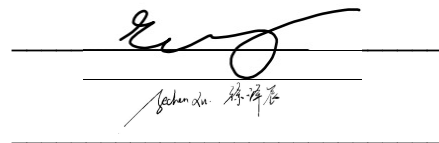


Programming Assignments 3 and 4 – 601.455/655 Fall 2023

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 4

Zechen Xu (zxu130), Esther Wang (wwang177)

Dec 2023

***Note: key addition specific to programming assignment 4 has been written in navy blue*

Computer Integrated Surgery 1 - Programming Assignment 4	1
Introduction and Background	1
Mathematical Approach for Rigid Body Transformation and Pivot Calibration	2
Overview on Algorithmic Steps	6
Program Structure	12
Results Validation and Discussion	15
Appendix	19

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.

Finally, in Programming Assignment (“PA4”), we tied together all concepts and algorithms implemented previously, and implemented some additional features (i.e. KDTree) to improve the time and space complexity of the searching algorithm to achieve searching the closest point on triangular mesh what we used to do in linear search. Moreover, we implemented the full

iterative-closest point algorithm and added max iteration /error tolerance parameters to minimize potential transformation error.

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.

Bernstein, Chebyshev), we chose Bernstein 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 Bernstein 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.

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

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

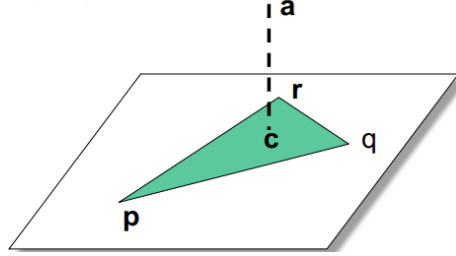


Figure. Finding the closest point on a triangle plane

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.

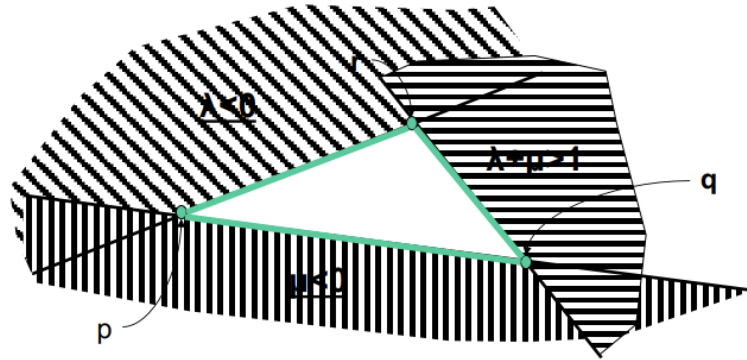
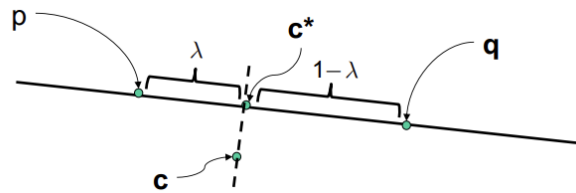


Figure. Finding the closest point on the triangular edge

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.

5. Efficient Closest Point Search on 3D Meshes via Bounding Boxes and k-d Tree Algorithms

The mathematical approach to finding the closest point on a 3D mesh using bounding boxes and K-D trees is a method that combines spatial subdivision and efficient search algorithms instead of the linear searching method we have mentioned. Below is the conclusion to the mathematical concepts behind this approach:

A bounding box is a volumetric enclosure that defines the extent of an object in 3D space which is defined by two opposite corners, representing the box's minimum and maximum extents along the coordinate axes. For a triangle with vertices p , q , and r , the axis-aligned bounding box is given by:

$$\begin{aligned} \text{Bounding Box} &= \{U, L\} \\ \text{Upper Corner (U)} &: \max(p, q, r) \\ \text{Lower Corner (L)} &: \min(p, q, r) \end{aligned}$$

k-d Tree Construction

The k-d tree is a pivotal structure in our methodology and it's one of the most popular approaches in finding closest neighbors. It is a binary tree where each node represents a k-dimensional point, specifically the center of a bounding box in our application. K stands for the number of dimensions of data to be partitioned. Once the base case is hit, the tree would verify the Euclidean distance to the point of interest to ensure it returns the closest neighbor in case of any ambiguity across the axis. In our case for instance, $k=3$, which means the layers with partitions along the X, Y, Z axis would alternate every three levels. Please note that considering accuracy in our implementation, the final submission only contains full implementation of linear and KD tree without using bounding box. Please refer to discussion section for more details. The construction of a k-d tree follows a recursive subdivision of space:

- The set of points (bounding box centers) is split into two equal parts along one of the k dimensions, based on the median value of the chosen dimension.
- This process recursively continues, with subsequent splits alternating between the remaining dimensions.
- Each non-leaf node in the tree represents a hyperplane that divides space relative to the median point along the chosen dimension.

Search Algorithm

Given a query point a search for its nearest point on the 3D mesh proceeds as follows: Initialize a variable bound with infinity, representing the closest distance found.

- Traverse the k-d tree to identify potential nodes of the tree (the data feed to construct the tree, can be a self-defined class / bounding box / point) within the bound distance of a .
- For each candidate node, calculate the closest point h on the encompassed triangle.
- If $||h - a|| < \text{bound}$, update bound with this new distance and record h as the closest point.

6. Iterative Approach to 3D Mesh Registration by the Closest Point Algorithm

For every contact point on the bone, presented by the probe's location, a computational process is developed to find the nearest triangular facet on the mesh. This process involves a k-d tree to accelerate the search speed, significantly reducing computational overhead compared to exhaustive search methods what we have introduced. Upon identifying the closest mesh point for each triangle, the error metric, calculated as the norm of the positional difference, is used to determine the most proximal point on the mesh surface.

Subsequently, the Point Cloud Registration method comes into play, generating a new transformation matrix that aligns the physical probe points to the digital mesh points. The iterative nature of the ICP algorithm ensures that each subsequent transformation is refined, progressively reducing the error and enhancing the congruence between the bone and mesh points.

The iterative process involves recalculating the transformed probe points and re-evaluating the nearest mesh points to these new coordinates. The refinement loop continues until the transformation converges, as evidenced by the error norm falling below a predefined threshold, or until a maximum number of iterations is reached. The convergence criteria incorporate both the magnitude of the transformation matrix adjustments and the history of error metrics.

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 below functions:

find_closest_point(vertices, triangles, startPoint, searchMode, maxIterations), hasConverged(tolerance, oldFrame, newFrame, errorHistory), find_closest_vertex_kd(point, kdtree, vertices, triangles), linear_search_closest_point(point, vertices, triangles) and project_on_segment(c, p, q). In PA4, we implemented two search methods: linear search which is driven by *linear_search_closest_point()*, and KD Tree search which is driven by *find_closest_vertex_kd(point, kdtree, vertices, triangles)*. In the meantime, we've also implemented the full ICP algorithm with tolerance and max-iteration features built in *find_closest_point()*.

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.

Please find below pseudocode and bullet clarification on core functions mentioned above:

function findClosestPoints(vertices, triangles, startPoint, searchMode, maxIterations):

 initialize registrationFrame as identity matrix 'eye(4)'

 initialize iteration counter and previousError deque

 create k-d tree from vertices with SciPy library

 while iteration < maxIterations:

 transform startPoint using registrationFrame

 transformedPoints = applyTransformation(startPoint, registrationFrame)

 find closest points on mesh for each transformed point

 - if searchMode is 'kd', use k-d tree

 - else use linear search

 calculate new transformation frame

check for convergence(*) -> hasConverged()
if converged, return the closest points and registration frame

update registration frame and iteration counter

return final closest points and registration frame

end function

- This function determines if the frame transformation is within a specified tolerance and serves as a helper function to **findClosestPoints()**

***function hasConverged**(tolerance, oldFrame, newFrame, errorHistory):
calculate error between oldFrame and newFrame

if error < tolerance or change in error is small, return True
else update errorHistory and return False

end function

- This function finds the closest points on a 3D mesh to a set of starting points and computes the registration frame using an iterative closest point algorithm with maxIterations set at 20 in our implementation when called in homework_4 driver code

function find_closest_vertex_kd(point, kdtree, vertices, triangles):

Find nearest vertex index from the k-d tree
nearest_vertex_index = kdtree.query(point, k=1)

Initialize the closest point and minimum distance
set closest_point = None; min_distance = infinity

Iterate over triangles containing the nearest vertex
for triangle_index in triangles containing nearest_vertex_index:
Calculate the closest point on the triangle to 'point'

current_closest_point = calculate_closest_point_on_triangle(point, triangle_index, vertices, triangles)

Update closest_point if a closer point is found

if distance(point, current_closest_point) < min_distance:

```
closest_point = current_closest_point
min_distance = distance(point, current_closest_point)
```

```
return closest_point
```

end function

- This function finds the closest point on a mesh to a given point using a k-d tree for initial vertex proximity; since a vertex may be shared by multiple triangular mesh, the functions checks all potential mesh and return the closest point

Function linear_search_closest_point(point, vertices, triangles)

- This function performs a linear search on the mesh triangles to find the closest point on the surface to a given point in 3D space. It utilizes linear algebra to determine coefficients for triangle interpolation and projections for point-to-triangle edge cases.
- Matches the closest mesh vertex to the given point through iterating through the 3D mesh triangles defined by the coordinates of its three vertices. For Each Triangle:
 - Compute vectors S based on the vertices of the current triangle.
 - Solve a linear system to find coefficients l and m.
 - Calculate the midpoint mid of the current triangle.
- To determine Closest Point, check if mid is inside the current triangle.
 - If inside, set c_star to mid; otherwise, project onto the nearest triangle edge.
 - Update Closest Point and store the computed closest point c_star in the array.
- To find the overall closest point, iterate over computed closest points to find the one with the minimum distance and return the overall closest point found on the mesh surface.

Function project_on_segment(c, p, q)

- This function projects point c onto the line segment defined by endpoints p and q, and ensure it's within bounds
- To determine point coordinates within the specific triangular mesh $c = p + (q - p) + (r - p)$ and 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
- The scalar projection (lambda) is the scalar projection of vector $c - p$ onto the direction vector $q - p$ is calculated using the dot product, which represents the distance of the projected point along the line

$$\lambda = \frac{(c-p) \cdot (q-p)}{(q-p) \cdot (q-p)}$$

- The calculated lambda is the clamped to ensure that the projected point lies within the sement, which was implemented using 'max' and 'min' functions

- Finally, the actual projected point on the line segment is computed as
 $p + \lambda(q - p)$ and $c = p + \lambda(q - p)$

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 PA4, 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 PA4, 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 'linear_search_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.

In addition, we have also tested features that are meant to improve the search efficiency and basic helper functions: namely, 'transform_tip_positions()', 'findClosestPoints()', 'hasConverged()' and 'find_closest_point_kd_on_triangle()'. For the first two, we mainly test whether the function has generated expected output. For 'hasConverged()', we tested three scenarios - converging, not converging, small difference within converging bounds. For 'find_closest_point_kd_on_triangle()', we tested two cases where the point is on or outside the specific triangular mesh when identifying the closest point.

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:

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

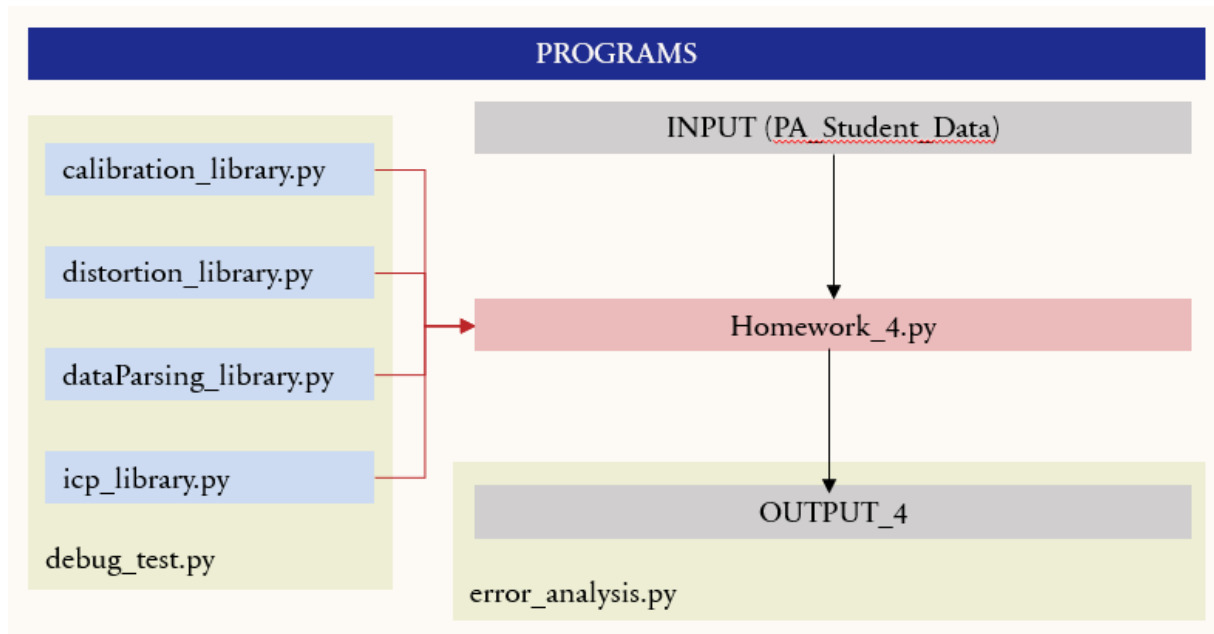
Where, p_x, p_y, p_z are the x, y, z coordinates of point p, respectively. q_x, q_y, q_z are the x, y, z coordinates of point q, respectively. $d(p, q)$ is the Euclidean distance (error) between points p and q.

Program Structure**Overview:**

The core functionality was implemented in five scripts - distortion_library.py, calibration_library.py, dataParsing_library.py, icp_library.py, homework_4.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_4.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_4.py.

Program Sturcture Overview



homework_4.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_4.py {choose_set} {input_type} {search_method}`** 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.” The third parameter `{search_method}` refers to the search method of finding closest point, namely ‘kd’ or ‘linear.’ For instance, a valid terminal command line should be formatted as **`python .\homework_4.py K Unknown kd`** The result will be outputted and saved under the same folder titled “PA4-`{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:

- `linear_search_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
- `transform_tip_positions(tip_positions, frame_transformation)`
 - Calculates transformed tip positions with the given frame transformation
- `findClosestPoints(vertices, triangles, startPoint, searchMode)`
 - Finds the registration transformation between a rigid reference body B and the bone using an iterative closest point algorithm
- `hasConverged(tolerance, oldFrame, newFrame, errorHistory)`
 - Determines if the frame transformation is within a specified tolerance
- `find_closest_point_kd(point, r, p, q)`
 - Find the closest point on the mesh surface defined by the three vertices r/p/q that form a specific triangular mesh
- `find_closest_point_vertex_kd(point, kdtree, vertices, triangles)`
 - Find the closest vertex to the given point and specific triangular mesh that contains this vertex, then returns the closest point

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 $1e-3$ or $1e-4$ 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_linear_search_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_transform_tip_position()
 - Check if the transformed coordinates for a given point and transformation matrix matches with expected coordinates
- test_findClosestPoint()
 - Check if the closest point on mesh is accurately identified given point of interest, list of triangles and vertices
- test_hasConverged()
 - Verifies if convergence is properly assessed given expected tolerance
- test_find_closest_point_kd_on_triangle()
 - Check if the projection to the mesh to find closest point given a point of interest and the three vertices of a specific triangular mesh matches with the expected result, if the the point is on the mesh
- test_find_closest_point_kd_outside_triangle()

- Check if the projection to the mesh to find closest point given a point of interest and the three vertices of a specific triangular mesh matches with the expected result, if the the point is outside the mesh
- `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 mutiple 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 and 4, we are working with coordinates of vertices, index of triangles and also applying transformaton 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.
- Tree search algorithm: specific to PA4, we have explored various ways to build a search tree and which tree to use (KD tree or octree). For the KD tree, we were initially going to use it in conjunction with a bounded box, whose centroid is defined as the centroid of each triangular mesh's max and min corners. However, output from such a design is prone to error due to distance heuristic (distance magnitude over 5 against sample) when constructing the tree and the error is subsequently amplified during projection to mesh

stage. Hence we decide to take a step back and simplify the algorithm: constructing the tree using vertex, and then perform subsequent calculations (i.e. calculating distance heuristics and identifying closest triangular mesh with this vertex). Due to search tree's recursive nature and the plethora of ways to implement it in the actual search scenario, it's better to start with a very small subtree and understand each step (i.e. base case and backtrack conditions) before implementing it. Moreover, it is also easier to build on top as opposed to being overly ambitious in the first place.

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 PA4, we 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 for PA4, when comparing against sample output, the error appears to be smaller than 0.25 for all other sets, which suggests a relatively high level of accuracy with our algorithm when finding closest points and with a slight improvement from PA3 (by ~ 0.05). Since we implemented the full ICP algorithm (with max iteration=20) and KD Tree, the magnitude difference when comparing the expected pointer tip location against the closest vertex based on triangle mesh has been greatly improved (by a decimal magnitude compared to PA3).
- Linear versus KD Tree: we implemented linear search to find the closest point in PA3, which serves our purpose yet not efficiency. Hence we implemented KD Tree in PA4 to improve time and memory complexity this time. At max iteration = 20, KD Tree can still generate results within a few seconds at most, whereas linear search would take up to a few minutes.
- Bounded node: we did not include the bounding box in our final submission due to its lower accuracy in our development process, which is likely due to a bug with regards to how we structure other parts of the program i.e. distance projection heuristics. However, it sheds light on potential area of improvement in the future and can potentially even further improvement our time/space complexity when using in conjunction with KD Tree

Tabular Summary of PA 4 Debug Data Results Versus Sample Output Error Analysis

Data Set	Average Error across All Data Inputs
A	0.008
B	0.147
C	0.126
D	0.247
E	0.139
F	0.179

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 4 Debug and Unknown Coordinates Magnitude Difference Results

Data Set	A	B	C	D	E	F	G	H	J	K
Avg Magnitude Difference between Target and Bone	0.003	0.044	0.034	0.061	0.069	0.082	0.028	0.024	0.064	0.074

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](#)