


# Programming Assignments 3 and 4 – 601.455/655 Fall 2024

Score Sheet (hand in with report) Also, PLEASE INDICATE WHETHER YOU ARE IN 601.455 or 601.655

(one in each section is OK)

Name 1	Aryavrat Gupta
Email	agupt110@jhu.edu
Other contact information (optional)	
Name 2	Benjamin Miller
Email	bmill119@jh.edu
Other contact information (optional)	
Signature (required)	<p>I (we) have followed the rules in completing this assignment</p> <p style="text-align: center;">  </p>

Grade Factor		
Program (40)		
Design and overall program structure	20	
Reusability and modularity	10	
Clarity of documentation and programming	10	
Results (20)		
Correctness and completeness	20	
Report (40)		
Description of formulation and algorithmic approach	15	
Overview of program	10	
Discussion of validation approach	5	
Discussion of results	10	
TOTAL	100	

# CIS: Programming Assignment 3

Aryavrat Gupta, Benjamin Miller

11/14/2024

## 1 Introduction

The overall goal of this assignment is to implement a simplified version of the Iterative Closest Point (ICP) registration algorithm to compute the position of pointer A in CT coordinates relative to a 3D surface mesh (representing a bone). We are provided with body definition files for two rigid bodies, A and B, as well as sample readings of LED markers for each body relative to an optical tracker. Body A is a pointer that interacts with the bone, while body B is fixed to the bone. The position of pointer A's tip on the bone surface in CT coordinates must be determined using the ICP algorithm and surface mesh geometry. This report outlines the mathematical foundations, algorithmic implementation, program structure, validation steps, and analysis of the obtained results.

## 2 Mathematical Approach

### 2.1 Frame Transformation

To calculate the position of the pointer tip, we used transformations between different coordinate systems. Frame transformations were represented as:

$$F_{ab} = [R_{ab}, p_{ab}]$$

where  $R_{ab}$  is the rotation matrix and  $p_{ab}$  is a translation vector.

The inverse frame transformation, which is also needed to move between coordinate systems, is computed as:

$$F_{ab}^{-1} = [R_{ab}^{-1}, -R_{ab}^{-1}p_{ab}]$$

### 2.2 Point Set to Point Set Registration

To compute the optimal rotation matrix and translation vector to align 3D point sets, we first calculated the average value of both point sets. Let source and target be the two point sets. We compute their averages as:

$$\text{average} = \frac{1}{N} \sum_{i=1}^N p_i$$

The points were then centered by subtracting the respective average. To solve for the optimal rotation matrix, we implement Arun's method. We compute the covariance matrix  $H$  of the centered point sets, given by:

$$H = \sum_{i=1}^N (p_{i,\text{source}} - \bar{p}_{\text{source}})(p_{i,\text{target}} - \bar{p}_{\text{target}})^T$$

We then apply Singular Value Decomposition (SVD) to the covariance matrix  $H$ :

$$H = U\Sigma V^T$$

From the matrices  $U$  and  $V$ , we find the optimal rotation matrix  $R$  using:

$$R = VU^T$$

We ensure that the rotation is correct by checking the determinant of  $R$ ; if  $\det(R) < 0$ , we modify  $V$  for a valid rotation.

The translation vector  $p$  can be computed as:

$$p = \bar{p}_{\text{target}} - R\bar{p}_{\text{source}}$$

We also compute the RMSE between the target and transformed points for algorithm verification:

$$\text{RMSE} = \sqrt{\frac{1}{N} \sum_{i=1}^N |p_{i,\text{target}} - Rp_{i,\text{source}} - p|^2}$$

### 2.3 Finding Pointer $A_{\text{Tip}}$ in Body B Coordinates

Given the position of the pointer tip  $A_{\text{Tip}}$  in local coordinates, for each sample frame  $k$ , we calculate its position in Body B coordinates using the given frame transformation:

$$p_{B_{\text{Tip}},k} = F_{B,k}^{-1} F_{A,k} p_{A_{\text{Tip}}}$$

where  $F_{A,k}$  and  $F_{B,k}$  are the frame transformations from the optical tracker to the local coordinates of pointers A and B, respectively.

### 2.4 Finding the Closest Point on a Triangle

To determine the closest point on a triangle to a given 3D point  $\mathbf{a}$ , where the triangle has vertices  $\mathbf{p}, \mathbf{q}, \mathbf{r}$ , we used the geometric approach described in the slides. The process consists of two main cases: when the point lies inside the triangle and when it lies outside, which requires projection onto the triangle's edges.

We first calculate the following dot products, which we use to determine the position of  $\mathbf{a}$  relative to the triangle:

$$\begin{aligned} d_1 &= (\mathbf{q} - \mathbf{p}) \cdot (\mathbf{a} - \mathbf{p}), \\ d_2 &= (\mathbf{r} - \mathbf{p}) \cdot (\mathbf{a} - \mathbf{p}), \\ d_3 &= (\mathbf{q} - \mathbf{p}) \cdot (\mathbf{q} - \mathbf{p}), \\ d_4 &= (\mathbf{r} - \mathbf{p}) \cdot (\mathbf{r} - \mathbf{p}), \\ d_5 &= (\mathbf{q} - \mathbf{p}) \cdot (\mathbf{r} - \mathbf{p}). \end{aligned}$$

Using these dot products, we compute the barycentric coordinates  $u, v, w$  of point  $\mathbf{a}$  relative to the triangle. The denominator of these coordinates is:

$$\text{denom} = d_3 \cdot d_4 - d_5^2.$$

The coordinates  $v$  and  $w$  are then calculated as:

$$\begin{aligned} v &= \frac{d_4 \cdot d_1 - d_5 \cdot d_2}{\text{denom}}, \\ w &= \frac{d_3 \cdot d_2 - d_5 \cdot d_1}{\text{denom}}, \\ u &= 1 - v - w. \end{aligned}$$

If the barycentric coordinates satisfy the conditions  $u \geq 0$ ,  $v \geq 0$ , and  $w \geq 0$ , then the point  $\mathbf{a}$  lies inside the triangle. In this case, the closest point on the triangle is the weighted sum of the triangle's vertices:

$$\text{closest\_point} = u \cdot \mathbf{p} + v \cdot \mathbf{q} + w \cdot \mathbf{r}.$$

If the point lies outside the triangle, the closest point must be on one of its edges. We compute the closest point on each of the triangle's three edges  $\mathbf{pq}, \mathbf{pr}, \mathbf{qr}$  by projecting  $\mathbf{a}$  onto each segment. The projection of a point  $\mathbf{a}$  onto a segment with endpoints **start** and **end** is:

$$\text{closest\_point\_segment} = \text{start} + t \cdot (\text{end} - \text{start}),$$

where the scalar  $t$  is computed as:

$$t = \frac{(\mathbf{a} - \text{start}) \cdot (\text{end} - \text{start})}{(\text{end} - \text{start}) \cdot (\text{end} - \text{start})}.$$

We constrain  $t$  to the interval  $[0, 1]$  to ensure that the projection lies on the segment. The three potential closest points are:

$$\text{closest\_pq} = \text{closest\_point\_segment}(\mathbf{p}, \mathbf{q}, \mathbf{a}),$$

$$\text{closest\_pr} = \text{closest\_point\_segment}(\mathbf{p}, \mathbf{r}, \mathbf{a}),$$

$$\text{closest\_qr} = \text{closest\_point\_segment}(\mathbf{q}, \mathbf{r}, \mathbf{a}).$$

Finally, we compute the distances from  $\mathbf{a}$  to each of the potential closest points. The distance between two points  $\mathbf{a}$  and  $\mathbf{b}$  is:

$$\text{dist}(\mathbf{a}, \mathbf{b}) = \|\mathbf{a} - \mathbf{b}\|.$$

The closest point is the one that minimizes the distance:

$$\text{closest\_point} = \begin{cases} \text{closest\_pq} & \text{if } \text{dist}(\mathbf{a}, \text{closest\_pq}) < \text{dist}(\mathbf{a}, \text{closest\_pr}) \text{ and } \text{dist}(\mathbf{a}, \text{closest\_pq}) < \text{dist}(\mathbf{a}, \text{closest\_qr}), \\ \text{closest\_pr} & \text{if } \text{dist}(\mathbf{a}, \text{closest\_pr}) < \text{dist}(\mathbf{a}, \text{closest\_qr}), \\ \text{closest\_qr} & \text{otherwise.} \end{cases}$$

## 2.5 Linear Search through Triangles

For the linear search through triangles, the process involves passing a point and the current triangle into the closest point on triangle method. After computing the closest point on the triangle, the distance between the query point and the closest point is calculated using the following:

$$\text{distance} = \|\mathbf{pt} - \text{closest\_point}\|_2.$$

The triangle with the closest point is selected by comparing these distances across all triangles (Brute Force).

## 2.6 K-D Tree Nearest-Neighbor Search

We can optimize the search for the closest triangle using a K-D Tree. The K-D Tree is built from the centroids of the triangles, and the nearest neighbor search is performed to find the closest triangle to the query point. The implementation is described in 3.6.

The centroid  $C$  of a triangle with vertices  $\mathbf{v}_1(x_1, y_1, z_1)$ ,  $\mathbf{v}_2(x_2, y_2, z_2)$ ,  $\mathbf{v}_3(x_3, y_3, z_3)$  is computed as:

$$\mathbf{C}_i = \frac{1}{3} ((x_1, y_1, z_1) + (x_2, y_2, z_2) + (x_3, y_3, z_3)).$$

Using the K-D Tree, we can find the nearest centroid to the current point, and the corresponding triangle can be retrieved.

## 2.7 Octree Search

An Octree can also be used to further optimize the search by partitioning the space into cubes. Each cube contains the triangles whose centroids fall within it. The implementation is described in 3.8.

## 3 Algorithm Overview

### 3.1 Frame Transformations

Our package utilizes the NumPy Python library to handle 3D points and to perform operations such as rotations and translations<sup>1</sup>. Using NumPy, 3D points can be represented as arrays, and coordinate sets can be stored as arrays of size  $n \times 3$ , where  $n$  is the number of depth calibration points. Additionally, transformation matrices are stored as  $4 \times 4$  arrays in homogeneous coordinates and are of the form:

$$T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix}$$

where  $R$  is a  $3 \times 3$  array containing the rotation matrix, and the translation vector  $p$  is  $3 \times 1$ . Therefore, to apply a rotation matrix to a point set, the point set is expanded to homogeneous coordinates of size  $4 \times 4$ . The same applies for an inverse frame transformation of the form:

$$T^{-1} = \begin{bmatrix} R^{-1} & -R^{-1}p \\ 0 & 1 \end{bmatrix}$$

Using NumPy arrays allows for efficient mathematical operations, such as vector algebra, matrix dot products, and Singular Value Decomposition (SVD).

### 3.2 Point Set to Point Set Registration

The following pseudocode outlines the registration process using Arun’s method for finding the optimal rigid transformation between two sets of corresponding 3D points.

```
function register_point_sets(source_points, target_points):  
  
    calculate average position of source_points  
    calculate average position of target_points  
  
    center source_points by subtracting their average  
    center target_points by subtracting their average  
  
    calculate covariance matrix from centered point sets  
  
    perform SVD on covariance matrix  
  
    calculate rotation matrix from SVD results  
  
    if determinant < 1:  
        adjust rotation matrix  
  
    calculate translation vector using rotation matrix and averages  
  
    apply rotation and translation to source_points  
  
    calculate RMSE between transformed source_points and target_points  
  
    return rotation matrix and translation vector
```

### 3.3 Finding $A_{\text{Tip}}$ in Body B Coordinates

The following pseudocode calculates the position of the pointer tip  $d_{\text{Tip}}$ :

```
function calculate_pointer_tip_position(F_A_k, F_B_k, Atip):  
  
    # Calculate the transformed tip position
```

---

<sup>1</sup><https://numpy.org/>

```

d = inv(F_B_k) * F_A_k * append(Atip, 1)

# Extract position of the tip
d_coords = d[:3]

return d_coords

```

### 3.4 Closest Point on Triangle

```

# Function to find the closest point on a segment
closest_point_on_segment(seg_start, seg_end, point):
    seg_vec = seg_end - seg_start
    t = dot(point - seg_start, seg_vec) / dot(seg_vec, seg_vec)
    t = clip(t, 0, 1)
    return seg_start + t * seg_vec

```

The closest point on each edge is found using the above function. Then, the distances between the point and the closest points on the edges are calculated. The closest point is the one with the smallest distance.

```

# Check the closest point on each triangle edge
closest_pq = closest_point_on_segment(p, q, a)
closest_pr = closest_point_on_segment(p, r, a)
closest_qr = closest_point_on_segment(q, r, a)

# Compare distances
dist_pq = norm(a - closest_pq)
dist_pr = norm(a - closest_pr)
dist_qr = norm(a - closest_qr)

# Return the closest point
if dist_pq < dist_pr and dist_pq < dist_qr:
    return closest_pq
elif dist_pr < dist_qr:
    return closest_pr
else:
    return closest_qr

```

### 3.5 Brute Force: Linear Search

For Linear Search, we calculate the closest point for each triangle in the mesh using the following pseudocode:

```

# Function to find the closest point on the mesh
closest_point_on_mesh(point, vertices, triangles):
    min_dist = inf
    closest_point = None

    # Iterate over all triangles in the mesh
    for each triangle in triangles:
        v1, v2, v3 = vertices[triangle[0]], vertices[triangle[1]], vertices[triangle[2]]

        # Calculate closest point on the triangle
        tri_point = closest_point_on_triangle(point, v1, v2, v3)

        # Calculate distance to the query point
        dist = norm(tri_point - point)

        # Update if this is the closest point so far

```

```

    if dist < min_dist:
        min_dist = dist
        closest_point = tri_point

return closest_point

```

### 3.6 K-D Trees Implementation

To optimize the search, we used a k-dimensional tree (K-D Tree) for nearest-neighbor searches on a 3D triangular mesh. We used the `KDTree` from the `scipy.spatial` library<sup>2</sup>, which enables nearest-neighbor searches by organizing the triangle centroids in a way that minimizes the search space. The following data structures were first implemented:

**Vertex:** A 3D point with coordinates  $(x, y, z)$ , representing a vertex in the mesh.

**Triangle:** Three vertices and three neighboring triangles.

$$\text{Triangle} = \{(v_1, v_2, v_3), (n_1, n_2, n_3)\}$$

where  $v_1, v_2, v_3$  are the indices of the triangle's vertices and  $n_1, n_2, n_3$  are the indices of neighboring triangles.

We constructed the K-D Tree from the centroids of the triangles.

### 3.7 K-D Tree: Nearest-Neighbor Search

To find the closest triangle to a point, we performed a K-D Tree search for the nearest centroid.

```

function closest_point_on_mesh(query_point, kdtree, triangle_data):

    # Find the nearest centroid C to the point P using the K-D Tree
    C = kdtree.query(query_point)

    # Retrieve the corresponding triangle T
    T = triangle_data[C]

    # Compute the closest point Q on triangle T to point P
    Q = None

    for each edge in T.edges:

        projected_point = project_point_on_edge(query_point, edge)

        dist = distance(query_point, projected_point)

        # Update closest point Q if needed
        if Q is None or dist < distance(query_point, Q):
            Q = projected_point

    # If the projection lies inside the triangle, use the projected point
    if is_point_inside_triangle(T, Q):
        return Q

    # If the projection lies outside the triangle, find the closest point to triangle's edge
    closest_edge_point = find_closest_edge_point(T, Q)

    return closest_edge_point

```

---

<sup>2</sup><https://docs.scipy.org/doc/scipy/reference/spatial.html>

### 3.8 Octrees Implementation

Next, we implemented an Octree data structure to use partitioned 3D space for spatial queries. The Octree is constructed by recursively dividing the 3D space into eight smaller cubes. Each node in the tree holds a set of triangles, and the space is divided into eight child nodes when necessary. Building an Octree follows this procedure:

- Calculate the center and size of the root node
- Recursively divide the space into octants and assign triangles to the appropriate nodes
- Insert each triangle into the tree. Divide when needed.
- For a given point, traverse the tree to find the closest point on the mesh by checking leaf nodes
- Return the closest point found on the mesh

### 3.9 Octree: Nearest-Neighbor Search

The following pseudocode outlines the Octree nearest-neighbor search:

```
function find_closest_point(query_point, tree):

    closest_point = None
    min_distance = infinity

    node_stack = [tree.root]

    while node_stack is not empty:

        current_node = node_stack.pop()

        if current_node.is_leaf():
            current_node.triangles:

                vertices = get_vertices(triangle)

                # Find the closest point on the triangle to the point P
                closest_point_on_triangle = closest_point_on_triangle(query_point, vertices)

                # Calculate the distance
                distance_to_point = distance(query_point, closest_point_on_triangle)

                if distance_to_point < min_distance:
                    # Update the closest point and minimum distance
                    closest_point = closest_point_on_triangle
                    min_distance = distance_to_point

            # Else, add the children of the node to the stack for further exploration
            else:
                node_stack.extend(current_node.children)

    return closest_point
```

## 4 Program Structure

### 4.1 Overview

Input Data Handling (readData):

- **parseMesh.py:** Parses mesh files containing vertices and triangles.



- **parseData.py:** Reads body marker coordinates and sample readings.

#### Utility Files (util):

- **geometry.py:** Contains Vertex and Triangle classes.
- **calculateTransformation.py:** Computes frame transformations between Body A and Body B using point set registration.
- **pointSetRegistration.py:** Implements Arun's method for registering point sets.

#### Data Structure Building (build):

- **buildKDTree.py:** Builds a KD-tree structure.
- **buildOctree.py:** Builds an octree structure.

#### Main, Search Optimization, and Generating Output:

- **main.py:** Reads input files, calculates transformations, finds the closest points on a mesh, and generates the output.
- **findClosestPoint.py:** Computes the closest point on a triangle.
- **findPointOnMesh.py:** Linear Search to finds closest point.
- **findPointOnMeshKDTree.py:** KD-tree Search.
- **findPointOnMeshOctree.py:** Octree Search.
- **generateOutput.py:** Writes transformed tip positions and closest points on the mesh to an output file.

#### Error and Performance Metrics (errorCalculation):

- **computeMean.py:** Computes the mean distance from the last column of output files.
- **compareOutput.py:** Computes the Mean Squared Error between the results and expected outputs.

Please look at the README for the file tree expressed as relative paths from the root.

## 4.2 File Descriptions

### parseData.py

The **readData.py** file handles data input operations, reading the body and sample files:

- **read\_body():** Reads marker coordinates for the body files, such as Problem3-BodyA.txt. Returns a tuple including the number of markers, a list of coordinates, and the coordinates of the tip.
- **read\_sample():** Reads sample readings from sample files, such as PA3-A-Debug-SampleReadingsTest.txt. Returns a tuple including the number of markers A, B, and dummy, the number of frames, and a dictionary of frame data containing lists of measured coordinates.

### parseMesh.py

The **parseMesh.py** file reads and processes the mesh data:

- **parse\_mesh\_file():** Reads vertex and triangle data from a mesh file, such as Problem3MeshFile.sur.

### geometry.py

The **geometry.py** file contains classes related to 3D geometry:

- **Vertex Class:** Represents a 3D point with x, y, and z coordinates.
- **Triangle Class:** Represents a triangle in a 3D mesh, defined by three vertices and their corresponding neighbor indices.

## pointSetRegistration.py

This is the same **pointSetRegistration.py** file from PA1/2. This file implements Arun's method for aligning two sets of 3D points:

- **point\_set\_registration()**: Uses Arun's method to compute the optimal rotation matrix and translation vector that best align a source point set to a target point set. SVD is performed on the covariance matrix of the point sets. Also computes the RMSE.

## calculateTransformation.py

The **calculateTransformation.py** file computes the frame transformations between the Body A and Body B coordinate systems:

- Reads the marker data for both bodies (A and B) and the sample readings, using **read\_body()** and **read\_sample()** from **readInput.py**. For each frame in the sample, it calculates the transformations using **point\_set\_registration()** from **pointSetRegistration.py**.
- **create\_homogenous\_matrix()**: Accepts the rotation matrix and translation vector obtained from the registration method and creates a 4x4 homogeneous transformation matrix.

## buildKDTree.py

The **buildKDTree.py** file is responsible for building a KD-tree structure from 3D mesh data via **build\_kd\_tree()**:

- Precomputes centroids of the triangles in the mesh by averaging their vertices.
- Builds a KD-tree using the triangle centroids.
- Stores the triangle vertex data.

## buildOctree.py

The **buildOctree.py** file is responsible for constructing an octree structure from 3D mesh data.

- **OctreeNode Class**: Represents a node in the octree, which divides 3D space into 8 components. Contains center, size, children (list), and triangles (list).
- **is\_leaf()**: Checks if the node is a leaf.
- **subdivide()**: Divides the node into 8 smaller child nodes, each representing an octant of the cube.
- **insert(triangle, vertices)**: Inserts a triangle into the node.
- **contains\_triangle(triangle, vertices)**: Checks if a triangle is fully contained within a node.
- **\_insert\_in\_children(triangle, vertices)**: Helper function to insert a triangle into a child node.
- **find\_octree\_root\_center\_and\_size(vertices)**: Calculates the center and size of the octree root node.
- **build\_octree(vertices, triangles)**: Builds the octree.

## main.py

The **main.py** file runs the overall program.

- Parses file names for bodies, sample readings, and mesh data.
- Calls the **calculate\_frame\_transformations()** function from **calculateTransformation.py** to compute the frame transformations  $FA,k$  and  $FB,k$ .
- Parses the mesh data using the **parse\_mesh\_file()** function in **parseMesh.py**.
- Calculates the closest point on the mesh for each frame using **find\_closest\_point()** and **find\_point\_on\_mesh()** functions, and writes the output to a file.

## findClosestPoint.py

The **findClosestPoint.py** file computes the closest point on a triangle to a given point via the **closest\_point\_on\_triangle()** method:

- Computes barycentric coordinates of a point with respect to a triangle.
- If the point is outside the triangle, it calculates the closest point on each triangle edge.

## findPointOnMesh.py

The **findPointOnMesh.py** file finds the closest point on a mesh via linear search in the **closest\_point\_on\_mesh()** function:

- Loops through all triangles in the mesh.
- For each triangle, it calculates the closest point to the given point by calling **closest\_point\_on\_triangle()**.
- Compares the distances from the given point to each triangle and updates the closest point.

## findPointOnMeshKDTree.py

The **findPointOnMeshKDTree.py** file uses the KD-tree for faster nearest-point searches via **closest\_point\_on\_mesh\_kd()**:

- Uses the KD-tree to find the nearest triangle centroids to the given point.
- For each of the closest triangles, it computes the closest point using **closest\_point\_on\_triangle()**.
- Selects the closest point across the nearest triangles.

## findPointOnMeshOctree.py

The **findPointOnMeshOctree.py** file uses an octree structure for faster spatial searches. It finds the closest point on a mesh to a given point by traversing the octree and calculating distances to the mesh's triangles via the **closest\_point\_on\_mesh\_octree()** function:

- Starts at the root of the octree and traverses through the tree, checking the leaf nodes.
- For each triangle in a leaf node, it uses the **closest\_point\_on\_triangle()** function to compute the closest point on the triangle.
- Keeps track of the closest point found across all triangles, updating it whenever a closer point is found.

## generateOutput.py

The **generateOutput.py** file writes the computed results to an output file. It calculates the transformed tip position and finds the closest point on a mesh via the **generate\_output()** method:

- For each frame, it computes the transformed tip position  $d$  using the frame's transformation matrices and the initial tip position  $A_{tip}$ .
- It finds the closest point on the mesh using the **closest\_point\_on\_mesh()** function.
- Writes the transformed tip position, the closest point on the mesh, and the magnitude of the difference between them to the output file.

## computeMean.py

The **calculateMeanDistance.py** file calculates the mean of the last column from multiple output files via the **calculate\_mean\_distance(assignment\_number, file\_letter, search\_type)** function:

- Calculates the mean of the last column.
- Returns the calculated mean value if successful, or None if the file is missing or empty.

## compareOutput.py

The `calculateMeanSquaredError.py` file computes the MSE between the results obtained from a specific output file and a debug output file.

- **read\_distance\_values(filepath):** Reads the last column of values from a specified file.
- **calculate\_mean\_squared\_error(assignment\_number, file\_letter, search\_type):** Calculates the MSE between the two files.

## 5 Debugging Methods

In our testing approach, we utilized unit tests to verify the correctness of various functions. We generated synthetic data for testing by manually defining vertices and triangles in controlled configurations, including right, equilateral, and axis-aligned triangles. Test points were selected with known closest projections on these triangles, enabling us to calculate expected results manually. Using these baseline expectations, we performed unit tests for individual functions and integration tests across components to ensure accuracy and consistency in finding closest points. This approach allowed us to verify both basic functionality and the effectiveness of search optimizations like KD-Trees and Octrees.

### 5.1 KD-Tree Construction (testBuildTree.py)

#### Test Data:

- **Vertices:** A set of points (e.g.,  $[0, 0, 0]$ ,  $[1, 0, 0]$ , etc.) was used to create a mock scenario for testing.
- **Triangles:** Mock triangles were formed using the indices of these vertices, ensuring a variety of points and edges were included to test the tree's response to different configurations.

**How we verified it:** We checked that the returned KDTree object was an instance of KDTree and that the number of elements in the `triangle_data` list matched the number of input triangles. For further verification, the centroid data of the KD-tree was printed, and the triangles were manually inspected to ensure they corresponded to the correct vertices.

### 5.2 Closest Point on Triangle (testClosestPoint.py)

#### Test Data:

- **Point inside the triangle:** We picked an interior point (e.g.,  $(0.5, 0.5, 0)$ ) and tested if the function returned this point as the closest point.
- **Points outside the triangle:** We then tested points outside the triangle, which were closest to specific edges or vertices of the triangle. Each test case involved setting up the point and the expected closest point on the triangle using manual calculations for reference.

**How we verified it:** For each case, we used the expected closest point (calculated manually) and compared it with the output of the `closest_point_on_triangle` function using `np.allclose` to ensure numerical precision. If the computed result deviated from the expected outcome, an error message would indicate the mismatch.

### 5.3 Optimization Comparison (testOptimization.py)

#### Test Data:

- **Test point:** A point in 3D space ( $(1.5, 9.5, 0.5)$ ) was selected to test how the different algorithms handle the query.
- **Meshes:** A mesh of vertices and triangles was parsed from an input file (`Problem3MeshFile.sur`) to simulate data.

**How we verified it:** For each algorithm, we calculated the closest point using the respective method and measured the time taken for each search. The output was verified manually by checking whether the point found by each method was reasonable given the configuration of the mesh. We also ensured that the KD-tree and octree algorithms were significantly faster than the linear search.

## 5.4 Point Set Registration (testPointSetRegistration.py)

### Test Data:

- **Pure translation:** We translated a simple set of points ( $\{(0, 0, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0)\}$ ) by a fixed vector  $(1, 2, 3)$ .
- **Rotation by 45 degrees:** We manually calculated the expected rotation matrix and used this to verify that the registration correctly identified the rotation matrix when applied to the point set.
- **Combined rotation and translation:** A 90-degree rotation around the Z-axis and a translation vector  $(1, 1, 1)$ .

**How we verified it:** For the translation test, we manually checked that the translation vector was correctly identified. For rotation, we verified the orthogonality of the rotation matrix and checked that its determinant was close to 1. For the combined rotation and translation test, we manually calculated the expected result and confirmed that both the rotation and translation were correct.

## 6 Results

Program File	Linear (Mean Distance)	KDTree (Mean Distance)	Octree (Mean Distance)
PA3-A	0.0032	0.0032	1.3343
PA3-B	1.4715	1.4715	2.2517
PA3-C	0.8111	0.8111	1.8029
PA3-D	1.4077	1.4077	2.2213
PA3-E	1.5799	1.5799	3.0119
PA3-F	1.2855	1.2855	2.0521

Table 1: Table 1: Known Data (A-F), with Mean Distance Data for Linear, KDTree, and Octree

Program File	Linear (Mean Distance)	KDTree (Mean Distance)	Octree (Mean Distance)
PA3-G	2.0604	2.0604	3.1376
PA3-H	1.3419	1.3419	1.9609
PA3-J	2.1775	2.1775	3.0487

Table 2: Table 2: Unknown Data (G-J), with Mean Distance Data for Linear, KDTree, and Octree

File	Linear (Search Time)	KDTree (Search Time)	KDTree (Build Time)	Octree (Search Time)	Octree (Build Time)
PA3-A	0.0835	0.0003	0.0098	0.0146	0.2083
PA3-B	0.1216	0.0003	0.0081	0.0170	0.2066
PA3-C	0.0785	0.0003	0.0086	0.0146	0.2106
PA3-D	0.0775	0.0003	0.0083	0.0148	0.2183
PA3-E	0.0775	0.0003	0.0087	0.0150	0.2125
PA3-F	0.0778	0.0003	0.0085	0.0156	0.2182
PA3-G	0.0792	0.0003	0.0084	0.0196	0.2150
PA3-H	0.0791	0.0003	0.0093	0.0174	0.2813
PA3-J	0.0775	0.0003	0.0090	0.0150	0.2165

Table 3: Search and Build Times of Each Search Method for Each File

## 7 Discussion

### 7.1 Known Data (A-F)

The results for the known data files (PA3-A to PA3-F) are presented in Table 1. The mean distance values for the three search algorithms—Linear, KDTree, and Octree—vary significantly, indicating different accuracies for each search method.

- **Linear Search:** The mean distances for the linear search method range from 0.0032 to 1.5799. The linear search method does not take advantage of any structure to optimize the search, which can lead to higher mean distances in datasets with more complex or higher-dimensional structures, such as the ones in PA3-E and PA3-F.
- **KDTree:** The KDTree search method yields the same mean distance values as the linear search for all program files. This suggests that, for these specific datasets, the KDTree algorithm did not offer an improvement over the linear search in terms of accuracy. The KDTree is designed to optimize the search by spatially reflecting the data points; however, this appears to not be a sufficient benefit. It is potentially due to the distribution of the data.
- **Octree:** The mean distances for the Octree algorithm are notably higher than those for both the Linear and KDTree methods, ranging from 1.3343 to 3.0119. The Octree algorithm performs spatial partitioning by recursively dividing the space into smaller octants, which can lead to higher distances if the partitioning does not fit well with the distribution of data points. The poor performance of the Octree method may be caused by over-partitioning or an inefficient indexing strategy, particularly with the complex data in PA3-E.

## 7.2 Unknown Data (G-J)

For the unknown data (PA3-G to PA3-J), as shown in Table 2, similar trends in the performance of the search methods are observed. However, the mean distances in the unknown datasets are higher compared to the known datasets, indicating potentially more complex data.

- **Linear Search:** The mean distances for the linear search are again high, ranging from 1.3419 to 2.1775. This is expected, as linear search lacks the efficiency optimizations of more structured methods.
- **KDTree:** The KDTree, however, does not behave as expected. It continues to exhibit the same mean distance values as the linear search, which implies that, once again, it does not provide a distinct advantage over linear search for these datasets. The data may not benefit from KDTree's hierarchical partitioning due to the spatial distribution of the points.
- **Octree:** As with the known data, the Octree results show higher mean distances, ranging from 1.9609 to 3.1376.

## 7.3 Search and Build Times

The search and build times for each program file are summarized in Table 3. These times represent computational complexity for each method, considering both search and build costs.

- **Linear Search:** The search times for the linear search method remain consistent across all datasets, ranging from 0.0775 to 0.1216 seconds. The search time for linear search is generally higher than both the KDTree and Octree methods, as expected, due to its brute-force approach.
- **KDTree:** The KDTree search time is remarkably low, often around 0.0003 seconds for each dataset. This is due to the efficiency of the KDTree structure, which allows for rapid querying using the binary tree representation. The KDTree's build times are also lower than Octree (between 0.0081 and 0.0098 seconds), indicating that construction is relatively fast even for large datasets.
- **Octree:** The Octree method exhibits both a higher search time than KDTree, ranging from 0.0146 to 0.0196 seconds, as well as a significantly higher build time, ranging from 0.2066 to 0.2813 seconds. This increase in search time could be attributed to the more complex structure of the Octree. The build time discrepancy also suggests that the Octree algorithm involves a more computationally expensive dividing process.

# 8 Who Did What?

## 8.1 Program

- Ben wrote the following files:

- `readData.py`
  - `pointSetRegistration.py`
  - `testPointSetRegistration.py`
- Aryavrat did everything else related to the program, including:
  - Implementing the core algorithms
  - Unit testing and optimizing
  - Handling integration of different data structures

## 8.2 Report

- Aryavrat was responsible for:
  - Debugging the code
  - Generating the results
- Ben did everything else for the report, including:
  - Writing the Mathematical and Algorithmic Approach
  - Writing the Program Structure
  - Writing the Discussion