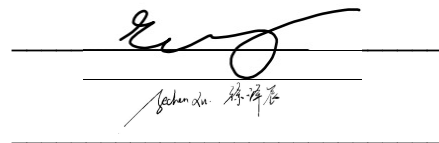


Programming Assignments 3 and 4 – 601.455/655 Fall 2023

Score Sheet (hand in with report) Also, PLEASE INDICATE WHETHER YOU ARE IN 601.455 or 601.655

(one in each section is OK)

Name 1	Esther Wang
Email	wwang177@jh.edu
Other contact information (optional)	
Name 2	Zechen Xu
Email	zxu130@jh.edu
Other contact information (optional)	
Signature (required)	<p>I (we) have followed the rules in completing this assignment</p> 

Grade Factor		
Program (40)		
Design and overall program structure	20	
Reusability and modularity	10	
Clarity of documentation and programming	10	
Results (20)		
Correctness and completeness	20	
Report (40)		
Description of formulation and algorithmic approach	15	
Overview of program	10	
Discussion of validation approach	5	
Discussion of results	10	
TOTAL	100	

Computer Integrated Surgery 1 - Programming Assignment 3

Zechen Xu (zxu130), Esther Wang (wwang177)

Nov 2023

***Note: key addition specific to programming assignment 3 has been written in navy blue*

Computer Integrated Surgery 1 - Programming Assignment 3	1
Introduction and Background	1
Mathematical Approach for Rigid Body Transformation and Pivot Calibration	2
Overview on Algorithmic Steps	5
Program Structure	8
Results Validation and Discussion	11
Appendix	13

Introduction and Background

Programming Assignment 1 focuses on building a foundational algorithm to perform 3D rigid body transformation, set registration, and pivot calibration in the context of surgical robot calibration. The program was implemented using Python with external libraries, namely Numpy and SciPy as supporting tools.

Building on top of functionalities that have been implemented in Programming Assignment 1 (“PA1”), Programming Assignment 2 (“PA2”) explores the distortion correction aspect of the same scenario. Furthermore, it is worth noting that we are primarily working with frame transformation and calibration relating to CT coordinate frames. From implementation and documentation perspective, we have also addressed feedback to improve the usability and readability of the program.

In Programming Assignment 3 (“PA3”), we explored how to make use of a 3D surface mesh and find the closest point (“matching” part prior to the full icp algorithm) on a rigid body (pointer tip in this case) in CT coordinates through an iterative-closest point registration algorithm. There are two rigid bodies in this problem - one is used as the reference tool and the other is denoted as the pointer tip.

Mathematical Approach for Rigid Body Transformation and Pivot Calibration

The transformation F in the following code is written as $[R, p]$ for concise representation and calculation.

1. 3D points Set Registration

The 3D rigid registration code achieves a 3D transformation matrix that maps a set of source points to a set of target points in three-dimensional space. We selected K. Arun's method to calculate directly. The following is how coding works mathematically.

The centroids of the source and target point sets are calculated to facilitate subsequent computation of rigid body transformation. The centroid is the average position of all the points in a shape, computed using the arithmetic mean of the positions of all the points along each dimension separately.

Then H is computed by the sum of the products of each corresponding point in the two frames, which is achieved by using the covariance of two points set. The covariance matrix captures how much the source and target points vary from the mean with respect to each other. In this context, it is used to determine how much one set of points must be rotated or scaled to match the other set.

Singular value decomposition(SVD) is a linear algebraic approach that decomposes a matrix into three sub matrices (U, S, V^T , the left-singular vectors, the singular values, and the right singular vectors), which was used to find the optimal rotation matrix R that will align the centered source points with the centered target points by decomposition of H .

R is computed from the matrices resulting from the SVD as $U \cdot V^T$. Then we derive the translational vector P as $P = \bar{b} - R \cdot \bar{a}$.

2. Pivot Calibration

Pivot calibration algorithm utilizes the least squares method to determine two points: P_{pivot} and P_{tip} , and get their precise positions. The objective function is defined to calculate the error between the transformed frames and the translation part of the transformation matrix. This

function depends on the parameters $error_j = [R_j I][\frac{p_{tip}}{p_{pivot}}] - R_j$, and the goal is to find the values of these parameters that minimize the error, and reach the precise position of P_{pivot} and P_{tip} .

3. 3D Distortion Calibration and Correction by Bernstein Polynomials Method

In fields like remote sensing and imaging, distortions often emerge due to discrepancies between sensor coordinate systems and real-world physical dimensions. Polynomial model fitting serves as a common approach to rectify these distortions. Among the various polynomial models (i.e. Berstein, Chebyshev), we chose Berstein polynomials considering that it is relatively more straightforward to implement and numerically stable. In our 3D distortion calibration case, we can directly construct an F matrix using the Berstein polynomial.

The first step of distortion calibration entails the scaling of input values to the range $[0,1]$, by picking the upper limits and the lower limits of the distorted data set which makes sure the Bernstein polynomial works well.

With values scaled appropriately, in our case, a 5th-degree Bernstein polynomial is constructed for every point. Choosing the 5th degree for the Bernstein polynomial not only prevents overfitting of the data by fitting it too closely, but it also helps to capture the most essential data to avoid high bias. The precise mathematical formulation is shown in slide 20 of the InterpolationReview.pdf lecture notes.

$$B_{N,k}(v) = \binom{N}{k} (1-v)^{N-k} v^k \quad 0 < v < 1$$

Following the polynomial construction, these polynomials are aggregated to formulate the F Matrix (size is $N_points * 216$) as the first matrix shown below. The ground truth matrix (size is N_points of points $* 3$) contains the points that are expected to have after distortion correction. Leveraging Singular Value Decomposition (SVD), a least squares problem is solved using the ground truth data. The methodology for this is elucidated in slide 48 of the lecture notes.

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

The result obtained from SVD is a calibration coefficient matrix (216x3). When multiple groups of distorted points are observed by the same sensor, the points should be normalized and input into the Bernstein polynomial. The resulting matrices are then stacked and multiplied by the coefficient matrix. This provides an overview of the distortion correction step.

4. Finding the Closest Point on a Triangle

To identify the nearest point on a bone surface to a given point, a triangle mesh fitting technique is utilized. This process involves determining whether the point lies inside or outside a specific triangle region. For this, we designate the point in the space as point 'a', with 'p', 'q', and 'r' being the vertices of the triangle. To ascertain the location, we solve two equations below to derive the proportional parameters λ (lambda) and μ (mu). If λ and μ satisfy certain constraints as $\lambda \geq 0$, $\mu \geq 0$, $\lambda + \mu \leq 1$, we conclude that the closest point, denoted as 'c', resides within the triangle. Conversely, if these constraints are not met, the point is deemed to be outside the triangle.

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

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

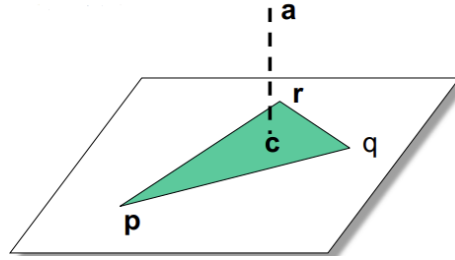


Figure. Finding the closest point on a triangle plane

If the point is out of the triangle, we suppose the closest point is on the edge of the triangle which is perpendicular to that point. The diagram illustrates how, by applying various constraints, we can determine the nearest point outside a triangle within three distinct zones on the triangle's plane.

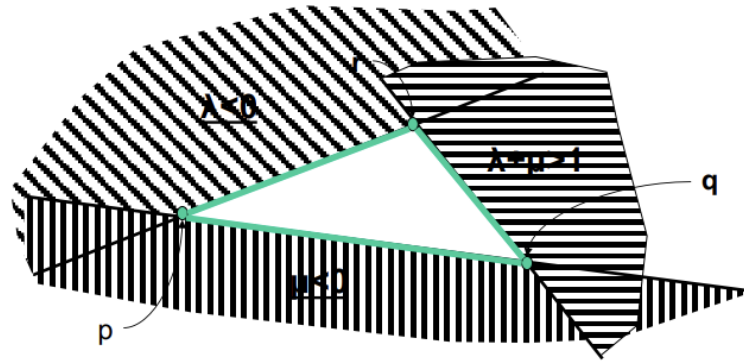
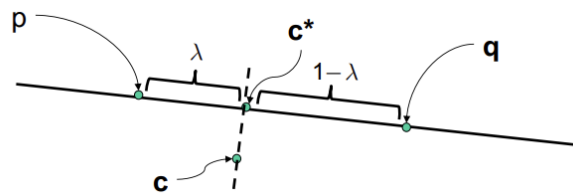


Figure. Finding the closest point on the triangular edge

We can calculate the nearest point on this edge using the specified equation where 'p' and 'q' are two vertices points and c is the point to be projected to the line segment. At this time λ is the ratio of normalized length from c_{seg} to 'p'.

$$\lambda = \frac{(c-p)(q-p)}{(q-p)(q-p)}, \lambda_{seg} = \max(0, \min(\lambda, 1))$$

$$c_{seg} = p + \lambda_{seg}(q - p)$$



By iterating this process for all the triangles on the mesh surface, we can accurately identify the nearest point on a specific triangle by comparing the distances each time.

Overview on Algorithmic Steps

3D Transformation:

3D Transformation was implemented in *calculate_3d_transformation(source_points, source_points)*. The function takes two inputs source and target, each of which is a 3D point set (numpy array with size Nx3), and returns a 4x4 transformation matrix. Given the nature of 3D rigid body transformation, we need to first calculate the centroids of source and target points, and then compute the translated sets of centered_source and centered_target by subtracting respective centroids from input points. Subsequently, we were able to calculate the 3x3 rotation matrix using the single value decomposition approach and check for reflection. Translation vector was calculated after applying a rotation matrix to the source and subtracting it from the target. Finally, the R and t were assembled into a 4x4 transformation matrix. In order to use the resulting transformation matrix, a function *apply_transformation(points, transformation)* was implemented.

Pivot Calibration:

Pivot calibration was implemented in *pivot_calibration(transformation_matrices)*. The function calibrates a pivot point, in this case the tip of the probe, based on a set of transformation matrices obtained from other points on the probe. To start with, transformation_matrices were converted into a NumPy array and we initialized parameters (p_tip and p_pivot) with an initial guess. Least squares optimization from the SciPy package were referenced. The least squares optimization method was used to find the optimal parameters by minimizing the error using a heuristic, which was implemented in the helper function *optimization_heuristics(parameter, transformation_matrices)*, which computes the error associated with a given set of parameters and a collection of transformation matrices. It iterates through the frames, applies the transformation using the provided parameters, and calculates the error as the difference between the transformed points and the corresponding points in the transformation matrices. The result provides the calibrated p_tip and p_pivot, which are returned as the output, which calibrates the pivot_tip to ensure accurate alignment.

Distortion Correction:

Distortion correction through Bernstein polynomials was implemented in a series of functions that work in tandem to scale data to fit within specified bounds, calculate Bernstein polynomials, building F matrix, fitting calibration coefficients and correct distortion by applying the coefficients. Class *DewarpingCalibrationCorrected* was initialized with four class variables degree, coefficients, q_min and q_max to facilitate subsequent calculation and allow us to change the degree of Bernstein polynomials if needed.

Static method *scale_to_box(data, q_min, q_max)* scales input data within the bounds defined by q_min and q_max, which are used for normalization. Static method *bernstein(degree, parameter, input)* calculates Bernstein polynomials, and the *build_f_matrix* method constructs an F matrix

based on input values using Bernstein polynomials. This F matrix is a key component of the distortion correction model.

The pivotal *fit(self, distorted_data, ground_truth)* method takes the distorted calibration data and computes the coefficients of the distortion correction model using a least squares optimization approach, which are essential for the correction. Detailed steps as follows:

- Determines the q_{\min} and q_{\max} values for each dimension based on the minimum and maximum values within *distorted_data*, which is used for scaling and normalization
- Normalizes *distorted_data* using the scaling factors q_{\min} and q_{\max} to ensure data is within the defined bounds.
- Then *build_f_matrix* method constructs an F matrix based on the normalized data
- Finally applies a least squares optimization technique to find the coefficients that best fit the distorted data to the ground truth data based on the constructed F matrix. These coefficients are the model's parameters and determine the distortion correction.
- Output coefficients are stored in the *self.coefficients* attribute for later use in the correction method.

The *correction(self, data)* method applies the distortion correction to input data. It checks if the model has been fitted and then normalizes the input data, constructs the F matrix, and applies the correction based on the coefficients obtained during calibration.

Instead of following the iterative approach discussed in class, we applied the calibration coefficient matrix by F matrix of the distorted dataset to compute the corrected dataset. Such an approach is more computationally efficient. To derive the corrected and undistorted point set, the calibration coefficient matrix is multiplied by the F Matrix of the distorted point set. This matrix multiplication, a computationally efficient approach, serves as an alternative to the iterative sum loop mentioned in the lecture slides.

Finding the closest point:

The core functionality of identifying closest points was implemented in two functions: *find_closest_point(point, vertices, triangles)* and *project_on_segment(c, p, q)*. The former matches the closest mesh vertex to the given point through iterating through the 3D mesh triangles defined by the coordinates of its three vertices, then constructing a matrix representing the two vectors formed by subtracting two vertices from the last one. To determine point coordinates within the specific triangular mesh $c = p + \lambda(q - p) + \mu(r - p)$, λ and μ are solved using least squares optimization, and then project onto the respective line segment defined by two vertices depending on the cases using *project_on_segment*.

Finally, one additional function *calc_difference(c_k_points, d_k_points)* is implemented to evaluate the magnitude difference (or “error”), defined as the Euclidean distance between the closest vertex and the pointer tip point.

Data Parsing:

Numpy library was used to parse the input raw txt files depending on the type of data (i.e. coordinates of the markers, number of frames etc) into a list of arrays for further processing. The index of the list indicates the specific frame of interest. Furthermore, for PA3, additional functions such as parseMesh were added to process the .sur file so that both coordinates of all vertices and the corresponding triangles are handled properly and more robustly. Copy, OS and RE library were also used in the driver code to facilitate reading input and writing output.

Debug Test:

For PA3, we implemented unit tests in the debug_test.py script for icp_library which entails the core functionality to find the closest point on a 3D surface mesh. We validate the functionality of 'find_closest_point()' and 'project_on_segment()' The unit tests for the former checks for a regular case to verify computation accuracy, and examines four scenarios: 1) point in triangle but not in plane; 2) point not in triangle and not in plane; 3) point in triangle, not in plane; and 4) point not in triangle but in plane. The unit tests for the latter check two cases: 1) c is on the side of the segment in the plane 2) c is on the segment in the plane. Meanwhile, test_parseMesh unittest was added to ensure the data parsing for PA3 related files are processed correctly for computation.

We have implemented unit tests in the debug_test.py script for distortion correction which is the core functionality in PA2. The checking process is broken down into 'test_calibration_and_correction' 'test_fit' and 'test_correction'. The first unit test checks if the corrected distortion sample is within our target tolerance, and the other two tests check if the valid input and output have been generated from the relevant functions that are supposed to work in concert. For PA1-related contents, we mainly check our 3D points registration function by the following debug methods as 'test_calculate_3d_transformation', 'test_compute_error', and 'test_apply_transformation' which validate the correctness and robustness of the 3D transformation processes. The first test checks the size of the transformation matrix returns from points registration is (4,4) as what we expect; the second test checks the error between the target points and source points after applying the transformation matrix.

Error Analysis:

We have further enhanced our algorithm with robust error-checking protocols to validate its accuracy. Specifically, we employ a function 'compute_error(computed_data, target_data)' that utilizes the Euclidean distance to ascertain discrepancies between computed and target data points across frames. The Euclidean method is our chosen metric for error computation because it is grounded in the Pythagorean theorem, a cornerstone of Euclidean geometry that defines the relationship between the sides of a right-angled triangle. This theorem is directly applicable to calculating distances within 3D Cartesian coordinates, ensuring that our error measurement is both geometrically sound and universally applicable. Geometrically, the error is represented by

the straight-line distance between the computed point and the target point, providing an intuitive and precise measure of our algorithm's performance. We can represent our algorithm as the following formula:

$$d(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2 + (p_z - q_z)^2}$$

Where, p_x, p_y, p_z are the x, y, z coordinates of point p, respectively. q_x, q_y, q_z are the x, y, z coordinates of point q, respectively. $d(p, q)$ is the Euclidean distance (error) between points p and q.

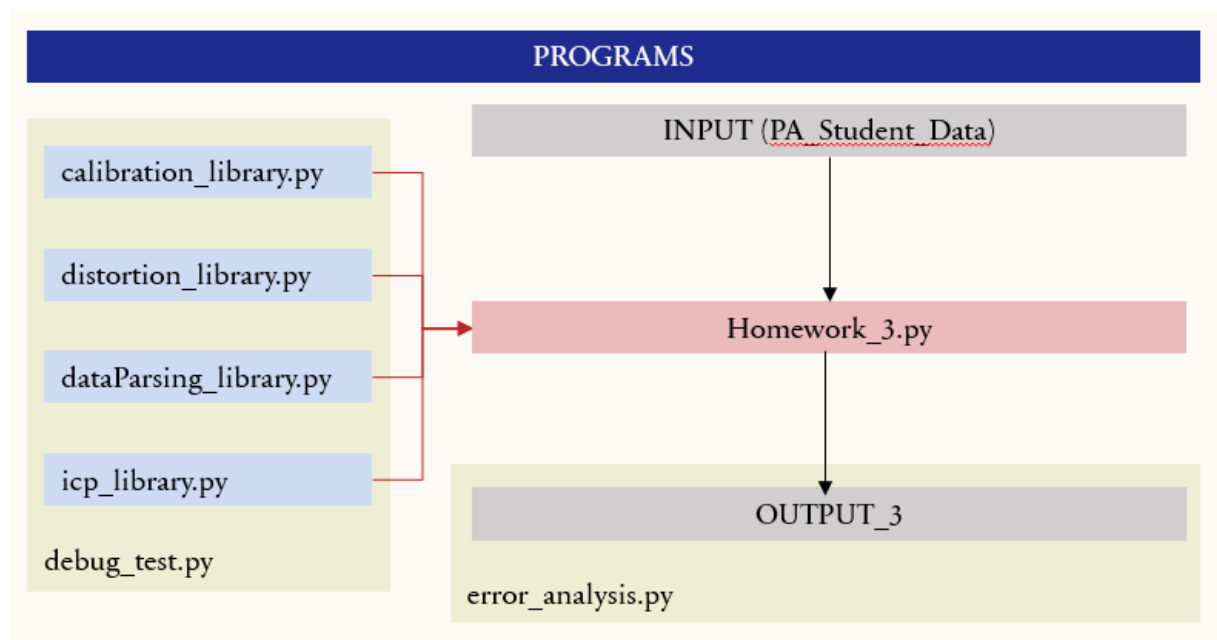
Program Structure

Overview:

The core functionality was implemented in five scripts - `distortion_library.py`, `calibration_library.py`, `dataParsing_library.py`, `icp_library.py`, `homework_3.py` - that work in tandem. `validation_test.py` serves as a checkpoint script that contains relevant unit tests to ensure proper implementation for the core functions in the other scripts. Finally, `error_analysis.py` contains relevant error analysis functions that allow us to evaluate the accuracy of our calibration.

The arrows below in the diagram indicate the program flow - please note that the red arrows mean `Homework_3.py` makes use of `calibration_library.py`, `distortion_library.py`, `icp_library.py`, and `dataParsing_library.py`. `Debug_test.py` covers all four libraries in the development process, and `error_analysis` is used to examine the output generated by `Homework_3.py`.

Program Structure Overview



homework_3.py:

The script is the driver code which imports relevant functions and classes from calibration_library.py, icp_library.py and dataParsing_library.py. The script will read the directory location and update the relative file path by itself. To run the executable, the user shall enter **python homework_3.py {choose_set} {input_type}** in the terminal. {input_type} specifies whether the input data is “Debug” or “Unknown”, while {choose_set} refers to the index of the data set i.e. “A.” For instance, a valid terminal command line should be formatted as **python .\homework_3.py K Unknown** The result will be outputted and saved under the same folder titled “PA3-{choose_set}-{input_type}-Output.txt”. Please kindly refer to the comments for calculation performed for each question.

icp_library.py:

The script serves as the library for core functionality for finding the closest point on a 3D mesh surface given a point. The list of functions and brief description is as follows:

- find_closest_point(point, vertices, triangles)
 - Finds the closest point on the mesh surface defined by vertices and triangles to the given point through projection to the respective line segment that forms the specific triangular mesh
- project_on_segment(c, p, q)
 - Computes the actual projection point from c onto the line segment defined by endpoints p and q, and returns the projected point
- calc_difference(c_k_points, d_k_points)
 - Calculates the Euclidean distance between corresponding points in two point clouds and returns a 1D array with the distance

distortion_library.py:

The script serves as the library for core functionality for distortion correction. The list of functions and brief description is as follows:

- scale_to_box(data, q_min, q_max)
 - Scales input data with upper and lower bounds to shape the data into [0,1] and return the scaled data
- bernstein(N, k, u)
 - Constructs Bernstein polynomial based on input degree and parameter of the polynomial, and the input value of the polynomial, then returns the polynomial
- build_f_matrix(self, u)
 - Constructs F matrix based on input values for the Bernstein polynomial
- fit(self, distorted_data, ground_truth)
 - Takes the distorted calibration data and computes the coefficients of the distortion correction model using a least squares optimization

- `correction(self, data)`
 - Applies the correct matrix to the distorted input data

calibration_library.py:

The script serves as the library for core functionality in the 3D transformation, set registration and pivot calibration. The list of functions and brief description is as follows:

- `calculate_3d_transformation(source_points, source_points)`
 - Calculates and returns the 4x4 transformation matrix based on two sets of Nx3 Numpy array source and target, both of which are sets of 3D points
- `apply_transformation(points, transformation)`
 - Returns the set of transformed 3D points using the input source 3D points and the 4x4 transformation matrix provided. Also ensure properly scale (normalization)
- `pivot_calibration(transformation_matrices)`
 - Calculates and returns the 3D location of the pivot based on the set of input 4x4 transformation matrices.

dataParsing_library.py:

The script serves as the library to handle data parsing for raw optical marker and EM marker locations to calculate pivot G, pivot H and expected C_i . The list of functions and brief description is as follows:

- `parseData(input_file)`
 - Read in raw data and stores the 3D points as Numpy array in a list called `point_cloud`, which is returned
- `parseMesh(input_file, vertices_num)`
 - Read in raw data of list of 3D vertices based on `vertices_num`, and subsequently the index of the list of triangles' vertices; then returns the point cloud of vertices and triangle indices
- `parseCalbody(point_cloud)`
 - Read in the point cloud for calbody dataset specifically and parse the data into three sets `d`, `a`, `c` which stores the original location of 8 optical markers on EM base, 8 optical markers on calibration object and 27 EM markers on calibration object
- `parseOptpivot(point_cloud, len_chunk_d, len_chunk_h)`
 - Read in the point cloud for optpivot dataset specifically, number of optical markers on EM base and number of optical markers on probe. The function then parses the point cloud into two sets, of which one contains optical markers on EM base only and the other contains optical markers on probe only.
- `parseFrame(point_cloud, frame_chunk):`
 - Read in the point cloud and number of rows in each frame, and return a list of arrays whose index indicates the specific frame.

Results Validation and Discussion

Results Validation Approach Taken

- During the development process, the intermediary results and variables were also closely examined and cross-checked with debug samples provided
- The result outputs from debug samples were cross-checked against our output both manually and through our program to verify our implementation. The functions related to calculating the difference between datasets are implemented in `error_analysis.py`
- A series of unit tests have been implemented in `debug_test.py` file to ensure core functionality was properly implemented in `distortion_library.py`, `calibration_library.py`, `dataParsing_library.py` and `icp_library.py`
- Please refer to below overview of the `debug_test.py` and `error_analysis.py`

debug_test.py:

The script serves as the unit test library to verify our intermediary steps during the development process. We mainly focus on ensuring valid input and output format, and verify whether the functionalities we implemented match with the expected output. To run the executable, the user shall enter `python validation_test.py` in the terminal. In our unit testing, we establish an error threshold of $1e-3$ to validate the accuracy of the results. This standard is crucial as it ensures our icp procedure, distortion and pivot calibrations run well without any problems. Consequently, maintaining minimal error in our sample outputs in the unit test is imperative to guarantee the overall functionality and reliability of the system. The list of functions and brief description is as follows:

- `test_find_closest_point()`
 - Check if the output closest point identified on a particular triangular mesh is as expected in various scenarios
- `test_project_on_segment()`
 - Check if the projection from a 3D point to line segment defined matches with expected projected point
- `test_calculate_3d_transformation()`
 - Check if the transformation matrix can be calculated correctly based on sets of input
- `test_apply_transformation()`
 - Check if transformation matrix can be applied properly to input
- `test_parseMesh()`
 - Verify if input has been read in properly and output contains correct data format for both vertices and triangle indices
- `test_parseFrame()`
 - Check if given a set of data `parseFrame` function will parse aggregated data into different frames

error_analysis.py:

To run the executable, the user shall enter `python error_analysis.py {choose_set} {input_type}` in the terminal.

- `compute_error()`
 - Computes the Euclidean distance (error) between computed data points and target data points.

Debugging Example

- Input validation: since we were working with a fairly large number of raw data (> 1k data points) and intermediary transformation matrix calculation steps, we encountered bugs that led to incorrect results in one axis due to wrong application of the transformation matrix. This has also shed light on how we can improve for future assignments in terms of overall error detection and algorithmic design approach to eliminate possibility of such errors.
- Intermediate result: during the development process, we either print or build in checks to ensure we are working with inputs with correct dimension i.e. 1D versus 2D array when applying multiple transformation matrices, and we are working with the correct sequence in matrix multiplication. There are occasions when we print out the dimension of the input or the intermediary output, it is clear that we are not working with the expected input.
- Matrix arithmetic: specific to PA3, we are working with coordinates of vertices, index of triangles and also applying transformation to them, we transpose the input array in certain cases i.e. calculating closest point to turn it into a 3xN format. However, it turns out *not* to be the optimal solution, given we would have to handle the transformation between 1D and 2D arrays in python, as well as some tricky `append()` / `matrix stack()` / `concatenate` methods. In the future, we may consider an alternative way to clean up the data structure instead of transposing the raw data to facilitate computation.

Discussion:

- We evaluate the error across all sample data sets with each point and have output the error for each point and each set as below. We have mentioned in our error analysis that we defined error as the Euclidean distance between our output points and the sample points.
- In PA3, we have also explored the difference between the actual expected pointer tip position and the vertex (closest point) of the mesh that we identified. The output analysis is shown in table 2 below.
- Overall, when comparing against sample output, the error appears to be smaller than 0.030 for all other sets, which suggests a relatively high level of accuracy with our algorithm when finding closest points. However, the magnitude difference when comparing the expected pointer tip location against the closest vertex based on triangle

mesh appears to be more significant. This may have to do with the pointer tip location relative to the mesh (i.e. distance from the closest line segment, projection, and the size of each individual mesh etc).

Tabular Summary of PA 3 Debug Data Results Versus Sample Output Error Analysis

Data Set / Point Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	0.017	0.000	0.010	0.000	0.014	0.020	0.014	0.014	0.000	0.020	0.014	0.000	0.014	0.017	0.014
B	0.000	0.010	0.017	0.010	0.000	0.010	0.014	0.014	0.000	0.010	0.010	0.010	0.010	0.010	0.010
C	0.000	0.014	0.010	0.000	0.010	0.000	0.000	0.014	0.014	0.000	0.010	0.010	0.017	0.010	0.014
D	0.010	0.010	0.017	0.014	0.014	0.010	0.024	0.017	0.010	0.010	0.010	0.010	0.017	0.014	0.020
E	0.014	0.017	0.014	0.017	0.010	0.010	0.000	0.010	0.014	0.010	0.014	0.028	0.014	0.010	0.014
F	0.017	0.014	0.014	0.000	0.000	0.014	0.014	0.020	0.014	0.014	0.000	0.010	0.014	0.010	0.010

Results Obtained for the Unknown Data:

Please kindly refer to the OUTPUT folder for the output files for the 6 sets of debug data and 4 sets of unknown data provided, and please find below list of output:

Tabular Summary of PA 3 Debug and Unknown Coordinates Magnitude Difference Results

Point Index / Data Set	A	B	C	D	E	F	G	H	J	K
1	0.001	1.402	0.381	1.929	2.631	0.002	0.323	4.729	0.153	1.716
2	0.001	1.585	1.713	2.708	2.325	0.278	2.072	1.435	4.166	2.280
3	0.002	0.064	0.553	0.145	3.443	1.294	0.395	0.250	0.671	1.075
4	0.001	3.109	0.688	1.205	3.738	0.748	0.665	1.379	2.380	3.172
5	0.001	2.542	0.828	2.26	0.153	1.348	1.127	0.653	3.075	3.276
6	0.000	0.238	1.013	0.061	3.983	0.640	1.614	0.031	1.026	1.813
7	0.007	1.435	2.212	0.968	1.111	1.120	0.238	0.127	2.035	2.248
8	0.001	0.735	0.053	0.402	0.776	2.621	0.619	3.447	3.906	0.396
9	0.001	2.454	0.76	0.761	3.323	2.944	3.103	0.348	0.046	1.694
10	0.001	2.325	0.021	0.345	3.713	0.232	1.893	2.499	3.029	2.247
11	0.000	3.064	1.016	2.145	1.584	1.158	0.944	0.065	2.688	1.314
12	0.002	0.193	1.542	1.469	2.947	1.723	3.198	1.765	2.029	1.018
13	0.002	3.505	0.086	0.273	3.884	1.824	2.172	0.420	4.716	3.507
14	0.004	1.699	0.577	1.64	3.717	2.268	0.183	1.335	1.107	0.257
15	0.000	0.032	1.734	0.759	2.906	2.745	0.845	0.301	1.162	0.960
16							0.759	0.196	2.745	2.390
17							2.662	1.347	0.581	0.154
18							2.602	0.149	0.790	1.982
19							0.057	0.580	2.492	0.265
20							0.257	0.802	2.659	1.287

Appendix

Responsibility Breakdown:

Both of us were heavily involved in the program structure and algorithmic design process, as well as the debugging process. Zechen was focusing more on the mathematical approach and Esther focused more on the implementation and drafting of relevant documents.

Reference:

[NumPy](#)

[SciPy](#)

[Argparse](#)

[OS](#)

[RE](#)

[Copy](#)