BLG202E - Project 5 - Kabsch-Umeyama Algorithm

Racha Badreddine
150210928

***Abstract*—This report describes the steps followed to implement Kabsch-Umeyama Algorithm given multiples datasets to work on.**

***Index Terms*—Kabsch-Umeyama - SVD decomposition - Rotation - Translation**

I. Introduction

In this project, we were asked to implement Kabsch-Umeyama Algorithm using python and work on different datasets provided. To be able to implement this algorithm I first started by implementing the Singular Value Decomposition (SVD) of a Matrix. The different results are provided in the Results section.

II. Methods

A. SVD decomposition

A = U Σ V^T represents the SVD decomposition of a given matrix A where U is the matrix whose columns are the left singular vectors, Σ is the matrix that has the singular values on its diagonal and V is the matrix whose columns are the right singular vectors. The left singular values are The orthonormal eigenvectors of AA^T where the right singular values are The orthonormal eigenvectors of A^TA, and the singular values are the square root of eigenvalues of these matrices.

There are different methods to follow while implementing the SVD. The first method i used was the QR method. This is an iterative Algorithm that finds the Eigenvalues and eigenvectors of a given Matrix using the QR factorization of the Matrix. After implementing the simple QR Method it had high errors when i worked with big matrices as the one that we will use for our Algorithm. That is why an improved method has been used. The Algorithm is as follows for a matrix A:

Algorithm 1 Improved QR Method

```
Input  : Matrix $A$
Output: Array of Eigenvalues and Matrix of Eigenvectors
$V = I$  for $k = 0, 1, \ldots, n$ do
    Find the last element of $A$'s diagonal
    Create a diagonal matrix out of this element $L$
    Calculate the QR factorization of $A_k - L$
    $A_{k+1} = R \times Q + L$
    $V = V \times Q$
end
```

After iterating n times the V Matrix will contain the Eigenvectors of the Matrix while A will contain the Eigenvalues on its diagonal. After computing them i sorted them in a descending order since it is a convention.[Appendix]

since it was not mentioned if we are allowed to use the QR factorization function from the numpy.linalg library i used it to work on the Algorithm. At the same time i implemented a function that calculates it using Gram Schmidt method. The problem i faced is that the error given by this function when woking on the plane example is higher than the one provided in the library. However, when i tried to use it on the other data sets i got nan result. That is why the results of the other data sets are provided using the qr factorization of the library. A comparison between both results will be provided in the Results Section.

Algorithm 2 QR Factorization with Gram-Schmidt

```
Input  : Matrix $A$
Output: Matrices $Q$ and $R$
Initialize $Q$ and $R$ with the appropriate size
for $i = 0$ to $columns - 1$ do
    $v = A_i$
    for $j = 0$ to $i - 1$ do
        $v = v -$ dot product$_i(v, Q_j) \times Q_j$
    end
    $Q_i = v / \|v\|$
end
for $i = 0$ to $columns - 1$ do
    for $j = i$ to $columns - 1$ do
        $R_{ij} =$ dot product$(Q_i, A_j)$
    end
end
return $Q, R$
```

Then for the SVD, i found the eigenvalues and vectors of A^TA using QR Method, the singular values were calculated as the square root of the eigen values. After that U was found as: $U = A \times V \times \Sigma^T$

B. Kabsch-Umeyama Algorithm Implementaion

Kabsch Umeyama ALgorithm aims to merge two point sets based on the correspondences (The common points between them). To implement This algorithm were provided by 4 different data sets each represents a different body, bottle, cup, chair and plane. For the last data set (Plane) we were also provided by the expected results, this is the set i will use to calculate the error of my implementation. Our data set consists of 3 text files representing point set1, point set2 and the indices of corresponding points. The data is the x, y, z coordinates of these points in the 3D space.

1. I started by reading the files and storing each one in a Matrix of size n x d (d = 3). After that, I created two Matrices that contain only the corresponding points provided in the third text file.

2. I applied the Kabsch Umeyama ALgorithm on these matrices. This algorithm returns the Rotation and translation needed to merge the data. The rotation matrix was calculated according to the Algorithm in the paper of Lawrence et. al. [1].

Algorithm 3 Kabsch-Umeyama Algorithm

```
Input  : Matrices $Q$ and $P$
Output: Rotation Matrix and Translation vector
Calculate $centroid_Q$ and $centroid_P$
Center Matrix $Q_{\text{centered}}$ and $P_{\text{centered}}$
Calculate Rotation Matrix with its Algorithm
$T = centroid_P - rotation \times centroid_Q$
return $rotation, T^T$
```

The steps provided above are the ones i followed to implement this Algorithm. The Rotation calculated using the Algorithm in the paper of Lawrence et. al. [1]. Then using the equation: $P_{dxn} = R_{dxd} \times Q_{dxn}$ + T_{dxn} I applied the inverse on P as: $Q_{dxn} = R^T_{dxd} \times (P_{dxn} - T_{dxn})$. Then i merged the data by dropping the overlapping points.

III. Results

A. Plane Example

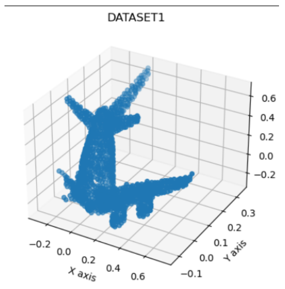


Fig. 1. points set 1

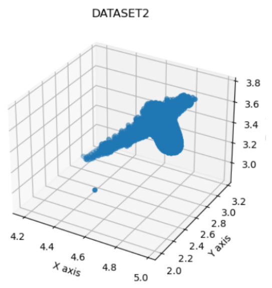


Fig. 2. points set 2

Merging the two point sets gave the following :

1. Using the QR factorization of the library. it gave the following rotation matrix:

$$\begin{bmatrix} 0.5560261 & 0.7127846 & -0.42751968 \\ 0.38869472 & 0.23166031 & 0.89176786 \\ 0.73467774 & -0.66202085 & -0.14824643 \end{bmatrix}$$

and translation vector:

$$\begin{bmatrix} 4.6744464 & 2.95364655 & 3.44697803 \end{bmatrix}$$

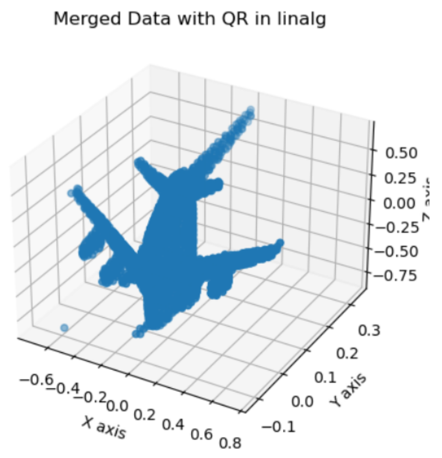


Fig. 3. Plane after merging (linalg.qr used)

Calculating the Absolute and Relative Error of the Rotation Matrix gave the following values:

Absolute Error:

$$\begin{bmatrix} 1.24454374 \times 10^{-5} & 3.86059631 \times 10^{-6} & 9.74945056 \times 10^{-6} \\ 1.19393144 \times 10^{-5} & 4.58098489 \times 10^{-6} & 6.39413385 \times 10^{-6} \\ 3.10215751 \times 10^{-6} & 5.75969855 \times 10^{-6} & 1.03468227 \times 10^{-5} \end{bmatrix}$$

Relative Error:

$$\begin{bmatrix} 2.23828294 \times 10^{-5} & 5.41621731 \times 10^{-6} & 2.28046822 \times 10^{-5} \\ 3.07164308 \times 10^{-5} & 1.97745778 \times 10^{-5} & 7.17017751 \times 10^{-6} \\ 4.22247377 \times 10^{-6} & 8.70017694 \times 10^{-6} & 6.97947518 \times 10^{-5} \end{bmatrix}$$

Then i calculated the average for each one:
Average Absolute Error: 7.575399579024595e-06
Average Relative Error: 2.122025750222485e-05

- While i got the following Errors for the Translation Vector:

Absolute Error:

$$\begin{bmatrix} 2.77155034 \times 10^{-6} & 1.79802410 \times 10^{-6} & 2.58128790 \times 10^{-6} \end{bmatrix}$$

Relative Error:

$$\begin{bmatrix} 5.92915203 \times 10^{-7} & 6.08747209 \times 10^{-7} & 7.48855339 \times 10^{-7} \end{bmatrix}$$

Average Absolute Error: $2.383620778238319 \times 10^{-6}$

Average Relative Error: $6.501725835088254 \times 10^{-7}$

2. Using the QR factorization I implemented with Gram-Schmidt Method. I found the following Rotation matrix:

$$\begin{bmatrix} 0.43423581 & 0.57196272 & -0.47111121 \\ 0.46790279 & 0.10816481 & 0.87442345 \\ 0.92750197 & -0.63044794 & -0.09684313 \end{bmatrix}$$

And the Translation Vector:

$$\begin{bmatrix} 4.67592364 & 2.94462581 & 3.43880002 \end{bmatrix}$$

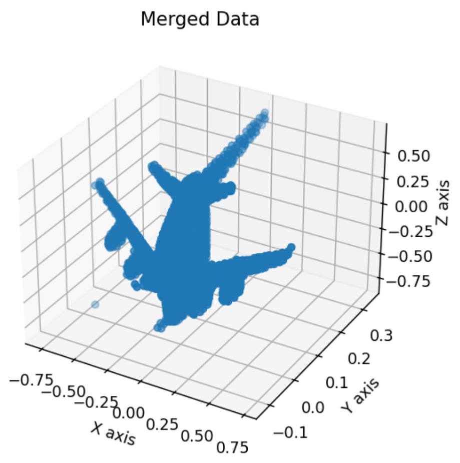


Fig. 4. Plane after merging (my qr implementation)

The errors of Rotation Matrix are as follows:

$$\text{Absolute Error: } \begin{bmatrix} 0.12177784 & 0.14082575 & 0.04358178 \\ 0.07919614 & 0.12350009 & 0.01733801 \\ 0.19282113 & 0.03156715 & 0.05141364 \end{bmatrix}$$

$$\text{Relative Error: } \begin{bmatrix} 0.28044173 & 0.24621491 & 0.09250848 \\ 0.16925767 & 1.14177699 & 0.01982794 \\ 0.20789296 & 0.05007099 & 0.53089612 \end{bmatrix}$$

Average Absolute Error: 0.08911350329080951
Average Relative Error: 0.3043208641106027

- While for the translation vector i got these results:

Absolute Error: $\begin{bmatrix} 0.00148001 & 0.00901894 & 0.00817543 \end{bmatrix}$
Relative Error: $\begin{bmatrix} 0.00031652 & 0.00306285 & 0.00237741 \end{bmatrix}$
Average Absolute Error: 0.0062247954121336475
Average Relative Error: 0.0019189248726910093

B. Cup Example

The Rotation Matrix found is:

$$\begin{bmatrix} -0.10160929 & -0.49427978 & -0.86334411 \\ -0.00591879 & 0.8681207 & -0.49631786 \\ 0.99480677 & -0.04532055 & -0.09113467 \end{bmatrix}$$

And Translation vector:

$$\begin{bmatrix} 1.51082729 & 1.89294346 & 1.89629477 \end{bmatrix}$$

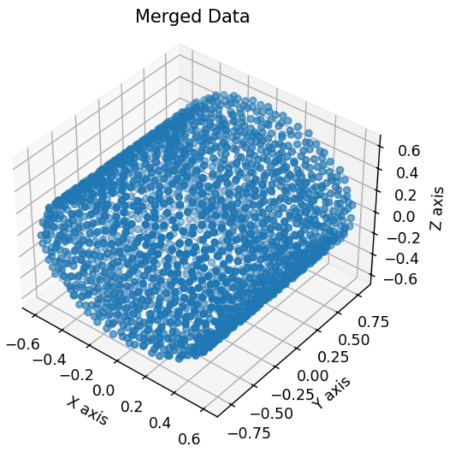


Fig. 5. Cup after merging

C. Chair Example

The Rotation Matrix is:

$$\begin{bmatrix} 0.12801764 & -0.14736851 & -0.98076195 \\ -0.45352553 & -0.88814468 & 0.07425379 \\ -0.88200118 & 0.43529479 & -0.18053357 \end{bmatrix}$$

And Translation vector:

$$\begin{bmatrix} 1.43531892 & 1.42036429 & 1.33033964 \end{bmatrix}$$

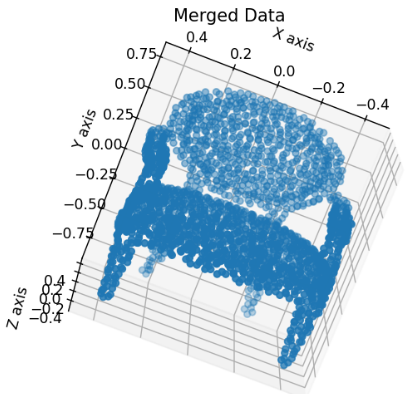


Fig. 6. Chair after merging

D. Bottle Example

The Rotation Matrix is:

$$\begin{bmatrix} -0.7468572 & -0.38868106 & 0.5395659 \\ -0.56646154 & -0.05314962 & -0.82237245 \\ 0.34831832 & -0.91983811 & -0.1804777 \end{bmatrix}$$

And Translation vector:

$$\begin{bmatrix} 1.53086172 & 1.23273344 & 1.01140052 \end{bmatrix}$$

E. Discussion

As it can be seen in the results part, The Algorithm worked as expected with small Error. As i have explained in the Methods part, The QR factorization implemented with Gram-Schmidt worked only for the plane points set, For this point set we can observe that the error of my implemented function is higher than the one of numpy library as expected but the overall error was tolerable. For the other point sets, using my implemented QR factorization gave ve nan as results since a division by 0 was encountered, but this was due to the method and roundoff errors so very small numbers were treated as 0. That is why i used the implemented QR factorization in numpy library but of course i used the QR Method to find the eigenvalues and vectors and my own SVD implementation. The results had small errors according to the plots found after merging the data sets.

F. Conclusion

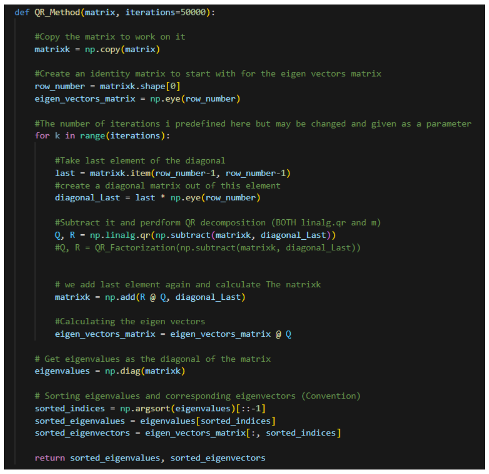
In this project, I learned different methods to implement SVD decomposition of a matrix, i implemented different methods and i chose the one that gave me smaller errors, QR Method. After that i implemented the Kabsch-Umeyama Algorithm and i was able to observe the results by plotting them for the 4 different data available. (ALL plots available in sub-folders and Notebook)

References

[1] Jim Lawrence, Javier Bernal, and Christoph Witzgall. A purely algebraic justification of the kabsch-umeyama algorithm. Journal of Research of the National Institute of Standards and Technology, 124, October 2019.

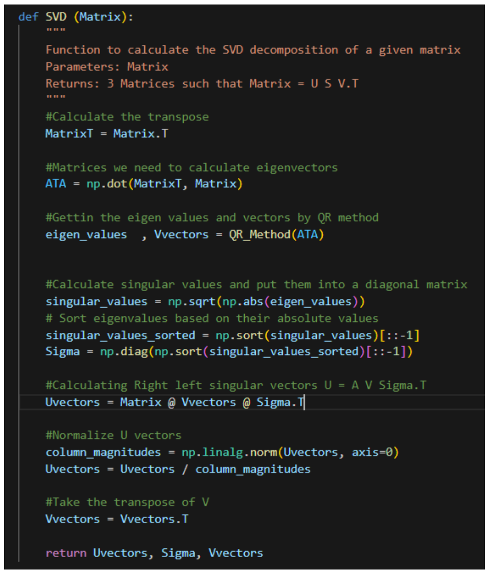
Appendix

All the SVD implementation is available in the Notebook uploaded in the project folder



```python
def QR_Method(matrix, iterations=50000):

    #Copy the matrix to work on it
    matrixk = np.copy(matrix)

    #Create an identity matrix to start with for the eigen vectors matrix
    row_number = matrixk.shape[0]
    eigen_vectors_matrix = np.eye(row_number)

    #The number of iterations i predefined here but may be changed and given as a parameter
    for k in range(iterations):

        #Take last element of the diagonal
        last = matrixk.item(row_number-1, row_number-1)
        #create a diagonal matrix out of this element
        diagonal_Last = last * np.eye(row_number)

        #Subtract it and perdform QR decomposition (BOTH linalg.qr and m)
        Q, R = np.linalg.qr(np.subtract(matrixk, diagonal_Last))
        #Q, R = QR_Factorization(np.subtract(matrixk, diagonal_Last))


        # we add last element again and calculate The natrixk
        matrixk = np.add(R @ Q, diagonal_Last)

        #Calculating the eigen vectors
        eigen_vectors_matrix = eigen_vectors_matrix @ Q

    # Get eigenvalues as the diagonal of the matrix
    eigenvalues = np.diag(matrixk)

    # Sorting eigenvalues and corresponding eigenvectors (Convention)
    sorted_indices = np.argsort(eigenvalues)[::-1]
    sorted_eigenvalues = eigenvalues[sorted_indices]
    sorted_eigenvectors = eigen_vectors_matrix[:, sorted_indices]

    return sorted_eigenvalues, sorted_eigenvectors
```

Fig. 7. QR Method Implementation



```python
def SVD (Matrix):
    """
    Function to calculate the SVD decomposition of a given matrix
    Parameters: Matrix
    Returns: 3 Matrices such that Matrix = U S V.T
    """
    #Calculate the transpose
    MatrixT = Matrix.T

    #Matrices we need to calculate eigenvectors
    ATA = np.dot(MatrixT, Matrix)

    #Gettin the eigen values and vectors by QR method
    eigen_values  , Vvectors = QR_Method(ATA)


    #Calculate singular values and put them into a diagonal matrix
    singular_values = np.sqrt(np.abs(eigen_values))
    # Sort eigenvalues based on their absolute values
    singular_values_sorted = np.sort(singular_values)[::-1]
    Sigma = np.diag(np.sort(singular_values_sorted)[::-1])

    #Calculating Right left singular vectors U = A V Sigma.T
    Uvectors = Matrix @ Vvectors @ Sigma.T

    #Normalize U vectors
    column_magnitudes = np.linalg.norm(Uvectors, axis=0)
    Uvectors = Uvectors / column_magnitudes

    #Take the transpose of V
    Vvectors = Vvectors.T

    return Uvectors, Sigma, Vvectors
```

Fig. 8. SVD Implementation