600.445 Computer Integrated Surgery
Programming Assignment 1

Ravindra Gaddipati, Doran Walsten

October 25, 2016

Basic Transformations & Pivot Calibration

1.1 Usage

The codebase is built with cmake. Inside the project directory,

```
mkdir build && cd build
cmake .. && make
```

The executable is also included. There are two main usages. To run with the data files, pass the file root. To run the test cases and print the summary, pass "test". Note that we expect to find an OUTPUT directory in the folder you are executing from. If the program finds a debug output file, an error report between the computed and debug values will be printed.

```
bin/ciscal <fileroot>
    e.x. bin/ciscal INPUT/pa1-debug-a
bin/ciscal test
```

The CLI usage information is printed if no arguments are passed.

```
EN.600.465 Computer Integrated Surgery PA 1,2
Ravi Gaddipati, Doran Walsten
Oct 25 2016

Usage:
ciscal <fileroot>

The following files should exist:
fileroot-calbody.txt
fileroot-calreadings.txt
fileroot-empivot.txt
fileroot-optpivot.txt
Output file written to:
OUTPUT/filename-output-1.txt
Where filename does not include the path.

To run test cases and exit:
ciscal test

This program makes use of Eigen (eigen.tuxfamily.org)
and doctest (github.com/onqtam/doctest)
```

This program uses Eigen (eigen.tuxfamily.org) for linear algebra and numerical methods as well as doctest (github.com/onqtam/doctest) to implement test cases.

1.2 Testing

While the code was implemented, unit tests were used to verify functionality. Doctest, developed by Victor Kirilov (https://github.com/onqtam/doctest), was used as a testing framework. With Doctest, test cases are outlined for each function at the bottom of their respective header files. The structure is as follows.

```
TEST_CASE("Description") {
[Preamble]
    SUBCASE {
        ex. Test Assertions
        CHECK(LHS == RHS)
        CHECK_THROWS(expr)
    }
    [Conclusion]
}
```

During the execution the preamble is executed for each subcase, followed by the test, followed by the conclusion. To execute all the tests, the argument "test" can be passed to the main executable. Specific tests implementations are outlined below. After unit tests functionality was tested on the debug files.

1.3 General

Eigen(https://eigen.tuxfamily.org/) was used as the central numerical library. Most functions return Eigen objects, and are templated on the type of Eigen coefficient. The majority of the implementations are in the header files. Specific algorithms implementations are described below. There are some common objects used throughout the codebase, described in the following files.

1.3.1 Point Clouds and Points

`pointcloud.h`

Provides a `PointCloud` object, internally representing the point cloud as an Eigen::Array, were each row is a point. The object provides various functions to easily transform the point cloud and compute the centroid. Tests developed for the class include the computation of the centroid, transformation, and point insertion. A `Point` is an alias for a matrix of size 3 by 1, or `Eigen::Matrix<T,3,1>`.

- `Point at(size_t i)` : Returns the ith point.

- `Point centroid()` : Compute the centroid.

- `PointCloud center()` : Returns a new `PointCloud` with its centroid at $(0, 0, 0)$

- `PointCloud center_self()` : Subtract the centroid from all points.

- `void transform(Eigen::Transform)` : Applies the transformation to each point.

1.3.2 File Parsers

`files.h`

This contains the the parsers for all the data files. They provide an interface to to read and interact with file data, returning `PointCloud` objects. The following file parsers are provided, where each can be constructed with a filename.

- **CalBody** : Represents a Calibration Body with markers. The defined marker `PointCloud` are obtained with the functions

    - `opt_marker_embase()` : Optical markers on the EM base
    - `opt_marker_calobj()` : Optical markers on the calibration object
    - `em_marker_calobj()` : EM markers on the calibration object

- **CalReadings** : Represents calibration readings varying with time. The same functions as `CalBody` are provided, but vectors of point clouds are provided.

- **EMPivot** : Provides a vector of `PointCloud` frames with `em_marker_probe()`.

- **OptPivot** : Provides vectors of `PointCloud` frames with `opt_marker_embase()` and `opt_marker_probe()`

- **OutputParse** : Parses an Output file. This is primarily used to compute the error between our output and the debug files.

Each file parser is unit tested by creating a temporary file with known data, and checking we parse the correct information.

1.3.3   Utilities

`utils.h utils.cpp`

These files contain general purpose functions. The main funciton used here is `split(str, delim)`, which splits a string by a delimiter and returns a vector.

1.4   3D point set to 3D point set registration algorithm

The first step of this assignment was to develop a method to complete a 3D point cloud to 3D point cloud registration. Ultimately, we are looking for the rotation R and the translation p which satisfies

$$R\vec{p}_{A,i} + \vec{p} = \vec{p}_{B,i}$$

for each point in each rigid body. To do so, we decided to use Horn's method with quaternions to compute the rotation between the point clouds. Using this rotation, we can estimate the translation between the point clouds by determining the distance between the centroid of the original point cloud and the rotated 2nd point cloud. The approach is described below.

Before the algorithm is run, the two point clouds supplied must be in respect to the same reference frame and have the same number of points in order to compute the transformation correctly. We will represent the original points in each cloud as $\vec{p}_{A,i}$ and $\vec{p}_{B,i}$ respectively.

The first step of the algorithm is to compute the centroid of each point cloud, $\hat{a}$ and $\hat{b}$ for the first and second clouds, respectively. Then, compute the difference between each point in the cloud and the centroid. For the first point cloud, we will call these vectors $\vec{a}_i$ and for the second point cloud $\vec{b}_i$.

$$\hat{a} = \frac{1}{N_A} \sum_i \vec{p}_{A,i}$$

$$\hat{b} = \frac{1}{N_B} \sum_i \vec{p}_{B,i}$$

$$\vec{a}_i = \vec{p}_{A,i} - \hat{a}$$

$$\vec{b}_i = \vec{p}_{B,i} - \hat{b}$$

From here, we begin to use Horn's method to compute the rotation between the two point clouds, described below. (Based on summary of the method described in the lecture notes).

1. Compute

$$H = \sum_i \begin{bmatrix} a_{ix}b_{ix} & a_{ix}b_{iy} & a_{ix}b_{iz} \\ a_{iy}b_{ix} & a_{iy}b_{iy} & a_{iy}b_{iz} \\ a_{iz}b_{ix} & a_{iz}b_{iy} & a_{iz}b_{iz} \end{bmatrix}$$

2. Compute

$$G = \begin{bmatrix} trace(H) & \Delta^T \\ \Delta & H + H^T - trace(H) * I \end{bmatrix}$$

where $\Delta^T = [H_{2,3} - H_{3,2}, H_{3,1} - H_{1,3}, H_{1,2} - H_{2,1}]$.

3. Compute the eigenvalue decomposition of G.

4. The eigenvector corresponding to the largest eigenvalue is the quaternion of the rotation between the two point clouds.

5. Once the Rotation R is computed, we use this rotation to estimate the translation $\vec{p} = \hat{b} - R\hat{a}$

Implementation

`horn.h`

```
template<typename T>
Eigen::Transform<T, 3, Eigen::Affine>
cloud_to_cloud(const PointCloud<T> &cloud1,
               const PointCloud<T> &cloud2);
```

This function accepts two `PointCloud` objects, and computes the transformation from $cloud1$ to $cloud2$ using Horn's method. The centroid of each cloud is taken, and the H and G matrices are constructed as outlined above. Since G is a symmetric matrix, the `Eigen::SelfAdjointSolver` was used to compute the eigenvectors of G. The eigenvector corresponding to the largest eigenvalue is then interpreted as a Quaternion, and the corresponding translation computed. The two operations are returned as an `Eigen::Transform<T,3,Eigen::Affine>` object that applies the rotation first, then the translation.

Testing

To test the function, a random `PointCloud` is first made. A second point cloud is generated by transforming each point by a known rotation about the X,Y,Z axes, followed by a translation. The computed transformation is then compared with this known transformation. We check that the computed rotation and translation matricies are equivalent to the original transformations. Please review the Doctest test cases at the bottom of `horn.h`.

1.5   Distortion Calibration

To complete the distortion calibration component of the assignment, we simply needed to read in frames of optical tracker data D_i, A_i, compute the transformations between the calibration object and EM base to the optical tracker, then use these transformations to compute the expected value of the EM markers in the EM Tracker reference frame.

The transformation from the optical tracker to the EM base is computed using the Horn method described above. We are provided locations of optical markers $\vec{d_i}$ on the EM base in the EM base coordinate system. We compute the transformation $F_D^{(k)}$ between that point cloud and the point cloud of measurements $\vec{D_i}$ in a specific frame of data k. The same is done for the optical markers on the calibration object itself to find the transformation $F_A^{(k)}$ between markers $\vec{a_i}$ on the calibration object and the optical tracker measurements A_i on frame k. Because the EM markers c_i on the calibration object belong to the same reference frame as the optical markers $\vec{a_i}$, the transformations above can be used to estimate the location of the EM markers in the EM base space. This is done through the following equation:

$$\vec{C}_{i,exp} = F_D^{-1} F_A \vec{c_i}$$

.

This calculation is completed for every marker on every frame of data provided.

1.5.1   Implementation

`distortion_calibration.h`

```
template<typename T>
Eigen::Matrix<T, 6, 1>
std::vector<PointCloud<T>> distortion_calibration(
const CalBody<T> &body, const CalReadings<T> &readings)
```

The distortion calibration function accepts a representation of the calibration body object, which contains point clouds for the EM base/calibration object markers $\vec{d_i}, \vec{a_i}, \vec{c_i}$. This information was extracted from the text files previously using functions within `files.h`. It also accepts a representation of the calibration readings, which contain a collection of vectors of point clouds representing data on each frame for each marker type. Within the body of the function, the point clouds representing the markers on the EM base and calibration object are stored in variables and used to compute the transformation from the base/object reference frame to the tracker reference frame on each frame of data. This transformation is computed using `cloud_to_cloud()`, the Horn method defined in `horn.h`. At this point, we have the two transformations F_D, F_A represented in Eigen. The final step of the inner loop is to transform the entire point cloud of EM

markers $\vec{c}_i$ to the EM frame using `c.transform(F_D.inverse() * F_A)`. This transformed point cloud is stored and written to the output file.

Because this problem is a simple example of the use of our Horn method, we were confident in the test cases in `horn.h` and simply compared the results of this function to the debug data provided.

1.6 Performing a Pivot Calibration

Within the pivot calibration, it is our goal to determine the vector representing the tip of the pointer in the probe frame and the location of the calibration post in the EM tracker frame. This is done for both the optical tracking system as well as the EM tracking system. This is accomplished by determining the transformation between the pointer reference axis and the EM tracker axis on each frame of data using the computed locations of the markers on the probe. Below describes the protocol of doing so for the probe with EM markers.

Define the Probe Axis
As suggested in the assignment, we define the probe coordinate system as the centroid of the EM markers in the probe. The vectors representing each of the markers in this space is the difference of the vector to the marker and this centroid. In all frames of data, these positions never change because we assume that the markers do not move on the probe.

As provided in the assignment, this axis can be defined using the first frame of data. The centroid of the markers in the EM Tracker space $\vec{G}_0$ is used to compute $\vec{g}_i$: the difference of the actual marker measurement with the centroid to generate coordinates in the probe space

$$\vec{G}_0 = \frac{1}{N_G} \sum_j G_j$$

$$\vec{g}_i = \vec{G}_i - \vec{G}_0$$

In addition to defining the axis of the probe, this operation generates a point cloud in the probe space which can be transformed to each new point cloud in each frame of data.

Compute Transformation from EM Tracker to Probe on each frame
The next step is to determine the transformation between the probe point cloud and the point cloud in the EM Tracker space on each frame k of the data, $F_{MG}^{(k)}$. This is completed using the Horn Method described previously.

Compute the tip and the post vectors
The true goal of the calibration step is to determine the vector representing the tip of the probe in the probe space as well as the location of the calibration post in the EM tracker space.

$$F_{DG}^{(k)} \vec{p}_{tip} = \vec{p}_{post}$$

$$R_{DG}^{(k)} \vec{p}_{tip} + \vec{p}_{DG}^{(k)} = \vec{p}_{post}$$

$$R_{DG}^{(k)} \vec{p}_{tip} - \vec{p}_{post} = -\vec{p}_{DG}^{(k)}$$

$$\begin{bmatrix} R_{DG}^{(k)} & -I \end{bmatrix} \begin{bmatrix} \vec{p}_{tip} \\ \vec{p}_{post} \end{bmatrix} = -\vec{p}_{DG}^{(k)}$$

When this is expanded to every frame in the data:

$$
\begin{bmatrix} \vdots & \vdots \\ R_{DG}^{(k)} & -I \\ \vdots & \vdots \end{bmatrix} \begin{bmatrix} \vec{p}_{tip} \\ \vec{p}_{post} \end{bmatrix} = \begin{bmatrix} \vdots \\ -\vec{p}_{DG}^{(k)} \\ \vdots \end{bmatrix}
$$

We can thus use least squares in order to solve this system of linear equations to solve for each vector.

For the optical tracking system data, an additional step needs to be completed in order complete the calibration within the EM tracker frame instead of the optical tracker frame. As above, we can define the EM base axis as the centroid of the optical markers, $\vec{D}_0$ and generate a point cloud in this reference frame by computing $\vec{d}_i = \vec{D}_i - \vec{D}_0$ for each measured vector D_i in the first frame of data. The transformation from the optical tracker space to the EM base space is computed using the Horn method as $F_{OD}^{(k)}$ on each frame of data k.

To complete the calibration of the probe, we relate the location of the post in the EM base space to the tip of the probe in the probe space. We first transform all vectors H_i in the optical tracker space into the EM base reference frame by multiplying by F_{OD}^{-1}. Because these points are now in the EM base reference frame, we can use the exact same method as described above to compute p_{tip} and p_{post}.

Implementation

`pivot_calibration.h`

```
template<typename T>
Eigen::Matrix<T, 6, 1>
pivot_calibration(const std::vector<PointCloud<T>> &frames)
```

The pivot calibration function accepts a a vector of `PointCloud` objects, where each point cloud is a frame of data. Internally, the problem is posed as an $\mathbf{A}\vec{x} = \vec{b}$ problem as outlined above. $\mathbf{A}$ is constructed with dimension $3N \times 6$, and $\vec{b}$ is constructed with dimension $3N \times 1$, where N is the number of frames. The first frame is centered to serve as a reference frame, the the transformation from the reference to each frame is computed using `cloud_to_cloud()` and used to populate the $\mathbf{A}$ and $\vec{b}$ matricies. Once populated, the Eigen Jacobi SVD solver is used to compute $\vec{x}$. This results in a 6×1 matrix, where the upper 3×1 block is the vector from the reference frame centroid to the tip $\vec{t}$, and the lower block is the the post position $\vec{p}$.

The above method is used directly for the EM coordinates case in problem 5. For the optical tracker, a separate method is called:

```
template<typename T>
Eigen::Matrix<T, 6, 1>
pivot_calibration_opt(const std::vector<PointCloud<T>> &opt,
const std::vector<PointCloud<T>> &em)
```

In this function, the optical coordinates stored in `opt` for each frame is transformed to EM base coordinate by simultaneously computing the transformation from optical tracker coordinates to EM base coordinates through the Horn method. Once this is completed for every frame of data, the frames are sent to the `pivot_calibration` method above to compute the estimated post location.

Testing

To test the pivot calibration function, a random point cloud was generated, as well as a random post position. The point cloud was rotated and translated to the post, giving us multiple frames about the post. In this scenario, the true $\vec{p}_{tip}$ is the post vector minus the point cloud centroid. We then performed the pivot calibration to ensure the predicted values were the same (within numerical error) of the actual $\vec{p}_{tip}$ and $\vec{p}_{post}$.

1.7 Discussion

In the `OUTPUT` directory, a summary of the errors on the provided debug data is written to `debug_error.txt`. For the EM Pivot calibration, our absolute error was minimal, always within 10^{-2} of the expected values. However, our optical pivot calibration data was consistently off by 75 in every dimension on the initial debug files (a - c) with fluctuations around these values for later files. We attempted to debug this issue but could not find the source of the problem. We think that somewhere in the transformation of the $\vec{H}_i$ vectors in the optical tracker space to the EM base space that some additional translation is thrown in. It may have something to do with the centroid of the optical trackers on the EM base because the x and y coordinates of the centroid are both 75. However, the z component is in the negatives thousands. This issue is something that we need to be wary of in future assignments.

For the calculations of the expected value of the EM markers in the calibration object, our RMS error was minimal as well. This error peaked at about 1 for some of the later debug files. Because of these results, we are confident that our point cloud transformation methods are accurate and can be helpful in future assignments.

1.8 Contributions

Both partners contributed equally. Doran primarily worked on the distortion calibration and pivot calibration functions, and Ravi primarily worked on the support code and Horn's method. We contributed in debugging all functions and test.