CIS I Programming Assignment 1

Student Grayson Byrd

Name 1	Grayson Byrd
Email	gbyrd23@jh.edu
Other contact information (optional)	
Name 2	
Email	
Other contact information (optional)	
Signature (required)	I (we) have followed the rules in completing this assignment M Byrd

Figure 1: "I did not cheat" signature.

Introduction

The purpose of this assignment is to serve as a gentle introduction into the fundamental concepts of the field of computer integrated surgery. The key topics explored in this assignment include calibration of a system via 3d pointcloud to 3d pointcloud registration and pivot calibration as well as using this calibration to build transformation chains between various coordinate systems to predict the location of a point in one coordinate system from its location in another coordinate system.

Specifically, in this assignment we are confronted with a system consisting optical and electromagnetic trackers, a pointer with optical markers, a pointer with electromagnetic markers, a workspace with several dimpled calibration posts, and a calibration object that contains both optical and electromagnetic markers. The location of all optical markers are measured by the optical tracker in the local coordinate frame of the optical tracker. Similarly, all electromagnetic markers are measured by the electromagnetic tracker in the local coordinate frame of the electromagnetic tracker. In this assignment, we assume that the optical tracker can measure the location of optical markers with a very high level of accuracy, and expect to see no geometric error from the optical tracker. In contrast, the measurements provided by the electromagnetic tracking system are subject to random distortions.

In next two paragraphs, I will provide the variable names describing various components of the system described at a high level in the previous paragraph. For clarity, I will keep these variables the same as they are described in [5]. The calibration object contains N_C electromagnetic markers at known positions, $\vec{c}_i$ for $i = 1...N_C$, relative to the calibration object's coordinate system, F_C. Likewise, the calibration object also contains N_A optical LED markers at known positions, $\vec{a}_j$, also in F_C. Elsewhere in the system, there are N_D optical LED markers that are placed at known positions, $\vec{d}_j$ in the electromagnetic tracking system's base coordinate system, F_{EM}. The location of the LED markers on the calibration object in the optical tracker's base coordinate system will be denoted as $\vec{A}_j$. The location of $\vec{d}_j$ in the optical tracker's base coordinate system will be denoted as $\vec{D}_j$. The locations of $\vec{c}_i$ measured from the electromagnetic tracker's coordinate frame will be denoted as $\vec{C}_i$. The distortion of the sensor system is large and can be up to several millimeters, but this distortion is very repeatable. In addition to this distortion, there is also random noise, v, which can range from 0 to 0.3 millimeters. The calibration object will be moved to several different positions in the workspace and several measurements of the markers will be taken, resulting

in multiple "frames" of data consisting of $[\vec{D}_1, ..., \vec{D}_{N_D}, \vec{A}_1, ..., \vec{A}_{N_A}, \vec{C}_1, ..., \vec{C}_{N_C}]$. The several dimpled calibration posts are located at unknown yet fixed locations in F_{EM}. During pivot calibrations, these dimpled calibration posts will be denoted, p_dimple. On the optical pointer probe, there are N_H markers at unknown but fixed locations, $\vec{h}_i$. The electromagnetic pointer, similarly, has N_G electromagnetic markers at unknown but fixed locations, $\vec{g}_i$. Frames from pivot measurements taken from placing the pointers in a calibration post dimple and rotating the pointer about the tip while maintaining contact with the dimple are collected. Each frame for the optical pointer contains the following information: $[\vec{D}_1, ..., \vec{D}_{N_D}, \vec{H}_1, ..., \vec{H}_{N_H}]$. Each frame for the electromagnetic pointer contains the following information: $[\vec{G}_1, ..., \vec{G}_{N_G}]$.

1 Mathematical Approach

1.1 3D Point Cloud to 3D Point Cloud Registration

The high level overview of my 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. My 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. I 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:

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

1.2 Pivot Calibration

My pivot calibration implementation was very straightforward and was taken directly from the CIS I lecture slides [5].

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,

$$\begin{pmatrix} R_1 & -I \\ R_2 & -I \\ \dots & \dots \\ R_N & -I \end{pmatrix} \begin{pmatrix} \vec{p}_{tip} \\ \vec{p}_{dimple} \end{pmatrix} = \begin{pmatrix} -\vec{p}_1 \\ -\vec{p}_2 \\ \dots \\ -\vec{p}_N \end{pmatrix} \tag{8}$$

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, I 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 $\vec{p}_{dimple}$.

1.3 Calculation of the Expected EM Marker Positions on the Calibration Object

To calculate the expected EM marker positions on the calibration object, I first used my 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_{Opt_EM}, by passing forming a pair of point clouds d_i, D_i and passing this to my registration algorithm. Next, I again used my 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_{Opt_C}, by forming a pair of point clouds a_i, A_i and passing this to my registration algorithm. Finally, the expected EM marker locations in F_{EM}, C_i could be found using the following equation:

$$C_i^{(expected)} = F_{Opt\_EM}^{-1} F_{Opt\_C} c_i \tag{9}$$

1.4 EM Probe Pivot Calibration

For the EM probe pivot calibration, I was given a set of point clouds measured in F_{EM}. To perform pivot calibration to find $\vec{p}_{tip}$ and $\vec{p}_{dimple}$, one must first define an initial coordinate system and then the frame transformations between that initial coordinate system and the local coordinate system of pointer at each frame in the pivot calibration dataset. To do this, I set the initial coordinate system to that of the local coordinate system of the pointer in the first frame of the pivot calibration dataset. Then, I used my point cloud registration algorithm to find the corresponding frame transformation between the initial coordinate frame and all other local coordinate frames for the pointer in each frame of the data. Once each frame transformation was know, I could then set up the least squares problem defined in Equation 8 and solve for $\vec{p}_{dimple}$.

1.5 Optical Probe Pivot Calibration

For the optical probe pivot calibration, my mathematical approach was essentially the same as in Section 1.4, with one extra step. Since the optical markers were in the optical tracker coordinate system, I needed to first translate the locations of the optical markers to be measured from F_{EM}. To do this, at each frame I computed $F_{EM_{Opt}}$, the frame transformation from the F_{Opt} to F_{EM}. I then transformed the pointer point clouds from F_{Opt} to F_{EM} at each frame and used these transformed point clouds in the exact same fashion as outline in Section 1.4 to solve for $\vec{p}_{dimple}$.

2 Algorithmic Approach

Note, it is not necessary for me to go over every function that I wrote for this assignment, as several functions are trivial or not crucial for understanding the overall code implementation. For example, I wrote a helper function *write_3d_point_to_file* to trivialize logging a 3D point to an output file. I will not discuss these types of functions, but will discuss the core functions that implement the mathematical approaches seen in Section 1. If you are looking through my code and see a function that I did not discuss in the report, that is because I deemed it unimportant. Read the docstring of that function and I hope you will agree with me.

I implemented this code in Python 3.11. Below, I have broken up each major function into its own subsection to provide pseudocode and basic analysis.

2.1 Utility Functions

The below functions serve as the core utility of the project and are constantly re-used. I 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`

2.2 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 6 Compute C_i^{expected} for Calibration Dataset

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

Algorithm 7 Compute p_{dimple} in EM Tracker Coordinate Frame

Input: `empivot_file_path`, path to the empivot dataset
Output: 3D coordinates of p_{dimple} in the EM tracker coordinate frame
`empivot_cal_frames` $\leftarrow$ `parse_empivot(empivot_file_path)`
`pcd_frames` $\leftarrow$ list of `empivot_cal_frames["G"]` ▷ Extract G marker sets from each frame
$(p_{tip}, p_{dimple}) \leftarrow$ `pivot_calibration(pcd_frames)` ▷ Run pivot calibration to compute p_{tip} and p_{dimple}
return p_{dimple} ▷ Return the computed location of p_{dimple}

Algorithm 8 Compute p_{dimple} in Opt Tracker Coordinate Frame

Input: `calbody_file_path`, `optpivot_file_path`
Output: 3D coordinates of p_{dimple} in the Opt tracker coordinate frame
`calbody` $\leftarrow$ `parse_calbody(calbody_file_path)`
`optpivot_cal_frames` $\leftarrow$ `parse_optpivot(optpivot_file_path)`
`d_vals` $\leftarrow$ `calbody["d"]` ▷ Extract d marker coordinates from calbody
`H_in_em_pcd_frames` $\leftarrow$ empty list ▷ Initialize an empty list for transformed H coordinates
for `frame` in `optpivot_cal_frames` **do**
  `D_vals` $\leftarrow$ `frame["D"]` ▷ Extract D markers from the frame
  `H_vals` $\leftarrow$ `frame["H"]` ▷ Extract H markers from the frame
  $F_{dD} \leftarrow$ `pcd_to_pcd_reg_w_known_correspondence(D_vals, d_vals)` ▷ Compute the transformation F_{dD}
  `H_in_em_pcd_frames.append(`$F_{dD}.transform\_pts(H\_vals)$`)` ▷ Transform H markers to the EM coordinate frame
end for
$(p_{tip}, p_{dimple}) \leftarrow$ `pivot_calibration(H_in_em_pcd_frames)` ▷ Run pivot calibration to compute $\vec{p}_{dimple}$
return $\vec{p}_{dimple}$ ▷ Return the computed location of $\vec{p}_{dimple}$

Algorithm 9 Full Main Function

Input: `calbody_path, calreadings_path, empivot_path, optpivot_path, output_folder, dataset_prefix`
$C_i^{\text{expected\_frames}} \leftarrow$ `compute_C_i_expected(calbody_path, calreadings_path)`
$p_{dimple\_em} \leftarrow$ `compute_p_dimple_in_em_coord_frame(empivot_path)`
$p_{dimple\_opt} \leftarrow$ `compute_p_dimple_in_opt_coord_frame(calbody_path, optpivot_path)`
`output_file` $\leftarrow$ `os.path.join(output_folder, f"dataset_prefixoutput1.txt")`
$N_C, N_{frames} \leftarrow$ `get_N_c_and_N_frames_from_calreadings(calreadings_path)`
Open file `output_file` for writing:
  write "N_C, N_{frames}, $dataset\_prefix$`output1.txt`" to file
  `write_3d_point_to_file(`p_{dimple_em}`, file)`
  `write_3d_point_to_file(`p_{dimple_opt}`, file)`
for each `frame` in $C_i^{\text{expected\_frames}}$ **do**
  for each C_i^{expected} in `frame` **do**
    `write_3d_point_to_file(`C_i^{expected}`, file)`
  end for
end for
Close file

3 Code Architecture

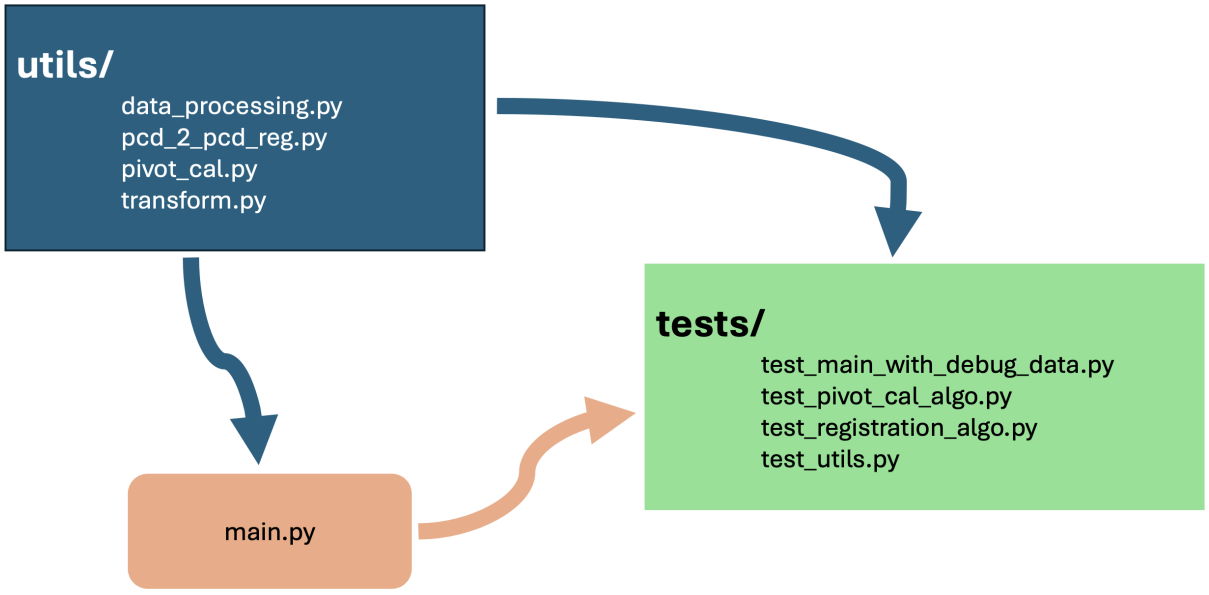

Figure 2: Code flow.

I tried to make my code as simple as possible. As such, a hierarchy table seemed unnecessary, and instead I provided a flow chart. This shows the flow of inheritance in my code. Simply put, the main.py file holds all of the main functions described in Section 2.2. The main.py file inherits from the functions found in the files in the utils directory. Finally, the files in the tests directory implement the testing and verification functions for the functions in the main.py file as well as the functions inside the files in the utils/ directory.

3.1 Description of Each Code File

File Path from PROGRAMS Dir	Description
main.py	Functions for completing PA1 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 performaning a pivot calibration.
utils/transform.py	Custom FT class used to perform frame transformtaion on 3D points.
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 my pivot calibration algorithm.

Table 1: Description of each code file.

3.2 Reusability and Modularity

I made my 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 I needed it, I wrote the function once, tested it, and re-used it. I did the same thing with my point cloud registration function, using it consistently in the testing functions and several functions in the main.py file. I also did this with the pivot calibration function and my function for getting the point cloud in the local frame. All of these functions are described in Section 2. It is possible to see them being used inside each other in a modular way in the pseudocode in section 2. Additionally, I wrote several small helper scripts that were too trivial/extraneous to include in Section 2 and re-used these constantly throughout my code.

4 Validation Approach

To ensure that my code worked consistently and my mathematical approach was well founded, I exhaustively tested my code via unit tests. To do this, I implemented the PyTest [3]. If you go into my directory, follow the install instructions, activate the environment, and run pytest, all tests described in this section will pass. This ensures that my implementation is correct and that even the low level utility functions are validated appropriately. Below, I describe each test in detail.

4.1 Point Cloud Registration Algorithm Testing

To test my 3D point cloud to 3D point cloud registration algorithm I 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. I used SciPy [4] 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 my 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).

4.2 Pivot Calibration Algorithm Testing

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

Algorithm 10 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`$(0, mse_{p_{dimple}})$

4.3 Full Testing with Debug Output Files

To test my full program pipeline, I used the full main() function to generate output files for each of the debug datasets. I 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. I then asserted that the difference was less than 0.1 for every single data point, thus ensuring my approach and implementation was correct and free from errors.

5 Results

To get results for this work, I calculated the Mean Squared Error (MSE) between each coordinate in the 3D points output in my predicted output file and in the debug datasets. Table 2 shows the results of this error calculation for each of the debug data sets. For the majority of the debug datasets, the mean squared error is quite small (¡ 1). For the final two debug datasets, datasets f and g, the MSE is slightly larger. This difference seems larger than than which would be incurred from differences in numerical approximations, but I could not find a good reason for why these are so high. I was able to confirm that my low level functions, the *Utility* functions were sound by validating them with incredibly low error thresholds in my tests, so I am confident in my low level functions. Therefore this error must be due to some discrepancy in the way I process or use the data. This error sits in an odd range, where it is large enough to rule out numerical approximation discrepancies but small enough to be 100% sure that my results are not due to luck. My process is obviously correct, but there is a small discrepancy arising from something that cannot be determined via my testing. Overall, my results show that my process is correct, and my various tests for each functions can validate that they are working to a high level of accuracy.

Dataset Prefix	Mean Square Error
pa1-debug-a	$7.033e^{-6}$
pa1-debug-b	0.076
pa1-debug-c	0.099
pa1-debug-d	$5.489e^{-5}$
pa1-debug-e	0.814
pa1-debug-f	1.697
pa1-debug-g	1.154

Table 2: Error analysis

Finally, the results of my approach can be shown in the OUTPUT/¡dataset_prefix¿output1.txt file for each dataset prefix (unknown and debug). This will serve as the tabular results for the unknown files. Since I have nothing to compare these files to, there is no direct numerical analysis on them, but the values seen reasonable according to a visual inspection, and I am confident that the error in these values is relatively low based on my results on the debug datasets and my testing and validation of lower level functions.

6 Statement of Who Did What

Grayson Byrd did everything in this work and I did not have a partner.

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

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

[5] Russel H. Taylor. 600.455/655 lecture notes, 1996-2024.