```cpp
//
// Created by Vivek Chari on 10/11/23.
//
// This file contains methods and classes that aid in point set registration
// Both classes take two set of points,  X and Y, and compute a transformation
// Y = RX + t.
// There are two methods exposed, Coherent Point Drift and Procrustes (Arun's method).
// There is also a draft of a function to compute distortion correction.

#ifndef PROGRAMMING_ASSIGMENT_ONE_REGISTER_H
#define PROGRAMMING_ASSIGMENT_ONE_REGISTER_H

#include <Eigen/Dense>
#include <Eigen/SVD>

namespace Registration {

uint64_t COHERENTPOINTDRIFT = 0;
uint64_t PROCRUSTES = 1;

template<typename T>
Eigen::Matrix<T, -1, 3> compute_distortion_correction(Eigen::Matrix<T, -1, 3> X, Eigen::Matrix<T, -1, 3> Y, T beta = 1, T lambda = 0.01) {
  Eigen::Matrix<T, -1, -1> G(X.rows(), X.rows());
  if (X.rows() != Y.rows())
    throw std::invalid_argument("X and Y must have the same number of rows.");
  for (size_t i = 0; i < X.rows(); i++) {
    for (size_t j = i; j < X.rows(); j++) {
      T val = -1.0 / (2 * std::pow(beta, 2)) * (X.row(i) - X.row(j)).squaredNorm();
      G(i, j) = G(j, i) = val;
```

```
30      }
31    }
32    Eigen::Matrix<T, -1, -1> regularized_G = (lambda
     > 0) ? G + Eigen::Matrix<T, -1, -1>::Identity(X.
    rows(), X.rows()) * lambda : G;
33    Eigen::Matrix<T, -1, -1> sol = regularized_G.
    bdcSvd(Eigen::ComputeThinU | Eigen::ComputeThinV).
    solve(Y - X);
34    return G * sol;
35  }
36
37
38  template<typename T>
39  class CoherentPointDrift {
40      long D = 3; //dimension
41      T w = 0.0;
42      Eigen::Matrix<T, -1, 3, Eigen::RowMajor> M;
43      Eigen::Matrix<T, -1, 3, Eigen::RowMajor> N;
44      Eigen::Matrix<T, -1, -1, Eigen::ColMajor> prob;
45      Eigen::JacobiSVD<Eigen::Matrix<T, -1, -1, Eigen
    ::RowMajor>> EIGENSVD;
46      T sigma_square = 0;
47      T tol;
48      long iter;
49      double size_N, size_M; //number of points
50      void expectation();
51      void maximization();
52  public:
53      Eigen::Matrix<T, 3, 3> B; //rotation
54      Eigen::Vector<T, 3> t; //translation
55      CoherentPointDrift(Eigen::Matrix<T, -1, 3,
    Eigen::RowMajor> M,
56                         Eigen::Matrix<T, -1, 3,
    Eigen::RowMajor> N,
57                         long niter = 250, T term_tol
     = 0.001);
58
59      void register_points_cpd();
```

```
60 };
61
62 template<typename T>
63 void CoherentPointDrift<T>::register_points_cpd() {
64   for (int i = 0; i < iter; i++) {
65     auto v = sigma_square;
66     expectation();
67     maximization();
68     if (abs(v - sigma_square) < tol) {
69       break;
70     }
71   }
72 }
73
74 template<typename T>
75 void CoherentPointDrift<T>::maximization() {
76   //follows the paper very closely (modulo the use
   of fast gauss transforms)
77   auto N_p = prob.sum();
78   auto colwise_prob = (prob.colwise().sum()).
   transpose(); //M
79   auto rowwise_prob = prob.rowwise().sum() * (1.0f
    / N_p); //N
80   auto mu_m_hat = (M.transpose() * colwise_prob * (
   1.0f / N_p));
81   auto mu_n_hat = (N.transpose() * rowwise_prob);
82   auto Mhat = M.rowwise() - mu_m_hat.transpose();
83   auto Nhat = N.rowwise() - mu_n_hat.transpose();
84
85   auto A = Mhat.transpose() * prob.transpose() *
   Nhat;
86   EIGENSVD.compute(A, Eigen::ComputeFullU | Eigen::
   ComputeFullV);
87   auto U = EIGENSVD.matrixU();
88   auto V_T = EIGENSVD.matrixV().transpose();
89   Eigen::Matrix<T, 3, 3, Eigen::RowMajor> C = Eigen
   ::Matrix<T, 3, 3, Eigen::RowMajor>::Identity();
90   C(2, 2) = (U * V_T).determinant();
```

```
 91   B = U * C * V_T;
 92
 93   t = mu_m_hat - B * mu_n_hat;
 94   sigma_square = (1.0f / (N_p * D)) *
 95                  ((Mhat.transpose() * colwise_prob
    .asDiagonal() * Mhat).trace() - (A.transpose() * B
    ).trace());
 96 }
 97
 98
 99 template<typename T>
100 void CoherentPointDrift<T>::expectation() {
101   T inv_sigma_square = -0.5 / sigma_square;
102   for (int m = 0; m < size_M; m++) {
103     T denom = (size_M / size_N) * (w / (1 - w));
    //* std::pow(2.0 * PI * sigma_square, D/2.0);
104     const auto x = M.row(m);
105     for (int n = 0; n < size_N; n++) {
106       const auto y = N.row(n);
107       T probmass = exp(inv_sigma_square * (x.
    transpose() - (B * y.transpose() + t)).squaredNorm
    ());
108       prob(n, m) = probmass;
109       denom += probmass;
110     }
111     prob.col(m) /= denom;
112   }
113 }
114
115 template<typename T>
116 CoherentPointDrift<T>::CoherentPointDrift(Eigen::
    Matrix<T, -1, 3, Eigen::RowMajor> M_arg,
117                                           Eigen::
    Matrix<T, -1, 3, Eigen::RowMajor> N_arg,
118                                           long
    niter,
119                                           T
    term_tol) {
```

```
120   //see paper alogorithm. there is an almost one-
    to-one correspondance in how this is implemented.
121   M = M_arg; N = N_arg;
122   tol = term_tol; iter = niter;
123   size_M = M.rows(); size_N = N.rows();
124   for (int n = 0; n < size_N; n++)
125     for (int m = 0; m < size_M; m++)
126       sigma_square += (N.row(n) - M.row(m)).
    squaredNorm();
127   sigma_square /= (T) (D * size_M * size_N);
128
129   prob = Eigen::Matrix<T, -1, -1, Eigen::ColMajor
    >::Zero(size_N, size_M);
130   B = Eigen::Matrix<T, 3, 3>::Identity();
131   t = Eigen::Vector<T, 3>::Zero();
132 }
133
134 template<typename T>
135 class Procrustes {
136     Eigen::JacobiSVD<Eigen::Matrix<T, 3, 3>>
    EIGENSVD;
137 public:
138     Eigen::Matrix<T, 3, 3> B;
139     Eigen::Vector<T, 3> t;
140
141     Procrustes(Eigen::Matrix<T, -1, 3> M, Eigen::
    Matrix<T, -1, 3> N);
142 };
143
144 template<typename T>
145 Procrustes<T>::Procrustes(const Eigen::Matrix<T, -
    1, 3> M,
146                           const Eigen::Matrix<T, -
    1, 3> N) {
147   // N = FM = RM + t
148   if (M.rows() != N.rows())
149     throw std::invalid_argument("M and N must have
     the same number of rows.");
```

```
150   auto num_points = M.rows();
151
152   // compute the centroid of both point sets.
153   auto center_M = M.colwise().sum() / num_points;
154   auto center_N = N.colwise().sum() / num_points;
155
156   // centering the point sets so that centroid
    lies at origin.
157   auto centered_M = M.rowwise() - center_M;
158   auto centered_N = N.rowwise() - center_N;
159
160   //Arun's method for point set registration
161   EIGENSVD.compute(centered_M.transpose() *
    centered_N, Eigen::ComputeFullU | Eigen::
    ComputeFullV);
162   auto U_T = EIGENSVD.matrixU().transpose();
163   auto V = EIGENSVD.matrixV();
164   Eigen::Matrix<T, 3, 3> C = Eigen::Matrix<T, 3, 3
    >::Identity();
165   C(2, 2) = (U_T * V).determinant();
166   // N = MB + t
167   B = V * C * U_T;
168   t = center_N - center_M * B.transpose();
169 }
170
171 }
172
173 #endif //PROGRAMMING_ASSIGMENT_ONE_REGISTER_H
174
```

```
 1 //
 2 // Created by Vivek Chari on 10/10/23.
 3 //
 4 // This file contains methods and classes that
   contruct "frame graphs"
 5 // that describe the relationship between different
    frames. It contains
 6 // a class FrameTransformation that can be used to
   store frame transformations
 7 // in a packed 4x4 matrix, along with methods to
   apply the transform to vectors
 8 // and matrices, as well as methods to compute (and
    apply) the inverse transform.
 9 // The FrameGraph class provides an interface to
   easily compute and apply
10 // transforms between frames, and has an internal
   BFS that computes the shortest path
11 // between any frames in the graph.
12
13 #ifndef PROGRAMMING_ASSIGMENT_ONE_FRAME_LIB_H
14 #define PROGRAMMING_ASSIGMENT_ONE_FRAME_LIB_H
15
16 #include <map>
17 #include <set>
18 #include <list>
19 #include <iostream>
20 #include "register.h"
21 #include "Eigen/Dense"
22 #include "Eigen/StdVector"
23
24 // class to house frame, inverse frame, and frame
   application methods.
25 template<typename T>
26 class FrameTransformation {
27     Eigen::Matrix<T, 4, 4> frame;
28     Eigen::Matrix<T, 4, 4> iframe;
29 public:
30     FrameTransformation(Eigen::Matrix<T, 3, 3> rot
```

```
30 , Eigen::Vector<T, 3> pos);
31     void compute_inverse();
32     Eigen::Vector<T,3> apply_forward_transform(
   Eigen::Vector<T,3> vec);
33     Eigen::Vector<T,3> apply_inverse_transform(
   Eigen::Vector<T,3> vec);
34     Eigen::Matrix<T,-1, 3> apply_forward_transform(
   Eigen::Matrix<T,-1, 3> mat);
35     Eigen::Matrix<T,-1, 3> apply_inverse_transform(
   Eigen::Matrix<T,-1, 3> packed_pts);
36     Eigen::Matrix<T, 4, 4> get_frame() {return
   frame;};
37     Eigen::Matrix<T, 4, 4> get_iframe() {return
   iframe;};
38 };
39
40 // this is not very efficient. it will be
   refactored into a single packed_pts multiply later
   on
41 template<typename T>
42 Eigen::Matrix<T,-1, 3> FrameTransformation<T>::
   apply_inverse_transform(const Eigen::Matrix<T, -1,
   3> packed_pts) {
43   Eigen::Matrix<T, -1, 3> tmp = Eigen::Matrix<T, -1
   , 3>::Ones(packed_pts.rows(), 3);
44   //for every point in packed_pts, compute the
   inverse transform and store it
45   for(int i = 0; i < packed_pts.rows(); i++) {
46     const auto vec = packed_pts.row(i);
47     Eigen::Vector<T, 4> tmpvec =  {vec[0], vec[1],
   vec[2], 1};
48     tmp.row(i) = (iframe * tmpvec).template head<3
   >();
49   }
50   return tmp;
51 }
52
53 // this is not very efficient. it will be
```

```
53 refactored into a single mat multiply later on
54 template<typename T>
55 Eigen::Matrix<T,-1, 3> FrameTransformation<T>::
   apply_forward_transform(const Eigen::Matrix<T, -1,
   3> mat) {
56   Eigen::Matrix<T, -1, 3> tmp = Eigen::Matrix<T, -1
   , 3>::Ones(mat.rows(), 3);
57   //for every point in packed_pts, compute the
   transform and store it
58   for(int i = 0; i < mat.rows(); i++) {
59     const auto vec = mat.row(i);
60     Eigen::Vector<T, 4> tmpvec =  {vec[0], vec[1],
   vec[2], 1};
61     tmp.row(i) = (frame * tmpvec).template head<3
   >();
62   }
63   return tmp;
64 }
65
66 template<typename T>
67 Eigen::Vector<T, 3> FrameTransformation<T>::
   apply_inverse_transform(const Eigen::Vector<T, 3>
   vec) {
68   Eigen::Vector<T, 4> tmp {vec[0], vec[1], vec[2],
   1};
69   return (iframe * tmp).template head<3>();
70 }
71
72 template<typename T>
73 Eigen::Vector<T, 3> FrameTransformation<T>::
   apply_forward_transform(const Eigen::Vector<T, 3>
   vec) {
74   Eigen::Vector<T, 4> tmp {vec[0], vec[1], vec[2],
   1};
75   return (frame * tmp).template head<3>();
76 }
77
78 template<typename T>
```

```
 79 void FrameTransformation<T>::compute_inverse() {
 80   //the 4x4 matrix "frame" stores the transform
 81   //transpose the top left 3x3 block (recall that
    R^{-1} = R^T)
 82   iframe = frame.transpose();
 83   //compute t^{-1} = -R^{-1}t
 84   iframe.template block<3,1>(0, 3) =
 85           -1 * iframe.template block<3,3>(0, 0) *
    iframe.template block<1, 3>(3, 0).transpose();
 86   //make sure to zero the bottom row of the packed
     transform. it is now [0,0,0,1]
 87   iframe.template block<1, 3>(3, 0).setZero();
 88 }
 89
 90
 91 template<typename T>
 92 FrameTransformation<T>::FrameTransformation(Eigen
    ::Matrix<T, 3, 3> rot, Eigen::Vector<T, 3> pos) {
 93   frame.setIdentity();
 94   //store the frame transform packed in a 4x4
    matrix. this just sets the appropriate blocks in
    the matrix.
 95   frame.template block<3, 3>(0, 0)= rot;
 96   frame.template block<3, 1>(0, 3) = pos;
 97   compute_inverse();
 98 }
 99
100 //helper method to test implementations. this
    generates a random 3x3 orthogonal matrix.
101 template<typename T>
102 Eigen::Matrix<T,3, 3> gen_random_orthogonal(){
103   return Eigen::Quaternion<T>::UnitRandom().
    toRotationMatrix();
104 }
105
106 //class to hold a graph of frame transforms, and
    call appropriate ones when needed.
107 template<typename T>
```

```
108 class FrameGraph {
109     //edge class to store information on a
    transform.
110     struct edge {
111         std::string dest;
112         FrameTransformation<T> transform;
113         //we need to know whether this edge is an
    inverse, because the same transform object
    representes both
114         // A->B and B->A
115         bool is_inverse = false;
116     };
117     std::map<std::string, std::vector<edge>> graph
    ;
118     uint64_t registration_method = -1;
119     //helper vars for BFS.
120     std::map<std::string, std::pair<std::string,
    uint64_t>> prev_and_dist;
121     std::set<std::string> set;
122   public:
123       explicit FrameGraph(uint64_t reg_method) {
    registration_method = reg_method; }
124       void _register_transform_procrustes(const
    std::string& frame_A_name, Eigen::Matrix<T, -1, 3
    > points_in_A, const std::string& frame_B_name,
    Eigen::Matrix<T, -1, 3> points_in_B);
125       void _register_transform_cpd(const std::
    string& frame_A_name, Eigen::Matrix<T, -1, 3>
    points_in_A, const std::string& frame_B_name,
    Eigen::Matrix<T, -1, 3> points_in_B);
126       void register_transform(std::string
    frame_A_name, Eigen::Matrix<T, -1, 3> points_in_A
    , std::string frame_B_name, Eigen::Matrix<T, -1, 3
    > points_in_B);
127       //returns points in B
128       Eigen::Matrix<T, -1, 3>
    apply_direct_transform(const std::string&
    frame_A_name, const std::string& frame_B_name,
```

```
128 Eigen::Matrix<T, -1, 3> points_in_A);
129       Eigen::Matrix<T, -1, 3> apply_transform(
    const std::string& frame_A_name, const std::string
    & frame_B_name, Eigen::Matrix<T, -1, 3>
    points_in_A);
130       //probably RVO-d out.
131       std::vector<Eigen::Matrix<T, 4, 4>, Eigen::
    aligned_allocator<Eigen::Matrix<T, 4, 4>>>
    retrieve_transform_matrices(std::vector<std::pair<
    std::string, std::string>> &transforms);
132       void clear() {graph.clear(); set.clear();
    prev_and_dist.clear();}
133 };
134
135
136 //given a vector of edges (A,B),(C, D),....
    retrieve the packed transform matrices for each of
     these edges, respecting
137 // whether the edge represents a forward or
    inverse transform. Mainly for use in pivot
    calibration.
138 template<typename T>
139 std::vector<Eigen::Matrix<T, 4, 4>, Eigen::
    aligned_allocator<Eigen::Matrix<T, 4, 4>>>
140 FrameGraph<T>::retrieve_transform_matrices(std::
    vector<std::pair<std::string, std::string>> &
    transforms) {
141   std::vector<Eigen::Matrix<T, 4, 4>, Eigen::
    aligned_allocator<Eigen::Matrix<T, 4, 4>>> res(
    transforms.size());
142   for(size_t i = 0; i < transforms.size(); i++){
143     for(edge e : graph[transforms[i].first]) {
144       if (e.dest == transforms[i].second) {
145         res[i] = (e.is_inverse) ? e.transform.
    get_iframe() : e.transform.get_frame();
146         break;
147       }
148     }
```

```
149   }
150   return res;
151 }
152
153 //given points in frame "A" and a frame "B"
    compute the transform "B from A". The points
    returned will now be in the
154 // coordinate frame "B". This methods traverses
    the internal graph using Djikstra's method to find
     the shortest
155 //path connecting the two frames, and then
    computes the appropriate transform by chaining
    transforms along
156 // traversed edges.
157 template<typename T>
158 Eigen::Matrix<T, -1, 3> FrameGraph<T>::
    apply_transform(const std::string& frame_A_name,
    const std::string& frame_B_name,
159
         Eigen::Matrix<T, -1, 3> points_in_A) {
160   //helper vars for Djikstra's
161   prev_and_dist.clear();
162   set.clear();
163   for(auto a : graph) {
164     prev_and_dist[a.first] = std::pair<std::string
    , uint64_t>("", INFINITY);
165     set.insert(a.first);
166   }
167   prev_and_dist[frame_A_name] = std::pair<std::
    string, uint64_t>("", 0);
168   //do Djikstra BFS. Not the most efficient
    implementation, but it works.
169   //TODO: Refactor with focus on speed
170   while (!set.empty()) {
171       std::string best; T dist = INFINITY;
172       for(const auto& node : set) {
173         auto vals = prev_and_dist[node];
174         if (vals.second < dist) {
```

```
175           best = node;
176           dist = vals.second;
177         }
178       }
179       set.erase(best);
180       for(edge neighbor : graph[best]){
181         if(set.contains(neighbor.dest)){
182           auto alt = prev_and_dist[best].second +
    1;
183           if (alt < prev_and_dist[neighbor.dest].
    second){
184             prev_and_dist[neighbor.dest].second =
    alt;
185             prev_and_dist[neighbor.dest].first =
    best;
186           }
187         }
188         if(neighbor.dest == frame_B_name) {
189           set.clear();
190           break;
191         }
192       }
193   }
194   //reconstruct the shortest path taken
195   std::list<std::string> transform_chain;
196   auto target = frame_B_name;
197   while(target != frame_A_name) {
198     transform_chain.push_front(target);
199     target = prev_and_dist[target].first;
200   }
201   transform_chain.push_front(frame_A_name);
202   Eigen::Matrix<T, -1, 3> res = points_in_A;
203   size_t sz = transform_chain.size();
204   //traverse the shortest path, applying
    transforms as we go along the path.
205   for(size_t i = 0; i < sz; i++) {
206     auto cur = transform_chain.front();
207     transform_chain.pop_front();
```

```
208     for (edge e : graph[cur])
209       if(e.dest == transform_chain.front()) {
210         res = (e.is_inverse) ? e.transform.
    apply_inverse_transform(res) :
211               e.transform.apply_forward_transform(
    res);
212         break;
213       }
214   }
215   //points are now in frame B.
216   return res;
217 }
218
219 //given two frames, and corresponding point sets,
    compute a transform and register it in the graph.
220 //call into different registration subroutines
    depending on user specification.
221 template<typename T>
222 void FrameGraph<T>::register_transform(std::string
     frame_A_name, Eigen::Matrix<T, -1, 3> points_in_A
    ,
223                                        std::string
     frame_B_name, Eigen::Matrix<T, -1, 3> points_in_B
    ) {
224   if(registration_method == Registration::
    PROCRUSTES)
225     return _register_transform_procrustes(
    frame_A_name, points_in_A, frame_B_name,
    points_in_B);
226   if (registration_method == Registration::
    COHERENTPOINTDRIFT)
227     return _register_transform_cpd(frame_A_name,
    points_in_A, frame_B_name, points_in_B);
228   throw std::invalid_argument("Registration Method
     invalid.");
229 }
230
231
```

```
232 //given points in frame "A" and a frame "B"
    compute the tranform "B from A". The points
    returned will now be in the
233 // coordinate frame "B".  This method assumes a
    direct connection between A and B
234 template<typename T>
235 Eigen::Matrix<T, -1, 3> FrameGraph<T>::
    apply_direct_transform(const std::string&
    frame_A_name, const std::string& frame_B_name,
236
                Eigen::Matrix<T, -1, 3> points_in_A) {
237   if(!graph.contains(frame_A_name) || !graph.
    contains(frame_B_name))
238     throw std::invalid_argument("either frame A or
     frame B doesn't exist");
239   const auto A_edges = graph[frame_A_name];
240   for(edge edge: A_edges)
241     if(edge.dest == frame_B_name)
242       return (edge.is_inverse) ? edge.transform.
    apply_inverse_transform(points_in_A) :
243       edge.transform.apply_forward_transform(
    points_in_A);
244   throw std::invalid_argument("Frame A and B don't
     have a direct edge");
245 }
246
247 template<typename T>
248 void FrameGraph<T>::_register_transform_procrustes
    (const std::string& frame_A_name, Eigen::Matrix<T
    , -1, 3> points_in_A,
249
     const std::string& frame_B_name,
250
     Eigen::Matrix<T, -1, 3> points_in_B) {
251   Registration::Procrustes<T> registration(
    points_in_A, points_in_B);
252   if(!graph.contains(frame_A_name)) {
253     graph[frame_A_name] = std::vector<edge>();
```

```
254   } else {
255     for(edge edge : graph[frame_A_name])
256       if(edge.dest == frame_B_name)
257         throw std::invalid_argument("Transform A
    <->B already exists.");
258   }
259   if(!graph.contains(frame_B_name))
260     graph[frame_B_name] =  std::vector<edge>();
261
262   FrameTransformation<T> t(registration.B,
    registration.t);
263   graph.at(frame_A_name).emplace_back(frame_B_name
    , t, false);
264   graph.at(frame_B_name).emplace_back(frame_A_name
    , t, true);
265 }
266
267 template<typename T>
268 void FrameGraph<T>::_register_transform_cpd(const
    std::string& frame_A_name,
269                                             Eigen
    ::Matrix<T, -1, 3> points_in_A,
270                                             const
    std::string& frame_B_name,
271                                             Eigen
    ::Matrix<T, -1, 3> points_in_B) {
272   Registration::CoherentPointDrift<T> registration
    (points_in_A, points_in_B);
273   if(!graph.contains(frame_A_name))
274     graph[frame_A_name] =  std::vector<edge>();
275   if(!graph.contains(frame_B_name))
276     graph[frame_B_name] =  std::vector<edge>();
277
278   FrameTransformation<T> t(registration.B,
    registration.t);
279   graph.at(frame_A_name).emplace_back(frame_B_name
    , t, false);
280   graph.at(frame_B_name).emplace_back(frame_A_name
```

```
280 , t, true);
281 }
282
283
284
285
286 #endif //PROGRAMMING_ASSIGMENT_ONE_FRAME_LIB_H
287
```

```cpp
//
// Created by Vivek Chari on 10/12/23.
//
// The following file contains methods for performing pivot calibration,
// given a vector of point measurements.

#ifndef PROGRAMMING_ASSIGMENT_ONE_CALIBRATION_H
#define PROGRAMMING_ASSIGMENT_ONE_CALIBRATION_H

#include "Eigen/Dense"
#include "Eigen/StdVector"
#include "frame_lib.h"

template<typename T>
Eigen::Vector<T, 6> solve_least_squares(const std::vector<Eigen::Matrix<T, 4, 4>,
        Eigen::aligned_allocator<Eigen::Matrix<T, 4, 4>>> &vec) {
  Eigen::Matrix<T, -1, 6> A = Eigen::Matrix<T, -1, 6>(vec.size() * 3,6);
  Eigen::Vector<T, -1> b = Eigen::Vector<T, -1>(vec.size() * 3);
  //create a new matrix that looks like
  //   R_1 | -I                              -d_1
  // [ R_2 | -I ]  = A     and a vector [  -d_2 ] = b
  //   R_3 | -I                              -d_3
  //      ...                                 ...
  //where R are the rotation component of the transform, and b are the translational component of the transform.
  for(size_t i = 0; i < vec.size(); i++){
    A.template block<3, 3>(3 * i, 0) = vec[i].template topLeftCorner<3,3>();
    A.template block<3, 3>(3 * i, 3) = -1 * Eigen::Matrix<T, 3, 3>::Identity();
    b.template block<3, 1>(3 * i, 0) = -1 * vec[i].
```

```
28 template topRightCorner<3,1>();
29   }
30   //now we have an equation Ax = b, where x
   contains our variables of interest (pivot location
   , tip location).
31   // solve the overdetermined system in the least
   squares sense.
32   return A.bdcSvd(Eigen::ComputeFullU | Eigen::
   ComputeFullV).solve(b);
33 }
34
35 // the pivot calibration routine takes a vector of
   points (size n_frames) and computes the location of
    the tip and
36 // pivot dimple using a lstsquares sol.
37 template<typename T>
38 std::pair<Eigen::Vector3<T>, Eigen::Vector3<T>>
   pivot_calibration_routine(const std::vector<Eigen::
   Matrix<T, -1, 3>,
39         Eigen::aligned_allocator<Eigen::Matrix<T, -
   1, 3>>> &obs){
40   FrameGraph<T> graph(Registration::PROCRUSTES);
41   std::vector<Eigen::Matrix<T, -1, 3>, Eigen::
   aligned_allocator<Eigen::Matrix<T, -1, 3>>>
42           centered_obs = std::vector<Eigen::Matrix<
   T, -1, 3>, Eigen::aligned_allocator<Eigen::Matrix<T
   , -1, 3>>>(obs.size());
43   //compute a reference origin based on frame zero
   of the data by centering frame zero observations
   around the origin.
44   Eigen::Matrix<T, -1, 3> g = obs[0].rowwise() -
   obs[0].colwise().mean(); //row vector
45
46   std::vector<std::pair<std::string, std::string>>
   frame_pairs(obs.size());
47   for(size_t i = 0; i < obs.size(); i++) {
48     //register a transform from our constructed "
   local frame", and other frames of data.
```

```
49      graph.register_transform("POINTER_LOCAL_FRAME"
    , g,
50                               "TRACKER_" + std::
    to_string(i), obs[i]);
51      //record what edges were added, so we can
    retrieve the transform matrices later.
52      frame_pairs[i] = std::pair<std::string, std::
    string> ("POINTER_LOCAL_FRAME", "TRACKER_" + std::
    to_string(i));
53    }
54    //solves the least squares problem given by the
    set of transforms, assuming that
55    // F_1 * g = F_2 * g = ... = F_k * g
56    auto sol = solve_least_squares( graph.
    retrieve_transform_matrices(frame_pairs));
57    //first 3 coeffs of vector are the location of
    the tip of the pointer in local pointer frame.
58    return std::pair<Eigen::Vector3<T>, Eigen::
    Vector3<T>>(sol.template head<3>(), sol.template
    tail<3>());
59  }
60
61  #endif //PROGRAMMING_ASSIGMENT_ONE_CALIBRATION_H
62
```