# Programming Assignments 3 and 4 – 601.455/655 Fall 2025

**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 | Sahana Raja - 601.455 |
| Email | sraja11@jh.edu |
| Other contact information (optional) | |
| Name 2 | Rohit Satish - 601.455 |
| Email | rsatish2@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 | |

**Overview**

In computer integrated surgery, accurate alignment between the patient's anatomy and the preoperative images available is essential for precise navigation and guidance. The Iterative Closest Point or ICP algorithm is one of the core methods that is used for this purpose, establishing a spatial correspondence between physical measurements and a reference model. This assignment specifically focuses on implementing the matching phase of the ICP algorithm, where each measured point is paired with its associated nearest point on the reference surface.

In the setup given by this assignment, two rigid bodies are tracked using an optical tracker system. Body A acts like a pointer, with a known tip position in Body A's coordinate frame. Body B is rigidly attached to the bone and actually defines the bone's coordinate frame. The optical tracker measures the positions of the LED markers on both bodies in the optical tracker coordinate frame while each body's local marker geometry (markers in their respective body' frame) is known. If we combine these measurements we can determine how each body is positioned relative to the tracker for any frame we measure and then express the location of the pointer tip in the coordinate system of the bone. Because in this assignment the bone frame is considered to coincide with the CT coordinate system, each measured tip point can be directly compared to the CT surface mesh. The algorithm we developed can then identify the closest surface point for every measured tip position, which is what we use to form a set of point correspondences (between the measured points and the closest surface mesh point) which is the very foundation of the full ICP registration that would be coming up in later assignments.

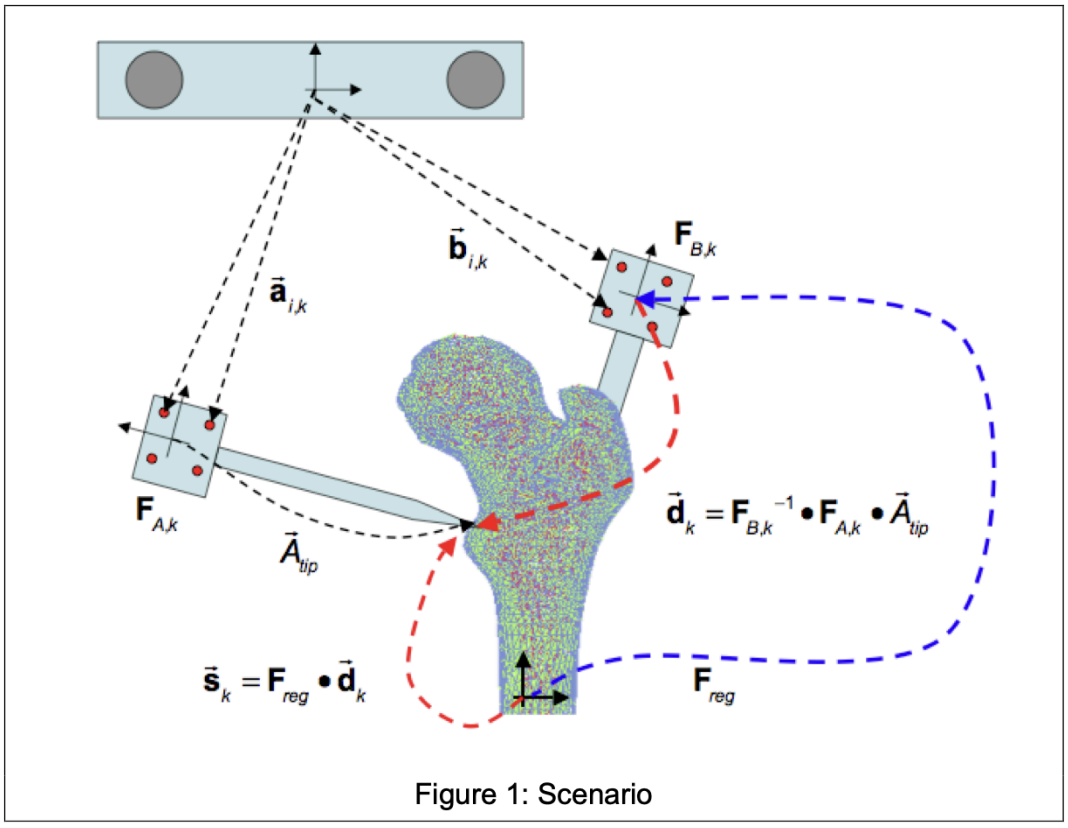



Figure 1: Scenario

**Mathematical Approach**

The goal of this assignment is to compute, for every sample frame *k*, the pointer tip position in the bone coordinate system so that we can then determine the closest corresponding point on the surface of the CT mesh. This would involve a series of rigid body transformations and point to surface matching operation which we detail below.

**Part 1: Rigid Body registration**

Each of the rigid bodies, A and B, has a fixed number of optical markers on them whose coordinates in the body frame are known to be $\vec{A}_i$ and $\vec{B}_i$. For each frame, k, the optical tracker measures the position of these same markers in the optical tracker coordinate system as $\vec{a}_{i,k}$ and $\vec{b}_{i,k}$. So using this information we can construct the following relations:

$\vec{a}_{i,k} = F_{A,k}\vec{A}_i = [R_{A,k} | \vec{t}_{A,k}]\ \vec{A}_k$ and similarly $\vec{b}_{i,k} = F_{B,k}\vec{B}_i = [R_{B,k} | \vec{t}_{B,k}]\ \vec{B}_k$ where $R_{A,k}$ and $R_{B,k}$ are 3x3 rotation matrices and $\vec{t}_{A,k}$ and $\vec{t}_{B,k}$ are translation vectors.

To compute these transformations, a point set registration method has to be used which in our case was SVD-based absolute orientation, which minimizes the sum of the squared distances between corresponding marker pairs.

$$\min_{R_{A,k} t_i} \sum_i ||R_{A,k}\vec{A}_i + \vec{t}_{A,k} - \vec{a}_{i,k}||^2$$

and similarly

$$\min_{R_{B,k} t_i} \sum_i ||R_{B,k}\vec{B}_i + \vec{t}_{B,k} - \vec{b}_{i,k}||^2$$

The optimal rotation and translation for this equation are computed from the singular value decomposition of the cross-covariance matrix between the two point sets. Let the centroids of the optical tracker and body coordinates, both for body A, be $\overline{a_{i,k}}$ and $\overline{A}$. We can then form the centered point sets $A_i' = A_i - \overline{A}$ and $\vec{a}_{i,k}' = \vec{a}_{i,k} - \overline{a_{i,k}}$.

From this we can then find the cross-covariance matrix as $H = \sum_i A_i'(\vec{a}_{i,k}')^T$ and then taking the singular value decomposition $H = USV^T$ we get that the optimal rotation we are looking for is $R = VU^T$ and $\vec{t} = \overline{a_{i,k}} - R\overline{A}$.

To avoid reflections on accident if det(R) < 0, the sign of the last column of V would be flipped before we then recompute $R = VU^T$. This guarantees a proper rotation where the condition det(R) = +1 must be met for the recommended best rotation to be valid. With this in mind we also checked if det(R)=0 in which case we produced a degenerate matrix and a rigid body transform does not exist.

**Part 2: Pointer Tip Transformation**

The pointer tip position in the coordinate frame of Body A is known as $\vec{A}_{tip}$. The position of that point in terms of the coordinate frame of Body B (the bone), we label as $\vec{d}_k$ and it can be found by: $\vec{d}_k = F_{B,k}^{-1} \bullet F_{A,k} \bullet \vec{A}_{tip}$

Now for each sample k we have the pointer tip of body A in the bone coordinate frame.

**Part 3: CT Coordinate Registration**

For this assignment the bone coordinate system is assumed to be perfectly aligned with that of the Ct coordinate system since $F_{reg} = I$ so this means that the pointer tip position $\vec{d}_k$ can be directly interpreted as already being in CT coordinates.

**Part 4: Surface Mesh Closest Point**

The bone model we have in CT coordinates is represented by a mesh of triangles with vertices $\vec{v}_j$. For each pointer tip $\vec{d}_k$, our algorithm would end up finding the closest point on the mesh $\vec{c}_k$. In a baseline implementation performing a brute force search across all the triangles in the mesh and computing the minimum Euclidean distance: $\vec{c}_k = arg \min_{\vec{x} \in mesh} ||\vec{d}_k - \vec{x}||$ could work but is sub-optimal. Instead we applied KDTree acceleration. We revised our implementation to a KDTree constructed out of a ll the mesh vertices. This tree performs an efficient nearest neighbor query so that we can receive the closest vertices to $\vec{d}_k$. Only triangles that are adjacent to those candidate vertices are checked using the exact point-on-triangle euclidean distance calculation. The resulting correspondence distance for each sample would thus be given by $Error_k = ||\vec{d}_k - \vec{c}_k||$.

Operationally the process is to:

1. Build a KDTree over all the mesh vertices
2. Query for the nearest vertices to $\vec{d}_k$
3. Gather the incident triangles
4. Compute exact closest point
5. Select the point with minimal euclidean distance.

This approach maintains the geometric efficiency of the brute force method but significantly reduces the computation time necessary by not ever calculating distances for obviously incorrect closest vertices. This is especially advantageous for meshes with thousands of triangles.

**Algorithmic Approach**

Programming language: Python 3
Nonstandard libraries: NumPy (array and linear algebra operations), All KD-tree functionality is implemented using an in house implemented media split KD-tree geometry.py

**Function: RigidRegistration (Body to Tracker)**

**Inputs:**

- Body frame marker set {Ai} or {Bi}
- Tracker frame marker set {ai,k} or {bi,k} for frame k

**Outputs:**

- Homogeneous transform FA,k or FB,k mapping the body frame to tracker coordinates

**Key Variables:**

- Ā,ā: centroids of body and tracker points
- H: 3×3 cross-covariance matrix
- U,Σ,V^⊤: SVD of H
- R: 3×3 rotation matrix
- t: 3×1 translation vector

**Pseudocode:**

RigidRegistration(BodyPoints, TrackerPoints):

1. N ← number of markers
2. Compute body centroid  A_bar = (1/N) * sum_i BodyPoints[i]
3. Compute tracker centroid a_bar = (1/N) * sum_i TrackerPoints[i]
4. For each i:

   A_prime[ i ] = BodyPoints[ i ]  - A_bar

   a_prime [ i ] = TrackerPoints[ i ] - a_bar
5. H ← 0 (3×3 matrix)
6. For each i:

   H ← H + A_prime[ i ] * (a_prime[ i ])^T
7. Compute SVD: H = U Σ V^T

8. R ← V U^T
9. If det(R) < 0:

   Flip sign of last column of V

   R ← V U^T
10. t ← a_bar - R * A_bar
11. Form F as 4×4 homogeneous matrix:

    [ R  t ]

    [ 0  1 ]
12. Return F

**Algorithm Description:**

To estimate the pose of each instrumented body in the tracker frame, we use an SVD based rigid registration algorithm that aligns two corresponding point clouds. For each frame, the known marker positions in the body coordinate system and their measured positions in tracker coordinates are first centered by subtracting their respective centroids. The 3×3 cross covariance matrix that is formed from the centered points is then decomposed via singular value decomposition. The optimal rotation is obtained as $R=VU^{\top}$, with a correction applied if $\det(R)<0$. The translation is computed as $t=\bar{a}-R\bar{A}$. These components are then assembled into a 4×4 homogeneous transform. This method is applied independently to Body A and Body B for every sample frame.

**Function: PointerTipTransform (Body A → Bone/CT frame)**

**Inputs:**

- Pointer tip Atip in Body A coordinates
- Body A transform FA,k
- Body B transform FB,k

**Output:**

- Sample point dk in CT coordinates

**Key Variables:**

- ptracker: tip in tracker frame
- pB: tip in Body B (bone) frame
- (FB,k)^ -1: inverse of Body B transform

**Pseudocode:**

PointerTipTransform(A_tip, F_Ak, F_Bk):

1. p_tracker ← F_Ak * A_tip
2. Compute inverse of F_Bk:

   R_B ← rotation part of F_Bk

   t_B ← translation part of F_Bk

   R_B_inv ← R_B^T

   t_B_inv ← -R_B_inv * t_B

   F_Bk_inv ← homogeneous transform with R_B_inv, t_B_inv
3. p_B ← F_Bk_inv * p_tracker
4. d_k ← p_B  // since bone frame = CT frame in PA3
5. Return d_k

**Algorithm description:**

To express the pointer tip in the CT frame, we chaied the rigid transforms estimated for Bodies A and B. The tip location is fixed in Body A coordinates as Atip. For each frame k, we first computed ptracker=FA,k * Atip. We then inverted the rigid transform FB,k by transposing its rotation and applying the inverse translation,and mapping the tracker frame tip into Body B coordinates via pB = (FB,k)^-1 * ptracker. Additionally, because the bone frame and CT frame are defined as identical, the Body B tip coordinates are then taken directly as the sample point dk. This process is repeated per frame to produce all sample points.

**Function: BuildTriangleAccelerator (KD tree and bounding spheres)**

**Inputs:**

- Vertex array V of size (Mv,3)
- Triangle index array T of size (Mt,3)

**Output:**

- Accelerator structure: centroids, radii, maximum radius, KD tree, neighbor count k

**Key Variables:**

- cj: centroid of triangle j
- rj : bounding-sphere radius of triangle j
- tree: root node of custom KD-tree built over triangle centroids
- max_radius: global maximum of {rj}

**Psuedocode:**

BuildTriangleAccelerator(V, T, k_default):

1. M_t ← number of triangles in T
2. Allocate arrays centroids[ 0..M_t-1 ], radii[ 0..M_t-1 ]

3. For each triangle index j from 0 to M_t-1:

(i1, i2, i3) ← T[ j ]

a ← V[ i1 ]; b ← V[ i2 ]; c ← V [ i3 ]

c_j ← (a + b + c) / 3

centroids[ j ] ← c_j

r_j ← max( || a - c_j ||, || b - c_j ||, || c - c_j|| )

radii[ j ] ← r_j

4. max_radius ← max_j radii[ j ]

5. If M_t > 0:

// Build custom KD-tree on centroids

tree ← KDTreeBuild (centroids, indices = [0..M_t-1], depth = 0)

// Where KDTreeBuilld is our own KD-tree constructor, described in its own function block below

Else:

tree ← None

6. accel ← { "tree": tree,

"centroids": centroids,

"radii": radii,

"max_radius": max_radius,

"k": max(1, k_default) }

7. Return accel

**Algorithm description:**

To efficiently search for nearest triangles on the CT surface, we precomputed a KD tree based accelerator. For each triangle with vertices from V, we computed its centroid cj as the mean of its vertices and a bounding sphere radius rj as the maximum distance between the centroid and any vertex. All the centroids and radii are stored in dense arrays, and the global maximum radius is also recorded. Using the centroid array, we constructed a custom median split KD tree: starting from all triangle indices, we recursively choose a split axis based on the current

depth, sorted/partitioned the centroids along that axis, picked the median centroid as the node's pivot, and then recursively build left and right child nodes on the two subsets. The root node of this tree, along with the centroid array, radius array, maximum radius, and the tunable parameter kk (initial neighbor count), are returned as a single reusable structure. This accelerator reduces the number of triangles examined in the closest point computation while keeping the entire implementation self contained.

**Function: KDTreeKNN (k-nearest neighbor search)**

**Inputs:**

- node: root of the KD tree
- points: centroid array
- q: query point
- k: number of neighbors to return

**Output:**

- List of up to kk triangle indices closest to q (unordered or sorted by distance)

**Key Variables:**

- axis: split dimension at each node
- best: small max-heap or list of (distance, index) pairs

**Pseudocode (concise):**

KDTreeKNN(node, points, q, k, best):

1. If node is None: return
2. idx ← node.idx

   p ← points[idx]

   d_sq ← ||p - q||^2

   Update best with (d_sq, idx), keeping at most k entries

3. axis ← node.axis
   diff ← q[axis] - p[axis]
   first ← node.left if diff < 0 else node.right
   second ← node.right if diff < 0 else node.left
4. KDTreeKNN(first, points, q, k, best)
5. If len(best) < k or diff^2 < max distance in best:
   KDTreeKNN(second, points, q, k, best)

**Algorithm description:**

This routine performs a recursive k nearest neighbor (KNN) search on the custom KD-tree. At each node, it updates a small heap of the current best candidates, then recurses into the subtree that contains the query point and only explores the opposite subtree if the splitting hyperplane is close enough that a nearer neighbor could exist there. The final best set contains up to k triangle indices with centroids closest to the query point.

**Function: KDTreeRadiusSearch (radius query)**

**Inputs:**

- node: root of the KD tree
- points: centroid array
- q: query point
- r: search radius

**Output:**

- List of triangle indices whose centroids lie within distance r of q

**Key Variables:**

- axis: split dimension

- r_sq: squared radius for comparison

**Pseudocode (concise):**

KDTreeRadiusSearch(node, points, q, r_sq, hits):

1. If node is None: return
2. idx ← node.idx
   p ← points[ idx]
   d_sq ← || p - q ||^2
   If d_sq ≤ r_sq:
      Add idx to hits
3. axis ← node.axis
   diff ← q[ axis] - p[axis]
4. If diff ≤ 0:
      KDTreeRadiusSearch(node.left, points, q, r_sq, hits)
      If diff^2 ≤ r_sq:
         KDTreeRadiusSearch(node.right, points, q, r_sq, hits)
   Else:
      KDTreeRadiusSearch(node.right, points, q, r_sq, hits)
      If diff^2 ≤ r_sq:
         KDTreeRadiusSearch(node.left, points, q, r_sq, hits)

**Algorithm description:**

This algorithm performs a radius search on the KD tree, collecting all triangle indices whose centroids lie within a ball of radius r around the query point. The subtrees are then pruned whenever the squared distance from the query to the splitting plane exceeds r2, ensuring that only nodes that could contain in range centroids are visited. The resulting hits list is used as the expanded candidate set in the mesh closest point computation.

**Function: ClosestPointOnTriangle (barycentric projection)**

**Inputs:**

- Query point p
- Triangle vertices a, b, c

**Output:**

- Closest point q on triangle $\triangle$ abc

**Key Variables:**

- Edge vectors: ab = b − a, ac = c − a, etc.
- Barycentric coordinates ( u, v, w ) with u + v + w = 1

**High Level Psuedocode:**

ClosestPointOnTriangle(p, a, b, c):

1. ab ← b - a

   ac ← c - a

   ap ← p - a

2. Compute dot products:

   d1 ← dot(ab, ap)

   d2 ← dot(ac, ap)

3. Check vertex region around a:

   if d1 <= 0 and d2 <= 0:

return a

4. bp ← p - b

d3 ← dot(ab, bp)

d4 ← dot(ac, bp)

5. Check vertex region around b:

if d3 >= 0 and d4 <= d3:

return b

6. vc ← d1*d4 - d3*d2

Check edge region of ab:

if vc <= 0 and d1 >= 0 and d3 <= 0:

v ← d1 / (d1 - d3)

return a + v * ab

7. cp ← p - c

d5 ← dot(ab, cp)

d6 ← dot(ac, cp)

8. Check vertex region around c:

if d6 >= 0 and d5 <= d6:

return c

9. vb ← d5*d2 - d1*d6

Check edge region of ac:

if vb <= 0 and d2 >= 0 and d6 <= 0:

w ← d2 / (d2 - d6)

return a + w * ac

10. va ← d3*d6 - d5*d4

Check edge region of bc:

if va <= 0 and (d4 - d3) >= 0 and (d5 - d6) >= 0:

w ← (d4 - d3) / ((d4 - d3) + (d5 - d6))

return b + w * (c - b)

11. Otherwise, inside face region:

denom ← 1 / (va + vb + vc)

v ← vb * denom

w ← vc * denom

return a + ab * v + ac * w

**Algorithm description:**

The closest point on an individual triangle is computed using a clamped barycentric coordinate method. The point p is first analyzed with respect to the triangle's edges and vertices via a sequence of dot product tests, which partitions space into vertex regions, edge regions, and the interior region. For each region, we either return a vertex, a point on an edge segment, or a barycentric combination of a, b, c in the interior. The returned point q minimizes the Euclidean distance to p, subject to the constraint that q lies on the triangle surface. This procedure forms the geometric core of the mesh wide closest point search.

**Function: KDTreeBuild (custom KD tree construction)**

**Inputs:**

- points: centroid array of shape (Mt,3)
- indices: list of triangle indices to store in this subtree
- depth: curent recursion depth (integer)

**Output:**

- Root node of a KD-tree subtree. Each node stores:
  - an index, idx, into the centroid array
  - a split axis- axis
  - left and right child pointers (or None for leaves)

**Key Variables:**

- axis: split dimension (0 for x, 1 for y, 2 for z), typically, depth % 3
- mid: median index in the sorted indices along axis

**Pseudocode:**

KDTreeBuild(points, indices, depth):

1. If indices is empty:

   return None

2. axis ← depth mod 3

3. Sort indices by points[ i ][ axis ]

4. mid ← len(i ndices) // 2

5. node.idx ← indices[ mid]

   node.axis ← axis

node.point ← points[ node.idx ]  // centroid at this node

6. left_indices ← indices[ 0 : mid ]

   right_indices ← indices[ mid+1 : end]

7. node.left ← KDTreeBuild( points, left_indices, depth + 1)

   node.right ← KDTreeBuild( points, right_indices, depth + 1)

8. Return node

**Algorithm Desciprtion:**

The custom KD-tree is built recursively over the centroid array. At each node, we choose a split axis based on the recursion depth (cycling through x, y, and z), sort the current list of centroid indices along that axis, and select the median index as the node's pivot. This gives a balanced partition of centroids into left and right subtrees. The node stores the chosen index (which references a centroid in the global array), the split axis, and pointers to its left and right child nodes. This process repeats recursively until subsets become empty, at which point we would return None. The resulting tree supports both k-nearest neighbor search and radius search using only this in house implemented data structure.

**Function: ClosestPointOnMesh**

**Inputs:**

- Sample point dk
- Mesh vertices V and triangles T
- Accelerator structure accel (custom KD tree root, centroids, radii, max radius, k)

**Output:**

- Closest surface point ck
- Distance || ck−dk ||

**Key Variables:**

- candidate_indices: initial triangles from KD tree
- best_point: current best closest point
- best_dist_sq: current best squared distance
- neighbors: expanded set from ball query

**Psuedocode:**

ClosestPointOnMesh(d_k, V, T, accel):

1. best_point ← None

   best_dist_sq ← +∞

   considered ← empty set

2. If accel has a valid KD-tree root:

   k ← min(accel["k"], number of triangles)

   candidate_indices ← KDTreeKNN(accel["tree"], accel["centroids"], d_k, k)

   //KDTreeKNN is the k nearest neighbor search that walks the tree, prunes subtrees using the split planes, and returns up to k nearest centroid indices.

   Else:

   candidate_indices ← all triangle indices

3. For each idx in candidate_indices:

   if idx in considered: continue

   considered.add(idx)

   (i1, i2, i3) ← T[ idx ]

a ← V[ i1 ]; b ← V[ i2 ]; c ← V[ i3 ]

q ← ClosestPointOnTriangle(d_k, a, b, c)

dist_sq ← || q - d_k ||^2

if dist_sq < best_dist_sq:

best_dist_sq ← dist_sq

best_point ← q

4.4. If accel has a valid KD-tree root and best_point is not None:

centroids ← accel["centroids"]

radii ← accel["radii"]

max_radius ← accel["max_radius"]

best_dist ← sqrt(best_dist_sq)

search_radius ← best_dist + max_radius

neighbors ← KDTreeRadiusSearch(accel["tree"], centroids, d_k, search_radius)

// KDTreeRadiusSearch is a radius search that collects all centroid indices within a given radius of the query point.

For each idx in neighbors:

if idx in considered: continue

centroid ← centroids[ idx ]

r_j ← radii[ idx ]

if ||centroid - d_k|| - r_j > best_dist:

```
            continue   // bounding sphere selection
        considered.add(idx)
        (i1, i2, i3) ← T[ idx ]
        a ← V[ i1 ]; b ← V[ i2 ]; c ← V[ i3 ]
        q ← ClosestPointOnTriangle(d_k, a, b, c)
        dist_sq ← || q - d_k ||^2
        if dist_sq < best_dist_sq:
            best_dist_sq ← dist_sq
            best_point ← q
5. If best_point is still None:
    // Fallback brute-force:
    For each idx in all triangle indices:
        if idx in considered: continue
        ( i1, i2, i3 ) ← T[ idx ]
        a ← V[ i1 ]; b ← V[ i2 ]; c ← V[ i3 ]
        q ← ClosestPointOnTriangle(d_k, a, b, c)
        dist_sq ← || q - d_k ||^2
        if dist_sq < best_dist_sq:
            best_dist_sq ← dist_sq
            best_point ← q
```

6. Return best_point, sqrt( best_dist_sq )

**Algorithm description:**

To find the closest point on the CT mesh, we combine our custom KD tree neighbor search with exact triangle geometry. The KD tree is first queried for the k nearest triangle centroids to dkdusing a recursive k nearest neighbor search that walks the tree, prunes subtrees whose split planes are too far from the query, and maintains a small heap of the current best candidates. These triangles form an initial candidate set. For each candidate, we compute its closest point using the single triangle barycentric projection and maintain the best squared distance encountered. We then expanded the search by issuing a radius query on the same KD tree: starting from the root, we recursively visited any node whose centroid could lie within a ball of radius equal to the current best distance plus the global maximum triangle radius, and collected all triangle indices whose centroids are within that radius. Each returned triangle undergoes a quick bounding sphere rejection test before performing the exact closest point projection. If no triangles are processed via the KD tree path, we then fall back to evaluating all triangles with brute force to maintain accuracy. The final closest point and Euclidean distance are then returned.

**Function: ComputeMatches (End to end pipeline)**

**Input:**

- Rigid body definitions for A and B (markers and tip)
- CT mesh vertices V and triangles T
- Per frame marker readings (sample frames)

**Output**:

- List of ( dk, ck, error k ) for all frames

**Key Variables:**

- FA,k , FB,k : body to tracker transforms
- dk: tip in CT frame
- ck: closest surface point
- accel: triangle accelerator

**Pseudocode:**

ComputeMatches(bodyA, bodyB, V, T, samples):

1. accel ← BuildTriangleAccelerator(V, T, k_default)
2. results ← empty list
3. For each sample frame k in samples:

// Rigid registration for each body

F_Ak ← RigidRegistration( bodyA.markers, sample_k.markers_a )

F_Bk ← RigidRegistration( bodyB.markers, sample_k.markers_b )

// Pointer tip in CT frame

d_k ← PointerTipTransform( bodyA.tip, F_Ak, F_Bk )

// Closest point on mesh

( c_k, dist ) ← ClosestPointOnMesh( d_k, V, T, accel )

// Recompute / post process distance

dist ← || c_k - d_k ||

if dist < threshold:

dist ← 0.0

else:

dist ← round_to_3_decimals( dist )

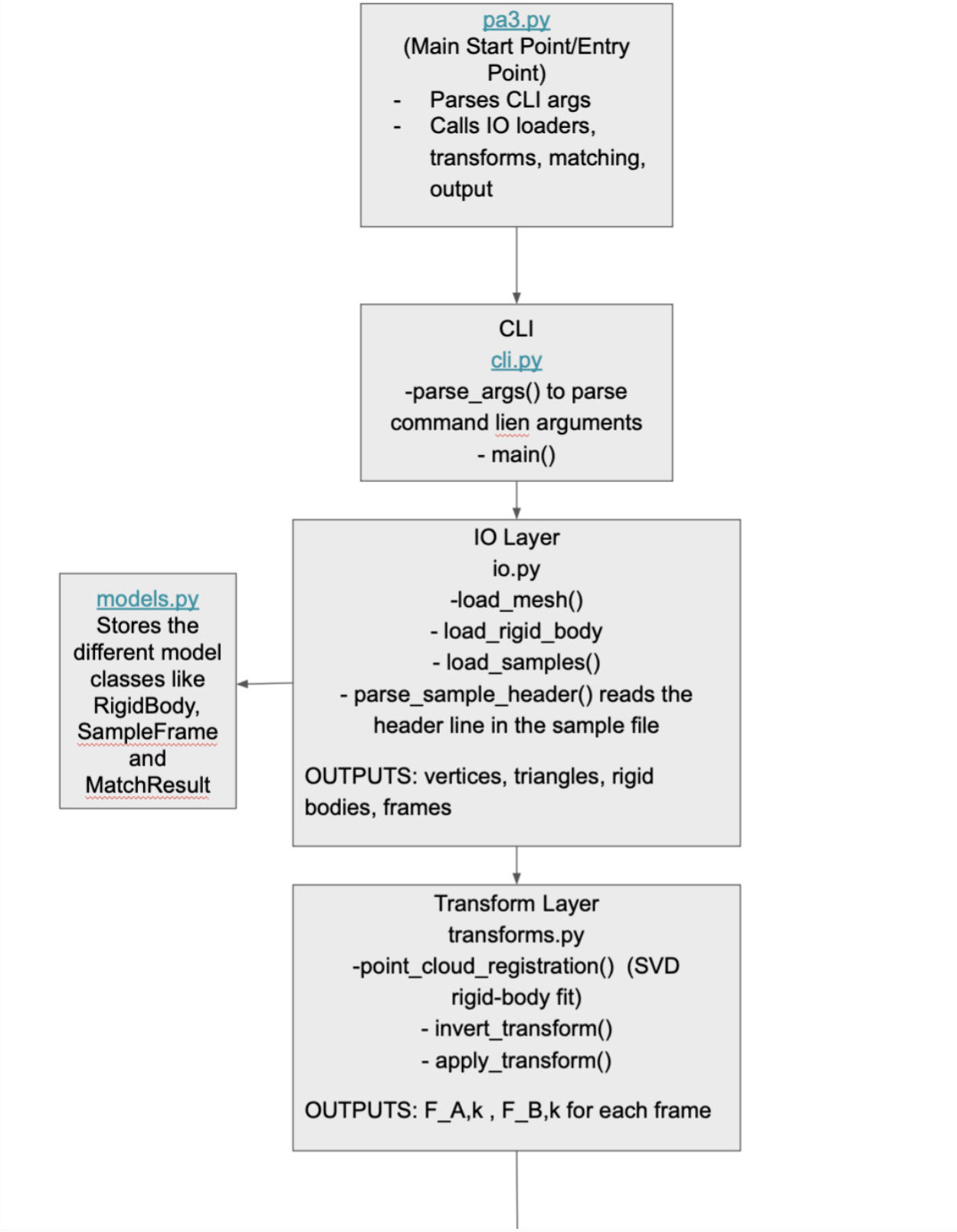
Append (d_k, c_k, dist ) to results

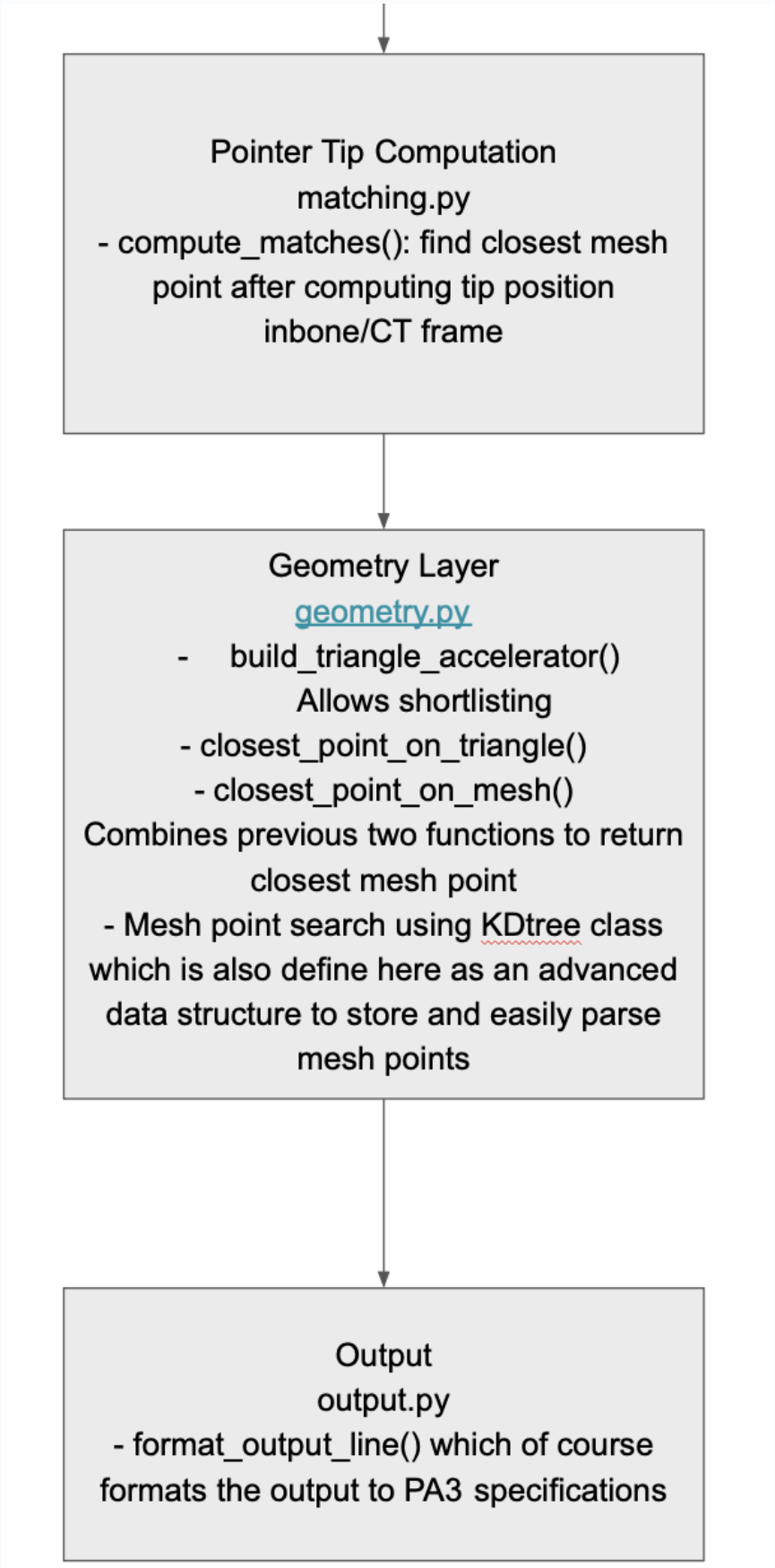
4. Return results

**Algorithmic description:**

The matching pipeline ties together the rigid registration, pointer tip transformation, and mesh closest point algorithms to produce the desired outputs. After loading the body definitions (marker layouts and tip position), sample readings, and the CT surface mesh from disk, we first built the triangle accelerator once for the mesh using our custom KD-tree procedure described above. We then iterated over all sample frames. For each frame, we then applied the SVD based rigid registration algorithm to estimate the pose of Body A and Body B in tracker coordinates, and used the pointer tip transformation chain to map the fixed tip Atip into CT coordinates, giving the sample point dk. We then called the KD-tree accelerated closest point routine to get the nearest point ck on the CT mesh and the corresponding distance between dk and ck. We then recomputed the Euclidean norm ||ck−dk||, snapped extremely small distances below a fixed threshold to exactly zero, and lastly rounded the remaining distances to three decimal places. The program writes one line per frame containing the coordinates of dk (the tip in CT), the coordinates of ck (the closest surface point), and the final scalar distance; together, these lines form the output file and, combined with the mathematical approach, can be used to reproduce the core functions without seeing the source code.

## Overview of Program Structure




Pointer Tip Computation
matching.py
- compute_matches(): find closest mesh point after computing tip position inbone/CT frame
Geometry Layer
geometry.py
- build_triangle_accelerator()
Allows shortlisting
- closest_point_on_triangle()
- closest_point_on_mesh()
Combines previous two functions to return closest mesh point
- Mesh point search using KDtree class which is also define here as an advanced data structure to store and easily parse mesh points
Output
output.py
- format_output_line() which of course formats the output to PA3 specifications

Our python script is organized modularly with each module responsible for a very specific step in the matching process. Ideally this program is defined so that each module handles one clearly defined task

1. pa3.py- This is the main controller which executes the entire pipeline. Thus it is responsible for starting the sequence of parsing command line arguments , loading data and functions, computing rigid body transforms, computing pointer tip position and determining mesh correspondences at a very high level. Actual execution is in other files but this starts the cascade.
2. cli.py- This is what handles the command line interface and uses the standard argparse module in python to do so. It handles the user input validation and then forwards the cleaned file paths back to pa3.py for the next steps.
3. io.py- This module executes input parsing. It contains all the functions for reading the mesh (load_mesh), the rigid body definitions (load_rigidbody), and the sample frames (load_samples) with reading header lines in the sample as a helper function(parse_sample_header)
4. models.py- This defines smaller helper classes, RigidBody, SampleFrame, and MatchResult which bundle the related data from each and make downstream computations easier and consistent.
5. transforms.py- This implements all the mathematical operations describes above. pointcloud_registration() returns the rigid transform using SVD. invert_transform() returns the inverse transform of course and apply_transform() of course applies a given transform to appoint
6. geometry.py- This module provides the triangle geometry utilities and provides ow level geometric operations pursuant to that. closest_point_on_triangle() returns an exact closest point computation and vector operations such as projection, sloping, and barycentric coordinate handling are executed as well. Overall, these routines interact with the matching module to evaluate distances from the pointer tip position to the triangles
7. matching.py- This is what implements the core of the PA3 logic. compute_tip_positions(F_A,F_B,A_tip) returns the $\vec{d}_k$ for the pointer tip transformation to the bone/CT space. The compute_matches(d_k, mesh) then returns the closest point $\vec{c}_k$ and the associated error. The function also used to support a brute force search algorithm for this step but now supports a KDTree implementation for computational efficiency.
8. output.py- This module is the output file writer. It formats and writes the final result in the prescribed format for PA3 and take sin lists of $\vec{d}_k$, $\vec{c}_k$, and errors so that it can be outputted one formatted line at a time.

## Validation

The goal of this section is to really make sure that each stage of the PA3 implementation (rigid body registration, pointer tip transformation, and surface to point matching) performs as expected.

### Validation methodology

Validation was the first step we needed to do to make sure that the program fully worked and executed the steps in the right way and order. This testing was done in two stages

Unit testing

Each core module was tested using unit tests which are contained in the tests folder: Each test file targets one subsystem in the whole modular PA3 implementation, so that we can verify the lower level operations behave as expected before we integrate them together into a full pipeline.

1. I/O testing in test_io.py
   For the I/O module, our unit tests mainly focused on making sure that all of the input files required by PA3 (mesh files, rigid body definitions, and sample readings) were parsed through and loaded correctly and also to make sure that corrupted or incomplete files raised appropriate errors. Specifically our tests verified that:
   a. Valid rigid body, sample, and mesh files, both simple and more complex are passed and that they are loaded into the correct data structures with the expected shape and values
   b. It also makes sure that the parsers are correctly extracting labels/headers, marker counts, and sample frame data amongst the other data provided by the files so that we know that the right data is being selected and shaped.
   c. Finally makes sure that error handling for files works as intended for any empty files, truncated marker lists, malformed headers, inconsistent marker counts, and incomplete sample blocks.

   Overall, the I/O test help ensure that all the downstream geometric and transformation computations begin with valid, well structured input data that is loaded properly, and that there are loud predictable errors when files do not conform to what is expected for an input file from PA3

2. Geometry Testing in test_geometry.py
   The geometry unit tests verify that the central geometric routines that are used during the surface matching behave correctly and consistently in the normal cases of course as well as typical edge-cases. These tests ensure that the algorithms for computing the closest point on a triangle, and thus by extension the closest point on the mesh are mathematically sound since they are used in the full PA3 pipeline. Thus our test focused on confirming that:

a. The closest-point computation on only a single triangle behaves correctly for points located inside, on the edge, on vertices, and outside the triangle.
b. Barycentric coordinate calculations correctly project a point onto the plane of a triangle and identify if the projection lies inside the triangle or outside.
c. The closest-point-on-mesh function correctly loops through triangles in our kdtree structure and selects the one with the minimum Euclidean distance
d. That synthetic meshes with known analytical results return the points and distances that are expected.

Overall these tests ensure that the geometric foundation of the matching phase is correct and reliable before we attempt integrations with the transformation and registration functions of PA3.

3. Transformation Testing in test_transforms.py
   The transformation unit tests help verify the correctness of the Single-Value-Decomposition-based-rigid-body-registration and the associated transformation utilities that are used all throughout PA3 as well. With these tests we can ensure the computed transformations accurately map between the body coordinates and the tracker coordinates, and that inversion and application of transforms work consistently and properly as well. Specifically:
   a. The rigid-body-registration function correctly recovers a known rotation and translation, i.e a known pose, when it is given synthetic point clouds related by a predefined and thus known transform
   b. The implementation also must handle the reflection case properly ensuring that the returned rotation matrix is actually a valid rotation for how the two point clouds are related
   c. We also checked the inverse transform routine by verifying the application of transform to appoint and then the inverse returns the original point within a certain error
   d. Finally we checked that the point transform function behaves consistently across the typical cases, which included transforming synthetic points and real marker sets from dataset A

   All together, these tests helped to validate the numerical stability and mathematical correctness of our rigid-body-transformation logic before we used it to compute pointer tip positions during the actual matching phase.
4. Matching Logic Testing in test_matching.py
   The matching unit tests verify that the end-to-end point matching logic is correctly transforming a pointer-tip position into the CT/bone frame and identifies the nearest point on the mesh to that pointer-tip position. Thus these tests ensure that the integration between the transforms, geometrical routines, and matching

logic behave predictably and properly before we forward with full pipeline testing.

We confirmed that:

a. The testing_matches() function properly applies the correct rigid-body transforms to convert the pointer tip in Body A's coordinate frame to the coordinate frame of the bone/CT
b. The transformed pointer tip is properly matched to the correct triangle when a small synthetic test mesh is used with analytically known nearest points
c. The function returns the match result with the expected structure, numerical properties and ordering which for us would be (tip point, closest point, Euclidian distance)
d. The corner cases are handled to produce stable, consistent results such as the pointer tip lying directly above triangle edges/vertices or near multiple triangles

5. Output Formatting Testing in test_output.py

   The final set of module unit tests focus on verifying that each line in the pA3 output file is written in correct structure and that it also has the correct numerical format. We verified that:

   a. Each of the output lines contains the correct fields in the correct order (pointer-tip coordinates, closest-surface point coordinates, and the Euclidean distance)
   b. All the numeric values are formatted with consistent floating point precision and any rounding behavior. This also includes handling small values that should round to 0.
   c. The formatting function returns the correctly structures strings which match the PA3 specifications to match the administrative constraints of the assignment

The purpose of this next step is integration testing which is done in test_all_ouptut_files.py

After verifying that each individual function worked, we also needed to make sure that the full program worked and that functions/modules communicated with each other properly. For this we used the debug datasets. For each dataset, the computed output file was compared to the corresponding ground truth file given to us. The comparison we did focused on the per-sample Euclidean distance values.

$$||\vec{d}_k - \vec{c}_k|| = \sqrt{(d_x - c_x)^2 + (d_y - c_y)^2 + (d_z - c_z)^2}$$

For each data set we computed the Mean Error, the RMS error, and the Max Error

Mean Error = $\frac{1}{N}\sum_{k} ||\vec{d}_k - \vec{c}_k||(program) - ||\vec{d}_k - \vec{c}_k||(reference)$

RMS Error = $\sqrt{\frac{1}{N}\sum_{k}(||\vec{d}_k - \vec{c}_k||(program) - ||\vec{d}_k - \vec{c}_k||(reference))^2}$

Datasets were marked as **Pass** if RMS error < 0.01 mm, consistent with expected floating-point precision.

**Quantitative Evidence for Results**

| Dataset | # Samples | Mean Error (mm) | RMS Error (mm) | Max Error (mm) | Validation Status |
|---|---|---|---|---|---|
| A | 15 | 0 | 0 | 0 | Pass |
| B | 15 | 0.0030 | 0.0038 | 0.0080 | Pass |
| C | 15 | 0.0023 | 0.0032 | 0.0070 | Pass |
| D | 15 | 0.0028 | 0.0033 | 0.0070 | Pass |
| E | 15 | 0.0027 | 0.0041 | 0.0100 | Pass |
| F | 15 | 0.0036 | 0.0052 | 0.0130 | Pass |

All datasets passed the validation criteria we set forward of the RMS error being below 0.01mm and so this confirmed to us that the computed rigid-body registration, transformation, and the surface matching behave consistently with the reference outputs. The small non-zero differences are most likely attributable to small variations in noise and floating point error in the computed calculations and so given this we don't think that there is any algorithmic incorrectness.

Next we evaluate the unknowns. For the unknown datasets that were given to us (G, H, J), no reference "ground truth" output files were provided to us, so we validated our implementation by checking our computed data for internal consistency and numerical stability. Specifically, we verified that all computed pointer tip positions, closest-point locations, and correspondence errors were within reasonable anatomical ranges (no anatomically incorrect outliers) and that no NaNs, infinities, or extreme outliers appeared in the results. We also confirmed that the distances stayed plausible relative to the size of the CT mesh and that the KD-tree search and brute-force search (which was spot-checked on several frames) produced consistent nearest-neighbor behavior for that computational advantage. Although we can't quantify accuracy without ground-truth files, these checks

indicate that the algorithm behaved correctly and produced stable, non-degenerate outputs for all unknown datasets so at the very least should be applicable in real world scenarios without massive error

**Unknown Output Results:**
The unknown output files (pa3-G-Unknown-Output.txt, pa3-H-Unknown-Output.txt, and pa3-J-Unknown-Output.txt) contain the computed closest point correspondences for the unknown test datasets, each containing 20 sample frames. These outputs were generated using the implemented SVD based rigid registration, custom KD tree accelerated mesh queries, and barycentric projection algorithms, producing sample point coordinates, closest mesh surface points, and Euclidean distances for each frame.

pa3-G-Unknown-Output.txt U

output > pa3-G-Unknown-Output.txt

```
20 pa3-G-Unknown-Output.txt 0
  -11.33   -27.57   -20.11        -10.66   -28.86   -19.76    1.495
   32.48    12.85     3.57         35.25    15.64     3.89    3.944
  -45.95   -12.07   -24.95        -44.08   -11.83   -25.16    1.899
    0.01   -13.79     7.89          0.22   -11.62     7.59    2.203
   25.91    -4.55    15.59         26.09    -5.03    15.72    0.520
   11.28   -10.38    12.58         11.09    -9.68    12.33    0.765
   -3.84   -14.30     0.87         -3.78   -13.24     0.42    1.157
   18.47     0.39    46.12         20.16    -1.50    46.66    2.592
  -14.09    -5.48   -49.79        -14.16    -5.99   -47.72    2.132
  -24.49   -25.88   -12.61        -24.50   -26.84   -11.60    1.392
  -44.76   -15.29   -22.15        -42.97   -14.97   -22.54    1.857
   11.35    20.47    23.64         11.43    23.42    24.23    3.014
  -32.10   -31.23   -23.23        -31.32   -30.00   -23.43    1.471
   -2.27    -0.22    61.04         -2.28    -0.22    63.43    2.389
    2.46   -14.73   -25.19          2.59   -14.78   -25.27    0.160
    4.42    18.91    33.52          4.63    21.71    34.07    2.863
   -1.53    -5.82   -29.22         -0.99    -5.67   -29.67    0.715
  -33.92   -29.95   -22.13        -33.11   -28.67   -22.40    1.543
  -37.66     3.29   -19.48        -37.51     3.13   -19.55    0.237
   35.24    -3.29   -10.87         35.89    -3.89   -10.63    0.911

```

output > pa3-J-Unknown-Output.txt

```
20 pa3-J-Unknown-Output.txt 0
  -22.89   -16.34    -7.61      -22.90   -16.39    -6.12    1.490
  -32.48     3.32   -31.06      -34.82     6.14   -32.10    3.818
   19.52    23.06     6.06       20.46    25.37     6.71    2.574
  -36.41    -3.11   -23.13      -41.04    -0.59   -22.17    5.365
   31.22     2.84   -29.92       30.16     2.73   -28.73    1.599
   -4.19    13.82     8.25       -4.67    14.37     7.71    0.908
  -38.70    -5.68   -30.93      -41.68    -6.84   -32.77    3.689
  -30.10   -32.85   -21.93      -29.15   -30.88   -22.43    2.243
   13.75    22.30    -7.39       11.65    24.33    -8.21    3.033
    8.99    -8.71    52.88        8.84    -7.03    52.90    1.685
   16.51    22.50    12.06       17.29    24.86    12.65    2.565
  -21.80     6.90   -27.43      -22.29    12.21   -28.28    5.405
  -31.48   -29.82   -18.98      -31.12   -29.18   -19.27    0.788
   -4.05    -5.92    37.58       -3.49    -5.35    37.48    0.799
   27.14     7.92    28.56       26.78     7.89    28.45    0.384
    8.39    -8.60    62.53        8.28    -7.36    62.12    1.316
   33.71     5.28   -27.38       31.88     5.42   -26.33    2.114
   14.35    -8.67    34.43       13.77    -6.03    34.20    2.716
   -6.84     8.47     9.81       -8.85    10.59     9.15    2.997
   15.14     7.84    62.28       15.12     7.83    63.15    0.869
```

pa3-H-Unknown-Output.txt U

output > pa3-H-Unknown-Output.txt

```
20 pa3-H-Unknown-Output.txt 0
  -30.72    -3.04   -43.00      -31.56    -2.35   -44.45    1.816
   -4.01    10.24    58.27       -5.75    11.11    58.12    1.944
  -36.50   -19.51   -10.24      -35.89   -18.79   -11.21    1.359
    0.92    -6.40    44.26        0.78    -6.72    44.28    0.347
  -17.43   -24.22   -45.49      -17.78   -23.37   -44.57    1.303
   19.57    17.26    56.32       18.33    16.30    56.27    1.566
   -9.69    -1.49    15.42      -10.22    -1.68    15.62    0.599
  -13.06    -5.28     5.53      -13.28    -5.39     5.66    0.276
  -31.05   -28.14   -13.20      -29.93   -26.94   -14.10    1.871
   25.97   -14.65   -20.76       25.43   -12.59   -19.54    2.452
   27.96    20.97    -9.66       28.63    22.01    -9.72    1.240
   24.76    22.55     5.51       25.27    23.65     5.74    1.231
    1.19    -4.62    48.53        0.33    -6.34    48.52    1.929
    4.19    22.39    21.49        4.24    23.83    21.70    1.457
  -21.44   -31.12   -11.71      -21.07   -28.65   -13.39    3.010
   -2.46     5.61   -15.45       -2.32     8.41   -16.18    2.894
  -35.92    -4.09   -13.66      -37.13    -3.28   -12.48    1.875
  -32.31    -4.54   -42.92      -33.09    -3.90   -44.30    1.705
   22.32     2.57    52.40       21.69     2.83    52.30    0.689
   -5.96    22.83    31.83       -5.64    23.85    32.35    1.191
```

**Interpretation and Discussion**

These results validate the correctness and numerical. The negligible differences between computed and reference outputs indicate that:

1. The Single Value Decomposition based rigid registration accurately recovers rigid body transformations
2. The pointer tip mapping to the bone, and thus CT, coordinate frame is consistent across frames
3. The closest point search correctly identifies the nearest mesh vertices and surfaces

Now even though the errors were within our predefined range there still was error and so it makes sense to discuss possible sources. One source of error is of course floating point arithmetic error which can propagate in matrix multiplications and distance calculations. Another source of error could have simply been an internal attribute of the mesh. It is not expected for the mesh to be homomorphic in that the centroids of adjacent triangles are equidistant from each other and so in that way there are different sized triangles that make up the mesh. This also introduces an intrinsic error because depending on where the pointer touches, larger triangles will obviously have the capacity to contribute more error in certain configurations and so in a way the resolution of the mesh is also a source of error. However, all this considered, the error was still within tolerance and so it is really attributes of the input data, and not purely the algorithm methodology, that contribute to slight inconsistencies in the output.

**Work Division:**

Rohit and Sahana collaborated using pair programming throughout development: while one drove (typed), the other navigated (planned, reviewed, and debugged), switching roles regularly. Both contributed to algorithm design, implementation, unit testing, and dataset validation. For the report, we drafted sections in parallel (Mathematical Approach, Algorithmic Approach, Program Structure/Diagrams, Results), then jointly reconciled style and content in a final pass. Contributions were comparable in scope and effort.

**Citations:**


- Course slides / problem statement

  Johns Hopkins University. *600.445/600.645 Computer Integrated Surgery I — Programming Assignment 3 Slides and Specification*. Fall 2025. (Provides the official PA3 problem description, datasets, and algorithmic expectations.)

- Project template

  Killeen, Benjamin D. “cispa: Template for CIS I programming assignments at Johns Hopkins.” GitHub, 2022. <https://github.com/benjamindkilleen/cispa>.

- Numerical linear algebra (SVD and registration math)

  Golub, Gene H., and Charles F. Van Loan. *Matrix Computations*, 4th ed. Johns Hopkins University Press, 2013. (Covers the SVD-based absolute orientation algorithm used for rigid-body registration.)

- Pytest (test runner; dev-only per requirements.txt/setup.py)

  Krekel, Holger, et al. *pytest Documentation*. Version 9.0.0, pytest-dev, 2024. <https://docs.pytest.org/en/latest/>.

- Mesh closest-point projection fundamentals

  Ericson, Christer. *Real-Time Collision Detection*. CRC Press, 2004. (Standard reference for barycentric triangle tests, edge/vertex region checks, and mesh proximity algorithms.)

- NumPy numerical routines

  Harris, Charles R., et al. “Array programming with NumPy.” *Nature* 585, no. 7825 (2020): 357–362. <https://numpy.org/doc/>. (Primary reference for vectorized operations, centroid computation, and linear algebra helpers used throughout.)

- Provided datasets

  Johns Hopkins University. *PA3 Debug and Unknown Dataset Bundle* (Problem3 mesh, body definitions, sample readings, and reference answer files). Distributed via the CIS I course portal, Fall 2025.