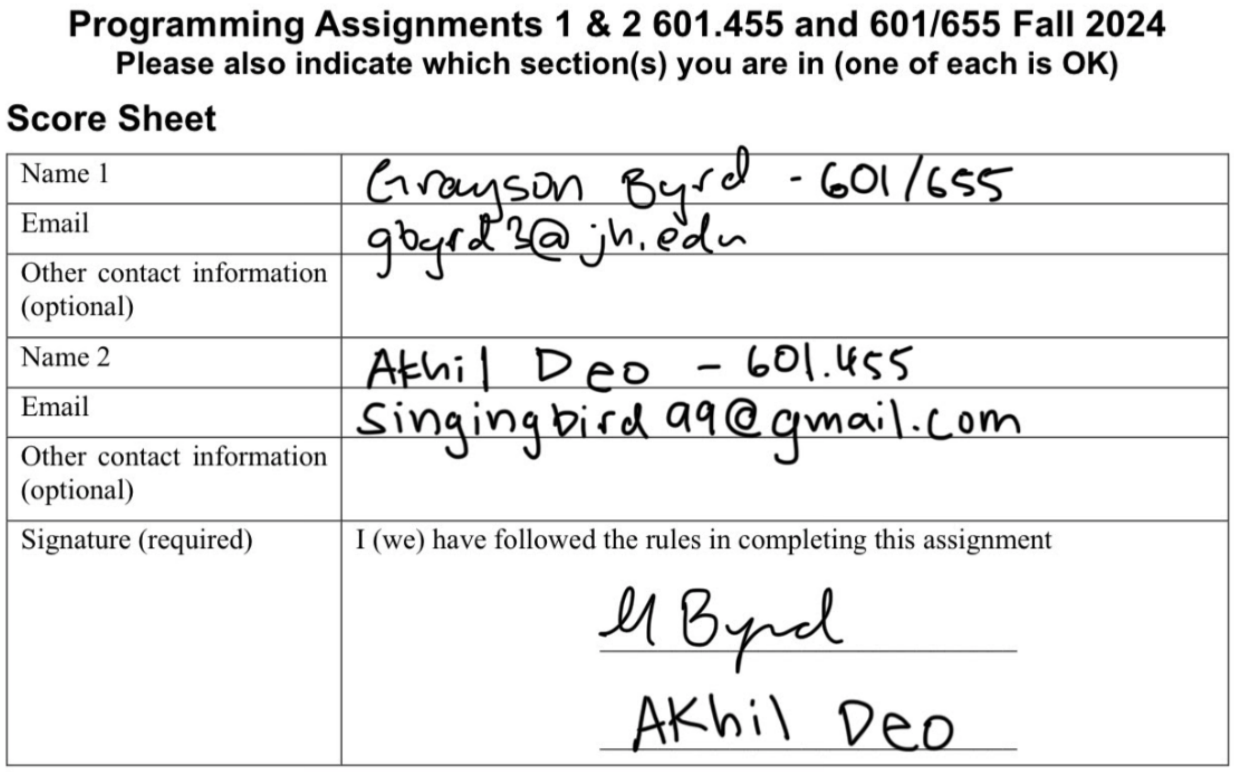
CIS PA2

Akhil Deo and Grayson Byrd

October 2024



Programming Assignments 1 & 2 601.455 and 601/655 Fall 2024
Please also indicate which section(s) you are in (one of each is OK)

Score Sheet

Name 1	Grayson Byrd - 601/655
Email	gbyrd3@jh.edu
Other contact information (optional)	
Name 2	Akhil Deo - 601.455
Email	singingbird99@gmail.com
Other contact information (optional)	
Signature (required)	I (we) have followed the rules in completing this assignment [illegible] AKhil Deo

Figure 1: ”I did not cheat” signature.

1 Introduction

In modern-day robotics systems, how robotics systems are registered and calibrated is crucial. This is especially true in surgical robotics, where even just slight registration or calibration errors could mean the difference between life and death. Unfortunately, distortion occurs, resulting in a warped space. In this assignment, we aim to properly “dewarp” the EM tracker space, and then use pivot calibration to calibrate the system now that we have a “dewarped” space. Once that is done, we will be able to properly compute the registration from the tracker space to the CT coordinate space.

The purpose of this assignment is to stress the importance of considering uncertainty in any robotic system that uses sensors to perceive and act in its environment. All sensors are subject to uncertainty and, in this assignment, we address the uncertainty in an electromagnetic tracking system introduced via magnetic distortion by computing a polynomial equation that models this distortion and using said polynomial equation to rectify measurements taken by the electromagnetic tracker. We leverage this distortion correction function to compute the registration between an electromagnetic tracker coordinate frame and a CT coordinate which we use to locate the tip of a pointer in the CT coordinate frame.

Specifically, we leverage a calibration object to gather ground truth and measured data points and use Bernstein polynomial interpolation to devise an equation that models the distortion between the ground truth and measured data. To do this, we first leverage our algorithms from PA1 to compute the ground truth locations of electromagnetic markers on the calibration object in the electromagnetic tracker frame by leveraging a highly accurate optical tracking system. We then use these ground truth point clouds as well as the electromagnetic tracker-measured point clouds at each frame to set up a system that we can solve to find a set of coefficients corresponding to a Bernstein polynomial equation that models the distortion in the system.

Next, we use this Bernstein polynomial to rectify all points measured in the electromagnetic tracker frame. For the remainder of this introduction, assume all measurements taken by the electromagnetic tracker have been rectified by our Bernstein polynomial distortion correction function. Using a pivot calibration dataset, we find the location of p_{tip} in relation to a local reference frame with origin at the centroid of the measured pointer electromagnetic marker point cloud. We then use this p_{tip} to touch several fiducials to get their measured locations in the electromagnetic coordinate frame. Given a dataset specifying the location of these fiducials in the CT coordinate frame, we can then compute the registration between the electromagnetic coordinate frame and the CT coordinate frame. Once this registration is known, we can then easily measure the location of the pointer tip in the electromagnetic tracker frame and then use our registration to transform the location of p_{tip} to the CT coordinate frame.

2 Mathematical Approach

2.1 Point Cloud Registration

The high level overview of our approach to the 3D point cloud to point cloud registration can be described by the following: given two point cloud sets of equal size with known correspondences, find the optimal transform, $F_{opt} = [R_{opt}, \vec{p}_{opt}]$, that minimized the mean squared error between the point cloud correspondences in the two sets. our approach was taken directly from [1].

I start with a pair point clouds of size N, p_i and p'_i where $i = 1, ..., N$. Each point cloud is measured in a different coordinate system and every ith point in one point cloud corresponds to the ith point in the other point cloud. we transform both point clouds to local coordinate frames centered at the centroid of the point cloud. To do this, first calculate the centroid of the point cloud, $p_{centroid}$.

$$p_{centroid} = (1/N)\sum_{i=1}^{N} p_i \tag{1}$$

Next, use the below equation transform the point cloud to the local coordinate frame centered at the centroid.

$$q_i = p_i - p_{centroid} \tag{2}$$

Where q_i is the ith point in the point cloud in the local coordinate frame centered at the point cloud's centroid.

Now that both point clouds are in the same coordinate frame, we can find the optimal rotation for the transformation between them by solving the following equation:

$$\arg\min_{R} \sum_{i=1}^{N} \|q'_i - Rq_i\|^2 \tag{3}$$

To solve this, we can use Singular Value Decomposition (SVD) as outlined in [1]. Begin by calculating the covariance matrix, H:

$$H = \sum_{i=1}^{N} q_i q_i'^T \tag{4}$$

Next, find the SVD of H:

$$H = USV^T \tag{5}$$

You can then calculate:

$$X = VU^T \tag{6}$$

Then, you calculate the determinant of X. If $\det(X) == +1$, then $R = X$. If $\det(X) == -1$, then the algorithm fails. In this case, we simply flip the third column of V, i.e. $V' = [v_1, v_2, -v_3]$ and then re-compute X as $X = V'U^T$. Then, we set $R = X$.

Finally, to get the translation, t, for the transformation, we simply use the following equation:

$$t = p'_{centroid} - Rp_{centroid} \tag{7}$$

2.2 Calculation of C Expected Values

This is the calculation of the expected EM marker positions on the calibration object. The objective of calculating $C_i^{expected}$ is to use these values as the ground truth for calibrating the distortion in the electromagnetic tracking frame. We can use this as ground truth since the optical tracker is considered highly accurate. To calculate the expected EM marker positions on the calibration object, we first used our 3D point cloud to 3D point cloud registration algorithm to calculate the transform from the electromagnetic tracking coordinate system to the optical tracking coordinate system, F_D^{-1}, by passing forming a pair of point clouds d_i, D_i and passing this to our registration algorithm. Next, we again used our 3D point cloud to 3D point cloud registration algorithm to calculate the transform from the calibration object coordinate system to the optical tracker coordinate system, F_A, by forming a pair of point clouds a_i, A_i and passing this to our registration algorithm. Finally, $C_i^{expected}$ could be found using the following equation:

$$\mathbf{C}_i^{(\text{expected})} = F_D^{-1} \cdot F_A \cdot \mathbf{c}_i$$

Where F_D^{-1} is the transform from the optical tracker frame to the electromagnetic tracker frame, F_A is the transform from the calibration object to the optical tracker frame, and $\mathbf{c}_i$ is the location of the electromagnetic markers on the calibration object with respect to the calibration objects coordinate frame.

2.3 Bernstein Polynomials for Distortion Correction

As stated in the programming assignment #2 instructions, there is distortion in the EM tracker space. As such, we need to create a solution to "dewarp" or correct this distortion in the EM tracker space. In order to do so, we follow the procedure detailed in class on the fifth slide of the Registration #3 slideshow [6].

Let $C_i^{(\text{measured})}$ be the measured EM marker positions, and $C_i^{(\text{expected})}$ be the expected positions calculated earlier. We aim to find a function f such that:

$$C_i^{(\text{expected})} = f(C_i^{(\text{measured})})$$

Given a dense enough dataset of measured and ground truth points, it is reasonable to interpolate between those points to model the distortion in the system. For this, we chose to use a seventh-degree Bernstein polynomials to interpolate between our data points to model the distortion. After we model the distortion, we can then correct for it. An n+1 Bernstein basis polynomials of degree, n, are defined as:

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad \text{for } we = 0, 1, \ldots, n \tag{8}$$

Where n is the degree of the polynomial, and $\binom{n}{i}$ is a binomial coefficient. A linear combination of Bernstein basis polynomials,

$$B_n(t) \sum_{i=0}^{n} \beta_i b_{i,n} \tag{9}$$

is called a Bernstein polynomial and is shown to model any curve defined by a set of points as its degree, n, goes to infinity [3]. The approximate curve generated by a Bernstein polynomial is continuous between sample points, making it a desirable equation for approximating/interpolating between points to calibrate for distortion.

For 3D Bernstein interpolation, we construct a "tensor form" interpolation polynomial using nth degree Bernstein polynomials, i.e.:

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

Where F is the Bernstein basis defined in Equation 8, i, j, and k are indexing variables each ranging from $0, ..., n$, and u_x, u_y, and u_z are the x, y, and z coordinates of the 3D sample points drawn from the distorted curve. Bernstein polynomials are only stable on the interval $[0, 1]$, thus we normalize all u values to be on this interval. Finally, we set up the following least squares problem and solve to get the coefficients of our Bernstein polynomial:

$$\begin{pmatrix} F_{000}(\vec{u}_s) & \cdots & F_{nnn}(\vec{u}_s) \end{pmatrix} \begin{pmatrix} c_{000}^x & c_{000}^y & c_{000}^z \\ \vdots & \vdots & \vdots \\ c_{nnn}^x & c_{nnn}^y & c_{nnn}^z \end{pmatrix} = \begin{pmatrix} & \vdots & \\ p_s^x & p_s^y & p_x^z \\ & \vdots & \end{pmatrix} \tag{10}$$

Where F_{ijk} is defined in Equation 8, c_{ijk}^a is the coefficient for the ijkth index of the a axis where a can be x, y, or z and p_s is the x, y, and z coordinates of the sample point.

For our Bernstein polynomial, we found through testing that a polynomial of degree 7 worked well across all datasets.

2.4 EM Probe Pivot Calibration using the Distortion Correction Function

Now that we have the distortion correction function, derived from the coefficients of the seventh-degree Bernstein polynomial discussed in the prior section, we can correct the distortions in the EM Probe Pivot calibration. For each frame we and each marker j, we correct the measured EM positions G_i^j using the distortion correction function f. This correction function is achieved by simply computing the F_{ijk} values for each new point and multiplying them by the computed coefficients from the calibration $C_i^{(expected)}$ values. Essentially, this correction function is just the left hand side of Equation 10. Thus,

$$G_{i,corrected}^{(j)} = f(G_i^j)$$

Once all the EM-Pivot frames are corrected, we can properly perform the pivot calibration with $G_{i,corrected}^{(j)}$, rather than G_i^j. Below is a description of how the EM probe pivot calibration is conducted with an array of frames G. In our workflow, these are corrected to become $G_{corrected}$ before the pivot calibration takes place.

2.4.1 EM probe pivot calibration

Our pivot calibration implementation was very straightforward and was taken directly from the CIS we lecture slides [6]. The objective of the EM probe pivot calibration is to compute the position of the dimple relative to the EM tracker and the position of the dimple relative to the probe body coordinate. Mathematically, we define an EM rigid body by computing the centroid($\vec{G_0}$) of the points relative to the EM tracker ($\vec{G_j}$) using the first "frame" data:

$$\vec{G_0} = \frac{1}{N_G}\sum_{i=1}^{N}\vec{G_j} \tag{11}$$

$$\vec{g_j} = \vec{G_j} - \vec{G_0} \tag{12}$$

Here, g_j represents the position of each EM marker relative to the chosen origin of the probe body coordinate system, which is G_0, the midpoint. We can then find a set of transforms between each subsequent frame of EM markers with this initial pointer reference frame. Given a list of N transforms, F_i for $i = 1, ..., N$, that were taken as a pointer probe was placed in a rigid dimple and rotated about its tip, you can get a system of equations and set up the least squares problem,

$$\begin{pmatrix} R_1 & -I \\ R_2 & -I \\ ... & ... \\ R_N & -I \end{pmatrix} \begin{pmatrix} \vec{p}_{tip} \\ \vec{p}_{dimple} \end{pmatrix} = \begin{pmatrix} -\vec{p}_1 \\ -\vec{p}_2 \\ ... \\ -\vec{p}_N \end{pmatrix} \tag{13}$$

Where R_i is the rotation component of the ith transform in the calibration frames, I is the identity matrix, $\vec{p}_{tip}$ is the location of the pointer tip in the pointer coordinate frame, $\vec{p}_{dimple}$ is the location of the calibration post dimple in F_{EM}, and $\vec{p}_i$ is the translation component of the ith transform in the calibration frames. To solve this, we simply set up the matrices as shown above and use an off the shelf least squares solver [2] and solve for $\vec{p}_{tip}$ and $\vec{p}_{dimple}$.

2.5 Calculation of the Fiducial Points Location w.r.t the EM Tracker Base Coordinate System

Now that we have the p_{tip} in the reference pointer marker frame, i.e. the x, y, and z position of the pointer tip from the centroid of the pointer markers when oriented as in the first pointer marker frame during the pivot calibrations, we need to use this to calculate the fiducials with respect to the EM tracker base. This involves transforming p_{tip} through a series of coordinate frames using point cloud registration and known correspondences between the reference pointer markers and the measured pointer markers in the EM tracker frame.

- The initial step is to transform the reference pointer marker positions to the local coordinate frame that was used as the initial local coordinate frame, $\vec{G_0}$, using Equations 11 and 12. The value for p_{tip}, solved during our pivot calibration, is measured from the origin of this local coordinate frame.

- Then, for each pointer marker point cloud measured when touching the pointer tip to the fiducial, we compute the respective local coordinate frame using Equations 11 and 12. We then compute a frame transformation using our point cloud to point cloud registration algorithm that maps the reference point cloud in local coordinate frame to each respective measured point cloud in local coordinate frame. We then apply this frame transformation to p_{tip}, compute the frame transform between the fiducial touching point cloud in the electromagnetic tracker frame and the fiducial touching point cloud in the local coordinate frame, and then used this frame transform to calculate the location of p_{tip} in the electromagnetic tracking frame. Mathematically, the location of p_{tip} in the electromagnetic tracking frame is:

$$\vec{p}_{fiducial} = F_{EM\_local} \cdot F_{local\_\vec{G}_0} p_{tip} \tag{14}$$

  Where $\vec{p}_{fiducial}$ is the location of the fiducial in the electromagnetic tracker coordinate system, F_{EM_local} is the frame transform from the local fiducial touching pointer coordinate frame to the electromagnetic coordinate frame, and $F_{local\_\vec{G}_0}$ if the frame transform from the local reference pointer coordinate frame to the local fiducial touching pointer coordinate frame.

2.6 Calculation of the Registration Frame

Once we compute the fiducial point locations with respect to the EM tracker base coordinate system, we conduct a basic point cloud to point cloud registration between the fiducial point location w.r.t the EM tracker base and the provided fiducial point locations w.r.t the CT coordinate system. The output of this is the registration frame transform. The math underpinning point cloud registration is described in 2.1.

2.7 Calculation of the Pointer Tip Location w.r.t CT image

The objective here is to compute the position of the pointer tip p_{tip} in the CT image coordinate frame. This involves transforming the pointer tip's position from the EM tracker frame, where it is measured, to the CT coordinate frame using a series of transformations and corrections.
First, we correct the measured positions of the pointer markers in the EM tracker frame in order to account for distortion.

$$G^{(j)}_{i,corrected} = f(G^j_i)$$

This process is described in Section 2.4 for a different set of data in the EM tracker base coordinate system.

We then compute p_{tip} w.r.t. the EM tracker frame using Equation 14. Here, the value, $p_{fiducial}$, in Equation 14 is the same location as p_{tip} in the EM tracker frame.

Then, we can apply the registration frame computed in 2.6 to calculate the pointer tip location with respect to the CT image. Let's assume that this registration frame between EM and CT is called F_{reg}, with its rotation matrix and translation vector called R_{reg} and t_{reg}, respectively.

$$p^{CT}_{tip} = R_{reg} p^{EM}_{tip} + t_{reg}$$

3 Description of Algorithmic Approach

Note, it is not necessary for me to go over every function that we wrote for this assignment with pseudocode, as several functions are trivial or not crucial for understanding the overall code implementation. For example, we wrote a helper function *write_3d_point_to_file* to trivialize logging a 3D point to an output file. we will not provide pseudocode for these types of functions, but will write short descriptions of this helper functions in Section 4.2. Here, we will provide pseudocode for the core functions that implement the mathematical approaches seen in Section 2. If you are looking through our code and see a function that we did not discuss in the report, that is because we deemed it a "helper" function and it was discussed in less detail in Section 4.2.

I implemented this code in Python 3.11. Below, we have broken up each major function into its own subsection to provide pseudocode and basic analysis. In our algorithms, we made extensive use of the NumPy [2], PyTest [4], and SciPy [5] libraries.

3.1 Utility Functions

The below functions serve as the core utility of the project and are constantly re-used. We also wrote a custom class called **FT** that implements basic frames transforms and inverse frame transforms. The pseudocode for these two functions is shown in this section as well.

Algorithm 1 Get Point Cloud in Local Frame

```
Input: A point cloud matrix pcd of size $(n, 3)$
Output: Centered point cloud pcd_local and centroid
centroid = mean(pcd, axis=0)
pcd_local = pcd - centroid
return pcd_local, centroid
```

Algorithm 2 Optimal Point Cloud Registration with Known Correspondence

```
Input: Target point cloud a, Source point cloud b
Output: Frame transform FT that aligns a to b
$(a\_local, a\_centroid) \leftarrow$ get_pcd_in_local_frame(a)
$(b\_local, b\_centroid) \leftarrow$ get_pcd_in_local_frame(b)
H $\leftarrow a\_local^T \cdot b\_local$                    ▷ Compute the covariance matrix
$(U, \Sigma, Vt) \leftarrow$ SVD(H)                         ▷ Perform Singular Value Decomposition (SVD)
R $\leftarrow Vt^T \cdot U^T$                               ▷ Compute the optimal rotation matrix
if det(R) $< 0$ then                                        ▷ Handle failure case.
    Vt[2, :] $\leftarrow -Vt[2, :]$
    R $\leftarrow Vt^T \cdot U^T$
end if
t $\leftarrow b\_centroid - (R \cdot a\_centroid^T)^T$      ▷ Compute the translation vector
return FT(R, t)
```

Algorithm 3 Pivot Calibration

```
Input: List of pointer marker point clouds pcd_frames in the tracker coordinate frame
Output: 3D locations p_tip in local pointer frame and p_dimple in tracker coordinate frame
FTs $\leftarrow$ empty list
$(pcd\_local, \_) \leftarrow$ get_pcd_in_local_frame(pcd_frames[0])    ▷ Get starting local pointer frame
for pcd in pcd_frames do                                     ▷ Get list of frame transformations for each frame
    FT $\leftarrow$ pcd_to_pcd_reg_w_known_correspondence(pcd_local, pcd)
    FTs.append(FT)
end for
x $\leftarrow$ run_pivot_least_squares(FTs)                  ▷ Solve least squares problem
p_tip $\leftarrow x[:3]^T$                                   ▷ Pointer tip in local pointer coordinate frame
p_dimple $\leftarrow x[3:]^T$                                ▷ Calibration dimple in tracker coordinate frame
return p_tip, p_dimple
```

Algorithm 4 Transform Points Between Coordinate Frames

```
Input: Set of points pts in one coordinate frame
Output: Transformed points in the target coordinate frame
transformed_pts $\leftarrow$ R $\cdot$ pts$^T$ + t
return transformed_pts
```

Algorithm 5 Inverse Transform Points Between Coordinate Frames

```
Input: Set of points pts in one coordinate frame
Output: Transformed points in the target coordinate frame using the inverse of the transform
transformed_pts $\leftarrow$ R$^T$ $\cdot$ pts$^T$ - R$^T$ $\cdot$ t$^T$
return transformed_pts
```

Algorithm 6 Normalize Point Cloud

```
Input: Point cloud pcd to normalize, minimum values min, maximum values max
Output: Normalized point cloud normalized_pcd
num_axes ← pcd.shape[1]
normalized_pcd ← np.zeros_like(pcd)
for $i = 0$ to num_axes - 1 do
    normalized_pcd[:, i] ← (pcd[:, i] - min[i]) / (max[i] - min[i])
end for
return normalized_pcd
```

Algorithm 7 Denormalize Point Cloud

```
Input: Point cloud pcd to denormalize, minimum values min, maximum values max
Output: Denormalized point cloud denormalized_pcd
num_axes ← pcd.shape[1]
denormalized_pcd ← np.zeros_like(pcd)
for $i = 0$ to num_axes - 1 do
    denormalized_pcd[:, i] ← pcd[:, i] * (max[i] - min[i]) + min[i]
end for
return denormalized_pcd
```

Algorithm 8 Compute Bernstein Polynomial

```
Input: Degree n, term index i, value x
Output: Bernstein polynomial value $B_i^n(x)$
B_value ← comb(n, i) · (x$^{i}$) · ((1 - x)$^{(n-i)}$)
return B_value
```

Algorithm 9 Construct Approximation Matrix for Bernstein Polynomial Interpolation

```
Input: Normalized point cloud pcd_norm, polynomial degree degree
Output: Approximation matrix A for Bernstein Polynomial Interpolation
assert np.min(pcd_norm) $\geq 0$ and np.min(pcd_norm) $< 1$
assert np.max(pcd_norm) $\leq 1$ and np.max(pcd_norm) $> 0$
num_samples ← pcd_norm.shape[0]
x_vals ← pcd_norm[:, 0]
y_vals ← pcd_norm[:, 1]
z_vals ← pcd_norm[:, 2]
A ← np.zeros((num_samples, (degree + 1) ** 3))
count ← 0
for $i = 0$ to degree do
    for $j = 0$ to degree do
        for $k = 0$ to degree do
            A[:, count] ← bernstein_poly(degree, i, x_vals) · bernstein_poly(degree, j, y_vals) ·
bernstein_poly(degree, k, z_vals)
            count ← count + 1
        end for
    end for
end for
return A
```

Algorithm 10 Compute Distortion Correction Coefficients using Bernstein Polynomial

```
Input: Ground truth points gt_pts, measured points measured_pts, polynomial degree degree, minimum values min, maximum values max
Output: Coefficients coeffs for Bernstein polynomial to correct distortion
gt_pts_norm ← normalize_pcd(gt_pts, min, max)
measured_pts_norm ← normalize_pcd(measured_pts, min, max)
A ← construct_approximation_matrix(measured_pts_norm, degree)
coeffs, _, _, _ ← np.linalg.lstsq(A, gt_pts_norm)
return coeffs
```

Algorithm 11 Apply Distortion Correction using Bernstein Polynomial

```
Input: Measured, distorted points measured_points, correction coefficients coeffs, polynomial degree degree, minimum values min, maximum values max
Output: Rectified (distortion-corrected) points rectified_points
assert measured_points.shape[1] == 3
assert $(\texttt{degree} + 1)^3 = $ coeffs.shape[0]
measured_points_norm ← normalize_pcd(measured_points, min, max)
A ← construct_approximation_matrix(measured_points_norm, degree)
rectified_points_norm ← A @ coeffs
rectified_points ← denormalize_pcd(rectified_points_norm, min, max)
return rectified_points
```

3.2 Main Functions

These functions are high level functions that are found in the *main.py* file. They utilize the lower level *Utility* functions to solve specific problems like the ones outlined in this assignment.

Algorithm 12 Compute C_i^{expected} for Calibration Dataset

```
Input: calbody_file_path, calreadings_file_path
Output: Computed $C_i^{\text{expected}}$ for each frame
data ← get_data
C_i_expected_frames ← empty list
for frame in calreadings do
    d_vals, a_vals, c_vals, D_vals, A_vals ← get_frame_data
    $F_{Dd}$ ← pcd_to_pcd_reg_w_known_correspondence(d_vals, D_vals)
    $F_{Aa}$ ← pcd_to_pcd_reg_w_known_correspondence(a_vals, A_vals)
    $C_i^{\text{expected}} \leftarrow F_{Dd}^{-1} \cdot F_{Aa} \cdot c_i$
    C_i_expected_frames.append($C_i^{\text{expected}}$)
end for
return np.array(C_i_expected_frames)
```

Algorithm 13 Get Distortion Calibration Coefficients for Bernstein Polynomial

```
Input: Calibration readings file path calreadings_path, expected frames C_i_expected_frames, polynomial degree degree
Output: Coefficients coeffs for Bernstein polynomial, global minimum global_min, global maximum global_max
calreadings ← parse_calreadings(calreadings_path)
C_i_measured ← np.array([x["C"] for x in calreadings])
C_i_expected ← C_i_expected_frames.reshape(C_i_expected_frames.shape[0] * C_i_expected_frames.shape[1], C_i_expected_frames.shape[2])
C_i_measured ← C_i_measured.reshape(C_i_measured.shape[0] * C_i_measured.shape[1], C_i_measured.shape[2])
min_expected ← np.expand_dims(np.min(C_i_expected, axis=0), axis=0)
max_expected ← np.expand_dims(np.max(C_i_expected, axis=0), axis=0)
min_measured ← np.expand_dims(np.min(C_i_measured, axis=0), axis=0)
max_measured ← np.expand_dims(np.max(C_i_measured, axis=0), axis=0)
global_min ← np.min(np.concatenate([min_expected, min_measured], axis=0), axis=0)
global_max ← np.max(np.concatenate([max_expected, max_measured], axis=0), axis=0)
pad_size ← (global_max - global_min) * 0.1
global_min ← global_min - pad_size
global_max ← global_max + pad_size
coeffs ← get_distortion_correction_coeffs_bernstein_3d(gt_pts=C_i_expected, measured_pts=C_i_measured, degree=degree, min=global_min, max=global_max)
return coeffs, global_min, global_max
```

Algorithm 14 Compute `p_tip` in Reference Pointer Frame

```
Input: empivot_file_path, Bernstein polynomial coefficients coeffs, polynomial degree degree, minimum values min, maximum values max
Output: p_tip (3D coordinates of p_tip in the EM tracker frame), reference_ptr_pcd (reference pointer point cloud)
empivot_cal_frames ← parse_empivot(empivot_file_path)
pcd_frames ← [x["G"] for x in empivot_cal_frames]
rectified_pcd_frames ← empty list
for pcd in pcd_frames do
    rectified_pcd ← apply_distortion_correction_bernstein(pcd, coeffs, degree, min, max)
    rectified_pcd_frames.append(rectified_pcd)
end for
p_tip, _, reference_ptr_pcd ← pivot_calibration(rectified_pcd_frames)
return p_tip, reference_ptr_pcd
```

Algorithm 15 Compute `p_tip` in Tracker Frame

```
Input: reference_ptr_pcd, pointer_markers_in_tracker_frame, p_tip_in_reference_pointer_frame
Output: p_tip_in_em_frame (location of p_tip in the tracker frame)
pointer_markers_in_local_pointer_frame, _ ← get_pcd_in_local_frame(pointer_markers_in_tracker_frame)
reference_ptr_pcd_in_local_pointer_frame, _ ← get_pcd_in_local_frame(reference_ptr_pcd)
FT_ptr_orientation ← pcd_to_pcd_reg_w_known_correspondence(reference_ptr_pcd_local, pointer_markers_in_local_pointer_frame)
p_tip_oriented ← FT_ptr_orientation.transform_pts(p_tip_in_reference_pointer_frame)
FT_pointer_from_tracker ← pcd_to_pcd_reg_w_known_correspondence(pointer_markers_local, pointer_markers_in_tracker_frame)
p_tip_in_em_frame ← FT_pointer_from_tracker.transform_pts(p_tip_oriented)
return p_tip_in_em_frame
```

Algorithm 16 Compute Fiducial Locations in EM Tracker Frame

```
Input: reference_ptr_pcd, em_fiducials_path, p_tip_in_pointer_frame, coeffs, degree, min, max
Output: em_fiducial_locations (locations of fiducials in EM coordinate frame)
em_fiducial_frames ← parse_em_fiducials(em_fiducials_path)
em_fiducial_locations ← empty list
for pointer_marker_pts in em_fiducial_frames do
    pointer_marker_pts_rectified ← apply_distortion_correction_bernstein(pointer_marker_pts, coeffs, degree, min, max)
    p_tip_in_em_frame ← compute_p_tip_in_tracker_frame(reference_ptr_pcd, pointer_marker_pts_rectified, p_tip_in_pointer_frame)
    em_fiducial_locations.append(p_tip_in_em_frame)
end for
return np.concatenate(em_fiducial_locations, axis=0)
```

Algorithm 17 Compute p_tip Locations in CT Frame

```
Input: em_nav_frames, p_tip_in_reference_pointer_frame, reference_ptr_pcd, FT_reg, coeffs, degree, min, max
Output: p_tips_in_ct_frame (locations of p_tip in CT frame for each navigation frame)
p_tips_in_ct_frame ← empty list
for ptr_marker_pts_em_frame in em_nav_frames do
    ptr_marker_pts_em_frame_rectified ← apply_distortion_correction_bernstein(ptr_marker_pts_em_frame, coeffs, degree, min, max)
    p_tip_em_frame ← compute_p_tip_in_tracker_frame(reference_ptr_pcd, ptr_marker_pts_em_frame_rectified, p_tip_in_reference_pointer_frame)
    p_tip_ct_frame ← FT_reg.transform_pts(p_tip_em_frame)
    p_tips_in_ct_frame.append(p_tip_ct_frame)
end for
return p_tips_in_ct_frame
```

Algorithm 18 Main Script for Programming Assignment #2

```
Input: dataset_prefix (prefix of the data to run, e.g., "pa1-debug-a-")
call validate_dataset_prefix(dataset_prefix)        ▷ Check if dataset prefix is valid
calbody_path, calreadings_path, empivot_path, em_fiducials_path, ct_fiducials_path, em_nav_path, output2_path ← get_data_paths(dataset_prefix)
C_i_expected_frames ← compute_C_i_expected(calbody_path, calreadings_path)
degree ← 5
coeffs, min, max ← get_distortion_calibration_bernstein_polynomial_coeffs(calreadings_path, C_i_expected_frames, degree)
p_tip_in_ref_pointer_frame, reference_ptr_pcd ← compute_p_tip_in_ref_pointer_frame(empivot_path, coeffs, degree, min, max)
fiducials_em_frame ← compute_fiducials_in_em_frame(reference_ptr_pcd, em_fiducials_path, p_tip_in_reference_pointer_frame, coeffs, degree, min, max)
fiducials_ct_frame ← parse_ct_fiducials(ct_fiducials_path)
FT_reg ← pcd_to_pcd_reg_w_known_correspondence(fiducials_em_frame, fiducials_ct_frame)
em_nav_frames ← parse_em_nav(em_nav_path)
p_tips_in_ct_frame ← compute_p_tips_in_ct_frame(em_nav_frames, p_tip_in_reference_pointer_frame, reference_ptr_pcd, FT_reg, coeffs, degree, min, max)
call save_to_output_file(dataset_prefix, p_tips_in_ct_frame)
```

4 Overview of Program Structure

```
main_PA2.py
|
|-- main(dataset_prefix)
|   |
|   |-- validate_dataset_prefix(dataset_prefix)
|   |-- get_data_paths(dataset_prefix)
|   |-- compute_C_i_expected(calbody_path, calreadings_path)
|   |   |
|   |   |-- parse_calbody(calbody_file_path)           [utils.data_processing]
|   |   |-- parse_calreadings(calreadings_file_path)   [utils.data_processing]
|   |   |-- For each frame in input calreadings.txt:
|   |       |
|   |       |-- pcd_to_pcd_reg_w_known_correspondence(d_vals, D_vals)
|   |       |   |
|   |       |   |-- get_pcd_in_local_frame(d_vals)      [utils.pcd_2_pcd_reg]
|   |       |   |-- Compute frame transformation w/ SVD [utils.pcd_2_pcd_reg]
|   |       |
|   |       |-- pcd_to_pcd_reg_w_known_correspondence(a_vals, A_vals)[utils.pcd_2_pcd_reg]
|   |       |-- Compute C_i_expected using transforms
|   |
|   |-- get_distortion_cal_bernstein_polynomial_coeffs(calreadings, C_i_expected, degree)
|   |   |
|   |   |-- parse_calreadings(calreadings)               [utils.data_processing]
|   |   |-- get_distortion_correction_coeffs_bernstein_3d()
|   |       |
|   |       |-- normalize_pcd()                          [utils.interpolation]
|   |       |-- construct_approximation_matrix()         [utils.interpolation]
|   |       |-- Solve least squares for coefficients
|   |
|   |-- compute_p_tip_in_pointer_frame(empivot_path, coeffs, degree, min, max)
|   |   |
|   |   |-- parse_empivot(empivot_file_path)           [utils.data_processing]
|   |   |-- For each frame:
|   |       |
|   |       |-- apply_distortion_correction_bernstein()
|   |           |
|   |           |-- normalize_pcd()                      [utils.interpolation]
|   |           |-- construct_approximation_matrix()     [utils.interpolation]
|   |           |-- Apply coefficients
|   |           |-- denormalize_pcd()                    [utils.interpolation]
|   |   |-- pivot_calibration()
|   |       |
|   |       |-- get_pcd_in_local_frame()              [utils.pcd_2_pcd_reg]
|   |       |-- For each frame:
|   |           |
|   |           |-- pcd_to_pcd_reg_w_known_correspondence()
|   |               |
|   |               |-- get_pcd_in_local_frame()              [utils.pcd_2_pcd_reg]
|   |               |-- Compute frame transformation via SVD  [utils.pcd_2_pcd_reg]
|   |       |-- run_pivot_least_squares()
|   |           |
|   |           |-- get_A_mat_for_pivot_cal_least_squares() [utils.pivot_cal]
|   |           |-- get_b_mat_for_pivot_cal_least_squares() [utils.pivot_cal]
|   |           |-- Solve least squares
|   |
|   |-- compute_fiducials_in_em_frame(...)
|   |   |
|   |   |-- parse_em_fiducials(em_fiducials_path)      [utils.data_processing]
|   |   |-- For each frame:
|   |       |
|   |       |-- apply_distortion_correction_bernstein()
|   |       |-- compute_p_tip_in_tracker_frame()
|   |           |
|   |           |-- get_pcd_in_local_frame()          [utils.pcd_2_pcd_reg]
|   |           |-- pcd_to_pcd_reg_w_known_correspondence()
|   |               |
|   |               |-- get_pcd_in_local_frame()      [utils.pcd_2_pcd_reg]
|   |               |-- Compute frame transform       [utils.pcd_2_pcd_reg]
|   |           |-- transforms points                 [utils.transform]
|   |
|   |-- parse_ct_fiducials(ct_fiducials_path)          [utils.data_processing]
|   |-- pcd_to_pcd_reg_w_known_correspondence(fiducials_in_em_frame, ct_fiducials)
|   |
|   |-- parse_em_nav(em_nav_path)                      [utils.data_processing]
|   |-- compute_p_tips_in_ct_frame()
|   |   |
|   |   |-- For each frame:
|   |       |
|   |       |-- apply_distortion_correction_bernstein()
|   |       |-- compute_p_tip_in_tracker_frame()
|   |           |
|   |           |-- get_pcd_in_local_frame()          [utils.pcd_2_pcd_reg]
|   |           |-- pcd_to_pcd_reg_w_known_correspondence()
|   |               |
|   |               |-- get_pcd_in_local_frame()      [utils.pcd_2_pcd_reg]
|   |               |-- Compute FT via SVD            [utils.pcd_2_pcd_reg]
|   |           |-- transform points                  [utils.transform]
|   |       |-- Transform p_tip_em_frame to CT frame using previously computed FT_reg
|   |
|   |-- parse_output_2(output2_path)                   [utils.data_processing]
|   |-- write_output2_to_file(...) [utils.data_processing]
|   |-- compute_debug_output_error()
|
|-- full_run()
|   |
|   |-- For each dataset prefix ("pa2-debug-a-" for example):
|       |
|       |-- main(prefix)
```

4.1 Description of Code Files

File Path from PROGRAMS Dir	Description
main_pa2.py	Functions for completing PA2 specific problems.
utils/data_processing.py	Functions for parsing the datasets.
utils/pcd_2_pcd_reg.py	Functions for 3D point cloud to 3D point cloud registration.
utils/pivot_cal.py	Functions for performing a pivot calibration.
utils/transform.py	Custom FT class used to perform frame transformation on 3D points.
utils/interpolation.py	Functions for conducting distortion correction
tests/test_main_with_debug_data.py	Testing for main.py file.
tests/test_utils.py	Basic helper functions for use in the test functions.
tests/test_registration_algorithm.py	Tests to validate 3D point cloud registration algorithm.
tests/test_pivot_cal_algo.py	Test functions to validate our pivot calibration algorithm.
tests/test_interpolation.py	Tests 3D Bernstein polynomial interpolation

Table 1: Description of each code file

4.2 Description of Helper Functions

Function Name	Description
parse_$< dataset_prefix >$	Takes dataset_prefix and returns the loaded data of that file
save_to_output_file	Takes dataset prefix, p_{tip}^{CT} and saves formatted to output2.txt file
write_3d_point_to_file	Takes in a 3D point and .txt file and saves the 3D point to the .txt file
compute_debug_output_error	Computes the MSE between the generated and test debug output2.txt file

Table 2: Description of each helper functions

4.3 Reusability and Modularity

We made our code modular and reused it constantly throughout this project in many instances. The code found in the PROGRAMS/utils/data_processing.py file provides functions to parse the various datasets in the DATA/ directory. Instead of re-writing the parsing whenever we needed it, we wrote the function once, tested it, and re-used it. We did the same thing with our point cloud registration function, using it consistently in the testing functions and several functions in the main.py file. we also did this with the pivot calibration function, the function for getting the point cloud in the local frame, as well as all of the functions used for interpolation and distortion correction. These functions are described in the Algorithmic Approach section. The modularity of these functions is also shown in this section. Additionally, we wrote several small helper scripts that were too trivial/extraneous to include in the algorithmic approach section, which we've reused constantly throughout the codebase.

5 Discussion of Validation Approach

To ensure that our code worked consistently and our mathematical approach was well founded, we exhaustively tested our code via unit tests. To do this, we implemented the testing using PyTest [4]. If you go into our directory, follow the install instructions, activate the environment, and run pytest, all tests described in this section will pass. This ensures that our implementation is correct and that even the low level utility functions are validated appropriately. Below, we describe each test in detail.

5.1 Point Cloud Registration Algorithm Testing

To test our 3D point cloud to 3D point cloud registration algorithm we wrote several tests. The first test was a test with custom data augmented with noise. Here is the procedure for testing:

1. Create a random target point cloud and random coordinate transformation. we used SciPy [5] to get a random rotation matrix.
2. Compute the source point cloud by transforming the target point cloud i.e. $pcd_{source} = Fpcd_{target}$
3. Add random noise to the target point cloud
4. Compute the predicted transform via our registration algorithm.
5. Assert that the error is below a certain threshold

I also implemented this test without noise augmentation. This implementation was exactly the same as the prior steps except it omitted step 3. Additionally, the error threshold was much, much smaller (essentially 0).

5.2 Pivot Calibration Algorithm Testing

To test our pivot calibration algorithm, we created custom data and used it to validate our algorithm in PyTest. This pseudocode describes this test.

Algorithm 19 Test Pivot Calibration with Custom Data

```
**Step 1:** Define test values for $p_{tip}$ and $p_{dimple}$
$FTs \leftarrow$ empty list
**for** $k = 1$ to $N$ **do**
    Generate random $R_k$ from uniform distribution over $[0, 360]$
    Set $t_k \leftarrow p_{dimple} - R_k \cdot p_{tip}$
    Set $F_k \leftarrow [R_k, t_k]$
    Append $F_k$ to $FTs$
**end for**
**Step 2:** Run pivot calibration to recover $p_{tip}$ and $p_{dimple}$
$x \leftarrow$ run_pivot_least_squares($FTs$)
$p_{tip\_pred} \leftarrow x[:3]$
$p_{dimple\_pred} \leftarrow x[3:]$
**Step 3:** Compute mean squared error (MSE) between true and predicted values
$mse_{p_{tip}} \leftarrow$ compute_avg_mse_between_two_pcds($p_{tip\_pred}, p_{tip}$)
$mse_{p_{dimple}} \leftarrow$ compute_avg_mse_between_two_pcds($p_{dimple\_pred}, p_{dimple}$)
**Step 4:** Assert that MSE is close to zero
**assert** np.isclose($0, mse_{p_{dimple}}$)
```

5.3 Distortion Calibration Testing ** Custom Dataset Creation **

To test our distortion calibration method, a custom dataset was created. In order to create our custom dataset, we first calculated the $C_i^{(expected)}$ values and loaded the $C_i^{(measured)}$ values from the dataset. We then took a random sample of 80% of the $C_i^{(expected)}$ and $C_i^{(measured)}$ values and used these as "train" values. We then took the remaining 20% of values and used this as "test" values. We computed the Bernstein polynomial for distortion correction using only the "train" set. We then used this polynomial to correct the values of the $C_i^{(meausured)}$ values in the "test" set. We compared these corrected $C_i^{(measured)}$ values to the $C_i^{(expected)}$ values in the "test" set and computed the error between the corrected measured and expected values. Using the pytest framework, we ensured that the mean squared error for each dataset in the DATA folder was below 0.5, which validated our approach, as it is statistically very unlikely that we would achieve such low mean squared error without our method being sound. Psuedocode for this test is provided below.

Algorithm 20 Test Bernstein Interpolation with Train/Test Split

```
Initialize mse_list as an empty list
for dataset_prefix in dataset_prefixes do
    call validate_dataset_prefix(dataset_prefix)
    dataset_folder ← path to data folder
    calbody_path ← dataset_folder + "/" + dataset_prefix + "calbody.txt"
    calreadings_path ← dataset_folder + "/" + dataset_prefix + "calreadings.txt"
    C_i_expected_frames ← compute_C_i_expected(calbody_path, calreadings_path)
    calreadings ← parse_calreadings(calreadings_path)
    C_i_measured ← array of C values from calreadings
    C_i_expected ← reshape C_i_expected_frames
    C_i_measured ← reshape C_i_measured
    train_size ← $0.8 \cdot$ len(C_i_expected)
    train_idxs ← random sample of indices for training set of size train_size
    test_idxs ← indices not in train_idxs
    C_i_expected_train ← C_i_expected[train_idxs]
    C_i_expected_test ← C_i_expected[test_idxs]
    C_i_measured_train ← C_i_measured[train_idxs]
    C_i_measured_test ← C_i_measured[test_idxs]
    min_expected ← minimum values along each axis of C_i_expected
    max_expected ← maximum values along each axis of C_i_expected
    min_measured ← minimum values along each axis of C_i_measured
    max_measured ← maximum values along each axis of C_i_measured
    global_min ← minimum values between min_expected and min_measured
    global_max ← maximum values between max_expected and max_measured
    best_degree ← None
    min_mse ← 10000000
    for degree from 0 to 9 do
        coeffs ← get_distortion_correction_coeffs_bernstein_3d(C_i_expected_train, C_i_measured_train, degree, global_min, global_max)
        rectified_C_i_measured_test ← apply_distortion_correction_bernstein(C_i_measured_test, coeffs, degree, global_min, global_max)
        mse_rectified ← mean squared error between C_i_expected_test and rectified_C_i_measured_test
        if mse_rectified < min_mse then
            best_degree ← degree
            min_mse ← mse_rectified
        end if
        print mse_rectified
    end for
    coeffs ← get_distortion_correction_coeffs_bernstein_3d(C_i_expected_train, C_i_measured_train, best_degree, global_min, global_max)
    rectified_C_i_measured_test ← apply_distortion_correction_bernstein(C_i_measured_test, coeffs, best_degree, global_min, global_max)
    mse_rectified ← mean squared error between C_i_expected_test and rectified_C_i_measured_test
    mse_list.append(mse_rectified)
end for
assert maximum value in mse_list < 0.5
```

5.4 Full Testing with Debug Output Files

To test our full program pipeline, we used the full main() function to generate output files for each of the debug datasets. we then ran the following code to get the difference between each numerical value corresponding the x, y, and z value for each corresponding point in the debug dataset. we then asserted that the difference was less than 0.5 for every single data point, thus ensuring our approach and implementation was correct and free from errors.

6 Discussion of Results

To determine the accuracy of our work, we calculated the Mean Squared Error (MSE) between each coordinate in the 3D points output in the predicted output file (<dataset_prefix>output2) and in the debug datasets. The table below shows the results of this error calculation for each of the debug data sets. For all the debug datasets, the mean squared error is quite small (<0.01), indicating that our results are highly accurate.

Dataset Prefix	Mean Square Error
pa2-debug-a	$1.74e^{-5}$
pa2-debug-b	0.0034
pa2-debug-c	0.00013
pa2-debug-d	$1.41e^{-5}$
pa2-debug-e	0.00409
pa2-debug-f	0.0083

Table 3: Error analysis

The results of our approach can be shown in the OUTPUT/<dataset_prefix>output2.txt file for each dataset prefix (unknown and debug). The data from the "unknown" datasets is presented in table 3.

Dataset Prefix	Generated Data
pa2-unknown-g	104.84, 107.54, 57.87 99.23, 128.33, 120.99 87.84, 73.47, 159.67 133.1, 75.96, 37.85
pa2-unknown-h	132.55, 137.03, 128.02 142.52, 108.89, 93.87 53.04, 106.01, 117.72 173.0, 84.41, 38.04
pa2-unknown-i	57.31, 142.71, 163.47 118.94, 49.79, 89.13 120.74, 63.32, 168.77 90.34, 160.53, 117.26
pa2-unknown-j	42.48, 37.99, 146.67 63.89, 54.02, 88.96 67.8, 168.64, 149.14 153.55, 105.4, 99.43

Table 4: Generated Data from "Unknown" Datasets

Since we do not know the ground truth of the unknown datasets, we cannot perform direct numerical analysis on them. However, the values do seem reasonable after conducting a basic visual inspection. Based on our results from the debug datasets and the testing and validation conducted on the functions in our workflow, we are confident that our approach will operate properly on unseen data like the unknown datasets.

7 Who Did What

Grayson and Akhil each contributed equally to this work. Grayson spent most of his time writing the code base, and Akhil helped with debugging, unit testing, and validation of the code base. Both Grayson and Akhil worked on the report together. Most sections of the report were done together, and we did not tend to split of sections to do independently. Overall, both team members were happy with the contributions of the other.

References

[1] K. S. Arun, T. S. Huang, and S. D. Blostein. Least-squares fitting of two 3-d point sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9(5):698–700, 1987.

[2] NumPy, Numerical Python. http://www.numpy.org.

[3] Bernstein Polynomial. Bernstein polynomial — Wikipedia, the free encyclopedia, 2024. [Online; accessed 3-November-2024].

[4] PyTest, Python Testing. https://docs.pytest.org/en/stable/.

[5] SciPy, Scientific Python. http://www.scipy.org.

[6] Russell H. Taylor. 600.455/655 lecture notes, 1996-2024.