

MPHYG002: Coursework #1

Due on Friday, March 18, 2016

Maria Ruxandra Robu- 14042500

This coursework implements the Iterative Closest Point (ICP) algorithm in two steps. The Point Based Task outputs a least squares estimation of the transformation between two given point clouds. The Surface Based Task takes as input two point sets with no assumption on the pair correspondence. However, an essential assumption with ICP is that there is a good initial alignment between the points. So, this algorithm iteratively minimizes the error:

$$RMS(R, t) = \sum_{i=1}^n ||p_i - Rq_i - t||^2 \quad (1)$$

ICP overview - Given two point clouds p and q :

- for each $q \in \{q\}_M$ find the closest match in $\{p\}_N$
- reject bad pairs (optional)
- get the least squares estimation of the transform $T = [R|t]$
 - center the point clouds at their origin by subtracting their means
 - compute the covariance matrix C of the centered \tilde{p} and \tilde{q}
 - extract R from the singular value decomposition of C ($R = UV'$)
 - the translation vector is $t = \bar{p} - R\bar{q}$
- apply the current transformation $q_i = Rq_i + t$ and start from the beginning

This implementation of the algorithm stops iterating when the RMS stops decreasing. So, if the difference between two consecutive errors is less than a threshold, the algorithm is considered to have converged.

All of the tasks have been implemented in one stand-alone project and their documentation is structured in five sections. Hence, the implementation and design choices are detailed in Section 1. The second section goes over the basic error checking and the testing implemented with Catch. Sections 3 and 4 illustrate the usage of the entry points for Point Based and Surface Based Registration, respectively. Finally, section 5 discusses the application of ICP in medical imaging.

Section 1

Least Squares Estimation and Surface Based Registration - Implementation

The project is structured in the main classes: PointCloud, ICPRegistration and ExceptionICP. See Fig. 1 for a UML diagram of the project. All of the functions and classes were built in the namespace ICP_MPHYG02, to avoid any name clashes with other libraries.

Whenever we have a point set, a new PointCloud object is created. The points are kept in a dynamic matrix from the Eigen library, which is efficient for big point clouds. New objects can be created by reading the input from a file or from another matrix. The copy constructor and assignment operator have been implemented as well.

The data is read from the file given a flag:

- POINT_BASED_FLAG for having one point per line (x y z)
- SURFACE_BASED_FLAG for having 3 points per line (x1 y1 z1 x2 y2 z2 x3 y3 z3)

The reading was made robust by using std::stringstream, which allows easy conversion between data types and store the coordinates as doubles. Furthermore, it is not dependent on a specific spacing between the numbers. A problem encountered in the beginning was that the reader was storing empty lines as 0s in the matrix, leading to point clouds of different sizes. So, in the current implementation, all of the empty lines are skipped.

ICPRegistration serves as a class that handles both tasks of this coursework. It contains defaults for the main parameters of ICP such that the instantiation is easier for the user. The flags used for reading from files in the initialization of PointClouds are used as inputs for the ICP solver. The default is the surface based algorithm since it is more general and it makes no assumptions regarding the point correspondences.

```
ICPRegistration(bool ICP_FLAG = SURFACE_BASED_FLAG,
               int  maxIter = 30,
               float errorThresh = 0.000001,
               float filterBadCorresp = 0.5);
```

All of the methods of ICPRegistration receive as parameters two PointCloud objects that correspond to the fixed and moving point sets. Since the PointCloud class has implemented methods such as applyTransformation() and getCenteredPCD(), this choice makes the code easier to follow and less prone to error.

In order to avoid calling the methods sequentially in a specific order to get the final matrix transform, the ICPRegistration class has the method solve(). Based on the flag set for the solver object, this method calls the appropriate algorithm for either point based or surface based registration and it outputs the RMS error and the final transformation matrix. In a more complex setting with more parameters and more methods, a Builder pattern could be used for this class.

```
void solve(PointCloud& pFixedPCD,
           PointCloud& qMovingPCD,
           double& out_err,
           Eigen::Matrix4d& out_transfMatrix,
           std::string& outputPath);
```

The Eigen library was heavily used in the implementation of these algorithms. Features like the dynamic allocation of matrices, advanced initialization (MatrixXd::Zero() or MatrixXd::Identity()), block operations

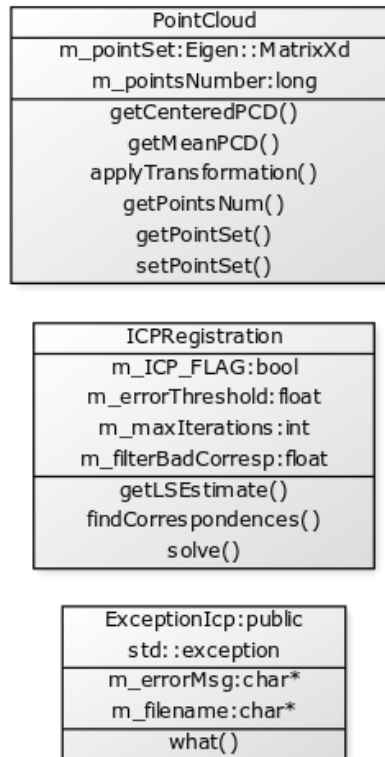


Figure 1: UML Diagram of the project class design

(easy access to specific rows or columns of blocks of values inside a matrix) make it attractive for working with big matrices of data. Plus, reductions and broadcasting were used in the computation of the least squares estimation. The Eigen solver efficiently computes the singular value decomposition, which was used for the estimation of the rotation matrix. In conclusion, the use of Eigen makes the code easier to read and provides efficient and fast computation of matrix and linear algebra operations, ideal for problems with a high number of data points.

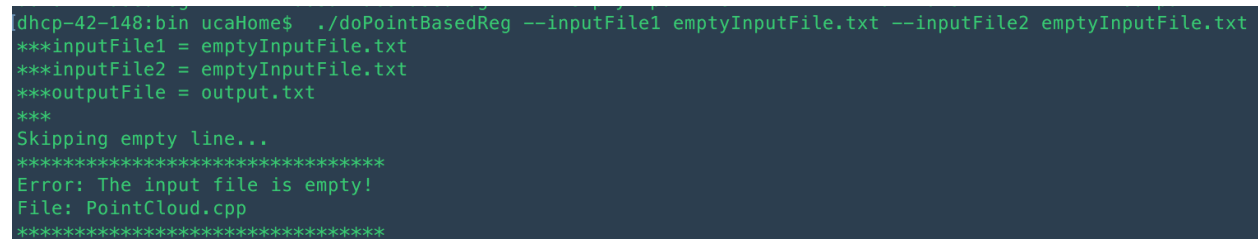
Section 2

Error checking and Testing with Catch

An exception class (ExceptionICP) was created to deal with all the errors raised during the execution of the ICP algorithm, which are then caught in the main entry point of the program. Given that the ExceptionICP class is derived from `std::exception`, the specific exceptions of the ICP program will be caught with:

```
try
{
    ...
}
catch(std::exception& err)
{
    ...
}
```

See Fig. 2 for an example of an exception thrown for an empty input file.



```
ldhcp-42-148:bin ucaHome$ ./doPointBasedReg --inputFile1 emptyInputFile.txt --inputFile2 emptyInputFile.txt
***inputFile1 = emptyInputFile.txt
***inputFile2 = emptyInputFile.txt
***outputFile = output.txt
***
Skipping empty line...
*****
Error: The input file is empty!
File: PointCloud.cpp
*****
```

Figure 2: Example of exception

Error checking:

- the input files for point based registration have to have a single point per line (x y z)
- the input files for surface based registration have to have 3 points per line (x1 y1 z1 x2 y2 z2 x3 y3 z3)
- the constructor for PointCloud cannot be called with an empty point set
- exceptions are thrown if the `std::stringstream` or the input streams cannot be opened
- the method `PointCloud::applyTransformation()` checks that the input transformation is in the format $T = [R|t]$
- the ICPRegistration solver checks if the two point clouds have the same size

The Catch library was used to implement unit testing on the ICP program (20 assertions in 6 test cases). The tests were structured in TEST_CASE and SECTIONs, using positive (REQUIRE) and negative tests (REQUIRE_THROWS). For the tests on Surface Based Registration, different input files were created with the same transformation matrix (with less points) to decrease the computation time.

Here is a list of some of the functions tested:

- Point Cloud Class
 - test whether the PointCloud class has correct methods - computes the mean correctly, it centers the point set correctly

- test whether the PointCloud class throws exceptions (negative tests) - empty point set, empty input file, invalid transformation, incorrect size of input point set
- Point Based Registration
 - test whether the PointBasedRegistration output is correct
 - test the least squares estimation
 - test whether the LS estimation throws an exception - unequal point set sizes
- Surface Based Registration
 - test whether the SurfaceBasedRegistration output is correct
 - test the findCorrespondences method - whether it throws an exception - unequal point set sizes
 - test the output of SurfaceBasedRegistration with identical matrices

finalTransf.txt				matrix4x4.txt			
0.743	-4.08e-07	-0.669	3.02e-05	0.7431	-0.0000	-0.6691	0
-0.421	0.777	-0.468	0.000121	-0.4211	0.7771	-0.4677	0
0.52	0.629	0.578	3.86e-05	0.5200	0.6293	0.5775	0
0	0	0	1	0	0	0	1

Figure 3: Point Based Registration: Left - Estimated transformation matrix; Right - Ground truth transformation matrix

finalTransf.txt				matrix4x4.txt			
0.996	-0.083	-0.0284	0.00228	0.9924	0.0793	0.0941	0
0.0803	0.993	-0.0869	-0.00289	-0.0868	0.9931	0.0793	0
0.0354	0.0843	0.996	0.00158	-0.0872	-0.0868	0.9924	0
0	0	0	1	0	0	0	1

Figure 4: Surface Based Registration: Left - Estimated transformation matrix; Right - Ground truth transformation matrix

The difference between the transformation matrices in the surface based case is bigger than in the least squares estimation. One solution would be to filter out the bad correspondences between the point sets. This could be done by thresholding the distance between them, or comparing their normals.

Most of the test cases implemented use fixtures to test if the estimated output is close enough to the ground truth. Moreover, they use a setup/teardown approach in which an object is created in the main TEST_CASE and different scenarios are tested in individual SECTIONS. The type of testing done in this project is closer to Behaviour Driven Development. Each test was created to check for errors that might be caused from the end-user perspective.

Section 3

Connect all the components together - Point Based Registration

The Boost library was used to create entry points for the Point Based and Surface Based Registration. Their robust implementation of command line parsing allows specifying default and implicit values, positional and optional arguments and descriptions for the different flags.

The argument `-e/ --example` was added to illustrate a call to the program. The files provided in the testing folders are used as input. The output is written to a text file by default.

```
[dhcp-42-148:bin ucaHome$ ./doPointBasedReg -h
PointBasedRegistration Options:
  -h [ --help ]           Help message
  -e [ --example ]        Show an example
  -v [ --version ]        Program version
  --inputFile1 arg         Input file with fixed 3D points
  --inputFile2 arg         Input file with moving 3D points
  --outputFile [=arg(=output.txt)] (=output.txt)
                           Output file for the final
                           transformation matrix

***outputFile = output.txt
[dhcp-42-148:bin ucaHome$ ./doPointBasedReg -e
*** Example of a program call:
./doPointBasedReg --inputFile1 ../../../../research-computing-with-cpp-demo/Testing/PointBasedRegistrationData/fixed.txt --inputFile2 ../../../../research-computing-with-cpp-demo/Testing/PointBasedRegistrationData/moving.txt --outputFile finalTransf.txt
```

Figure 5: Point Based Registration - Command line entry point and example of a call

Section 4

Connect all the components together - Surface Based Registration

The Surface Based Registration entry point is structured in a similar way to Point Based Registration. The example calls the function with the text files in the testing folder.

```
[dhcp-42-148:bin ucaHome$ ./doSurfaceBasedReg -h
SurfaceBasedRegistration Options:
  -h [ --help ]           Help message
  -e [ --example ]        Show an example
  -v [ --version ]        Program version
  --inputFile1 arg         Input file with fixed 3D points
  --inputFile2 arg         Input file with moving 3D points
  --outputFile [=arg(=output.txt)] (=output.txt)
                           Output file for the final
                           transformation matrix

***outputFile = output.txt
[dhcp-42-148:bin ucaHome$ ./doSurfaceBasedReg -e
*** Example of a program call:
./doSurfaceBasedReg --inputFile1 ../../../../research-computing-with-cpp-demo/Testing/SurfaceBasedRegistrationData/fran_cut.txt --inputFile2 ../../../../research-computing-with-cpp-demo/Testing/SurfaceBasedRegistrationData/fran_cut_transformed.txt --outputFile finalTransf.txt
***outputFile = output.txt]
```

Figure 6: Surface Based Registration - Command line entry point and example of a call

Section 5

Medical Imaging application

The slowest part of the current algorithm is the correspondence step. In this implementation, a naive approach is taken where for each point, the closest point is found by computing all the distances between the two point sets. This step could be significantly improved in order to speed up the process. One approach would be to store the data points in a kd-tree. This way, the search of the pair would be much faster. Libraries like ANN or FASTANN implement approximate nearest neighbour algorithms based on kd-trees efficiently. With this approach, more neighbouring points could be found. The pair could be selected based on additional criteria other than the distance, like the orientation of the normals or the colour/texture of the mesh.