Programming Assignment 2

I. Overview

Problem to be solved: Something about achieved fiducials or bs. We followed the outline suggested on the document

With the nature of EM distortion being largely translational, we aim to correct _____ with polynomial fitting. As presented in class, Bernstein Polynomial fitting was chosen because it is bounded, smooth, and relatively numerically stable for our targeted spatial distortion.[Citation] An interesting aspect of this problem is that the fit, and therefore correction success, of the polynomial model depends heavily on the order of the polynomial. This is almost like an additional degree of freedom we must optimize. We chose an iterative method to select the optimal order of our Bernstein Polynomial model by gradually increasing the order and determining the RMS error between $C_{expected}$ and $C_{measured,corrected}$. This process iterates until we resolve that we have reached a convergence (see Algorithmic Approach section).

Goals:

Brief Summary of successes

As a note, our PCR and Pivot calibration algorithms were successful in PA1, so they are unchanged. The testing validation methods for both are identical as well compared to PA1. Some sections of our transformation chains in main_2.py are unchanged because they were proven to work in PA1.

II. Mathematical Approach

Point Cloud Registration:

With two point clouds $\boldsymbol{a_i}, \boldsymbol{b_i} \in \mathbb{R}^3$; $i = 1, \dots, n \in \mathbb{Z}$

Our goal is to find a frame transformation $F = [R, \boldsymbol{p}]$ such that $\boldsymbol{b_i} = F \cdot \boldsymbol{a_i} = R \cdot \boldsymbol{a_i} + \boldsymbol{p}$

The error or distance between corresponding points within the point clouds can be written as a least squares problem:

1. For each corresponding point:

$$e(R, \boldsymbol{p}) = \|\boldsymbol{b} - (R\boldsymbol{a} + \boldsymbol{p})\|^2$$

2. To minimize the error, we differentiate and set to zero:

$$\frac{\partial e(R, \boldsymbol{p})}{\partial \boldsymbol{p}} = -2(\boldsymbol{b} - R\boldsymbol{a} - \boldsymbol{p}) = 0$$

Solving for $\boldsymbol{p}$ allows us to isolate translation away from rotation. $\boldsymbol{p} = \boldsymbol{b} - R\boldsymbol{a}$

Now, all that is left is to solve for R

1. We remove parameter $\boldsymbol{p}$ from the error equation:

$$e(R) = \|\boldsymbol{b} - (R\boldsymbol{a} + (\boldsymbol{b} - R\boldsymbol{a})\|^2$$

2. Notice that this equals 0. This makes sense as rotation between two individual points cannot be resolved. This is why we need sufficient points in each point cloud.
3. Lets add in the rest of the point clouds:
   a. We start by centering the two point clouds, ie removing the translational aspect, as explained by Taylor[1].

$$\boldsymbol{a_i} = \boldsymbol{a_i} - \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{a_i}\,, \qquad \boldsymbol{b_i} = \boldsymbol{b_i} - \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{b}$$

   b. We then rewrite out error function considering all corresponding points.

$$e(R) = \frac{1}{n}\sum_{i=1}^{n} \|\boldsymbol{b_i} - (R\boldsymbol{a_i})\|^2$$

4. The Kabsch algorithm solves the error minimizing least squares function using SVD, as explained by Stachniss and Lawrence, et al. It is also referenced in Taylor's slides from Arun, et al.
   a. Create a cross-covariance matrix

$$BLAH$$

   b. Compute SVD pseudo inverse

$$BLAH$$

c. Solve for R

$$BLAH$$

d. Verify $R \in SO(3)$. We check our computed R matrix for two distinct edge cases, when we produce a reflection $\det(R) = -1$, or we have degenerate data to produce a singular case $\det(R) = 0$. Both cases are handled in our implementation.

Pivot Calibration:

The goal here is that given multiple point clouds taken from the trackers on a tool while rotating it on its tip about a fixed pivot point, we can find the location of the pivot point in the tracker base's frame, as well as the offset of the tool tip in the tool frame.

1. First step is data collection. The setup describes each of the tool's having trackers in the handle, which is what the trackers (both EM and optical) get readings of. The one important thing to note here is that every reading is taken with the tool tip positioned in the same pivot point, which is fixed in the tracker base's frame. All raw data was provided as part of the assignment.
2. For each frame, we use the point cloud registration algorithm described previously to find the frame transforms from the tracker base frame to tool frame. Let $\boldsymbol{F_1} \dots \boldsymbol{F_j}$ be the calculated frame transforms from $\boldsymbol{j}$ point cloud registrations of the same set of tool trackers, where $\boldsymbol{F_i} = [R_i, \boldsymbol{p_i}]$.
3. Let
   $\boldsymbol{p_{world}}$ = position of tool tip in world frame
   $\boldsymbol{p_{tip}}$ = position of tool tip in the tool frame
   $\boldsymbol{p_{pivot}}$ = position of the pivot in the world frame
   We can see that for any of the calculated frame transforms,
   $$\boldsymbol{p_{world}} = R_i \cdot \boldsymbol{p_{tip}} + \boldsymbol{p_i}$$
   But from out data collection procedure, we can see that $\boldsymbol{p_{world}} = \boldsymbol{p_{pivot}}$, which gives
   $$\boldsymbol{p_{pivot}} = R_i \cdot \boldsymbol{p_{tip}} + \boldsymbol{p_i}$$
   Rearranging:
   $$R_i \cdot \boldsymbol{p_{tip}} - \boldsymbol{p_{pivot}} = -\boldsymbol{p_i}$$
   After computing j unique frame transforms, we can stack the measurements together to arrive at this system of equations:
   $$\begin{bmatrix} R_1 & -I \\ \dots & \dots \\ R_j & -I \end{bmatrix} \begin{bmatrix} \boldsymbol{p_{tip}} \\ \boldsymbol{p_{pivot}} \end{bmatrix} = \begin{bmatrix} -\boldsymbol{p_1} \\ \dots \\ -\boldsymbol{p_j} \end{bmatrix}$$
   We can see that to solve for p_{tip} and p_{pivot}, we just need a way to solve this $Ax = b$ problem.
4. Since this is a linear system, we can use a least squares method to solve x. In our case, we decided to use SVD based least squares solver, which outputs the final values of $\boldsymbol{p_{tip}}$ and $\boldsymbol{p_{pivot}}$

Distortion Correction: BPoly

Given measured and expected point clouds with direct correspondence, we attempt to solve for a correction point cloud such that:

$$expected \approx measured + correction$$

1. We choose a Bernstein Polynomial to produce this correction component. This is beacue _____. All Bernstein bounded from $[0,1]$, so we normalize our input to the bounds. Additionally, through testing, we add padding. Initially, we clipped any slight deviations above the strict bounds but this led to numerical instability in the unit testing so the padding method was chosen and proved to be successful.

$$BLAH$$

2. Next, since we have spatial point clouds, we start by constructing three, one-dimensional Bernstein bases based on the input data $\alpha \in norm(x, y, z)$. This is based on BPoly order n. [CITATION]

$$B(t) = \binom{n}{i} \alpha^i (1-\alpha)^{n-i}, for\ i = 0, \ldots, n$$

3. Then, with three, one-dimensional bases, we construct the three-dimensional Bernstein basis A that determines how our polynomial evaluates at each point of the point cloud.

$$BLAH$$

4. Finally, we define $correction \doteq AC$, where A is our Bernstein basis and C a matrix of coefficients for our Bernstein Polynomial. We attempt “fit” to the polynomial by minimizing correction and error.
   a. This becomes the classic $Ax = B$ problem that can be solved many numerical methods.

$$argmin\|AC - (expected - measured)\|^2 \rightarrow ((A^T A)C = A^T(expected - measured)$$

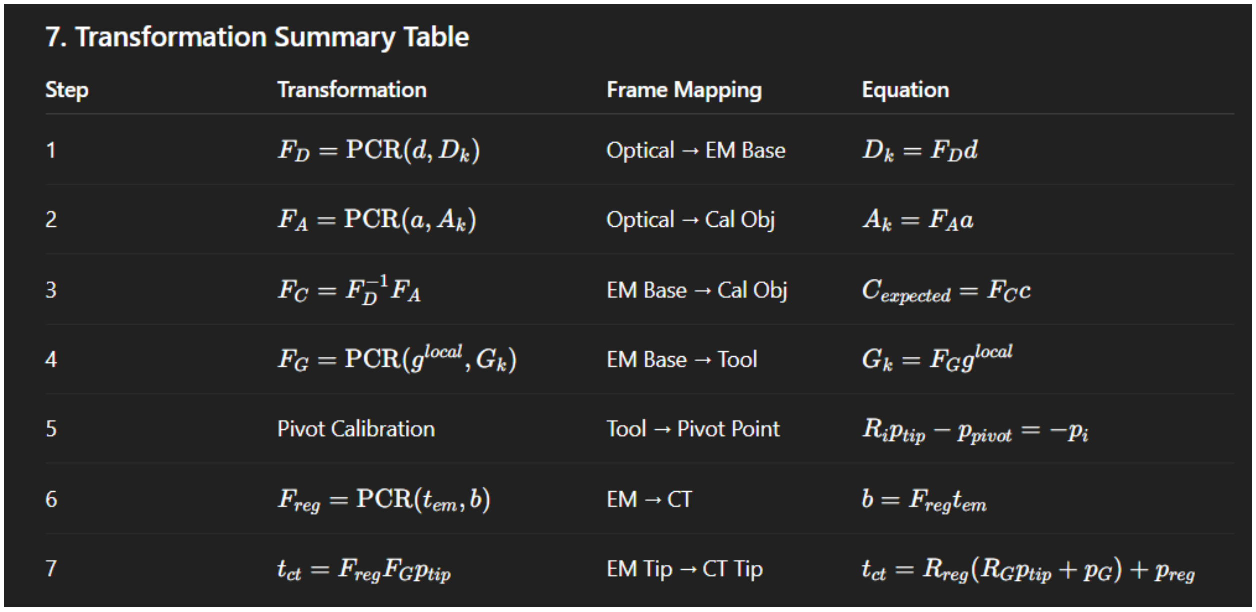
   b. However, due to reasons explain in Algorithmic Approach and Validation Approach, we observed numerical instability, even with the Bernstein Basis. This was resolved using a method known as “Tikhonov Regularization”.

$$(A^T A + \lambda I)C = A^T(expected - measured)$$

To apply the correction AC, all we must do is generate a new A basis matrix based on the given point cloud, and then add the correction point cloud.

$$measured + correction = measured + A_{new}C$$

Frame Transformations:



7. Transformation Summary Table

Step	Transformation	Frame Mapping	Equation
1	$F_D = \mathrm{PCR}(d, D_k)$	Optical → EM Base	$D_k = F_D d$
2	$F_A = \mathrm{PCR}(a, A_k)$	Optical → Cal Obj	$A_k = F_A a$
3	$F_C = F_D^{-1} F_A$	EM Base → Cal Obj	$C_{expected} = F_C c$
4	$F_G = \mathrm{PCR}(g^{local}, G_k)$	EM Base → Tool	$G_k = F_G g^{local}$
5	Pivot Calibration	Tool → Pivot Point	$R_i p_{tip} - p_{pivot} = -p_i$
6	$F_{reg} = \mathrm{PCR}(t_{em}, b)$	EM → CT	$b = F_{reg} t_{em}$
7	$t_{ct} = F_{reg} F_G p_{tip}$	EM Tip → CT Tip	$t_{ct} = R_{reg}(R_G p_{tip} + p_G) + p_{reg}$

III. Algorithmic Approach

This project was written in python 3.11.9 and utilizes numpy[citation], pandas[citation], matplotlib[citation], python.os, python.math, python.sys libraries.

Point Cloud Registration

```
Algorithm: Point Cloud Registration
Inputs:
Outputs:  A rigid transformation HTM H
1:
2:
3:
4:
5:
6:
7:
8:
9:
10:
11:
12:
13:
14:
15:
```

Between PA1 and PA2, this algorithm remained largely the same, aside from additional logic to handle the $\det(R) = 0$ singular case.

Pivot Calibration

```
Algorithm: Pivot Calibration
Inputs:
Outputs:  That p_tip translational offset??????
1:
2:
3:
4:
5:
6:
7:
8:
9:
10:
11:
12:
13:
14:
15:
```

Bernstein Polynomial Fitting

```
Algorithm: BPoly Fitting
Inputs:     Expected and Measured nx3 point clouds, BPoly Order
Outputs:    Assign private fields: BPoly Coeffient Matrix
1:          Computer residual: delta=Expected-Measured
2:          Normalize measured point cloud
3:          for x in range 0 -> n
4:              for y in range 0 -> n
5:                  for z in range 0 -> n
6:                      Construct Bx 1D Bernstein basis
7:                      Construct By 1D Bernstein basis
8:                      Construct Bz 1D Bernstein basis
9:                      Construct A 3D Bernstein basis
10:                 endfor
11:             endfor
12:         endfor
13:         define noise = near zero number
14:         solve (A+noise)x=b problem for coefficient matrix C
15:         assign self.coeff = C
```

In initial testing of the BPoly fitting and application, we generated very large coefficients on a few of the data sets. This resulted in divergence as BPoly order increased, not convergence towards a minimized error. After testing the identity functionality of our BPoly function (see Validation Approach section), we realized that our initial least squares minimizing solution with “clean” data sets caused the divergence. After some research, we found that added noise by a method called “Tikhonov Regularization” helped stabilize the least squares solution. [Citation]

$$(A^T A)\boldsymbol{c} = A^T \boldsymbol{b} \rightarrow (A^T A + \lambda I)\boldsymbol{c} = A^T \boldsymbol{b}, \text{ where } \lambda \text{ is tiny.}$$

Bernstein Polynomial Application

```
Algorithm: BPoly Apply
Inputs:     Distorted nx3 point cloud, BPoly Coeff Matrix
Outputs:    Corrected nx3 point cloud
1:          Normalize distorted point cloud
2:          for x in range 0 -> n
3:              for y in range 0 -> n
4:                  for z in range 0 -> n
5:                      Construct Bx 1D Bernstein basis
6:                      Construct By 1D Bernstein basis
7:                      Construct Bz 1D Bernstein basis
8:                      Construct A 3D Bernstein basis
9:                  endfor
10:             endfor
11:         endfor
12:         correction = A @ C
13:         assign corrected point cloud = distorted point cloud + correction
14:         return corrected point cloud
```

BPoly Order Selection/Optimization

```
Algorithm: BPoly Order Selection/Optimization
Inputs:     Expected and Measured nx3 point clouds, Distortion Threshold, Convergence Threshold
Outputs:    Best BPoly Obj, Best Order
```

```
1:        for order = 1 -> 99
2:            Fit BPoly with order with Measured and Expected
3:            Corrected = Apply BPoly Corrections to Measured
4:            Calculate RMS_Error between Corrected and Expected
5:            if RMS_Error < Best_RMS
6:                Best_RMS = RMS_Error
7:                Best_Order = order
8:                Best_BPoly = BPolt
9:            endif
10:           if (Prev_RMS – RMS_Error) < Convergence Threshold
11:               break
12:           endif
13:           Prev_RMS = RMS_Error
14:       endfor
```

Main Transformations

Algorithm: Main Transformation Chain

```
Inputs:
Outputs:  Output-1.txt and Output-2.txt
1:
2:
3:
4:
5:
6:
7:
8:
9:
10:
11:
12:
13:
14:
15:
```

IV. Overview of Structure

The source code is structured in a way that promotes code reuse and easy debugging. As each algorithm was developed, multiple helper scripts were created in ./utils. Additionally, each algorithm was thoroughly tested with scripts in ./tests. Each test is fully self-contained, so simple navigate to the ./tests directory and run the test either from command line or in your development environment. If the test is utilizing data from ./data, it will navigate to the correct directory automatically. Main_2.py is the primary script for PA2. Running main_2.py it will produce all outputs for all datasets.

[HEIARCHY GRAPH DIAGRAM THING FUCK]

Main Scripts

main_2.py

Iteratively loops through all datasets from debug A-F to unknown G-J and follows the steps outlined in the PA2 document. For convenience, step number is clearly declared in comments. For each dataset, corresponding output-1 and output-2 txt files are produced and saved.

main_2_plot.py

A direct copy of the main_2.py algorithm, but with plotting functionality to visualize distortion correction.

Utility Scripts

./utils/parse.py

```
function [d, a, c] = parse_calbody(path)
function [D_frames, A_frames, C_frames] = parse_calreadings(path)
function G_frames = parse_empivot(path)
function [D_frames, H_frames] = parse_optpivot(path)
function b_ct = parse_ctfiducials(path)
function G_frames = parse_emfiducials(path)
function G_frames = parse_emnav(path)
function [C_expected_frames, p_post_em, p_post_opt] = parse_output_1(path)
function p_ct = parse_output_2(path)
```

All the parse functions follow the structure outlined in the PA document. Most read the header line and extract parameters like number of frames or markers and then read and reshape the following data within each provided txt file. These functions use pandas to parse and strip the formatted txt file inputs and numpy to convert the character data to floats/ints.

./utils/plot.py

```
function [fig, ax] = plot_data_1(data, var_name, number_points)
function [fig, ax] = plot_data_2(data1, data2, var_name1, var_name2, number_points)
function [fig, ax] = plot_data_error_vectors(data1, data2, var_name1, var_name2)
```

plot_data_1 and plot_data_2 are used extensively during debugging and development to visualize a point cloud. They plot one point cloud and two point clouds, respectively. This offers a visualize guide to look at registration alignment or distortion.

plot_data_error_vectors takes two point clouds and draws error vectors proportional to the error between point clouds. This is helpful to visualize before/after distortion correction as well as the shape of any uncorrected distortion.
Matplotlib is used to visualize and plot.

./utils/calculate_errors.py

```
function rms_error = calculate_rms_error(pc1, pc2)
function [F_diff, angle_error, translation_error] = calculate_error_transformation(Fa, Fb)
function stats = calculate_error_stats(pc1, pc2)
function print_error_stats(stats)
```

The calculate_errors.py script contains a suite of helpful error calculations for evaluating our algorithmic approach. Calculate_erorr_stats() is the primary function that, given two point clouds, computes mean, max, min, std, and rms error.
Calculate_error_transformation() is used primarily on evaluating PCR which outputs a HTM. It calculates angular and translational error between two HTMs.

./utils/write_out.py

```
function write_output_pa1(C_expected_frames, p_post_em, p_post_opt, output_path)
function write_output_pa2(tip_positions_ct, output_path)
```

Both write_output functions take the necessary frames, coordinates, and translations and write out a txt in the format specified on the PA2 document.

./utils/calibrator.py

```
function F = point_cloud_registration(a, b)
function [p_tip, p_pivot] = pivot_calibration(T_all)
```

The Calibrator object has two primary functions. point_cloud_registration() takes two point clouds and outputs the optimal rigid transformation between the two. It assumes both point clouds have point-to-point correspondence and a rigid transformation (point clouds cannot be scaled).
Pivot_calibration() computes the translational tool tip offset in the tool frame and as a bonus, the pivot point dimple in the world frame. The inputs are a series of well-distributed frames/poses of the tool.

./utils/bpoly.py

```
function B = bernstein_1d(n, norm_points)
function A = bernstein_3d(n, norm_point_cloud)
function fit(measured, expected)
function corrected_points = apply(point_cloud)
```

The BPoly object has two primary functions. fit() takes two corresponding point clouds that are expected to close. It then computes a pair-data error pointcloud and a 3D Bernstein basis based on the normalized error pointcloud. It then solves a regularized least squares problem to fit the error to the basis and assigns a coefficient matrix, a private field of this object.
Apply() requires that fit() has already been called and the coefficient matrix exists. It then takes a new point clouds, normalizes it, and then constructs a new 3D Bernstein basis. It then applies the “trained” coefficient matrix to compute a paired correction output a “de-warped” the point cloud.

Testing Scripts

./tests/bpoly_test.py

```
function stats = identity_bpoly_test(order)
function stats = random_bpoly_test(order)
```

Two primary unit tests are done on the BPoly class. The first identity_bpoly_test ensures that fitting and applying our Bernstein Polynomial to an identity mapping does not cause warping, scaling, or clipping, under a small error threshold. Essentially, a non-distorted data set should see the BPoly function as a identity transformation. The second random_bpoly_test generates known distortion between two random point clouds and verifies that BPoly can be fit to the distortion and the original random point cloud can be recovered. BPoly fit depends on a specified order so we sweep the order from 1-9 to observe underfitting and wellfitting.

./tests/pcr_test.py

```
function [C_expected_pc, C_frames] = find_expected_calibration_object(calbody_path, calreadings_path)
function [F_diff, angle_error, translation_error] = random_pcr_test()
```

./tests/pivot_test.py

```
function [tip_translation_error, pivot_translation_error] = test_pivot_calibration()
function em_pivot_calibration_test()
function opt_pivot_calibration_test()
```

./tests/compare_outputs.py

```
function compare_output_1(letter)
function compare_output_2(letter)
```

Datasets A-F have sample “ground truth” outputs that allow us to compare our algorithm’s generated output to a sample. This test script compares output 1 and output 2, for each data set, to the sample output and computes error metrics (see Validation Approach section) between the “ground truth” output and our generated output.

V. Validation Approach

Point Cloud Registration

To validate our Point Cloud Registration algorithm, we generated two identical point clouds and applied a known rigid-body transformation F_{ab} to one of them. The validation goal was to determine whether our PCR implementation could recover this known transformation within an acceptable error threshold.

Using the error transformation equations and solving for translational and angular error, we can quantify an error.

$$\Delta F = F_a^{-1} F_b = [\Delta R, \boldsymbol{\Delta p}],\ e_\theta = \cos^{-1}(\frac{tr(\Delta R)-1}{2}),\ e_t = \|\boldsymbol{\Delta p}\|$$

Successful validation is achieved as long as e_θ and e_t are acceptably low.

Special consideration was given to whether it would be beneficial to test the PCR algorithm under extreme rotational, translational, or otherwise edge-case transformations. However, after reviewing the mathematical properties of the Kabsch algorithm and related literature, we found that PCR will always converge to a valid rigid transformation given sufficient, non-degenerate point correspondences. As such, additional stress testing was deemed unnecessary.

Table. PCR Errors	
Angle Error ($\boldsymbol{\theta}$)	**Trans Error (mm)**
2.1073424255447017e-08	3.5214902285927624e-14

Pivot Calibration

For the Pivot Calibration algorithm, validation followed a similar principle of using a known transformation. In this case, we generated synthetic probe motion data by applying a known pivot position and tool tip offset F_{tip} across multiple frames. The objective was to verify that the least-squares pivot calibration method could recover the same pivot position relative to the tracker base.

Since pivot calibration estimates only the translational components, validation focuses on the translation difference:

$$e_t = \|\boldsymbol{p_{known}} - \boldsymbol{p_{estimate}}\|$$

A small translational error indicates that the pivot calibration procedure correctly determines both the tool-tip location in the local tool frame and the fixed pivot point in the tracker frame.

Table. Pivot Calibration Errors	
Tip Trans Error (mm)	**Pivot Trans Error (mm)**
2.2842349715679205e-15	1.6764000044290905e-15

Bernstein Polynomial Distortion Correction

To ensure the correctness and numerical stability of our Bernstein Polynomial distortion model, we designed two complementary validation tests: an identity test and a random fit test. These tests confirm that module is working before using BPoly for distortion correction in the EM calibration workflow. Since the performance of the correction is dependent on the order of the BPoly, we also sweep the order from 1-9 for each of these tests.

The identity test validates that our BPoly mapping does not add unnecessary distortion or scaling. We generate a random 3D point cloud $X \in [0,1]^3$and fit the BPoly function such that it maps X to itself. Applying the fitted model back to X should ideally reproduce the same point cloud. We then compute the RMS and maximum errors. If those errors fall below a threshold, we consider the test passed.

The second validation introduces a known nonlinear distortion to random data. We then fit BPoly and undistort the data. This determines if the mapping approximates the known function. If the resulting RMS error falls below a threshold, we validate this test.

Table. BPoly Identity Transformation Test

Order	Mean Error (mm)	RMS Error (mm)	Std Dev	Min Error	Max Error	Pass?
1	0.000000	0.000000	0.000000	0.000000	0.000000	Yes
2	0.000000	0.000000	0.000000	0.000000	0.000000	Yes
3	0.000000	0.000000	0.000000	0.000000	0.000000	Yes
4	0.000000	0.000000	0.000000	0.000000	0.000000	Yes
5	0.000000	0.000000	0.000000	0.000000	0.000000	Yes
6	0.000000	0.000000	0.000000	0.000000	0.000000	Yes
7	0.000000	0.000000	0.000000	0.000000	0.000000	Yes
8	0.000000	0.000000	0.000000	0.000000	0.000000	Yes
9	0.000000	0.000000	0.000000	0.000000	0.000000	Yes

Table. BPoly Random Fitting Test

Order	Mean Error (mm)	RMS Error (mm)	Std Dev	Min Error	Max Error	Pass?
1	0.018597	0.018950	0.003641	0.003526	0.025038	Yes
2	0.021726	0.022232	0.004716	0.002670	0.034748	Yes
3	0.023900	0.024580	0.005743	0.001961	0.036964	Yes
4	0.024175	0.024804	0.005547	0.001007	0.034265	Yes
5	0.024223	0.024841	0.005506	0.001840	0.034410	Yes
6	0.024223	0.024844	0.005520	0.001823	0.034459	Yes
7	0.024225	0.024844	0.005510	0.001768	0.034433	Yes
8	0.024225	0.024844	0.005511	0.001767	0.034435	Yes
9	0.024225	0.024844	0.005511	0.001768	0.034435	Yes

We can see that for the random test, our order 1 BPoly has the lowest mean and it rises are order increases, approaching a convergence as order approaches 9. Calculations for RMS, mean, and std error are below:

$$e_{rms} = \sqrt{\frac{1}{N}\sum_{i}^{N}\left\|pc_1^{(i)} - pc_2^{(i)}\right\|^2}\,, mean = \frac{1}{N}\sum_{i}^{N}\left\|pc_1^{(i)} - pc_2^{(i)}\right\|\,, std = \sqrt{\frac{1}{N}\sum_{i}^{N}(pc_1^{(i)} - mean)^2}$$

Overall Algorithmic Workflow

Debug Data Sets

The debug sets are accompanied with sample output data that we can compare to. These serve as "ground truths" that we can use to verify the overall pipeline: PCR, pivot calibration, distortion correction, and more registration. To automate the comparison, we created the compare_outputs.py script that calculates error statistics using calculate_errors.py, namely, RMS error.

Table. Comparison of Output Files Without Distortion Correction				
Data Set	Pivot EM Error (PA1) (mm)	Pivot Opt Error (PA1) (mm)	RMS Reg Error (PA1) (mm)	RMS Tip Position Error (PA2) (mm)
A	0.000000	0.000000	0.004710	0.008536
B	0.094340	0.000000	0.502218	0.027752
C	2.692452	0.014142	0.522093	2.172187
D	0.000000	0.000000	0.012221	0.005000
E	7.195561	0.000000	1.577330	3.727905
F	4.655985	0.000000	1.705271	2.322526

Table. Comparison of Output Files With Distortion Correction				
Data Set	**Pivot EM Error (PA1) (mm)**	**Pivot Opt Error (PA1) (mm)**	**RMS Reg Error (PA1) (mm)**	**RMS Tip Position Error (PA2) (mm)**
A	0.000000	0.000000	0.004710	0.008536
B	0.094340	0.000000	0.502218	0.027752
C	**0.024495**	0.014142	0.522093	**0.027866**
D	0.000000	0.000000	0.012221	0.005000
E	**0.120830**	0.000000	1.577330	**0.122705**
F	**0.260192**	0.000000	1.705271	**0.239766**

Our validation shows that with distortion correction, we have extreme reduction in error. However, we see that RMS Reg Error has some large differences with the sample sets, especially when we have distortion in the data sets. [POTENTIAL ERROR]

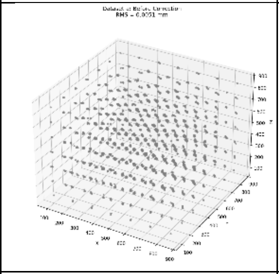
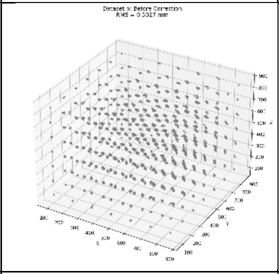
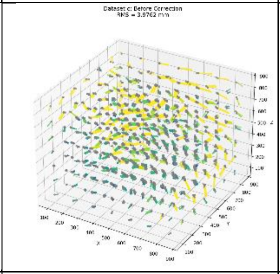
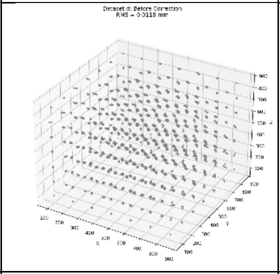
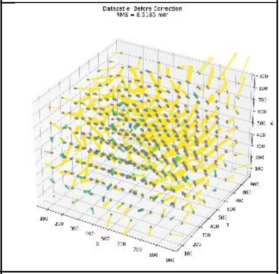
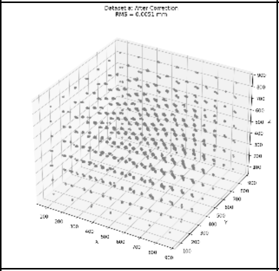
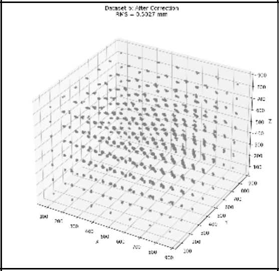
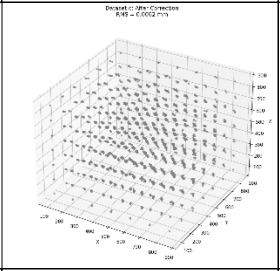
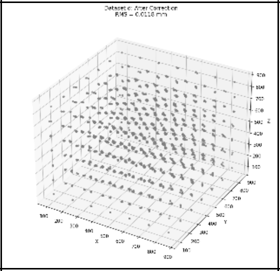
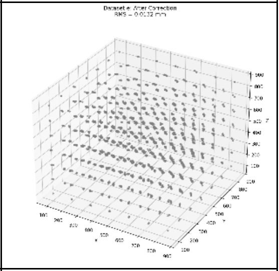
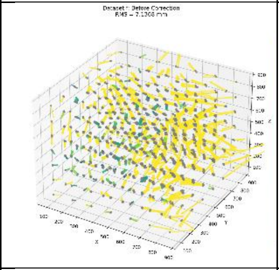
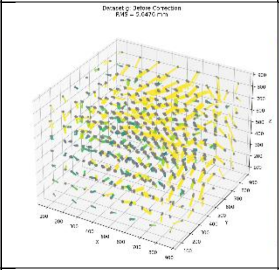
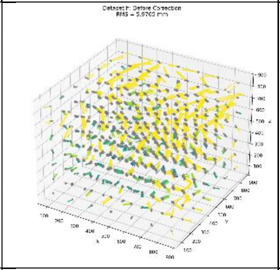
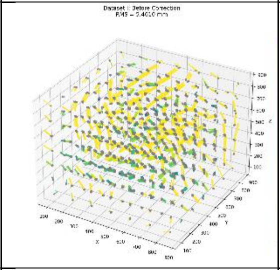
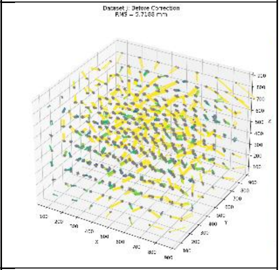
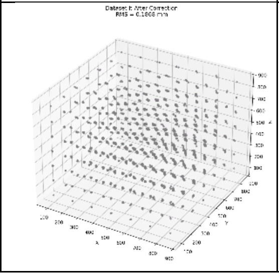
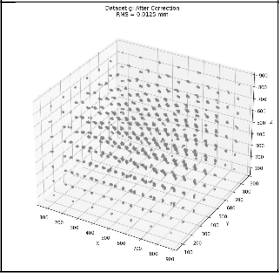
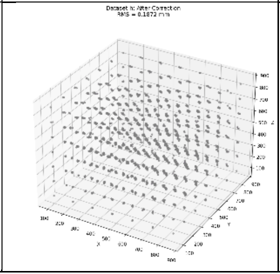
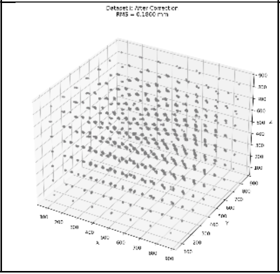
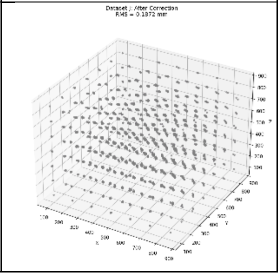
Unknown Data Sets

Since it is hard to find a "ground truth" for the unknown data sets, the best we can do is to make determinations of our distortion correction by comparing to the $C_{expected}$ frames, which still are affected by noise and jiggle. Therefore, the best way to quantify how our algorithm is doing is to look at the RMS error between $C_{expected}$ and $C_{measured,corrected}$.

Table. Convergence of RMS Error on Unknown Data Sets											
	RMS Error (mm) per BPoly Order										
Data Set	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
G	4.9926	3.2580	2.2409	0.1683	0.0678	0.0368	0.0279	0.0217	0.0175	0.0150	0.0135
H	5.5123	3.5698	2.4569	0.2602	0.2141	0.1978	0.1935	0.1908	0.1893	0.1881	0.1872

I	4.9781	3.1467	2.0584	0.2333	0.2019	0.1933	0.1911	0.1892	0.1879	0.1869	0.1860
J	5.2651	3.5601	2.5400	0.2964	0.2154	0.1989	0.1942	0.1912	0.1894	0.1881	0.1872

Notice that all 4 unknown sets reach convergence at around the same RMS value. This indicates that this residual likely is the effect of noise or jiggle, not of distortion.

Table: Before vs After Distortion Correction for Datasets A-J					
	A	**B**	**C**	**D**	**E**
Before Distortion Correction					
After Distortion Correction					
	F	**G**	**H**	**I**	**J**
Before Distortion Correction					
After Distortion Correction					

Accompanying our numerical error metrics, we present before- and after-distortion correction error-vector plots for all data sets. We can see that distortion is largely corrected, as the error vectors significantly decrease in both magnitude and variance. Note that the remaining vectors are almost invisible and do not form patterns like distortion, indicating that the residual error is dominated by sensor noise rather than systematic distortion.

VI. Results

TODO: Yap about how close our debugs are to the sample debug outputs

Below, we show again our results on the debug and unknown data sets.

Table. Comparison of Output Files With Distortion Correction

Data Set	**Pivot EM Error (PA1) (mm)**	**Pivot Opt Error (PA1) (mm)**	**RMS Reg Error (PA1) (mm)**	**RMS Tip Position Error (PA2) (mm)**
A	0.000000	0.000000	0.004710	0.008536
B	0.094340	0.000000	0.502218	0.027752
C	**0.024495**	0.014142	0.522093	**0.027866**
D	0.000000	0.000000	0.012221	0.005000
E	**0.120830**	0.000000	1.577330	**0.122705**
F	**0.260192**	0.000000	1.705271	**0.239766**

Table. Convergence of RMS Error on Unknown Data Sets

	RMS Error (mm) per BPoly Order										
Data Set	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
G	4.9926	3.2580	2.2409	0.1683	0.0678	0.0368	0.0279	0.0217	0.0175	0.0150	0.0135
H	5.5123	3.5698	2.4569	0.2602	0.2141	0.1978	0.1935	0.1908	0.1893	0.1881	0.1872
I	4.9781	3.1467	2.0584	0.2333	0.2019	0.1933	0.1911	0.1892	0.1879	0.1869	0.1860
J	5.2651	3.5601	2.5400	0.2964	0.2154	0.1989	0.1942	0.1912	0.1894	0.1881	0.1872

VII. Works Cited

[1] Taylor, R. H. (n.d.). Point cloud to point cloud rigid transformations [PowerPoint slides]. Computer Integrated Surgery I (601.455/655), Johns Hopkins University, Whiting School of Engineering. Laboratory for Computational Sensing and Robotics.

[2] Murray, Richard M. (2012). A mathematical introduction to robotic manipulation. CRC Pr I Llc 2012.

[3] Stachniss, C. (2021). ICP & Point Cloud Registration – Part 1: Known data association & SVD [Video]. YouTube.

[4] Lawrence, J., Bernal, J., & Witzgall, C. (2019, October 9). A purely algebraic justification of the Kabsch-Umeyama algorithm. Journal of Research of the National Institute of Standards and Technology, 124, 124028.

[5] K. Arun, et. al., IEEE PAMI, Vol 9, no 5, pp 698-700, Sept 1987

[6] Taylor, R. H. (n.d.). Calibration [PowerPoint slides]. Computer Integrated Surgery I (601.455/655), Johns Hopkins University, Whiting School of Engineering. Laboratory for Computational Sensing and Robotics.

[7] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. Nature 585, 357–362 (2020.

[8] McKinney, W. (2010). Data structures for statistical computing in python. In S. van der Walt & J. Millman (Eds.), Proceedings of the 9th Python in Science Conference (pp. 51–56).

[9] J. D. Hunter, "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007PCR: Taylor Slides and Kabsch SVD, ICP & Point Cloud Registration - Part 1: Known Data Association & SVD (Cyrill Stachniss, 2021), A Purely Algebraic Justification of the Kabsch-Umeyama Algorithm, K. Arun, et. al., IEEE PAMI, Vol 9, no 5, pp 698-700, Sept 1987

[10] BPoly slides

[11] BPoly Princeton formula

[12] BPoly Regularization

[13]