CIS Software Manual

Release v4.0

Joshua Shubert and Nathan Smith

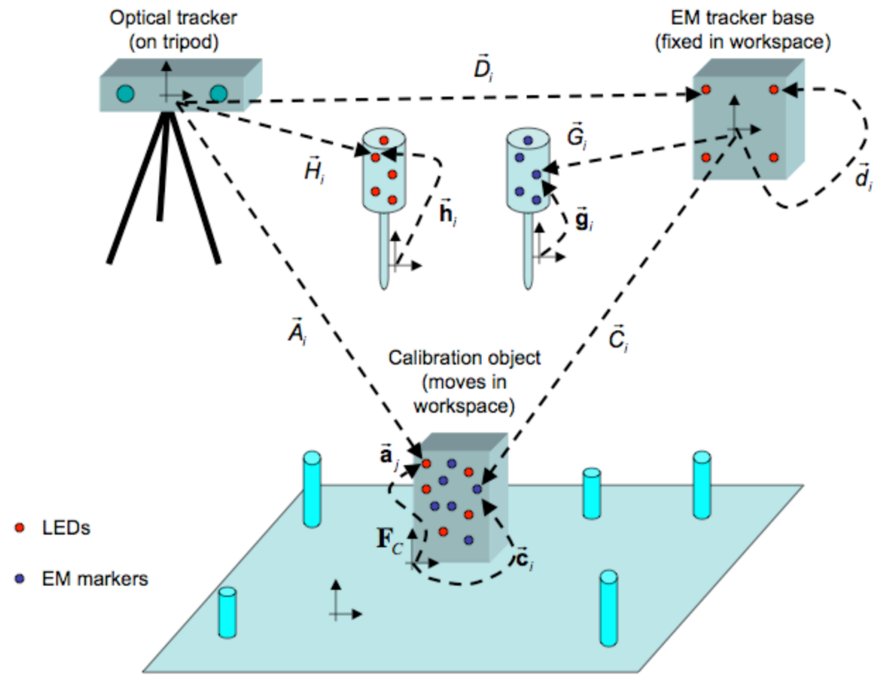
Dec 2, 2016

Contents

1 Introduction

1.1 PA1

The purpose of PA1 was to develop an algorithm for 3D point set to 3D point set registration and pivot calibration. The problem involved a stereo tactic navigation system and an electromagnetic positional tracking device. Tracking markers were placed on objects so the optical tracking device and an electromagnetic tracking device could measure the 3D positions of objects in space relative to measuring base units. These objects were then registered so that they could be related in the same coordinate frames. Pivot calibration posts were placed in the system so pivot calibration could be performed and the 3D position of two different probes could be tracked throughout the system. The diagram below from the assignment document gives a visual description of the system.



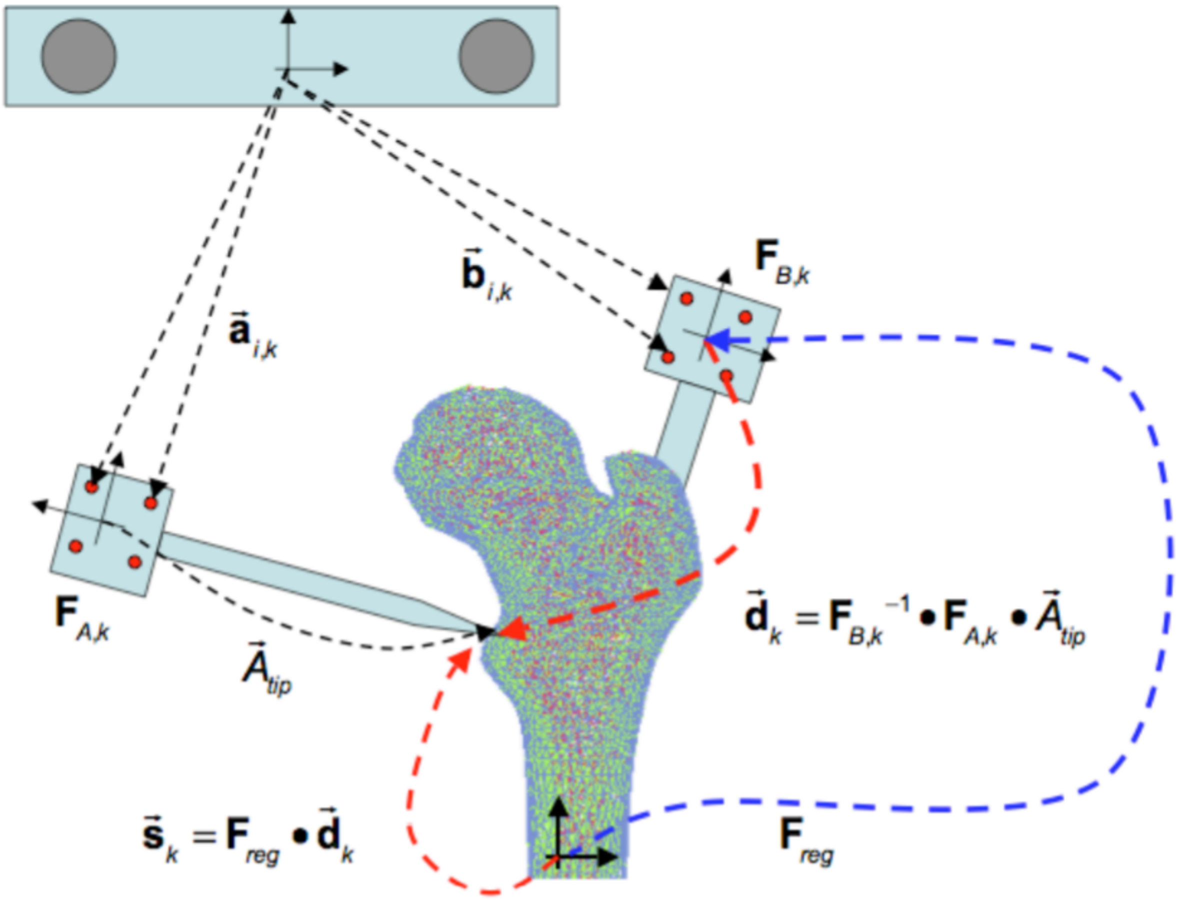
1.2 PA2

PA2 built upon the same setup as PA1 but with two additional steps. The first step was the development of a distortion calibration algorithm that would be used to model and dewarp the EM tracker frame transformations. We then used this dewarped EM tracker space to repeat our pivot calibration calculations. The second step involved computing the CT to EM tracker registration. The locations of several fiducial pins were determined using the EM tracked probe.

Then point cloud to point cloud registration was performed between these fiducials in EM tracker space and in CT space to compute the registration.

1.3 PA3

The purpose of PA3 was to implement the matching step of the Iterative Closest Point algorithm. The first step was to get the coordinates of the tip of tool A into the coordinate system of the embedded tool B. This produces a set of points d on the perimeter of the bone mesh. The next step is the matching step of ICP where these d_i are matched with their corresponding closest point on the mesh. For PA3 a naive brute force search method is used for finding these closest points among the triangles of the mesh.


$\vec{\mathbf{b}}_{i,k}$
$\mathbf{F}_{B,k}$
$\vec{\mathbf{a}}_{i,k}$
$\mathbf{F}_{A,k}$
$\vec{A}_{tip}$
$\vec{\mathbf{d}}_k = \mathbf{F}_{B,k}^{-1} \bullet \mathbf{F}_{A,k} \bullet \vec{A}_{tip}$
$\vec{\mathbf{s}}_k = \mathbf{F}_{reg} \bullet \vec{\mathbf{d}}_k$
$\mathbf{F}_{reg}$

1.4 PA4

The purpose of PA4 was to expand the implemented matching step of the closest point algorithm, and create an Iterative Closest Point algorithm. We first get the coordinates of the tip of tool A into the coordinate system of the embedded tool B, creating a set of points d on the perimeter of the bond mesh. The next step is the matching step of ICP where these d_i are matched with their corresponding closest point on the mesh. Under the naive initial assumption that F_reg is the identity matrix, we use ICP to iteratively generate new estimates of F_reg, iteratively improving our estimate of c = F_reg * d. The previously implemented linear brute force model is very slow however, and thus bounding sphere and covariance tree optimizations were implemented to significantly improve the runtime of the program.

2 Mathematical Approach

2.1 Point Cloud Registration

A number of least squares methods could be used to determine a transformation matrix for a 3D point set registration. We chose a solution from ‘A Method for Registration of 3-D Shapes’, by Besl and McKay, 1992. We seek to solve for R and t, in the equation B = R*A + t, where (R, t) are transforms applied to the dataset A to align it with dataset B, as closely as possible. Finding the optimal rigid transformation matrix can be broken down into the following steps:

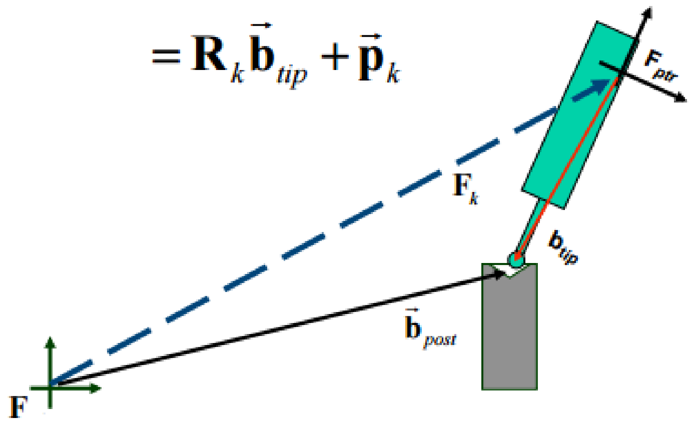
1. Find the centroids of both datasets
2. Bring both matrices to the origin then find the optimal rotation, R
3. Find the translation t

The centroids are simply found by calculating the average (x,y,z) values for each point in datasets A and B. To find the optimal rotation, we make use of singular value decomposition. SVD decomposes the matrix into three other matrices, such that [U,S,V] = SVD(E), and E = USV^t. We first re-center both datasets so that both centroids are at the origin, removing the translation component. We then accumulate matrix H, and solve for [U,S,V] = SVD(H), and find the rotation matrix R = VU^t. The translation is t = -R * centroid(A) + centroid(B).

2.2 Pivot Calibration

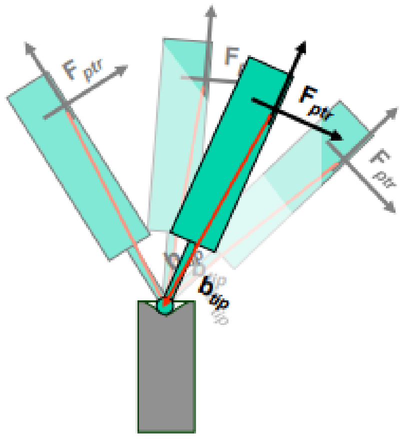
To perform pivot calibration, we need to simultaneously determine the location of the fiducial post b_{post} and the position of the tip of the tracked tool b_{tip}. We use a least squares method to solve for these values simultaneously. This method requires the transformation, F_k between the tracker and the tool for each of the k frames of data, which we determine using our point cloud registration technique between the tracker and the fiducials placed on the tool.

$$\begin{aligned}\vec{\mathbf{b}}_{post} &= \mathbf{F}_k \vec{\mathbf{b}}_{tip} \\ &= \mathbf{R}_k \vec{\mathbf{b}}_{tip} + \vec{\mathbf{p}}_k\end{aligned}$$



To determine the local tool frame, we first compute the midpoint of all the fiducial locations, G_i, in the tracker frame. This gives us a reference, g_0, from which to localize all the other fiducial locations. Once all the g_i are localized we compute the transformation between the gi and the G_i for each frame k of data to get a set of F_k. We then solve the below least squares problem to solve for b_{post} in the tool frame and b_{tip} in the tracker frame.

$$\begin{bmatrix} \vdots & \vdots \\ \mathbf{R}_k & -\mathbf{I} \\ \vdots & \vdots \end{bmatrix} \begin{bmatrix} \vec{\mathbf{b}}_{tip} \\ \vec{\mathbf{b}}_{post} \end{bmatrix} \cong \begin{bmatrix} \vdots \\ -\vec{\mathbf{p}}_k \\ \vdots \end{bmatrix}$$



2.3 Distortion Calibration & Correction

In PA1, we took the naive approach, ignoring possible distortion in our EM tracker frame. In PA2 we address this problem. There are several methods for modelling distortion including Bezier curves, B-Splines, and Bernstein polynomials. We used the Bernstein polynomial approach. nth order Bernstein polynomials are defined as follows:

$$B_{n,\nu}(x) = \binom{n}{\nu} x^{\nu}(1-x)^{n-\nu}, \quad \nu = 0, \ldots, n.$$

Bernstein polynomials have good numerical stability when x is between 0 and 1, which is easy to guarantee by using a “bounding box” to scale the set of points using the maximum and minimum distance values in each direction. Each point x is bounded into a new point u as follows:

$$\vec{\mathbf{u}} = \frac{x - x^{\min}}{x^{\max} - x^{\min}}$$

Before we can correct and dewarp our EM tracker we must first calibrate the Bernstein polynomial coefficient matrix. To do this you need a set of "ground truth" points p_s that are known and a set of distorted points u_s as measured by the EM tracker.
Then we must come up with the following least squares formulation to solve for the c_{ijk} Bernstein polynomial coefficients:

$$\begin{bmatrix} & \vdots & \\ F_{000}(\vec{\mathbf{u}}_s) & \cdots & F_{555}(\vec{\mathbf{u}}_s) \\ & \vdots & \end{bmatrix} \begin{bmatrix} c_{000}^x & c_{000}^y & c_{000}^z \\ \vdots & \vdots & \vdots \\ c_{555}^x & c_{555}^y & c_{555}^z \end{bmatrix} \cong \begin{bmatrix} & \vdots & \\ p_s^x & p_s^y & p_s^z \\ & \vdots & \end{bmatrix}$$

Where each F_{ijk} is defined as follows:

$$F_{ijk}(u_x, u_y, u_z) = B_{5,i}(u_x) B_{5,j}(u_y) B_{5,k}(u_z)$$

Solving the previous least-squares problems yields coefficient matrix **C** that can be used to correct for distortion. Given a new set of distorted points q_s we first scale them to a new set of points u_s using the same "bounding box" as earlier. Then for each u_s we compute its corresponding corrected point p by computing the following:

$$\vec{\mathbf{p}} = \sum_{i=0}^{5} \sum_{j=0}^{5} \sum_{k=0}^{5} \vec{\mathbf{c}}_{i,j,k} B_{5,i}(u_x) B_{5,j}(u_y) B_{5,k}(u_z)$$

2.4 CT Fiducial Registration

After calibrating the Bernstein polynomial coefficient matrix, we corrected our set of tracker measurements and repeated our pivot calibration following the method described in 2.3. For calculating the location of the other k fiducial pins we had a single frame of data, G_k per pin. We computed the frame transformation F_k between the original pivot calibration data and the new G_k data and then applied this transformation, F_k,to the original pivot location to find the location g_k of the k^{th} fiducial pin. At this point we have the coordinates of the k fiducial pins in the EM tracker frame, g_k, and also the coordinates of the fiducial pins in CT space, b_k. From this we just compute the point cloud to point cloud registration to compute the transformation between CT space and the EM tracker frame, F_{reg}. This F_{reg} is used to compute the location of the EM tracked in CT space for any arbitrary position.

2.5 ICP Registration

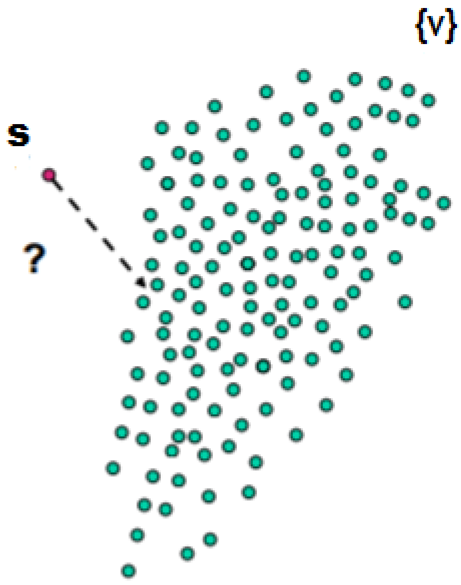
Iterative Closest Point registration is used to match preoperative data (perhaps CT data) to a set of data points collected with a tracked tool during the actual operation. To perform ICP you first need a mesh of points describing the 3D structure you would like to register to the patient, a set of points d_k gathered around the real world structure that our 3D mesh corresponds to, and an initial guess F_{reg} for the registration transformation between the two. Given the initial guess Freg and d_k, transform d_k using F_{reg} and perform ICP matching (described below) to obtain the point c_k on the mesh that is closest to d_k. After performing this for each point d_k, perform point cloud-to-point cloud registration to obtain ΔF_{reg} . Then update F_{reg} such that $F_{reg\text{-}new} = \Delta F_{reg} F_{reg}$ and repeat until a termination condition is reached such as maximum iterations or convergence between the preoperative and intraoperative data set.

2.6 ICP Matching

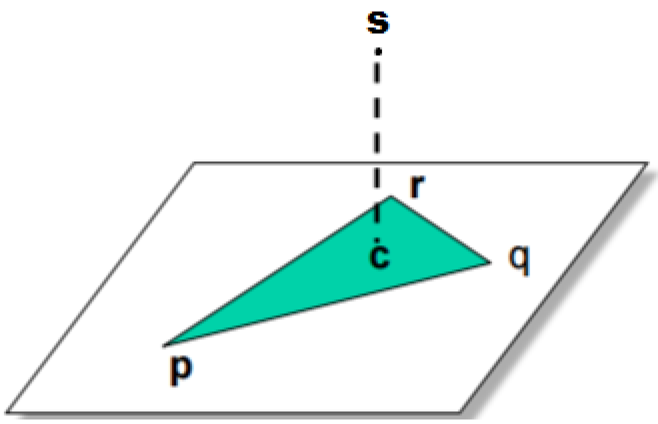
ICP matching is the part of the ICP algorithm where we search for the closest point on a mesh M to a given point s. There are many ways to accomplish this search including brute force linear search, bounding boxes and spheres, covariance trees, octrees and kD trees. The methods we implemented are described below.

2.6.1 Brute Force Linear Search

Brute force linear search is a straightforward method to solve the problem of finding the best match between a point s and a mesh of vertices or triangles.



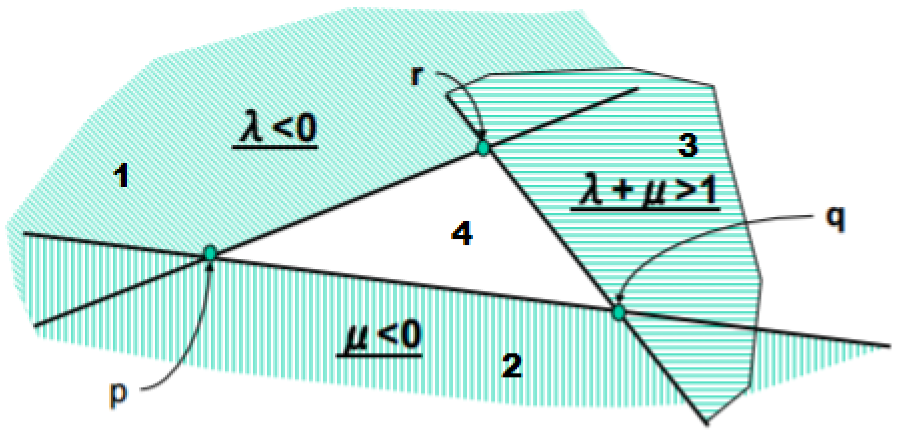
The process works as follows. Start by iterating through the mesh, triangle by triangle. The order or relative 3D position of the triangles is not considered. Once we have a triangle, we need to find the closest point c on the triangle compared to the given point s.



In order to find this point c, we need to project the vector s - p onto the triangle. However the projection equation changes depending on whether we over/inside the triangle or if we are outside the triangle. To figure out what projection equation to use we need to solve following linear combination equation:

$$s - p = \lambda(q - p) + \mu(r - p)$$

Once we solve this (likely with a least squares method due to the underdetermined nature of the system) we will have two parameters, λ and μ. Based on these two parameters and the following diagram and the included constraints on λ and μ, we select the appropriate projection equation from the table.



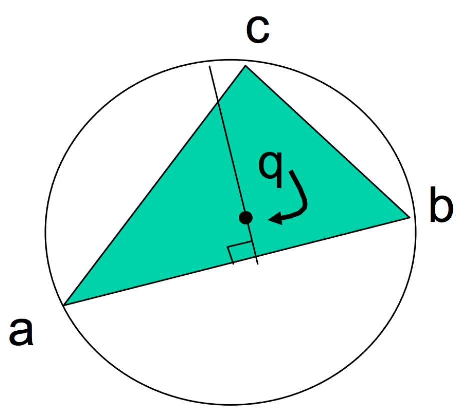
Region	Condition	Projection Equation
1	$\lambda < 0$	$c^* = r + \lambda^*(p - r)$
2	$\mu < 0$	$c^* = p + \lambda^*(q - p)$
3	$\lambda + \mu > 1$	$c^* = q + \lambda^*(r - q)$
4	$0 < \lambda + \mu \leq 1$	$c = p + \lambda(q - p) + \mu(r - p)$

We use the projection equation to determine a value for c_i, the point that is closest to s on the i^{th} triangle. Once we have repeated this process for all triangles, we still need to accomplish our original goal of finding the closest point to s on the whole mesh. We do this by computing the distance between s and c_i for all i and then determining which c_i produced the shortest s - c_i distance. This ci will be the closest point on the entire mesh to s. We then return this value to algorithm and repeat for the next s value.

2.6.2 Bounding Sphere Search

The use of bounding spheres during ICP provides greater time optimization than the above method of linear search. As stated above, we are presented the problem of given point a, we wish to find a corresponding point b on the surface. We start by iterating through the mesh, triangle by triangle. The order or relative 3D position of the triangles is not considered. Once we have a triangle, we need to find the closest point c on the triangle compared to the given point s. For our bounding sphere optimization, the basic approach is to divide the space into regions. Suppose we have a point bk* that is a possible match for point ak. The distance Δ*=|| bk* - ak|| acts as an upper bound on the distance of the closest point to the surface. Given a region R containing many possible points bj, if we can compute a lower bound ΔL on the distance from a to any point in R, then we need only consider points inside R if ΔL < Δ*.

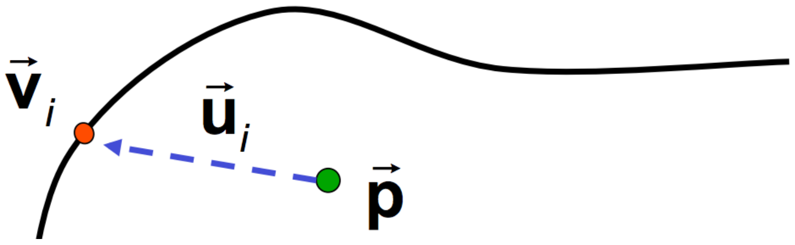
To improve our brute force linear approach, we first implement a bounding sphere approach, where we construct bounding spheres around each triangle and use these to reduce the number of careful checks required.



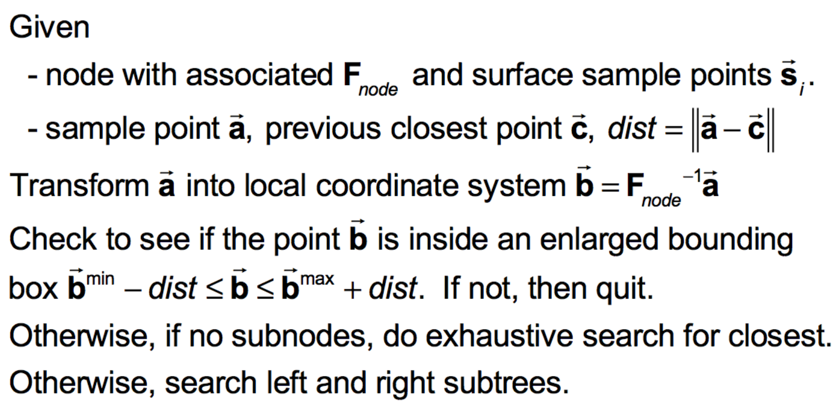
Suppose we have the above bounding sphere, and are trying to find the closest triangle (a, b, c) to p. If qk is the center of a sphere of radius ρk enclosing (ak , bk , ck), then we only need to check carefully if ||p − qk|| − ρk < ||p − rj||. Assuming edge (a,b) is the longest, we can approximate q = (a+b)/2. If inequality does not hold, we may solve for the system to get q. The radius may then be calculated as ρ = ||q - a||.

Finally, we wish to find the closest point on the triangle [p,q,r] to the point a. We can do so by solving the system a-p = λ(q-p) + μ(r-p) via least squares for λ and μ. We then compute the closest point c = p + λ(q-p) + μ(r-p).

2.6.3 Covariance Tree Optimization



To further optimize the runtime of our OCP algorithm, we may implement a covariance tree. Here, we define a local node coordinate system F = [R,p] and sort the surface points according to the sign of the x component of b = inv(R) * u. We then compute the bounding box of b_{min} and b_{max}, and assign these points to the left and right subtree nodes. We then form the outer product matrix A, and compute the eigenvalues and eigenvectors of A. We then determine a rotation R such that R_x is the eigenvector corresponding to the largest eigenvalue. To then search this covariance tree, we use the following method as outlined in class:



Given
- node with associated $\mathbf{F}_{node}$ and surface sample points $\vec{\mathbf{s}}_i$.
- sample point $\vec{\mathbf{a}}$, previous closest point $\vec{\mathbf{c}}$, $dist = \left\|\vec{\mathbf{a}} - \vec{\mathbf{c}}\right\|$

Transform $\vec{\mathbf{a}}$ into local coordinate system $\vec{\mathbf{b}} = \mathbf{F}_{node}^{-1}\vec{\mathbf{a}}$

Check to see if the point $\vec{\mathbf{b}}$ is inside an enlarged bounding box $\vec{\mathbf{b}}^{\min} - dist \le \vec{\mathbf{b}} \le \vec{\mathbf{b}}^{\max} + dist$. If not, then quit.

Otherwise, if no subnodes, do exhaustive search for closest.

Otherwise, search left and right subtrees.

3 Algorithmic Approach

3.1 Parsing

Our program was developed in Python. The Numpy library was used as a Cartesian math package for 3D points, rotations, and frame transformations. The linear algebra package numpy.linalg was also used for matrix operations including determinants and inversions. The first step was to develop a parser that could interpret and load the provided data. The parser for each file observes the first line of the file to determine the number of elements and frames for the dataset. It then iterated through the dataset line by line, accumulating arrays where each element represents a 3-dimensional point. These arrays are stored as numpy arrays, allowing for numpy functions and transformations to be applied.

3.2 Point Cloud Transformation

We next developed the algorithm for point cloud to point cloud registration, and determining the rotation and translation components of the transformation between them. As covered above, the function uses singular value decomposition to separate the R and t elements of the transformation. We first load in both point clouds, represented as numpy arrays. The centroids of each array are determined by finding the mean of the [x,y,z] values of each element in the arrays. These mean values are then subtracted from every point in each array to localize the arrays and determine the rotation component. Numpy's linalg.svd function is used to decompose H into U, S, V, and R is then calculated as the multiplication of V and U. The translation t is then calculated as -R * centroid(A) + centroid(B), using the numpy.dot function to calculate matrix dot products. These R and t values are then rounded and returned by the function.

3.3 Pivot Calibration

Both the EM pivot calibration and the Optical pivot calibration are solved using the same function calibrate_pivot(). For the optical pivot calibration there is a wrapper around calibrate_pivot() called opt2em_calibrate_pivot() that translates the optical fiducial locations from the optical tracker frame to the EM tracker frame before performing the pivot calibration. In essence, calibrate_pivot() performs the same tasks as mentioned above for pivot calibration. A set of points describing the locations of the fiducial trackers on the tool in the tracker frame are localized by first computing the midpoint of the set of midpoints using compute_midpoint(), and then creating a new set of points where each point is the original tracker frame point with the location of the midpoint subtracted from it using localize(). Then point cloud to point cloud registration is performed between these two sets of points to get the transformation from tracker to tool for this frame of data.
Once the frame transformation has been calculated for each frame of data, the frames are formatted into a matrix according to the following graphic by the pivot_calibration() function:

$$\begin{bmatrix} \vdots & \vdots \\ \mathbf{R}_k & -\mathbf{I} \\ \vdots & \vdots \end{bmatrix} \begin{bmatrix} \vec{\mathbf{b}}_{tip} \\ \vec{\mathbf{b}}_{post} \end{bmatrix} \cong \begin{bmatrix} \vdots \\ -\vec{\mathbf{p}}_k \\ \vdots \end{bmatrix}$$

Once the R and p matrices are assembled, we call Numpy's lstsq() function to solve the least squares problem. It returns a vector b that contains the b_{tip} and b_{post} solutions.

3.4 Distortion Calibration & Correction

Distortion calibration is done using the calibrate_distortion() function. It takes in a set of ground truth points p, a set of distorted points q, and the order of the Bernstein polynomial to use which has a default value of 5. First q is bounded between 0 and 1 with a call to ScaleToBox() which takes as an argument q, as well as q_{min} and q_{max} and returns a bounded set u. Then we iterate through each point u_s in u and compute the Fijk as described in 2.3. For each point u_s we compute an array of n^3 F_{ijk}, where n is the order of the Bernstein polynomial. Each array of F_{ijk} is concatenated vertically to create a s x n3 matrix called **F**. Then we solve the least squares problem **FC** = p for **C** which is our Bernstein polynomial coefficient matrix.

Now that we have **C** we can correct the distortion in other sets of data using correct_distortion() which takes as arguments a set of distorted points q, the previously calculated matrix C, and the q_{min} and q_{max} bounds that we originally calculated **C** with. First we create a bounded set u from q just like we did for the calibration. Then we iterate through each point u_s in u calculating the corrected point p according this equation as mentioned in 2.3:

$$\vec{\mathbf{p}} = \sum_{i=0}^{5} \sum_{j=0}^{5} \sum_{k=0}^{5} \vec{\mathbf{c}}_{i,j,k} B_{5,i}(u_x) B_{5,j}(u_y) B_{5,k}(u_z)$$

This set of points p will be the set of distortion-corrected points.

3.5 CT Fiducial Registration

The first step in CT registration is to repeat the pivot calibration using our distortion corrected data, which we do by calling em_pivot_calibration(). Then we iterate through the k data frames to determine the locations g_k of the fiducial pins in EM tracker space. For each data frame Gk we compute the frame transformation F_k between G_k and the set of data G used for the pivot calibration using get_transform(G, G_k). Then we apply this transform F_k to the location of the fiducial pin used for pivot calibration to get the location of the new fiducial pin in EM tracker space. This is done using transform_3D().

Then we simply call get_transform() to compute F_{reg} between the set of fiducials in EM tracker space g_k and the set of fiducials in CT space b_k. This transformation F_{reg} is then used to compute the CT coordinates of the EM tracked tool tip for a set of orientations. This transformation calculation is handled by the function get_tip_coordinates() which first transforms the new orientation of EM tracked tool into the frame used for pivot calibration, and then transforms that into CT coordinates.

3.6 ICP Registration

The complete ICP registration is implemented through the icp_match_iterator() function. This function can run three different methods for the matching part of the registration. These methods include brute force linear search, bounding sphere, and covariance tree. These are specified by user input in the form of strings e.g. “linear” “sphere” or “tree” at run time. This function iterates either 40 times or the number of times it takes to get the registration error below a threshold. In each iteration first the set of d points is transformed with the current estimate of the registration transformation to a set s_i. This set is then matched to points on the mesh using the user-specified search method. Then a point cloud to point cloud registration is performed between s_i and the matches found on the mesh to calculate an frame transformation adjustment. This frame transformation adjustment is applied to the old estimate of the registration transformation to determine the new registration transformation. If the error produced with this new registration transformation falls below a set threshold, the function returns and error results are reported. Otherwise the process will iterate again.

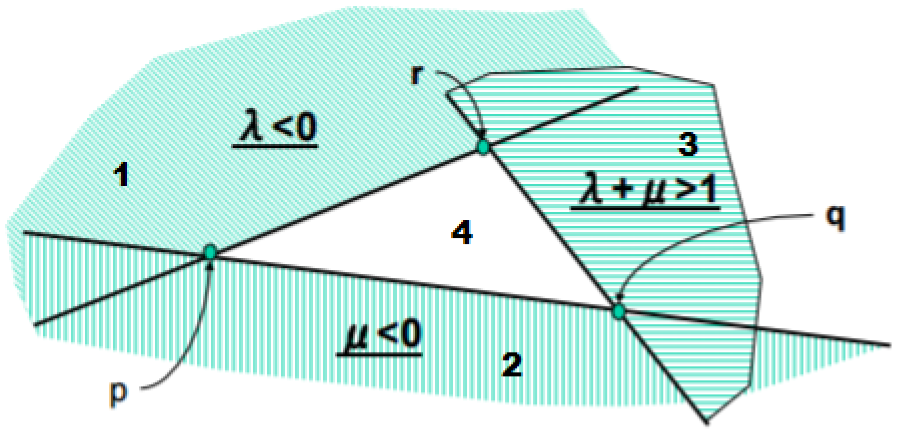
3.7 ICP Matching

ICP matching is entirely handled by the icp_match() function. In PA3 it only implements brute force linear search, but in PA4 it will be expanded to utilize other search methods. In PA4, the icp_match_iterator() method is called before icp_match(), and first loads in the list of triangles.

icp_match() takes in four arguments: a list of triangle coordinates, a list of triangle indices to construct the triangles, a set of points d_k corresponding to intraoperatively marked locations that we would like to register our mesh to, and a string that specifies what match search method to use. There are three options for the search method: “linear” “sphere” and “tree” which correspond to the 3 methods described below.

3.7.1 Brute Force Linear Search

If icp_match() is called with the “brute” method we will begin iterating through the d_k points around our mesh M. For each d_k we will transform it with the current best guess for F_{reg} to produce a value c_0. Once we have performed the registration transformation, we need to find what point is closest to c_0 on the mesh. This is accomplished by calling the function closest_point_linear(). closest_point_linear() starts by iterating through the triangles comprising the mesh one by one. For each triangle in the mesh we check to see where the current c_0 (or s as it is referred to in the function) lies with respect to the triangles edges. There are four cases to consider. The first three cases are when the point s is outside the 3 different sides of the triangle and the fourth case is when the point lies within the bounds of the triangle (or in 3D space the point s lies within the triangular prism defined by the triangle). λ and μ are found by solving s_i - p = λ(q - p) + μ(r - p) using a least squares method. Also note that if s in regions 1, 2, or 3 then λ becomes λ* = Max(0, Min(λ,1)) e.g. λ is bounded between 0 and 1.



Region	Condition	Projection Equation
1	**λ < 0**	**c* = r + λ*(p - r)**
2	**μ < 0**	**c* = p + λ*(q - p)**
3	**λ + μ > 1**	**c* = q + λ*(r - q)**
4	**0 < λ + μ ≤ 1**	**c = p + λ(q - p) + μ(r - p)**

Once we have determined what region s is in, we use projection equations to determine the point c on the triangle that is closest to s and then we append this c to a list of c values, one for each triangle. After we have found the closest point c to s for all triangles in the mesh, we iterate through the c values and calculate the distance from each c_i to s. We then determine which c_i produced the shortest s to ci distance. This c_i is the true closest point to s on the mesh, and thus we return it.

3.7.2 Bounding Sphere Search

For PA4, the first steps of ICP have been expanded to iteratively improve F_reg to optimize the final closest points c with respect to d. If the driver is called with the “sphere” method, we will use the bounding sphere implementation to improve runtimes for ICP iterations. For each d_k we will transform it with the current best guess for F_{reg} to produce a value c_0. Once we have performed the registration transformation, we need to find what point is closest to c_0 on the mesh. To accomplish this, we call the function closest_point_sphere(). As outlined in the mathematical section above, this loads in the three coordinates of the triangle and runs the closest_point_triangle() method to determine the minimum distance point on the triangle from s. This follows the same method as linear search, however we now do not need to linearly check every possible triangle. Each triangle of the triangle class has a respective bounding sphere, and if qk is the center of the sphere of radius ρk enclosing (ak , bk , ck), then we only need to check carefully if $||p − qk|| − ρk < ||p − rj||$. Thus, we consider a small fraction of the triangles previously checked in the linear search method for determining the closest point.

3.7.3 Covariance Tree Optimization

Given a list of triangles, icp_match_iterator() can then build the covariance tree. To do this, the cov_tree_node class loads its triangles, initializes its bounds to none, and then determines its local coordinate system frame. When finding the covariance frame, the outer product matrix A formed, and the eigenvalues and eigenvectors of A are determined. Given the maximum eigenvector, we then determine the covariance frame rotation matrix R such that R_x is the eigenvector corresponding to the largest eigenvalue.

Then, it builds its subtrees by sorting the triangles according to the sign of the x component of b = inv(R) * u, as detailed in the mathematical approach section. We used np.signbit, which returns true if any of the signs of the x component of b are negative. If the split point is determined to be valid, we then create subtrees[0] and subtrees[1], two new instances of a covariance tree node where the triangles of each tree are split at the determined splitpoint. Then when searching the covariance tree, we again follow the method outlined in class. Given our node with its covariance frame and a sample point s, we transform it into the local covariance coordinate system via a frame transformation, and check to see if b is within the covariance node bounds. If now, we quit, while if there are subtrees we then recursively call the find closest point algorithm on the respective subtrees. If there are no subnodes and we are within the bounds, we then perform an exhaustive search for the closest point.

4 Structure of Program

4.1 PA1/2

The functions making up the assignment 1+2 implementation are as follows, with the newly implemented functions in **bold**:

p1_driver	The main function for program 1. Loads in each of the datasets, performs pivot calibration and point cloud transformations. Creates and populates output files for evaluation.
p2_driver	The main function for program assignment 2. Loads each of the datasets, performs pivot calibration and point cloud transformations, as well as computing and applying distortion correction. Creates output files in the format of assignments 1 and 2.
read_cal_readings	Reads the cal_readings file into a numpy array dataset.
read_cal_body	Reads the cal_body file into a numpy array dataset.
read_em_pivot	Reads the em_pivot file in a numpy array dataset.
read_opt_pivot	Reads the opt_pivot file into a numpy array dataset.
get_transform	Takes in two matrices A and B. Determines the rotation matrix R and translation component t such that B = R*A + t, and returns R and t.
calibrate_pivot	Takes in the em_pivot dataset G, the set of points where the tool is in contact with a fiducial pin. Returns p_tip, the vector from the tool frame to the end of the tool, and p_post, the vector from the EM base frame to the top of the fiducial post.
compute_midpoint	Called by calibrate_pivot, returning the mean points [x, y, z] of the input array.
localize	Called by calibrate_pivot, returning a numpy array with the mean points [x,y,z] subtracted from each point element.
opt2em_calibrate_pivot	Calculates the transform between the EM and Optical tracker for each frame of data.
p1_tests	Runs unit tests on the dataset loading, transformation, and pivot calibration functions to ensure they're working properly outside of the final output.
p2_tests	Runs unit tests on the new dataset loading, transformation and

	pivot calibration, as well as the distortion correction functions.
calibrate_distortion	Creates a coefficient matrix for nth-order Bernstein polynomial distortion correction. Requires a ground truth data set and a corresponding distorted data set.
correct_distortion	Given a previously calculated Bernstein polynomial coefficient matrix, corrects for the error in a set p of EM tracked points.
get_tip_coordinates	Computes the location of the tool tip in CT coordinates after both pivot calibration and registration has been performed.

4.2 These functions have been organized into the following files:

p1_driver.py	The driver for PA1, producing output files.
p1_tests.py	Contains the unit tests for all core functions of PA1.
read_datasets.py	Contains all dataset parsing functions.
transformation.py	Contains all point cloud transformation functions.
calibrate_pivot.py	Contains all pivot calibration functions.
p2_driver.py	The main driver for PA2, producing output files.
p2_tests.py	Contains the unit tests for all core functions of PA2.
correct_distortion.py	Contains all functions for distortion calibration and correction

Assignment 2 Run Instructions

Additionally, data and output folders contain all input datasets, and all output produced for each of these datasets, respectively. To run the program on all datasets at once, navigate to the source folder, and enter:

```
python p2_driver.py
```

Then, refer to the output folder to find all generated output files. Each output-1 file contains the em and opt pivot calibration position estimates as the second and third lines. The remaining lines cover the expected values C for each point and frame in the respective dataset. Additionally, each output-difference file lists the difference between the expected and true C values for each of the datasets.

The assignment 2 driver also produces output-2 files for each of the datasets. These display the positions of the probe tip in CT coordinates, corresponding to the frames of data in the em-nav files for each dataset. These have been compared against the provided output-2 files for evaluating the program.

Extensive comments have been provided in each of the source files to outline the structure and flow of the program. The driver iterates through each of the 10 provided datasets in assignment 2, each time setting the names of each of the dataset components and loading them into the system as a numpy array. The EM and Optical pivot calibration functions are then called, with their values stored to be printed in the output file. The C values are then iterated through frame by frame, with the expected Cs calculate through get_transform and appended to the final expected C list. Finally, output files are generated and saved in the output folder.

Then, moving on to the requirements of assignment 2, the distortion correction function is built using the calibrate_distortion as described in the analytical breakdown above. We then use this corrected distortion to repeat pivot calibration for the EM probe, by running correct_distortion followed by calibrate_pivot. Next, we compute the registration frame Freg by determining the transform between the CT image and the EM tracker. Finally, we compute the EM tracked tool's coordinates in the CT image by iterating through the frames of EM_Nav. The new output-2 files contain the calculated positions of the probe in CT coordinates, and are filled following the completion of the driver algorithm for each dataset.

To run the program's unit tests, navigate to the source folder and enter:

```
python p2_tests.py
```

All unit tests have been found to pass properly, as outlined in the next section.

4.3 PA 3

The functions making up the assignment 3 implementation are as follows:

p3_driver	The program driver, taking the sample readings letter and data type as input and writing to the output file.
read_rigid_body	Reads the A and B rigid body data files into an array dataset.
read_sample_readings	Reads the sample readings data file into array datasets for A and B frames.
read_mesh	Reads the mesh data file into an array dataset of triangles.
icp_get_d	Calculates the set of points d around the mesh to be used in ICP matching.
icp_match	Finds the matches between a set of points 'a' and a 3D structure defined by 'M'.
findClosestPoint_brute	Searches through the triangles composing a 3D mesh to find the closest position on the 3D mesh to a given point s.
transform3D	Performs a simple Ra + p transform.
get_transform	Determines the rotation and translation components of the transformation between the two provided point cloud frames a and b.
get_inverse	Returns the inverse of a frame transformation.
get_distance	Returns the euclidian distance between two 3D points.
test_find_b_tip	Tests the find_b_tip method with randomly generated data.
get_rotation	Creates a 3D rotation matrix for testing the find b_pointer method.
test_find_closest_point	With a single triangle, tests the find_closest_point method on a variety of input points.
test_icp_match_linear	Tests linear ICP using a given mesh and specified points to search.
test_transform	Tests that the get_transform function correctly determines the rotation and translation components of the registration.

4.3.1 The files making up the assignment 3 submission are as follows:

p3_driver.py	The program's main driver, which generates the output file for a given sample readings input.
ICPread.py	Contains all files for reading input data and generating arrays.
ICPmatch.py	Contains all files for find point cloud transformations and determining ICP matches.
p3_tests.py	Contains test files for calculating d, finding the closest point in a triangle, and performing the full linear ICP matching, as well as previous tests for point cloud transformations.

Assignment 3 Run Instructions

To run the assignment 3 program, navigate to p3/programs/source/p3_driver.py. Enter the following to generate output data for the A-Debug sampleReadingsTest set:

```
python p3_driver.py A-Debug
```

Similarly, provide the letter and type for each SampleReadingsTest dataset you would like to generate an output file for. Each output file exactly follows the format specified in the assignment handout. The leftmost column lists the x,y,z coordinates of d, with the next column listing the coordinates of c, and the final column listing the magnitude of difference between the two sets of points.

Extensive comments have been provided in each of the source files to outline the structure and flow of the program. After specifying an input dataset, the driver program sets the file paths for each of the required input files. The methods outlined in ICPread.py are called to load each of these data files into the necessary arrays. Next, icp_get_d is called to determine the position of the pointer tip with respect to the rigid body B. This is done by first calling get_transform to determine the rotation and translation transforms between the sample readings and rigid body point clouds for each frame, and then applying transform3D to get the final coordinates d.

With d determined, icp_match is called, which finds the closest matches between the provided points and the given mesh. This is done using the method outlined in section, where a brute force linear approach iterates through each of the mesh's triangles, determining the minimum distance between the given d point and the triangle. The points c on the surface mesh that are closest to the to the points s = F_reg * d are selected and returned in the output file. The output file lists the calculated d and c coordinates, as well as the difference between the two, with identical format to the provided answer files.

To run the program’s unit tests, navigate to the source folder and enter:

python p3_test.py

Unit tests were written for each of the program’s core components and pass successfully, as detailed in the next section.

4.4 PA 4

The functions for our PA4 implementation are as follows:

p4_driver	The program driver, taking the sample readings letter and data type as input and writing to the output file.
read_rigid_body	Reads the A and B rigid body data files into an array dataset.
read_sample_readings	Reads the sample readings data file into array datasets for A and B frames.
read_mesh	Reads the mesh data file into an array dataset of triangles.
icp_get_d	Calculates the set of points d around the mesh to be used in ICP matching.
icp_match_iterator	Iteratively runs ICP for the tree implementation of our program. Runs for a max of 30 iterations, exiting if the error is below our tolerance level. Finds the closest points c for d using the selected method.
get_error	Calculates the summed squared error between elements of F_reg and F_new
closest_point_sphere	Runs the closest point method using the bounding sphere implementation.
closest_point_triangle	Determines the closest point c on the given triangle to the given point s.
closest_point_linear	Determines the closest point using the brute force linear method.
get_distance	Determines the distance between points c and d.
calc_cr	Calculates the center and radius of the bounding sphere.
point_mean	Calculates the mean of the triangle coordinates
get_closest_point	Determines the closest point on the triangle to a given vector.
enlarge_bounds	Enlarges the triangle’s bounding box.
update_closest	Determines if the given point is in the bounding sphere, and if so find the closest point on the triangle to the given point.
find_closest_point_tree	Finds the closest point in the covariance tree to the given point, recursively calling on its subtrees.
create_cov_frame	Generates the initial frame of the covariance tree node.

find_cov_frame	Finds the frame of the given covariance tree node.
find_bounding_box	Finds the bounds of a bounding box around the given covariance tree node.
build_subtrees	Splits the node's triangles based on their x coordinate, and constructs subtree nodes from the resulting split.
inv	Inverts a given frame array
compose	Runs a composition of two input frames
transform3D	Runs a frame transformation on given point cloud a
get_transform	Determines the rotation and translation components of the transformation between the two provided point clouds.

The files making up the assignment 4 submission are as follows:

p4_driver.py	The program's main driver, which generates the output file for a given sample readings input.
ICPread.py	Contains all files for reading input data and generating arrays.
ICPmatch.py	Contains all files for find point cloud transformations and determining ICP matches. Added methods for iterative ICP and sphere method of finding closest point.
p4_tests.py	Contains test files for calculating d, finding the closest point in a triangle, and performing the full linear ICP matching, sphere ICP matching, and Covariance Tree ICP matching, as well as previous tests for point cloud transformations.
data_methods.py	Contains methods for computations and transformations on frames and point clouds.
sphere.py	Contains the sphere class holding a center point and radius, and the method for calculating these.
triangle.py	Contains the triangle class for determining the closest point on triangles with bounded spheres.
cov_tree_node.py	Contains the covariance tree node class and all essential functions discussed above.

Assignment 4 Run Instructions

To run the assignment 3 program, navigate to p3/programs/source/p3_driver.py. Enter the following to generate output data using the covariance tree method for the A-Debug sampleReadingsTest set:

python p4_driver.py A-Debug tree

The third argument from the user designates the ICP method they would like to use. Thus, to use the simple bounding spheres method one can write:

python p4_driver.py A-Debug sphere

As in assignment 3, provide the letter and type for each SampleReadingsTest dataset you would like to generate an output file for, such as B-Debug or G-Unknown. Each output file exactly follows the format specified in the assignment handout. The leftmost column lists the x,y,z coordinates of d, with the next column listing the coordinates of c, and the final column listing the magnitude of difference between the two sets of points.

Comments have been provided in each of the source files to outline the structure and flow of the program. After specifying an input dataset, the driver program sets the file paths for each of the required input files. The methods outlined in ICPread.py are called to load each of these data files into the necessary arrays. Next, icp_get_d is called to determine the position of the pointer tip with respect to the rigid body B. This is done by first calling get_transform to determine the rotation and translation transforms between the sample readings and rigid body point clouds for each frame, and then applying transform3D to get the final coordinates d.

The main changes in this assignment are the new icp_match_iterator method, and the three different implemented ICP methods. Depending on the icp_type selected before runtime, the icp_match_iterator method will first either construct the covariance tree or construct the list of bounding spheres. After this, it performs a maximum of 30 iterations, resetting the initial F_reg with each iteration, and exiting if the squared element-wise difference between the previous and current F_reg are within a specified threshold, in this instance being 1e-6. Depending on the selected icp_type, either closest_point_linear, closest_point_sphere, or closest_point_tree will be called in icp_match_iterator to find the new c approximation. The resulting c and F_reg are provided, after which the distances between the computed c and d points are calculated and the output files are generated.

To run the program’s unit tests, navigate to the source folder and enter:

python p4_test.py

Unit tests were written for each of the program’s core components and pass successfully, as detailed in the next section.

External Libraries

As stated above, the NumPy and SciPy libraries are used as our Cartesian math package. The specific submodules of NumPy used were np.linalg, np.matrix, and np.testing, and scipy.linalg.

5 Results and Discussion

5.1 Programming Assignment 1

Validation

We used several approaches to ensure the validation and accuracy of our program. Methods were developed incrementally, first verifying that dataset input and matrix generation was working as intended, before moving on to point cloud transformations and ultimately pivot calibration. To assess the accuracy of our final output, we generate not only an output1.txt file for each dataset, but also an output-difference.txt for each file to determine the accuracy of our C expectations. Pivot calibration output was also compared against the auxiliary files for each of the debug datasets. In addition to assessing final output, unit tests were written to ensure the accuracy of the program's subroutines. These include the dataset input, determining the rotation and translation matrices during point cloud registration, producing expected values from given transformations, and validating the pivot calibration output.

Point Cloud Registration

We have been able to ensure that point cloud registration is working properly by finding the transformation of one point cloud to another and then the opposite. Multiplying these two transformation matrices together resulted in an identity matrix, as was expected. We also tested the input data set, ensuring that our final output was within a reasonable tolerance range.

Pivot Calibration

The position of the tip of the probe when calibrated by RM gave results very similar to those listed in the auxiliary files for each dataset, well within our tolerance range. For further information on testing, please refer to the p1_tests.py file in the source folder. To view our output for each dataset, refer to the output folder.

Unknown Data Summary

The following table covers the em and opt pivot estimations, and average difference between the expected and listed C points, for each of the unknown datasets:

Dataset	EM Pivot	Optical Pivot	C Difference
unknown-h	[188.238, 231.994, 215.170]	[398.694, 393.650, 203.851]	20.525
unknown-i	[190.662, 204.362, 210.669]	[396.570, 398.576, 204.645]	17.336

unknown-j	[192.482, 194.217, 200.665]	[390.077, 397.445, 195.166]	16.636
unknown-k	[201.905, 187.854, 197.049]	[398.649, 412.293, 197.833]	16.509

For the debugging datasets where an auxiliary file was provided we were able to compare our EM and Optical pivot outputs against the provided coordinates. It was found that our output coordinates were very accurate, as shown below:

Dataset	EM Pivot	Optical Pivot
debug-a-out	[191.74, 200.48, 198.22]	[401.57, 392.03, 191.57]
debug-a-aux	[191.72, 200.48, 198,22]	[401.60, 392.03, 191.59]

Results were found to be similarly accurate for each of the output datasets. For further information, please refer to the driver's output in the output/ folder.

5.2 Programming Assignment 2

Distortion Correction

In addition to the previous unit tests from assignment 1, new unit tests have been added to verify that the calibrate_distortion and correct_distortion are working correctly. A distortion function may be applied to a dataset, followed by calling the distortion functions, and checking that the output is within a threshold of the initial pre-deformation dataset. Additionally, the test_correct_distortion function checks that the final probe tip positions in CT coordinates are within an error threshold of the output values provided. Here, our metric for error is the is the relative difference between our calculations and the debug output files. It was found that each individual point coordinate has a less than 20% difference from the debug output, with the vast majority of points having well below a 5% difference error.

For the assignment 2 datasets, error sources and propagation can come from a variety of sources, including EM distortion, EM Noise, and OT jiggle, as covered in the assignment outline. While we were able to account for the EM distortion through our distortion correction functions, it is expected that some amount of EM Noise, distortion, and jiggle will be propagated throughout the system that we are unable to account for. For instance, if both the optical and EM tracker are off with a common distortion component, this could cause the Bernstein curve to misestimate the actual curve, and consequently cause the registration between the CT scan and the other sensors to have a higher error.

Unknown Data Summary

The following tables list the estimated probe tip positions in CT coordinates returned by our p2_driver for each of the unknown datasets.

Dataset	Nav Point 1	Nav Point 2	Nav Point 3	Nav Point 4
g	119.01, 31.06,65.27	98.11, 171.4, 59.15	87.41, 131.41, 91.54	79.61, 96.98, 116.09
h	159.03, 119.39, 66.38	127.7, 103.74, 60.28	141.38, 152.01, 76.38	162.08, 162.18, 151.88
i	82.75, 38.98, 78.32	86.64, 29.3, 49.23	115.25, 109.68, 21.02	129.62, 171.1, 164.75
j	40.68, 66.52, 106.67	34.01, 109.05, 157.35	102.52, 157.02, 32.65	160.69, 28.51, 137.32

Each of these nav point coordinates was compared against the output file for each debugging dataset, and were also tested to be within a threshold of the output in the test_correct_distortion function, leading us to conclude the program was functioning properly. Assignment 1 and 2 outputs for each of the assignment 2 datasets may be found in the output/ folder.

5.3 Programming Assignment 3

Validation

Unit tests were written for each of the PA3 functions listed above with custom data inputs to ensure they were working as intended. Specifically, a test of the find_b_tip method is done by randomly generating rotation and translation transformations, and finding if the d output of the find_b_tip method is within a margin of error of the known d, as we have all the rotation and translation pieces and may directly calculate it. To test the findClosestPoint_brute method, we set a simple mesh of a single triangle, and ensure that the returned closest points for each given input s are as expected. To test the full icp_match function, we again construct a mesh of 9 triangles, and for given inputs s ensure the function output is as expected. Previously implemented tests for point cloud to point cloud transformations are also included. We found the single matching function to perform very quickly, however for PA4 data structures such as octree and bounded spheres will be implemented for time optimization over multiple iterations.

Output files were generated in the requested format, and we found that our output for dataset A yielded practically zero error, with the output for the remaining datasets being very similar to the provided answers. The output for debugging dataset A is as follows:

Dataset	D_Coords	C_Coords	Difference
A	-31.35 -6.20 -8.51	-31.35 -6.20 -8.51	0.001
	-6.48 11.34 45.49	-6.49 11.34 45.49	0.006
	13.91 9.46 -15.97	13.91 9.46 -15.97	0.002
	-31.30 4.63 -39.50	-31.30 4.64 -39.50	0.001

The following table shows the individual and average differences for each debug dataset:

A	B	C	D	E	F
0.001 0.006 0.002 0.001 0.001 0.001 0.002	1.342 1.350 0.108 1.818 0.747 2.138 0.892	1.159 0.167 2.943 0.977 1.702 0.615 2.945	1.651 1.795 0.124 3.566 1.572 1.467 0.744	3.792 2.773 1.059 0.198 4.174 4.893 2.924	2.156 1.881 1.285 0.929 4.481 0.518 1.226

0.001 0.001 0.002 0.000 0.001 0.001 0.003 0.002	1.756 0.969 2.547 0.028 3.264 3.483 3.300 2.714	2.514 1.811 0.100 1.155 2.035 2.139 1.536 1.719	3.107 1.735 0.322 2.550 1.553 1.042 1.994 2.589	2.825 3.826 4.221 2.198 0.204 2.668 3.588 1.418	5.143 0.089 1.844 0.770 1.098 1.796 1.257 0.883
0.00169	1.76367	1.56790	1.72074	2.71739	1.69049

We observed that our generated output and errors were very similar to those given in the -output and -answer files provided, further validating that our program is working correctly. While the distortion and noise of each input dataset are not provided, given this output we may assume that A-Debug contains no noise or distortion, explaining its near nonexistant calculation error, while each of the other Debug datasets contains some form of noise or distortion. These could include Optical distortion, Optical Noise, Optical Jiggle, CT distortion, or CT noise.

The first 4 lines of output for unknown datasets G, H, and J are as follows. For the full output, please refer to the output folder:

Dataset G	D_Coords	C_Coords	Difference
	-20.46 -9.23 -9.22	-23.40 -8.41 -5.27	4.986
	-3.29 -7.97 37.59	-2.68 -5.80 37.04	2.316
	23.58 -10.49 -23.56	23.51 -10.10 -23.15	0.567
	-9.80 -28.18 -42.15	-11.20 -25.60 -40.15	3.554

Dataset H	D_Coords	C_Coords	Difference
	9.64 0.64 63.63	9.64 0.61 63.24	0.401
	-2.33 17.22 51.73	-1.89 16.71 51.74	0.667
	6.68 -8.92 -22.45	6.96 -9.41 -23.19	0.933
	-19.86 -8.79 -49.10	-19.91 -8.81 -48.93	0.180

Dataset J	D_Coords	C_Coords	Difference
	-16.59 27.29 28.25	-13.40 23.38 28.08	5.049
	27.10 -0.33 16.10	28.82 -2.61 17.12	3.036
	13.88 21.83 -29.06	13.89 21.78 -29.02	0.061
	-2.40 -3.44 -28.54	-1.73 -3.04 -28.97	0.894

The following table shows the individual and average differences for each unknown dataset:

G	H	J
4.986	0.401	5.049
2.316	0.667	3.036
0.567	0.933	0.061
3.554	0.180	0.894
5.102	1.204	2.093
3.766	0.110	0.688
4.574	1.038	2.532
1.999	0.056	0.186
0.422	2.670	4.026
4.428	0.170	0.005
5.061	1.635	2.356
0.304	0.369	1.242
3.533	0.784	0.565
2.293	1.966	2.939
3.972	0.415	4.821
2.864	1.926	1.543
2.589	0.504	0.587
3.083	1.047	3.083
2.754	0.323	0.889
2.893	1.776	5.120
3.05295	0.90863	2.08567

5.4 Programming Assignment 4

Unit Tests

Unit tests were written for each of the primary P4 functions with custom data inputs to ensure they were working as intended. Specifically, a test of the find_b_tip method is done by randomly generating rotation and translation transformations, and finding if the d output of the find_b_tip method is within a margin of error of the known d, as we have all the rotation and translation pieces and may directly calculate it. To test the closest_point_linear method, we set a simple mesh of a single triangle, and ensure that the returned closest points for each given input s are as expected. The same is done with closest_point_sphere, checking to ensure that it outputs the same c as the linear method. Finally, the tree method is tested to have identical output as the linear and sphere methods, verifying that each are producing the correct output. Additionally, we ran the program with each of the icp types on the A and B-Debug datasets, and observed that the final output was identical in each of these instances. Thus, we can conclude that iteratively recorrecting F_reg and running the icp methods is yielding the same results across each method.

Stopping Criteria

We have two approaches for stopping criteria in ICP. Firstly, we set there to be a maximum of 30 iterations, as we found that there is generally a high level of convergence after this many iterations. Secondly, the method within_tolerance() summates the squared error between the 9 elements of F_reg and F_new's rotation matrix, as well as the 3 elements of their translation matrices. After each iteration, we check if this error is under 1e-6, in which case we return the current c and F_reg as the final output.

Output Data

Output files were generated in the requested format, and we found that our output for dataset A yielded practically zero error, with the output for the remaining datasets being very similar to the provided answers, much like in programming assignment 3. The following outputs were produced using our covariance tree method, with a maximum possible iterations of 30. The error for the first 20 points, as well as the average error, for each of the debug datasets is as follows:

A	B	C	D	E	F
0.004 0.000 0.001	0.007 0.001 0.002	0.008 0.003 0.005	0.002 0.001 0.005	0.005 0.065 0.179	0.083 0.113 0.020

0.004	0.002	0.003	0.005	0.240	0.011
0.002	0.002	0.006	0.003	0.009	0.051
0.000	0.003	0.002	0.013	0.071	0.018
0.001	0.003	0.003	0.002	0.070	0.066
0.006	0.001	0.003	0.012	0.047	0.007
0.001	0.005	0.006	0.004	0.084	0.022
0.002	0.005	0.005	0.008	0.038	0.171
0.001	0.004	0.002	0.014	0.167	0.079
0.005	0.004	0.005	0.002	0.047	0.026
0.001	0.002	0.004	0.007	0.044	0.060
0.002	0.002	0.008	0.002	0.039	0.053
0.002	0.001	0.003	0.004	0.028	0.009
0.001	0.002	0.003	0.008	0.104	0.067
0.002	0.002	0.008	0.005	0.098	0.011
0.002	0.003	0.003	0.002	0.011	0.077
0.003	0.003	0.009	0.004	0.026	0.099
0.003	0.004	0.012	0.003	0.172	0.168
0.002278307	0.003691584	0.006934245	0.005216394	0.071383804	0.063001328

We again assume that A-Debug contains no noise or distortion, while each of the other Debug datasets contains some form of noise or distortion. These could include Optical distortion, Optical Noise, Optical Jiggle, CT distortion, or CT noise. While in our PA3 output the D-F debug datasets yielded notably higher error, we observe here that the error is relatively consistent across each dataset, showing that the iterative ICP method helped to account for this.

The first 4 lines of output for the G-Unknown dataset is as follows. We see that there is minimal difference between C and D, and equivalent values were generated for each of our 3 icp types. For the full output, please refer to the output folder.

G-Unknown	D_Coords	C_Coords	Difference
	20.88 -0.92 55.40	20.06 0.11 55.56	0.016
	21.25 23.23 12.20	21.63 23.62 12.04	0.005
	24.31 4.94 -30.22	25.26 4.66 -30.06	0.010
	13.83 -7.17 -22.18	14.45 -7.18 -22.04	0.005

Finally, the first 20 lines of difference and average difference for each of the unknown datasets are as follows:

G	**H**	**J**	**K**
0.016 0.005 0.010 0.005 0.004 0.014 0.005 0.008 0.004 0.006 0.018 0.005 0.006 0.031 0.019 0.022 0.014 0.010 0.010 0.007	0.002 0.004 0.003 0.001 0.002 0.006 0.004 0.004 0.007 0.006 0.005 0.002 0.004 0.007 0.006 0.010 0.009 0.009 0.013 0.002	0.003 0.095 0.080 0.006 0.191 0.018 0.006 0.022 0.025 0.038 0.034 0.034 0.024 0.112 0.014 0.019 0.197 0.124 0.092 0.105	0.008 0.137 0.071 0.103 0.007 0.003 0.235 0.081 0.163 0.034 0.041 0.096 0.111 0.048 0.133 0.014 0.107 0.048 0.173 0.003
0.0105492203833	0.00457938697168	0.0611247439589	0.0680952270602

The error in the unknown datasets above appears to be consistent with those in the debug datasets, with the average errors varying closely with those of the debug sets.

Runtime

Finally, we observe how runtime differs across the three ICP methods. The following table shows the runtimes, in seconds, of each method on A-Debug and B-Debug:

Method	A-Debug	B-Debug
linear	100.530121088	+2000
sphere	14.8298611641	370.647783041
tree	8.15314292908	101.233943939

As expected, the covariance tree icp implementation yields the best times, followed by the bounding sphere and linear implementations. Their differences increase in B-Debug where there are more points and the added introduction of noise.

6 Additional Information

This program was developed collaboratively by Josh Shubert and Nathan Smith, with the vast majority of the program written together. For assignment 1, Nathan covered most of the file input and output, while Josh covered most of the pivot calibration functions, however all files were developed with input from both members. For programming assignment 2, all coding was completed together. For assignments 3 and 4, all programming was completed together.