Programming Assignment #3

Computer Integrated Surgery I

11/18/2021

Sean Darcy
sdarcy2@jhu.edu

Alexandra Szewc
aszewc1@jhu.edu

Description of formulation and algorithmic approach

We decided to develop our program in Python. Specifically, we chose to develop a python package in Visual Studio Code (https://code.visualstudio.com).

In this programming assignment, we implemented the linear search/matching component of the Iterative Closest Point (ICP) algorithm (Besl and McKay, 1992). We took a triangular surface mesh modeling a bone as input, as well as a set of points "sample" readings giving the positions $\mathbf{a}_{i,k}$ and $\mathbf{b}_{i,k}$ of optical markers corresponding to two rigid bodies A and B. Rigid body A is used as a tracked pointer while rigid body B is rigidly screwed into the bone in some unknown orientation.

We began by writing file Input/Output objects, readers.py and writers.py, respectively. More information on the structure of the program can be found in the following "Program overview" section. Once implemented, we stored the triangular mesh data in two numpy arrays- one of which corresponding to the vertex coordinates and the other corresponding to the indices of the vertices for each triangle.

Next, we implemented a closest.py object, with notable methods find_closest(), brute_force(), triangle_bound(), and distance(). find_closest() takes a point of interest, as well as the two arrays corresponding to the bone mesh data described above as arguments. In addition, there is a boolean parameter "brute" that currently defaults to true. "Brute" refers to the sequential search for a closest point through each triangle in our linear data structure. In the next programming assignment, this boolean parameter will offer flexibility in our "find closest point on mesh" routine as we implement more efficient methods of search through data structures like covariance trees. For this assignment, we decided to stick to linear search in order to ensure a solid understanding of the working algorithm and to thoroughly debug our implementation before trying to speed things up with more efficient O(lg(N)) data structures.

find_closest() passes its arguments to brute_force(), which finds the closest point on the mesh corresponding to the point of interest.

To find the closest point, we begin by defining a variable min as infinity. Next, we begin to iterate through each triangle. For each triangle, we define corresponding vertices **p**, **r** and **q**. Next, we solve the following system for lambda and mu via numpy least squares:

$$\mathbf{a} - \mathbf{p} = \text{lambda}(\mathbf{q} - \mathbf{p}) + \text{mu}(\mathbf{r} - \mathbf{p})$$

Where **a** is the point of interest and **p**, **r**, and **q** are the triangle vertices. If lambda is negative, mu is negative, or lambda + mu is greater than one, then we have determined that the closest point on the triangle is on one of its edges, and thus call the method triangle_bound(), with parameters corresponding to the two vertices defining a particular edge of the triangle depending on which of the three cases just mentioned evaluated to true as well as a parameter again corresponding to the point of interest. This method finds the closest point on that particular edge. If lambda and mu are both greater than or equal to 0 and mu + lambda is less than 1, then the closest point is actually on the face of the triangle and we solve for that point via the following equation:

$$\mathbf{c} = \mathbf{p} + \text{lambda}(\mathbf{q} - \mathbf{p}) + \text{mu}(\mathbf{r} - \mathbf{p})$$

For each triangle, we update the value for min by evaluating whether min is greater than the distance between the point of interest and the closest point and updating its value to that distance if it is, because we have found a new minimum distance. We also update a pointer variable closest to that point. By the end of the linear search, we have evaluated every triangle in the mesh for its closest point corresponding to our point of interest and return the closest out of these points as contained by the variable closest, with min corresponding to the distance between this point and our point of interest.

This process is repeated for every point of interest in the data set. The points of interest themselves represent readings from the tracked pointer (Rigid Body A) with respect to Rigid Body B coordinates. To get these points, labeled **d**, we must first compute the Frames F_A and F_B of the rigid bodies with respect to the optical tracker using rigid registration between the optical markers on each respective rigid body as read by the optical tracker and their locations with respect to the rigid body itself. Once we have F_A and F_B, we can compute **d** as follows:

$$\mathbf{d} = F\_B^{-1} F\_A A_{tip}$$

We repeat for each set of tracked pointer readings. Now, we assume Freg is the unknown transformation:

$$\mathbf{c} = \text{Freg} * \mathbf{d}$$

In this assignment, since we are not computing Freg iteratively by optimizing a distance function, we assume Freg = I. Hence, the final output corresponds to each closest point **c** on the mesh corresponding to sample point **s** = I * **d** = **d,** as well as the corresponding distance between **c** and **s**.

Program overview

In the past, we had developed our programming assignments on Google Collab as Python Notebooks (.ipynb). This worked well early on because it allowed us to work together on the same code in real time, but became tedious as our program grew. In programming assignment 2, debugging had become incredibly time consuming. Even navigating the notebook itself became a chore. Thus, in this assignment we focused on program modularization to easily document, debug, and share our work.

The program is contained within the PA3 folder in our CISProgrammingAssignments repository, linked here:

https://github.com/SeanSDarcy2001/CISProgrammingAssignments/tree/main/PA3

The program was developed using a Python 3.8.8 conda interpreter. The program itself contains 4 key directories: ciscode, data, outputs and tests. In addition are two .py files- setup.py and pa3.py. pa3.py contains the main() driver function. The following provides a hierarchical overview of our program's structure.

PA3

Note: Source code descriptions also available in README.TXT. Class comments in code as well.

Sub-Directories:

ciscode:

Sources:

frame.py: contains frame class with frame manipulation methods

pointer.py: contains pivot calibration code (unused here)

readers.py: contains classes for reading input files

writers.py: contains classes for writing output files

closest.py: contains methods for computing closest point on triangle mesh

testing.py: contains testing and result validation suite for PA3

data:

Contents:

Abundance of provided input and debug data in .txt format.

outputs:

Contents:

Output .txt files generated from input data via classes defined in writers.py

tests:

Sources:

test_something.py: unused

Source Code:

closest.py

Methods:

find_closest(): Computes closest vertex to point and returns distance.

brute_force(): Linearly searches triangles for closest surface triangle.

triangle_bound(): Computes point projected on triangle edge.

distance(): Computes distance between two points.

frame.py

Note:

This version of the object was developed by Benjamin D. Killeen.

pointer.py

Note:

This version of the object was developed by Benjamin D. Killeen.

readers.py

Reader Classes:

ProblemXBodyY: Parses the LED marker data.

ProblemXMesh: Parses the body surface definition data.

SampleReadings: Parses the sample readings data.

OutputReader: Parses a formatted output file for programming assignment 3.

writers.py

Writer Classes:

Writer: Abstract output formatter class.

PA3 (writer): Output formatter class for programming assignment 3.

testing.py

Methods:

test_similarity(): Function to test similarity between given output file, computed output values.

resultsTable(): Function to generate % error vs coordinate threshold value plots for each debugging set.

pa3.py: main method + program driver

Class Summary:

ProblemXBodyY

ProblemXMesh

SampleReadings

OutputReader

Writer

PA3(Writer)

pa3.py

Validation approach

We implemented and conducted testing for each component of our program. We chose to organize our tests in code debug logs directly beneath those code segments corresponding to the subject of each respective test for convenience, display, and efficiency. These unit tests include:

-mesh input printing
-code execution progress printing
-output distance printing

-test_similarity(comp, true, error)

Function to test similarity between given output file and computed output values. Takes parameters comp, true, and err. Parameters comp and true are lists of computed and given output values respectively for some required output parameter. Parameter error is the threshold representing the maximum difference between any coordinate true value and our computed value for computed output to be considered correct. The function returns the % accuracy of correct values.

-resultsTable(name, out, ref, ans)

Function generates content described below and takes debug data—including the name of the set, the computed outputs, the outputs given, and the answers given—as input.

Discussion of Results

We were able to verify that individual components of our program work as expected via unit testing, as discussed previously. To evaluate the overall performance of our program, we ran a suite of test_similarity() tests for every debugging data set, including “-Output.txt” and “-Answer.txt” files provided for programming assignment 3, for multiple values of the error parameter, which represents some threshold value for computed coordinates to be considered accurate. This test_similarity suite can be found in “ciscode/testing.py”. In addition, the driver file “pa3.py” also generates output for the unknown data sets, but this output has no reference to test similarity. Below are the results for each debug set for varying values of error threshold. Each debug set has a plot for each output type (d, c, or differences) and for each comparison file given (“-Output.txt” and “-Answer.txt”).

We wanted to find a threshold margin such that each set of debugging data had an accuracy rate greater than 99%, or an error rate less than 1% to determine the optimal error margin of error for our data. The optimal values for each debug set are shown on each plot in the “plots” folder of our submission.

Contributions

Seby

- Wrote algorithm and mathematics documentation
- Helped debug code with erroneous outputs
- Wrote description and formulation of algorithmic approach and related

Alex

- Reformatted programming environment
- Wrote code for project
- Added validation/testing suite
- Documented results discussion
- Documented methods
- Documented validation approach

References

@misc{benjamindkilleen2021Nov,
author = {benjamindkilleen},
title = {{ciscode}},
journal = {GitHub},
year = {2021},
month = {Nov},
note = {[Online; accessed 12. Nov. 2021]},
url = {https://github.com/benjamindkilleen/ciscode}
}