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	Benjamin Miller
Email	bmill119@jh.edu
Other contact information (optional)	
Name 2	Aryavrat Gupta
Email	agupt110@jh.edu
Other contact information (optional)	
Signature (required)	I (we) have followed the rules in completing this assignment

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 4

Aryavrat Gupta, Benjamin Miller

12/6/2024

1 Introduction

The objective of this assignment is to implement a complete Iterative Closest Point (ICP) registration algorithm, building on the work developed in PA3. The goal is to compute the position of pointer A in CT coordinates relative to a 3D surface mesh, iteratively refining the transformation until convergence. This report builds on that of PA3 by adding the iterative component, highlighting modifications to the previous implementation, and analyzing updated results.

2 Updates from PA3 to PA4

- Added the iterative component to refine the rigid transformation (F_{reg}) until convergence.
- Wrote a new script, `generateOutput4.py`, to generate the new output.
- Fixed Arun's method in `pointSetRegistration.py` to handle cases with zero singular values based on feedback.
- Created an iterative registration function in `iterativeRegistration.py`.
- Improved program documentation.
- Added a visual representation of the program structure for clarity.

3 Mathematical Approach

3.1 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:

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

3.2 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:

$$R = V U^T$$

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:

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

3.3 Finding Pointer $A_{\textbf{Tip}}$ in Body B Coordinates

Given the position of the pointer tip A_{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.

3.4 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:

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

$$u = 1 - v - w.$$

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:

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

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:

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

where the scalar t is computed as:

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

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

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

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:

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

3.5 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:

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

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

3.6 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:

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

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

3.7 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.

3.8 Iterative Refinement of Transformation (F_{reg})

The Iterative Closest Point algorithm builds on the initial transformation F_{reg} by iteratively refining it to minimize the distance between points (s_k) and the closest points on the mesh (c_k). This process ensures convergence to an optimal alignment.

First, apply the current transformation F_{reg} to the d_k points (pointer tip positions in Body B coordinates) to generate s_k:

$$s_k = F_{reg} \cdot d_k$$

Next, for each transformed point s_k, determine the closest point c_k on the surface mesh using one of the previous search methods. Then, register the d_k points to the c_k points by recalculating the transformation F_{reg} with point set registration:

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

where R and p are updated. Finally, evaluate the Frobenius norm of the difference between successive transformations:

$$\Delta F = \|F_{reg}^{(i+1)} - F_{reg}^{(i)}\|_F$$

If $\Delta F <$ tol (e.g., $1e^{-5}$) or the maximum number of iterations is reached, stop. Now, update F_{reg} and repeat steps 1-4 until convergence.

3.9 Convergence Parameters and Tolerance

Convergence is monitored using:

$$\Delta F = \|F_{reg}^{(i+1)} - F_{reg}^{(i)}\|_F$$

where $\|\cdot\|_F$ is the Frobenius norm. This ensures that eventually, updated transformations differ by a negligible amount. The maximum number of iterations is capped (in our case, 10), and the tolerance for convergence is set to $1e^{-5}$ to optimize computational cost.

4 Algorithm Overview

4.1 Frame Transformations

Our implementation uses 3D points and transformation matrices to handle rotations and translations in homogeneous coordinates. The transformation matrix has the form:

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

where R is a 3×3 rotation matrix, and p is a 3×1 translation vector.

Inverse transformations are computed as:

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

This allows us to transform points between coordinate systems efficiently.

4.2 Point Set to Point Set Registration

The registration algorithm computes the optimal transformation (rotation and translation) to align two sets of corresponding 3D points.

1. Compute the average positions of both the source and target point sets.
2. Center each point set by subtracting averages.

3. Compute covariance matrix of centered point sets.
4. Perform Singular Value Decomposition on the covariance matrix.
5. Compute the rotation matrix R from the SVD results. If $\det(R) < 0$, adjust for a valid rotation.
6. Handle cases where singular values are zero or near-zero.
7. Compute translation vector p using the computed rotation.
8. Return final transformation matrix combining R and p.

Update: We included a check for zero or near-zero singular values to maintain stability during SVD.

4.3 Finding the Pointer Tip in Body B Coordinates

To compute the pointer tip position in Body B coordinates:

1. Use the transformation matrices (F_A) and (F_B) to compute the pointer tip position.

$$d_k = F_B^{-1} \cdot F_A \cdot A_{\text{tip}}$$

2. Extract position of the pointer tip from transformed coordinates.

4.4 Finding the Closest Point on a Triangle

The closest point to a given 3D point on a triangle is computed as follows:

1. Calculate barycentric coordinates to determine if the point lies inside the triangle.
2. If the point is inside the triangle, return it.
3. If the point lies outside the triangle, compute projections onto each triangle edge and find closest edge point.
4. Return closest point found.

4.5 Linear Search for Closest Point on a Mesh

Brute force:

1. For each triangle in the mesh:
   (a) Compute closest point on the triangle to the point.
   (b) Calculate distance between query point and closest point.
2. Keep track of smallest distance and corresponding point.
3. Return closest point found.

4.6 Optimizing with K-D Trees

To improve efficiency, we use a K-D Tree.

1. Build K-D Tree from the centroids of all triangles in the mesh.
2. For a given point,
   (a) Find nearest centroid using the K-D Tree.
   (b) Retrieve corresponding triangle and compute closest point on it.
3. Return closest point found.

4.7 Octree-Based Search

For also tested using Octrees:

1. Build Octree by recursively dividing the 3D space into octants.
2. For a given point,
   (a) Traverse Octree, starting from the root node.
   (b) Check triangles in leaf nodes for closest point.
3. Return closest point found.

4.8 Iterative Closest Point (ICP) Algorithm

Our most recent update for PA4 was including the iterative component of the ICP algorithm:

1. Start with initial transformation matrix (the identity matrix).
2. Transform ource points using current transformation.
3. Find closest points on mesh for each transformed source point (using a previous method).
4. Compute new transformation by registering source points to their corresponding closest points.
5. Check for convergence using the Frobenius norm:
$$\Delta F = \|F_{\text{new}} - F_{\text{old}}\|_F$$
6. Stop if change is below a threshold or the maximum number of iterations is reached.

Update: Convergence is monitored with a tolerance of 10^{-5} and a maximum of 10 iterations.

5 Program Structure

5.1 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.
- **iterativeRegistration.py:** Implements the iterative closest point (ICP) algorithm for refining transformations.

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.

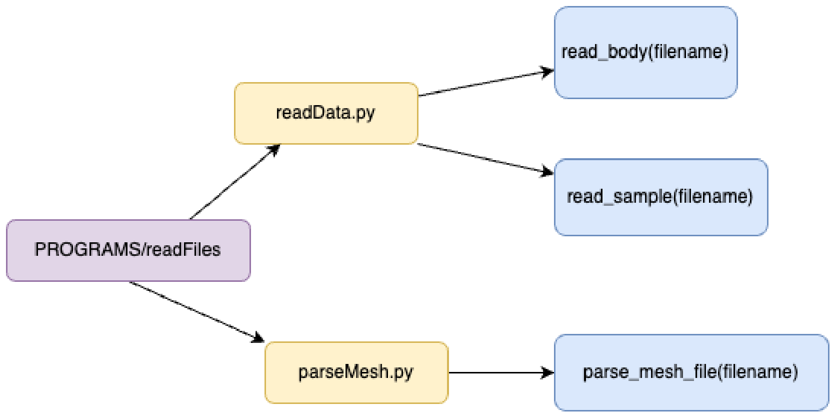
- **findPointOnMesh.py:** Linear Search to finds closest point.
- **findPointOnMeshKDTree.py:** KD-tree Search.
- **findPointOnMeshOctree.py:** Octree Search.
- **generateOutput.py:** Writes transformed tip positions and closest points on the mesh to an output file.
- **generateOutputPA4.py:** Performs similar output generation tasks as **generateOutput.py**, adapted for PA4 to include ICP.

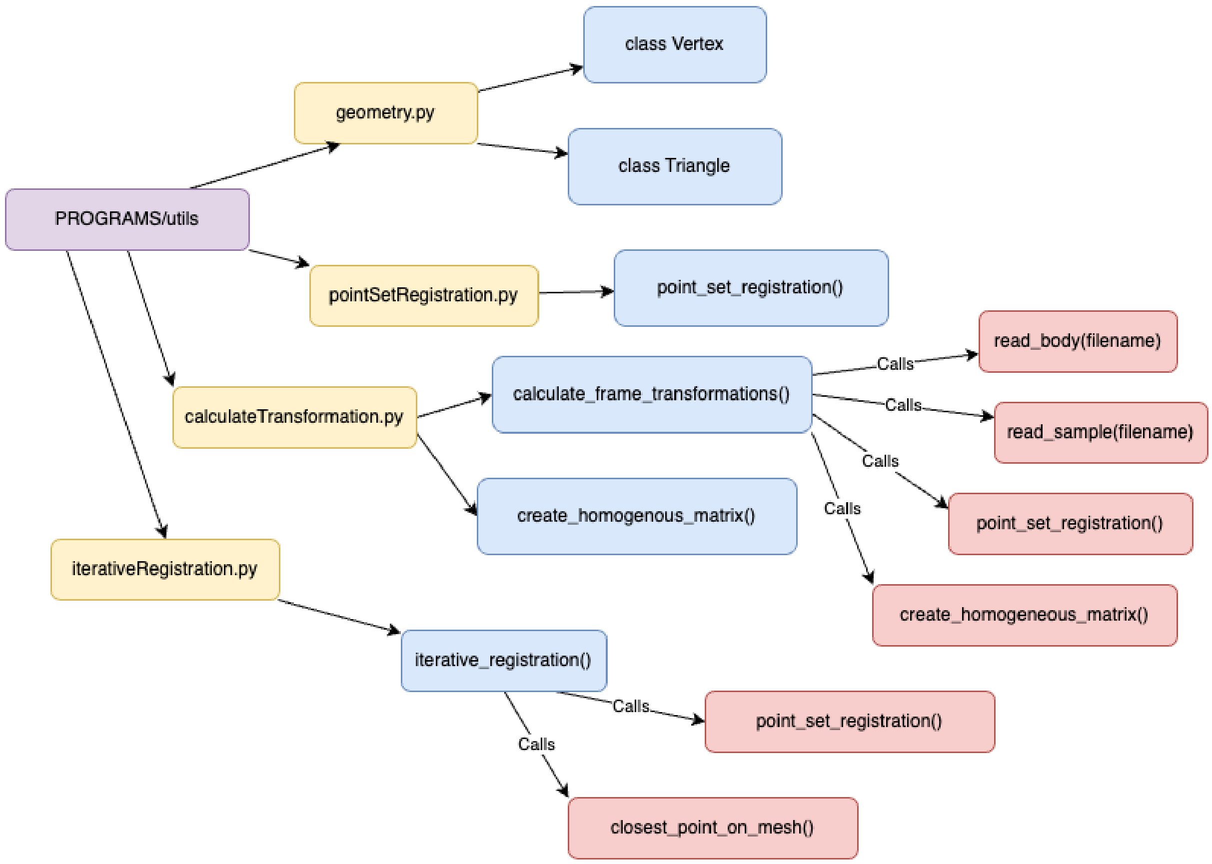
Error and Performance Metrics (errorCalculation):

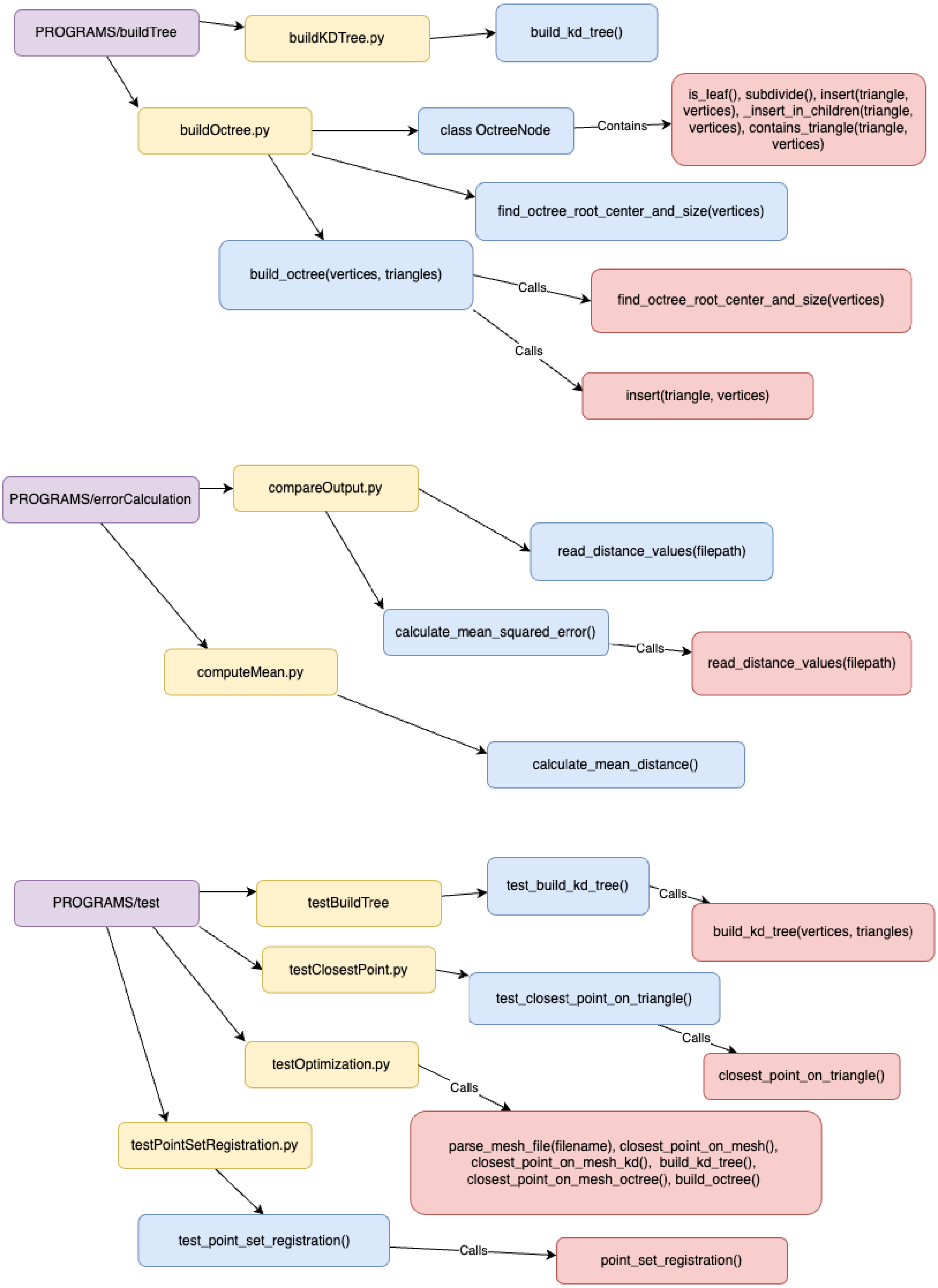
- **computeMean.py:** Computes the mean distance from the last column of output files.
- **compareOutput.py:** Computes the Mean Squared Error between the results and expected outputs.

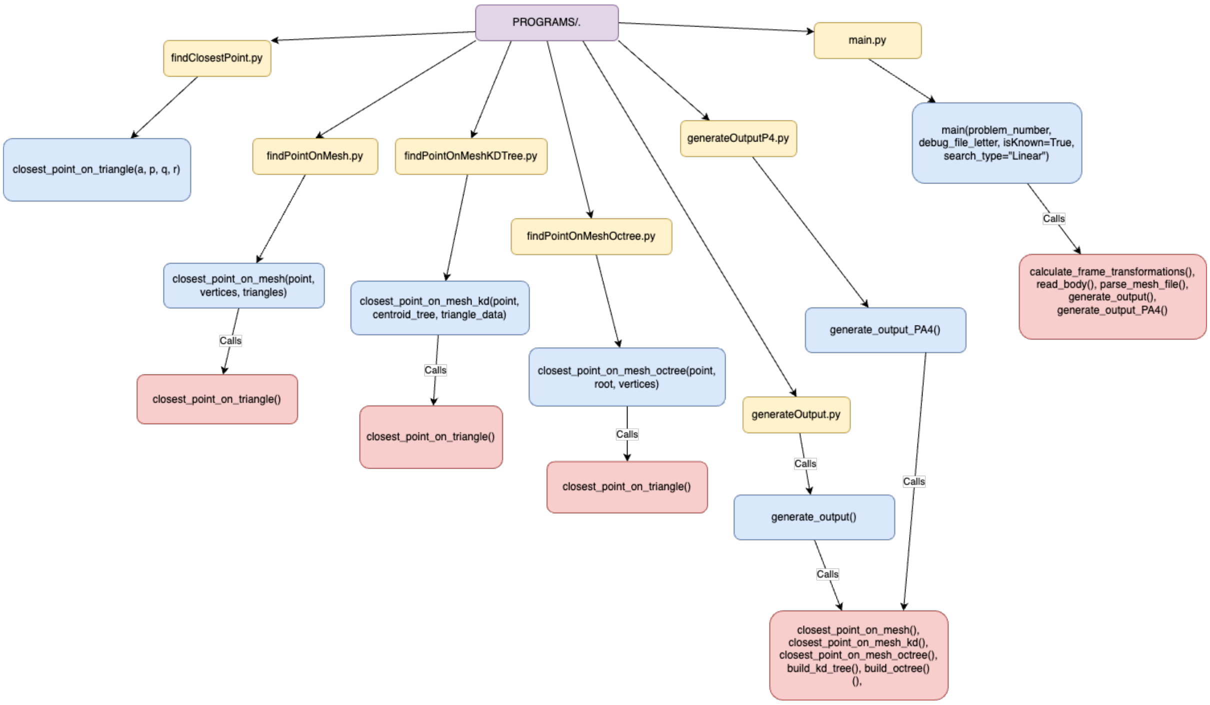
Please look at the README for the file tree expressed as relative paths from the root.

5.2 Visual Representation






PROGRAMS/buildTree
buildKDTree.py
build_kd_tree()
buildOctree.py
class OctreeNode
Contains
is_leaf(), subdivide(), insert(triangle, vertices), _insert_in_children(triangle, vertices), contains_triangle(triangle, vertices)
find_octree_root_center_and_size(vertices)
build_octree(vertices, triangles)
Calls
find_octree_root_center_and_size(vertices)
Calls
insert(triangle, vertices)
PROGRAMS/errorCalculation
compareOutput.py
read_distance_values(filepath)
calculate_mean_squared_error()
Calls
read_distance_values(filepath)
computeMean.py
calculate_mean_distance()
PROGRAMS/test
testBuildTree
test_build_kd_tree()
Calls
build_kd_tree(vertices, triangles)
testClosestPoint.py
test_closest_point_on_triangle()
Calls
closest_point_on_triangle()
testOptimization.py
Calls
parse_mesh_file(filename), closest_point_on_mesh(), closest_point_on_mesh_kd(), build_kd_tree(), closest_point_on_mesh_octree(), build_octree()
testPointSetRegistration.py
test_point_set_registration()
Calls
point_set_registration()



5.3 File Descriptions

readData.py

The **readData.py** file handles data input operations to read body marker and sample data files.

- **read_body(filename):** Reads marker LED coordinates and tip data.
  - **Input:**
    - **filename (str):** Prefix of calibration body file ("Problem3-BodyA").
  - **Output:**
    - **N_markers (int):** Number of marker LEDs in calibration body.
    - **marker_coords (list of list of float):** List of coordinates for each marker.
    - **tip_coords (list of float):** Coordinates of the pointer tip.
- **read_sample(filename):** Reads sample readings, including marker coordinates for body A, body B, and dummy markers across multiple frames.
  - **Input:**
    - **filename (str):** Prefix of the sample readings file ("PA3-A-Debug-SampleReadingsTest").
  - **Output:**
    - **NA (int):** Number of A LED markers.
    - **NB (int):** Number of B LED markers.
    - **ND (int):** Number of dummy LED markers.
    - **Nsamps (int):** Number of sample frames.
    - **readings (list of dict):** List of sample frame data:
      - **'A' (np.ndarray):** Coordinates of A body markers.
      - **'B' (np.ndarray):** Coordinates of B body markers.
      - **'D' (np.ndarray):** Coordinates of dummy markers.

parseMesh.py

The **parseMesh.py** file reads and processes the mesh data, including vertices and triangles.

- **parse_mesh_file(filename):** Reads vertex and triangle data.
  - **Input:**
    - **filename (str):** Path to mesh file ("PROGRAMS/data/Problem3MeshFile.sur").
  - **Output:**
    - **vertices (list of Vertex):** A list of **Vertex** objects representing the coordinates (x, y, z) of each vertex in mesh.
    - **triangles (list of Triangle):** A list of **Triangle** objects representing the vertices and neighbor information for each triangle in the mesh. Each triangle includes:
      - **v1, v2, v3 (int):** Indices of the vertices forming the triangle.
      - **n1, n2, n3 (int):** Indices of the neighboring triangles.

geometry.py

The **geometry.py** file defines classes for vertices and triangles.

- **Vertex Class:**
  - **Description:** 3D point with x, y, and z coordinates.
  - **Attributes:**
    - **x (float):** x-coordinate
    - **y (float):** y-coordinate
    - **z (float):** z-coordinate
  - **Initialization:**
    - Constructor `__init__(x, y, z)` initializes the vertex with the given x, y, and z coordinates.
- **Triangle Class:**
  - **Description:** Represents a triangle in a 3D mesh, defined by vertex indices and neighboring triangle indices.
  - **Attributes:**
    - **vertices (tuple of int):** A tuple containing the indices of three vertices that form the triangle.
    - **neighbors (tuple of int):** A tuple containing indices of three neighboring triangles.
  - **Initialization:**
    - Constructor `__init__(v1, v2, v3, n1, n2, n3)` initializes the triangle with the given vertex indices ($v1$, $v2$, $v3$) and neighbor indices ($n1$, $n2$, $n3$).

pointSetRegistration.py

The **pointSetRegistration.py** file from PA1/2. This file implements Arun's method for aligning two sets of 3D points. It also includes a check for degenerate cases where singular values are zero.

- **point_set_registration():**
  - **Inputs:**
    - **source (numpy.ndarray):** An $(N, 3)$ array representing source point set.
    - **target (numpy.ndarray):** An $(N, 3)$ array representing target point set.
  - **Outputs:**
    - **R (numpy.ndarray):** A $(3, 3)$ array representing the optimal rotation matrix.
    - **p (numpy.ndarray):** A $(3,)$ array representing the optimal translation vector.

calculateTransformation.py

The **calculateTransformation.py** file computes the frame transformations between the Body A and Body B coordinate systems for each frame in the sample data. It utilizes point set registration.

- **calculate_frame_transformations():**
  - **Inputs:**
    - **body_filename_a (str):** Filename prefix for Body A marker LED data, "Problem3-BodyA".
    - **body_filename_b (str):** Filename prefix for Body B marker LED data, "Problem3-BodyB".
    - **sample_filename (str):** Filename prefix for the sample readings file, "PA3-A-Debug-SampleReadingsTest".
  - **Outputs:**
    - **transformations (list):** A list of tuples $(F_{A,k}, F_{B,k})$, where $F_{A,k}$ and $F_{B,k}$ are 4x4 homogeneous transformation matrices for each frame.
- **create_homogeneous_matrix():**
  - **Inputs:**
    - **R (numpy.ndarray):** A 3×3 rotation matrix.
    - **p (numpy.ndarray):** A 3×1 translation vector.
  - **Outputs:**
    - **homogeneous_matrix (numpy.ndarray):** A 4×4 transformation matrix in homogeneous coordinates.

iterativeRegistration.py

The **iterativeRegistration.py** file performs iterative point cloud registration to align a set of 3D points to a target mesh using an ICP algorithm.

- **iterative_registration():**
  - **Inputs:**
    - **d_points (numpy.ndarray):** A $N \times 3$ array of points to be registered.
    - **vertices (list of Vertex):** A list of **Vertex** objects representing the vertices of the mesh.
    - **triangles (list of Triangle):** A list of **Triangle** objects representing the mesh's triangles.
    - **max_iterations (int, optional):** Maximum number of iterations (default: 100).
    - **tol (float, optional):** Convergence tolerance (default: 1×10^{-5}).
  - **Outputs:**
    - **F_reg (numpy.ndarray):** A 4×4 homogeneous transformation matrix representing the registration transformation.

buildKDTree.py

The **buildKDTree.py** file is responsible for building a KD-tree structure for 3D mesh data using **build_kd_tree()**.

- **build_kd_tree():**
  - **Inputs:**
    - **vertices (list of Vertex):** A list of **Vertex** objects.
    - **triangles (list of Triangle):** A list of **Triangle** objects.
  - **Outputs:**
    - **KDTree:** A KD-tree built from the centroids of the triangles. Each centroid represents the average position of a triangle's vertices.
    - **triangle_data (list of tuple):** A list where each entry is a tuple containing three **Vertex** objects, corresponding to the vertices of each triangle.

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.
  - **Attributes:**
    - **center (np.array):** Center point of the cube.
    - **size (float):** Length of the side of the cube.
    - **children (list):** A list of child **OctreeNode** objects representing subdivided regions.
    - **triangles (list):** A list of **Triangle** objects contained within this node.
  - **Methods:**
    - **is_leaf():** Returns `True` if the node has no children.
    - **subdivide():** Divides the node into 8 smaller sub-nodes, each representing an octant of the original cube.
    - **insert(triangle, vertices):** Inserts a triangle into the node. If the node exceeds a threshold of 5 triangles, it subdivides and redistributes the triangles among its children.
    - **_insert_in_children(triangle, vertices):** Attempts to insert a triangle into one of the child nodes.
    - **contains_triangle(triangle, vertices):** Checks if a triangle is fully contained within a node.
- **find_octree_root_center_and_size(vertices):**
  - **Inputs:**
    - **vertices (list of Vertex):** A list of **Vertex** objects representing the mesh's vertices in 3D space.
  - **Outputs:**
    - **center (np.array):** The computed center point of the octree root node.
    - **size (float):** The length of the side of the root node's cube.
- **build_octree(vertices, triangles):**
  - **Inputs:**
    - **vertices (list of Vertex):** A list of 3D vertices.
    - **triangles (list of Triangle):** A list of triangles.
  - **Outputs:**
    - **OctreeNode:** The root node of the constructed octree.

main.py

The **main.py** file orchestrates overall program execution by coordinating input parsing, transformation calculations, mesh processing, and output generation.

- **Parses Input Files:**
  - Reads body marker data for Body A and Body B using **read_body()** from **readData.py**.
  - Reads sample readings using **read_sample()** from **readData.py**.
  - Parses mesh data using **parse_mesh_file()** from **parseMesh.py**.
- **Calculates Transformations:**
  - Calls **calculate_frame_transformations()** from **calculateTransformation.py** to compute the frame transformations $F_{A,k}$ and $F_{B,k}$ for each frame.
- **Generates Output:**

  - Computes the closest points on the mesh using different search algorithms (**Linear**, **KDTree**, or **Octree**).
  - Writes the transformed tip positions and closest points to an output file.
  - For Problem 3: Calls **generate_output()** from **generateOutput.py**.
  - For Problem 4: Calls **generate_output_PA4()** from **generateOutputPA4.py**.

- **Outputs Results:**
  - Outputs are written to the specified directory (`./OUTPUT`) with filenames indicating the problem number, dataset type, and search method.

- **Inputs:**
  - **problem_number (int):** Specifies the problem number (3 or 4).
  - **debug_file_letter (str):** Specifies the test case ('A', 'B', etc.).
  - **isKnown (bool):** Indicates whether the dataset is known or unknown.
  - **search_type (str):** Specifies the search algorithm (**Linear**, **KDTree**, or **Octree**).

- **Outputs:**
  - **Output File:** Writes results to a file named `PA{problem_number}-{debug_file_letter}-OurOutput-{search` in the `./OUTPUT` directory.

findClosestPoint.py

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

- **Inputs:**
  - **a (tuple):** Coordinates of the query point (x, y, z).
  - **p, q, r (tuple):** Coordinates of the triangle vertices (x, y, z).

- **Outputs:**
  - **Closest Point (numpy.ndarray):** The closest point on the triangle or its edges to the query point.

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:** Iterates through each triangle in the mesh.
- **Computes closest point on each triangle:** For each triangle, calculates the closest point to the given point by calling **closest_point_on_triangle()** with the triangle's vertex coordinates.
- **Tracks the closest point:** Compares distances from the point to the closest points on all triangles and updates.

Inputs:

- **point (array-like):** The point as (x, y, z).
- **vertices (list of Vertex):** A list of vertices in the mesh.
- **triangles (list of Triangle):** A list of triangles.

Outputs:

- **Closest Point (numpy.ndarray):** The 3D coordinates of the closest point on the mesh to the given point.

findPointOnMeshKDTree.py

The **findPointOnMeshKDTree.py** file uses the KD-tree for faster nearest-point searches via **closest_point_on_mesh_kd()**:

- **Finds nearest centroids:** Utilizes the KD-tree to quickly find the closest triangle centroids to the given point.
- **Computes closest points:** For each triangle corresponding to the nearest centroids, calculates the closest point on the triangle using **closest_point_on_triangle()**.
- **Selects the closest point:** Compares the distances from the point to all candidate points and selects the closest one.

Inputs:

- **point (array-like):** The point as (x, y, z).
- **centroid_tree (KDTree):** A KD-tree built from the centroids of the triangles in the mesh.
- **triangle_data (list of tuple):** A list where each entry contains three vertices corresponding to the vertices of a triangle.

Outputs:

- **Closest Point (numpy.ndarray):** The 3D coordinates of the closest point.

findPointOnMeshOctree.py

The **findPointOnMeshOctree.py** file uses an octree structure for faster spatial searches.

- **Traverses the octree:** Begins at the root of the octree and iteratively checks all relevant nodes.
- **Handles leaf nodes:** For each triangle in a leaf node, calculates the closest point on the triangle using **closest_point_on_triangle()**.
- **Tracks the closest point:** Maintains the minimum distance and updates the closest point if a better candidate is found.
- **Processes child nodes:** If the node is not a leaf, recursively checks its children to continue the search.

Inputs:

- **point (tuple):** The point as (x, y, z).
- **root (OctreeNode):** The root of the octree containing the mesh triangles.
- **vertices (list of Vertex):** A list of vertices, where each vertex provides its 3D coordinates.

Outputs:

- **Closest Point (tuple):** The 3D coordinates of the closest point on the mesh.

generateOutput.py

The **generateOutput.py** file writes the computed results to an output file.

- **Handles search structures:** Builds the KDTree or Octree search structure if the corresponding search type is specified.
- **Calculates transformed tip position:** For each frame, computes the transformed tip position d.
- **Finds closest points:** Determines the closest point c on the mesh to d using **closest_point_on_mesh()** for Linear search, **closest_point_on_mesh_kd()** for KDTree, or **closest_point_on_mesh_octree()** for Octree.

- **Writes output:** Records d, c, and the magnitude of their difference $|d_k - c_k|$ to the output file.
- **Computes timing metrics:** Measures and prints the average build and search times for the specified search method.

Inputs:

- **output_dir (str):** Directory path where the output file will be saved.
- **filename (str):** Name of the output file.
- **search_type (str):** Search method to use ("Linear", "KDTree", or "Octree").
- **frames (list):** List of transformation matrices $F_{A,k}$ and $F_{B,k}$ for each frame.
- **Atip (numpy.ndarray):** Initial tip position in local coordinates.
- **vertices (list of Vertex):** List of vertices representing the mesh.
- **triangles (list of Triangle):** List of triangles in the mesh.

Outputs:

- **Output File:** Contains d (transformed tip position), c (closest point on the mesh), and $|d_k - c_k|$ (magnitude of their difference) for each frame.
- **Timing Metrics:** Average build and search times for the specified search method.

generateOutputPA4.py

The **generateOutputPA4.py** file writes the computed results to an output file. It iteratively refines a transformation F_{reg} using the ICP algorithm and calculates the transformed tip position s_k and its closest point c_k on a mesh:

- **Initial tip positions:** Computes initial d_k positions for each frame using the frame transformation matrices and the initial tip position A_{tip}.
- **Builds search structures:** Constructs KDTree or Octree structures for faster nearest-point searches if the corresponding search type is specified.
- **Iterative refinement:** Applies ICP to refine F_{reg}, transforming d_k into s_k and computing their closest points c_k on the mesh.
- **Output results:** Writes s_k, c_k, and $|s_k - c_k|$ to the output file, along with timing metrics.

Inputs:

- **output_dir (str):** Directory path where the output file will be saved.
- **filename (str):** Name of the output file.
- **search_type (str):** Search method to use ("Linear", "KDTree", or "Octree").
- **frames (list):** List of transformation matrices $F_{A,k}$ and $F_{B,k}$ for each frame.
- **Atip (numpy.ndarray):** Initial tip position in local coordinates.
- **vertices (list of Vertex):** List of vertices representing the mesh.
- **triangles (list of Triangle):** List of triangles in the mesh.
- **max_iterations (int, optional):** Maximum number of ICP iterations. Defaults to 10.
- **tol (float, optional):** Convergence tolerance for the Frobenius norm of F_{reg} updates. Defaults to 1×10^{-5}.

Outputs:

- **Output File:** Contains s_k (transformed tip position), c_k (closest point on the mesh), and $|s_k - c_k|$ (magnitude of their difference) for each frame.
- **Timing Metrics:** Average build and search times for the specified search method.

computeMean.py

The **computeMean.py** file calculates the mean of the last column from the output files via the **calculate_mean_distance(assignment_number, file_letter, search_type)** function.

- **Reads data from the output file:** Opens the specified file and extracts the last column values, which represent the magnitude of the difference $|s_k - c_k|$.
- **Calculates the mean:** Computes the mean of the last column values if the file contains valid data.
- **Handles missing or empty files:** Returns `None` if the file is missing or contains no data, along with an appropriate message.

Inputs:

- **assignment_number (int):** The assignment number (3,4).
- **file_letter (str):** The letter identifier for the file ('A', 'B').
- **search_type (str):** The search type ('Linear', 'KDTree', or 'Octree').

Outputs:

- **Mean Value (float or None):** The mean of the last column values if successful, or `None` if the file is missing.
- **Console Message:** Prints the mean value or an error message if the file is missing.

compareOutput.py

The **compareOutput.py** file computes the Mean Squared Error between the results from an output file and a corresponding debug output file.

- **read_distance_values(filepath):** Reads the last column values from the specified file. If the file does not exist, it returns `None` and prints an error message.
- **calculate_mean_squared_error(assignment_number, file_letter, search_type):** Calculates the MSE between the last column values of the specified result file and the corresponding debug file. It ensures both files exist and have the same number of rows before computation.

Inputs:

- **assignment_number (int):** The assignment number (3, 4).
- **file_letter (str):** The letter identifier for the file ('A', 'B').
- **search_type (str):** The search type ('Linear', 'KDTree', or 'Octree').

Outputs:

- **Mean Squared Error (float or None):** MSE value if successful, or `None` if there is an issue.
- **Console Message:** Prints the MSE value.

6 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.

6.1 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.

6.2 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.

6.3 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.

6.4 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.

Program File	Linear (Mean Distance)	KDTree (Mean Distance)	Octree (Mean Distance)
PA4-A	0.0020	0.0020	1.4297
PA4-B	0.2345	0.2345	1.5899
PA4-C	0.4462	0.4462	1.5111
PA4-D	0.4717	0.4717	1.5702
PA4-E	0.5146	0.5146	1.5447
PA4-F	0.4400	0.4400	1.5592

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)
PA4-G	0.2281	0.2281	1.4484
PA4-H	0.5222	0.5222	1.6200
PA4-J	0.4361	0.4361	1.6120

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)
PA4-A	0.078314	0.000261	0.011163	0.015126	0.226013
PA4-B	0.078772	0.000251	0.008834	0.015340	0.208840
PA4-C	0.079411	0.000259	0.008361	0.015446	0.210563
PA4-D	0.080529	0.000251	0.008346	0.015517	0.207480
PA4-E	0.080435	0.000260	0.008293	0.015375	0.214430
PA4-F	0.081942	0.000263	0.008651	0.015039	0.206194
PA4-G	0.078889	0.000262	0.008903	0.014869	0.214367
PA4-H	0.078170	0.000258	0.008638	0.014912	0.207535
PA4-J	0.078810	0.000254	0.008937	0.014690	0.206208

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

7 Results

8 Discussion

8.1 Known Data (A-F)

The results for the known data files (PA4-A to PA4-F) are presented in Table 4. The mean distance for each algorithms (Linear, KDTree, and Octree) show notable differences, indicating varying levels of accuracy for each method.

- **Linear Search**: The mean distances for the linear search method range from 0.0020 to 0.5146. This method calculates distances without optimizations, which explains higher distances in complex datasets like PA4-E and PA4-F. However, simpler datasets such as PA4-A shows a very low mean distance, meaning the method has basic reliability.

- **KDTree**: The KDTree search method produces identical mean distance values to the linear search for all program files. This suggests that KDTree matches the accuracy of linear search with potential computational benefits. Consistency across datasets similarly indicates that KDTree preserves accuracy in both simple and complex datasets.

- **Octree**: Finally, the Octree algorithm exhibits significantly higher mean distances compared to both Linear and KDTree methods, ranging from 1.4297 to 1.5592. The discrepancy may come from Octree's partitioning strategy, which leads to inefficiencies when the data distribution does not match the octree divisions. CLearly, this was not the optimal data structure for datasets like PA4-E and PA4-F.

8.2 Unknown Data (G-J)

For the unknown data files (PA4-G to PA4-J), as shown in Table 2, a similar pattern is demonstrated. Mean distances in these datasets are comparable to the known data, reflecting consistent limitations and benefits of each search algorithm.

- **Linear Search**: The mean distances for linear search range from 0.2281 to 0.5222, indicating a straightforward but computationally expensive approach. Accuracy remains reliable but unoptimized for complex datasets.
- **KDTree**: The KDTree algorithm continues to yield the same mean distance values as linear search. This reinforces that KDTree preserves the accuracy of linear search.
- **Octree**: Octree produces higher mean distances, ranging from 1.4484 to 1.6200. This confirms that Octree's performance is less effective for the tested datasets.

8.3 Search and Build Times

Search and build times for each file are summarized in Table 3.

- **Linear Search**: The search times for linear search are consistent across all datasets, ranging from 0.078170 to 0.081942 seconds. As expected, the brute-force approach is computationally expensive, particularly for larger datasets.
- **KDTree**: The KDTree search time, on the other hand, is remarkably low, ranging from 0.000251 to 0.000263 seconds. The efficiency likely comes from the hierarchical structure of the KDTree, which allows rapid nearest-neighbor searches. The build times for KDTree, ranging from 0.008293 to 0.011163 seconds, are also relatively low, meaning the method is suitable for datasets requiring repeated searches.
- **Octree**: The Octree method exhibits much higher search times than KDTree, ranging from 0.014690 to 0.015517 seconds, and significantly higher build times, between 0.206194 and 0.226013 seconds. THe build times reflect the computational cost of recursively dividing 3D space into octants. In this case, costs may outweigh the benefits for the Octree dataset.

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

8.4 Known Data (A-F)

The results for the known data files (PA3-A to PA3-F) without ICP highlight the impact of iterative registration on the accuracy of each search method.

- **Linear Search**: With ICP, the mean distances for the linear search range from 0.0020 to 0.5146, significantly improving compared to the non-ICP results, which range from 0.0032 to 1.5799.
- **KDTree**: Similar to the linear search, KDTree shows a reduction in mean distances when ICP is applied, with values ranging from 0.0020 to 0.5146 compared to 0.0032 to 1.5799 without ICP. While the KDTree algorithm inherently optimizes search efficiency, it also benefits from the optimized alignment from ICP.
- **Octree**: Octree method benefits the most from ICP, with mean distances decreasing from a range of 1.3343 to 3.0119 without ICP to a range of 1.4297 to 1.5592 with ICP. This suggests that the partitioning strategy of Octree is particularly sensitive to the alignment of the data.

9 Who Did What?

9.1 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

9.2 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