LiDAR-Inertial Mapping

Objective

A rosbag with velodyne point clouds and inertial measurements is provided, and the goal is to build a map using those measurements.

Direct pose graph optimization

A naive way to build a map is to directly use the point clouds, and build a pose graph. The node in the graph represents the point cloud Pi from each frame i associated with a transformation Ti in SE(3), which transforms the point cloud Pi from the body frame to the world frame.

The set of poses {Ti} are the variables to be optimized. Due to the limited observability, T0 is set to be the origin of the world frame. An edge in the graph connects two nodes that overlap, whereby each edge contains a transformation Ti,j aligning Pi to Pj. The relative transformations are estimated via Iterative closest point (ICP).

Since the local transformations are prone to drift, global loop-closures are also included in the edges. These edges connect non-neighboring nodes and the transformation is obtained via global registration [1].

After we define the nodes and obtain the edges, we can proceed to do the least squares minimization. This step employs the Levenberg–Marquardt algorithm to minimize the residual of the relative transformations. The pose graph optimization is implemented with Open3D [2]. The optimized point cloud map is shown below.

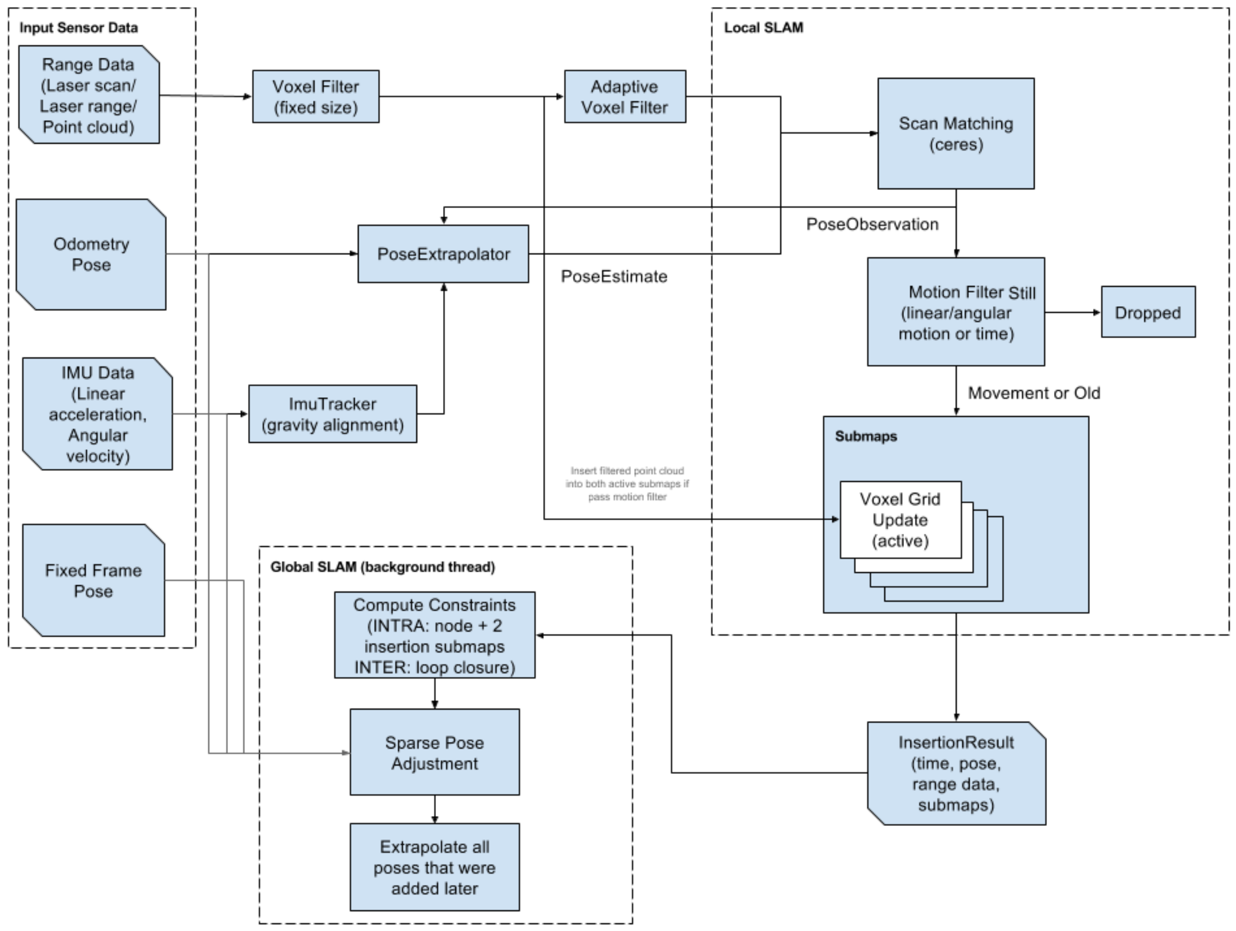
The resulting map does not have any apparent structure, which indicates that this naive pose graph optimization does not work well. There are several reasons for its failure:

- there is no front end to preprocess the raw point clouds, and the raw data contains lots of noise, it is hard to align them,

- inertial data is provided but not used, which undermines the mapping accuracy.

Cartographer

Cartographer [3] is a real-time 2D/3D SLAM system developed by Google, which supports the fusion of heterogeneous sensor data such as LiDAR, IMU, GPS, etc. Here is an overview of the framework:


Input Sensor Data
Range Data (Laser scan/ Laser range/ Point cloud)
Odometry Pose
IMU Data (Linear acceleration, Angular velocity)
Fixed Frame Pose
Voxel Filter (fixed size)
Adaptive Voxel Filter
PoseExtrapolator
ImuTracker (gravity alignment)
PoseEstimate
PoseObservation
Local SLAM
Scan Matching (ceres)
Motion Filter Still (linear/angular motion or time)
Dropped
Movement or Old
Submaps
Voxel Grid Update (active)
Insert filtered point cloud into both active submaps if pass motion filter
Global SLAM (background thread)
Compute Constraints (INTRA: node + 2 insertion submaps INTER: loop closure)
Sparse Pose Adjustment
Extrapolate all poses that were added later
InsertionResult (time, pose, range data, submaps)

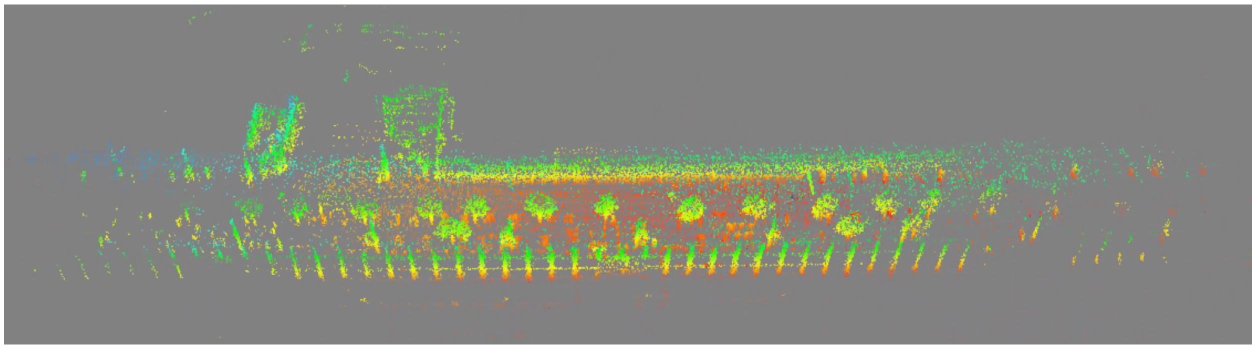
Cartographer consists of two subsystems, one is the local SLAM for building a succession of submaps, whereas the other is the global SLAM to detect loop closure via scan-matching. An IMU is required for 3D global SLAM to provide an initial orientation with respect to gravity, such that the search space of scan-matching can be reduced. The global SLAM involves pose graph optimization, similar to the naive approach described above. For efficiency, branch and bound is used when a node and a submap are considered for building a constraint.

To make sure both local SLAM and global SLAM work properly, it is vital to visualize the point clouds in rviz in real time. The optimized variables are stored in the backend in `PoseGraphData` defined in `cartographer/cartographer/mapping/internal/pose_graph_data.h`, where `trajectory_nodes` in `cartographer/cartographer/mapping/trajectory_node.h` stores the `transform::Rigid3d global_pose`, which is the node pose in the global SLAM frame. The API that returns this data structure is `mapping::MapById<mapping::NodeId, mapping::TrajectoryNode> PoseGraphStub::GetTrajectoryNodes()`. We can then use the pose to transform the local point cloud map to global frame.

The ROS wrapper retrieves the local maps using `MapBuilderBridge` defined in `cartographer_ros/cartographer_ros/cartographer_ros/map_builder_bridge.h`, so we need to add `map_builder_->pose_graph()->GetTrajectoryNodes()`. Then in `cartographer_ros/cartographer_ros/cartographer_ros/node.h`, add `::ros::Publisher point_cloud_map_publisher_;` for publishing the `high_resolution_point_cloud`. Eventually, the final point cloud map is saved via `pcl::fromROSMsg(ros_point_cloud_map_, *pcl_point_cloud_map);` using the PCL library, defined in `bool Node::HandleWriteState`.

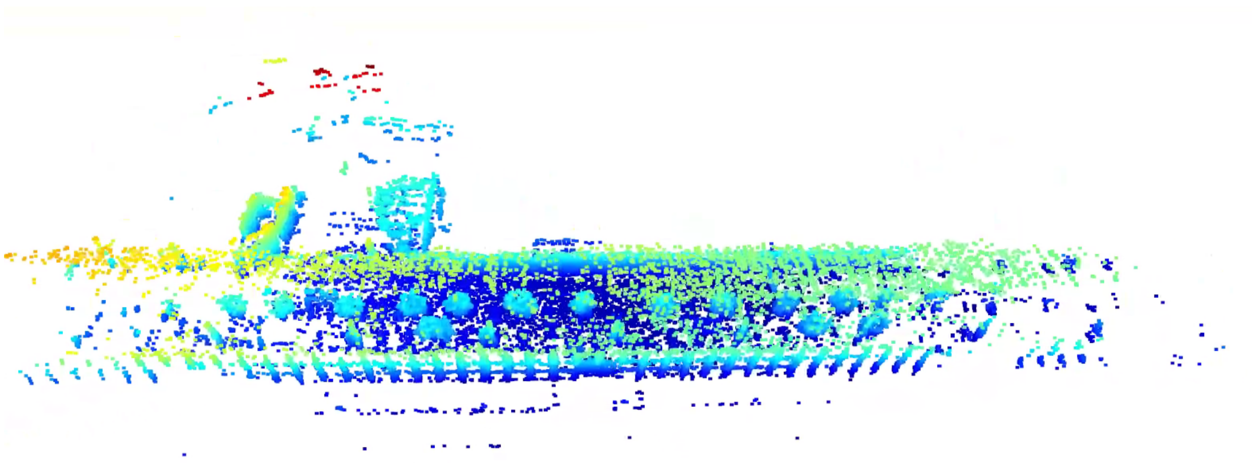
Results

- A demo video for the mapping process can be found at https://youtu.be/B6nrmXvqAuo, it shows how the map is built gradually, and also the final optimization to correct the drift.
- Point cloud map in rviz



In the visualization, the trees and the corridor structure are quite visible, demonstrating a consistent map.

- Point cloud map in Open3D [2]



We can save the map in pcd format and use Open3D library, a popular tool for processing RGB-D camera data, to visualize it. Similar as the rviz map, this one also clearly shows the trees and the corridor.

References

- [1] TEASER: Fast and Certifiable Point Cloud Registration
- [2] Open3D – A Modern Library for 3D Data Processing
- [3] Real-Time Loop Closure in 2D LIDAR SLAM