sad-L5-hw

Yixiao Feng

23/06/2023

1 实现基于优化器的点到点 ICP、点到线 ICP

基于优化器的实现选用了 g2o 工具。首先，在 g2o_types.h 文件中实现 point2point 和 point2line 两种形式的边，具体实现如下：

```
// g2o_types.h 111行
/**
 * TODO: 第五章习题1，P2P配准的Edge定义
 */
class EdgeSE2Point2Point : public g2o::BaseUnaryEdge<2, Vec2d, VertexSE2> {
  public:
   EIGEN_MAKE_ALIGNED_OPERATOR_NEW
   using Point2d = pcl::PointXY;
   using Cloud2d = pcl::PointCloud<Point2d>;

   EdgeSE2Point2Point(pcl::search::KdTree<Point2d>* kdtree, Cloud2d::ConstPtr target_cloud, double range,
        double angle)
       : kdtree_(kdtree), target_cloud_(target_cloud), range_(range), angle_(angle) {}

   void computeError() override {
       VertexSE2* v = (VertexSE2 *)_vertices[0];
       SE2 pose = v->estimate();
       Vec2d pw = pose * Vec2d(range_ * std::cos(angle_), range_ * std::sin(angle_));
       Point2d pt;
       pt.x = pw.x();
       pt.y = pw.y();

       nn_idx.clear();
       nn_dis.clear();
       kdtree_->nearestKSearch(pt, 1, nn_idx, nn_dis);

       if (nn_idx.size() > 0 && nn_dis[0] < max_dis2) {
           Point2d nn_pt = target_cloud_->points[nn_idx[0]];
           _error = Vec2d(pt.x - nn_pt.x, pt.y - nn_pt.y);
       } else {
           _error = Vec2d(0, 0);
           setLevel(1);
       }
   }

   void linearizeOplus() override {
       VertexSE2* v = (VertexSE2 *)_vertices[0];
       SE2 pose = v->estimate();
       float theta = pose.so2().log();

       if (nn_idx.size() > 0 && nn_dis[0] < max_dis2) {
           _jacobianOplusXi << 1, 0, -range_ * std::sin(angle_ + theta), 0, 1, range_ * std::cos(angle_ + theta
               );
       } else {
           _jacobianOplusXi.setZero();
           setLevel(1);
       }
   }

   bool read(std::istream& is) override { return true; }
   bool write(std::ostream& os) const override { return true; }
```

```
  private:
   pcl::search::KdTree<Point2d>* kdtree_;
   std::vector<int> nn_idx;
   std::vector<float> nn_dis;
   Cloud2d::ConstPtr target_cloud_;
   double range_ = 0;
   double angle_ = 0;
   const float max_dis2 = 0.01;
};

/**
 * TODO: P2L配准的Edge定义
 */
class EdgeSE2Point2Line : public g2o::BaseUnaryEdge<1, double, VertexSE2> {
  public:
   EIGEN_MAKE_ALIGNED_OPERATOR_NEW
   using Point2d = pcl::PointXY;
   using Cloud2d = pcl::PointCloud<Point2d>;

   EdgeSE2Point2Line(pcl::search::KdTree<Point2d>* kdtree, Cloud2d::ConstPtr target_cloud, double range,
        double angle)
       : kdtree_(kdtree), target_cloud_(target_cloud), range_(range), angle_(angle) {}

   void computeError() override {
       VertexSE2* v = (VertexSE2 *)_vertices[0];
       SE2 pose = v->estimate();
       Vec2d pw = pose * Vec2d(range_ * std::cos(angle_), range_ * std::sin(angle_));
       Point2d pt;
       pt.x = pw.x();
       pt.y = pw.y();

       nn_idx.clear();
       nn_dis.clear();
       kdtree_->nearestKSearch(pt, 5, nn_idx, nn_dis);

       effective_pts.clear();
       for (int i = 0; i < nn_idx.size(); ++i) {
           if (nn_dis[i] < max_dis) {
               effective_pts.emplace_back(
                   Vec2d(target_cloud_->points[nn_idx[i]].x, target_cloud_->points[nn_idx[i]].y));
           }
       }

       if (effective_pts.size() < 3) {
           fit_success = false;
           _error[0] = 0;
           setLevel(1);
       } else {
           if (math::FitLine2D(effective_pts, line_coeffs)) {
               fit_success = true;
               _error[0] = line_coeffs[0] * pw.x() + line_coeffs[1] * pw.y() + line_coeffs[2];
           } else {
               fit_success = false;
               _error[0] = 0;
               setLevel(1);
           }
       }
```

```
    }

    void linearizeOplus() override {
        VertexSE2* v = (VertexSE2 *)_vertices[0];
        SE2 pose = v->estimate();
        float theta = pose.so2().log();

        if (!fit_success) {
            _jacobianOplusXi.setZero();
            setLevel(1);
        } else {
            _jacobianOplusXi << line_coeffs[0], line_coeffs[1],
                -line_coeffs[0] * range_ * std::sin(angle_ + theta) + line_coeffs[1] * range_ * std::cos(angle_
                    + theta);
        }
    }

    bool read(std::istream& is) override { return true; }
    bool write(std::ostream& os) const override { return true; }

  private:
    pcl::search::KdTree<Point2d>* kdtree_;
    std::vector<int> nn_idx;
    std::vector<float> nn_dis;
    Cloud2d::ConstPtr target_cloud_;
    std::vector<Vec2d> effective_pts;
    Vec3d line_coeffs;
    bool fit_success = false;
    double range_ = 0;
    double angle_ = 0;
    const float max_dis = 0.3;
};
```

然后，在 icp_2d.cc 文件中定义优化器、求解器、顶点和边，最后进行迭代优化。test_2d_icp_s2s.cc 中也增加了相应的使用优化器求解 ICP 的 gflags。具体实现如下：

```
// icp_2d.cc 189行
/**
 * TODO: 第五章习题1，使用G2O进行P2P配准
 * @param init_pose
 * @return
 */
bool Icp2d::AlignG2OP2P(SE2& init_pose) {
    using BlockSolverType = g2o::BlockSolver<g2o::BlockSolverTraits<3, 2>>;
    using LinearSolverType = g2o::LinearSolverCholmod<BlockSolverType::PoseMatrixType>;
    auto* solver = new g2o::OptimizationAlgorithmLevenberg(
        g2o::make_unique<BlockSolverType>(g2o::make_unique<LinearSolverType>()));
    g2o::SparseOptimizer optimizer;
    optimizer.setAlgorithm(solver);

    auto* v = new VertexSE2();
    v->setId(0);
    v->setEstimate(init_pose);
    optimizer.addVertex(v);

    const double rk_delta = 0.8;
```

```
    for (size_t i = 0; i < source_scan_->ranges.size(); ++i) {
        float r = source_scan_->ranges[i];
        if (r < source_scan_->range_min || r > source_scan_->range_max) {
            continue;
        }

        float angle = source_scan_->angle_min + i * source_scan_->angle_increment;
        if (angle < source_scan_->angle_min + 30 * M_PI / 180.0 || angle > source_scan_->angle_max - 30 * M_PI
             / 180.0) {
            continue;
        }

        auto e = new EdgeSE2Point2Point(&kdtree_, target_cloud_, r, angle);
        e->setVertex(0, v);
        e->setInformation(Eigen::Matrix2d::Identity());
        auto rk = new g2o::RobustKernelHuber;
        rk->setDelta(rk_delta);
        e->setRobustKernel(rk);
        optimizer.addEdge(e);
    }

    optimizer.setVerbose(false);
    optimizer.initializeOptimization();
    optimizer.optimize(10);

    init_pose = v->estimate();
    return true;
}

/**
 * TODO: 使用G2O进行P2L配准
 * @param init_pose
 * @return
 */
bool Icp2d::AlignG2OP2L(SE2& init_pose) {
    using BlockSolverType = g2o::BlockSolver<g2o::BlockSolverTraits<3, 1>>;
    using LinearSolverType = g2o::LinearSolverCholmod<BlockSolverType::PoseMatrixType>;
    auto* solver = new g2o::OptimizationAlgorithmLevenberg(
        g2o::make_unique<BlockSolverType>(g2o::make_unique<LinearSolverType>()));
    g2o::SparseOptimizer optimizer;
    optimizer.setAlgorithm(solver);

    auto* v = new VertexSE2();
    v->setId(0);
    v->setEstimate(init_pose);
    optimizer.addVertex(v);

    const double rk_delta = 0.8;

    for (size_t i = 0; i < source_scan_->ranges.size(); ++i) {
        float r = source_scan_->ranges[i];
        if (r < source_scan_->range_min || r > source_scan_->range_max) {
            continue;
        }

        float angle = source_scan_->angle_min + i * source_scan_->angle_increment;
        if (angle < source_scan_->angle_min + 30 * M_PI / 180.0 || angle > source_scan_->angle_max - 30 * M_PI
```

```
            / 180.0) {
            continue;
        }

        auto e = new EdgeSE2Point2Line(&kdtree_, target_cloud_, r, angle);
        e->setVertex(0, v);
        e->setInformation(Eigen::Matrix<double, 1, 1>::Identity());
        auto rk = new g2o::RobustKernelHuber;
        rk->setDelta(rk_delta);
        e->setRobustKernel(rk);
        optimizer.addEdge(e);
    }

    optimizer.setVerbose(false);
    optimizer.initializeOptimization();
    optimizer.optimize(10);

    init_pose = v->estimate();
    return true;
}
```

使用 G2O 优化的 P2L 实验结果如图 1（单帧截图）:

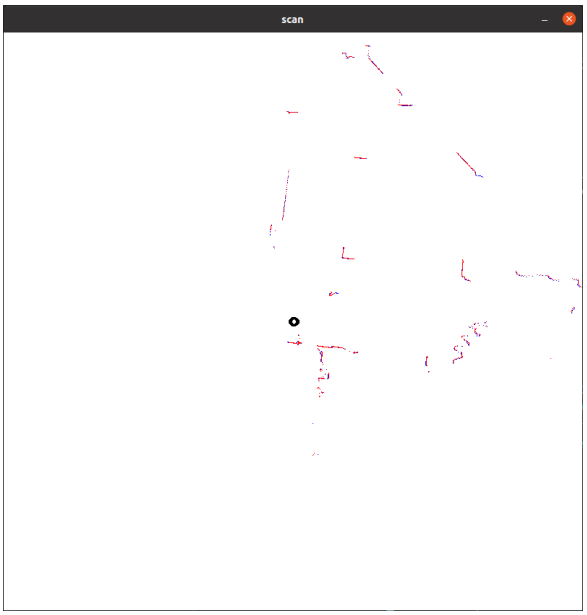

图 1: 基于 G2O 优化的 Point2Line ICP 实验结果

2 实现对似然场图像进行插值后进行 scan matching 的算法

基于 G2O 图优化的方法已经做了对似然场图像的双线性插值（双线性插值的实现在 math_utils.h 文件中），手写高斯牛顿的方法中也可以做相应改动，实现很简单，不做过多解释了，具体实现如下：

```
// 第五章习题2 (likelihood_field.cc 75)
for (size_t i = 0; i < source_->ranges.size(); ++i) {
    float r = source_->ranges[i];
    if (r < source_->range_min || r > source_->range_max) {
        continue;
    }

    float angle = source_->angle_min + i * source_->angle_increment;
    if (angle < source_->angle_min + 30 * M_PI / 180.0 || angle > source_->angle_max - 30 * M_PI / 180.0) {
        continue;
    }

    float theta = current_pose.so2().log();
    Vec2d pw = current_pose * Vec2d(r * std::cos(angle), r * std::sin(angle));

    // 在field中的图像坐标
    Vec2i pf = (pw * resolution_ + Vec2d(500, 500)).cast<int>();

    if (pf[0] >= image_boarder && pf[0] < field_.cols - image_boarder && pf[1] >= image_boarder &&
        pf[1] < field_.rows - image_boarder) {
        effective_num++;

        // 图像梯度
        // TODO: 第五章习题2, 改成双线性插值
        float dx = 0.5 * (math::GetPixelValue<float>(field_, pf[0] + 1, pf[1]) -
                          math::GetPixelValue<float>(field_, pf[0] - 1, pf[1]));
        float dy = 0.5 * (math::GetPixelValue<float>(field_, pf[0], pf[1] + 1) -
                          math::GetPixelValue<float>(field_, pf[0], pf[1] - 1));

        Vec3d J;
        J << resolution_ * dx, resolution_ * dy,
            -resolution_ * dx * r * std::sin(angle + theta) + resolution_ * dy * r * std::cos(angle + theta);
        H += J * J.transpose();

        float e = math::GetPixelValue<float>(field_, pf[0], pf[1]);
        b += -J * e;

        cost += e * e;
    } else {
        has_outside_pts_ = true;
    }
}
```

3 基于直线拟合的方法，讨论并实现对单个 Scan 退化检测的算法

首先，需要对每帧 scan 进行聚类，我选用的是 DBSCAN 的方法，具体实现在 dbscan.h 文件中，代码如下：

```
// dbscan.h
#ifndef SLAM_IN_AUTO_DRIVING_DBSCAN_H
#define SLAM_IN_AUTO_DRIVING_DBSCAN_H

#include "common/lidar_utils.h"

#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/search/impl/kdtree.hpp>

namespace sad {

#define UN_PROCESSED 0
#define PROCESSING 1
#define PROCESSED 2

template <typename PointT>
class DBSCANSimpleCluster {
  public:
   typedef typename pcl::PointCloud<PointT>::Ptr PointCloudPtr;
   typedef typename pcl::search::KdTree<PointT>::Ptr KdTreePtr;

   DBSCANSimpleCluster() {}

   virtual void setInputCloud(PointCloudPtr cloud) {
      input_cloud_ = cloud;
   }

   void setSearchMethod(KdTreePtr tree) {
      search_method_ = tree;
   }

   void extract(std::vector<pcl::PointIndices>& cluster_indices) {
      std::vector<int> nn_indices;
      std::vector<float> nn_distances;
      std::vector<bool> is_noise(input_cloud_->points.size(), false);
      std::vector<int> types(input_cloud_->points.size(), UN_PROCESSED);
      for (int i = 0; i < input_cloud_->points.size(); i++) {
         if (types[i] == PROCESSED) {
            continue;
         }
         int nn_size = radiusSearch(i, eps_, nn_indices, nn_distances);
         if (nn_size < minPts_) {
            is_noise[i] = true;
            continue;
         }
         std::vector<int> seed_queue;
         seed_queue.push_back(i);
         types[i] = PROCESSED;
         for (int j = 0; j < nn_size; j++) {
            if (nn_indices[j] != i) {
               seed_queue.push_back(nn_indices[j]);
               types[nn_indices[j]] = PROCESSING;
            }
```

```
        } // for every point near the chosen core point.
        int sq_idx = 1;
        while (sq_idx < seed_queue.size()) {
            int cloud_index = seed_queue[sq_idx];
            if (is_noise[cloud_index] || types[cloud_index] == PROCESSED) {
                // seed_queue.push_back(cloud_index);
                types[cloud_index] = PROCESSED;
                sq_idx++;
                continue; // no need to check neighbors.
            }
            nn_size = radiusSearch(cloud_index, eps_, nn_indices, nn_distances);
            if (nn_size >= minPts_) {
                for (int j = 0; j < nn_size; j++) {
                    if (types[nn_indices[j]] == UN_PROCESSED) {

                        seed_queue.push_back(nn_indices[j]);
                        types[nn_indices[j]] = PROCESSING;
                    }
                }
            }
            types[cloud_index] = PROCESSED;
            sq_idx++;
        }
        if (seed_queue.size() >= min_pts_per_cluster_ && seed_queue.size () <= max_pts_per_cluster_) {
            pcl::PointIndices r;
            r.indices.resize(seed_queue.size());
            for (int j = 0; j < seed_queue.size(); ++j) {
                r.indices[j] = seed_queue[j];
            }
            // These two lines should not be needed: (can anyone confirm?) -FF
            std::sort (r.indices.begin (), r.indices.end ());
            r.indices.erase (std::unique (r.indices.begin (), r.indices.end ()), r.indices.end ());

            r.header = input_cloud_->header;
            cluster_indices.push_back (r); // We could avoid a copy by working directly in the vector
        }
    } // for every point in input cloud
    std::sort (cluster_indices.rbegin(), cluster_indices.rend(), [&](const pcl::PointIndices &a, const pcl
        ::PointIndices &b) {
        return (a.indices.size() < b.indices.size());
    });
}

void setClusterTolerance(double tolerance) {
    eps_ = tolerance;
}

void setMinClusterSize(int min_cluster_size) {
    min_pts_per_cluster_ = min_cluster_size;
}

void setMaxClusterSize(int max_cluster_size) {
    max_pts_per_cluster_ = max_cluster_size;
}

void setCorePointMinPts(int core_point_min_pts) {
    minPts_ = core_point_min_pts;
```

```
    }

  private:
    PointCloudPtr input_cloud_;
    double eps_ = 0.0;
    int minPts_ = 1; // not including the point itself.
    int min_pts_per_cluster_ = 1;
    int max_pts_per_cluster_ = std::numeric_limits<int>::max();
    KdTreePtr search_method_;

    virtual int radiusSearch(
        int index, double radius, std::vector<int> &k_indices,
        std::vector<float> &k_sqr_distances) const {
        return this->search_method_->radiusSearch(index, radius, k_indices, k_sqr_distances);
    }
};  // class DBSCANCluster
} // namespace sad

#endif // SLAM_IN_AUTO_DRIVING_DBSCAN_H
```

而后，我对每帧 scan 处理的结果进行了可视化，具体实现在 degeneration_detection.h 和 *.cc 两个文件中，实现如下：

```
// degeneration_detection.h
#ifndef SLAM_IN_AUTO_DRIVING_DEGENERATION_DETECTION_H
#define SLAM_IN_AUTO_DRIVING_DEGENERATION_DETECTION_H

#include "ch6/dbscan.h"
#include "common/math_utils.h"

namespace sad {

class DegenDetection {
  public:
    using Point2d = pcl::PointXY;
    using Cloud2d = pcl::PointCloud<Point2d>;

    DegenDetection() {}

    void SetScan(Scan2d::Ptr scan) {
        scan_ = scan;
        BuildKdTree();
    }

    /**
     * 初始化DBSCAN聚类
     * @param coreMinPts 核心点邻域内最少点数
     * @param minPtsPerCluster 每个cluster最少点数
     * @param maxPtsPerCluster 每个cluster最多点数
     * @param eps 邻域距离
     */
    void InitDBSCAN(int coreMinPts, int minPtsPerCluster, int maxPtsPerCluster, double eps);

    void ShowClustering(cv::Mat& image, int image_size, float resolution);

    void BuildKdTree();
```

```
  private:
   Scan2d::Ptr scan_ = nullptr;
   Cloud2d::Ptr cloud_;
   pcl::search::KdTree<Point2d>::Ptr kdtree_;

   DBSCANSimpleCluster<Point2d> dbscan_;
   std::vector<pcl::PointIndices> cluster_indices;
};

}

#endif // SLAM_IN_AUTO_DRIVING_DEGENERATION_DETECTION_H
```

```
// degeneration_detection.cc
#include "ch6/degeneration_detection.h"

#include <glog/logging.h>

namespace sad {

void DegenDetection::InitDBSCAN(int coreMinPts, int minPtsPerCluster, int maxPtsPerCluster, double eps) {
    dbscan_.setCorePointMinPts(coreMinPts);
    dbscan_.setClusterTolerance(eps);
    dbscan_.setMinClusterSize(minPtsPerCluster);
    dbscan_.setMaxClusterSize(maxPtsPerCluster);
    dbscan_.setSearchMethod(kdtree_);
    dbscan_.setInputCloud(cloud_);
    cluster_indices.clear();
    dbscan_.extract(cluster_indices);
    LOG(INFO) << "cluster个数: " << cluster_indices.size();
}

void DegenDetection::ShowClustering(cv::Mat& image, int image_size, float resolution) {
    if (image.data == nullptr) {
        image = cv::Mat(image_size, image_size, CV_8UC3, cv::Vec3b(255, 255, 255));
    }

    for (size_t i = 0; i < cluster_indices.size(); ++i) {
        for (size_t j = 0; j < cluster_indices[i].indices.size(); ++j) {
            Point2d pt = cloud_->points[cluster_indices[i].indices[j]];
            int image_x = int(pt.x * resolution + image_size / 2);
            int image_y = int(pt.y * resolution + image_size / 2);
            if (image_x >= 0 && image_x < image.cols && image_y >= 0 && image_y < image.rows) {
                image.at<cv::Vec3b>(image_y, image_x) = cv::Vec3b(
                    (i+1) * 25 % 256, (i+1) * 75 % 256, (i+1) * 125 % 256);
            }
        }
    }
}

void DegenDetection::BuildKdTree() {
    if (scan_ == nullptr) {
        LOG(ERROR) << "scan is not set";
        return;
    }

    cloud_.reset(new Cloud2d);
```

```
    for (size_t i = 0; i < scan_->ranges.size(); ++i) {
        if (scan_->ranges[i] < scan_->range_min || scan_->ranges[i] > scan_->range_max) {
            continue;
        }

        double real_angle = scan_->angle_min + i * scan_->angle_increment;

        Point2d p;
        p.x = scan_->ranges[i] * std::cos(real_angle);
        p.y = scan_->ranges[i] * std::sin(real_angle);
        cloud_->points.push_back(p);
    }

    cloud_->width = cloud_->points.size();
    cloud_->is_dense = false;
    kdtree_.reset(new pcl::search::KdTree<Point2d>);
    kdtree_->setInputCloud(cloud_);
}

}
```

聚类的可视化结果如图 2：

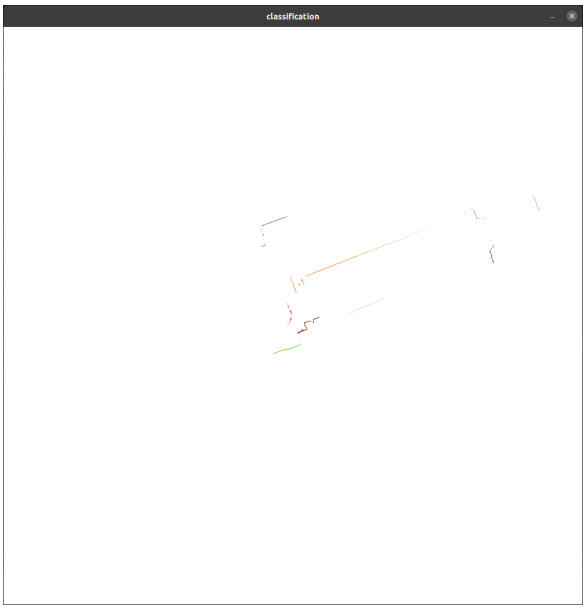

图 2: DBSCAN 聚类结果

聚类后需要对每个 cluster 做二维直线拟合，对有效拟合的直线检查直线方向（可以检查斜度 $-\frac{a}{b}$），如果满足两条直线的方向一致且没有其他方向的直线（或者说，直线方向较为统一），就很有可能是走廊场景。实现的代码如下：

```
// degeneration_detection.cc 23行
bool DegenDetection::DetectDegeneration() {
    if (cluster_indices.empty()) {
        LOG(ERROR) << "no enough clusters";
        return false;
    }

    // 简化了检测退化场景的过程，仅检测了走廊场景
```

```
    if (cluster_indices.size() == 2) {
        line_coeffs.clear();
        for (size_t i = 0; i < cluster_indices.size(); ++i) {
            effective_pts.clear();
            for (size_t j = 0; j < cluster_indices[i].indices.size(); ++j) {
                Point2d pt = cloud_->at(cluster_indices[i].indices[j]);
                effective_pts.emplace_back(Vec2d(pt.x, pt.y));
            }
            Vec3d coeffs;
            math::FitLine2D(effective_pts, coeffs);
            line_coeffs.emplace_back(coeffs);
            slope.emplace_back(- coeffs[0] / coeffs[1]);
        }

        if (slope[0] - slope[1] < slope_tolerance)
            return true;
    }
    return false;
}
```

退化场景的检测结果如图 3:

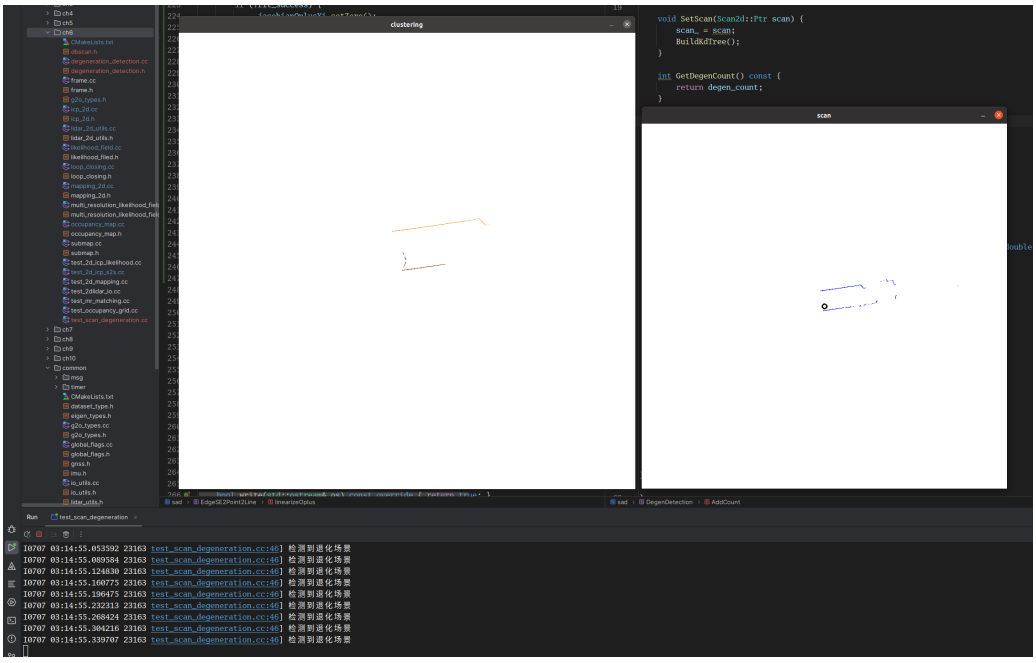

图 3: 退化检测结果

4 讨论在实际机器人中，如何处理悬空物体和低矮物体

- 提供先验地图：对已经建立好的地图标注悬空物体和低矮物体以及其他潜在的未被观测到的障碍物，用于只有单一观测或观测不足的机器人。
- 添加其他传感器：如测量低矮物体可以给机器人安装上超声波传感器，利用超声波传感器避障的具体实现可以写在下位机里，作为软件层面之上的避障策略。除此之外，还可以添加摄像头等三维测量传感器，对三维场景进行建图（octomap、elevation mapping 等），使用带有三维信息的地图导航。使用图像信息还可以对场景进行语义建图、traversability analysis 等，用来判断机器人是否可以从低矮物体上方通行，以及对不同障碍物应用不同的避障策略。