```cpp
#include <cstdio>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include <Eigen/Core>
#include <Eigen/Dense>

#include "common.h"
#include "rotation.h"
#include "random.h"

typedef Eigen::Map<Eigen::VectorXd> VectorRef;
typedef Eigen::Map<const Eigen::VectorXd> ConstVectorRef;

template<typename T>
void FscanfOrDie(FILE *fptr, const char *format, T *value) {
    int num_scanned = fscanf(fptr, format, value);
    if (num_scanned != 1)
        std::cerr << "Invalid UW data file. ";
}

void PerturbPoint3(const double sigma, double *point) {
    for (int i = 0; i < 3; ++i)
        point[i] += RandNormal() * sigma;
}

double Median(std::vector<double> *data) {
    int n = data->size();
    std::vector<double>::iterator mid_point = data->begin() + n / 2;
    std::nth_element(data->begin(), mid_point, data->end());
    return *mid_point;
}

BALProblem::BALProblem(const std::string &filename, bool use_quaternions) {
    FILE *fptr = fopen(filename.c_str(), "r");

    if (fptr == NULL) {
        std::cerr << "Error: unable to open file " << filename;
        return;
    };

    // This wil die horribly on invalid files. Them's the breaks.
    FscanfOrDie(fptr, "%d", &num_cameras_);
    FscanfOrDie(fptr, "%d", &num_points_);
    FscanfOrDie(fptr, "%d", &num_observations_);

    std::cout << "Header: " << num_cameras_
              << " " << num_points_
              << " " << num_observations_;

    point_index_ = new int[num_observations_];
    camera_index_ = new int[num_observations_];
    observations_ = new double[2 * num_observations_];

    num_parameters_ = 9 * num_cameras_ + 3 * num_points_;
    parameters_ = new double[num_parameters_];

    for (int i = 0; i < num_observations_; ++i) {
        FscanfOrDie(fptr, "%d", camera_index_ + i);
        FscanfOrDie(fptr, "%d", point_index_ + i);
        for (int j = 0; j < 2; ++j) {
            FscanfOrDie(fptr, "%lf", observations_ + 2 * i + j);
        }
    }

    for (int i = 0; i < num_parameters_; ++i) {
        FscanfOrDie(fptr, "%lf", parameters_ + i);
    }
```

```
 70
 71      fclose(fptr);
 72
 73      use_quaternions_ = use_quaternions;
 74      if (use_quaternions) {
 75          // Switch the angle-axis rotations to quaternions.
 76          num_parameters_ = 10 * num_cameras_ + 3 * num_points_;
 77          double *quaternion_parameters = new double[num_parameters_];
 78          double *original_cursor = parameters_;
 79          double *quaternion_cursor = quaternion_parameters;
 80          for (int i = 0; i < num_cameras_; ++i) {
 81              AngleAxisToQuaternion(original_cursor, quaternion_cursor);
 82              quaternion_cursor += 4;
 83              original_cursor += 3;
 84              for (int j = 4; j < 10; ++j) {
 85                  *quaternion_cursor++ = *original_cursor++;
 86              }
 87          }
 88          // Copy the rest of the points.
 89          for (int i = 0; i < 3 * num_points_; ++i) {
 90              *quaternion_cursor++ = *original_cursor++;
 91          }
 92          // Swap in the quaternion parameters.
 93          delete[]parameters_;
 94          parameters_ = quaternion_parameters;
 95      }
 96  }
 97
 98  void BALProblem::WriteToFile(const std::string &filename) const {
 99      FILE *fptr = fopen(filename.c_str(), "w");
100
101      if (fptr == NULL) {
102          std::cerr << "Error: unable to open file " << filename;
103          return;
104      }
105
106      fprintf(fptr, "%d %d %d %d\n", num_cameras_, num_cameras_, num_points_, num_obser
    vations_);
107
108      for (int i = 0; i < num_observations_; ++i) {
109          fprintf(fptr, "%d %d", camera_index_[i], point_index_[i]);
110          for (int j = 0; j < 2; ++j) {
111              fprintf(fptr, " %g", observations_[2 * i + j]);
112          }
113          fprintf(fptr, "\n");
114      }
115
116      for (int i = 0; i < num_cameras(); ++i) {
117          double angleaxis[9];
118          if (use_quaternions_) {
119              //OutPut in angle-axis format.
120              QuaternionToAngleAxis(parameters_ + 10 * i, angleaxis);
121              memcpy(angleaxis + 3, parameters_ + 10 * i + 4, 6 * sizeof(double));
122          } else {
123              memcpy(angleaxis, parameters_ + 9 * i, 9 * sizeof(double));
124          }
125          for (int j = 0; j < 9; ++j) {
126              fprintf(fptr, "%.16g\n", angleaxis[j]);
127          }
128      }
129
130      const double *points = parameters_ + camera_block_size() * num_cameras_;
131      for (int i = 0; i < num_points(); ++i) {
132          const double *point = points + i * point_block_size();
133          for (int j = 0; j < point_block_size(); ++j) {
134              fprintf(fptr, "%.16g\n", point[j]);
135          }
136      }
137
```

```
138      fclose(fptr);
139  }
140
141  // Write the problem to a PLY file for inspection in Meshlab or CloudCompare
142  void BALProblem::WriteToPLYFile(const std::string &filename) const {
143      std::ofstream of(filename.c_str());
144
145      of << "ply"
146         << '\n' << "format ascii 1.0"
147         << '\n' << "element vertex " << num_cameras_ + num_points_
148         << '\n' << "property float x"
149         << '\n' << "property float y"
150         << '\n' << "property float z"
151         << '\n' << "property uchar red"
152         << '\n' << "property uchar green"
153         << '\n' << "property uchar blue"
154         << '\n' << "end_header" << std::endl;
155
156      // Export extrinsic data (i.e. camera centers) as green points.
157      double angle_axis[3];
158      double center[3];
159      for (int i = 0; i < num_cameras(); ++i) {
160          const double *camera = cameras() + camera_block_size() * i;
161          CameraToAngelAxisAndCenter(camera, angle_axis, center);
162          of << center[0] << ' ' << center[1] << ' ' << center[2]
163             << "0 255 0" << '\n';
164      }
165
166      // Export the structure (i.e. 3D Points) as white points.
167      const double *points = parameters_ + camera_block_size() * num_cameras_;
168      for (int i = 0; i < num_points(); ++i) {
169          const double *point = points + i * point_block_size();
170          for (int j = 0; j < point_block_size(); ++j) {
171              of << point[j] << ' ';
172          }
173          of << "255 255 255\n";
174      }
175      of.close();
176  }
177
178  void BALProblem::CameraToAngelAxisAndCenter(const double *camera,
179                                              double *angle_axis,
180                                              double *center) const {
181      VectorRef angle_axis_ref(angle_axis, 3);
182      if (use_quaternions_) {
183          QuaternionToAngleAxis(camera, angle_axis);
184      } else {
185          angle_axis_ref = ConstVectorRef(camera, 3);
186      }
187
188      // c = -R't
189      Eigen::VectorXd inverse_rotation = -angle_axis_ref;
190      AngleAxisRotatePoint(inverse_rotation.data(),
191                           camera + camera_block_size() - 6,
192                           center);
193      VectorRef(center, 3) *= -1.0;
194  }
195
196  void BALProblem::AngleAxisAndCenterToCamera(const double *angle_axis,
197                                              const double *center,
198                                              double *camera) const {
199      ConstVectorRef angle_axis_ref(angle_axis, 3);
200      if (use_quaternions_) {
201          AngleAxisToQuaternion(angle_axis, camera);
202      } else {
203          VectorRef(camera, 3) = angle_axis_ref;
204      }
205
206      // t = -R * c
```

```
207      AngleAxisRotatePoint(angle_axis, center, camera + camera_block_size() - 6);
208      VectorRef(camera + camera_block_size() - 6, 3) *= -1.0;
209  }
210
211  void BALProblem::Normalize() {
212      // Compute the marginal median of the geometry
213      std::vector<double> tmp(num_points_);
214      Eigen::Vector3d median;
215      double *points = mutable_points();
216      for (int i = 0; i < 3; ++i) {
217          for (int j = 0; j < num_points_; ++j) {
218              tmp[j] = points[3 * j + i];
219          }
220          median(i) = Median(&tmp);
221      }
222
223      for (int i = 0; i < num_points_; ++i) {
224          VectorRef point(points + 3 * i, 3);
225          tmp[i] = (point - median).lpNorm<1>();
226      }
227
228      const double median_absolute_deviation = Median(&tmp);
229
230      // Scale so that the median absolute deviation of the resulting
231      // reconstruction is 100
232
233      const double scale = 100.0 / median_absolute_deviation;
234
235      // X = scale * (X - median)
236      for (int i = 0; i < num_points_; ++i) {
237          VectorRef point(points + 3 * i, 3);
238          point = scale * (point - median);
239      }
240
241      double *cameras = mutable_cameras();
242      double angle_axis[3];
243      double center[3];
244      for (int i = 0; i < num_cameras_; ++i) {
245          double *camera = cameras + camera_block_size() * i;
246          CameraToAngelAxisAndCenter(camera, angle_axis, center);
247          // center = scale * (center - median)
248          VectorRef(center, 3) = scale * (VectorRef(center, 3) - median);
249          AngleAxisAndCenterToCamera(angle_axis, center, camera);
250      }
251  }
252
253  void BALProblem::Perturb(const double rotation_sigma,
254                           const double translation_sigma,
255                           const double point_sigma) {
256      assert(point_sigma >= 0.0);
257      assert(rotation_sigma >= 0.0);
258      assert(translation_sigma >= 0.0);
259
260      double *points = mutable_points();
261      if (point_sigma > 0) {
262          for (int i = 0; i < num_points_; ++i) {
263              PerturbPoint3(point_sigma, points + 3 * i);
264          }
265      }
266
267      for (int i = 0; i < num_cameras_; ++i) {
268          double *camera = mutable_cameras() + camera_block_size() * i;
269
270          double angle_axis[3];
271          double center[3];
272          // Perturb in the rotation of the camera in the angle-axis
273          // representation
274          CameraToAngelAxisAndCenter(camera, angle_axis, center);
275          if (rotation_sigma > 0.0) {
```

```
276             PerturbPoint3(rotation_sigma, angle_axis);
277         }
278         AngleAxisAndCenterToCamera(angle_axis, center, camera);
279
280         if (translation_sigma > 0.0)
281             PerturbPoint3(translation_sigma, camera + camera_block_size() - 6);
282     }
283 }
```