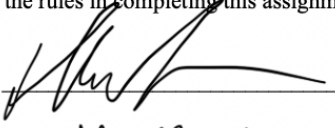


## Programming Assignments 1 & 2 601.455 and 601/655 Fall 2023

Please also indicate which section(s) you are in (one of each is OK)

### Score Sheet

Name 1	Hannah Puhov
Email	hpuhov1@jhu.edu
Other contact information (optional)	
Name 2	Nidhi Batra
Email	nbatra3@jhu.edu
Other contact information (optional)	
Signature (required)	<p>I (we) have followed the rules in completing this assignment</p> <div style="text-align: center;">   <u>Nidhi Batra</u> </div>

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	

## Algorithm of Approach

In this assignment, we are fitting polynomials to model distortion and then using these polynomials to “de- warp” the EM tracker space. We will use the de-warped EM tracker space to repeat our pivot calibration. Then we will use the resulting calibration to compute registration to a CT coordinate system. Finally, given some pointer data frames, we will report corresponding CT coordinates.

We are following the following procedure outlined in the assignment.

1. Input the body calibration data file and process it to determine the values of  $\vec{C}_i^{(\text{expected})}[k]$  corresponding to each  $\vec{C}_i[k]$  in each “frame”  $k$  of data.
2. Use the method described in class to produce a suitable distortion correction function.
3. Use this distortion correction function to repeat your “pivot calibration” for the EM probe.
4. Using the distortion correction and the improved pivot value, compute the locations  $\vec{b}_j$  of the fiducials points with respect to the EM tracker base coordinate system.
5. Compute the registration frame  $\mathbf{F}_{reg}$ .
6. Apply the distortion correction to all the subsequent values of  $[\vec{G}_1, \dots, \vec{G}_{N_g}]$ , compute the pointer tip coordinates with respect to the tracker base, and apply  $\mathbf{F}_{reg}$  to compute the tip location with respect to the CT image.

The first step of the procedure simply involves using the method we wrote in question 4 of programming assignment 1 to determine the values for  $C_i$  expected that correspond to each  $C_i$ . We iterated over each frame to calculate the expected positions and wrote these to an output1 datafile to account for when this wasn't given to us in non debugging data sets. We used pandas to assist with reading this data from the file and storing coordinates in a data frame<sup>1</sup>.

We then moved on to producing a distortion correction function. We followed the 3D calibration method of distortion outlined in class. This first involved creating a scale box to scale the values returned by the navigational sensor since Bernstein polynomials(the approach that will follow) are designed to work will for values between 0 and 1.

This was the equation followed to create the scale box.

$$: \frac{X - X^{\min}}{X^{\max} - X^{\min}}$$

Each  $x$  point ( $C_i$ ) was passed in and computed with the stored minimum and maximum point from its frame. We had multiplied each stored minimum and maximum by a tolerance of 1.01 that we found worked well for values that fell on the edge or outside the box. This gives us the point scaled between 0 and 1 in relationship to its relationship to the min and max values of the frame.

We then computed the interpolation polynomial using 5th degree Bernstein polynomials. We chose to use 5th degree as it was recommended in the slides, worked with our approach, and because by increasing it we could risk overfitting the data. We computed the polynomial for each coordinate,  $u$ , of the scaled point computed from the box scaling using the below formula.

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

$N$  was the degree of the Bernstein, 5 and  $k$  will vary from 0 to  $N^2$ , explained more below. The comb function from python's math package was used to assist in calculations<sup>3</sup>.

For later use, we wanted to store our Bernsteins in an  $F$  matrix as seen below.

$$\begin{bmatrix} \vdots \\ F_{000}(\vec{u}_s) & \cdots & F_{555}(\vec{u}_s) \\ \vdots \end{bmatrix}$$

The  $F$  is the tensor form interpolation polynomial that is found from the product of the Bernsteins as seen below.

$$F_{ijk}(u_x, u_y, u_z) = B_{5,i}(u_x) B_{5,j}(u_y) B_{5,k}(u_z)$$

We used this information to vary our  $i, j, k$  when computing the Bernstein for the coordinate (all the  $k$  variables in the general Bernstein equation) to find the corresponding interpolation polynomial that we could store in our matrix.

Once we had this  $F$  matrix with all the points, we knew we could be using it in a least squares problem since we had data about the known ground truth data. Our goal is to find the calibration matrix of  $c_s$  because this is what is needed for undistorting points from their interpolation polynomials.

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

We solved the least squares to find and store the calibration matrix for each frame. The  $p$  matrix used was the  $C_i$  expected calculated earlier for the corresponding frame.

We used this calibration matrix to find the undistorted EM pivot  $G$  array of values read by the sensor. This involved dotting the  $F$  matrix created for those values with the calibration matrix found above to find the ground truth of undistorted values read from the sensor.

We reperformed pivot calibration by calling our pivot calibration method from assignment 1 and passing in our undistorted  $G$  array of values read from the EM sensor.

Next, we wanted to find locations of fiducials with respect to the EM tracker base coordinate system. We were given the fiducial locations with respect to the CT system. We first set our base EM coordinate system as the first frame from the EM Pivot values, in order to keep it consistent with the calculation used for the EM Pivot Calibration. We then undistorted the entire array of EM Fiducials by running it through the above procedure. For each frame of EM Fiducials, we found a registration that would map the original pivot frame to the fiducial frame. We transformed the pivot to the fiducial frame using this registration in order to find the location of the fiducial. We did this because the calibrated pivot value shows the difference between the measured coordinates and the where the end of the pointer is, which would correspond to the exact location of the fiducials.

Once we had the fiducials in EM coordinates, we could find the registration frame. For this we found the registration between the bjs converted to point sets in both coordinate systems using our 3D point set to 3D point set registration algorithm written as part of programming assignment 1. This returned the rotation matrix and translation vector so we called a transformation method written in `pyratransform`<sup>4</sup> (the package adopted as part of assignment 1) to find the corresponding registration frame.

Lastly, we applied the distortion correction to our G data points from the NAV data file. We then computed the pointer tip coordinates with respect to the tracker base. To do this, we followed a similar procedure as in the step in which we found the EM fiducial locations in EM coordinates. We found a transformation from the base EM coordinate system to each frame of the navigation data, and transformed the tip location using that transformation. The difference between this step and the one for finding the fiducials, is that there is an additional transformation that must occur. Once the EM Nav data is in EM Pivot coordinates, we could transform the tip location using the EM to CT  $F_{\text{registration}}$  frame that we found earlier. This allowed us to find the tip location with respect to CT coordinates. Finally, we outputted each EM Nav tip location to the corresponding output file for the program as it was calculated.

### Structure of Program

The program is split into four types of files: the main executable file, .txt input/output (I/O) classes, the two main functional files, and helper functions. The main functional file, `main2.py`, is the one that becomes an executable to be able to run through the program. It takes in the user input for the file name, and calls all necessary functions to generate the final output files. The file I/O classes are split up so that there is one for each file type. For example, `CalReadings`, `EM Pivot`, `EM Nav`, etc. each have their own class. Creating objects of the class stores all respective information in that object. This includes all frames, points, and header values (i.e. the number of frames in the file). Helper functions are the ones that do the brunt of the mathematical work for the program. These are separated into different files and organized based on their corresponding task. For example, all functions involved in distortion correction are stored together.

- `main2.py`
  - This is the function that is generated into a Python executable. It prompts the user to enter the file name (i.e. 'pa2-debug-a'). It creates the class objects for all of the file I/O classes, and calls all relevant main functional methods from `distortionCorrection.py`, `FindingTipCT.py`, and the output writer class for `output1`.
- .txt I/O files

- This includes calBody.py, calReadings.py, ctFiducials.py, emFiducials.py, emNav.py, emOutput.py, emPivot.py, optPivot.py, outputWriter.py
- Each of the input classes contains two methods: an initialization method that gives the object the file the filename and the file folder, and a data setup method that reads in all relevant data from the text file. The attributes of each class are the data array(s) and number of frames, probe markers, etc. as necessary for each file type
- The output classes contain an initialization method that stores the file header and the output folder, as well as any relevant data that needs to be printed to the first line of the output file. They also contain methods to write any additional lines to the text file, such as pivot locations (for output1) and em Nav locations in ct coordinates (for output2).
- Main functional classes
  - distortionCorrection.py contains a class that holds all necessary functions for calculating distortion correction. The class takes in the calreadings and the calobject classes, and stores them as class attributes for later access. It generates the scale box and scales points for use with Bernstein polynomials. Using this, as well as the data from calreadings and calobject after being passed through the function from output1 that generates  $ci_{expected}$ , it generates a calibration matrix. This is all incorporated in a class because that way every time a data array needs to be undistorted, it can use the existing calibration matrix that is stored in the class object, rather than remaking it from scratch, which is computationally very expensive.
  - FindingTipCT.py contains a class that holds all necessary functions for finding the EM Nav tip location in CT coordinates. EM Pivot, CT Fiducials, EM Fiducials, EM Navigation, EM Output, and DistortionCorrection classes are all passed into it from the main method. The functions included are ones that find the fiducials in EM coordinates, generate a registration frame from EM to CT coordinates, and find the EM Nav tip locations in CT coordinates and output them to the output file. There is also a method to find a new EM pivot tip location based on the undistorted  $ci_{expected}$  array. These are all incorporated into a class for easy access of class attributes, such as the new EM pivot location, as well as to make it easier to call relevant methods from main2.py without needing to pass in an excess of parameters.
- Helper functions
  - Helper functions include mean\_point, which finds the average point from a list, pivot\_calibration, and other methods that were written for the first assignment that allow for code reuse such as registration finders using a PointSet class that we created.
  - We used python math.comb to find combinatoric values, numpy.linalg.svd to do single value decomposition, and pytransform3D to create and run frame transformations from a rotation and translation matrix<sup>5, 6, 4</sup>.

### Validation of Approach

We knew our  $ci_{expected}$  function worked since we tested it in programming assignment 1 and also have the correct expected output in that assignment. Similarly, we have already tested our function to find the registration between 2 point sets and pivot calibration function.

The simplest tests to run were for testing our scale to box function and our bernstein function generator. We wrote unit tests for these and the file is included. These tests passed. We move where the functions are located during testing since we had written in debugging tests so this needs to be checked each time before running the tests.

To test our distortion correction function, we compared our distorted points with the new points and looked at the distortion factor as seen in the auxiliary file for each debug set to make sure the more distorted points had corrected points that were more different. We also ran a pivot calibration with the undistorted points to confirm our undistortion was working. We used Cal EM observed - cal EM estimated from auxiliary 2 to compare points from Ci expected and output 1.

We were able to confirm our EM pivot post position matched the expected as seen in the auxiliary file for the debug sets using the debugger to see that value.

To test the locations of our fiducials, we tried to look at the  $F_{\text{registration}}$  we got compared to the expected  $F_{\text{registration}}$  from the auxiliary files of the debug data sets. This is where we encountered the first error and then spent time trying to debug our fiducial algorithm (since the code to find the registration had been tested and was correct from assignment 1). We were not able to figure out why our fiducial algorithm was not working.

Since we then found the tip location with respect to the CT image using the locations of the fiducials and registration frame, the last check we did was comparing our tip locations with the expected tip locations. Our goal was to fix our fiducial and finding tip location algorithms (they followed similar procedures) in order to minimize the error of our tip locations and the tip location we should be getting. The code we submitted got us as close as we could get after spending time debugging and modifying our algorithm.

## Discussion of Results

We knew our Ci expected values matched the correct (as in output one) based on our output analysis from programming 1. We spot checked to confirm this.

### Percent Error For Each Value in Data Set A

First Coordinate	Second Coordinate	Third Coordinate	Average
86.4572%	5.9618%	86.8864%	
96.263%	9.8117%	41.7969%	
109.7481%	1.0423%	41.5041%	
19.6468%	5.0627%	38.534%	
			<b>45.22%</b>

There was no distortion, noise, or jiggle to data set A. Most of our other data followed similar trends and there didn't seem to be a trend about which data differed more according to the distortion, noise, or jiggle. The distribution of error seemed pretty random among data sets.

Our best guess for reasons for our error are errors in our fiducial location calculations as explained in the validation of results section. This was confirmed by our output 1s pretty much matching the expected for all data sets, independent of distortion, and there not being a significant difference among datasets with the amount of error in the tip calculation.

Like our programming 1 assignment, sometime when running a LinAlgError appears but to solve this we found just rerunning the program resolves it.

The exact outputs for each data set are given in the OUTPUT folder.

#### Workload Division Statement

Most of this assignment was done together. We both came up with approaches for how to solve the problems and wrote the code while meeting in person. We also debugged primarily together. The report was added to by both of us.

## Sources

1. McKinney, W., & others. (2010). Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference* (Vol. 445, pp. 51–56).
2. Joy, K. (2000). Bernstein Polynomials. Davis, CA: Computer Science Department, University of California, Davis. Retrieved from <http://www.inf.ufsc.br/~aldo.vw/grafica/apostilas/Bernstein-Polynomials.pdf>
3. Van Rossum, G. (2020). *The Python Library Reference, release 3.8.2*. Python Software Foundation.
4. Fabisch, A. (2019). pytransform3d: 3D Transformations for Python. *Journal of Open Source Software*, 4(33), 1159. <https://doi.org/10.21105/joss.01159>
5. *Numpy.linalg.svd*#. numpy.linalg.svd - NumPy v1.26 Manual. (n.d.). <https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html>
6. *Numpy.linalg.lstsq*#. numpy.linalg.lstsq - NumPy v1.26 Manual. (n.d.). <https://numpy.org/doc/stable/reference/generated/numpy.linalg.lstsq.html>