Name: Ishita Unde and Angela Appiah
Date: 10/5/2025

Structure of the Program
The overall file structure of our code is as follows:

1. main.py: this file reads input data from the command line and creates an output file for each input file.
2. math_utils_cart.py contains all of our classes necessary for the cartesian math package (Frame, Rotations, and Points) as described in problem 1. It also contains our 3D point set registration algorithm, calibration method, methods for question 4, 5, and 6 as well as other necessary helper functions. Further description of each of these functions is presented below.
3. testing.py contains all of our units tests to test the functionality of each of the individual functions present in math_utils_cart.py. Further description of each of these functions is below.
4. file.io: contains our method to read 3D points from the input file in the command line.
5. setup.py: contains general description about the assignment as outlined in the assignment.
6. output: this folder contains all the outputs from tests a through k.

Our program functions are written in the file math_utils_cart.py. The functionality, both mathematical and algorithmic and the primary question answered by each function is as follows:

Primary Classes and Functions:

Class Point3D: This is a Python class designed to represent a point in three-dimensional space. Each point has x,y,and z coordinates. The class provides an initializer method which sets these coordinates to the parameters when a new Point3D object is created. It also includes a method to_array that converts the point into a NumPy array, allowing the coordinates to be easily used in numerical computations or vector operations. Additionally, there is a static method from_array that takes a NumPy array and returns a new Point3D object with the corresponding coordinates. This class is written as part of Question 1.

Class Rotation: The class is designed to represent and apply rotations in three-dimensional space using a 3×3 rotation matrix. It has a single attribute, matrix, which stores the rotation as a NumPy array. The class provides an initializer method that converts a given matrix into a NumPy array when creating a new Rotation object. It also defines the matrix multiplication operator allowing for rotation operations. If the right-hand operand is another Rotation object, the method returns a new Rotation representing the composition of the two rotations. If the operand is a Point3D object, the method applies the rotation matrix to the point's coordinates and returns a new rotated Point3D. This class is written as part of Question 1.

Class Frame: This class represents a rigid body transformation in three-dimensional space, combining a rotation and a translation. It has two attributes: rotation, which is a Rotation object

representing the orientation, and translation, which is a Point3D object representing the position. The transform_point method applies the frame transformation to a given Point3D by first rotating the point using the frame's rotation and then translating it by the frame's translation vector. The inverse method computes the inverse of the frame transformation, returning a new Frame whose rotation is the transpose of the original rotation and whose translation is adjusted accordingly to reverse the effect of the original transformation. The compose method allows combining two frame transformations, producing a new Frame that applies the effects of the original frame followed by another frame. This class allows for a convenient way to represent, invert, and combine rigid transformations in 3D space. This class is written as part of Question 1.

def. find_rigid_transform(points_A, points_B): This function computes the rigid body transformation that best aligns two sets of corresponding 3D points, points_A and points_B. It follows the mathematical method described by Arun et al. (IEEE PAMI, 1987) which was outlined in class. First, the points are converted into NumPy arrays and centered by subtracting their respective centroids to remove translation effects. Next, the covariance matrix between the centered point sets is calculated, and Singular Value Decomposition (SVD) is used to derive the optimal rotation matrix. The function handles potential reflection cases to ensure the resulting rotation has a determinant of 1, preserving proper orientation. Once the rotation is determined, the translation vector is computed to align the centroids of the two point sets. Finally, a Frame object is created using the calculated rotation and translation, representing the complete rigid transformation that best maps points_A onto points_B. This function is written as part of Question 2.

Class PivotCalibration: This class implements the pivot calibration method, which is used to determine the position of a tool tip relative to a tracked rigid body, as well as the fixed pivot point in the world coordinate system. The class has three attributes: tip_position, representing the estimated position of the tool tip in tool coordinates; pivot_point, representing the estimated fixed pivot point in world coordinates; and residual_error, representing the calibration error. The main method, calibrate, follows the pointing device calibration and parameter estimation described in class as part of the Calibrationlecture. This method takes a list of poses, where each pose consists of a rotation matrix and a translation vector. It constructs a linear system from these poses and solves it using least squares to estimate both the tip position and pivot point. The method also checks the solution quality by examining the system rank and verifying that the rotation matrices are valid. Finally, it updates the class attributes with the estimated points and the residual error, returning the tip position and pivot point. This class is written as part of Question 3.

def find_fd(frames, d_points): This function computes the fixed transformation F_d between the electromagnetic (EM) tracker coordinate system and the optical tracker coordinate system. As parameters, it takes all the calibration frames containing measured D marker positions from the optical tracker and the known geometry d markers in the EM coordinates. Then, it performs point-based registration to find the optimal rigid transformation that aligns the known D marker geometry in EM coordinates to their measured positions in optical coordinates. The result is the

transformation F_d, which represents the spatial relationship between the two tracking systems and it is used to convert points between EM and optical coordinate spaces.

def find_fa(frames, a_points): This function computes the transformation F_a for each calibration frame that relates the calibration object coordinate system to the optical tracker coordinate system. For every frame, it takes the known geometry of A markers on the calibration object and their corresponding measured position from the optical tracker. Then, similar to find_fd, it performs point-based registration to find the optimal rigid transformation that aligns the known A marker geometry to their measured optical positions. The result is a list of F_a transformations where each element corresponds to the position and orientation to the calibration object relative to the optical tracker in that particular frame. This enables the system to track how the calibration object moves through space during the calibration process.

def compute_expected_C(frames, fd, all_fa, c_points): This function calculates the expected positions of the EM sensors. As parameters, each calibration frame is passed, alongside the fixed transformation F_d, a list of all the transformations F_a, and all the known C marker positions. The function takes each calibration frame and computes the inverse of the fixed EM-to-optical transformation F_d. Then, it composes this with the frame specific calibration-object-to-optical transformation F_a. This computation creates a direct path from calibration object coordinates to the EM tracker. The result frame transformation is applied to each known C marker position on the calibration object, to generate expected positions. Additionally, the function computes the actual error by comparing the expected positions against the measured EM sensor readings for each frame. The function returns a list of all expected C marker positions.

def em_tracking(frames): This function performs electromagnetic (EM) tracking and pivot calibration to determine the position of a tool's tip and its fixed dimple point in 3D space. It begins by extracting the first frame of EM data to compute the centroid of its marker points, which is then used to define the local coordinate system (g_j_points). For each frame after that, it calculates the rigid transformation that aligns the current frame's points to this reference configuration, producing a series of frame transformations (F_G_frames). These transformations are then converted into pose representations of rotation and translation, which are passed to the PivotCalibration class to estimate the tool tip position (t_g) and the fixed pivot point (P_dimple) using the calibration method. This function is written as a part of question 5.

def opt_pivot_calibration(frames, d_points): This function performs optical probe pivot calibration to determine the probe tip position relative to the optical tracking markers. This function follows a similar implementation to em_tracking. First, a probe coordinate system established using the first frame's H markers. Their centroid is computed and a normalized H is created relative to this centroid (h_j_points). For each frame, it calculates the transformation F_d from known D marker geometry to measured D positions. Then, the inverse of this transformation is converted to the H marker measurements into EM coordinate space. This produces the transformation F_h that aligns the normalized H to the EM H positions. The rotation and translation are extracted and then, pivot calibration is performed on all the poses to

solve for the probe tip positions t_h and the pivot point p_dimple. These are the values that are returned.

Helper Functions and Classes:

def calculate_centroid(points): The calculate_centroid function computes the geometric center (average position) of a list of 3D points and returns it as a Point3D object.

def transform_points(frame, points): The transform_points function applies a given frame transformation to a list of 3D points, returning their transformed positions.

class CalibrationFrame: This CalibrationFrame serves a data container for organizing all the tracking measurements from a single calibration data frame. It stores three sets of 3D points: D, A, and C. These are the optical tracker measurements of the fixed D markers on the EM base, the optical tracker measurements of the A markers on the calibration object, and the EM tracker measurements of the C markers on the calibration object, respectively.

def read_calreadings(filename): This function parses through calreadings data files.It begins by reading the first line to determine the number of D markers, A markers, C markers, and the total number of measurement frames. For each frame, it reads all the D marker positions, A marker positions, and C markers and organizes them into the CalibrationFrame object. A list of these objects is returned to represent the comprehensive dataset.

def read_calbody(filename): This function parses through calbody data files. It begins by reading the first line to determine the number of D, A, and C markers and extracts the 3D coordinates of each. This function returns three separate lists of Point3D objects for D points, A points, and C points.

def read_optpivot(filename): This function reads an OptPivot data file and converse it into a list of OptPivotFrame objects. It reads the header to determine the number of D markers, H markers, and measurement frames. It, then, iterates through each frame to extract the 3D coordinates of all D and H markers. It organizes this data into a list of OptPivotFrames and returns it.

class EmPivotFrame: The EmPivotFrame class represents a single frame of pivot calibration data, storing a list of 3D points (g_points) captured in that frame.

def read_empivot(filename): This function reads an EMPivot data file and converts it into a list of EmPivotFrame objects. It first reads the header to determine the number of points per frame and the total number of frames. Then, for each frame, it reads the 3D coordinates of the points, creates Point3D objects, and stores them in an EmPivotFrame. The function returns a list of all frames from the file.

class OptPivotFrame: The OptPivotFrame class serves as a data container for organizing optical pivot calibration measurements from a frame. It stores two sets of 3D points, d_points and h_points, which contain the measured position of the fixed D makers on the EM base as captured by the optical tracking system and the measured positions of the H markers attached to the optical probe being calibrated, respectively.

def write_registration(output_file, N_c, N_frames, all_C_expected): This function outputs the expected C positions in the required file formation by writing the number of C markers and frames on the first line. Then, writing all the computed expected C marker coordinates for every frame on a new line.

def write_em_pivot(output_file, em_post_pivot): This function outputs the EM pivot calibration results by writing the 3D coordinates of the computed pivot post position to a file.

def write_opt_pivot(output_file, opt_post_pivot): This function output the Opt pivot calibration results by writing the 3D coordinates of the computed pivot post position to a file.

def write_output(reg_file, em_file, opt_file, output_file): This function combines the resulting files from write_registration, write_em_pivot, and write_opt_pivot into a single output file that follows the given format.

Validation

To verify that our program is working correctly, we executed two important actions. First, we created unit tests for each primary function (defined above) to test its mathematical and algorithmic approaches. Here, we weren't looking to see if it could produce reasonable results, rather if the functionality was correct. The tests we performed and what we were testing are listed below.

Unit Tests	Reasoning
test_basic_math_operations(): This test verifies the core 3D math functionality of the system, including point representation, rotation, and frame transformations. It ensures that rotations and translations are applied correctly and that inverse transformations restore the original coordinates.	*Test 1* checks that Point3D correctly stores coordinates and converts them to an array. *Test 2* confirms that rotation matrices are applied properly to points (e.g., a 90° Z-rotation). *Test 3* verifies that combining rotation and translation in a Frame produces the correct transformed point. *Test 4* ensures that applying the inverse frame transformation restores the original point, confirming mathematical consistency.
test_rigid_transform_algorithm() This test validates the accuracy of the 3D	*Test 1* checks that the algorithm correctly identifies and applies a pure translation between two point sets shifted by (3, 3, 3).

point registration algorithm implemented in find_rigid_transform(), ensuring it correctly estimates the rigid transformation (rotation and translation) between two point sets. It confirms that transformed points align with their expected positions, verifying both translation and rotation handling in 3D space	*Test 2* verifies that the function accurately recovers a 90° rotation around the Z-axis, confirming proper rotation computation without translation.
test_calibrate_function() This test validates the accuracy and robustness of the pivot calibration algorithm, ensuring it correctly computes the tool's tip position and pivot point from multiple tracked poses. It confirms that the algorithm performs reliably under both ideal and edge-case conditions while maintaining proper output structure and precision.	*Test 1* verifies that the calibration algorithm accurately recovers a known tip and pivot point when given consistent rotational and translational data. *Test 2* checks robustness against near-degenerate rotations (almost identity matrices), ensuring stability under minimal movement. *Test 3* confirms that the algorithm returns correctly shaped outputs ((3,) arrays) and a valid residual error value, verifying consistent output formatting.
This set of tests validates the em_tracking() function, ensuring it correctly processes electromagnetic (EM) tracking frames to estimate the tool's global transformation and dimple position. It checks for proper functionality under normal, error, and degenerate conditions to confirm robustness and reliability.	*Test 1* verifies that em_tracking() returns valid Point3D outputs when given multiple valid frames, confirming correct handling of standard input data. *Test 2* ensures the function raises a ValueError when provided with only one frame, validating input error handling. *Test 3* checks behavior with identical frames, confirming that the algorithm produces consistent and predictable results even when no movement occurs.

Second, we used the debugging files provided in PA1 Student Data. The results from these files and percent error is as follows.

File Dataset	N_c	# frames	Em_pivot post	Opt_pivot_post	Average C error (mm)
debug-a	27	8	(190.55, 207.35, 209.17)	(400.55, 402.12, 203.54)	0.00467
debug-b	27	8	(194.07, 209.94, 201.24)	(397.44, 395.78, 200.21)	0.457

debug-c	27	8	(195.55, 200.00, 205.23)	(392.64, 409.41, 202.13)	3.51
debug-d	27	8	(201.12, 191.98, 208.74)	(408.71, 401.41, 208.70)	8.39
debug-e	27	8	(200.55, 202.48, 195.47)	(392.77, 391.97, 209.77)	9.085
debug-f	27	8	(193.85, 189.06, 208.58)	(402.68, 396.81, 204.77)	11.15
debug-g	27	8	(201.02, 196.56, 205.46)	(405.57, 394.97, 207.55)	12.09

From these results, we were able to ascertain that our program is complete and produces reasonable results.

Results

We ran our program on the unknown files provided in PA1 Student Data.

File Dataset	N_c	# frames	EM_pivot post	Opt_pivot_post	Average C error (mm)
unknown-h	27	8	(190.46, 199.45, 186.46)	(395.82, 402.08, 192.61)	21.90
unknown-i	27	8	(207.57, 191.94, 197.84)	(399.87, 408.28, 194.15)	18.36
unknown-j	27	8	(197.92, 205.56, 188.93)	(408.50, 408.47, 203.17)	4.38
unknown-k	27	8	(187.13, 199.33,	(396.67, 393.62,	20.51

			206.38)	191.14)	

Our results show some variation in registration accuracy across datasets. For example, the best performance was displayed with a dataset unknown-j with an average error of 4.38mm. Whereas, our worst performance was with a dataset unknown-h with an average error of 21.90mm.This suggests that our implementation may be sensitive to input data that may not be of the best quality. Some potential reasoning for this could be the EM distortion, EM noise, and OT jiggle. However, these only contribute minimally to the overall error. For unknown h / i / k, it is likely that since we sample the first frame as our localization coordinate, those could be the noisiest measurements. This would skew the error of all the measurements to follow, hence why the overall average mean is so large.

Credits

The contribution from each member for this assignment was equal. Ishita and Angela discussed the methodology/pseudocode for each question before coding it to ensure both members knew the algorithmic and mathematical approach. Ishita coded questions 1,2,3, and 5. Angela coded questions 4,6. Angela also formatted main.py to match instructions and ran our program on the data provided. Ishita wrote the unit tests to test the functionality of each individual function. Both members contributed equally to the writing of this document.