CIS PA3

Justin Wang, Alan You

November 2025

1 Overview

In this assignment, we are asked to implement the matching portion of a simplified Iterative Closest Point (ICP) algorithm to match a triangle mesh of a bone to a set of points sampled from a pointing device. The problem setup includes the following objects:

1. Optical tracker base
2. A bone (the target of the ICP)
3. Rigid body A, a free moving object with optical markers and known offset to the tip A_{tip}
4. Rigid body B, an object with optical markers on its body which is firmly attached to the bone

The following data about these objects is available:

1. A triangle mesh of the bone taken from a CT scan, containing the vertices of all of the triangles and their associated indices
2. A set of readings of the optical markers on rigid bodies A and B using the optical tracker base as rigid body A is moved to be in contact to the bone at different positions.

The ultimate goal is to find the registration between the point cloud generated by sampling A_{tip} while moving rigid body A around and the triangle mesh of the bone. In PA3 specifically, we focus on the matching step of the ICP algorithm which will eventually be used to achieve this.

As a note, the PCR algorithm used in this PA is the exact same as used in PA1 and PA2. For the sake of concision, the algorithm is not described in this writeup as it was already covered thoroughly in PA1 and PA2.

2 Mathematical approach

1. **Transformation Chain:**

   We first need to find the coordinates of the optical trackers on bodies A and B in the optical tracker base frame. This is required all k samples we take. After performing PCR, we are left with F_{A_k} and F_{B_k}, which are the HTMs for both bodies for the kth sample. We are interested in finding the location of A_{tip} in the frame of body B, since that frame is static in relation to the bone. This step is needed for all k samples to generate the source point cloud, To obtain, this we can perform the following frame transformation chain for each of the k samples:
   $(F_{B_k}^{-1} * F_{A_k} * A_{tip})$.
   With this transformation chain, we can now construct a point cloud in the frame of rigid body B, constructed by sampling and transforming A_{tip}.

2. **Closest Point on Triangular Mesh:**

   To find the closest point on a triangle to an arbitrary point $\mathbf{a}$, we follow the method from Taylor's slides [1]. Given triangle vertices $\mathbf{p}, \mathbf{q}, \mathbf{r}$, any point on the triangle can be written

   $$\mathbf{c}(\lambda, \mu) = \mathbf{p} + \lambda(\mathbf{q} - \mathbf{p}) + \mu(\mathbf{r} - \mathbf{p}).$$

   We first compute the projection of $\mathbf{a}$ onto the triangle by solving the least–squares problem:

   $$\min_{\lambda,\mu} \left\|\mathbf{a} - \mathbf{c}(\lambda, \mu)\right\|^2, \qquad \mathbf{A} = \begin{bmatrix}\mathbf{q} - \mathbf{p} & \mathbf{r} - \mathbf{p}\end{bmatrix}, \quad \mathbf{b} = \mathbf{a} - \mathbf{p}.$$

   $$[\lambda\ , \mu] = \arg\min \|\mathbf{A}[\lambda\ , \mu] - \mathbf{b}\|^2,$$

   This is solved easily enough with a numerical linear algebra library. With λ and μ, we can now divide the area around and including the triangle into two primarily areas:

   (a) Point lies inside the triangle:

   $$\lambda \geq 0, \qquad \mu \geq 0, \qquad \lambda + \mu \leq 1,$$

   in which case $\mathbf{c}$ is the closest point.

   (b) Point lies outside the triangle on one of three exterior regions:

   $$\lambda < 0 : \text{Line } \mathbf{rp}, \qquad \mu < 0 : \text{Line } \mathbf{pq}, \qquad \lambda + \mu > 1 : \text{Line } \mathbf{qr}$$

   In this case $\mathbf{c}$ must lie on one of the three exterior lines. An example is with this edge with vertices $\mathbf{q}$ and $\mathbf{p}$:

   $$\lambda = \frac{(\mathbf{c} - \mathbf{p}) \cdot (\mathbf{q} - \mathbf{p})}{\|\mathbf{q} - \mathbf{p}\|^2}, \qquad \lambda^{(\text{seg})} = \max(0, \min(\lambda), 1), \qquad \mathbf{c} = \mathbf{p} + \lambda^{(\text{seg})}(\mathbf{q} - \mathbf{p}).$$

   This is generalized for all possibilities $\mathbf{qr}, \mathbf{pq}$, and $\mathbf{rp}$.

Using this method, regardless if the point lines inside of outside the triangle, it is boxed to a point on the triangle.

Note: in our implementation, we used a kd-tree to store the triangle mesh. Since we construct the kd-tree based off of the centroid of each of the triangles but are interested in the closest point anywhere on a surface of any of the triangles, we used a method as described in [2] and [3] to construct bounding boxes around groups of triangles, and computing distance from query to bounding boxes to allow accurate search. The distance from point to bounding box is simple Euclidean distance formula, bounded below by 0 (points inside a bounding box has distance 0 to it).

3 Algorithmic approach

Algorithm 1 FindClosestPoint($\mathbf{a}$, $\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$)

$\mathbf{qp} \leftarrow \mathbf{q} - \mathbf{p}$
$\mathbf{rp} \leftarrow \mathbf{r} - \mathbf{p}$
Solve the least-squares problem:

$$\mathbf{a} - \mathbf{p} \approx \lambda\,\mathbf{qp} + \mu\,\mathbf{rp}.$$

Extract (λ, μ).

if $\lambda \geq 0$ **and** $\mu \geq 0$ **and** $\lambda + \mu \leq 1$ **then**
    return $\mathbf{c}^* = \mathbf{p} + \lambda\,\mathbf{qp} + \mu\,\mathbf{rp}$
end if

$\mathbf{c} \leftarrow \mathbf{p} + \lambda\,\mathbf{qp} + \mu\,\mathbf{rp}$

if $\lambda < 0$ **then**
    return ProjectOnSegment($\mathbf{c}$, $\mathbf{p}$, $\mathbf{r}$)
else if $\mu < 0$ **then**
    return ProjectOnSegment($\mathbf{c}$, $\mathbf{p}$, $\mathbf{q}$)
else
    return ProjectOnSegment($\mathbf{c}$, $\mathbf{q}$, $\mathbf{r}$)
end if

Algorithm 2 ProjectOnSegment($\mathbf{c}, \mathbf{u}, \mathbf{v}$)

$\mathbf{d} \leftarrow \mathbf{v} - \mathbf{u}$
$t \leftarrow \dfrac{(\mathbf{c} - \mathbf{u}) \cdot \mathbf{d}}{\|\mathbf{d}\|^2}$
$t \leftarrow \min(1, \max(0, t))$
return $\mathbf{u} + t\,\mathbf{d}$

Description of kd-tree

We decided to store the triangle mesh in a kd-tree (defined in ./utils/kdtree.py) to allow faster nearest-neighbor matching for points (as compared to brute-force linear search). The kd-tree is designed to work in 3D, two children per node, one triangle per node, split along the median of each axis in round robin fashion. We also found the implementation of bounding boxes as described in [2] and [3] to be necessary to match the performance of brute force search. Based off of our examination of the datasets which showed roughly uniform distribution of points along each axis and our implementation of our kd-tree, our nearest-neighbor search should be $O(log\ \mathrm{n})$ on average.

The kd-tree is an object with two functions, whose algorithms are described below. Their purposes are to construct a kd-tree from a traingle mesh, and search for a closest point on the triangle mesh given a 3D point.

Description of kd-tree node

Each node is defined as follows with private fields:

```
class KDTreeNode
tri_idx :  triangle index stored in current node
point :  centroid for splitting
left, right :  subtree nodes
axis :  splitting axis (0,1, or 2)
bbox_min, bbox_max :  bounding-box limits
```

Algorithm 3 BuildKDTree(tri_indices, depth)

if |tri_indices| = 0 **then**
    return None
end if
axis ← depth mod 3
sorted_idx ← argsort(centroids[tri_indices, axis])
tri_indices ← tri_indices[sorted_idx]
median_idx ← ⌊|tri_indices|/2⌋
median_tri ← tri_indices[median_idx]
centroid ← centroids[median_tri]
bbox_min ← min(tri_bbox_min[tri_indices])
bbox_max ← max(tri_bbox_max[tri_indices])
if |tri_indices| = 1 **then**
    return KDTreeNode([median_tri], centroid, None, None, axis, bbox_min, bbox_max)
end if
left ← BuildKDTree(tri_indices[:median_idx], depth + 1)
right ← BuildKDTree(tri_indices[median_idx+1:], depth + 1)
return KDTreeNode([median_tri], centroid, left, right, axis, bbox_min, bbox_max)

Algorithm 4 ClosestPoint(p)

```
best ← None
stack ← [root]                                                    ▷ DFS stack
while stack is not empty do
    node ← stack.pop()
    if node = None then
        continue
    end if
    if best ≠ None then
        box_dist ← point_bbox_distance(p, node.bbox_min, node.bbox_max)
        if box_dist ≥ best[0] then
            continue                                  ▷ This subtree gets pruned
        end if
    end if
    for all tri_idx ∈ node.tri_indices do
        (v0, v1, v2) ← vertices[triangles[tri_idx]]
        q ← find_closest_point(p, v0, v1, v2)
        dist ← ‖p − q‖
        if best = None or dist < best[0] then
            best ← (dist, q, [tri_idx])
        end if
    end for
    diff ← p[node.axis] − node.point[node.axis]
    if diff < 0 then
        near, far ← node.left, node.right
    else
        near, far ← node.right, node.left
    end if
    stack.append(far)
    stack.append(near)
end while
(dist, closest_point, tri_idx) ← best
return closest_point, dist, tri_idx[0]
```

4 Overview of structure

The criteria for structuring our code was simplicity, ease of debugging,and compatible with PA4. Since the nature PA3 is quite procedural with no obvious object, we decided it would make the most sense for this to be structured as a script that relies on helper functions. In PA4, we will consider a more object oriented approach with the work done here as a member function.

1. **Main script**
   main.py
   PA3 is fully answered in main.py. In main.py, these steps are followed:
   (a) Load in rigid body marker and mesh files
   (b) Build a kd-tree to store the triangle mesh
   (c) Load in a dataset
   (d) Perform PCR between the two rigid bodies
   (e) Calculate A_{tip} points in rigid body B frame
   (f) Apply registration transformation (assumed to be identity for PA3
   (g) Find closes point on the triangle mesh for every point generated from A_{tip}
   (h) Output results in desired formate
   (i) Repeat (c) through (h) for all of the datasets
2. **Utility Scripts**

`./utils/calculate_errors.py`

```
function rms = calculate_rms_error(pc1, pc2)
function [F_diff, angle_err, trans_err] =
calculate_error_transformation(Fa, Fb)
function stats = calculate_error_stats(pc1, pc2)
function print_error_stats(stats)
```

This file provides basic error–evaluation utilities for PA3/PA4. It computes RMS point–cloud error, rotational and translational differences between two rigid transforms, and summary statistics for point-wise residuals.

`./utils/icp.py`

```
function c = project_on_segment(c, p, q)
function c = find_closest_point(a, p, q, r)
function [q_closest, d, idx] = linear_search_closest_poin
ts_on_mesh(p, V, T)
function test_closest_point_on_triangle()
```

This file implements closest point computations on triangles and triangle meshes by constructing a linear search. It includes small unit test validates correctness on all triangle regions.

./utils/kdtree.py

```
function d = point_aabb_distance(p, aabb_min, aabb_max)
class KDTreeNode
class KDTreeTriangles
method build_kdtree(tri_indices)
method closest_point(p)
method _closest_point_recursive(p, node)
```

This file constructs a per-triangle kd-tree to accelerate closest-point queries on a mesh. Each node stores triangle indices, centroid split axis, and a bounding box for improved pruning. Queries descend the KD-tree, skipping subtrees whose bounding boxes cannot contain a closer point, and use `find_closest_point` to compute the exact closest point on each candidate triangle. The kd-tree object defined here encompasses both the structure, building of, and execution of a closest point search in a kd-tree containing a triangle mesh.

./utils/parse.py

```
function [markers, tip, N] = parse_rigid_bodies(path)
function [V, T] = parse_mesh(path)
function [A_tr, B_tr, Ns] = parse_readings(path, NA, NB)
function [d_k, c_k] = parse_output(path)
```

This file provides all input–output parsing utilities for PA3. It loads rigid-body definitions, surface-mesh vertex and triangle lists, tracker sample readings for bodies A and B, and previously generated output files. All routines convert the assignment's text formats into clean NumPy arrays for downstream computation.

./utils/pcr.py

```
function F = point_cloud_registration(a, b)
function random_pcr_test()
```

This file implements the rigid point–cloud registration method used in PA3. The main routine applies the Kabsch algorithm: centering both

point sets, forming the cross–covariance matrix, performing SVD, enforcing a proper rotation in $SO(3)$, and constructing the resulting 4×4 homogeneous transform. A built-in randomized unit test verifies correctness by comparing the recovered transform against a known ground-truth transformation.

`./utils/write_out.py`

```
function write_p3_output(d_points, c_points, errors,
path)
```

This file formats and writes the PA3 output file in the exact specification required by the assignment. It writes each sample's $\mathbf{d}_k$, $\mathbf{c}_k$, and the magnitude of their difference, using the correct header format and column spacing. The routine ensures directory creation, validates input array shapes, and outputs a file directly compatible with the provided grading scripts.

3. Testing Scripts

`./tests/compare_outputs.py`

```
function compare_output(letter)
```

This test script compares the PA3 output files against the provided debug "ground truth" outputs. It loads each dataset's $\mathbf{d}_k$ and $\mathbf{c}_k$ values using `parse_output`, computes RMS error using `calculate_error_stats`, and reports discrepancies for all debug sets A–F. This tool is for local verification only and is not part of the required submission.

`./tests/kdtree_test.py`

```
function test_triangle_kdtree()
```

This script provides unit tests for the triangle kd-tree implementation. It constructs simple meshes, queries closest points, and verifies correctness against analytically known solutions for interior, exterior, and multi-triangle cases. These tests validate pruning logic and exact closest-point evaluation on triangles.

`./tests/closest_point_test.py`

```
function assert_close(x, y, msg)
function run_tests()
```

This script provides a correctness test for the triangle closest–point routine. It evaluates all barycentric regions (interior, vertices, and all three edges), including near-edge projections, and compares results against analytically computed ground truth using `project_on_segment`.

5 Validation Approach

In general, we used the following equations implemented in `calculate_errors.py` to evaluate our algorithms:

$$e_{\text{rms}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left\|\mathbf{pc}_1^{(i)} - \mathbf{pc}_2^{(i)}\right\|^2}.$$

$$\text{mean} = \frac{1}{N}\sum_{i=1}^{N}\left\|\mathbf{pc}_1^{(i)} - \mathbf{pc}_2^{(i)}\right\|.$$

$$\text{std} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\left\|\mathbf{pc}_1^{(i)} - \mathbf{pc}_2^{(i)}\right\| - \text{mean}\right)^2}.$$

1. **Comparing Outputs**

   Datasets A-F contain sample output files. We can treat these files are "ground truths" to validate out generated output files. For each dataset, we compute a RMS error between the encoded d_k and c_k point clouds.

2. **Validating Closest Point Algorithm**

   The primary ethos when testing our Closest Point Algorithm is to generate known triangle, place points on the inside face and outside face (three regions of the outside face), and determine if the algorithm correctly solves for the correct point.

3. **Validating kd-Tree** Our approach to validating our kd-tree implementation focused on building the tree, and performing a nearest point search. Since these two functions are so closely aligned and dependent on each other, we test them in conjunction.
   We first test with a mesh of just one triangle. This test case helped us verify that we are instantiating the kd-tree at all, and the ability to generate a root node. Then, we tried two test queries, one for a point inside

the triangle, and one outside. These two options cover all possibilities, where the distance from the query point to the triangle is either positive or zero (a query for a point on the boundary of triangle will also result in distance of zero). We verified the correctness of our code by comparing the code's output with out manually computed expected outputs.
We then move on to testing the kd-tree on a mesh with multiple triangles. In this case, there is three triangles. Again, we tested querying a point inside one of the triangles, and another point outside of all of the triangles, again verifying by hand. We believe this covers all cases, as this case shows the ability to give a node two children and still correctly calculate nearest point on mesh.
Finally, the ultimate test for our kd-tree was comparing its performance to the brute force linear search for nearest point on a triangle mesh on the real dataset. We discuss this in more detail in the results section, but briefly speaking, the kd-tree implementation achieved the same accuracy and faster runtime when compared to the brute force linear search, which is verification enough that our implementation is correct for the given problem.

6 Results

Dataset	**RMSE $\mathbf{d_k}$**	**RMSE $\mathbf{c_k}$**
A	0.008165	0.007303
B	0.007746	0.006831
C	0.008944	0.008563
D	0.010954	0.008944
E	0.009309	0.007303
F	0.008944	0.007746

Table 1: RMSE comparison between "Ground Truth" debug output and generated output for datasets A–F.

When comparing the outputs for debug data sets A-F in Table 1, we see very low error. This indicates our closest point algorithm performs very well across multiple shaped meshes and orientations.
We also came upon some interesting results when comparing the results of our algorithm when using brute force linear search versus kd-tree search, namely about the accuracy and the runtime. In terms of accuracy, we know that the linear search must always give the true nearest point for a given query, so we take that as ground truth. When we ran our code with the kd-tree implementation, we arrived at the same RMSE errors for both c_k and d_k as with the linear search. This proved that our kd-tree based search returned the same point on the triangle mesh for every query as the linear search did. In terms of runtime, the kd-tree implementation consistently took near 1.5 seconds for all

of main.py, whereas the linear search took around 7 seconds. This is another step in verifying the correctness of our code, as this magnitude of decrease in runtime is around what we expect (all times measured using the same personal computer).
Note: Due to these very conclusive results in the clear advantage of the kd-tree, we did not intend on the linear search based algorithm to be easily accessible in our final code submission, although the function is still included in ./utils/icp.py for reference.

Alan and Justin both contributed equally to this PA with both partners researching the best mathematical implementation for closest point on mesh algorithm, Alan implementing the closest point algorithm, and Justin implementing the kd-Tree.

7 References

[1] R. Taylor, "Frames, Transformations, and Rigid-Body Kinematics," *CIS 455/655 Lecture Notes*, Johns Hopkins University. `https://ciis.lcsr.jhu.edu/lib/exe/fetch.php?media=courses:455-655:lectures:frames.pdf`

[2] R. Taylor, "Finding point-pairs," 2022. Accessed: Nov. 13, 2025. [Online]. Available: `https://ciis.lcsr.jhu.edu/lib/exe/fetch.php?media=courses:455-655:lectures:finding_point-pairs.pdf`

[3] "KD-Trees," Accessed: Nov. 13, 2025. [Online]. Available: `https://cgvr.cs.uni-bremen.de/teaching/geom_notes_from_lorenzo/KD-Trees.pdf`

[4] R. Taylor, "Registration - Part 1 WHITING SCHOOL OF ENGINEERING And many more applications... Medical interventions Slide Credit: Ayushi Sinha 2," Oct. 2025. Accessed: Nov. 13, 2025. [Online]. Available: `https://ciis.lcsr.jhu.edu/lib/exe/fetch.php?media=courses:455-655:lectures:registration_part_1.pdf`

[5] R. Taylor, "Registration - Part 2 WHITING SCHOOL OF ENGINEERING," Oct. 2023. Accessed: Nov. 13, 2025. [Online]. Available: `https://ciis.lcsr.jhu.edu/lib/exe/fetch.php?media=courses:455-655:lectures:registration_part_2.pdf`

[6] GeeksforGeeks, "Search and Insertion in K Dimensional tree," GeeksforGeeks, Oct. 31, 2014. `https://www.geeksforgeeks.org/dsa/search-and-insertion-in-k-dimensional-tree/`