0., 1., 0.]])
```

Listing 2: Gaussian Elimination Class usage for a sample 3×3 matrix A

To reproduce the attached code, run 16-811/Assignment-1/question1.ipynb

Question 2 Compute the $PA = LDU$ decomposition and the SVD decomposition for each of the following matrices:

$$A_1 = \begin{pmatrix} 10 & -10 & 0 \\ 0 & -4 & 2 \\ 2 & 0 & -4 \end{pmatrix} \quad A_2 = \begin{pmatrix} 5 & -5 & 0 & 0 \\ 5 & 5 & 5 & 0 \\ 0 & -1 & 4 & 1 \\ 0 & 4 & -1 & 2 \\ 0 & 0 & 2 & 1 \end{pmatrix} \quad A_3 = \begin{pmatrix} 1 & 1 & 1 \\ 10 & 2 & 9 \\ 8 & 0 & 7 \end{pmatrix}$$

Solution

1. First lets compute SVD of A_1

```
>>> import numpy as np
>>> A_1 = np.array([[10,-10, 0],
                    [0, -4,  2],
                    [2,  0, -4]])
>>> U, S, VT = np.linalg.svd(A_1)
>>> U
array([[-0.97586837, -0.17298018,  0.13326206],
       [ 0.19874941, -0.45086605,  0.87018301],
       [-0.09044108,  0.87566983,  0.47436563]])
>>> S
array([14.49903717,  4.57803496,  3.13361088])
>>> VT
array([[-0.68553282, -0.72788843, -0.01493928],
       [ 0.00470461,  0.0160904 , -0.99985947],
       [ 0.72802652, -0.68550677, -0.00760607]])
```

Now lets compute the LDU decomposition.

Operation	P	L	DU
$r_3 \leftarrow r_3 - 0.2 \cdot r_1$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 10 & -10 & 0 \\ 0 & -4 & 2 \\ 0 & 2 & -4 \end{pmatrix}$
$r_3 \leftarrow r_3 + 0.5 \cdot r_2$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & -0.5 & 1 \end{pmatrix}$	$\begin{pmatrix} 10 & -10 & 0 \\ 0 & -4 & 2 \\ 0 & 0 & -3 \end{pmatrix}$

Hence we get the following result

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & -0.5 & 1 \end{pmatrix} \quad U = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -0.5 \\ 0 & 0 & 1 \end{pmatrix} \quad D = \begin{pmatrix} 10 & 0 & 0 \\ 0 & -4 & 0 \\ 0 & 0 & -3 \end{pmatrix}$$

And the Permutation matrix P is I_3

2. Lets compute SVD of A_2

```
>>> import numpy as np
>>> A_2 = np.array([[ 5, -5, 0, 0],
                    [ 5,  5, 5, 0],
                    [ 0, -1, 4, 1],
                    [ 0,  4, -1, 2],
                    [ 0,  0, 2, 1]])
>>> U, S, VT = np.linalg.svd(A_2)
>>> U
array([[ 0.1126266 ,  0.86708016,  0.37480574, -0.30750061, -0.0212435 ],
       [-0.93215705,  0.15225202,  0.1626033 ,  0.2846251 ,  0.0212435 ],
       [-0.20196646,  0.2257811 , -0.74973268, -0.31691184, -0.4956816 ],
       [-0.2398716 , -0.41225956,  0.38334628, -0.77085017, -0.17702914],
       [-0.14166742,  0.06368894, -0.35217519, -0.36026217,  0.84973988]])
>>> S
array([9.14492811, 7.79814769, 4.42070712, 2.23976139])
>>> VT
array([[-0.44807922, -0.65407164, -0.60275097, -0.09003647],
       [ 0.65357327, -0.69875055,  0.28263403, -0.06861233],
       [ 0.60783153,  0.27645026, -0.74051747, -0.07582844],
       [-0.05106685,  0.08667875,  0.09188657, -0.99067443]])
```

Listing 3: SVD of A_2

Now lets compute the LDU decomposition.

Operation	P	L	DU
$r_3 \leftarrow r_3 - r_1$	$\begin{pmatrix}1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\end{pmatrix}$	$\begin{pmatrix}1&0&0&0&0\\1&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\end{pmatrix}$	$\begin{pmatrix}5&-5&0&0\\0&10&5&0\\0&-1&4&1\\0&4&-1&2\\0&0&2&1\end{pmatrix}$
$r_3 \leftarrow r_3 + 1/10 \cdot r_2$ $r_4 \leftarrow r_4 - 2/5 \cdot r_2$	$\begin{pmatrix}1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\end{pmatrix}$	$\begin{pmatrix}1&0&0&0&0\\1&1&0&0&0\\0&-0.1&1&0&0\\0&2/5&0&1&0\\0&0&0&0&1\end{pmatrix}$	$\begin{pmatrix}5&-5&0&0\\0&10&5&0\\0&0&4.5&1\\0&0&-3&2\\0&0&2&1\end{pmatrix}$
$r_4 \leftarrow r_4 + 2/3 \cdot r_3$ $r_5 \leftarrow r_5 - 4/9 \cdot r_3$	$\begin{pmatrix}1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\end{pmatrix}$	$\begin{pmatrix}1&0&0&0&0\\1&1&0&0&0\\0&-0.1&1&0&0\\0&2/5&-2/3&1&0\\0&0&4/9&0&1\end{pmatrix}$	$\begin{pmatrix}5&-5&0&0\\0&10&5&0\\0&0&4.5&1\\0&0&0&8/3\\0&0&0&5/9\end{pmatrix}$
$r_5 \leftarrow r_5 - 5/24 \cdot r_4$	$\begin{pmatrix}1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\end{pmatrix}$	$\begin{pmatrix}1&0&0&0&0\\1&1&0&0&0\\0&-0.1&1&0&0\\0&2/5&-2/3&1&0\\0&0&4/9&5/24&1\end{pmatrix}$	$\begin{pmatrix}5&-5&0&0\\0&10&5&0\\0&0&4.5&1\\0&0&0&8/3\\0&0&0&0\end{pmatrix}$

Assuming $D_{(5,5)} = 1$ (since the last row is zeros for U), from above we can infer that

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & -1/10 & 1 & 0 & 0 \\ 0 & 2/5 & -2/3 & 1 & 0 \\ 0 & 0 & 4/9 & 5/24 & 1 \end{pmatrix} \quad U = \begin{pmatrix} 1 & -1 & 0 & 0 \\ 0 & 1 & 1/2 & 0 \\ 0 & 0 & 1 & 2/9 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad D = \begin{pmatrix} 5 & 0 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & 9/2 & 0 & 0 \\ 0 & 0 & 0 & 3/8 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

and P is I_5

3. Lets compute SVD of A_3

```
>>> import numpy as np
>>> A_3 = np.array([[ 1, 1, 1],
                    [ 10, 2, 9],
                    [ 8, 0, 7]])
>>> U, S, VT = np.linalg.svd(A_3)
>>> U
array([[-0.08686637, -0.57077804, -0.81649658],
       [-0.78592384, -0.4643889 ,  0.40824829],
       [-0.6121911 ,  0.67716718, -0.40824829]])
>>> S
array([1.72832333e+01, 1.51322343e+00, 1.63228421e-15])
>>> VT
array([[-0.74312678, -0.09597244, -0.66223249],
       [ 0.1339329 , -0.99096789, -0.0066797 ],
       [ 0.65561007,  0.09365858, -0.74926865]])
```

Listing 4: SVD of A_3

Now lets do the LDU decomposition of A_3

Operation	P	L	DU
$r_2 \leftarrow r_2 - 10 \cdot r_1$ $r_3 \leftarrow r_3 - 8 \cdot r_1$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 10 & 1 & 0 \\ 8 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & -8 & -1 \\ 0 & -8 & -1 \end{pmatrix}$
$r_3 \leftarrow r_3 - \cdot r_2$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 10 & 1 & 0 \\ 8 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & -8 & -1 \\ 0 & 0 & 0 \end{pmatrix}$

Hence from the above calculation, we get:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 10 & 1 & 0 \\ 8 & 1 & 1 \end{pmatrix} \quad U = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1/8 \\ 0 & 0 & 0 \end{pmatrix} \quad D = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -8 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

To reproduce the attached code, run 16-811/Assignment-1/question2.ipynb

Question 3 olve the systems of equations Ax = b for the values of A and b given below. For each system, specify whether the system has zero, one, or many exact solutions. If a system has zero exact solutions, give “the SVD solution” (as defined in class) and explain what this solution means. If a system has a unique exact solution, compute that solution. If a system has more than one exact solution, specify both “the SVD solution” and all solutions, using properties of the SVD decomposition of the matrix A, as discussed in class. Show your work, including verifying that your answers are correct.

$$\text{a)} \quad A = \begin{pmatrix} 10 & -10 & 0 \\ 0 & -4 & 2 \\ 2 & 0 & -4 \end{pmatrix} \quad b = \begin{pmatrix} 10 \\ 2 \\ 13 \end{pmatrix}$$

$$\text{b)} \quad A = \begin{pmatrix} 1 & 1 & 1 \\ 10 & 2 & 9 \\ 8 & 0 & 7 \end{pmatrix} \quad b = \begin{pmatrix} 1 \\ 3 \\ 1 \end{pmatrix}$$

$$\text{c)} \quad A = \begin{pmatrix} 1 & 1 & 1 \\ 10 & 2 & 9 \\ 8 & 0 & 7 \end{pmatrix} \quad b = \begin{pmatrix} 3 \\ 2 \\ 2 \end{pmatrix}$$

Solution:

a) Borrowing from the solution of A_1 from **Question 2**, we use the L, D, U matrix. We use the equation $Ly = b$ where $y = DUx$

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & -0.5 & 1 \end{pmatrix} \;\&\; Ly = \begin{pmatrix} 10 \\ 2 \\ 13 \end{pmatrix} \implies y = \begin{pmatrix} 10 \\ 2 \\ 12 \end{pmatrix}$$

Since $DUx = y$, $Ux = D^{-1}y$, and since D is a diagonal matrix with non-zero entries we get

$$Ux = D^{-1}y = \begin{pmatrix} 1 \\ -1/2 \\ -4 \end{pmatrix}$$

Now to solve for x we solve from the bottom row and back substitute the values for the upper row equations.

$$\begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -0.5 \\ 0 & 0 & 1 \end{pmatrix} x = \begin{pmatrix} 1 \\ -1/2 \\ -4 \end{pmatrix} \implies x = \begin{pmatrix} -3/2 \\ -5/2 \\ -4 \end{pmatrix}$$

This system has 1 exact solution since $\mathcal{N}(A) = \phi$

b) we shall use the solution for A_3 from **Question 2**.

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 10 & 1 & 0 \\ 8 & 1 & 1 \end{pmatrix} \;\&\; Ly = \begin{pmatrix} 1 \\ 3 \\ 1 \end{pmatrix} \implies y = \begin{pmatrix} 1 \\ -7 \\ 0 \end{pmatrix}$$

Since $DUx = y$, $Ux = D^{-1}y$, and since D is a diagonal matrix with non-zero entries we get

$$Ux = D^{-1}y = \begin{pmatrix} 1 \\ 7/8 \\ 0 \end{pmatrix}$$

Note that we assumed $D_{3,3} = 1$ for convenience, there can be infintely more solutions. Now we solve for $D^{-1}y = Ux$ by going row wise from bottom.
Since U_1 row is all zeros, we assign any value to x_3. Let $x_3 = 7$, this will simplify calculations.
Hence:

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1/8 \\ 0 & 0 & 0 \end{pmatrix} x = \begin{pmatrix} 1 \\ 7/8 \\ 0 \end{pmatrix} \implies x = \begin{pmatrix} -6 \\ 0 \\ 7 \end{pmatrix}$$

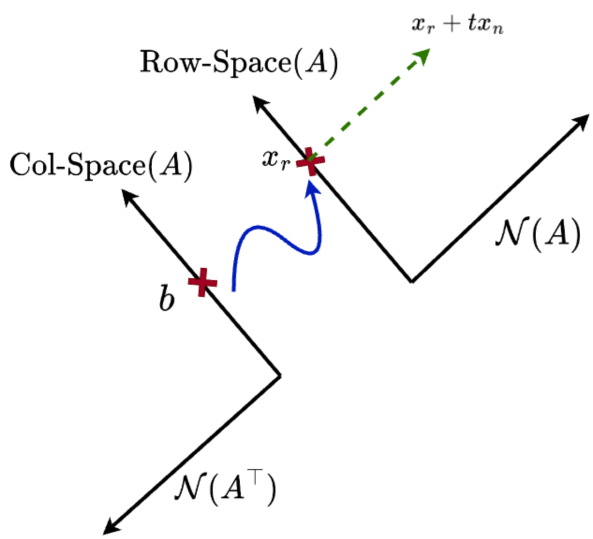

Figure 1: Diagramatic representation of problem 3.b The case of infinite solutions, b resides in the column space of A and the solution is of the form $x_r + tx_n$

Note: this system has infinitely many solutions since $\text{Rank}(A) = 2 < 3 = \text{Dim}(x)$ and all $x = x_r + tx_n$ is a solution of $Ax = b$ where xn is a sample in the null space of A and x_r resides in the row space. Let us assume $x_3 = t$, where $t \in \mathbb{R}$. Solving the above system gives us:

$$x = \begin{pmatrix} (1 - 7t)/8 \\ (7 - t)/8 \\ t \end{pmatrix}$$

Hence all solutions of the form $x = \begin{pmatrix} 1/8 \\ 7/8 \\ 0 \end{pmatrix} + t \begin{pmatrix} -7/8 \\ -1/8 \\ 1 \end{pmatrix}$ is a solution to the given $Ax = b$.

Now lets compare this to the SVD solution:

```
>>>import numpy as np
>>>A_3 = np.array([[ 1, 1, 1],
                   [ 10, 2, 9],
                   [ 8, 0, 7]])

>>>U, S, VT = np.linalg.svd(A_3)
>>>b=np.array([1,3,1])
>>> S
array([1.72832333e+01, 1.51322343e+00, 1.63228421e-15])
>>>x_= VT.T @ np.linalg.inv(np.diag(S)) @ U.T @ b
>>>x_
array([-0.1875    ,  0.8359375,  0.3125    ])
>>>VT.T[:, 2]
array([ 0.65561007,  0.09365858, -0.74926865])
```

Listing 5: SVD solution for $Ax = b$ under infite solution case example

As we can see the singular value is 0 (numerical implementation of svd in numpy yields a number of order 10^{-15}) for the last entry, (Ref. Listing 4.) the singular vector corrsponding to this (V_3) spans the null space and the component of solution in row space x_r is spanned by V_1 and V_2. Also we can clearly see that V_3 is just a scaling of $\begin{pmatrix} -7/8 \\ -1/8 \\ 1 \end{pmatrix}$ which we know spans the null space of A.

c) Along similar lines of previous question,

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 10 & 1 & 0 \\ 8 & 1 & 1 \end{pmatrix} \ \& \ Ly = \begin{pmatrix} 3 \\ 2 \\ 2 \end{pmatrix} \implies y = \begin{pmatrix} 3 \\ -28 \\ 6 \end{pmatrix}$$

Since $DUx = y$, $Ux = D^{-1}y$, and since D is a diagonal matrix with non-zero entries we get

$$Ux = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1/8 \\ 0 & 0 & 0 \end{pmatrix} x = D^{-1}y = \begin{pmatrix} 3 \\ 7/2 \\ 6 \end{pmatrix}$$

Uh oh!, we have a problem, no matter what x is the last row will always be 0 hence this system has zero exact solution. Lets find the SVD solution using numpy.

$$x = V\Sigma^{-1}U^{\top}b$$

Upon back substitution from the solution of **Question 2** part 3, we get:

```
>>> import numpy as np
>>> A = np.array([[ 1, 1, 1],
                  [ 10, 2, 9],
                  [ 8, 0, 7]])
>>> U, S, VT = np.linalg.svd(A)
>>> b=np.array([3,2,2])
>>> x_= VT.T @ np.linalg.inv(np.diag(S)) @ U.T @ b
>>> x_
array([-9.83842227e+14, -1.40548890e+14,  1.12439112e+15])
```

Listing 6: SVD solution for $Ax = b$

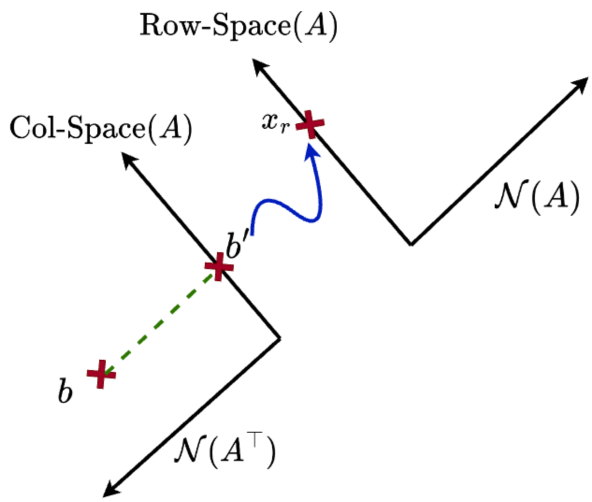

Figure 2: Representation of problem 3.c The case of inexact solution (no-solution), $x_r = V\Sigma^{-1}U^\top b$

This solution corresponds to b' obtained by projecting b onto the columns space of A. $U^\top b$ projects b to the column space and Σ^-1 rescales the vector along the column space. Upon multiplication with V reprojects it back to the row-space with no component existing in the null space of A

To reproduce the attached code, run 16-811/Assignment-1/question3.ipynb

Question 4: Suppose that u is an n-dimensional column vector of unit length in $\mathbb{R}^n$, and let $u^\top$ be its transpose. Then $uu^\top$ is a matrix. Consider the $n \times n$ matrix $A = I - uu^\top$.

(a) Describe the action of the matrix A geometrically.

(b) Give the eigenvalues of A.

(c) Describe the null space of A.

(d) What is A^2?

Solution:

a) $Av = v - \begin{pmatrix} u_1 \cdot u^\top \\ ... \\ u_n \cdot u^\top \end{pmatrix} v = v - (u^\top v) \cdot u$. Hence geometrically this matrix operation eliminates all components of a vector along the direction of u resulting in only the component of v orthogonal to u since $(Av)^\top u = 0$. uv.

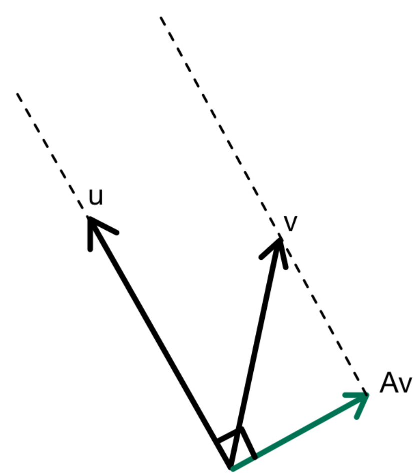

Figure 3: The effect of applying the transformation A on a vector v

b) $Au = 0.u$ implies u is an eigenvector with eigenvalue 0. Which also implies u spans the null space of A. Since we are in $\mathbb{R}^n$, it would have a basis on n orthogonal vectors (eigenvectors). since u is one of them and the rest are orthogonal to it, all v s.t. $v \perp u$ satisfies $Av = v$ implying an eigen value of 1. Hence there are $n - 1$ eigenvalue 1 and the last eigen value is 0.

c) From b), the null space is the space of all vectors aligned with vector u, i.e. of the for tu

d) $A^2 = A^\top A = (I - (uu^\top))^\top (I - uu^\top) = (I - uu^\top)(I - uu^\top) = I - 2uu^\top + uu^\top uu^\top = I - uu^\top$, Hence this is the same matrix transformation again.

Question 5: The following problem arises in a large number of robotics and vision problems: Suppose $p_1, \ldots, p_n$ are the 3D coordinates of n points located on a rigid body in three-space. Suppose further that $q_1, \ldots, q_n$ are the 3D coordinates of these same points after the body has been translated and rotated by some unknown amount. Derive an algorithm in which Singular Value Decomposition (SVD) plays a central role for inferring the body's translation and rotation. (You may assume that the coordinate values are precise, not noisy, but see the comment and caution below.)

Show (in your PDF) that your algorithm works correctly by running it on some examples.

Comment: This problem requires some thought. There are different approaches. Although you can find a solution on the web or in a vision textbook, try to solve the problem yourself before looking at any such sources. Spend some time on the problem. It is good practice to develop your analytic skills. Feel free to discuss among yourselves. (As always, cite any sources, including discussions with others.)

Requirement: Your algorithm should make use of all the information available. True, in principle, you only need three pairs of points — but if you use more points, your solution will be more robust, something that might come in handy some day when you need to do this for real with noisy data.

Caution: A common mistake is to derive an algorithm that finds the best affine transformation, rather than the best rigid body transformation. Even though you may assume precise coordinate values, imagine how your algorithm would behave with noise. Your algorithm should still produce a rigid body transformation.

Hint: Suppose for a moment that both sets of points have the origin as centroid. Assemble all the points $\{p_i\}$ into a matrix P and all the points $\{q_i\}$ into another matrix Q. Now think about the relationship between P and Q. You may wish to find a rigid body transformation that minimizes the sum of squared distances between the points $\{q_i\}$ and the result of applying the rigid body transformation to the points $\{p_i\}$.

You may find the following facts useful (assuming the dimensions are sensible):

$$\|x\|^2 = x^\top x, \quad x^\top R^\top y = \mathrm{Tr}(Rxy^\top).$$

[Here x and y are column vectors (e.g., 3D vectors) and R is a matrix (e.g., a 3×3 rotation matrix). The superscript T means transpose, so $x^\top x$ is a number and $xy^\top$ is a matrix. Also, Tr is the trace operator that adds up the diagonal elements of its square matrix argument.]

You will have more complicated expressions for x and y, involving the points $\{p_i\}$ and $\{q_i\}$.

Solution : Let us solve the Translation and rotation independently. The translation required to go from $P \to Q$ can be found by:

$$\begin{aligned}
\arg\min_T \sum_{i=1\ldots n} ||p_i + T - q_i||^2 &= \arg\min_T \sum_{i=1\ldots n} (p_i + T - q_i)^\top (p_i + T - q_i) \\
&= \arg\min_T \sum_{i=1\ldots n} (p_i^\top p_i + q_i^\top q_i) + nT^\top T + 2T^\top \sum_{i=1..n} (p_i - q_i)
\end{aligned}$$

Since $\sum_{i=1\ldots n}(p_i^\top p_i + q_i^\top q_i)$ is not a function of T we can get rid of them from the objective.

$$\begin{aligned}
&\arg\min_T nT^\top T + 2T^\top (\sum_{i=1..n} (p_i - q_i)) \\
&= \arg\min_T nT^\top T + 2T^\top (\sum_{i=1..n} (p_i - q_i))
\end{aligned}$$

Upon factorizing we get the quadratic equation :

$$\arg\min_{T} T^{\top}(nT + 2(\sum_{i=1..n}(p_i - q_i))$$

We know this is a upward facing parabola and the minima is achieved at the mid point of the two roots, hence $T = \sum_{i=1}^{n}(p_i - q_i)/n$.

For ease of computation of the required rotation R we shall assume $\tilde{q}_i = q_i - T$ so that their centroid align to compute the rotation. We know that $Rp_i = \tilde{q}_i$ where R is a rotation matrix. We know that since R is a rotation only, there is no scaling along the axis. R minimizes the square distance error post-rotation, hence

$$\begin{aligned} &\arg\min_{R} \sum_{i} (Rp_i - \tilde{q}_i)^2 \\ = &\arg\min_{R} \sum_{i} p_i^{\top} R^{\top} R p_i - \tilde{q}_i^{\top} \tilde{q}_i - 2\tilde{q}_i^{\top} R p_i \\ = &\arg\min_{R} \sum_{i} p_i^{\top} R^{\top} R p_i - \tilde{q}_i^{\top} \tilde{q}_i - 2\tilde{q}_i^{\top} R p_i \end{aligned}$$

Since $R^{\top}R = I$ and $p_i^{\top}p_i$, $\tilde{q}_i^{\top}\tilde{q}_i$ are not a function of R

$$\arg\min_{R} \sum_{i} -2\tilde{q}_i^{\top} R p_i = \arg\max_{R} \sum_{i} \tilde{q}_i^{\top} R p_i$$

The above objective can be rewritten as the trace of $\tilde{Q}^{\top}RP$ where $\tilde{Q}$ and P are the matrices whose columns are the vectors $\tilde{q}_i$ and p_i respectively. By using the properties of the trace Tr() of a matrix, the objective can be rewritten as:

$$= \arg\max_{R} \mathrm{Tr}(\tilde{Q}^{\top} R P) = \arg\max_{R} \mathrm{Tr}(R P \tilde{Q}^{\top})$$

Upon performing the SVD of $P\tilde{Q}^{\top} = USV^{\top}$ and using $\mathrm{Tr}(AB) = \mathrm{Tr}(BA)$ we get

$$\arg\max_{R} \mathrm{Tr}(RUSV^{\top}) = \arg\max_{R} \mathrm{Tr}(SV^{\top}RU)$$

Let's assume $G = V^{\top}RU$, which is an orthogonal matrix since it is the product of 3 orthogonal matrices. The trace of SM is $\sum_{i}^{n}(S_{i,i}M_{i,i})$ which only maximizes when $M_{ii} = 1$, but since it is also orthogonal, $M = I$, hence

$$M = I \implies V^{\top}RU = I \implies R = VU^{\top}$$

Algorithm 1 Rigid Body Translation and Rotation via SVD

Input: 3D point sets $\{p_i\}$, $\{q_i\}$ for $i = 1, \ldots, n$
Output: Translation T, Rotation R
Compute translation $T \leftarrow \frac{1}{n}\sum_{i=1}^{n}(q_i - p_i)$
Align centroids: $\tilde{q}_i \leftarrow q_i - T$
Compute rotation by solving $\arg\max_R \operatorname{Tr}(RP\tilde{Q}^\top)$
Perform SVD: $P\tilde{Q}^\top \leftarrow USV^\top$
Set $R \leftarrow VU^\top$
Return: T, R

The following Python class calculates the translation and rotation matrices needed for rigid body motion between two 3D point sets using Singular Value Decomposition (SVD).

```
import numpy as np

class RigidBodyTransformation:
    def __init__(self, P, Q):
        """
        Initialize with two 3D point sets P and Q.

        Parameters:
        P (numpy.ndarray): 3D point set before transformation (n x 3)
        Q (numpy.ndarray): 3D point set after transformation (n x 3)
        """
        self.P = P
        self.Q = Q
        self.T = None  # Translation vector
        self.R = None  # Rotation matrix

    def compute_translation(self):
        """
        Compute the translation vector T as the difference between centroids.
        """
        centroid_P = np.mean(self.P, axis=0)
        centroid_Q = np.mean(self.Q, axis=0)
        self.T = centroid_Q - centroid_P

    def compute_rotation(self):
        """
        Compute the rotation matrix R using SVD.
        """
        # Step 2: Align centroids
        P_centered = self.P - np.mean(self.P, axis=0)
        Q_centered = self.Q - np.mean(self.Q, axis=0)

        # Step 3: Compute rotation matrix
        H = P_centered.T @ Q_centered  # Covariance matrix
        U, S, Vt = np.linalg.svd(H)    # SVD decomposition
        self.R = Vt.T @ U.T

        # Ensure a proper rotation (det(R) = 1, no reflection)
        if np.linalg.det(self.R) < 0:
            Vt[2, :] *= -1
            self.R = Vt.T @ U.T
```

```python

    def infer_transformation(self):
        """
        Perform the full inference of translation and rotation.
        """
        self.compute_translation()
        self.compute_rotation()
        return self.T, self.R

    def apply_transformation(self, P):
        """
        Apply the computed transformation to a given 3D point set P.

        Parameters:
        P (numpy.ndarray): 3D point set to transform (n x 3)

        Returns:
        numpy.ndarray: Transformed point set.
        """
        if self.T is None or self.R is None:
            raise ValueError("Transformation not yet computed. Call infer_transformation first.")

        return (P @ self.R.T) + self.T
```

Listing 7: Calculate Rotation and Translation for a rigid body motion in Python

```python
>>> import numpy as np
>>>P = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])
>>>Q = np.array([[0.0, 1.0, 1.0], [1.0, 0.0, 1.0], [1.0, 1.0, 0.0]])
>>>transformer = RigidBodyTransformation(P, Q)
>>>T, R = transformer.infer_transformation()
>>>T
array([0.33333333, 0.33333333, 0.33333333])
>>>R
array([[-0.33333333,  0.66666667,  0.66666667],
       [ 0.66666667, -0.33333333,  0.66666667],
       [ 0.66666667,  0.66666667, -0.33333333]])
>>>transformed_P = transformer.apply_transformation(P)
>>>transformed_P
array([[ 1.11022302e-16,  1.00000000e+00,  1.00000000e+00],
       [ 1.00000000e+00, -1.11022302e-16,  1.00000000e+00],
       [ 1.00000000e+00,  1.00000000e+00,  5.55111512e-17]])
```

Listing 8: Checking Rigid body transformation

To reproduce the attached code, run 16-811/Assignment-1/question5.ipynb