Homework 4

[Question 1] Develop NEARBY14 for the KNN search in voxels

Source

For Nearby 14 voxel search, we just need to additionally search the corner 8 voxels of the search box:

```
nearby_grids_ = {KeyType(0, 0, 0),  KeyType(-1, 0, 0), KeyType(1, 0, 0), KeyType(0, 1, 0),
                     KeyType(0, -1, 0), KeyType(0, 0, -1), KeyType(0, 0, 1) , KeyType(1, 1, 1
                     KeyType(1, 1, -1), KeyType(1, -1, 1), KeyType(1, -1, -1), KeyType(-1, 1,
                     KeyType(-1, 1, -1), KeyType(-1, -1, 1), KeyType(-1, -1, -1)};
```

However, **I think it's more reasonable to define NEARBY18 instead of NEARBY14, as they are closer**

```
_nearby_grids = {NNCoord(0, 0, 0), NNCoord(-1, 0, 0),
                 NNCoord(1, 0, 0), NNCoord(0, 1, 0),
                 NNCoord(0, -1, 0), NNCoord(0, 0, -1),
                 NNCoord(1, 1, 0), NNCoord(1, -1, 0),
                 NNCoord(-1, 1, 0), NNCoord(-1, -1, 0),
                 NNCoord(0, 1, 1), NNCoord(0, 1, -1),
                 NNCoord(0, -1, 1), NNCoord(0, -1, -1),
                 NNCoord(1, 0, 1), NNCoord(1, 0, -1),
                 NNCoord(-1, 0, 1), NNCoord(-1, 0, -1)
            };
```

Performance is:

- 19k points, 4ms (NEARBY 18), recall: 76.6%, precision: 78.6%

[Question 2] Proof

Prove that the solution to the question below is the largest eigen vector or Singular vector

$$d^* = argmax_d |Ad|^2$$

Proof:

$$\begin{aligned}
&|Ad^2| = (Ad)^T(Ad) = d^T A^T A d \\
&\text{Using Eigen Value Decomposition:} \\
&A^T A = \\
&\Rightarrow \\
&|Ad^2| = d^T V \Lambda V^T d
\end{aligned}$$

Where V and Λ are eigen vectors and their eigen values:

$$V = [v_1|v_2 \cdots |v_n]$$
$$\Lambda = diag(\lambda_1^2, \lambda_2^2 \cdots)$$

Let:

$$d = \alpha_1 v_1 + \alpha_2 v_2 + \cdots + \alpha_n v_n$$

Then we have:

$$|Ad^2| = d^T V \Lambda V^T d = [\alpha_1 | \alpha_2 | \cdots | \alpha_n]$$

Since we have imposed the length constraint:

$$|d| = 1 = \alpha_1^2 + \alpha_2^2 + \cdots \alpha_n^2$$

Maximum of $|Ad^2|$ is achieved when $\alpha_n = 1$ for the largest eigen value λ_n. d is now v_n (They are also "singular vectors if we do SVD on A")

[Question 3] Compare The Performance of `nanoflann` With This Chapter's KNN Searches

J. L. Blanco and P. K. Rai, "nanoflann: a C++ header-only fork of FLANN, a library for nearest neighbor (NN) with kd-trees." https://github.com/jlblancoc/nanoflann, 2014.

Given 19k `XYZI` points:

The performance of Nanoflann is (`max leaf size = 10`):

- k = 1, 100%, 4ms
- k = 5, 100%, 4ms

The performance of my KD tree is:

- k = 1, 100%, 6ms
- k = 5, 100%, 6ms

Therefore, Nanoflann KD Tree is by far the best KNN Search method

Here is the code for testing:

```
#pragma once
#include <halo/common/sensor_data_definitions.hpp>
#include <nanoflann/nanoflann.hpp>

namespace halo{
struct NanoflannPointCloudAdaptor {
```

```
    // Reference to the actual point cloud data
    const halo::PointCloudType &pts;

    // Constructor
    explicit NanoflannPointCloudAdaptor (const halo::PointCloudType &points) : pts(points) {

    // Must return the number of data points
    inline size_t kdtree_get_point_count() const { return pts.points.size(); }

    // Returns the dim'th component of the idx'th point in the class:
    inline float kdtree_get_pt(const size_t idx, const size_t dim) const {
        if (dim == 0) return pts.points[idx].x;
        else if (dim == 1) return pts.points[idx].y;
        else return pts.points[idx].z;
    }

    // Optional bounding-box computation: return false to default to a standard bbox comput
    template <class BBOX>
    bool kdtree_get_bbox(BBOX & /*bb*/) const { return false; }
};

template <typename PointT, int dim>
class NanoFlannKDTree {
public:
    using CloudPtr = std::shared_ptr<pcl::PointCloud<PointT>>;
    using PointCloudAdaptor = NanoflannPointCloudAdaptor;
    // Using 3 dimensions (for 3D point clouds). If you need a different dimensionality,
    // you could templatize the dimension.
    using KDTreeType = nanoflann::KDTreeSingleIndexAdaptor<
        nanoflann::L2_Simple_Adaptor<float, PointCloudAdaptor>,
        PointCloudAdaptor,
        dim /* dimensionality */
    >;

    // Constructor:
    // The KDTree parameters (e.g., maximum leaf size).
    NanoFlannKDTree(const PointCloudAdaptor &adaptor,
                    const nanoflann::KDTreeSingleIndexAdaptorParams &params)
        : adaptor_(adaptor), kd_tree_(dim, adaptor_, params) {kd_tree_.buildIndex();}

    // This function performs a multi-threaded nearest neighbor search.
    // It expects:
    // - query_cloud: the point cloud whose points will be searched against the kd-tree.
    // - matches: a pre-sized vector where each query point will yield k matches
    //            (i.e. matches.size() should equal query_cloud->points.size() * k).
    // - k: the number of nearest neighbors to find for each query point.
```

```
    //
    // Returns true on success, false otherwise.
    bool search_tree_multi_threaded(const CloudPtr &query_cloud,
                                    std::vector<NNMatch> &matches, size_t k) const {
        if (!query_cloud || query_cloud->points.empty()) {
            return false;
        }
        size_t num_points = query_cloud->points.size();
        matches.resize(num_points * k);

        const size_t num_results = k;
        // Create an index container [0, 1, 2, ..., num_points-1]
        std::vector<size_t> indices(num_points);
        std::iota(indices.begin(), indices.end(), 0);

        // Process each query point in parallel.
        std::for_each(std::execution::par_unseq, indices.begin(), indices.end(),
            [&](size_t i) {
                const auto &pt = query_cloud->points[i];
                float query_pt[3] = { pt.x, pt.y, pt.z };

                // Allocate temporary storage for this iteration.
                std::vector<typename KDTreeType::IndexType> local_ret_index(num_results);
                std::vector<float> local_out_dist_sqr(num_results);

                kd_tree_.knnSearch(query_pt, num_results, local_ret_index.data(), local_out_
                for (size_t j = 0; j < k; ++j) {
                    matches[i * k + j].idx_in_this_cloud = i;
                    matches[i * k + j].closest_pt_idx_in_other_cloud = local_ret_index[j];
                }
            });

        return true;
    }


private:
        // We store a copy of the adaptor here. It holds a reference to the original cloud.
    PointCloudAdaptor adaptor_;
    KDTreeType kd_tree_;
};
};

TEST(TestKNN, test_nanoflann_kdtree) {
    // Load point clouds from files.
    halo::CloudPtr first(new halo::PointCloudType);
```

```cpp
    halo::CloudPtr second(new halo::PointCloudType);

    pcl::io::loadPCDFile(first_scan_path, *first);
    pcl::io::loadPCDFile(second_scan_path, *second);

    // Use the second cloud as the query set.
    halo::CloudPtr test_cloud = second;
    std::vector<halo::NNMatch> matches;
    std::vector<halo::NNMatch> ground_truth_matches =
        halo::brute_force_nn(first, test_cloud, true);

    {
        std::cout << "=====================NanoFlann Case0: k = 1 for Nanoflann KD Tree=====
        halo::RAIITimer timer;
        halo::NanoflannPointCloudAdaptor adaptor(*first);
        halo::NanoFlannKDTree<halo::PointType, 3> nano_tree(adaptor,
        nanoflann::KDTreeSingleIndexAdaptorParams(10));
        size_t k = 1;
        nano_tree.search_tree_multi_threaded(test_cloud, matches, k);
        EXPECT_EQ(matches.size(), second->points.size() * k);
    }
    evaluate_matches(matches, ground_truth_matches, 1, first, second);

}
```