Homework 5

[Question 1] Develop Point-Point and Point-2-Line ICP Using G2O

Please checkout this file in my own 2D SLAM implementation, inspired by the course

Point - Point ICP

```
class EdgeICP2D : public g2o::BaseUnaryEdge<2, Vec2d, VertexSE2> {
  public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW
    EdgeICP2D(size_t point_idx, std::vector<NNMatch> *matches_ptr,
              PCLCloud2DPtr source_map_cloud, const PCLCloud2DPtr pcl_target_cloud,
              const std::vector<ScanObj> *source_scan_objs_ptr) : point_idx_(point_idx),
                                                                  matches_ptr_(matches_ptr),
                                                                  pcl_target_cloud_(pcl_targ
    }

    void computeError() override {

        const auto &match = matches_ptr_->at(point_idx_);
        auto target_pt    = pcl_target_cloud_->points.at(match.closest_pt_idx_in_other_clouc
        auto target_vec   = to_eigen(target_pt);
        auto source_pt    = source_map_cloud_->points.at(point_idx_);   // this changes in c
        auto source_vec   = to_eigen(source_pt);
        double dist       = math::get_squared_distance(target_vec, source_vec);

        // Invalid point
        if (dist > PT_MAX_VALID_SQUARED_DIST) {
            _error = Vec2d(0, 0);
            setLevel(1);   // marks the edge out of bound, so it will be ignored during opt
            std::cerr << "Could happen - EdgeICP2D: point match is too far" << dist << std::
            return;
        }

        VertexSE2 *v     = (VertexSE2 *)_vertices[0];
        SE2 relative_pose         = v->estimate();
        double r     = source_scan_objs_ptr_->at(point_idx_).range;
        double angle = source_scan_objs_ptr_->at(point_idx_).angle;
        Vec2d pw = relative_pose * Vec2d(r * std::cos(angle), r * std::sin(angle));
        _error = pw - target_vec;
    }

    // Not called for optimization if the edge already is bad
    void linearizeOplus() override {
```

```cpp
        VertexSE2 *v     = (VertexSE2 *)_vertices[0];
        SE2 pose         = v->estimate();
        float pose_angle = pose.so2().log();
        double r         = source_scan_objs_ptr_->at(point_idx_).range;
        double angle     = source_scan_objs_ptr_->at(point_idx_).angle;
        //         Eigen::MatrixXd J(2, 3);
        _jacobianOplusXi << 1, 0, -r * std::sin(angle + pose_angle), 0, 1, r * std::cos(ang
    }

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

  private:
    size_t point_idx_;
    std::vector<NNMatch> *matches_ptr_                = nullptr;
    PCLCloud2DPtr source_map_cloud_                   = nullptr;
    PCLCloud2DPtr pcl_target_cloud_                   = nullptr;
    const std::vector<ScanObj> *source_scan_objs_ptr_ = nullptr;
};

class ICP2DG2O : public ICP2D {
    // 1. Create A vertex is an SE2 pose.
    // 2. Add edges. An edge is the distance between each point and its the closest point
    // 3. Iteration: Quit optimization using chi^2.
    //  1. Conduct KD_tree search.
  public:
    explicit ICP2DG2O(LaserScanMsg::SharedPtr source, LaserScanMsg::SharedPtr target) : ICP2

    bool point_line_icp_g2o(SE2 &relative_pose, double &cost) const {
        double pose_angle = relative_pose.so2().log();
        const size_t n    = source_scan_objs_.size();
        cost              = 0;
    }

    bool point_point_icp_g2o(SE2 &relative_pose, double &cost) const {
        double pose_angle      = relative_pose.so2().log();
        const size_t n         = source_scan_objs_.size();
        cost                   = 0;
        using BlockSolverType  = g2o::BlockSolver<g2o::BlockSolverTraits<3, 1>>;
        using LinearSolverType = g2o::LinearSolverCholmod<BlockSolverType::PoseMatrixType>;
        auto *solver           = new g2o::OptimizationAlgorithmLevenberg(
                      std::make_unique<BlockSolverType>(std::make_unique<LinearSolverType>()
        g2o::SparseOptimizer optimizer;
        optimizer.setAlgorithm(solver);
        auto *v = new VertexSE2();
        v->setEstimate(relative_pose);
```

```
v->setId(0);   // So optimizer will create a look up [id, vertex]?
optimizer.addVertex(v);

std::vector<NNMatch> matches;
PCLCloud2DPtr source_map_cloud(new pcl::PointCloud<PCLPoint2D>());

// Add edges:
for (size_t point_idx = 0; point_idx < n; ++point_idx) {
    auto e = new EdgeICP2D(point_idx, &matches, source_map_cloud, pcl_target_cloud_
    e->setVertex(0, v);   // 0 is the index of the vertex within this edge
    e->setInformation(Eigen::Matrix<double, 2, 2>::Identity());
    auto rk               = new g2o::RobustKernelHuber;
    const double rk_delta = 0.8;
    rk->setDelta(rk_delta);
    e->setRobustKernel(rk);
    optimizer.addEdge(e);
}

optimizer.setVerbose(true);

for (size_t iter = 0; iter < GAUSS_NEWTON_ITERATIONS; ++iter) {
    // 1: Get each source point's map pose
    source_map_cloud->points =
        std::vector<pcl::PointXY, Eigen::aligned_allocator<pcl::PointXY>>(n);
    std::transform(std::execution::par_unseq,
                   source_scan_objs_.begin(), source_scan_objs_.end(), source_map_cl
                   [&](const ScanObj &s) {
                       Vec2d p_vec = scan_point_to_map_frame(s.range, s.angle, relat
                       pcl::PointXY pt;
                       pt.x = p_vec[0];
                       pt.y = p_vec[1];
                       return pt;
                   });
    // 2: Find nearest point in target.
    matches.clear();
    bool found_neighbors = nano_tree_->search_tree_multi_threaded(
        source_map_cloud, matches, 1);
    if (!found_neighbors) {
        std::cerr << "No KD TREE Neighbor found in point-point icp g2o!" << std::enc
        return false;
    }
    // update edges using the matches, because matches[i] is the match of source_map
    // See "matches[i * k + j].idx_in_this_cloud             = i;", (nanoflann_kdtre
    optimizer.initializeOptimization();
    optimizer.optimize(1);
```

```
            // Update pose and chi2
            relative_pose = v->estimate();
            cost          = optimizer.chi2();
        }
        return true;
    }
};
```

From this implementation, we get the following scan matching result, with and without the handwritten Jacobian:

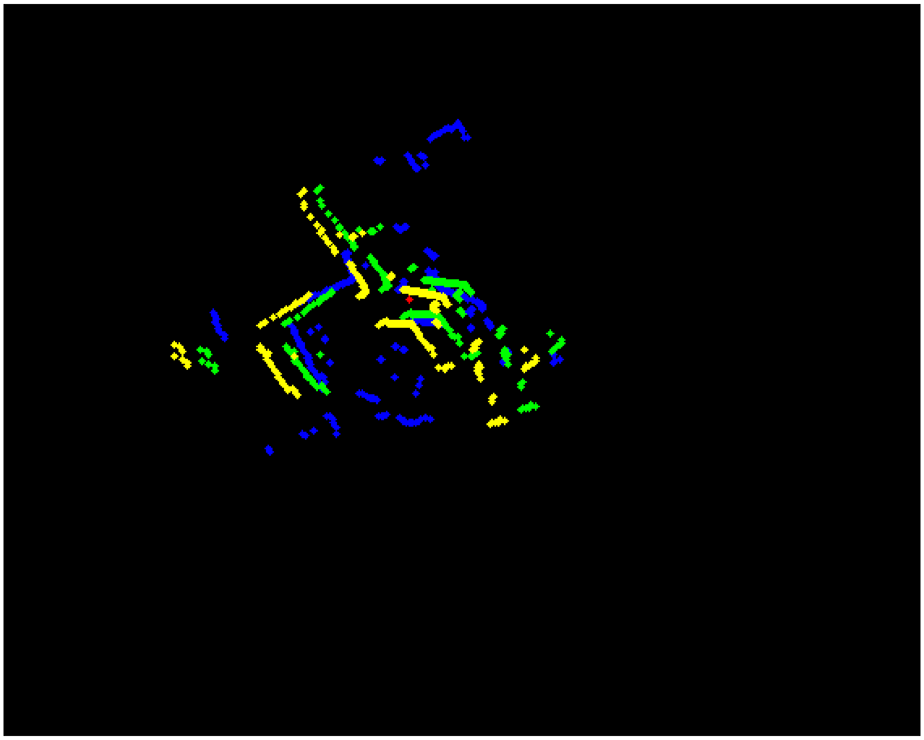

Figure 1: map4

Point-Line ICP

```
class EdgeICP2D_PT2Line : public g2o::BaseUnaryEdge<1, double, VertexSE2> {
  public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW
    EdgeICP2D_PT2Line(size_t point_idx, std::vector<PointLine2DICPData> *point_line_data_vec
                      PCLCloud2DPtr source_map_cloud, const PCLCloud2DPtr pcl_target_cloud,
                      const std::vector<ScanObj> *source_scan_objs_ptr) : point_idx_(point_i
                                                                          point_line_data_ve
```

```
                                                                          source_map_cloud_(
                                                                          pcl_target_cloud_(
                                                                          source_scan_objs_p
    }

    // point_line_data_vec[point_idx_] is the data for this edge
    void computeError() override {
        auto *pose   = dynamic_cast<const VertexSE2 *>(_vertices[0]);   // static cast?
        double range = source_scan_objs_ptr_->at(point_idx_).range;
        double angle = source_scan_objs_ptr_->at(point_idx_).angle;

        Vec2d pw          = Vec2d(range * std::cos(angle), range * std::sin(angle));
        auto line_coeffs = point_line_data_vec_ptr_->at(point_idx_).params_;
        _error[0]         = line_coeffs[0] * pw[0] + line_coeffs[1] * pw[1] + line_coeffs[2]
    }

    // Not called for optimization if the edge already is bad
    void linearizeOplus() override {
        VertexSE2 *v = (VertexSE2 *)_vertices[0];
        SE2 pose     = v->estimate();
        float pose_angle = pose.so2().log();
        double r          = source_scan_objs_ptr_->at(point_idx_).range;
        double angle      = source_scan_objs_ptr_->at(point_idx_).angle;
        double a = point_line_data_vec_ptr_->at(point_idx_).params_[0], b = point_line_data_
        _jacobianOplusXi << a, b, -a * r * std::sin(angle + pose_angle) + b * r * std::cos(a
    }

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

  private:
    size_t point_idx_;
    std::vector<PointLine2DICPData> *point_line_data_vec_ptr_ = nullptr;
    PCLCloud2DPtr source_map_cloud_                           = nullptr;
    PCLCloud2DPtr pcl_target_cloud_                           = nullptr;
    const std::vector<ScanObj> *source_scan_objs_ptr_         = nullptr;
};

bool point_line_icp_g2o(SE2 &relative_pose, double &cost) {
    const size_t n = source_scan_objs_.size();
    cost           = 0;

    for (size_t iter = 0; iter < GAUSS_NEWTON_ITERATIONS; ++iter) {
        using BlockSolverType  = g2o::BlockSolver<g2o::BlockSolverTraits<3, 1>>;
        using LinearSolverType = g2o::LinearSolverCholmod<BlockSolverType::PoseMatrixType>;
        auto *solver           = new g2o::OptimizationAlgorithmLevenberg(
```

```
                std::make_unique<BlockSolverType>(std::make_unique<LinearSolverType>
g2o::SparseOptimizer optimizer;
optimizer.setAlgorithm(solver);
auto *v = new VertexSE2();
v->setEstimate(relative_pose);
v->setId(0);   // So optimizer will create a look up [id, vertex]?
optimizer.addVertex(v);

std::vector<NNMatch> matches;
PCLCloud2DPtr source_map_cloud(new pcl::PointCloud<PCLPoint2D>());
std::vector<PointLine2DICPData> point_line_data_vec;

// Add edges:
for (size_t point_idx = 0; point_idx < n; ++point_idx) {
    auto e = new EdgeICP2D_PT2Line(point_idx, &point_line_data_vec, source_map_clouc
    e->setInformation(Eigen::Matrix<double, 1, 1>::Identity() * 1e4);   // TODO
    e->setVertex(0, v);                                                 // 0 is the
    auto rk               = new g2o::RobustKernelHuber;
    const double rk_delta = 0.8;
    rk->setDelta(rk_delta);
    e->setRobustKernel(rk);
    optimizer.addEdge(e);
}

// 1: Get each source point's map pose
source_map_cloud->points =
    std::vector<pcl::PointXY, Eigen::aligned_allocator<pcl::PointXY>>(n);
std::transform(std::execution::par_unseq,
                source_scan_objs_.begin(), source_scan_objs_.end(), source_map_clouc
                [&](const ScanObj &s) {
                    Vec2d p_vec = scan_point_to_map_frame(s.range, s.angle, relative
                    pcl::PointXY pt;
                    pt.x = p_vec[0];
                    pt.y = p_vec[1];
                    return pt;
                });
// 2: Find nearest K point in target.
matches.clear();

bool found_neighbors = nano_tree_->search_tree_multi_threaded(
    source_map_cloud, matches, PL_ICP_K_NEAREST_NUM);
if (!found_neighbors) {
    std::cerr << "No KD TREE Neighbor found in point-line icp g2o!" << std::endl;
    return false;
}
```

```
        point_line_data_vec = knn_to_line_fitting_data(matches, PL_ICP_K_NEAREST_NUM, source
        // update edges using the matches, because matches[i] is the match of source_map_cl
        // See "matches[i * k + j].idx_in_this_cloud            = i;", (nanoflann_kdtree.h
        optimizer.setVerbose(true);
        optimizer.initializeOptimization();
        optimizer.optimize(1);

        // Update pose and chi2
        relative_pose = v->estimate();
        cost          = optimizer.chi2();
    }
    return true;
}
```

I encountered an *epic bug* that took me a few hours to debug. See this blogpost

[Question 2] Implement Bicubic Interpolation.

"When there is no disaster, there is no comparison" - Rico Jia (Me). Dr. Gao's dataset is extremely clean, which I love. However, we can only see the effect of Bi-cubic in a "messier case". I'm using my own multi-resolution likelihood-field implementation, with bicubic and bi-linear functions. Here are the results:

- Bi-Linear: my robot has turned 270 deg in a room. At the end, there are quite a bit of accumulated errors, and the scan is matched to a very different orientation.
- Bi-Cubic: though there are multiple shades of walls (accumulated errors), the scan is generally matched correctly!

Bicubic Interpolation Implementation

```
// Cubic interpolation using the Catmull-Rom spline
inline float cubic_interpolate(float p0, float p1, float p2, float p3, float t) {
    float a0 = -0.5f * p0 + 1.5f * p1 - 1.5f * p2 + 0.5f * p3;
    float a1 = p0 - 2.5f * p1 + 2.0f * p2 - 0.5f * p3;
    float a2 = -0.5f * p0 + 0.5f * p2;
    float a3 = p1;
    return ((a0 * t + a1) * t + a2) * t + a3;
}

template <typename T>
inline float get_bicubic_interpolated_pixel_value(const cv::Mat &img, float y, float x) {
    // Clamp coordinates to valid range.
    if (x < 0)
        x = 0;
    if (y < 0)
        y = 0;
```

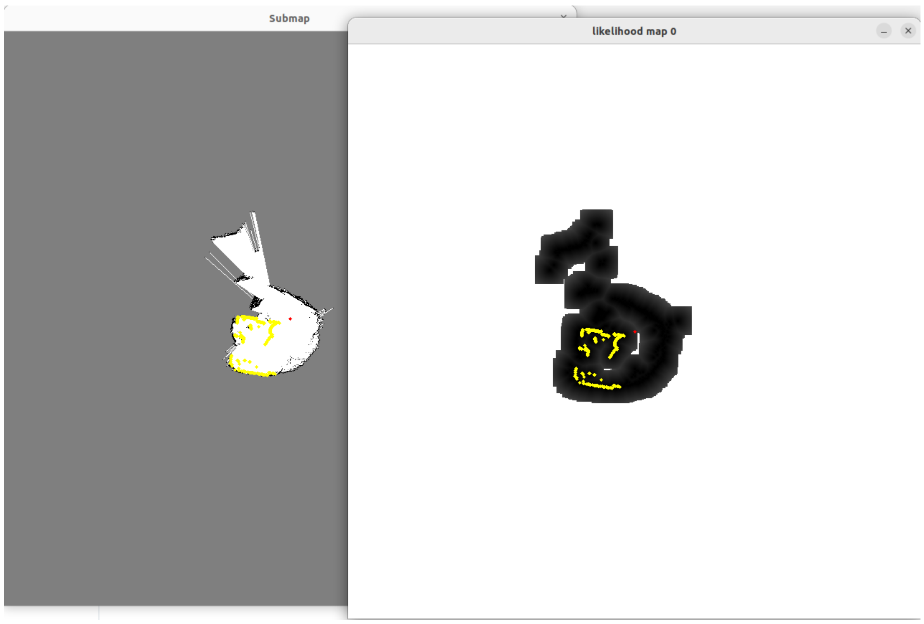

Figure 2: map1

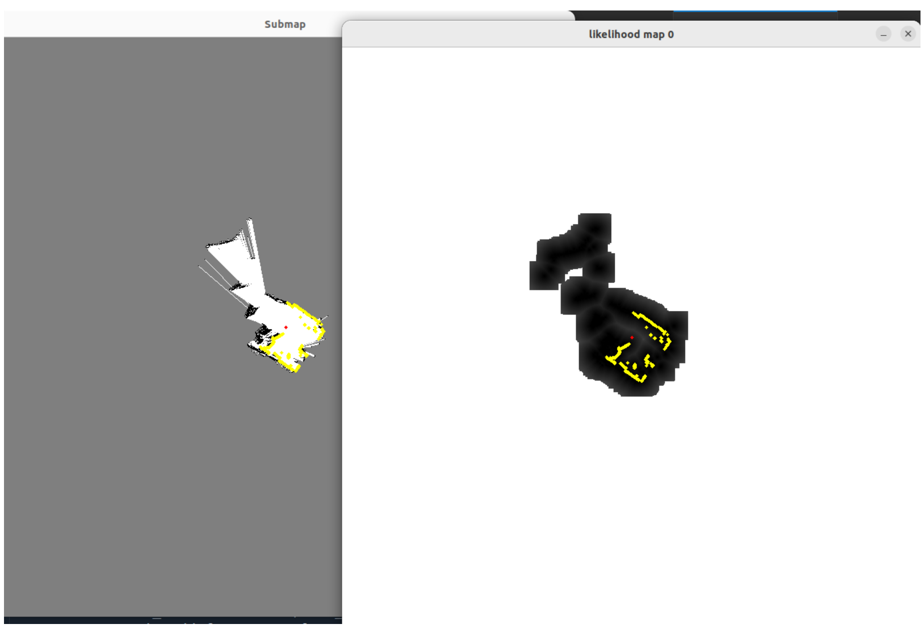

Figure 3: map2

```
    if (x >= img.cols)
        x = img.cols - 1;
    if (y >= img.rows)
        y = img.rows - 1;

    // Get integer and fractional parts.
    int x_int    = static_cast<int>(std::floor(x));
    int y_int    = static_cast<int>(std::floor(y));
    float x_frac = x - x_int;
    float y_frac = y - y_int;

    // Extract a 4x4 neighborhood; indices relative to floor(x) and floor(y) are: -1, 0, 1,
    float patch[4][4];
    for (int m = -1; m <= 2; ++m) {
        // Clamp y index.
        int y_index = std::min(std::max(y_int + m, 0), img.rows - 1);
        for (int n = -1; n <= 2; ++n) {
            // Clamp x index.
            int x_index         = std::min(std::max(x_int + n, 0), img.cols - 1);
            patch[m + 1][n + 1] = static_cast<float>(img.at<T>(y_index, x_index));
        }
    }

    // Interpolate in x-direction for each of the 4 rows.
    float col[4];
    for (int m = 0; m < 4; ++m) {
        col[m] = cubic_interpolate(patch[m][0], patch[m][1], patch[m][2], patch[m][3], x_fra
    }

    // Interpolate in y-direction using the x-interpolated values.
    return cubic_interpolate(col[0], col[1], col[2], col[3], y_frac);
}
```

https://ricojia.github.io/2017/01/26/interpolation/#2d-case

[Question 3] Based On Line Fitting, What Can Be Done For A Single Scan's Degradation Detection?

What is scan degradation? Scan degradation happens when the environment lacks features, like a long hallway. In that case, all major lines are pointing roughly in the same direction.

In that case, an intuitive option is to use an IMU for a short period of time.

Here we have two cases:

1. A certain segment has `~max_range` consecutively
2. We have lines

What the program does is:

1. outlier removal:
    1. For each point, find 30 nearest neighbors
    2. Find the mean and standard deviation of all points.
    3. Remove points that are outside of 1.0 standard deviation.
2. Euclidean Distance Clustering
    1. Pick a point as the seed of a cluster
    2. Iterate through all points,
        - add a new point to the cluster if it's within a distance threshold to any existing point in the cluster
    3. Add a new cluster with a new point if the previous cluster is full, or the new point is outside of the previous clusters.
    4. Repeat the above steps
3. Line Direction Detection
    1. Fit a line in each "big" clusters: `ax + by +c = 0`
    2. Calculate directions of these clusters. If the directions of these lines are the similar, then we can say a scan degradation occured.

```
// REFERENCE: https://zhuanlan.zhihu.com/p/642853423
bool detect_2d_degradation(std::shared_ptr<sensor_msgs::msg::LaserScan> current_scan_ptr
    PCLCloud3DPtr cloud = laser_scan_2_PointXYZ(current_scan_ptr);
    PCLCloud3DPtr cloud_filtered(new pcl::PointCloud<PCLPoint3D>);

    pcl::StatisticalOutlierRemoval<PCLPoint3D> sor;
    sor.setInputCloud(cloud);
    sor.setMeanK(30);
    sor.setStddevMulThresh(1.0);
    sor.filter(*cloud_filtered);

    pcl::search::KdTree<PCLPoint3D>::Ptr kdtree_; // (new pcl::search::KdTree<PCLPoint3
    kdtree_ = std::make_shared<pcl::search::KdTree<PCLPoint3D>>();
    kdtree_->setInputCloud(cloud_filtered);      // filtered cloud is our source of KD

    pcl::EuclideanClusterExtraction<PCLPoint3D> clusterExtractor_;
    // Vector that stores clustering results
    std::vector<pcl::PointIndices> cluster_indices;
    clusterExtractor_.setClusterTolerance(0.1);
    clusterExtractor_.setMinClusterSize(10);            // each cluster contains at leas
    clusterExtractor_.setMaxClusterSize(1000);          // each cluster contains at mos
    clusterExtractor_.setSearchMethod(kdtree_);         // Find nearby neighbors using
    clusterExtractor_.setInputCloud(cloud_filtered);
    clusterExtractor_.extract(cluster_indices);

    CloudViewer<PCLPoint3D> cloud_viewer;
    auto viewer = cloud_viewer.get_viewer();
```

```
int clusterNumber = 1;
std::vector<Vec3f> line_coeffs;
for (const auto& indices : cluster_indices) {
    std::cout << "Cluster " << clusterNumber << " has " << indices.indices.size() <<
    pcl::PointCloud<PCLPoint3D>::Ptr cluster(new pcl::PointCloud<PCLPoint3D>);
    pcl::copyPointCloud(*cloud_filtered, indices, *cluster);
    double r = static_cast<double>(rand()) / RAND_MAX;
    double g = static_cast<double>(rand()) / RAND_MAX;
    double b = static_cast<double>(rand()) / RAND_MAX;

    std::string clusterId = "cluster_" + std::to_string(clusterNumber);
    viewer->addPointCloud<PCLPoint3D>(cluster, clusterId);
    viewer->setPointCloudRenderingProperties(pcl::visualization::PCL_VISUALIZER_COLO
    clusterNumber++;

    // Calculate line coeffs if the point cloud size is larger than 10
    if (cluster ->size() > 10){
        line_coeffs.emplace_back(math::fit_line_2d(cluster));
    }
}

size_t same_dir_count = 0;
for(size_t i = 0; i < line_coeffs.size(); ++ i){
    const auto& l1 = line_coeffs.at(i);
    float a1 = l1[0], b1 = l1[1];
    for(size_t j = i+1; j < line_coeffs.size(); ++ j){
        const auto& l2 = line_coeffs.at(j);
        float a2 = l2[0], b2 = l2[1];
        if (std::fabs(a1 * b2 - a2 * b1) < 1e-1){
            same_dir_count ++;
        }
    }
}
// // Keep the viewer running until the user closes the window.
// viewer->spin();
// // Reset the viewer to avoid potential segfaults.
// viewer.reset();
// //TODO
// std::cout<<"same_dir_count"<<same_dir_count<<std::endl;

float line_combo_num = (clusterNumber * (clusterNumber-1))/2;
if(line_combo_num * 2 / 3.0 < same_dir_count){
    std::cout<<"Scan has no degredation"<<std::endl;
    return false;
} else {
    std::cout<<"Scan has degredation"<<std::endl;
```

```
            return true;
        }
    }
```

Sample output:

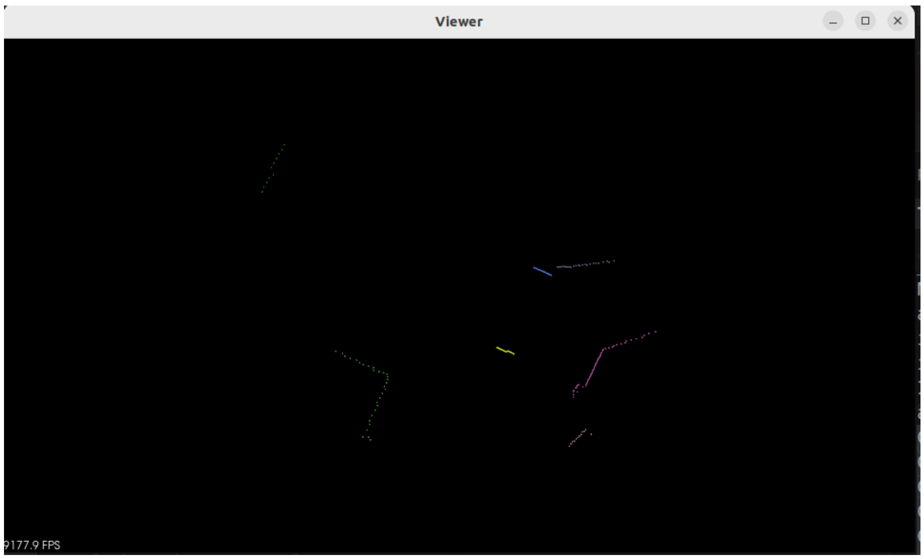

Figure 4: img

```
Cluster 1 has 52 points.
Cluster 2 has 33 points.
Cluster 3 has 30 points.
Cluster 4 has 25 points.
Cluster 5 has 20 points.
Cluster 6 has 12 points.
Cluster 7 has 11 points.
same_dir_count5
Scan has no degredation
```

[Question 4] Discuss how to handle low-hanging objects in actual robots

If they are within the heights of the robot, we need to have a 3D Lidar / RGBD Camera that can cover that. Then, they are projected down to 2D. Or, one can incorporate height information to each pixel, so you get a 2.5D map.