实验四 基于 EIGEN 的相机几何

2151094 宋正非

1. 实验任务

基于 Eigen 进行相机几何计算，将世界点投射到图像中。使用 Holmenkollen 数据集，每张图像包含固有校准 Intrinsic calibration、直升机在地理坐标系中的姿态 Helicopter pose in geographical coordinates 与相机相对于直升机的姿态 Camera pose relative to helicopter。

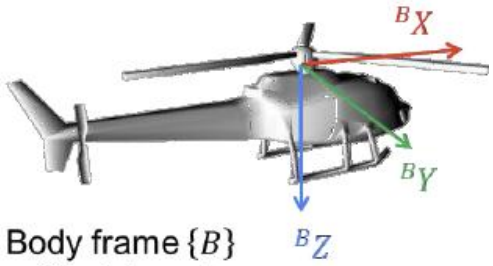

图 1 直升机坐标系

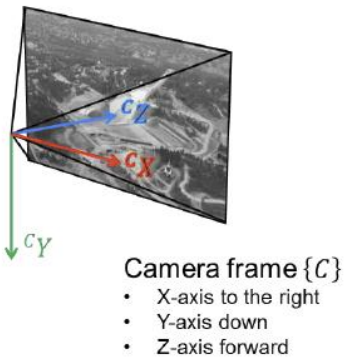

图 2 相机坐标系

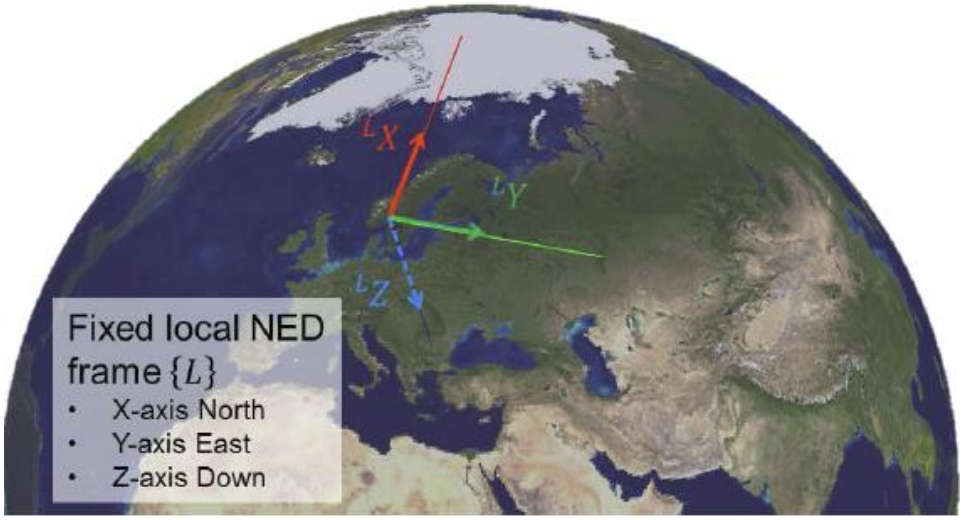

图 3 地球坐标系

2. 准备工作

检查 fmt 库是否已安装。该库应位于/usr/local/lib。可以使用 git clone 下载 fmt 源代码。

```
git clone https://git.tongji.edu.cn/ipmv/mirrors/fmt.git
```

并使用以下命令编译 fmt 源代码并安装 fmt lib。

```
cd fmt
mkdir build
cd build
cmake ..
make -j4
```

```
sudo make install
```

使用 git 命令从存储库克隆，下载未完成的 lab3 代码。

```
cd _to_your_work_folder
git clone https://git.tongji.edu.cn/ipmv/examples/lab3.git
```

3. 代码完善

根据 lab_3.cpp 中的提示完善代码。

```
#include "lab_3.h"

#include "dataset.h"
#include "local_coordinate_system.h"
#include "viewer_3d.h"
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"

Lab_3::Lab_3(const std::string& data_path)
 : data_path_{data_path}
 , window_name_{"World point in camera"}
{}

void Lab_3::run()
{
 // Set up dataset.
 Dataset dataset{data_path_};

 // Define local coordinate system based on the position of a light pole.
 const GeodeticPosition light_pole_position{59.963516, 10.667307, 321.0};
 const LocalCoordinateSystem local_system(light_pole_position);

 // Construct viewers.
 cv::namedWindow(window_name_);
 Viewer3D viewer;

 // Process each image in the dataset.
 for (DataElement element{}; dataset >> element;)
 {
  // Compute the pose of the body in the local coordinate system.
  // Todo: Finish Attitude::toQuaternion().
  const Sophus::SE3d pose_local_body = local_system.toLocalPose(element.body_position_in_geo,
                                                element.body_attitude_in_geo.toSO3());

  // Add body coordinate axis to the 3D viewer.
  // Todo: Write line of code below to add body to viewer.

  // Compute the pose of the camera relative to the body.
  // Todo: Finish CartesianPosition::toVector().
  // Todo: Construct pose_body_camera correctly using element.
  const Sophus::SE3d pose_body_camera{};

  // Compute the pose of the camera relative to the local coordinate system.
  // Todo: Construct pose_local_camera correctly using the poses above.
  const Sophus::SE3d pose_local_camera{};

  // Construct a camera model based on the intrinsic calibration and camera pose.
  // Todo: Finish Intrinsics::toCalibrationMatrix().
```

```
  // Todo: Finish Intrinsics::toDistortionVector().
  const PerspectiveCameraModel camera_model{element.intrinsics.toCalibrationMatrix(),
                                            pose_local_camera,
                                            element.intrinsics.toDistortionCoefficientVector()};

  // Undistort image.
  // Todo: Undistort image using the camera model. Why should we undistort the image?
  cv::Mat undistorted_img;

  // Project world point (the origin) into the image.
  // Todo: Finish PerspectiveCameraModel::computeCameraProjectionMatrix().
  // Todo: Finish PerspectiveCameraModel::projectWorldPoint().
  // Todo: Optionally finish PerspectiveCameraModel::projectWorldPoints().
  const Eigen::Vector2d pix_pos = camera_model.projectWorldPoint(Eigen::Vector3d::Zero());

  // Draw a marker in the image at the projected position.
  const Eigen::Vector2i pix_pos_int = (pix_pos.array().round()).cast<int>();
  cv::drawMarker(undistorted_img, {pix_pos_int.x(), pix_pos_int.y()}, {0.,0.,255.},
cv::MARKER_CROSS, 40, 3);

  // Show the image.
  // Todo: Write line of code below to show the image with the marker.

  // Add the camera to the 3D viewer.
  // Todo: Write line of code below to add body to viewer.

  // Update the windows.
  viewer.spinOnce();
  cv::waitKey(100);
 }

 // Remove image viewer.
 cv::destroyWindow(window_name_);

 // Run 3D viewer until stopped.
 viewer.spin();
}
```

代码的初始化部分包含：

- 读取数据集；
- 根据灯杆的位置定义局部坐标系；
- 创建 viewers。

循环每一张图片，图像处理部分包含：

- 读取图像，计算 body 在 local 坐标系中的姿势，此处需要完善 Attitude::toQuaternion()；
- 在 3D viewer 中增加 body 坐标轴，需要写代码将 body 增加到 viewer；
- 计算 camera 相对于 body 的姿势，完善 CartesianPosition::toVector()，构建 pose_body_camera；
- 计算 camera 相对于 local 坐标系的姿势，构建 pose_local_camera；
- 根据内在校准和相机姿势构建相机模型，完善 Intrinsics::toCalibrationMatrix()与 Intrinsics::toDistortionVector()；
- 使用相机模型对图像进行去畸变处理；

• 将世界中的原始点投影到图像中，完善 PerspectiveCameraModel 的成员函数 computeCameraProjectionMatrix(), projectWorldPoint()与 projectWorldPoints(). ；

• 在图像中投影的位置绘制一个标记并展示图像，需要写代码显示有标记的图像；

• 把相机增加到 3D viewer，需要写代码把 body 增加到 viewer 上；

• 更新窗口。

1) Finish Attitude::toQuaternion().

```
Eigen::Quaterniond Attitude::toQuaternion() const
{
 // Todo: Implement this correctly using Eigen::AngleAxisd (Z->Y->X).
 Eigen::Matrix3d rotation;
 rotation = Eigen::AngleAxisd(z_rot, Eigen::Vector3d::UnitZ()).toRotationMatrix();
 rotation *= Eigen::AngleAxisd(y_rot, Eigen::Vector3d::UnitY()).toRotationMatrix();
 rotation *= Eigen::AngleAxisd(x_rot, Eigen::Vector3d::UnitX()).toRotationMatrix();

 return Eigen::Quaterniond(rotation);
}
```

这个函数实现了将欧拉角（绕 Z 轴、Y 轴、X 轴的旋转角度）转换为四元数的过程，使用 Eigen 库的 AngleAxisd 类和 Quaterniond 类。四元数是处理 3D 旋转的更稳定方式，比传统的欧拉角表示方法更有效。

2) Write line of code below to add body to viewer.

```
  viewer.addBody(pose_local_body, element.img_num); //todo
```

3) Finish CartesianPosition::toVector().

```
Eigen::Vector3d CartesianPosition::toVector()
{
 // Todo: Implement this correctly.
 return Eigen::Vector3d(x, y, z);
}
```

这个函数将 CartesianPosition 类的坐标数据（即 x, y, z）转化为 Eigen 库的 Eigen::Vector3d 类型的三维向量。这样可以利用 Eigen::Vector3d 提供的各种向量操作（如加法、点积、叉积等）来处理这些坐标。

4) Construct pose_body_camera correctly using element.

```
  const Eigen::Vector3d t = element.camera_position_in_body.toVector(); //todo
  const Sophus::SO3d R = element.camera_attitude_in_body.toSO3(); //todo
  const Sophus::SE3d pose_body_camera(R, t); //todo
```

通过 element 对象中的位姿和姿态数据，构造一个表示相机相对于 body 坐标系的完整变换（pose_body_camera）。该变换包含了相机的旋转和平移部分，可以表示相机的在 body 坐标系中的位置与姿态。

5) Construct pose_local_camera correctly using the poses above.

```
  const Sophus::SE3d pose_local_camera = pose_local_body * pose_body_camera; //todo
```

先将物体从局部坐标系变换到 body 坐标系，然后再从 body 坐标系变换到相机坐标系，得到物体在相机坐标系下的位姿。

6) Finish Intrinsics::toCalibrationMatrix().

```
Eigen::Matrix3d Intrinsics::toCalibrationMatrix()
{
 // Todo: Construct the calibration matrix correctly.
 Eigen::Matrix3d K = Eigen::Matrix3d::Zero();
 K(0, 0) = fu;
 K(0, 1) = s;
 K(0, 2) = cu;
 K(1, 1) = fv;
 K(1, 2) = cv;
 K(2, 2) = 1.0;
 return K;
}
```

构建相机内参矩阵，结构如下：

$$K = \begin{bmatrix} f_u & s & c_u \\ 0 & f_v & c_v \\ 0 & 0 & 1 \end{bmatrix},$$

其中，fu 为水平方向的焦距，fv 为垂直方向的焦距，s 为剪切因子，主点 x 坐标 cu，主点 y 坐标 cv。

7) Finish Intrinsics::toDistortionCoefficientVector().

```
Intrinsics::Vector5d Intrinsics::toDistortionCoefficientVector()
{
 // Todo: Construct the distortion coefficients on the form [k1, k2, 0, 0, k3].
 Vector5d dist;
 dist << k1, k2, 0.0, 0.0, k3;
 return dist;
}
```

将相机的畸变系数构造为一个 5 维向量，格式为 [k1, k2, 0, 0, k3]，然后返回这个向量。这个向量通常用于相机标定中，纠正图像中的畸变。

8) Undistort image using the camera model.

```
  cv::Mat undistorted_img = camera_model.undistortImage(element.image); //todo
```

9) Finish PerspectiveCameraModel::computeCameraProjectionMatrix().

```
PerspectiveCameraModel::Matrix34d PerspectiveCameraModel::computeCameraProjectionMatrix()
{
 // Todo: Compute camera projection matrix.
 Matrix34d P;
 Sophus::SE3d T_camera_world = pose_world_camera_.inverse();
 Eigen::Matrix3d R = T_camera_world.rotationMatrix();
 Eigen::Vector3d t = T_camera_world.translation();
 P.leftCols<3>() = K_ * R;
```

```
 P.col(3) = K_ * t;
 return P;
}
```

实现计算相机投影矩阵的过程。首先，通过对世界到相机坐标系的变换矩阵 pose_world_camera_ 进行求逆，得到从相机坐标系到世界坐标系的变换矩阵。然后，从该变换矩阵中提取旋转矩阵和平移向量，分别表示相机相对于世界坐标系的旋转和平移。接着，将相机的内参矩阵与旋转矩阵相乘，得到投影矩阵的前三列，再将内参矩阵与平移向量相乘，得到投影矩阵的最后一列。最终，返回计算得到的 3x4 投影矩阵，它描述了如何将三维空间中的点映射到二维图像平面上。

10) Finish PerspectiveCameraModel::projectWorldPoint().

```
Eigen::Vector2d PerspectiveCameraModel::projectWorldPoint(Eigen::Vector3d world_point) const
{
 // Todo: Implement projection using camera_projection_matrix_.
 // return Eigen::Vector2d::Zero();
 Eigen::Vector4d world_point_h;
 world_point_h << world_point, 1.0;
 Eigen::Vector3d proj = camera_projection_matrix_ * world_point_h;
 return proj.hnormalized();
}
```

实现了将世界坐标系中的三维点投影到二维图像平面上的功能。首先，将三维世界点 world_point 扩展为齐次坐标形式，得到一个 4x1 的向量 world_point_h。然后，通过将该向量与相机投影矩阵 camera_projection_matrix_ 相乘，得到一个新的三维向量 proj，表示该点在图像平面上的投影（包括归一化的齐次坐标）。最后，通过调用 hnormalized() 将其归一化，得到二维图像平面上的坐标，并将其返回。

11) Optionally finish PerspectiveCameraModel::projectWorldPoints().

```
Eigen::Matrix2Xd PerspectiveCameraModel::projectWorldPoints(Eigen::Matrix3Xd world_points)
const
{
 // Todo: Optionally implement projection using camera_projection_matrix_.
 const int num_points = world_points.cols();
 Eigen::Matrix4Xd world_points_h(4, num_points);
 world_points_h.topRows<3>() = world_points;
 world_points_h.bottomRows<1>().setOnes();
 Eigen::Matrix3Xd projected = camera_projection_matrix_ * world_points_h;
 Eigen::Matrix2Xd pixel_coords(2, num_points);
 for (int i = 0; i < num_points; ++i) {
  double w = projected(2, i);
  pixel_coords(0, i) = projected(0, i) / w;
  pixel_coords(1, i) = projected(1, i) / w;
 }
 return pixel_coords;
}
```

实现了将多个世界坐标系中的三维点投影到二维图像平面上的功能，首先将世界点扩展为齐次坐标，然后通过相机投影矩阵计算投影结果，最后将齐次坐标归一化，得到每个点在图像平面上的二维坐标。

12) Write line of code below to show the image with the marker.

```
cv::imshow(window_name_, undistorted_img); //todo
```

13) Write line of code below to add body to viewer.

```
viewer.addCamera(camera_model, undistorted_img, element.img_num); //todo
```

4. 实验结果

运行之后得到结果如下。Vtk 3D 可视化窗口可以通过拖动鼠标来旋转视角。

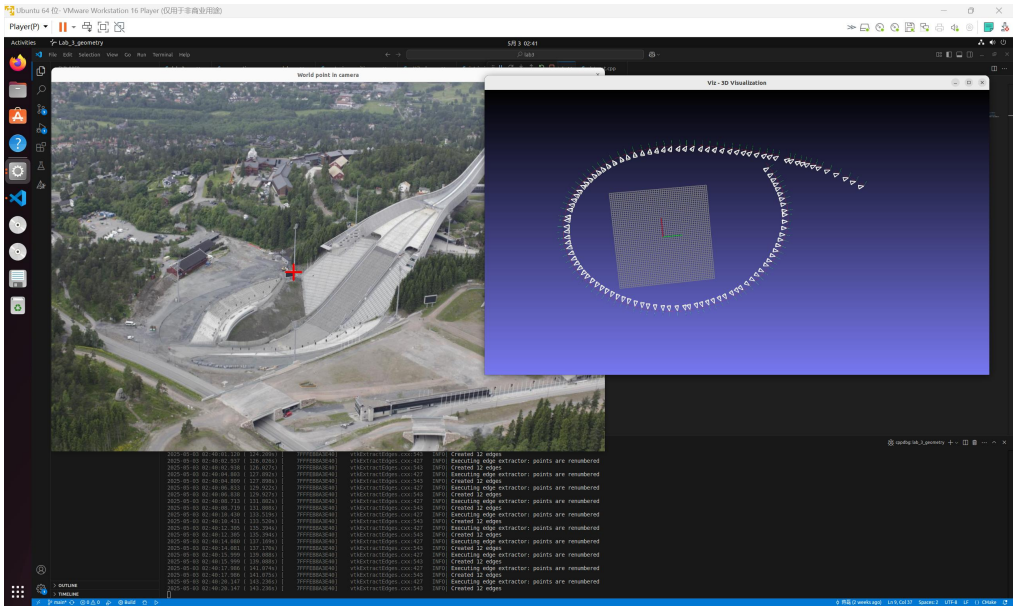

图 4 运行结果

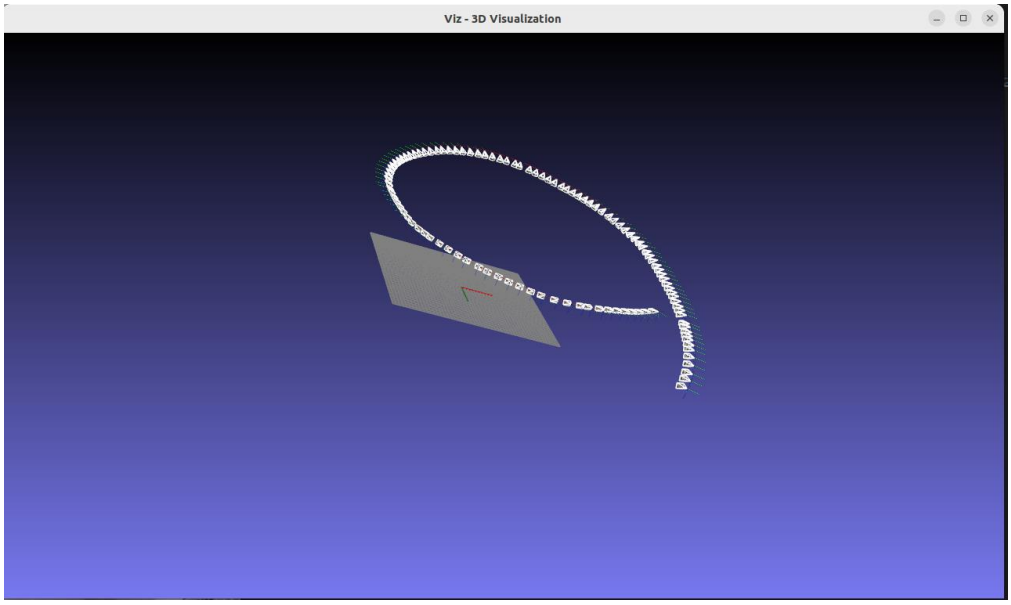

图 5 3D Visualization1

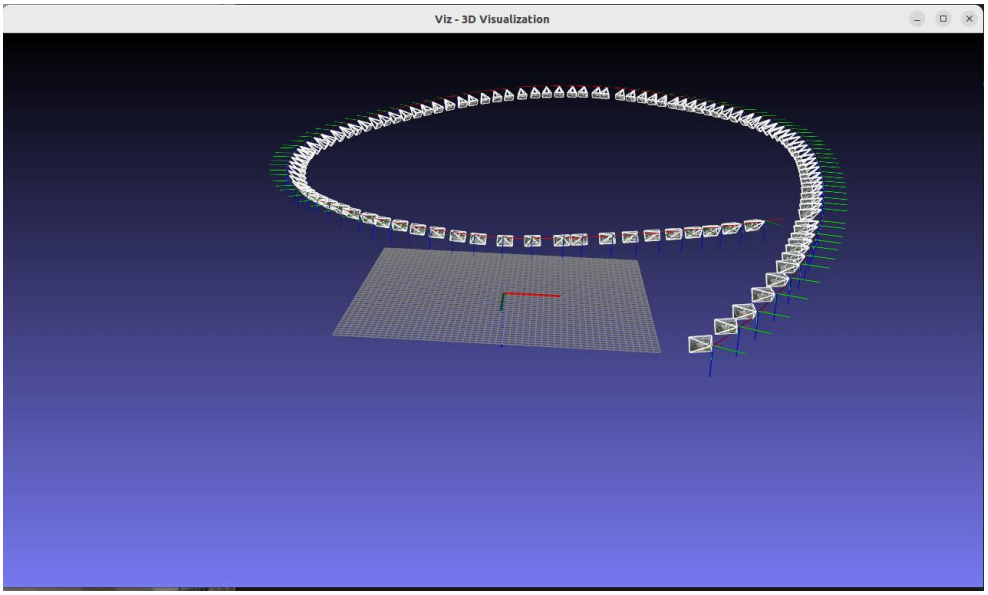

图 6 3D Visualization2

5. 附录（源代码）

1) lab_3.cpp

```
#include "lab_3.h"

#include "dataset.h"
#include "local_coordinate_system.h"
#include "viewer_3d.h"
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"

Lab_3::Lab_3(const std::string& data_path)
 : data_path_{data_path}
 , window_name_{"World point in camera"}
{}

void Lab_3::run()
{
 // Set up dataset.
 Dataset dataset{data_path_};

 // Define local coordinate system based on the position of a light pole.
 const GeodeticPosition light_pole_position{59.963516, 10.667307, 321.0};
 const LocalCoordinateSystem local_system(light_pole_position);

 // Construct viewers.
 cv::namedWindow(window_name_);
 Viewer3D viewer;

 // Process each image in the dataset.
 for (DataElement element{}; dataset >> element;)
 {
  // Compute the pose of the body in the local coordinate system.
  // Todo: Finish Attitude::toQuaternion().
  const Sophus::SE3d pose_local_body = local_system.toLocalPose(element.body_position_in_geo,
                                                element.body_attitude_in_geo.toSO3());
```

```
    // Add body coordinate axis to the 3D viewer.
    // Todo: Write line of code below to add body to viewer.
    viewer.addBody(pose_local_body, element.img_num);//todo

    // Compute the pose of the camera relative to the body.
    // Todo: Finish CartesianPosition::toVector().
    // Todo: Construct pose_body_camera correctly using element.
    const Eigen::Vector3d t = element.camera_position_in_body.toVector(); //todo
    const Sophus::SO3d R = element.camera_attitude_in_body.toSO3(); //todo
    const Sophus::SE3d pose_body_camera(R, t); //todo

    // Compute the pose of the camera relative to the local coordinate system.
    // Todo: Construct pose_local_camera correctly using the poses above.
    const Sophus::SE3d pose_local_camera = pose_local_body * pose_body_camera; //todo

    // Construct a camera model based on the intrinsic calibration and camera pose.
    // Todo: Finish Intrinsics::toCalibrationMatrix().
    // Todo: Finish Intrinsics::toDistortionVector().
    const PerspectiveCameraModel camera_model{element.intrinsics.toCalibrationMatrix(),
                                              pose_local_camera,
                                              element.intrinsics.toDistortionCoefficientVector()};

    // Undistort image.
    // Todo: Undistort image using the camera model. Why should we undistort the image?
    cv::Mat undistorted_img = camera_model.undistortImage(element.image); //todo

    // Project world point (the origin) into the image.
    // Todo: Finish PerspectiveCameraModel::computeCameraProjectionMatrix().
    // Todo: Finish PerspectiveCameraModel::projectWorldPoint().
    // Todo: Optionally finish PerspectiveCameraModel::projectWorldPoints().
    const Eigen::Vector2d pix_pos = camera_model.projectWorldPoint(Eigen::Vector3d::Zero());

    // Draw a marker in the image at the projected position.
    const Eigen::Vector2i pix_pos_int = (pix_pos.array().round()).cast<int>();
    cv::drawMarker(undistorted_img, {pix_pos_int.x(), pix_pos_int.y()}, {0.,0.,255.},
cv::MARKER_CROSS, 40, 3);

    // Show the image.
    // Todo: Write line of code below to show the image with the marker.
    cv::imshow(window_name_, undistorted_img); //todo

    // Add the camera to the 3D viewer.
    // Todo: Write line of code below to add body to viewer.
    viewer.addCamera(camera_model, undistorted_img, element.img_num); //todo

    // Update the windows.
    viewer.spinOnce();
    cv::waitKey(100);
  }

  // Remove image viewer.
  cv::destroyWindow(window_name_);

  // Run 3D viewer until stopped.
  viewer.spin();
}
```

2) attitude.cpp

```
#include "attitude.h"
#include "Eigen/Geometry"

Eigen::Quaterniond Attitude::toQuaternion() const
{
 // Todo: Implement this correctly using Eigen::AngleAxisd (Z->Y->X).
 Eigen::Matrix3d rotation;
 rotation = Eigen::AngleAxisd(z_rot, Eigen::Vector3d::UnitZ()).toRotationMatrix();
 rotation *= Eigen::AngleAxisd(y_rot, Eigen::Vector3d::UnitY()).toRotationMatrix();
 rotation *= Eigen::AngleAxisd(x_rot, Eigen::Vector3d::UnitX()).toRotationMatrix();

 return Eigen::Quaterniond(rotation);
}

Sophus::SO3d Attitude::toSO3() const
{
 return Sophus::SO3d(toQuaternion());
}

std::istream& operator>>(std::istream& is, Attitude& att)
{
 is >> att.x_rot
    >> att.y_rot
    >> att.z_rot;

 if (!is)
 {
  throw std::runtime_error("Could not read Attitude data");
 }

 return is;
}

std::ostream& operator<<(std::ostream& os, const Attitude& att)
{
 os << "x_rot: " << att.x_rot << "\n"
    << "y_rot: " << att.y_rot << "\n"
    << "z_rot: " << att.z_rot << "\n";

 return os;
}
```

3) cartesian_position.cpp

```
#include "cartesian_position.h"

Eigen::Vector3d CartesianPosition::toVector()
{
 // Todo: Implement this correctly.
 return Eigen::Vector3d(x, y, z);
}

std::istream& operator>>(std::istream& is, CartesianPosition& pos)
{
 is >> pos.x
    >> pos.y
    >> pos.z;

 if (!is)
 {
```

```
   throw std::runtime_error("Could not read CartiesianPosition data");
  }

  return is;
}

std::ostream& operator<<(std::ostream& os, const CartesianPosition& pos)
{
  os << "x: " << pos.x << "\n"
     << "y: " << pos.y << "\n"
     << "z: " << pos.z << "\n";

  return os;
}
```

4) intrinsics.cpp

```
#include "intrinsics.h"

Eigen::Matrix3d Intrinsics::toCalibrationMatrix()
{
  // Todo: Construct the calibration matrix correctly.
  Eigen::Matrix3d K = Eigen::Matrix3d::Zero();
  K(0, 0) = fu;
  K(0, 1) = s;
  K(0, 2) = cu;
  K(1, 1) = fv;
  K(1, 2) = cv;
  K(2, 2) = 1.0;
  return K;
}

Intrinsics::Vector5d Intrinsics::toDistortionCoefficientVector()
{
  // Todo: Construct the distortion coefficients on the form [k1, k2, 0, 0, k3].
  Vector5d dist;
  dist << k1, k2, 0.0, 0.0, k3;
  return dist;
}

std::istream& operator>>(std::istream& is, Intrinsics& intrinsics)
{
  is >> intrinsics.fu
     >> intrinsics.fv
     >> intrinsics.s
     >> intrinsics.cu
     >> intrinsics.cv
     >> intrinsics.k1
     >> intrinsics.k2
     >> intrinsics.k3;

  if (!is)
  {
    throw std::runtime_error("Could not read Intrinsics data");
  }

  return is;
}

std::ostream& operator<<(std::ostream& os, const Intrinsics& intrinsics)
```

```
{
 os << "fu: " << intrinsics.fu << "\n"
    << "fv: " << intrinsics.fv << "\n"
    << " s: " << intrinsics.s << "\n"
    << "cu: " << intrinsics.cu << "\n"
    << "cv: " << intrinsics.cv << "\n"
    << "k1: " << intrinsics.k1 << "\n"
    << "k2: " << intrinsics.k2 << "\n"
    << "k3: " << intrinsics.k3 << "\n";

 return os;
}
```

5) perspective_camera_model.cpp

```
#include "perspective_camera_model.h"
#include "opencv2/core/eigen.hpp"
#include "opencv2/imgproc.hpp"
#include "opencv2/calib3d.hpp"

PerspectiveCameraModel::PerspectiveCameraModel(const Eigen::Matrix3d& K,
                                               const Sophus::SE3d& pose_world_camera,
                                               const Vector5d& distortion_coeffs)
 : K_{K}
 , pose_world_camera_{pose_world_camera}
 , distortion_coeffs_{distortion_coeffs}
{
 camera_projection_matrix_ = computeCameraProjectionMatrix();
}

Sophus::SE3d PerspectiveCameraModel::getPose() const
{
 return pose_world_camera_;
}

Eigen::Matrix3d PerspectiveCameraModel::getCalibrationMatrix() const
{
 return K_;
}

PerspectiveCameraModel::Matrix34d PerspectiveCameraModel::getCameraProjectionMatrix() const
{
 return camera_projection_matrix_;
}

Eigen::Vector2d PerspectiveCameraModel::projectWorldPoint(Eigen::Vector3d world_point) const
{
 // Todo: Implement projection using camera_projection_matrix_.
 Eigen::Vector4d world_point_h;
 world_point_h << world_point, 1.0;
 Eigen::Vector3d proj = camera_projection_matrix_ * world_point_h;
 return proj.hnormalized();
}

Eigen::Matrix2Xd PerspectiveCameraModel::projectWorldPoints(Eigen::Matrix3Xd world_points)
const
{
 // Todo: Optionally implement projection using camera_projection_matrix_.
 const int num_points = world_points.cols();
 Eigen::Matrix4Xd world_points_h(4, num_points);
```

```
  world_points_h.topRows<3>() = world_points;
  world_points_h.bottomRows<1>().setOnes();
  Eigen::Matrix3Xd projected = camera_projection_matrix_ * world_points_h;
  Eigen::Matrix2Xd pixel_coords(2, num_points);
  for (int i = 0; i < num_points; ++i) {
    double w = projected(2, i);
    pixel_coords(0, i) = projected(0, i) / w;
    pixel_coords(1, i) = projected(1, i) / w;
  }
  return pixel_coords;
}

PerspectiveCameraModel::Matrix34d PerspectiveCameraModel::computeCameraProjectionMatrix()
{
  // Todo: Compute camera projection matrix.
  Matrix34d P;
  Sophus::SE3d T_camera_world = pose_world_camera_.inverse();
  Eigen::Matrix3d R = T_camera_world.rotationMatrix();
  Eigen::Vector3d t = T_camera_world.translation();
  P.leftCols<3>() = K_ * R;
  P.col(3) = K_ * t;
  return P;
}

cv::Mat PerspectiveCameraModel::undistortImage(cv::Mat distorted_image) const
{
  // Convert to cv::Mats
  cv::Mat K_cv;
  cv::eigen2cv(K_, K_cv);
  cv::Mat dist_coeffs_cv;
  cv::eigen2cv(distortion_coeffs_, dist_coeffs_cv);

  // Undistort image.
  cv::Mat undistorted_image;
  cv::undistort(distorted_image, undistorted_image, K_cv, dist_coeffs_cv);

  return undistorted_image;
}
```