# CIS I - Programming Assignment 4 Report

Shreya Terala & Yvonne Zhang

January 16, 2026

## 1 Introduction

This report will focus on the tasks presented in Assignment 4, including the development of mathematical tools and subroutines for implementing an iterative-closest point (ICP) registration [8]. All programs were written in MATLAB for this project and will be used for all subsequent programming assignments.

## 2 Summary of Problems To Be Solved

The primary goal of this assignment is to implement and use a complete version of the Iterative Closest Point (ICP) algorithm for registering a pointer to a 3D surface, and ultimately aligning it with a mesh of the surface captured in CT coordinates. The setup includes:

- A 3D surface represented as a triangular mesh, with vertices defined in CT space.
- Two rigid bodies, **A** and **B**, each defined by LED marker positions and pointer tips in body coordinates. The tip of body **B** is rigidly attached to the bone in an unknown orientation, while body **A** serves as a pointer to sample points on the bone surface.
- A set of sample readings that provide the positions of the LED markers relative to an optical tracker for each sample frame.

### 2.1 ICP Registration

This process allows the registration of the pointer tip to the CT surface, enabling accurate mapping of sample points into CT coordinates and forming the basis for precise stereotactic navigation.

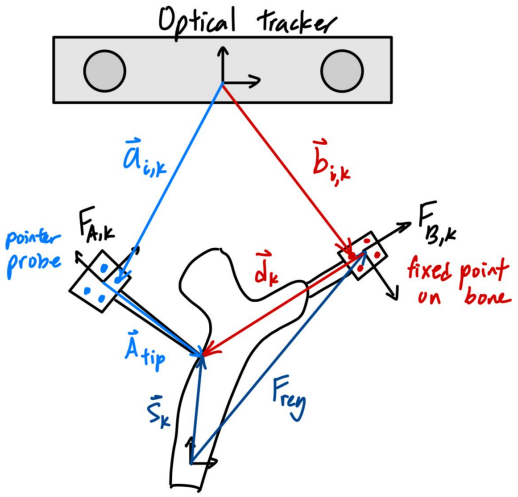


Figure 1: ICP Registration System

The following defined variables are important for the problems solved in this assignment.

$F_A$ - Pointer probe frame
$F_B$ - Rigid frame screwed into the bone with unknown orientation
$F_{reg}$ - Frame transformation from sample frame to rigid body frame $F_B$
$A_{tip}$ - Tip of pointer
$B_{tip}$ - Tip of rigid body

$a_{i,k}$ - LED markers for the pointer
$b_{i,k}$ - LED markers for rigid body
$d_k$ - Pointer tip with respect to $F_{B,k}$
$s_k$ - Sample probe pointer tip data

# 3 Mathematical Approach

## 3.1 3D Point Cloud Registration Algorithm

A primary function, "**point_cloud_registration.m**", was developed to register the correspondence between the two 3D point sets. This function accepts two sets of corresponding points and returns the aligning transformation matrix using Arun's method [5]. The output is the rotation matrix and the translation vector that form the frame transformation matrix.

The following will use the lower case $x_i$ for the calibration markers vector and $X_i$ for the positions of the markers. This algorithm was developed in PA1 and is used to find $d_k$ for every set of $a_{i,k}$ and $b_{i,k}$, and $A_{markers}$ and $B_{markers}$ respectively from the body data files in PA3. See Section 4.3.1 for further details on finding $d_k$. The following is the mapping between $x_i$ and $X_i$ for A and B data points.

First, the inputs from the data reading function is allocated to variables $x_i$ and $X_i$ for calibration markers and calibration frames readings respectively. The size of both markers and frames are also stored. Arun's method is used to find the optimal frame transformation in closed form to estimate the frame transformation that best aligns the set of known $x_i$ object points with corresponding 3D points $X_i$ for each frame. For each frame, the following are solved to form the $SE(3)$ frame transformation [5]:

$$X_i \approx Rx_i + t, \quad R \in SO(3), \quad t \in R^3 \tag{1}$$

for $i = 1, ..., N$ in each frame.

Next, the translation is removed for centroid alignment to isolate the rotation from the translation effects of the transformation. The centroids of both point sets are calculated:

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i, \quad \bar{X} = \frac{1}{N}\sum_{i=1}^{N} X_i \tag{2}$$

The centroids are subtracted from each data point to obtain the centered coordinates:

$$x_i' = x_i - \bar{x}, \quad X_i' = X_i - \bar{X} \tag{3}$$

The optimal rotation is determined by minimizing the following equation with translation effects removed [5]:

$$\min_{R \in \mathrm{SO}(3)} \sum_{i=1}^{N} \left\| X_i' - Rx_i' \right\|^2 \tag{4}$$

By expanding the squared norm:

$$\left\| X_i' - Rx_i' \right\|^2 = \left(X_i' - Rx_i'\right)^T \left(X_i' - Rx_i'\right) = \left\| X_i' \right\|^2 + \left\| Rx_i' \right\|^2 - 2\left(X_i'\right)^T Rx_i' \tag{5}$$

where rotation has no effect inside a norm for $||Rx_i'||^2 = ||x_i'||^2$, and $||X_i'||^2$ and $||x_i'||^2$ are independent of rotation, then:

$$\min_R \sum_i \left\| X_i' - Rx_i' \right\|^2 \longleftrightarrow \max_R \sum_i \left(X_i'\right)^T Rx_i' \tag{6}$$

which identifies the covariance matrix that is defined as,

$$H = \sum_{i=1}^{N} (x_i') (X_i')^\top = (x_i') \left(X_i'^\top\right) \tag{7}$$

The covariance matrix H is the correlation between the two centered point sets by encapsulating how the two sets are aligned in orientation. Its singular value decomposition can be found using the svd function for the following equation:

$$H = USV^\top \tag{8}$$

The optimal rotation is solved by utilizing the diagonal matrix for the weight of the diagonal entries of the covariance matrix H, therefore the weights of the mapping from $x_i$ to $X_i$. The diagonal is therefore the trace of (7) and by equating to (8) [6],

$$tr((X')^T R x') = tr(RH) = tr(RUSV^\top) = tr(SV^\top RU) \tag{9}$$

Let $V^\top RU = M$, where $M$ is an orthogonal matrix for orthogonal $V^\top, R$ and $U$, and $tr(SM)$ is maximized if m-components equal to 1. Therefore, $M$ is the identity matrix and the optimal rotation is given by the svd orthogonal matrices:

$$M = I = V^\top RU \rightarrow V = RU \rightarrow R = VU^\top \tag{10}$$

Finally, the translation is determined by aligning the centroids such that:

$$t = \bar{X} - R\bar{x} \tag{11}$$

The rotations and translations of each frame is stored in "rotationMatrix_all" and "translationVector_all" respectively. The function outputs "rotationMatrix" and "translationVector" for each X dataset (i.e. $d_k$).

### 3.1.1 Zero Singular Value Case

In practice, there are degenerate cases when the covariance matrix H is rank-deficient ($det(R) \approx 0$)). This occurs if the points are collinear or coplanar, therefore reducing the dataset dimension. In these cases, H carries at least one singular value of zero or numerically very small; therefore, the corresponding rotation is under-determined.

The algorithm detects the singular values from the SVD as follows:

```
if min(diag(S)) < 1e-12
    warning('Frame %d: Covariance matrix near-singular / zero
        singular value detected, therefore best-fit rotation will
        be used.', f);
    % R=V*U' gives best-fit rotation even with zero singular value
end
```

Even though the rotation is not fully determined along the degenerate axes, the least-squares best-fit rotation is still found.

### 3.1.2 Reflection Case

In addition, there are reflection cases ($det(R) < 0$) handled by using the following:

```
if det(R) < 0
    V(:,3) = -V(:,3); % flip last column of V to correct
        reflection
    R = V * U'; % resolve for R
    warning('Frame %d: Reflection detected and corrected.', f);
end
```

SVD orders the singular values along its diagonal from largest to smallest. Therefore, the last column is often the least constrained by the data and has minimal effects on the axes' alignment. Flipping the last column of V therefore corrects the determinant to positive while preserving the main alignment.

These two pieces of code are used in the "point_cloud_registration.m" function to mediate corner cases.

### 3.2 Finding Closest Point on Triangle

Given a query point $\mathbf{P} \in \mathbb{R}^3$ and a triangle defined by vertices $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}^3$, the closest point $\mathbf{P}_{\text{closest}}$ on the triangle can be written $\mathbf{P}$ projected onto the triangle's plane:

$$\mathbf{P}_{\text{proj}} = u\mathbf{A} + v\mathbf{B} + w\mathbf{C}, \qquad \text{with } u + v + w = 1. \tag{12}$$

Geometrically, the barycentric weights $(u, v, w)$ are proportional to the areas of the sub-triangles formed by $\mathbf{P}$ and the triangle's edges. For instance:

$$u = \frac{\text{Area}(\triangle PBC)}{\text{Area}(\triangle ABC)}.$$

The area of a triangle defined by vectors $\mathbf{x}$ and $\mathbf{y}$ is given by:

$$\text{Area} = \frac{1}{2}\|\mathbf{x} \times \mathbf{y}\|.$$

Hence, using the cross product form, the equations for the weights can be simplified to the following [3]:

$$u = \frac{\|(\mathbf{C}-\mathbf{B}) \times (\mathbf{P}-\mathbf{B})\|}{\|(\mathbf{B}-\mathbf{A}) \times (\mathbf{C}-\mathbf{A})\|} = \frac{\|\mathbf{A}\|}{\|\mathbf{B}\|} = \frac{\|\mathbf{A}\| \cdot \|\mathbf{B}\|}{\|\mathbf{B}\| \cdot \|\mathbf{B}\|} = \frac{((\mathbf{C}-\mathbf{B}) \times (\mathbf{P}-\mathbf{B})) \cdot ((\mathbf{B}-\mathbf{A}) \times (\mathbf{C}-\mathbf{A}))}{((\mathbf{B}-\mathbf{A}) \times (\mathbf{C}-\mathbf{A})) \cdot ((\mathbf{B}-\mathbf{A}) \times (\mathbf{C}-\mathbf{A}))}, \tag{13}$$

where $cos(\theta)\|A\|\|B\| = \|A\| \cdot \|B\| \rightarrow (\|A\| \cdot \|B\|)^2 = A \cdot B$ since $\theta = 0$ due to $A$ and $B$ being collinear. Hence,

$$v = \frac{((\mathbf{C}-\mathbf{A}) \times (\mathbf{P}-\mathbf{C})) \cdot ((\mathbf{B}-\mathbf{A}) \times (\mathbf{C}-\mathbf{A}))}{\|(\mathbf{B}-\mathbf{A}) \times (\mathbf{C}-\mathbf{A})\|^2}, \tag{14}$$

and, by rearranging equation 12,

$$w = 1 - u - v. \tag{15}$$

After calculating the weights, the projected point lies inside the triangle if:

$$u \geq 0, \quad v \geq 0, \quad w \geq 0.$$

If this condition is true, then:

$$\mathbf{P}_{\text{closest}} = \mathbf{P}_{\text{proj}}.$$

If any barycentric weight is negative, the projection falls outside the triangle. Then the closest point is found on the closest edge:

- If $u < 0$, use edge $\overline{\mathbf{BC}}$.
- If $v < 0$, use edge $\overline{\mathbf{CA}}$.
- If $w < 0$, use edge $\overline{\mathbf{AB}}$.

The closest point to each edge is computed using the segment projection formula described in Section 3.2.1. Among the candidate edge points, the closest point to $\mathbf{P}$ is selected:

$$\mathbf{P}_{\text{closest}} = \arg \min_{\mathbf{Q} \in \{\text{edge projections}\}} \|\mathbf{P} - \mathbf{Q}\|^2.$$

Thus, $\mathbf{P}_{\text{closest}}$ captures the closest point to the triangle regardless of whether that is a point inside the triangle or on the edge of the triangle.

### 3.2.1 Finding the Closest Point on a Segment

Given a point $\mathbf{c} \in \mathbb{R}^3$ and a line segment defined by its endpoints $\mathbf{a}, \mathbf{b} \in \mathbb{R}^3$, the goal is to find the closest point $\mathbf{c}_{\text{closest}}$ on the segment $\overline{\mathbf{ab}}$. The scalar projection of the vector $(\mathbf{c} - \mathbf{a})$ onto the segment direction $(\mathbf{b} - \mathbf{a})$ is computed using the following equation:

$$\lambda = \frac{(\mathbf{c} - \mathbf{a}) \cdot (\mathbf{b} - \mathbf{a})}{(\mathbf{b} - \mathbf{a}) \cdot (\mathbf{b} - \mathbf{a})}. \tag{16}$$

The parameter $\lambda$ represents where the perpendicular projection of $\mathbf{c}$ falls relative to the line defined by $\mathbf{a}$ and $\mathbf{b}$:

$$\lambda = \begin{cases} 0 & \text{at point } \mathbf{a}, \\ 1 & \text{at point } \mathbf{b}, \\ (0,1) & \text{within the segment.} \end{cases}$$

Since the segment is finite, it needs to be restricted by $\lambda$ to the range $[0, 1]$:

$$\lambda_{\text{seg}} = \max(0, \min(\lambda, 1)). \tag{17}$$

Finally, the closest point on the segment is obtained as a convex combination of the endpoints:

$$\mathbf{c}_{\text{closest}} = \mathbf{a} + \lambda_{\text{seg}} (\mathbf{b} - \mathbf{a}). \tag{18}$$

Geometrically, this operation projects $\mathbf{c}$ onto the infinite line through $\mathbf{a}$ and $\mathbf{b}$, then constrains the result to remain on the line segment between $\mathbf{a}$ and $\mathbf{b}$. If the projection lies beyond either endpoint, the closest point becomes that endpoint.

## 3.3 Linear Search for Closest Point on Mesh

Given query points $\mathbf{s}_k \in \mathbb{R}^3$, the closest point on the triangular mesh needs to be identified. For each sample point $\mathbf{s}_k$, the following must be found:

$$\mathbf{c}_k = \arg\min_{\mathbf{p} \in \mathcal{M}} \|\mathbf{s}_k - \mathbf{p}\|^2, \tag{19}$$

where $\mathcal{M}$ is the surface mesh composed of all triangles:

$$\mathcal{M} = \bigcup_{t=1}^{N_{\text{tri}}} \triangle(\mathbf{A}_t, \mathbf{B}_t, \mathbf{C}_t),$$

and each triangle $t$ is defined by its vertices

$$\mathbf{A}_t = \mathbf{V}_{T_{t,1}}, \quad \mathbf{B}_t = \mathbf{V}_{T_{t,2}}, \quad \mathbf{C}_t = \mathbf{V}_{T_{t,3}}.$$

For a given triangle $\triangle(\mathbf{A}_t, \mathbf{B}_t, \mathbf{C}_t)$, we compute the closest point $\mathbf{p}_{k,t}$ on the triangle to $\mathbf{s}_k$ using the *closest-triangle-point* formulation (see Section 3.2):

$$\mathbf{p}_{k,t} = \text{Proj}_{\triangle(\mathbf{A}_t, \mathbf{B}_t, \mathbf{C}_t)}(\mathbf{s}_k). \tag{20}$$

This step returns the orthogonal projection of $\mathbf{s}_k$ onto the triangle (or the nearest edge/vertex if the projection falls outside the triangle). For each triangle $t$, the squared Euclidean distance is computed as:

$$d_{k,t}^2 = \|\mathbf{s}_k - \mathbf{p}_{k,t}\|^2. \tag{21}$$

The closest point on the entire mesh for $\mathbf{s}_k$ is then found by selecting the triangle that minimizes the distance:

$$t_k^* = \arg\min_{t \in \{1, \ldots, N_{\text{tri}}\}} d_{k,t}^2, \tag{22}$$

and

$$\mathbf{c}_k = \mathbf{p}_{k,t_k^*}. \tag{23}$$

This linear search approach will always identify the closest mesh point, but is computationally expensive for large meshes.

### 3.4 Covariance Tree Search for Closest Point on Mesh

The covariance tree search achieves the same goal as the linear search but utilizes the geometric structure of the mesh to accelerate nearest–triangle queries. For each triangle $\tau_t = (i_1, i_2, i_3) \in \mathcal{T}$ with vertices

$$\mathbf{A}_t = \mathbf{v}_{i_1}, \qquad \mathbf{B}_t = \mathbf{v}_{i_2}, \qquad \mathbf{C}_t = \mathbf{v}_{i_3},$$

the centroid is computed as

$$\mathbf{g}_t = \frac{1}{3}(\mathbf{A}_t + \mathbf{B}_t + \mathbf{C}_t), \qquad t = 1, \ldots, N_{\text{tri}}. \tag{24}$$

Collecting all centroids gives

$$\mathcal{G} = \{\mathbf{g}_1, \ldots, \mathbf{g}_{N_{\text{tri}}}\}.$$

A covariance tree is constructed over these centroids following Section 3.4.1. Each node of the tree stores an *oriented bounding box (OBB)* that contains all triangle vertices associated with the centroids assigned to that node. In the local covariance frame of the node, each triangle vertex $\mathbf{v}$ satisfies

$$\mathbf{LB} \leq \mathbf{v}^{\text{local}} \leq \mathbf{UB}.$$

Once the tree is constructed, each query point $\mathbf{s}_k$ is processed using a recursive OBB-guided nearest–neighbor search:

$$(\mathbf{g}^*,\, d^*,\, t^*) = \text{CovTreeQueryBB}(\text{tree},\, \mathbf{s}_k,\, \mathcal{G},\, d_{\min},\, t_{\min}), \tag{25}$$

where

- $\mathbf{g}^*$ is the centroid of the closest triangle,
- $d^* = \|\mathbf{s}_k - \mathbf{g}^*\|_2$,
- $t^*$ is the index of the closest triangle.

If no valid triangle is found, the closest point is undefined:

$$\mathbf{c}_k = [\text{NaN},\, \text{NaN},\, \text{NaN}].$$

For the closest triangle $\tau_{t^*}$, the exact closest point $\mathbf{c}_k$ is computed using the closest–point-on-triangle method in Section 3.2, projecting the query onto the triangle plane and clamping to edges or vertices as needed.

#### 3.4.1 Constructing the Covariance Tree

The covariance tree is built by recursively partitioning the centroid set into a binary tree based on covariance alignment. Each node stores:

$$\mathbf{R},\ \mathbf{C}_{\text{cov}},\ \mathbf{LB},\ \mathbf{UB},\ \mathcal{P}_{\text{node}},\ \text{and child pointers.}$$

At each node, the points are transformed into a local covariance frame using the process in Section 3.4.2. This frame transformation is applied to all points in the node:

$$\mathbf{p}_i^{\text{local}} = (\mathbf{p}_i - \mathbf{C}_{\text{cov}})^\top \mathbf{R}. \tag{26}$$

For every centroid in the node, all of its triangle's vertices are also transformed:

$$\mathbf{v}^{\text{local}} = (\mathbf{v} - \mathbf{C}_{\text{cov}})^\top \mathbf{R}.$$

From there, a bounding box enclosing all points in the node are defined by the bounds:

$$\mathbf{LB} = \min_{\mathbf{v}} \mathbf{v}^{\text{local}}, \qquad \mathbf{UB} = \max_{\mathbf{v}} \mathbf{v}^{\text{local}}. \tag{27}$$

If

$$N \leq N_{\max},$$

the node becomes a leaf and recursion stops. Otherwise, the node is split along the first principal axis $\mathbf{r}_1$:

$$\mathcal{P}_{\text{left}} = \{\mathbf{p}_i : (\mathbf{p}_i - \mathbf{C}_{\text{cov}})^\top \mathbf{r}_1 < 0\}, \tag{28}$$

$$\mathcal{P}_{\text{right}} = \{\mathbf{p}_i : (\mathbf{p}_i - \mathbf{C}_{\text{cov}})^\top \mathbf{r}_1 \geq 0\}. \tag{29}$$

If either subset is empty, the node becomes a leaf to prevent degeneracy. The two child nodes are constructed recursively:

$$\text{node.left} = \text{ConstructCovTreeBB}(\mathcal{P}_{\text{left}}),$$
$$\text{node.right} = \text{ConstructCovTreeBB}(\mathcal{P}_{\text{right}}).$$

This structure is then used in the previously described search approach and enables more efficient nearest-point queries and hierarchical geometric reasoning than the linear search method in Section 3.3

### 3.4.2 Computing the Covariance Frame

Given a set of points, a local frame $F = (\mathbf{R}, \mathbf{C})$ consisting of a rotation matrix $\mathbf{R} \in SO(3)$ and a centroid $\mathbf{C} \in \mathbb{R}^3$ is found to redefine the points in a more convenient frame for future calculations.

$$\mathbf{C} = \frac{1}{n_P} \sum_{i=1}^{n_P} \mathbf{P}_i \text{ where } P_i \in \mathbb{R}^3, \quad i = 1, \ldots, n_P \tag{30}$$

For each point, its deviation from the centroid is computed, and the outer products are summed to construct the scatter covariance matrix:

$$\mathbf{A} = \sum_{i=1}^{n_P} (\mathbf{P}_i - \mathbf{C})(\mathbf{P}_i - \mathbf{C})^T \tag{31}$$

The matrix $\mathbf{A} \in \mathbb{R}^{3\times3}$ is symmetric and positive semi-definite. Optionally, one may normalize by $n_P - 1$ to obtain the sample covariance matrix. The principal axes of the point distribution are obtained by eigen-decomposition of $\mathbf{A}$:

$$\mathbf{A} = \mathbf{R}\,\mathbf{\Lambda}\,\mathbf{R}^T \tag{32}$$

where

- $\mathbf{R} = [\mathbf{r}_1\ \mathbf{r}_2\ \mathbf{r}_3]$ contains orthonormal eigenvectors (principal directions),
- $\mathbf{\Lambda} = \text{diag}(\lambda_1, \lambda_2, \lambda_3)$ contains the corresponding eigenvalues, ordered so that $\lambda_1 \geq \lambda_2 \geq \lambda_3 \geq 0$.

The rotation $\mathbf{R}$ aligns the local frame with the principal directions of the point distribution, while the centroid $\mathbf{C}$ specifies its origin. Hence, $F$ transforms the representative mesh data for which the covariance tree is being built into a local frame that originates at the centroid of the mesh.

## 3.5 Standard ICP

The Iterative Closest Point (ICP) algorithm was used in addition to the programs developed in Programming Assignment 3. ICP is a method for rigid registration between two 3D point sets in which, given a source and target scan, the algorithm iteratively alternates between finding closest-point correspondences and estimating the rigid transformation that minimizes the alignment error [4]. The standard ICP formulation relies on point-to-point Euclidean distances and, at each iteration, performs a full 3D point-set registration using the least-squares solution summarized in Section 3.1.

At each iteration, the current estimate of the registration frame $F_{reg} = (R_{reg}, t_{reg})$ is applied to the pointer tip samples $d_k$ to obtain the transformed positions $s_k$. Each $s_k$ pairs with the closest point on the mesh to define the set of correspondence points $c_k$. Using the paired sets $(d_k, c_k)$, the least-squares point cloud registration updates the new registration frame transformations. The residual alignment error is evaluated using the point-to-point distances, repeated until this residual falls below a convergence tolerance.

### 3.6 Generalized ICP

Although standard ICP provides a straightforward point-to-point alignment method, its underlying assumption of isotropic identical uncertainty at every point makes it sensitive to noise, surface anisotropy, and incorrect correspondences [4]. In particular, modeling each correspondence as having equal variance in all directions limits the accuracy of registration on locally planar surfaces, where the geometry is highly directional for our applications [4].

Generalized ICP models each correspondence as arising from two locally planar surface patches, one from the source scan and one from the target scan, each with its own covariance [4]. These covariances encode the tangent and normal directions of the surface, allowing the optimization to penalize residuals more heavily in directions of high certainty along the surface normal and less in directions of high uncertainty along the tangents.

The current estimate of the registration frame is applied to compute the transformed samples $s_k$, closest-point correspondences $c_k$ and surface normals are retrieved from the mesh, and the registration is updated. Instead of solving a least-squares problem, the generalized ICP formulates each correspondence as a Mahalanobis-weighted residual as:

$$d_i = c_k - s_k \tag{33}$$

with an associated covariance:

$$C_i = C_B + R_{reg} C_A R_{reg}^T \tag{34}$$

where $C_A$ models source uncertainty and $C_B$ encodes the locally planar covariance of the mesh surface. These weighted residuals define a quadratic cost function whose minimizer is found by solving the normal equations:

$$H\xi = b, \tag{35}$$

where the Jacobian for each correspondence is

$$J_i = \begin{bmatrix} -\text{skew}(R_{\text{reg}} s_i) & I_3 \end{bmatrix}, \tag{36}$$

linking the twist $\xi \in \mathbb{R}^6$ to the residual $d_i$ through the Mahalanobis weighting

$$W_i = C_i^{-1}.$$

The twist $\xi$ is then applied via a left-multiplicative SE(3) exponential map to update the rotation $R_{\text{reg}}$ and translation $t_{\text{reg}}$, and the process repeats until the norm of $\xi$ falls below a convergence tolerance [4]. This formulation allows the registration to incorporate local surface structure and anisotropic noise, improving accuracy and robustness compared to standard ICP.

## 4 Algorithmic Approach

### 4.1 Cartesian Math Package

MATLAB's Symbolic Math Toolbox was used as the Cartesian math package for 3D points, rotations, and frame transformations [12].

Basic mathematical operations for the matrices were used, such as addition, matrix multiplication, and matrix transpose using single quotes ('). These operations were used throughout the assignment.

From Matlab's core numerical linear algebra libraries, the **svd** function performs a singular value decomposition of matrix H, such that [U,S,V] = svd(H) is used to solve $H = USV^T$ [11]. Singular value decomposition is used to solve for the covariance matrix in its matrix-vector product components. This includes the two orthogonal matrices U and V with columns formed by the left and right singular vectors of H respectively. S is a diagonal matrix containing the singular values of H [2]. This decomposition of the covariance will be utilized to determine the rotation matrix.

The **lsqr** function is used to solve the system of linear equations using the iterative least squares method. It solves the system by approximating $Ax \approx b$ by minimizing the Euclidean norm of the residual of [5]:

$$x = min_x ||b - Ax||^2 \tag{37}$$

The least squares algorithm is an adaptation of the conjugate gradient method for rectangular matrices, where $Ax = b$ produces the same residuals as the conjugate gradient for the normal equation $A^T Ax = A^T b$, without forming the explicit $A^T A$ that results in superior numerical stability [13]. Note, the lsqr function will use a tolerance of $1e-6$, which is sufficiently stable for our use case [13].

## 4.2 Reading Data Files

Algorithms were developed to read the provided data files by extracting the relevant geometric and sample information, including marker coordinates, pointer tip positions and mesh topology.

Three main data parsing functions were implemented: **read_body_data.m**, **read_mesh_data.m**, and **read_sample_data.m**. Each function reads formatted input text files containing numerical data for markers, vertices, and frames. The output of these routines is stored in matrices and arrays organized according to the number of entities (markers, vertices, or triangles) and their spatial dimensionality.

All parsing routines follow a consistent procedure:

1. Open the file and extract structural data from the header.
2. Parse numerical data blocks using MATLAB I/O functions (`fscanf`, `textscan`).
3. Validate dimensions and reshape data into standardized 3D arrays.
4. Close the file and return structured outputs for downstream processing.

**Algorithm 1** General Data Parsing Algorithm

**Input:** Body file, Mesh file, Sample file
**Output:** Parsed marker, mesh, and frame data structures

**for** each input file $f_i$ in {body, mesh, sample} **do**
    fid ← `fopen`($f_i$, 'r')
    **if** fid == -1 **then**
        **error:** Unable to open file
    **end if**
    Read header line → extract counts (e.g., $N_{\text{markers}}$, $N_{\text{vertices}}$, $N_{\text{samp}}$)
    Parse remaining numerical data using `fscanf` or `textscan`
    Validate data size against expected dimensions
    **if** $f_i$ == body file **then**
        Separate marker coordinates ($N_{\text{markers}} \times 3$) and pointer tip ($1 \times 3$)
    **else if** $f_i$ == mesh file **then**
        Extract vertex coordinates ($N_{\text{vertices}} \times 3$)
        Extract triangle connectivity ($N_{\text{triangles}} \times 6$)
    **else if** $f_i$ == sample file **then**
        Reshape marker data into ($N_s \times 3 \times N_{\text{samp}}$) array
    **end if**
    `fclose`(fid)
**end for**

Return all parsed data structures for use in calibration, registration, and tracking algorithms

#### 4.2.1 Body Data Parsing

The function **read_body_data.m** reads the number of markers, their 3D coordinates, and the pointer tip position from a calibration body file. The function outputs:

- $N_{\text{markers}}$: total number of markers,
- **markers**: $N_{\text{markers}} \times 3$ matrix of marker coordinates,
- **pointer**: $1 \times 3$ vector representing the pointer tip position.

This information defines the probe geometry for rigid-body calibration.

#### 4.2.2 Mesh Data Parsing

The function **read_mesh_data.m** reads the mesh geometry used for spatial search and projection algorithms. The function outputs:

- $N_{\text{vertices}}$: number of vertices,
- **vertices**: $N_{\text{vertices}} \times 3$ matrix of vertex coordinates,
- $N_{\text{triangles}}$: number of triangular faces,
- **triangles**: $N_{\text{triangles}} \times 6$ matrix of vertex indices and neighbor relationships.

This structure provides an efficient representation of 3D surfaces for geometric computation and correspondence estimation.

#### 4.2.3 Sample Data Parsing

The function **read_sample_data.m** processes time-varying marker data collected by the tracker. The header specifies the number of markers per frame $N_s$ and the total number of samples $N_{\text{samp}}$. The data are reshaped into:

$$\text{markers} \in \mathbb{R}^{N_s \times 3 \times N_{\text{samp}}}$$

where each slice corresponds to the marker configuration in one frame. The function outputs $N_s$, $N_{\text{samp}}$, and the 3D marker array for frame-wise transformation estimation.

### 4.3 3D Point Set Registration Algorithm

The "**point_cloud_registration.m**" function computes the rigid-body transformation to align two corresponding sets of 3D points based on Arun's method as described mathematically in Section 3.1.

The **svd** function is used to solve for the rotation matrix. The **mean** function is used throughout this algorithm to find the centroids of the markers.

Note that this function is called for finding $d_k$, as described in the next Section.

**Algorithm 2** Point Cloud Registration Using Arun's Method

**Input:** $calmarkers$ (Nx3), $calframereadings$ (Nx3xF)
**Output:** $rotationMatrix$ (3x3xF), $translationVector$ (3xF)

$N_{\text{markers}}$ = number of rows in $calmarkers$
$N_{\text{frames}}$ = number of frames in $calframereadings$

$centroid_{x_i}$ = mean of $calmarkers$ along rows
Initialize $rotationMatrix\_all = zeros(3, 3, N_{\text{frames}})$
Initialize $translationVector\_all = zeros(3, N_{\text{frames}})$

**for** $f = 1$ to $N_{\text{frames}}$ **do**
    $X_{\text{frame}} = calframereadings(:, :, f)$ ▷ Current frame positions
    $centroid_X$ = mean of $X_{\text{frame}}$ along rows

    $x' = calmarkers - centroid_{x_i}$
    $X' = X_{\text{frame}} - centroid_X$

    $H = (x')^T \cdot X'$ ▷ Covariance matrix
    $[U, S, V] = \text{svd}(H)$
    $R = V \cdot U^T$ ▷ Compute rotation

    **if** $\det(R) < 0$ **then**
        $V(:, 3) = -V(:, 3)$ ▷ Correct reflection
        $R = V \cdot U^T$
    **end if**

    $t = centroid_X^T - R \cdot centroid_{x_i}^T$ ▷ Compute translation

    $rotationMatrix\_all(:, :, f) = R$
    $translationVector\_all(:, f) = t$
**end for**

**return** $rotationMatrix\_all$, $translationVector\_all$

#### 4.3.1 Determining $d_k$

The function **pa3_finding_pointertip** finds the pointer tip position $d_k$ to solve the equation:

$$d_k = F_{B,k}^{-1} F_{A,k} A_t ip \tag{38}$$

This process uses the results of point cloud registration to estimate the rigid body transformations between the tracker frame and the local frames of bodies $A$ and $B$ at sample $k$.

Marker positions and pointer calibration data are first loaded from the provided body and sample files. The algorithm verifies that the number of measured markers matches the expected total for bodies $A$ and $B$, discarding any extraneous data points to ensure consistency.

Then, for each sample $k$, the corresponding subsets of markers belonging to bodies $A$ and $B$ are extracted. Point cloud registration is used to compute the rigid transformations $F_{A_k}$ and $F_{B_k}$. These transformations consist of a rotation matrix and a translation vector, and they optimally align the measured markers with the known body geometries, minimizing the least-squares error. $F_{A_k}$ and $F_{B_k}$ are used to determine $d_k$ and iterating for all samples as shown below:

**Algorithm 3** Pointer Tip Localization in Body $B$ Frame

**Input:**
$bodya\_filepath$ (string path to body A marker and tip data)
$bodyb\_filepath$ (string path to body B marker data)
$sample\_file$ (string path to sampled marker measurements)
**Output:**
$\mathbf{d}_k$ (pointer tip positions in body $B$, $N_{\text{samp}} \times 3$)

Load body A data from $bodya\_filepath$: $N_A, A_{\text{markers}}, \mathbf{p}_{\text{tip}}^A$
Load body B data from $bodyb\_filepath$: $N_B, B_{\text{markers}}$
Load sample marker data from $sample\_file$: $N_s, N_{\text{samp}}, markers$
Verify that the number of measured markers matches expectations
Initialize $\mathbf{d}_k \leftarrow \mathbf{0}_{N_{\text{samp}} \times 3}$

**for** $k = 1$ to $N_{\text{samp}}$ **do**
    Extract markers of body $A$ for frame $k$: $a_{ik} \leftarrow markers(1 : N_A, :, k)$
    Extract markers of body $B$ for frame $k$: $b_{ik} \leftarrow markers(N_A + 1 : N_A + N_B, :, k)$
    Compute rigid transformation $F_{A_k}$ by registering $A_{\text{markers}}$ to $a_{ik}$
    Compute rigid transformation $F_{B_k}$ by registering $B_{\text{markers}}$ to $b_{ik}$
    Transform pointer tip into tracker frame: $\mathbf{p}_{\text{tip}}^{\text{tracker}} \leftarrow F_{A_k}\, \mathbf{p}_{\text{tip}}^A$
    Transform pointer tip into body $B$ frame: $\mathbf{d}_k(k, :) \leftarrow F_{B_k}^{-1}\, \mathbf{p}_{\text{tip}}^{\text{tracker}}$
**end for**

**return** $\mathbf{d}_k$

## 4.4 Calculating Sample Points $s_k$

The function **pa3_body_to_mesh** converts the pointer tip locations $d_k$ from the body B frame to the CT scan mesh frame. $F_{reg}$ represents the corresponding transformation.

For the purposes of this assignment, $F_{reg}$ is an identity transformation.

## 4.5 Finding Closest Points on Mesh

This section describes algorithms for finding the closest points on a mesh from a set of probe points. The goal is to compute the closest point on the mesh surface for each probe tip location by extracting the vertices and triangles from the mesh file. The two methods used are linear search and covariance tree search, which are both called using the function **pa3_find_closest_mesh_points** [7].

### 4.5.1 Finding Closest Point on Triangle

The function **closest_triangle_point** is uses the barycentric coordinates to project the point onto the triangle plane [7]. If the projected point lies outside of the triangle, the closest point on the nearest triangle is selected by using the helper function defined as **closes_point_on_segment**. The following two algorithms demonstrate the connection between these two functions. They are used for both linear and covariance tree searches.

**Algorithm 4** Closest Point on Triangle

**Input:**
$P$ ($N_{\text{points}} \times 3$ array of points)
$A, B, C$ ($1 \times 3$ vertices of the triangle)
**Output:**
closest_points ($N_{\text{points}} \times 3$ closest points on the triangle)

Initialize closest_points $\leftarrow \mathbf{0}_{N_{\text{points}} \times 3}$
**for** $i = 1$ to $N_{\text{points}}$ **do**
    Compute barycentric coordinates $(u, v, w)$ of $P(i, :)$ relative to triangle $(A, B, C)$
    Project point onto triangle plane: $P_{\text{proj}} \leftarrow uA + vB + wC$
    **if** $P_{\text{proj}}$ is inside the triangle **then**
        closest_points$(i, :) \leftarrow P_{\text{proj}}$
    **else**
        Compute closest points on each triangle edge using **closest_point_on_segment**
        Select nearest edge point: closest_points$(i, :) \leftarrow$ nearest edge point
    **end if**
**end for**

**return** closest_points

**Algorithm 5** Closest Point on Segment

**Input:**
$C$ ($N \times 3$ points to project)
$A$ ($N \times 3$ segment start points)
$B$ ($N \times 3$ segment end points)
**Output:**
$C_{\text{closest}}$ ($N \times 3$ closest points on the segment)

Compute vector along segment: $AB \leftarrow B - A$
Compute vector from start to point: $AC \leftarrow C - A$
Compute projection scalar: $\lambda \leftarrow \frac{\text{dot}(AC, AB)}{\text{dot}(AB, AB)}$
Clamp projection to segment: $\lambda_{\text{seg}} \leftarrow \max(0, \min(1, \lambda))$
Compute closest point: $C_{\text{closest}} \leftarrow A + \lambda_{\text{seg}} \cdot AB$

**return** $C_{\text{closest}}$

### 4.5.2 Linear Search

The function **linear_search** is a brute-force method to find the closest point on a mesh for each probe tip. It iterates over all triangles and uses **closest_triangle_point** to determine the nearest point on each triangle, and the closest point out of all the triangles is then selected for each probe tip.

The method iterates through all mesh triangles and evaluates the nearest point on each triangle surface to a given probe tip. For each input point $s_k(i,:)$, the function **closest_triangle_point** computes the orthogonal projection of the point onto the triangle plane, and determines whether the projection lies inside the triangle bounds. If not, the closest edge or vertex is selected. The squared Euclidean distance between the probe tip and the resulting point is then compared to the current minimum, ensuring that the smallest distance and corresponding triangle point are stored. This process is repeated for all triangles and all probe samples.

See Section 3.3 for the mathematical breakdown.

**Algorithm 6** Linear Search for Closest Mesh Points

**Input:**
  $s_k$ ($N_{\text{samp}} \times 3$ probe tip positions)
  vertices ($N_{\text{vertices}} \times 3$ mesh vertices)
  triangles ($N_{\text{triangles}} \times 3$ vertex indices)
**Output:**
  $c_k$ ($N_{\text{samp}} \times 3$ closest points on the mesh)

Initialize $c_k \leftarrow \mathbf{0}_{N_{\text{samp}} \times 3}$
Initialize distance array $dist2 \leftarrow \text{Inf}_{N_{\text{samp}} \times 1}$

**for** $t = 1$ to $N_{\text{triangles}}$ **do**
  $A, B, C \leftarrow$ vertices of triangle $t$
  $closest\_points \leftarrow$ **closest_triangle_point**$(s_k, A, B, C)$
  Compute squared distances: $d^2 \leftarrow \sum(s_k - closest\_points)^2$
  **for** $i = 1$ to $N_{\text{samp}}$ **do**
    **if** $d^2(i) < dist2(i)$ **then** ▷ Check if this triangle gives a closer point than previous
      $c_k(i,:) \leftarrow closest\_points(i,:)$
      $dist2(i) \leftarrow d^2(i)$
    **end if**
  **end for**
**end for**

**return** $c_k$

This method has a computational complexity of $(N_{\text{samp}} \times N_{\text{tri}})$, making it accurate but computationally expensive for large meshes. More efficient spatial search methods described below will be compared to it for validating correctness and computational time.

### 4.5.3 Covariance Tree Search

The function **covariance_tree_search** determines the closest points on a triangulated mesh to a set of probe tip samples [7]. Unlike the brute-force linear search, this method constructs a hierarchical covariance tree using bounding boxes to accelerate nearest-neighbour calculations. Each node of the tree contains a subset of triangle centroids, a bounding box enclosing them, and its left and right children.

The overall algorithm proceeds as follows: given the mesh vertices and triangle indices, the centroids of all triangles are first computed. If no tree exists, the function **construct_covariance_tree_bb** is called recursively to partition these centroids along their dominant covariance directions, until a leaf node containing fewer than a predefined number of points is found. Each node stores its bounding box

parameters (upper and lower bounds in the local covariance frame) for later pruning.

During the search phase, each probe tip position $s_k(i,:)$ is queried using **covariance_tree_query_bb**, which recursively traverses the tree to locate the nearest triangle centroid by discarding nodes whose bounding boxes lie entirely outside the current best distance along any axis. Once the nearest centroid (and its corresponding triangle) is identified, the function **closest_triangle_point** computes the exact point on that triangle surface that minimizes the Euclidean distance to the query point. The set of all such points forms the output matrix $c_k$.

**Algorithm 7** Covariance Tree Search for Closest Mesh Points

**Input:**
  $s_k$ ($N_{\text{samp}} \times 3$ probe tip positions)
  vertices ($N_{\text{vertices}} \times 3$ vertex locations)
  triangles ($N_{\text{triangles}} \times 3$ vertex indices)
  tree (empty or preconstructed)
**Output:**
  $c_k$ ($N_{\text{samp}} \times 3$ closest mesh points)

Compute triangle centroids: $\text{centroid}_t = (A + B + C)/3$
**if** tree is empty **then**
  $tree \leftarrow$ **construct_covariance_tree_bb**(centroids, $vertices, triangles, m_{\text{leaf}}$)
**end if**
**for** $i = 1$ to $N_{\text{samp}}$ **do**
  $query \leftarrow s_k(i,:)$ ▷ Current probe tip position to query the tree for nearest triangle centroid
  $(\_, \_, idx) \leftarrow$ **covariance_tree_query_bb**($tree, query$, centroids, $\infty, -1$)
  **if** $idx < 1$ **then**
    $c_k(i,:) \leftarrow$ [NaN, NaN, NaN] ▷ No triangle found -¿ mesh is empty
  **else**
    $(A, B, C) \leftarrow$ vertices of triangle $idx$
    $c_k(i,:) \leftarrow$ **closest_triangle_point**($query, A, B, C$)
  **end if**
**end for**
**return** $c_k, tree$

**Helper Functions:**

- **construct_covariance_tree_bb(points, vertices, triangles, $m_{\text{leaf}}$)** — recursively partitions centroid data into left and right subtrees based on the principal covariance direction, and computes a bounding box for each node.
- **covariance_tree_query_bb(node, query, allPoints, bestDist$^2$, idx)** — performs a recursive nearest-centroid search using bounding box pruning to minimize distance comparisons.
- **compute_cov_frame(points)** — computes centroid $C$ and rotation frame $R$ from the eigenvectors of the covariance matrix of the points.
- **closest_triangle_point(query, A, B, C)** — computes the closest point on triangle $(A, B, C)$ to the query point using projection and barycentric coordinates.

**Algorithm 8** construct_covariance_tree_bb(points, vertices, triangles, $m_{\text{leaf}}$)

**Input:** points ($N \times 3$ triangle centroids), vertices, triangles, $m_{\text{leaf}}$
**Output:** node (covariance tree structure)

Compute covariance frame: $(R, C) \leftarrow$ **compute_cov_frame**($points$)
Transform points and triangle vertices to local frame
Compute bounding box: $LB \leftarrow \min(V_{\text{local}})$, $UB \leftarrow \max(V_{\text{local}})$

**if** $N \leq m_{\text{leaf}}$ **then**
    $node.haveSubtrees \leftarrow$ false; **return** node ▷ Store points directly, no further splitting
**end if**

Split points along principal axis: $leftMask = pts_{local}(:, 1) < 0$, $rightMask = \neg leftMask$
**if** all points on one side **then**
    $node.haveSubtrees \leftarrow$ false; **return** node
**end if**

Recursively build subtrees:
    $node.left \leftarrow$ **construct_covariance_tree_bb**($points(leftMask, :), vertices, triangles, m_{\text{leaf}}$)
    $node.right \leftarrow$ **construct_covariance_tree_bb**($points(rightMask, :), vertices, triangles, m_{\text{leaf}}$)
**return** node

**Algorithm 9** covariance_tree_query_bb(node, query, allPoints, bestDist$^2$, idx)

**Input:** node, query, allPoints, bestDist$^2$, idx
**Output:** $closestPoint$, $bestDist^2$, $closest\_index$

**if** node is empty **then**
    **return**
**end if**

Transform query to local frame: $v_{\text{local}} = (query - node.C)R$
**for** each axis $j \in \{x, y, z\}$ **do**
    **if** $v_{\text{local}}(j) < node.LB(j) - \sqrt{bestDist^2}$ **or** $v_{\text{local}}(j) > node.UB(j) + \sqrt{bestDist^2}$ **then**
        Prune node and **return**
    **end if**
**end for**

**if** node is leaf **then**
    **for** each point $p$ in node.points **do**
        $d^2 = \|query - p\|^2$
        **if** $d^2 < bestDist^2$ **then**
            $bestDist^2 \leftarrow d^2$, $closestPoint \leftarrow p$
            $closest\_index \leftarrow$ index of $p$ in allPoints
        **end if**
    **end for**
    **return**
**end if**

Compute distances to child nodes along split axis
**if** distance to left child < distance to right child **then**
    Recurse left then right
**else**
    Recurse right then left
**end if**
**return** closest point found

**Algorithm 10** compute_cov_frame(points)

**Input:** points ($N \times 3$ centroid positions)
**Output:** $R$ (rotation matrix), $C$ (centroid)

Compute centroid: $C = \text{mean}(points)$
Center points: $P_c = points - C$
Compute covariance matrix: $A = P_c^T P_c$
Solve eigen-decomposition: $[V, D] = \text{eig}(A)$
Sort eigenvalues in descending order
Form rotation matrix: $R = V(:, \text{sorted indices})$
**return** $R, C$

This approach significantly reduces the computational cost compared to a linear search, especially for large meshes, as distant triangle groups are discarded early using bounding box checks.

Running this method on the mesh given in this assignment generates a covariance tree that is visualized by Figure 2.

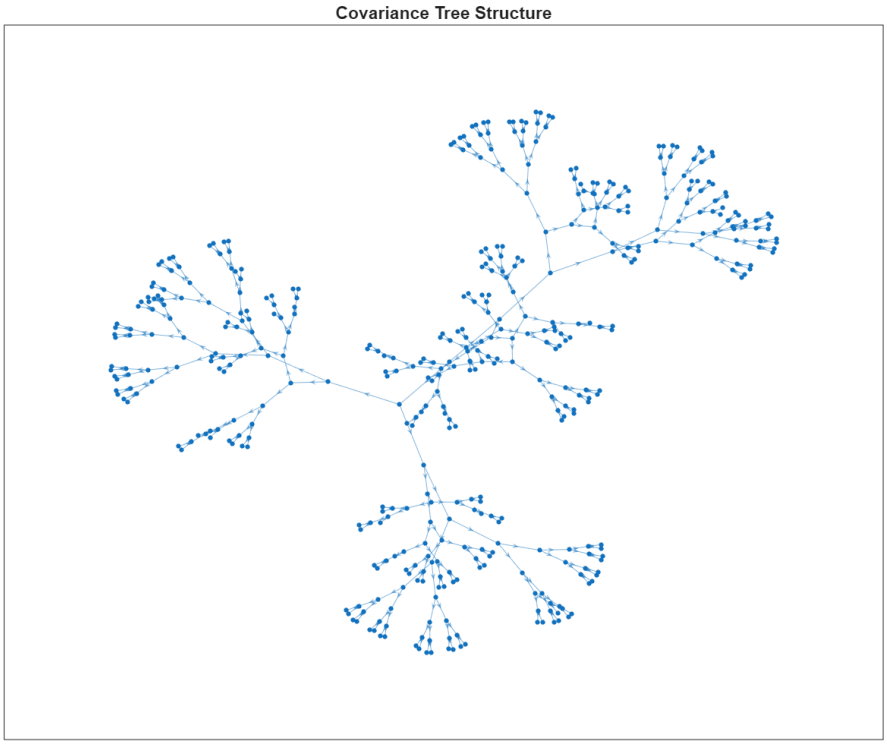


Figure 2: Covariance Tree Structure

#### 4.5.4 Standard ICP

The function **pa4_standard_icp** implements the point-to-point ICP algorithm. For each iteration, the pointer tip samples $d_k$ are transformed using the current registration frame $F_{\text{reg}} = (R_{\text{reg}}, t_{\text{reg}})$ to obtain the transformed points $s_k$. The closest points on the mesh $c_k$ are then determined using either linear or covariance tree search.

The rigid-body transformation that best aligns $s_k$ to $c_k$ is computed using a least-squares point cloud registration, updating $R_{\text{reg}}$ and $t_{\text{reg}}$. The residual distances from point-to-point between $s_k$ and $c_k$ are evaluated to determine the convergence. The iteration continues until the residual falls below a specified tolerance or a maximum number of iterations is reached.

**Algorithm 11** Standard ICP Algorithm [4]

**Input:** $d_k$, initial $R_{\text{reg}}, t_{\text{reg}}$, mesh file, search type
**Output:** Final $R_{\text{reg}}, t_{\text{reg}}$

**for** each iteration until convergence **do**
    Transform samples: $s_k = R_{\text{reg}} d_k + t_{\text{reg}}$
    Find closest points on mesh: $c_k \leftarrow$ **pa4_find_closest_mesh_points**$(s_k)$

    Update registration frame using least-squares point cloud registration

    Compute residuals: $d_i = \|c_k - s_k\|$
    **if** residual $<$ tolerance **then**
        **break**
    **end if**
**end for**
**return** $R_{\text{reg}}, t_{\text{reg}}$

### 4.5.5 Generalized ICP

The function **pa4_generalized_icp** extends standard ICP by incorporating local surface covariances to weight the residuals. At each iteration, the pointer tip samples $d_k$ are transformed to $s_k$, closest mesh points $c_k$ and normals are retrieved, and inliers are selected based on a distance threshold [4].

For each inlier correspondence, the residuals are weighted using the Mahalanobis matrix

$$W_i = (C_B + R_{\text{reg}} C_A R_{\text{reg}}^T)^{-1},$$

where $C_A$ models source uncertainty and $C_B$ encodes local surface covariance. The Jacobians are computed for each correspondence, and the normal equations are solved to obtain the incremental twist $\xi$. This twist is applied via a left-multiplicative SE(3) exponential map to update $R_{\text{reg}}$ and $t_{\text{reg}}$, and the process repeats until $\|\xi\|$ falls below the convergence tolerance.

**Algorithm 12** Generalized ICP Algorithm [4]

**Input:** $d_k$, initial $R_{\text{reg}}, t_{\text{reg}}$, mesh file, search type, parameters
**Output:** Final $R_{\text{reg}}, t_{\text{reg}}$

**for** each iteration until convergence **do**
    Transform samples: $s_k = R_{\text{reg}} d_k + t_{\text{reg}}$
    Find closest points and normals on mesh:
    $(c_k, normals_k) \leftarrow$ **pa4_find_closest_mesh_points**$(s_k)$

    Select inliers based on distance threshold
    Compute residuals $d_i$ and covariance matrices $C_i$ for inliers

    Build normal equations: $H = \sum J_i^T W_i J_i$, $b = \sum J_i^T W_i d_i$
    Solve for twist: $\xi = H \backslash b$

    Update registration frame: $[R_{\text{reg}}, t_{\text{reg}}] \leftarrow$ **se3_exp_update**$(\xi, R_{\text{reg}}, t_{\text{reg}})$
    **if** $\|\xi\| <$ tolerance **then**
        **break**
    **end if**
**end for**
**return** $R_{\text{reg}}, t_{\text{reg}}$

# 5 Overview of Program Structure

The algorithms described in Section 4 are summarized below. A complete overview of the program structure is provided to illustrate the organization of all scripts, helper functions, and unit tests within the project directory. The corresponding folder hierarchy and locations of the `.m` files are shown in Figure 3.

Each function's inputs and outputs have been explicitly defined and commented within the MATLAB source code, and are also summarized in the pseudocode provided in Section 4. This section therefore focuses on the structural relationships between scripts and how the functions, data processing routines, and test classes interact within the overall workflow.

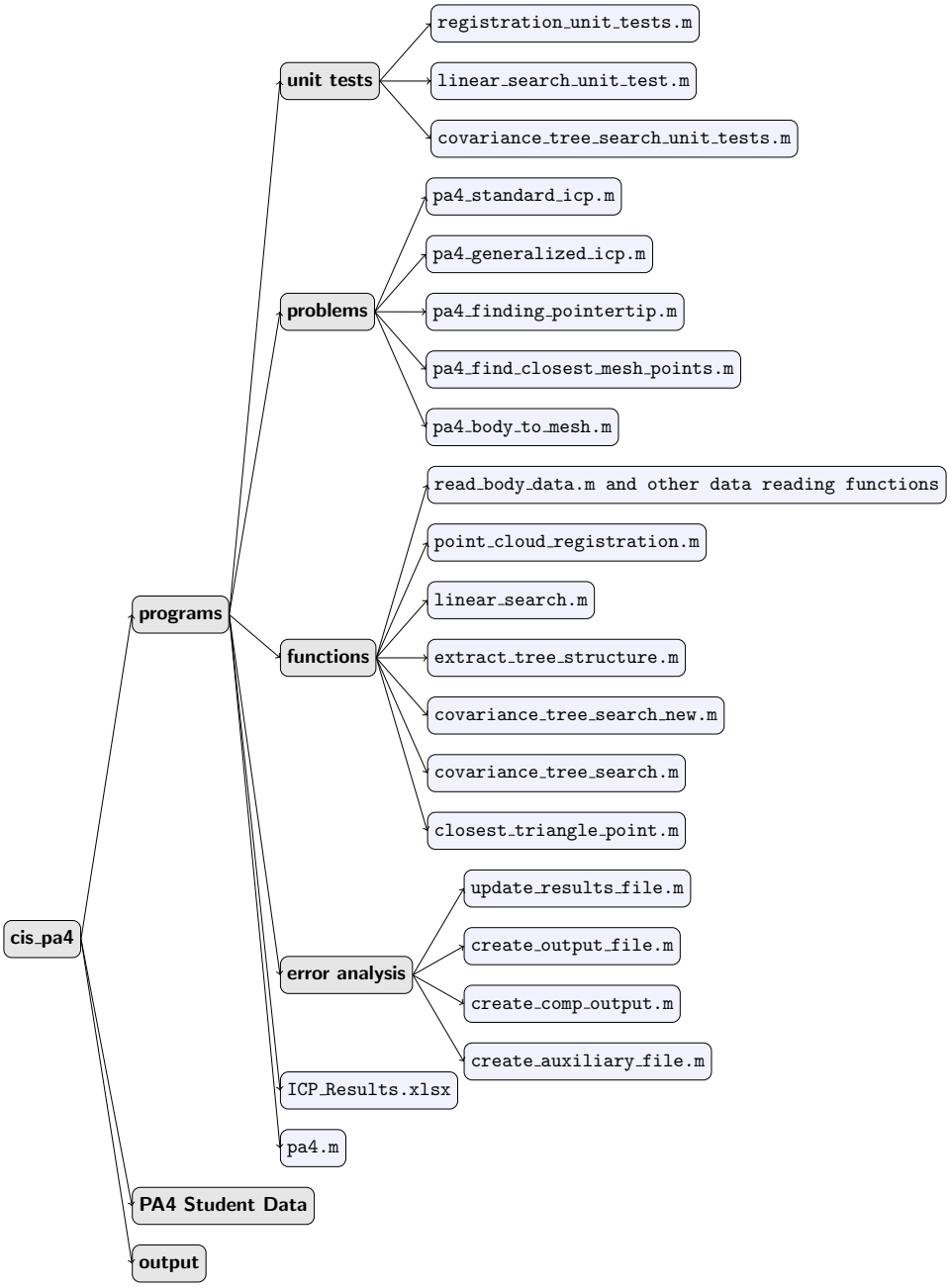


Figure 3: Program hierarchy showing folders, functions, and scripts in `cis_pa4`.

As described in the **README.md** file, navigate to the `programs` folder and run the following code:

```
pa4({search_type},{icp_type});
```

Replace {`search_type`} with either `Linear` or `Covariance`, depending on the desired search method, and {`icp_type`} with either `Standard` or `Generalized`, depending on the desired search method (e.g., `pa4('Linear','Standard');`). The code executes for all provided datasets simultaneously.

Each function's inputs and outputs have been explicitly defined and commented within the MATLAB source code, and are also summarized in the pseudocode provided in Section 4. All outputs are saved to the `output` folder for further review.

Figure 4 illustrates the main program workflow, showing how the primary scripts call supporting functions and how data flows through the PA4 pipeline. This flow chart complements the hierarchy chart in Figure 3 by making explicit the functional relationships between scripts, helper functions, and analysis routines.

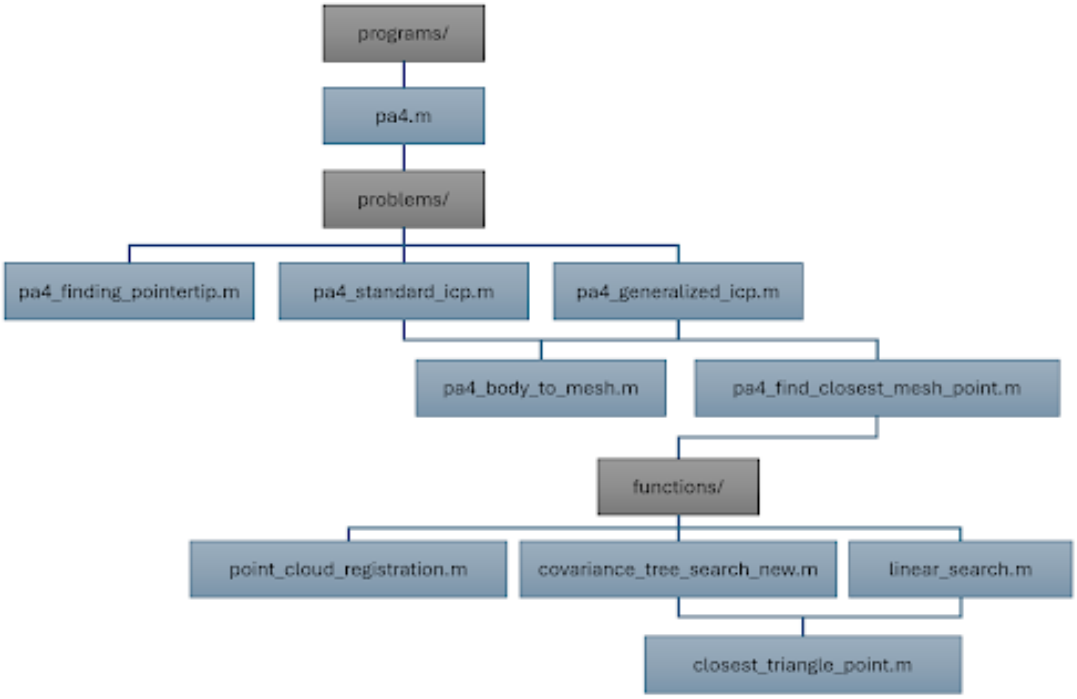


Figure 4: PA4 program flow showing how pa4.m calls the problems folder and functions to show their connections.

# 6 Validation Approach

Three main unit tests were created for testing and validating the point cloud registration, point set registration, and distortion correction algorithms.

## 6.1 Point Cloud Registration Unit Tests

The unit tests used to verify the functionality of the point cloud registration algorithm described in Section 3.1 is called **"registration_unit_tests.m"**, located in the unit tests folder of the program files.

This unit test class evaluates the functionality and accuracy of the point cloud registration function. It does so by defining a ground truth dataset consisting of a set of marker locations in the calibration object frame and applying a random transformation to get the marker locations in the tracker object frame. The unit test generates a synthetic dataset that has multiple frames of markers locations in the tracker object frame, each with their own associated random transformation.

The subclass of "**matlab.unittest.TestCase**" implements the test class to generate a synthetic, noise-free test dataset. A set of transformation matrices, using roll pitch and yaw for rotations and randomized translation, is stored as the known ground truth values. The test class then creates an initial set of 3D marker coordinates ($x_i$) within the calibration frame and applies the known transformation matrix to produce the corresponding transformed coordinates ($X_i$) in the tracker frame.

The same test case data for $x_i$ and $X_i$ is fed into the "**point_cloud_registration**" function for comparison to the known ground truth transformations. The difference of the the estimated rotations and translations from the "**point_cloud_registration**" to the known test case rotations and translations is calculated and the norm is found using the norm function. Note that using the norm function utilizes the Euclidean norm [14]. The unit test will also detect errors in the transformation matrix outside a tolerance of $1 \times 10^{-6}$.

This unit test class tests that the "**point_cloud_registration**" function returns accurate transformations for both a single frame and multiple frames passed into the function. It also tests for the minimum number of markers required to calculate an accurate transformation between the marker and tracker data. Running this test yielded that a minimum of 3 markers are required to calculate a transformation matrix within the defined accuracy bounds.

## 6.2 Linear Search Unit Tests

The robustness and precision of the **linear_search** function were evaluated using a unit test designed to compare its output against a reliable reference implementation, **point2trimesh** from MATLAB [1].

Each trial generated a triangular mesh by consturcting a planar grid over the unit square using the function **meshgrid** [10], ensuring that all vertices were properly referenced within these boundaries. The vertices are then randomly perturbed by a small factor to introduce variability in the mesh geometry. Finally, the triangular faces were obtained via Delaunay triangulation using the function **delaunay** [9].

A set of $N_{\text{samp}} = 10$ random 3D points were generated to serve as the input to the ground truth function, **point2trimesh** [1]. This MATLAB function determines the closest points by finding the shortest line connecting a point and a triangulation by finding the nearest vertex, point on the edges and on the triangle's surface to find the minimum distance between these three values.

Accuracy was assessed by evaluating the Frobenius norm of the difference between these results, $\|\mathbf{c}_{\text{ls}} - \mathbf{c}_{\text{gt}}\|_F$, with a tolerance of $10^{-6}$. This procedure was repeated for **numTrials** $= 25$ iterations, each with independently randomized meshes and query points, to rigorously test the function under a wide range of conditions.

Across all trials, the maximum difference between **linear_search** results and the ground truth remained below the specified tolerance, confirming the reliability and robustness of our function. All 25

trials passed successfully, demonstrating that **linear_search** consistently computes accurate closest points on randomized triangular meshes.

### 6.3 Covariance Tree Unit Tests

The new **covariance_tree_search** function was tested for its robustness and precision using a similar unit test as the **linear_search** function, by using the **point2trimesh** function in MATLAB to create the ground truth data. Results of this unit test are found in the function **covariance_tree_search_unit_tests**.

Similarily, each trial generated a triangular mesh to determine the vertices and triangle parameters, and $s_k$ is randomly generated as well. Accuracy was also assessed by evaluating the Frobenius norm of the difference between the results of the new covariance tree search and the ground truth.

Since the search method for the covariance tree compares the points to the centroid of the triangles rather than the actual points on the mesh, there is an increased chance for the closest point found to not be the actual closest point on the mesh. Hence, the search enclosure method was updated to bounding boxes rather than bounding sphere to generate a more roboust spatial implementation. However, additional edge cases rigorously searching neighboring nodes need to be implemented in the future to address the remaining error issues. For the purpose of the current unit test implementation, the covariance tree search algorithm can find the closest point within 0.25 mm.

## 7 Results

### 7.1 Debug Dataset

The datasets using the debug data are stored in the output folder and named accordingly from A to F for 'pa4-X-Linear-Generalized-Output.txt', 'pa4-X-Linear-Standard-Output.txt', 'pa4-X-Covariance-Generalized-Output.txt' and 'pa4-X-Covariance-Standard-Output.txt'. The following sections analyze those datasets for a better comparison between the standard and generalized ICP methods.

#### 7.1.1 Run Time

All combinations of search type and ICP method were tested to determine the best combination in terms of run time for the complete ICP algorithm. The results are detailed in the table below.

| Dataset | Linear + Standard | Linear + Generalized | Covariance + Standard | Covariance + Generalized |
|---|---|---|---|---|
| A | 27.8459 | 3.1182 | 3.5554 | 0.7770 |
| B | 36.1267 | 4.5730 | 8.7267 | 2.5375 |
| C | 35.4231 | 7.6806 | 9.0438 | 4.0172 |
| D | 35.2758 | 12.6245 | 8.8747 | 5.2792 |
| E | 34.8651 | 14.9382 | 10.0188 | 5.1572 |
| F | 35.5113 | 18.0580 | 9.7333 | 5.1379 |
| G | 35.5571 | 11.1692 | 9.8944 | 5.4343 |
| H | 36.2716 | 17.7839 | 9.3849 | 5.6418 |
| J | 36.5884 | 6.0837 | 9.6877 | 4.1115 |
| K | 58.2642 | 14.0868 | 10.9596 | 4.3021 |
| **Average** | **37.17292** | **11.01161** | **8.98793** | **4.23957** |

Table 1: Execution Times for all Search Type + ICP Type Combinations

As expected, the combination of the covariance tree search and the generalized ICP method yielded the quickest results. Since both methods are able to take into account relevant features of the dataset, they can converge on a result quicker than the other two algorithms. However, this comes at the expense of some accuracy due to implementation structure of the algorithms as detailed in 7.1.2.

### 7.1.2 Error Analysis

Errors in the $d_k$ pointer tip location calculations would be due to registration errors between the optical tracker system frame and the pointer probe frame. Furthermore, additional sources of error between $d_k$ and $c_k$ include registration error between the optical tracker system and the CT image frame. For this assignment, the ICP method updates the transformation $F_{reg}$ to minimize registration errors. However, convergence on local minima away from the absolute minimum solution for $F_{reg}$ result in the errors between the expected values from the given output data and the calculated output data are detailed in the table below.

| **Dataset** | **Linear + Standard** | **Linear + Generalized** | **Covariance + Standard** | **Covariance + Generalized** |
|---|---|---|---|---|
| A | 0.0018 | 0.0018 | 0.0266 | 0.0146 |
| B | 0.0019 | 0.0019 | 0.0257 | 0.0175 |
| C | 0.0020 | 0.0020 | 0.0242 | 0.0143 |
| D | 0.0033 | 0.0033 | 0.0287 | 0.0215 |
| E | 0.0679 | 0.0683 | 0.0845 | 0.0828 |
| F | 0.0666 | 0.0666 | 0.0757 | 0.0755 |
| G | 0.0033 | 0.0034 | 0.0529 | 0.0405 |
| H | 0.0033 | 0.0033 | 0.0225 | 0.0190 |
| J | 0.0667 | 0.0668 | 0.0762 | 0.0771 |
| K | 0.0632 | 0.0631 | 0.0791 | 0.0796 |
| **Average** | **0.02800** | **0.02805** | **0.04961** | **0.04424** |

Table 2: Average of Norm Error for all Search Type + ICP Type Combinations

Both ICP algorithms are able to get results within 0.05mm of the expected output data, with the Standard ICP algorithm performing the same as the Generalized ICP algorithm. The main sources of error for the given implementation are the search types rather than the ICP algorithms themselves.

## 7.2 Unknown Dataset

The output files for unknown data G, H, J, and K are provided in the output files with the naming convention "pa4-X-SearchType-ICPType-Output.txt".

A summary of the run time and error results are included in the tables described in 7.1.1 and 7.1.2. Both ICP methods yield very similar results; however, the Generalized ICP method takes approximately half the amount of run time, making it more optimal than the Standard ICP method. One note on the Generalized ICP method is that the parameters are highly tuned to the given datasets and may run into performance issues when used on other datasets.

# 8 Work Distribution

The distribution of the work was shared between both team members. Shreya and Yvonne worked together on the mathematical approach and understanding the setup. Shreya focused on the mathematical approaches, the functions used to read data files, the search functions, and writing the report. Yvonne focused on the mathematical approaches, the $d_k$ functions, the point set registration algorithm, unit testing, and translating the team's work into the report.

# References


[1] Daniel Frisch. point2trimesh() — Distance Between Point and Triangulated Surface. MATLAB Central File Exchange, 2016. Version 1.0.0.0, updated 25 Sep 2016. Accessed November 13, 2025. URL: https://www.mathworks.com/matlabcentral/fileexchange/52882-point2trimesh-distance-between-point-and-triangulated-surface.

[2] G. Gundersen. A non-technical introduction to svd, 2018. Accessed: 2025-10-03. URL: https://gregorygundersen.com/blog/2018/12/10/svd/.

[3] Scratchapixel. Ray-tracing: Rendering a triangle, 2025. Accessed: 2025-11-08. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/barycentric-coordinates.html.

[4] Aleksandr V. Segal, Dirk Haehnel, and Sebastian Thrun. Generalized-icp, 2009. Accessed November 27, 2025. URL: https://www.robots.ox.ac.uk/~avsegal/resources/papers/Generalized_ICP.pdf.

[5] J. Shi. Arun's method for 3d registration, 2018. Accessed: 2025-09-21. URL: https://jingnanshi.com/blog/arun_method_for_3d_reg.html.

[6] O. Sorkine-Hornung and M. Rabinovich. Least-squares rigid motion using svd. Technical Report, ETH Zurich, 2017. Accessed: 2025-09-21. URL: https://igl.ethz.ch/projects/ARAP/svd_rot.pdf.

[7] R. Taylor. Finding point-pairs, 2025. Accessed: 2025-11-10. URL: https://ciis.lcsr.jhu.edu/lib/exe/fetch.php?media=courses:455-655:lectures:finding_point-pairs.pdf.

[8] R. Taylor. Programming assignment 3 and 4, 2025. Accessed: 2025-11-10. URL: https://ciis.lcsr.jhu.edu/lib/exe/fetch.php?media=courses:455-655:2025:programming_3_and_4_600-445-2025.pdf.

[9] Inc. The MathWorks. `delaunay` — delaunay triangulation. MATLAB documentation, 2025. Accessed November 13, 2025. URL: https://www.mathworks.com/help/matlab/ref/delaunay.html.

[10] Inc. The MathWorks. `meshgrid` — 2-d and 3-d grids. MATLAB documentation, 2025. Accessed November 13, 2025. URL: https://www.mathworks.com/help/matlab/ref/meshgrid.html.

[11] Inc. The MathWorks. svd - singular value decomposition, 2025. Accessed: 2025-10-03. URL: https://www.mathworks.com/help/matlab/ref/double.svd.html.

[12] The MathWorks, Inc. Linear algebra - symbolic math toolbox documentation, 2025. Accessed: 2025-10-03. URL: https://www.mathworks.com/help/symbolic/linear-algebra.html.

[13] The MathWorks, Inc. lsqr - solve system of linear equations: least-squares method, 2025. Accessed: 2025-10-01. URL: https://www.mathworks.com/help/matlab/ref/lsqr.html.

[14] The MathWorks, Inc. norm - vector and matrix norms, 2025. Accessed: 2025-10-06. URL: https://www.mathworks.com/help/matlab/ref/norm.html.