Introduction

This assignment is a continuation of the programming assignments that we've had in the past, regards to registration. In this homework specifically, we were tasked with handling deformable registration, which is a type of registration that may not necessarily be rigid.

Mathematical Approach

To summarize, we used approach #1 recommended to us in the programming assignment. The initial steps are similar to previous assignments.

1) We find a transformation F_{AB}, the transformation between the rigid body on the bone and the probe. This was done through point cloud to point cloud registration using the Quaternion method, namely

Quaternion method for R

Step 1: Compute

$$\mathbf{H}=\sum_i \begin{bmatrix} \tilde{a}_{i,x}\tilde{b}_{i,x} & \tilde{a}_{i,x}\tilde{b}_{i,y} & \tilde{a}_{i,x}\tilde{b}_{i,z} \\ \tilde{a}_{i,y}\tilde{b}_{i,x} & \tilde{a}_{i,y}\tilde{b}_{i,y} & \tilde{a}_{i,y}\tilde{b}_{i,z} \\ \tilde{a}_{i,z}\tilde{b}_{i,x} & \tilde{a}_{i,z}\tilde{b}_{i,y} & \tilde{a}_{i,z}\tilde{b}_{i,z} \end{bmatrix}$$

Step 2: Compute

$$\mathbf{G}=\begin{bmatrix} trace(\mathbf{H}) & \Delta^T \\ \Delta & \mathbf{H}+\mathbf{H}^T-trace(\mathbf{H})\mathbf{I} \end{bmatrix}$$

where $\Delta^T=\begin{bmatrix} \mathbf{H}_{2,3}-\mathbf{H}_{3,2} & \mathbf{H}_{3,1}-\mathbf{H}_{1,3} & \mathbf{H}_{1,2}-\mathbf{H}_{2,1} \end{bmatrix}$

Step 3: Compute eigen value decomposition of $\mathbf{G}$

$\text{diag}(\bar{\lambda})=\mathbf{Q}^T\mathbf{G}\mathbf{Q}$

Step 4: The eigenvector $\mathbf{Q}_k=\begin{bmatrix} q_0,q_1,q_2,q_3 \end{bmatrix}$ corresponding to the largest eigenvalue λ_k is a unit quaternion corresponding to the rotation.

Figure 1: Summary of the Quaternion Method for R

We can use the unit quaternion found in step 4 to explicitly solve for the R matrix using the following expression:

$$\mathbf{R}(\mathbf{q})=\begin{bmatrix} q_0^2+q_1^2-q_2^2-q_3^2 & 2(q_1q_2-q_0q_3) & 2(q_1q_3+q_0q_2) \\ 2(q_1q_2+q_0q_3) & q_0^2-q_1^2+q_2^2-q_3^2 & 2(q_2q_3-q_0q_1) \\ 2(q_1q_3-q_0q_2) & 2(q_2q_3+q_0q_1) & q_0^2-q_1^2-q_2^2+q_3^2 \end{bmatrix}$$

Figure 2: Closed form solution for a rotation matrix from a unit quaternion

Finally, the translation component is calculated by simply performing

$$\vec{\mathbf{p}} = \overline{\mathbf{b}} - \mathbf{R} \cdot \overline{\mathbf{a}}$$

Figure 3: Closed form solution for translation vector from R and the mean of two point clouds

2) We are now able to get an initial estimate of $s_k = F_{reg} d_k$, where we solve for F_{reg} via iterative measures. We have an initial guess of $F_{reg} = I$, and we use an iterative closest point approach to continually update the transformation.

```
# iterate to find F_reg, the initial estimate of the rigid transformation
F_reg = Frame(np.identity(3), np.zeros(3))
eps = 1
print('Calculating optimal F_reg')
iter = 0
while eps > 0.1 and iter < 100:
    F_old = F_reg
    s_k = F_reg.transformVector(d_k)
    c_k, tri_k = FindClosestPointMesh_BS(s_k, mesh.verts, mesh.tris)
    F_reg = Rigid_Transformation(d_k, c_k).getTransform()
    eps = F_reg.MSE(F_old)
    iter = iter + 1
```

Figure 4: Routine that finds the optimal F_reg by iterative measures

The maximum is either 100 iterations or if the MSE between the consecutive F_{reg} is low.

3) We begin the calculations for the lambda values, which represent the weights for the deformation modes.

We first begin with c_k, which is the point closest to the mesh from s_k. Then, we find its barycentric coordinates relative to the triangle it lies in. This was done by the teast squares method in homogeneous coordinates.

```python
def barycentric(verts, p):
    """
    Function to find the barycentric weights of p relative to the vertices of the triangle
    """

    # least squares approach
    a = verts[0][...,None]
    b = verts[1][...,None]
    c = verts[2][...,None]

    temp = np.hstack((a,b,c))
    X = np.vstack((temp, np.array([1,1,1])))

    y_temp = np.append(p, 1)
    y = y_temp[...,None]

    XTX = X.T @ X
    XTy = X.T @ y

    # there exists a closed form solution for the weights in linear regression
    weights = np.linalg.inv(XTX) @ XTy
```

Figure 5: Subroutine that finds the barycentric coordinates of p in triangle verts

We then find the corresponding q_k, which is the triangle with the deformations applied.

This is done by using the barycentric weights and adding them to a linear combination of the mode weights, ie

$$\vec{\mathbf{q}}_{m,k} = \zeta_k \vec{\mathbf{m}}_{m,s} + \xi_k \vec{\mathbf{m}}_{m,t} + \psi_k \vec{\mathbf{m}}_{m,u}$$

Figure 6: Equation for q_m,k given the barycentric weights and mode values

Then we can use least squares to find the lambdas explicitly. We solved for the lambdas in a $Ax = b$ format from the following equation:

$$\vec{\mathbf{c}}_k^{\,(t)} = \vec{\mathbf{q}}_{0,k} + \sum_{m=1}^{N_{modes}} \lambda_m^{(t)} \vec{\mathbf{q}}_{m,k}$$

Figure 7: Equation relating c_k, q_k, and lambdas

To summarize,

```python
def find_lambdas(c_k, tri_k, atlas_modes, s_k, mesh):
    """
    Function to find the lambdas, or the weights of the deformed barycentric centers
    input: the center point, triangle vertices, mode data, and the mesh data
    """
    # first find the weights
    verts_tri = []
    verts_tri.append(mesh.verts[tri_k[0]])
    verts_tri.append(mesh.verts[tri_k[1]])
    verts_tri.append(mesh.verts[tri_k[2]])
    bary_weights = barycentric(verts_tri, c_k)

    # now we need to find q
    q_k = []
    q_k.append(c_k[...,None])
    for i in range(atlas_modes.Nmodes):
        q_mk = find_qmk(atlas_modes, tri_k, bary_weights, i+1)
        q_mk = q_mk[...,None]
        q_k.append(q_mk)

    # finally, solve the least squares problem to find the lambdas
    y = s_k - c_k[...,None]
    X = np.zeros((3, atlas_modes.Nmodes))

    for i in range(atlas_modes.Nmodes):
        j = i + 1
        q_j = q_k[j]
        X[0, i] = q_j[0][0]
        X[1, i] = q_j[1][0]
        X[2, i] = q_j[2][0]

    # lambdas = (np.linalg.inv(XTX) @ XTy).T
    lambdas = np.linalg.lstsq(X, y, rcond=None)[0]
    lambdas = lambdas.reshape(atlas_modes.Nmodes)
    return lambdas.tolist(), q_k
```

```python
def find_qmk(atlas_modes, tri_k, bary_weights, m):
    """
    Function to find q_mk, the barycentric coordinate of a point inside a triangle
    with variation
    input: the mode data, triangle vertices, barycentric weights, and mode number
    output: a list of vertices of the deformed vertex coordinates
    """
    atlas_i = atlas_modes.data[m]

    m_ms = atlas_i[tri_k[0]]
    m_mt = atlas_i[tri_k[1]]
    m_mu = atlas_i[tri_k[2]]

    result = bary_weights[0] * m_ms + bary_weights[1] * m_mt + bary_weights[2] * m_mu
    return result
```

Figure 8: Subroutines to find lambda values and q_m,k from the mode values and barycentric weights

4) Then we used the lambdas to find the new vertices. This was incorporated to solving for the new c_k values in ICP for the next iteration. This allowed our algorithm to converge since we were getting closer to the actual values of the deformed vertices. We compared subsequent lambda values together to check this convergence. This was how we incorporated the deformation:

```
# need to add in the deformation from Lambdas and mdoes
for i in range(modes.Nmodes):
    modes_i = modes.data[i+1]
    ms = modes_i[tri[0]]
    mt = modes_i[tri[1]]
    mu = modes_i[tri[2]]
    sum_s = np.add(sum_s, Lambdas[i]*ms[...,None])
    sum_t = np.add(sum_t, Lambdas[i]*mt[...,None])
    sum_u = np.add(sum_u, Lambdas[i]*mu[...,None])

vert0 = np.add(vert0, sum_s.reshape(3))
vert1 = np.add(vert1, sum_t.reshape(3))
vert2 = np.add(vert2, sum_u.reshape(3))

A.append(vert0)
A.append(vert1)
A.append(vert2)
```

Figure 9: Subroutine to account for deformation in mesh

The overall protocol for steps 3-4 are as follows

```
print('Calculating Lambda values')
eps = 5
s_k = s_k[...,None]
if (total_iter == 0):
    Lambdas = 100*np.ones((6))
    iter_lambda = 0
    while iter_lambda < 100 and eps > .1:
        prev_Lambdas = Lambdas
        c_k_prev = c_k
        tri_k_prev = tri_k
        Lambdas, q_k = find_lambdas(c_k_prev, tri_k_prev, atlas_modes, s_k, mesh)
        c_k, tri_k = FindClosestPointMesh_Deformed(c_k_prev, mesh.verts, mesh.tris, Lambdas, atlas_modes)
        eps = np.linalg.norm(np.subtract(Lambdas, prev_Lambdas))
        iter_lambda = iter_lambda + 1
    print('Done in ' + str(iter_lambda) + ' iterations.')
else:
    c_k, tri_k = FindClosestPointMesh_Deformed(s_k, mesh.verts, mesh.tris, Lambdas, atlas_modes)
```

Figure 10: Overall routine to find lambdas that converge as well as new c_k values

Structure and File Organization

- OUTPUT
    - Contains all the output files from the deformable registration algorithm
- data
    - Contains all the data files to use in the algorithm
- src
    - pa5.py
    - utils.py
    - frameTransform.py
    - Octree.py
    - pointsetRegistration.py
    - reader.py
    - writer.py
    - test.py
    - config.ini
    - __init__.py

A short explanation of each script is given in the README.txt.

reader.py

This script gets the data from the relevant text and mesh files. The user has to type in the data file names i.e., LED for Body A, Body B, Mesh file and Sample Readings in the config.ini file while running the *pa5.py* script from the command line. The script has 4 classes, each for different input types. BodyLEDs class is used to read the readings of LEDs of Bodies(A and B). SurfaceMesh class is used to read and parse the data of input mesh files. SampleReadings class is used to read the data from the sample reading files. Finally, the AtlasModes read in the modes data.

writer.py

This script is used to write the output data generated i.e., *c, s* and $||c - s||$ as mentioned in the assignment in the output file, the path and name of which is to be mentioned while executing *pa5.py* as argument. The output text file name will be automatically generated, and saved to the OUTPUT directory.

frameTransform.py

A script which declares another class - Frame. Frame is an object to represent and handle rotation and translation in a better way. It has helper functions declared namely F_inv to

calculate inverse of a transformation matrix, ComposeTransform to get the resultant transformation matrix after multiplying two transformation matrices and finally, transformVector, to get the resultant vector after applying a transformation on it. An MSE() function was added so that it could calculate the sum of the L2 norm between all the elements in R and p components of the Frame.

pointsetRegistration.py
Script to implement 3d Point cloud to point cloud registration. It employs the Quaternion method(which in turn uses the eigenvalue decomposition method) to calculate the Rotation matrix. Using this Rotation Matrix, we calculate the position vector. The return type is a Frame type object as mentioned in the frameTransform.py with Rotation and Position as its attribute

Pa5.py - the main driver of the deformable registration algorithm. It reads in the data from the config file and the appropriate folder, finds rigid transformations, lambda values, and iterates over every frame. Finally, it writes all the data found into an output directory.

Octree.py
This is the script that contains the Octree class, which contains implementation for an Octree data structure. It has functionality for finding and adding nodes, or points into the data structure.

test.py
This script has simple test cases for all the functions like reader, writer, finding closest point on the triangle and finding closest point on the mesh. One can change the values in the test and check whether the algorithm is able to give out a meaningful output or not.

running the algorithm
In order to execute the algorithm, change the contents of config.ini to the desired filenames and settings and simply run:

$ python pa4.py

To run the test scripts, we can similarly run

$ python test.py

Validation
Unit tests were made in order to test various subroutines of our assignment. This includes reader, writer, finding lambdas, closest point to triangle, closest point to mesh, bounding

sphere, and barycentric coordinates. The *assert* command was used in this step so that when the entire unit test is completed, no errors will be thrown. Although the reader was used often in data creation so that these functions could be tested, there were times that artificial data had to be created. For example, checking the closest point to triangle and mesh involved creating our own data points for validation checking.

In addition, for checking algorithm speed we used the *time* module in python so that we could continually check its efficiency. For example, bounding spheres were useful to the speed up process, as shown in programming assignment 4.

Results

Our results were not ideal. Although lambda values did converge, the errors between s_k and c_k were considerably large, and we were not able to produce the correct results.

Example output for PA5-A-Output.txt (for PA5-A-Debug-SampleReadingsTest.txt) is shown on the following page:

```
150, -test-Output.txt 6
75.4370 -112.8269   -71.8239    12.7633 159.6027    12.5891
-8.29    11.20    47.18         -4.30    16.33    46.18         93.18
-13.30   7.86     -11.41        -16.55   -5.87    -1.18         35.23
-3.08    -12.04   0.47          -7.28    -11.27   1.02          22.21
3.13     25.34    12.52         -7.37    -4.91    26.64         60.10
21.71    3.79     54.32         2.70     18.73    27.56         76.43
-6.32    12.03    61.53         -6.78    -6.19    36.15         113.88
0.42     -6.74    58.18         -6.78    -6.19    36.15         109.82
23.69    14.33    43.27         3.06     18.86    27.14         56.98
13.48    13.82    -32.43        -7.37    -4.91    26.64         82.37
17.77    25.64    0.51          -7.37    -4.91    26.64         63.43
-8.89    0.51     47.31         -6.78    -6.19    36.15         96.55
17.71    1.30     -27.86        -7.37    -4.91    26.64         76.76
-17.66   15.48    18.21         -7.37    -4.91    26.64         67.47
12.48    -4.92    64.88         -6.78    -6.19    36.15         118.25
-16.47   5.13     12.35         -7.37    -4.91    26.64         60.71
-18.88   -29.34   -33.99        -38.76   -19.85   -32.92        31.76
31.01    -4.60    -25.97        -7.37    -4.91    26.64         85.57
0.45     23.62    10.31         -7.37    -4.91    26.64         58.05
-17.26   3.54     -1.09         -11.41   -9.56    0.24          31.40
22.00    0.52     42.54         -6.78    -6.19    36.15         89.66
-12.78   -28.01   -36.86        -27.94   -30.55   -23.96        31.29
-26.51   9.59     -21.94        -16.55   -5.87    -1.18         54.07
20.32    3.23     -31.80        -7.37    -4.91    26.64         83.10
-2.13    15.53    63.16         -6.78    -6.19    36.15         115.46
5.70     -7.35    52.08         -6.78    -6.19    36.15         101.15
-7.94    1.15     58.55         -6.78    -6.19    36.15         110.70
-9.95    13.71    11.99         -7.37    -4.91    26.64         56.60
-16.19   2.15     8.28          -7.37    -4.91    26.64         59.46
-14.99   9.64     -22.06        -17.58   -7.11    -2.37         45.00
-6.92    7.71     -25.49        -17.13   -6.56    -1.84         45.05
20.58    -12.35   -1.19         -7.37    -4.91    26.64         62.40
34.73    -6.19    -6.70         -7.37    -4.91    26.64         74.87
-32.61   -22.92   -14.29        -27.84   -26.58   -11.40        31.98
-2.80    -3.02    -33.08        -16.55   -5.87    -1.18         49.26
-17.36   3.99     -0.76         -11.41   -9.56    0.24          32.13
-4.77    -4.60    42.27         -6.78    -6.19    36.15         90.23
-5.18    -3.97    58.99         -6.78    -6.19    36.15         111.49
-28.16   -20.39   -9.61         -10.25   -20.39   -9.47         32.70
41.98    5.88     6.78          -7.37    -4.91    26.64         79.52
-3.77    -24.23   -22.50        -5.53    -14.51   -4.83         40.05
-13.71   9.07     -31.32        -16.55   -5.87    -1.18         54.64
-13.87   9.03     -19.76        -16.78   -6.14    -1.44         41.94
3.10     -17.03   -11.08        -17.96   -7.58    -2.82         31.82
-14.91   7.08     -37.42        -16.55   -5.87    -1.18         61.74
```

```
34.68    11.09    -14.83        -7.37    -4.91    26.64        78.18
-8.94    7.39     -18.02        -16.55   -5.87    -1.18        37.20
23.53    24.36    -11.00        -7.37    -4.91    26.64        71.48
11.28    8.89     -16.52        -7.37    -4.91    26.64        60.80
-8.95    -11.48   1.31          -8.55    -10.66   1.00         22.53
15.31    -11.66   -21.36        -7.37    -4.91    26.64        73.16
-32.66   -10.72   -9.27         -10.95   -11.95   0.89         47.70
37.84    -5.18    5.62          -7.37    -4.91    26.64        75.73
5.55     14.72    -11.14        -7.37    -4.91    26.64        56.71
-9.54    22.36    18.82         -7.37    -4.91    26.64        65.51
22.47    -7.34    18.05         -7.37    -4.91    26.64        63.75
16.42    -1.52    63.66         -6.78    -6.19    36.15        116.21
20.94    -9.91    3.61          -7.37    -4.91    26.64        59.94
-7.85    0.84     54.17         -6.78    -6.19    36.15        104.78
-12.09   10.73    10.76         -7.37    -4.91    26.64        56.79
-4.29    -11.94   0.05          -7.28    -11.27   1.02         21.42
-37.85   -13.88   -12.89        -27.84   -26.58   -11.40       41.26
22.37    5.07     47.40         2.70     18.73    27.56        65.87
3.02     23.30    26.05         4.20     19.28    25.80        41.17
-3.04    -26.54   -25.24        -17.25   -22.17   -5.66        39.67
21.32    -14.75   -4.16         -7.37    -4.91    26.64        66.06
-27.72   -6.73    -8.94         -10.95   -11.95   0.89         39.51
23.94    7.30     39.53         3.88     19.16    26.17        52.94
-4.19    2.03     -31.22        -16.55   -5.87    -1.18        48.40
26.00    20.07    -17.86        -7.37    -4.91    26.64        75.80
18.67    19.01    47.69         2.70     18.73    27.56        62.78
-8.52    4.86     63.82         -6.78    -6.19    36.15        117.86
23.54    9.19     39.76         3.89     19.17    26.16        52.19
25.51    -4.82    -29.57        -7.37    -4.91    26.64        85.23
38.49    1.59     -16.27        -7.37    -4.91    26.64        83.22
22.06    13.03    -30.77        -7.37    -4.91    26.64        83.96
-29.99   -8.05    -8.01         -10.95   -11.95   0.89         42.99
-34.11   -27.10   -25.32        -38.41   -19.02   -27.39       26.46
-2.60    17.74    52.26         -6.78    -6.19    36.15        101.03
8.98     25.77    2.79          -7.37    -4.91    26.64        59.52
29.13    12.13    16.48         4.37     19.34    25.60        35.40
15.41    -7.89    16.84         -7.37    -4.91    26.64        58.43
36.34    -4.40    -3.15         -7.37    -4.91    26.64        74.75
-5.25    -5.59    -43.95        -9.13    -12.03   -1.20        64.89
-29.86   -3.58    -41.12        -25.52   -28.46   -13.45       51.55
9.03     10.31    -11.95        -7.37    -4.91    26.64        56.07
-3.23    4.15     -31.79        -16.55   -5.87    -1.18        50.87
22.89    16.74    51.06         2.70     18.73    27.56        68.56
26.51    -2.89    -30.71        -7.37    -4.91    26.64        86.78
```

-5.42	13.44	55.61	-6.78	-6.19	36.15	105.51
-7.82	12.63	44.93	-6.78	-6.19	36.15	92.66
14.11	25.12	6.71	11.00	22.38	6.26	31.08
-13.07	-9.84	-46.29	-24.15	-29.56	-14.66	52.77
-33.49	5.04	-20.98	-10.17	-11.99	-0.01	57.55
37.66	1.74	14.28	-7.37	-4.91	26.64	75.48
-20.31	-30.13	-31.48	-38.70	-19.70	-31.88	29.13
10.34	-9.28	-22.72	-7.37	-4.91	26.64	71.52
2.71	20.07	46.12	-6.78	-6.19	36.15	92.69
-38.37	-15.58	-33.59	-38.76	-19.85	-32.92	38.01
12.65	-16.25	-11.87	-7.37	-4.91	26.64	67.11
-7.62	-11.11	9.64	-7.37	-4.91	26.64	58.72
-11.95	-1.79	-42.20	-9.13	-12.03	-1.20	62.99
-11.34	8.48	-10.56	-7.37	-4.91	26.64	60.70
23.84	-13.66	-4.80	-7.37	-4.91	26.64	67.41
-6.97	9.75	59.68	-6.78	-6.19	36.15	111.33
12.45	1.89	66.02	-6.78	-6.19	36.15	118.32
28.95	-9.32	13.77	-7.37	-4.91	26.64	68.90
-35.48	4.50	-25.20	-9.13	-12.03	-1.20	62.60
8.14	23.03	16.60	6.51	19.94	21.88	27.49
8.61	23.79	17.25	6.51	19.94	21.88	27.75
-27.29	-28.87	-22.70	-38.41	-19.02	-27.39	25.78
-36.53	3.24	-27.89	-11.89	-17.33	-5.34	59.58
-2.02	-5.15	64.30	-6.78	-6.19	36.15	118.41
13.99	-12.35	-20.31	-7.37	-4.91	26.64	71.99
-10.35	8.24	-24.43	-17.54	-7.06	-2.32	44.45
25.37	-12.43	-27.37	-7.37	-4.91	26.64	86.15
-7.78	-10.90	2.39	-8.09	-10.59	1.79	23.39
18.09	18.25	61.99	2.70	18.73	27.56	85.02
-30.71	-8.31	-8.35	-10.95	-11.95	0.89	44.17
33.12	11.11	-18.00	-7.37	-4.91	26.64	79.01
23.29	23.81	-3.49	-7.37	-4.91	26.64	66.91
-3.92	-16.55	-39.48	-9.84	-20.19	-9.95	50.60
-7.03	-11.18	3.76	-8.21	-10.48	1.90	24.96
33.12	-9.77	-17.40	-7.37	-4.91	26.64	81.73
21.52	17.66	20.02	21.39	21.23	20.22	6.20
-4.35	-21.14	-36.32	-24.15	-29.56	-14.66	43.80
-4.62	6.31	-23.11	-16.55	-5.87	-1.18	41.29
-19.83	-23.69	-9.69	-10.25	-20.39	-9.47	26.61
23.55	15.52	37.57	4.37	19.34	25.60	46.99
30.11	8.41	-26.35	-7.37	-4.91	26.64	83.83
-39.60	-16.88	-33.16	-38.75	-19.83	-32.76	37.24
21.58	14.60	53.59	2.70	18.73	27.56	72.17
39.05	6.05	-8.02	-7.37	-4.91	26.64	78.59

19.02	19.32	51.62	2.70	18.73	27.56	68.83
22.09	-15.65	-21.84	-7.37	-4.91	26.64	80.23
-20.15	15.59	23.30	-7.37	-4.91	26.64	73.49
11.41	-8.50	8.95	-7.37	-4.91	26.64	53.58
26.05	22.35	2.71	8.29	19.00	-8.90	56.60
2.29	18.93	57.70	-6.78	-6.19	36.15	107.68
39.94	2.21	-14.70	-7.37	-4.91	26.64	83.80
-23.69	-30.23	-35.18	-38.76	-19.85	-32.92	27.70
20.91	-0.86	47.24	-6.78	-6.19	36.15	95.20
-7.35	-13.87	-43.90	-24.15	-29.56	-14.66	51.30
-38.01	-3.13	-16.49	-20.91	-20.36	-5.54	49.31
6.23	-12.43	-21.75	-18.04	-7.67	-2.91	39.72
30.60	19.88	-2.06	-7.37	-4.91	26.64	70.58
12.33	-17.43	-8.44	-7.37	-4.91	26.64	65.83
24.86	-14.03	-7.91	-7.37	-4.91	26.64	70.08
-8.66	15.25	13.10	-7.37	-4.91	26.64	56.88
-8.19	21.11	16.85	-7.37	-4.91	26.64	62.44
-9.54	-13.70	-7.07	-18.04	-7.67	-2.91	20.73

Figure 11: Example output for PA5-A dataset

I believe that there was a fundamental misunderstanding in the overall algorithm in its entirety. In general, keeping track of so many variables such as c_k, s_k, q_{mk} and many more was difficult. In addition, there were inconsistencies during the implementation process in regard to the data structures (for example sometimes a 3D point would be represented as a row vector or sometimes a column vector, which made it difficult to carry out certain processes).

Distribution of Tasks

In regard to the *new* algorithms created for PA5, these were the tasks that Unnat and I did:

Unnat Antani - reader.py, writer.py, barycentric(), FindClosestPointMesh_Deformed()
John Han - pa5.py, find_lambdas(), find_qmk(), frameTransform()