Cartographer ROS Documentation

The Cartographer Authors

Oct 27, 2022

Contents

Cartographer is a system that provides real-time simultaneous localization and mapping (SLAM) in 2D and 3D across multiple platforms and sensor configurations. This project provides Cartographer's ROS integration.

CHAPTER 1

Compiling Cartographer ROS

1.1 System Requirements

The Cartographer ROS requirements are the same as the ones from Cartographer.

The following ROS distributions are currently supported:

- Melodic
- Noetic

1.2 Building & Installation

In order to build Cartographer ROS, we recommend using wstool and rosdep. For faster builds, we also recommend using Ninja.

On Ubuntu Focal with ROS Noetic use these commands to install the above tools:

```
sudo apt-get update
sudo apt-get install -y python3-wstool python3-rosdep ninja-build stow
```

On older distributions:

```
sudo apt-get update
sudo apt-get install -y python-wstool python-rosdep ninja-build stow
```

After the tools are installed, create a new cartographer_ros workspace in 'catkin_ws'.

```
mkdir catkin_ws
cd catkin_ws
wstool init src
wstool merge -t src https://raw.githubusercontent.com/cartographer-project/
↪cartographer_ros/master/cartographer_ros.rosinstall
wstool update -t src
```

Now you need to install cartographer_ros' dependencies. First, we use `rosdep` to install the required packages. The command 'sudo rosdep init' will print an error if you have already executed it since installing ROS. This error can be ignored.

```
sudo rosdep init
rosdep update
rosdep install --from-paths src --ignore-src --rosdistro=${ROS_DISTRO} -y
```

Cartographer uses the abseil-cpp library that needs to be manually installed using this script:

```
src/cartographer/scripts/install_abseil.sh
```

Due to conflicting versions you might need to uninstall the ROS abseil-cpp using

```
sudo apt-get remove ros-${ROS_DISTRO}-abseil-cpp
```

Build and install.

```
catkin_make_isolated --install --use-ninja
```

This builds Cartographer from the latest HEAD of the master branch. If you want a specific version, you need to change the version in the cartographer_ros.rosinstall.

CHAPTER 2

Running Cartographer ROS on a demo bag

Now that Cartographer and Cartographer's ROS integration are installed, you can download example bags (e.g. 2D and 3D backpack collections of the Deutsches Museum) to a known location, in this case `~/Downloads`, and use `roslaunch` to bring up the demo.

The launch files will bring up `roscore` and `rviz` automatically.

Warning: When you want to run cartographer_ros, you might need to source your ROS environment by running `source install_isolated/setup.bash` first (replace bash with zsh if your shell is zsh)

2.1 Deutsches Museum

Download and launch the 2D backpack demo:

```
wget -P ~/Downloads https://storage.googleapis.com/cartographer-public-data/bags/
↪backpack_2d/cartographer_paper_deutsches_museum.bag
roslaunch cartographer_ros demo_backpack_2d.launch bag_filename:=${HOME}/Downloads/
↪cartographer_paper_deutsches_museum.bag
```

Download and launch the 3D backpack demo:

```
wget -P ~/Downloads https://storage.googleapis.com/cartographer-public-data/bags/
↪backpack_3d/with_intensities/b3-2016-04-05-14-14-00.bag
roslaunch cartographer_ros demo_backpack_3d.launch bag_filename:=${HOME}/Downloads/b3-
↪2016-04-05-14-14-00.bag
```

2.2 Pure localization

Pure localization uses 2 different bags. The first one is used to generate the map, the second to run pure localization.

Download the 2D bags from the Deutsche Museum:

```
wget -P ~/Downloads https://storage.googleapis.com/cartographer-public-data/bags/
↪backpack_2d/b2-2016-04-05-14-44-52.bag
wget -P ~/Downloads https://storage.googleapis.com/cartographer-public-data/bags/
↪backpack_2d/b2-2016-04-27-12-31-41.bag
```

Generate the map (wait until cartographer_offline_node finishes) and then run pure localization:

```
roslaunch cartographer_ros offline_backpack_2d.launch bag_filenames:=${HOME}/
↪Downloads/b2-2016-04-05-14-44-52.bag
roslaunch cartographer_ros demo_backpack_2d_localization.launch \
   load_state_filename:=${HOME}/Downloads/b2-2016-04-05-14-44-52.bag.pbstream \
   bag_filename:=${HOME}/Downloads/b2-2016-04-27-12-31-41.bag
```

Download the 3D bags from the Deutsche Museum:

```
wget -P ~/Downloads https://storage.googleapis.com/cartographer-public-data/bags/
↪backpack_3d/b3-2016-04-05-13-54-42.bag
wget -P ~/Downloads https://storage.googleapis.com/cartographer-public-data/bags/
↪backpack_3d/b3-2016-04-05-15-52-20.bag
```

Generate the map (wait until cartographer_offline_node finishes) and then run pure localization:

```
roslaunch cartographer_ros offline_backpack_3d.launch bag_filenames:=${HOME}/
↪Downloads/b3-2016-04-05-13-54-42.bag
roslaunch cartographer_ros demo_backpack_3d_localization.launch \
   load_state_filename:=${HOME}/Downloads/b3-2016-04-05-13-54-42.bag.pbstream \
   bag_filename:=${HOME}/Downloads/b3-2016-04-05-15-52-20.bag
```

2.3 Static landmarks

```
# Download the landmarks example bag.
wget -P ~/Downloads https://storage.googleapis.com/cartographer-public-data/
↪bags/mir/landmarks_demo_uncalibrated.bag

# Launch the landmarks demo.
roslaunch cartographer_mir offline_mir_100_rviz.launch bag_filename:=${HOME}/
↪Downloads/landmarks_demo_uncalibrated.bag
```

2.4 Revo LDS

Download and launch an example bag captured from a low-cost Revo Laser Distance Sensor from Neato Robotics vacuum cleaners:

```
wget -P ~/Downloads https://storage.googleapis.com/cartographer-public-data/bags/revo_
↪lds/cartographer_paper_revo_lds.bag
roslaunch cartographer_ros demo_revo_lds.launch bag_filename:=${HOME}/Downloads/
↪cartographer_paper_revo_lds.bag
```

2.5 PR2

Download and launch an example bag captured from a PR2 R&D humanoid robot from Willow Garage:

```
wget -P ~/Downloads https://storage.googleapis.com/cartographer-public-data/bags/pr2/
↪2011-09-15-08-32-46.bag
roslaunch cartographer_ros demo_pr2.launch bag_filename:=${HOME}/Downloads/2011-09-15-
↪08-32-46.bag
```

2.6 Taurob Tracker

Download and launch an example bag captured from a Taurob Tracker teleoperation robot:

```
wget -P ~/Downloads https://storage.googleapis.com/cartographer-public-data/bags/
↪taurob_tracker/taurob_tracker_simulation.bag
roslaunch cartographer_ros demo_taurob_tracker.launch bag_filename:=${HOME}/Downloads/
↪taurob_tracker_simulation.bag
```

CHAPTER 3

Running Cartographer ROS on your own bag

Now that you've run Cartographer ROS on a couple of provided bags, you can go ahead and make Cartographer work with your own data. Find a `.bag` recording you would like to use for SLAM and go through this tutorial.

Warning: When you want to run cartographer_ros, you might need to source your ROS environment by running `source install_isolated/setup.bash` first (replace bash with zsh if your shell is zsh)

3.1 Validate your bag

Cartographer ROS provides a tool named `cartographer_rosbag_validate` to automatically analyze data present in your bag. It is generally a good idea to run this tool before trying to tune Cartographer for incorrect data.

It benefits from the experience of the Cartographer authors and can detect a variety of mistakes commonly found in bags. For instance, if a `sensor_msgs/Imu` topic is detected, the tool will make sure that the gravity vector has not been removed from the IMU measurements because the gravity norm is used by Cartographer to determine the direction of the ground.

The tool can also provide tips on how to improve the quality of your data. For example, with a Velodyne LIDAR, it is recommended to have one `sensor_msgs/PointCloud2` message per UDP packet sent by the sensor instead of one message per revolution. With that granularity, Cartographer is then able to unwarp the point clouds deformation caused by the robot's motion and results in better reconstruction.

If you have sourced your Cartographer ROS environment, you can simply run the tool like this:

```
cartographer_rosbag_validate -bag_filename your_bag.bag
```

3.2 Create a .lua configuration

Cartographer is highly flexible and can be configured to work on a variety of robots. The robot configuration is read from a `options` data structure that must be defined from a Lua script. The example configurations

are defined in `src/cartographer_ros/cartographer_ros/configuration_files` and installed in `install_isolated/share/cartographer_ros/configuration_files/`.

Note: Ideally, a .lua configuration should be robot-specific and not bag-specific.

You can start by copying one of the example and then adapt it to your own need. If you want to use 3D SLAM:

```
cp install_isolated/share/cartographer_ros/configuration_files/backpack_3d.lua␣
↪install_isolated/share/cartographer_ros/configuration_files/my_robot.lua
```

If you want to use 2D SLAM:

```
cp install_isolated/share/cartographer_ros/configuration_files/backpack_2d.lua␣
↪install_isolated/share/cartographer_ros/configuration_files/my_robot.lua
```

You can then edit `my_robot.lua` to suit the needs of your robot. The values defined in the `options` block define how the Cartographer ROS frontend should interface with your bag. The values defined after the `options` paragraph are used to tune the inner-working of Cartographer, we will ignore these for now.

See also:

The reference documentation of the Cartographer ROS configuration values and of the Cartographer configuration values.

Among the values you need to adapt, you probably have to provide the TF frame IDs of your environment and robot in `map_frame`, `tracking_frame`, `published_frame` and `odom_frame`.

Note: You can either distribute your robot's TF tree from a `/tf` topic in your bag or define it in a `.urdf` robot definition.

Warning: You should trust your poses! A small offset on the link between your robot and IMU or LIDAR can lead to incoherent map reconstructions. Cartographer can usually correct small pose errors but not everything!

The other values you need to define are related to the number and type of sensors you would like to use.

- `num_laser_scans`: Number of `sensor_msgs/LaserScan` topics you'll use.
- `num_multi_echo_laser_scans`: Number of `sensor_msgs/MultiEchoLaserScan` topics you'll use.
- `num_point_clouds`: Number of `sensor_msgs/PointCloud2` topics you'll use.

You can also enable the usage of landmarks and GPS as additional sources of localization using `use_landmarks` and `use_nav_sat`. The rest of the variables in the `options` block should typically be left untouched.

Note: even if you use a 2D SLAM, the landmarks are 3D objects and can mislead you if viewed only on the 2D plane due to their third dimension.

However, there is one global variable that you absolutely need to adapt to the needs of your bag: `TRAJECTORY_BUILDER_3D.num_accumulated_range_data` or `TRAJECTORY_BUILDER_2D.num_accumulated_range_data`. This variable defines the number of messages required to construct a full scan (typically, a full revolution). If you follow `cartographer_rosbag_validate`'s advices and use 100 ROS

messages per scan, you can set this variable to 100. If you have two range finding sensors (for instance, two LIDARs) providing their full scans all at once, you should set this variable to 2.

3.3 Create .launch files for your SLAM scenarios

You may have noticed that each demo introduced in the previous section was run with a different roslaunch command. The recommended usage of Cartographer is indeed to provide a custom `.launch` file per robot and type of SLAM. The example `.launch` files are defined in `src/cartographer_ros/cartographer_ros/launch` and installed in `install_isolated/share/cartographer_ros/launch/`.

Start by copying one of the provided example:

```
cp install_isolated/share/cartographer_ros/launch/backpack_3d.launch install_isolated/
↪share/cartographer_ros/launch/my_robot.launch
cp install_isolated/share/cartographer_ros/launch/demo_backpack_3d.launch install_
↪isolated/share/cartographer_ros/launch/demo_my_robot.launch
cp install_isolated/share/cartographer_ros/launch/offline_backpack_3d.launch install_
↪isolated/share/cartographer_ros/launch/offline_my_robot.launch
cp install_isolated/share/cartographer_ros/launch/demo_backpack_3d_localization.
↪launch install_isolated/share/cartographer_ros/launch/demo_my_robot_localization.
↪launch
cp install_isolated/share/cartographer_ros/launch/assets_writer_backpack_3d.launch␣
↪install_isolated/share/cartographer_ros/launch/assets_writer_my_robot.launch
```

- `my_robot.launch` is meant to be used on the robot to execute SLAM online (in real time) with real sensors data.
- `demo_my_robot.launch` is meant to be used from a development machine and expects a `bag_filename` argument to replay data from a recording. This launch file also spawns a rviz window configured to visualize Cartographer's state.
- `offline_my_robot.launch` is very similar to `demo_my_robot.launch` but tries to execute SLAM as fast as possible. This can make map building significantly faster. This launch file can also use multiple bag files provided to the `bag_filenames` argument.
- `demo_my_robot_localization.launch` is very similar to `demo_my_robot.launch` but expects a `load_state_filename` argument pointing to a `.pbstream` recording of a previous Cartographer execution. The previous recording will be used as a pre-computed map and Cartographer will only perform localization on this map.
- `assets_writer_my_robot.launch` is used to extract data out of a `.pbstream` recording of a previous Cartographer execution.

Again, a few adaptations need to be made to those files to suit your robot.

- Every parameter given to `-configuration_basename` should be adapted to point to `my_robot.lua`.
- If you decided to use a `.urdf` description of your robot, you should place your description in `install_isolated/share/cartographer_ros/urdf` and adapt the `robot_description` parameter to point to your file name.
- If you decided to use `/tf` messages, you can remove the `robot_description` parameter, the `robot_state_publisher` node and the lines statring with `-urdf`.
- If the topic names published by your bag or sensors don't match the ones expected by Cartographer ROS, you can use `<remap>` elements to redirect your topics. The expected topic names depend on the type of range finding devices you use.

Note:

- The IMU topic is expected to be named "imu"
- If you use only one `sensor_msgs/LaserScan` topic, it is expected to be named `scan`. If you have more, they should be named `scan_1`, `scan_2` etc...
- If you use only one `sensor_msgs/MultiEchoLaserScan` topic, it is expected to be named `echoes`. If you have more, they should be named `echoes_1`, `echoes_2` etc...
- If you use only one `sensor_msgs/PointCloud2` topic, it is expected be named `points2`. If you have more, they should be named `points2_1`, `points2_2`, etc...

3.4 Try your configuration

Everything is setup! You can now start Cartographer with:

```
roslaunch cartographer_ros my_robot.launch bag_filename:=/path/to/your_bag.bag
```

If you are lucky enough, everything should already work as expected. However, you might have some problems that require tuning.

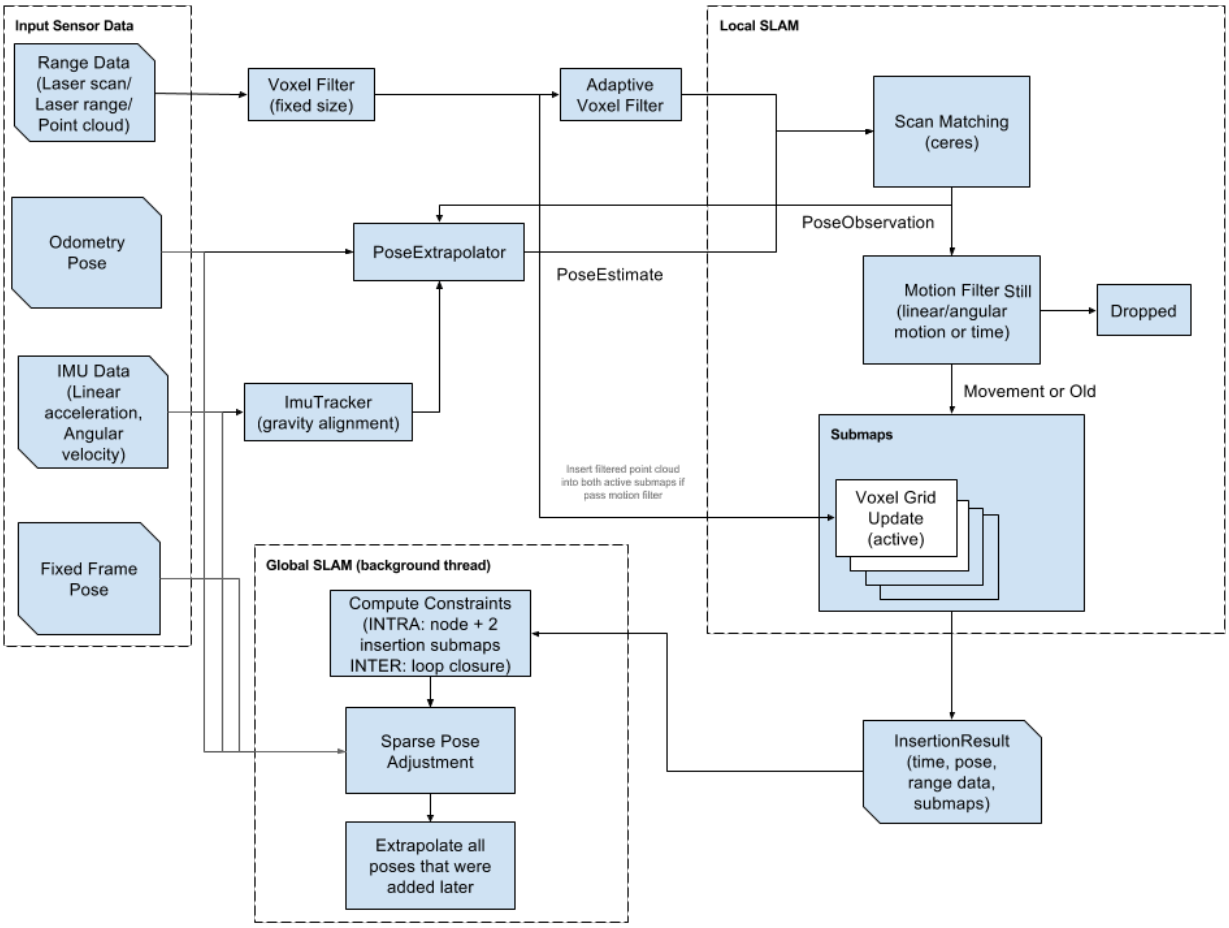
CHAPTER 4

Algorithm walkthrough for tuning

Cartographer is a complex system and tuning it requires a good understanding of its inner working. This page tries to give an intuitive overview of the different subsystems used by Cartographer along with their configuration values. If you are interested in more than an introduction to Cartographer, you should refer to the Cartographer paper. It only describes the 2D SLAM but it defines rigourously most of the concepts described here. Those concepts generally apply to 3D as well.

W. Hess, D. Kohler, H. Rapp, and D. Andor, Real-Time Loop Closure in 2D LIDAR SLAM, in *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE, 2016. pp. 1271–1278.

4.1 Overview



Cartographer can be seen as two separate, but related subsystems. The first one is **local SLAM** (sometimes also called **frontend** or local trajectory builder). Its job is to build a succession of **submaps**. Each submap is meant to be locally consistent but we accept that local SLAM drifts over time. Most of the local SLAM options can be found in install_isolated/share/cartographer/configuration_files/trajectory_builder_2d.lua for 2D and install_isolated/share/cartographer/configuration_files/trajectory_builder_3d.lua for 3D. (for the rest of this page we will refer to *TRAJECTORY_BUILDER_nD* for the common options)

The other subsystem is **global SLAM** (sometimes called the **backend**). It runs in background threads and its main job is to find **loop closure constraints**. It does that by scan-matching **scans** (gathered in **nodes**) against submaps. It also incorporates other sensor data to get a higher level view and identify the most consistent global solution. In 3D, it also tries to find the direction of gravity. Most of its options can be found in install_isolated/share/cartographer/configuration_files/pose_graph.lua

On a higher abstraction, the job of local SLAM is to generate good submaps and the job of global SLAM is to tie them most consistently together.

4.2 Input

Range finding sensors (for example: LIDARs) provide depth information in multiple directions. However, some of the measurements are irrelevant for SLAM. If the sensor is partially covered with dust or if it is directed towards a part

of the robot, some of the measured distance can be considered as noise for SLAM. On the other hand, some of the furthest measurements can also come from undesired sources (reflection, sensor noise) and are irrelevant for SLAM as well. To tackle those issue, Cartographer starts by applying a bandpass filter and only keeps range values between a certain min and max range. Those min and max values should be chosen according to the specifications of your robot and sensors.

```
TRAJECTORY_BUILDER_nD.min_range
TRAJECTORY_BUILDER_nD.max_range
```

Note: In 2D, Cartographer replaces ranges further than max_range by `TRAJECTORY_BUILDER_2D.missing_data_ray_length`. It also provides a `max_z` and `min_z` values to filter 3D point clouds into a 2D cut.

Note: In Cartographer configuration files, every distance is defined in meters

Distances are measured over a certain period of time, while the robot is actually moving. However, distances are delivered by sensors "in batch" in large ROS messages. Each of the messages' timestamp can be considered independently by Cartographer to take into account deformations caused by the robot's motion. The more often Cartographer gets measurements, the better it becomes at unwarping the measurements to assemble a single coherent scan that could have been captured instantly. It is therefore strongly encouraged to provide as many range data (ROS messages) by scan (a set of range data that can be matched against another scan) as possible.

```
TRAJECTORY_BUILDER_nD.num_accumulated_range_data
```

Range data is typically measured from a single point on the robot but in multiple angles. This means that close surfaces (for instance the road) are very often hit and provide lots of points. On the opposite, far objects are less often hit and offer less points. In order to reduce the computational weight of points handling, we usually need to subsample point clouds. However, a simple random sampling would remove points from areas where we already have a low density of measurements and the high-density areas would still have more points than needed. To address that density problem, we can use a voxel filter that downsamples raw points into cubes of a constant size and only keeps the centroid of each cube.

A small cube size will result in a more dense data representation, causing more computations. A large cube size will result in a data loss but will be much quicker.

```
TRAJECTORY_BUILDER_nD.voxel_filter_size
```

After having applied a fixed-size voxel filter, Cartographer also applies an **adaptive voxel filter**. This filter tries to determine the optimal voxel size (under a max length) to achieve a target number of points. In 3D, two adaptive voxel filters are used to generate a high resolution and a low resolution point clouds, their usage will be clarified in *Local SLAM*.

```
TRAJECTORY_BUILDER_nD.*adaptive_voxel_filter.max_length
TRAJECTORY_BUILDER_nD.*adaptive_voxel_filter.min_num_points
```

An Inertial Measurement Unit can be an useful source of information for SLAM because it provides an accurate direction of gravity (hence, of the ground) and a noisy but good overall indication of the robot's rotation. In order to filter the IMU noise, gravity is observed over a certain amount of time. If you use 2D SLAM, range data can be handled in real-time without an additional source of information so you can choose whether you'd like Cartographer to use an IMU or not. With 3D SLAM, you need to provide an IMU because it is used as an initial guess for the orientation of the scans, greatly reducing the complexity of scan matching.

```
TRAJECTORY_BUILDER_2D.use_imu_data
TRAJECTORY_BUILDER_nD.imu_gravity_time_constant
```

Note: In Cartographer configuration files, every time value is defined in seconds

4.3 Local SLAM

Once a scan has been assembled and filtered from multiple range data, it is ready for the local SLAM algorithm. Local SLAM inserts a new scan into its current submap construction by **scan matching** using an initial guess from the **pose extrapolator**. The idea behind the pose extrapolator is to use sensor data of other sensors besides the range finder to predict where the next scan should be inserted into the submap.

Two scan matching strategies are available:

- The `CeresScanMatcher` takes the initial guess as prior and finds the best spot where the scan match fits the submap. It does this by interpolating the submap and sub-pixel aligning the scan. This is fast, but cannot fix errors that are significantly larger than the resolution of the submaps. If your sensor setup and timing is reasonable, using only the `CeresScanMatcher` is usually the best choice to make.
- The `RealTimeCorrelativeScanMatcher` can be enabled if you do not have other sensors or you do not trust them. It uses an approach similar to how scans are matched against submaps in loop closure (described later), but instead it matches against the current submap. The best match is then used as prior for the `CeresScanMatcher`. This scan matcher is very expensive and will essentially override any signal from other sensors but the range finder, but it is robust in feature rich environments.

Either way, the `CeresScanMatcher` can be configured to give a certain weight to each of its input. The weight is a measure of trust into your data, this can be seen as a static covariance. The unit of weight parameters are dimensionless quantities and can't be compared between each others. The bigger the weight of a source of data is, the more emphasis Cartographer will put on this source of data when doing scan matching. Sources of data include occupied space (points from the scan), translation and rotation from the pose extrapolator (or `RealTimeCorrelativeScanMatcher`)

```
TRAJECTORY_BUILDER_2D.ceres_scan_matcher.occupied_space_weight
TRAJECTORY_BUILDER_3D.ceres_scan_matcher.occupied_space_weight_0
TRAJECTORY_BUILDER_3D.ceres_scan_matcher.occupied_space_weight_1
TRAJECTORY_BUILDER_nD.ceres_scan_matcher.translation_weight
TRAJECTORY_BUILDER_nD.ceres_scan_matcher.rotation_weight
```

Note: In 3D, the `occupied_space_weight_0` and `occupied_space_weight_1` parameters are related, respectively, to the high resolution and low resolution filtered point clouds.

The `CeresScanMatcher` gets its name from Ceres Solver, a library developed at Google to solve non-linear least squares problems. The scan matching problem is modelled as the minimization of such a problem with the **motion** (a transformation matrix) between two scans being a parameter to determine. Ceres optimizes the motion using a descent algorithm for a given number of iterations. Ceres can be configured to adapt the convergence speed to your own needs.

```
TRAJECTORY_BUILDER_nD.ceres_scan_matcher.ceres_solver_options.use_nonmonotonic_steps
TRAJECTORY_BUILDER_nD.ceres_scan_matcher.ceres_solver_options.max_num_iterations
TRAJECTORY_BUILDER_nD.ceres_scan_matcher.ceres_solver_options.num_threads
```

The `RealTimeCorrelativeScanMatcher` can be toggled depending on the trust you have in your sensors. It works by searching for similar scans in a **search window** which is defined by a maximum distance radius and a

maximum angle radius. When performing scan matching with scans found in this window, a different weight can be chosen for the translational and rotational components. You can play with those weight if, for example, you know that your robot doesn't rotate a lot.

```
TRAJECTORY_BUILDER_nD.use_online_correlative_scan_matching
TRAJECTORY_BUILDER_nD.real_time_correlative_scan_matcher.linear_search_window
TRAJECTORY_BUILDER_nD.real_time_correlative_scan_matcher.angular_search_window
TRAJECTORY_BUILDER_nD.real_time_correlative_scan_matcher.translation_delta_cost_weight
TRAJECTORY_BUILDER_nD.real_time_correlative_scan_matcher.rotation_delta_cost_weight
```

To avoid inserting too many scans per submaps, once a motion between two scans is found by the scan matcher, it goes through a **motion filter**. A scan is dropped if the motion that led to it is not considered as significant enough. A scan is inserted into the current submap only if its motion is above a certain distance, angle or time threshold.

```
TRAJECTORY_BUILDER_nD.motion_filter.max_time_seconds
TRAJECTORY_BUILDER_nD.motion_filter.max_distance_meters
TRAJECTORY_BUILDER_nD.motion_filter.max_angle_radians
```

A submap is considered as complete when the local SLAM has received a given amount of range data. Local SLAM drifts over time, global SLAM is used to fix this drift. Submaps must be small enough so that the drift inside them is below the resolution, so that they are locally correct. On the other hand, they should be large enough to be distinct for loop closure to work properly.

```
TRAJECTORY_BUILDER_nD.submaps.num_range_data
```

Submaps can store their range data in a couple of different data structures: The most widely used representation is called probability grids. However, in 2D, one can also choose to use Truncated Signed Distance Fields (TSDF).

```
TRAJECTORY_BUILDER_2D.submaps.grid_options_2d.grid_type
```

Probability grids cut out space into a 2D or 3D table where each cell has a fixed size and contains the odds of being obstructed. Odds are updated according to "*hits*" (where the range data is measured) and "*misses*" (the free space between the sensor and the measured points). Both *hits* and *misses* can have a different weight in occupancy probability calculations giving more or less trust to occupied or free space measurements.

```
TRAJECTORY_BUILDER_2D.submaps.range_data_inserter.probability_grid_range_data_
↪inserter.hit_probability
TRAJECTORY_BUILDER_2D.submaps.range_data_inserter.probability_grid_range_data_
↪inserter.miss_probability
TRAJECTORY_BUILDER_3D.submaps.range_data_inserter.hit_probability
TRAJECTORY_BUILDER_3D.submaps.range_data_inserter.miss_probability
```

In 2D, only one probability grid per submap is stored. In 3D, for scan matching performance reasons, two *hybrid* probability grids are used. (the term "hybrid" only refers to an internal tree-like data representation and is abstracted to the user)

- a low resolution hybrid grid for far measurements
- a high resolution hybrid grid for close measurements

Scan matching starts by aligning far points of the low resolution point cloud with the low resolution hybrid grid and then refines the pose by aligning the close high resolution points with the high resolution hybrid grid.

```
TRAJECTORY_BUILDER_2D.submaps.grid_options_2d.resolution
TRAJECTORY_BUILDER_3D.submaps.high_resolution
TRAJECTORY_BUILDER_3D.submaps.low_resolution
TRAJECTORY_BUILDER_3D.high_resolution_adaptive_voxel_filter.max_range
TRAJECTORY_BUILDER_3D.low_resolution_adaptive_voxel_filter.max_range
```

Note: Cartographer ROS provides an RViz plugin to visualize submaps. You can select the submaps you want to see from their number. In 3D, RViz only shows 2D projections of the 3D hybrid probability grids (in grayscale). Options are made available in RViz's left pane to switch between the low and high resolution hybrid grids visualization.

TODO: *Documenting TSDF configuration*

4.4 Global SLAM

While the local SLAM generates its succession of submaps, a global optimization (usually referred to as "*the optimization problem*" or "*sparse pose adjustment*") task runs in background. Its role is to re-arrange submaps between each other so that they form a coherent global map. For instance, this optimization is in charge of altering the currently built trajectory to properly align submaps with regards to loop closures.

The optimization is run in batches once a certain number of trajectory nodes was inserted. Depending on how frequently you need to run it, you can tune the size of these batches.

```
POSE_GRAPH.optimize_every_n_nodes
```

Note: Setting POSE_GRAPH.optimize_every_n_nodes to 0 is a handy way to disable global SLAM and concentrate on the behavior of local SLAM. This is usually one of the first thing to do to tune Cartographer.

The global SLAM is a kind of "*GraphSLAM*", it is essentially a pose graph optimization which works by building **constraints** between **nodes** and submaps and then optimizing the resulting constraints graph. Constraints can intuitively be thought of as little ropes tying all nodes together. The sparse pose adjustment fastens those ropes altogether. The resulting net is called the "*pose graph*".

Note: Constraints can be visualized in RViz, it is very handy to tune global SLAM. One can also toggle `POSE_GRAPH.constraint_builder.log_matches` to get regular reports of the constraints builder formatted as histograms.

- Non-global constraints (also known as intra submaps constraints) are built automatically between nodes that are closely following each other on a trajectory. Intuitively, those "*non-global ropes*" keep the local structure of the trajectory coherent.
- Global constraints (also referred to as loop closure constraints or inter submaps constraints) are regularly searched between a new submap and previous nodes that are considered "*close enough*" in space (part of a certain **search window**) and a strong fit (a good match when running scan matching). Intuitively, those "*global ropes*" introduce knots in the structure and firmly bring two strands closer.

```
POSE_GRAPH.constraint_builder.max_constraint_distance
POSE_GRAPH.fast_correlative_scan_matcher.linear_search_window
POSE_GRAPH.fast_correlative_scan_matcher_3d.linear_xy_search_window
POSE_GRAPH.fast_correlative_scan_matcher_3d.linear_z_search_window
POSE_GRAPH.fast_correlative_scan_matcher*.angular_search_window
```

Note: In practice, global constraints can do more than finding loop closures on a single trajectory. They can also align different trajectories recorded by multiple robots but we will keep this usage and the parameters related to "global localization" out of the scope of this document.

To limit the amount of constraints (and computations), Cartographer only considers a subsampled set of all close nodes for constraints building. This is controlled by a sampling ratio constant. Sampling too few nodes could result in missed constraints and ineffective loop closures. Sampling too many nodes would slow the global SLAM down and prevent real-time loop closures.

```
POSE_GRAPH.constraint_builder.sampling_ratio
```

When a node and a submap are considered for constraint building, they go through a first scan matcher called the `FastCorrelativeScanMatcher`. This scan matcher has been specifically designed for Cartographer and makes real-time loop closures scan matching possible. The `FastCorrelativeScanMatcher` relies on a "*Branch and bound*" mechanism to work at different grid resolutions and efficiently eliminate incorrect matchings. This mechanism is extensively presented in the Cartographer paper presented earlier in this document. It works on an exploration tree whose depth can be controlled.

```
POSE_GRAPH.constraint_builder.fast_correlative_scan_matcher.branch_and_bound_depth
POSE_GRAPH.constraint_builder.fast_correlative_scan_matcher_3d.branch_and_bound_depth
POSE_GRAPH.constraint_builder.fast_correlative_scan_matcher_3d.full_resolution_depth
```

Once the `FastCorrelativeScanMatcher` has a good enough proposal (above a minimum score of matching), it is then fed into a Ceres Scan Matcher to refine the pose.

```
POSE_GRAPH.constraint_builder.min_score
POSE_GRAPH.constraint_builder.ceres_scan_matcher_3d
POSE_GRAPH.constraint_builder.ceres_scan_matcher
```

When Cartographer runs *the optimization problem*, Ceres is used to rearrange submaps according to multiple *residuals*. Residuals are calculated using weighted cost functions. The global optimization has cost functions to take into account plenty of data sources: the global (loop closure) constraints, the non-global (matcher) constraints, the IMU acceleration and rotation measurements, the local SLAM rough pose estimations, an odometry source or a fixed frame (such as a GPS system). The weights and Ceres options can be configured as described in the *Local SLAM* section.

```
POSE_GRAPH.constraint_builder.loop_closure_translation_weight
POSE_GRAPH.constraint_builder.loop_closure_rotation_weight
POSE_GRAPH.matcher_translation_weight
POSE_GRAPH.matcher_rotation_weight
POSE_GRAPH.optimization_problem.*_weight
POSE_GRAPH.optimization_problem.ceres_solver_options
```

Note: One can find useful information about the residuals used in the optimization problem by toggling `POSE_GRAPH.log_residual_histograms`

As part of its IMU residual, the optimization problem gives some flexibility to the IMU pose and, by default, Ceres is free to optimize the extrinsic calibration between your IMU and tracking frame. If you don't trust your IMU pose, the results of Ceres' global optimization can be logged and used to improve your extrinsic calibration. If Ceres doesn't optimize your IMU pose correctly and you trust your extrinsic calibration enough, you can make this pose constant.

```
POSE_GRAPH.optimization_problem.log_solver_summary
POSE_GRAPH.optimization_problem.use_online_imu_extrinsics_in_3d
```

In residuals, the influence of outliers is handled by a **Huber loss** function configured with a certain a Huber scale. The bigger the Huber scale, the higher is the impact of (potential) outliers.

```
POSE_GRAPH.optimization_problem.huber_scale
```

Once the trajectory is finished, Cartographer runs a new global optimization with, typically, a lot more iterations than previous global optimizations. This is done to polish the final result of Cartographer and usually does not need to be real-time so a large number of iterations is often a right choice.

```
POSE_GRAPH.max_num_final_iterations
```

CHAPTER 5

Tuning methodology

Tuning Cartographer is unfortunately really difficult. The system has many parameters many of which affect each other. This tuning guide tries to explain a principled approach on concrete examples.

5.1 Built-in tools

Cartographer provides built-in tools for SLAM evaluation that can be particularly useful for measuring the local SLAM quality. They are stand-alone executables that ship with the core `cartographer` library and are hence independent, but compatible with `cartographer_ros`. Therefore, please head to the Cartographer Read the Docs Evaluation site for a conceptual overview and a guide on how to use the tools in practice.

These tools assume that you have serialized the SLAM state to a `.pbstream` file. With `cartographer_ros`, you can invoke the `assets_writer` to serialize the state - see the *Exploiting the map generated by Cartographer ROS* section for more information.

5.2 Example: tuning local SLAM

For this example we'll start at `cartographer` commit aba4575 and `cartographer_ros` commit 99c23b6 and look at the bag `b2-2016-04-27-12-31-41.bag` from our test data set.

At our starting configuration, we see some slipping pretty early in the bag. The backpack passed over a ramp in the Deutsches Museum which violates the 2D assumption of a flat floor. It is visible in the laser scan data that contradicting information is passed to the SLAM. But the slipping also indicates that we trust the point cloud matching too much and disregard the other sensors quite strongly. Our aim is to improve the situation through tuning.

If we only look at this particular submap, that the error is fully contained in one submap. We also see that over time, global SLAM figures out that something weird happened and partially corrects for it. The broken submap is broken forever though.

Since the problem here is slippage inside a submap, it is a local SLAM issue. So let's turn off global SLAM to not mess with our tuning.

```
POSE_GRAPH.optimize_every_n_nodes = 0
```

5.2.1 Correct size of submaps

The size of submaps is configured through `TRAJECTORY_BUILDER_2D.submaps.num_range_data`. Looking at the individual submaps for this example they already fit the two constraints rather well, so we assume this parameter is well tuned.

5.2.2 Tuning the `CeresScanMatcher`

In our case, the scan matcher can freely move the match forward and backwards without impacting the score. We'd like to penalize this situation by making the scan matcher pay more for deviating from the prior that it got. The two parameters controlling this are `TRAJECTORY_BUILDER_2D.ceres_scan_matcher.translation_weight` and `rotation_weight`. The higher, the more expensive it is to move the result away from the prior, or in other words: scan matching has to generate a higher score in another position to be accepted.

For instructional purposes, let's make deviating from the prior really expensive:

```
TRAJECTORY_BUILDER_2D.ceres_scan_matcher.translation_weight = 1e3
```

This allows the optimizer to pretty liberally overwrite the scan matcher results. This results in poses close to the prior, but inconsistent with the depth sensor and clearly broken. Experimenting with this value yields a better result at `2e2`.

Here, the scan matcher used rotation to still slightly mess up the result though. Setting the `rotation_weight` to `4e2` leaves us with a reasonable result.

5.2.3 Verification

To make sure that we did not overtune for this particular issue, we need to run the configuration against other collected data. In this case, the new parameters did reveal slipping, for example at the beginning of `b2-2016-04-05-14-44-52.bag`, so we had to lower the `translation_weight` to `1e2`. This setting is worse for the case we wanted to fix, but no longer slips. Before checking them in, we normalize all weights, since they only have relative meaning. The result of this tuning was PR 428. In general, always try to tune for a platform, not a particular bag.

5.3 Special Cases

The default configuration and the above tuning steps are focused on quality. Only after we have achieved good quality, we can further consider special cases.

5.3.1 Low Latency

By low latency, we mean that an optimized local pose becomes available shortly after sensor input was received, usually within a second, and that global optimization has no backlog. Low latency is required for online algorithms, such as robot localization. Local SLAM, which operates in the foreground, directly affects latency. Global SLAM builds up a queue of background tasks. When global SLAM cannot keep up the queue, drift can accumulate indefinitely, so global SLAM should be tuned to work in real time.

There are many options to tune the different components for speed, and we list them ordered from the recommended, straightforward ones to the those that are more intrusive. It is recommended to only explore one option at a time, starting with the first. Configuration parameters are documented in the Cartographer documentation.

To tune global SLAM for lower latency, we reduce its computational load until is consistently keeps up with real-time input. Below this threshold, we do not reduce it further, but try to achieve the best possible quality. To reduce global SLAM latency, we can

- decrease `optimize_every_n_nodes`
- increase `MAP_BUILDER.num_background_threads` up to the number of cores
- decrease `global_sampling_ratio`
- decrease `constraint_builder.sampling_ratio`
- increase `constraint_builder.min_score`
- for the adaptive voxel filter(s), decrease `.min_num_points`, `.max_range`, increase `.max_length`
- increase `voxel_filter_size`, `submaps.resolution`, decrease `submaps.num_range_data`
- decrease search windows sizes, `.linear_xy_search_window`, `.linear_z_search_window`, `.angular_search_window`
- increase `global_constraint_search_after_n_seconds`
- decrease `max_num_iterations`

To tune local SLAM for lower latency, we can

- increase `voxel_filter_size`
- increase `submaps.resolution`
- for the adaptive voxel filter(s), decrease `.min_num_points`, `.max_range`, increase `.max_length`
- decrease `max_range` (especially if data is noisy)
- decrease `submaps.num_range_data`

Note that larger voxels will slightly increase scan matching scores as a side effect, so score thresholds should be increased accordingly.

5.3.2 Pure Localization in a Given Map

Pure localization is different from mapping. First, we expect a lower latency of both local and global SLAM. Second, global SLAM will usually find a very large number of inter constraints between the frozen trajectory that serves as a map and the current trajectory.

To tune for pure localization, we should first enable `TRAJECTORY_BUILDER.pure_localization = true` and strongly decrease `POSE_GRAPH.optimize_every_n_nodes` to receive frequent results. With these settings, global SLAM will usually be too slow and cannot keep up. As a next step, we strongly decrease `global_sampling_ratio` and `constraint_builder.sampling_ratio` to compensate for the large number of constraints. We then tune for lower latency as explained above until the system reliably works in real time.

If you run in `pure_localization`, `submaps.resolution` **should be matching** with the resolution of the submaps in the `.pbstream` you are running on. Using different resolutions is currently untested and may not work as expected.

5.3.3 Odometry in Global Optimization

If a separate odometry source is used as an input for local SLAM (`use_odometry = true`), we can also tune the global SLAM to benefit from this additional information.

There are in total four parameters that allow us to tune the individual weights of local SLAM and odometry in the optimization:

```
POSE_GRAPH.optimization_problem.local_slam_pose_translation_weight
POSE_GRAPH.optimization_problem.local_slam_pose_rotation_weight
POSE_GRAPH.optimization_problem.odometry_translation_weight
POSE_GRAPH.optimization_problem.odometry_rotation_weight
```

We can set these weights depending on how much we trust either local SLAM or the odometry. By default, odometry is weighted into global optimization similar to local slam (scan matching) poses. However, odometry from wheel encoders often has a high uncertainty in rotation. In this case, the rotation weight can be reduced, even down to zero.

5.4 Still have a problem ?

If you can't get Cartographer to work reliably on your data, you can open a GitHub issue asking for help. Developers are keen to help, but they can only be helpful if you follow an issue template containing the result of `rosbag_validate`, a link to a fork of `cartographer_ros` with your config and a link to a `.bag` file reproducing your problem.

Note: There are already lots of GitHub issues with all sorts of problems solved by the developers. Going through the closed issues of cartographer_ros and of cartographer is a great way to learn more about Cartographer and maybe find a solution to your problem !

CHAPTER 6

Exploiting the map generated by Cartographer ROS

As sensor data come in, the state of a SLAM algorithm such as Cartographer evolves to stay *the current best estimate* of a robot's trajectory and surroundings. The most accurate localization and mapping Cartographer can offer is therefore the one obtained when the algorithm finishes. Cartographer can serialize its internal state in a `.pbstream` file format which is essentially a compressed protobuf file containing a snapshot of the data structures used by Cartographer internally.

To run efficiently in real-time, Cartographer throws most of its sensor data away immediately and only works with a small subset of its input, the mapping used internally (and saved in `.pbstream` files) is then very rough. However, when the algorithm finishes and a best trajectory is established, it can be recombined *a posteriori* with the full sensors data to create a high resolution map.

Cartographer makes this kind of recombination possible using `cartographer_assets_writer`. The assets writer takes as input

1. the original sensors data that has been used to perform SLAM (in a ROS `.bag` file),
2. a cartographer state captured while performing SLAM on this sensor data (saved in a `.pbstream` file),
3. the sensor extrinsics (i.e. TF data from the bag or an URDF description),
4. and a pipeline configuration, which is defined in a `.lua` file.

The assets writer runs through the `.bag` data in batches with the trajectory found in the `.pbstream`. The pipeline can be used to color, filter and export SLAM point cloud data into a variety of formats. There are multiple of such points processing steps that can be interleaved in a pipeline - several ones are already available from cartographer/io.

6.1 Sample Usage

When running Cartographer with an offline node, a `.pbstream` file is automatically saved. For instance, with the 3D backpack example:

```
wget -P ~/Downloads https://storage.googleapis.com/cartographer-public-data/bags/
↪backpack_3d/b3-2016-04-05-14-14-00.bag
roslaunch cartographer_ros offline_backpack_3d.launch bag_filenames:=${HOME}/
↪Downloads/b3-2016-04-05-14-14-00.bag
```

(continues on next page)

(continued from previous page)

Watch the output on the commandline until the node terminates. It will have written `b3-2016-04-05-14-14-00.bag.pbstream` which represents the Cartographer state after it processed all data and finished all optimizations.

When running as an online node, Cartographer doesn't know when your bag (or sensor input) ends so you need to use the exposed services to explicitly finish the current trajectory and make Cartographer serialize its current state:

```
# Finish the first trajectory. No further data will be accepted on it.
rosservice call /finish_trajectory 0

# Ask Cartographer to serialize its current state.
# (press tab to quickly expand the parameter syntax)
rosservice call /write_state "{filename: '${HOME}/Downloads/b3-2016-04-05-14-14-00.
↪bag.pbstream', include_unfinished_submaps: "true"}"
```

Once you've retrieved your `.pbstream` file, you can run the assets writer with the sample pipeline for the 3D backpack:

```
roslaunch cartographer_ros assets_writer_backpack_3d.launch \
   bag_filenames:=${HOME}/Downloads/b3-2016-04-05-14-14-00.bag \
   pose_graph_filename:=${HOME}/Downloads/b3-2016-04-05-14-14-00.bag.pbstream
```

All output files are prefixed with `--output_file_prefix` which defaults to the filename of the first bag. For the last example, if you specify `points.ply` in the pipeline configuration file, this will translate to `${HOME}/Downloads/b3-2016-04-05-14-14-00.bag_points.ply`.

6.2 Configuration

The assets writer is modeled as a pipeline of PointsProcessor steps. PointsBatch data flows through each processor and they all have the chance to modify the `PointsBatch` before passing it on.

For example the assets_writer_backpack_3d.lua pipeline uses `min_max_range_filter` to remove points that are either too close or too far from the sensor. After this, it saves "*X-Rays*" (translucent side views of the map), then recolors the `PointsBatch`s depending on the sensor frame ids and writes another set of X-Rays using these new colors.

The available `PointsProcessor`s are all defined in the cartographer/io sub-directory and documented in their individual header files.

- **color_points**: Colors points with a fixed color by frame_id.
- **dump_num_points**: Passes through points, but keeps track of how many points it saw and output that on Flush.
- **fixed_ratio_sampler**: Only let a fixed 'sampling_ratio' of points through. A 'sampling_ratio' of 1. makes this filter a no-op.
- **frame_id_filter**: Filters all points with blacklisted frame_id or a non-whitelisted frame id. Note that you can either specify the whitelist or the blacklist, but not both at the same time.
- **write_hybrid_grid**: Creates a hybrid grid of the points with voxels being 'voxel_size' big. 'range_data_inserter' options are used to configure the range data ray tracing through the hybrid grid.
- **intensity_to_color**: Applies ('intensity' - min) / (max - min) * 255 and color the point grey with this value for each point that comes from the sensor with 'frame_id'. If 'frame_id' is empty, this applies to all points.
- **min_max_range_filtering**: Filters all points that are farther away from their 'origin' as 'max_range' or closer than 'min_range'.

- **voxel_filter_and_remove_moving_objects**: Voxel filters the data and only passes on points that we believe are on non-moving objects.
- **write_pcd**: Streams a PCD file to disk. The header is written in 'Flush'.
- **write_ply**: Streams a PLY file to disk. The header is written in 'Flush'.
- **write_probability_grid**: Creates a probability grid with the specified 'resolution'. As all points are projected into the x-y plane the z component of the data is ignored. 'range_data_inserter' options are used to configure the range data ray tracing through the probability grid.
- **write_xray_image**: Creates X-ray cuts through the points with pixels being 'voxel_size' big.
- **write_xyz**: Writes ASCII xyz points.

6.3 First-person visualization of point clouds

Two `PointsProcessors` are of particular interest: `pcd_writing` and `ply_writing` can save a point cloud in a `.pcd` or `.ply` file format. These file formats can then be used by specialized software such as point_cloud_viewer or meshlab to navigate through the high resolution map.

The typical assets writer pipeline for this outcome is composed of an IntensityToColorPointsProcessor giving points a non-white color, then a PlyWritingPointsProcessor exporting the results to a `.ply` point cloud. An example of such a pipeline is in assets_writer_backpack_2d.lua.

Once you have the `.ply`, follow the README of point_cloud_viewer to generate an on-disk octree data structure which can be viewed by one of the viewers (SDL or web based) in the same repo. Note that color is required for `point_cloud_viewer` to function.

CHAPTER 7

Going further

Cartographer is not only a great SLAM algorithm, it also comes with a fully-featured implementation that brings lots of "extra" features. This page lists some of those less known functionalities.

7.1 More input

If you have a source of odometry (such as a wheel encoder) publishing on a `nav_msgs/Odometry` topic and want to use it to improve Cartographer's localization, you can add an input to your `.lua` configuration files:

```
use_odometry = true
```

The messages will be expected on the `odom` topic.

A GPS publishing on a `sensor_msgs/NavSatFix` topic named `fix` can improve the global SLAM:

```
use_nav_sat = true
```

For landmarks publishing on a `cartographer_ros_msgs/LandmarkList` (message defined in cartographer_ros) topic named `landmark`:

```
use_landmarks = true
```

7.2 Localization only

If you have a map you are happy with and want to reduce computations, you can use the localization-only mode of Cartographer which will run SLAM against the existing map and won't build a new one. This is enabled by running `cartographer_node` with a `-load_state_filename` argument and by defining the following line in your lua config:

```
TRAJECTORY_BUILDER.pure_localization_trimmer = {
    max_submaps_to_keep = 3,
}
```

7.3 IMU Calibration

When performing the global optimization, Ceres tries to improve the pose between your IMU and range finding sensors. A well chosen acquisition with lots of loop closure constraints (for instance if your robot goes on a straight line and then back) can improve the quality of those corrections and become a reliable source of pose correction. You can then use Cartographer as part of your calibration process to improve the quality of your robot's extrinsic calibration.

7.4 Multi-trajectories SLAM

Cartographer can perform SLAM from multiple robots emitting data in parallel. The global SLAM is able to detect shared paths and will merge the maps built by the different robots as soon as it becomes possible. This is achieved through the usage of two ROS services `start_trajectory` and `finish_trajectory`. (refer to the ROS API reference documentation for more details on their usage)

7.5 Cloud integration with gRPC

Cartographer is built around Protobuf messages which make it very flexible and interoperable. One of the advantages of that architecture is that it is easy to distribute on machines spread over the Internet. The typical use case would be a fleet of robots navigating on a known map, they could have their SLAM algorithm run on a remote powerful centralized localization server running a multi-trajectories Cartographer instance.

TODO: Instructions on how to get started with a gRPC Cartographer instance

CHAPTER 8

Getting involved

Cartographer is developed in the open and allows anyone to contribute to the project. There are multiple ways to get involved!

If you have question or think you've found an issue in Cartographer, you are welcome to open a GitHub issue.

If you have an idea of a significant change that should be documented and discussed before finding its way into Cartographer, you should submit it as a pull request to the RFCs repository first. Simpler changes can also be discussed in GitHub issues so that developers can help you get things right from the first try.

If you want to contribute code or documentation, this is done through GitHub pull requests. Pull requests need to follow the contribution guidelines.

CHAPTER 9

Lua configuration reference documentation

Note that Cartographer's ROS integration uses tf2, thus all frame IDs are expected to contain only a frame name (lower-case with underscores) and no prefix or slashes. See REP 105 for commonly used coordinate frames.

Note that topic names are given as *base* names (see ROS Names) in Cartographer's ROS integration. This means it is up to the user of the Cartographer node to remap, or put them into a namespace.

The following are Cartographer's ROS integration top-level options, all of which must be specified in the Lua configuration file:

map_frame The ROS frame ID to use for publishing submaps, the parent frame of poses, usually "map".

tracking_frame The ROS frame ID of the frame that is tracked by the SLAM algorithm. If an IMU is used, it should be at its position, although it might be rotated. A common choice is "imu_link".

published_frame The ROS frame ID to use as the child frame for publishing poses. For example "odom" if an "odom" frame is supplied by a different part of the system. In this case the pose of "odom" in the *map_frame* will be published. Otherwise, setting it to "base_link" is likely appropriate.

odom_frame Only used if *provide_odom_frame* is true. The frame between *published_frame* and *map_frame* to be used for publishing the (non-loop-closed) local SLAM result. Usually "odom".

provide_odom_frame If enabled, the local, non-loop-closed, continuous pose will be published as the *odom_frame* in the *map_frame*.

publish_frame_projected_to_2d If enabled, the published pose will be restricted to a pure 2D pose (no roll, pitch, or z-offset). This prevents potentially unwanted out-of-plane poses in 2D mode that can occur due to the pose extrapolation step (e.g. if the pose shall be published as a 'base-footprint'-like frame)

use_odometry If enabled, subscribes to nav_msgs/Odometry on the topic "odom". Odometry must be provided in this case, and the information will be included in SLAM.

use_nav_sat If enabled, subscribes to sensor_msgs/NavSatFix on the topic "fix". Navigation data must be provided in this case, and the information will be included in the global SLAM.

use_landmarks If enabled, subscribes to cartographer_ros_msgs/LandmarkList on the topic "landmarks". Landmarks must be provided, as cartographer_ros_msgs/LandmarkEntry within cartographer_ros_msgs/LandmarkList. If cartographer_ros_msgs/LandmarkEntry data is provided the information will be included in the SLAM according to the ID of the cartographer_ros_msgs/LandmarkEntry. The

cartographer_ros_msgs/LandmarkList should be provided at a sample rate comparable to the other sensors. The list can be empty but has to be provided because Cartographer strictly time orders sensor data in order to make the landmarks deterministic. However it is possible to set the trajectory builder option "collate_landmarks" to false and allow for a non-deterministic but also non-blocking approach.

num_laser_scans Number of laser scan topics to subscribe to. Subscribes to sensor_msgs/LaserScan on the "scan" topic for one laser scanner, or topics "scan_1", "scan_2", etc. for multiple laser scanners.

num_multi_echo_laser_scans Number of multi-echo laser scan topics to subscribe to. Subscribes to sensor_msgs/MultiEchoLaserScan on the "echoes" topic for one laser scanner, or topics "echoes_1", "echoes_2", etc. for multiple laser scanners.

num_subdivisions_per_laser_scan Number of point clouds to split each received (multi-echo) laser scan into. Subdividing a scan makes it possible to unwarp scans acquired while the scanners are moving. There is a corresponding trajectory builder option to accumulate the subdivided scans into a point cloud that will be used for scan matching.

num_point_clouds Number of point cloud topics to subscribe to. Subscribes to sensor_msgs/PointCloud2 on the "points2" topic for one rangefinder, or topics "points2_1", "points2_2", etc. for multiple rangefinders.

lookup_transform_timeout_sec Timeout in seconds to use for looking up transforms using tf2.

submap_publish_period_sec Interval in seconds at which to publish the submap poses, e.g. 0.3 seconds.

pose_publish_period_sec Interval in seconds at which to publish poses, e.g. 5e-3 for a frequency of 200 Hz.

publish_to_tf Enable or disable providing of TF transforms.

publish_tracked_pose Enable publishing of tracked pose as a geometry_msgs/PoseStamped to topic "tracked_pose".

trajectory_publish_period_sec Interval in seconds at which to publish the trajectory markers, e.g. 30e-3 for 30 milliseconds.

rangefinder_sampling_ratio Fixed ratio sampling for range finders messages.

odometry_sampling_ratio Fixed ratio sampling for odometry messages.

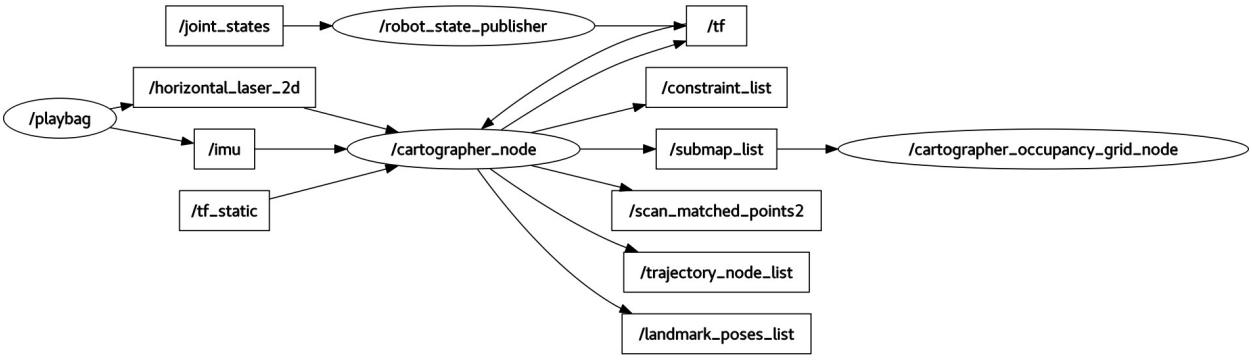
fixed_frame_sampling_ratio Fixed ratio sampling for fixed frame messages.

imu_sampling_ratio Fixed ratio sampling for IMU messages.

landmarks_sampling_ratio Fixed ratio sampling for landmarks messages.

CHAPTER 10

ROS API reference documentation



10.1 Cartographer Node

The cartographer_node is the SLAM node used for online, real-time SLAM.

10.1.1 Command-line Flags

Call the node with the `--help` flag to see all available options.

10.1.2 Subscribed Topics

The following range data topics are mutually exclusive. At least one source of range data is required.

scan (sensor_msgs/LaserScan) Supported in 2D and 3D (e.g. using an axially rotating planar laser scanner). If *num_laser_scans* is set to 1 in the *Lua configuration reference documentation*, this topic will be used as input for SLAM. If *num_laser_scans* is greater than 1, multiple numbered scan topics (i.e. scan_1, scan_2, scan_3, ... up to and including *num_laser_scans*) will be used as inputs for SLAM.

echoes (sensor_msgs/MultiEchoLaserScan) Supported in 2D and 3D (e.g. using an axially rotating planar laser scanner). If *num_multi_echo_laser_scans* is set to 1 in the *Lua configuration reference documentation*, this topic will be used as input for SLAM. Only the first echo is used. If *num_multi_echo_laser_scans* is greater than 1, multiple numbered echoes topics (i.e. echoes_1, echoes_2, echoes_3, ... up to and including *num_multi_echo_laser_scans*) will be used as inputs for SLAM.

points2 (sensor_msgs/PointCloud2) If *num_point_clouds* is set to 1 in the *Lua configuration reference documentation*, this topic will be used as input for SLAM. If *num_point_clouds* is greater than 1, multiple numbered points2 topics (i.e. points2_1, points2_2, points2_3, ... up to and including *num_point_clouds*) will be used as inputs for SLAM.

The following additional sensor data topics may also be provided:

imu (sensor_msgs/Imu) Supported in 2D (optional) and 3D (required). This topic will be used as input for SLAM.

odom (nav_msgs/Odometry) Supported in 2D (optional) and 3D (optional). If *use_odometry* is enabled in the *Lua configuration reference documentation*, this topic will be used as input for SLAM.

10.1.3 Published Topics

scan_matched_points2 (sensor_msgs/PointCloud2) Point cloud as it was used for the purpose of scan-to-submap matching. This cloud may be both filtered and projected depending on the *Lua configuration reference documentation*.

submap_list (cartographer_ros_msgs/SubmapList) List of all submaps, including the pose and latest version number of each submap, across all trajectories.

tracked_pose (geometry_msgs/PoseStamped) Only published if the parameter `publish_tracked_pose` is set to `true`. The pose of the tracked frame with respect to the map frame.

10.1.4 Services

All services responses include also a `StatusResponse` that comprises a `code` and a `message` field. For consistency, the integer `code` is equivalent to the status codes used in the gRPC API.

submap_query (cartographer_ros_msgs/SubmapQuery) Fetches the requested submap.

start_trajectory (cartographer_ros_msgs/StartTrajectory) Starts a trajectory using default sensor topics and the provided configuration. An initial pose can be optionally specified. Returns an assigned trajectory ID.

trajectory_query (cartographer_ros_msgs/TrajectoryQuery) Returns the trajectory data from the pose graph.

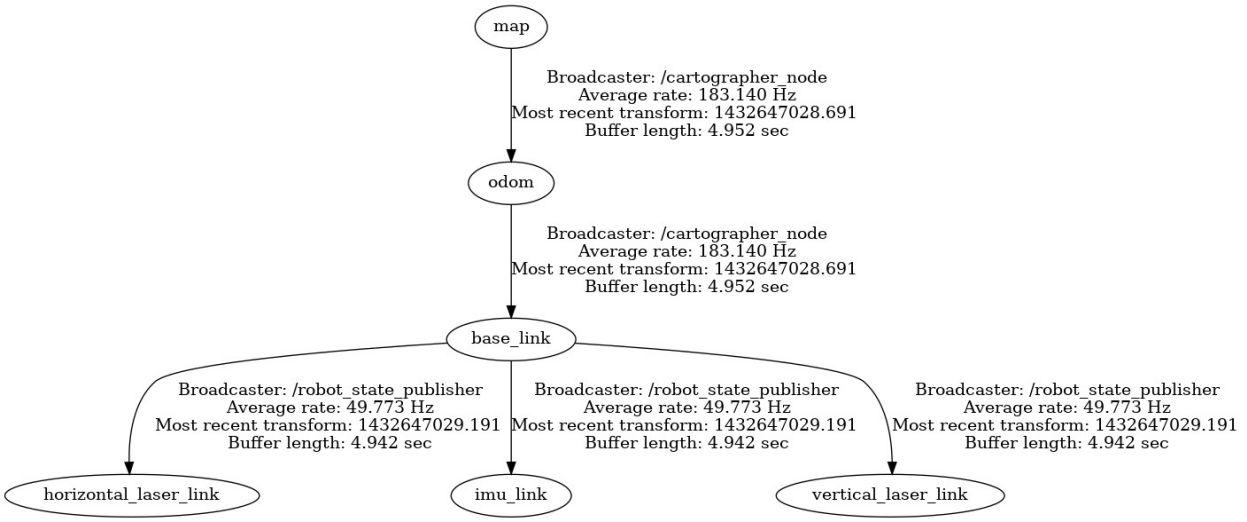
finish_trajectory (cartographer_ros_msgs/FinishTrajectory) Finishes the given *trajectory_id*'s trajectory by running a final optimization.

write_state (cartographer_ros_msgs/WriteState) Writes the current internal state to disk into *filename*. The file will usually end up in *~/.ros* or *ROS_HOME* if it is set. This file can be used as input to the *assets_writer_main* to generate assets like probability grids, X-Rays or PLY files.

get_trajectory_states (cartographer_ros_msgs/GetTrajectoryStates) Returns the IDs and the states of the trajectories. For example, this can be useful to observe the state of Cartographer from a separate node.

read_metrics (cartographer_ros_msgs/ReadMetrics) Returns the latest values of all internal metrics of Cartographer. The collection of runtime metrics is optional and has to be activated with the `--collect_metrics` command line flag in the node.

10.1.5 Required tf Transforms



Transforms from all incoming sensor data frames to the *configured* *tracking_frame* and *published_frame* must be available. Typically, these are published periodically by a *robot_state_publisher* or a *static_transform_publisher*.

10.1.6 Provided tf Transforms

The transformation between the *configured* *map_frame* and *published_frame* is provided unless the parameter `publish_to_tf` is set to `false`.

If *provide_odom_frame* is enabled in the *Lua configuration reference documentation*, additionally a continuous (i.e. unaffected by loop closure) transform between the *configured* *odom_frame* and *published_frame* will be provided.

10.2 Offline Node

The offline_node is the fastest way of SLAMing a bag of sensor data. It does not listen on any topics, instead it reads TF and sensor data out of a set of bags provided on the commandline. It also publishes a clock with the advancing sensor data, i.e. replaces `rosbag play`. In all other regards, it behaves like the `cartographer_node`. Each bag will become a separate trajectory in the final state. Once it is done processing all data, it writes out the final Cartographer state and exits.

10.2.1 Published Topics

In addition to the topics that are published by the online node, this node also publishes:

~bagfile_progress (cartographer_ros_msgs/BagfileProgress) Bag files processing progress including detailed information about the bag currently being processed which will be published with a predefined interval that can be specified using `~bagfile_progress_pub_interval` ROS parameter.

10.2.2 Parameters

~bagfile_progress_pub_interval (double, default=10.0): The interval of publishing bag files processing progress in seconds.

10.3 Occupancy grid Node

The occupancy_grid_node listens to the submaps published by SLAM, builds an ROS occupancy_grid out of them and publishes it. This tool is useful to keep old nodes that require a single monolithic map to work happy until new nav stacks can deal with Cartographer's submaps directly. Generating the map is expensive and slow, so map updates are in the order of seconds. You can can selectively include/exclude submaps from frozen (static) or active trajectories with a command line option. Call the node with the `--help` flag to see these options.

10.3.1 Subscribed Topics

It subscribes to Cartographer's `submap_list` topic only.

10.3.2 Published Topics

map (nav_msgs/OccupancyGrid) If subscribed to, the node will continuously compute and publish the map. The time between updates will increase with the size of the map. For faster updates, use the submaps APIs.

10.4 Pbstream Map Publisher Node

The pbstream_map_publisher is a simple node that creates a static occupancy grid out of a serialized Cartographer state (pbstream format). It is an efficient alternative to the occupancy grid node if live updates are not important.

10.4.1 Subscribed Topics

None.

10.4.2 Published Topics

map (nav_msgs/OccupancyGrid) The published occupancy grid topic is latched.

CHAPTER 11

Public Data

11.1 2D Cartographer Backpack – Deutsches Museum

This data was collected using a 2D LIDAR backpack at the Deutsches Museum. Each bag contains data from an IMU, data from a horizontal LIDAR intended for 2D SLAM, and data from an additional vertical (i.e. push broom) LIDAR.

11.1.1 License

11.1.2 Data

ROS Bag	Duration	Size	Floor	Known Issues
b0-2014-07-11-10-58-16.bag	149 s	38 MB	1. OG	
b0-2014-07-11-11-00-49.bag	513 s	135 MB	1. OG	

Continued on next page

Table 1 – continued from previous page

ROS Bag	Duration	Size	Floor	Known Issues
b0-2014-07-21-12-42-53.bag	244 s	64 MB	1. OG	
b0-2014-07-21-12-49-19.bag	344 s	93 MB	EG	1 gap in vertical laser data
b0-2014-07-21-12-55-35.bag	892 s	237 MB	EG	
b0-2014-07-21-13-11-35.bag	615 s	162 MB	EG	
b0-2014-08-14-13-23-01.bag	768 s	204 MB	1. OG	
b0-2014-08-14-13-36-48.bag	331 s	87 MB	1. OG	
b0-2014-10-07-12-13-36.bag	470 s	125 MB	1. OG	
b0-2014-10-07-12-34-42.bag	491 s	127 MB	1. OG	
b0-2014-10-07-12-43-25.bag	288 s	77 MB	1. OG	
b0-2014-10-07-12-50-07.bag	815 s	215 MB	1. OG	
b1-2014-09-25-10-11-12.bag	1829 s	480 MB	EG	
b1-2014-10-02-14-08-42.bag	930 s	245 MB	1. OG	
b1-2014-10-02-14-33-25.bag	709 s	181 MB	1. OG	
b1-2014-10-07-12-12-04.bag	737 s	194 MB	1. OG	
b1-2014-10-07-12-34-51.bag	766 s	198 MB	1. OG	
b2-2014-11-24-14-20-50.bag	679 s	177 MB	1. OG	
b2-2014-11-24-14-33-46.bag	1285 s	330 MB	1. OG	
b2-2014-12-03-10-14-13.bag	1051 s	275 MB	1. OG	

Continued on next page

Table 1 – continued from previous page

ROS Bag	Duration	Size	Floor	Known Issues
b2-2014-12-03-10-33-51.bag	356 s	89 MB	1. OG	
b2-2014-12-03-10-40-04.bag	453 s	119 MB	1. OG	
b2-2014-12-12-13-51-02.bag	1428 s	368 MB	1. OG	
b2-2014-12-12-14-18-43.bag	1164 s	301 MB	1. OG	
b2-2014-12-12-14-41-29.bag	168 s	46 MB	1. OG	
b2-2014-12-12-14-48-22.bag	243 s	65 MB	1. OG	
b2-2014-12-17-14-33-12.bag	1061 s	277 MB	1. OG	
b2-2014-12-17-14-53-26.bag	246 s	62 MB	1. OG	
b2-2014-12-17-14-58-13.bag	797 s	204 MB	EG	
b2-2015-02-16-12-26-11.bag	901 s	236 MB	1. OG	
b2-2015-02-16-12-43-57.bag	1848 s	475 MB	1. OG	
b2-2015-04-14-14-16-36.bag	1353 s	349 MB	1. OG	
b2-2015-04-14-14-39-59.bag	670 s	172 MB	1. OG	
b2-2015-04-28-13-01-40.bag	618 s	162 MB	1. OG	
b2-2015-04-28-13-17-23.bag	2376 s	613 MB	1. OG	
b2-2015-05-12-12-29-05.bag	942 s	240 MB	1. OG	2 gaps in laser data
b2-2015-05-12-12-46-34.bag	2281 s	577 MB	1. OG	14 gaps in laser data

Continued on next page

Table 1 – continued from previous page

ROS Bag	Duration	Size	Floor	Known Issues
b2-2015-05-26-13-15-25.bag	747 s	195 MB	1. OG	
b2-2015-06-09-14-31-16.bag	1297 s	336 MB	1. OG	
b2-2015-06-25-14-25-51.bag	1071 s	272 MB	1. OG	
b2-2015-07-07-11-27-05.bag	1390 s	362 MB	1. OG	
b2-2015-07-21-13-03-21.bag	894 s	239 MB	1. OG	
b2-2015-08-04-13-39-24.bag	809 s	212 MB	1. OG	
b2-2015-08-18-11-42-31.bag	588 s	155 MB	UG	
b2-2015-08-18-11-55-04.bag	504 s	130 MB	UG	
b2-2015-08-18-12-06-34.bag	1299 s	349 MB	EG	
b2-2015-09-01-11-55-40.bag	1037 s	274 MB	UG	
b2-2015-09-01-12-16-13.bag	918 s	252 MB	EG	
b2-2015-09-15-14-19-11.bag	859 s	225 MB	1. OG	
b2-2015-11-24-14-12-27.bag	843 s	226 MB	1. OG	
b2-2016-01-19-14-10-47.bag	310 s	81 MB	1. OG	
b2-2016-02-02-14-01-56.bag	787 s	213 MB	EG	1 gap in laser data
b2-2016-03-01-14-09-37.bag	948 s	255 MB	EG	
b2-2016-03-15-14-23-01.bag	810 s	215 MB	EG	
b2-2016-04-05-14-44-52.bag	360 s	94 MB	1. OG	
b2-2016-04-27-12-31-41.bag	881 s	234 MB	1. OG	

11.2 3D Cartographer Backpack – Deutsches Museum

This data was collected using a 3D LIDAR backpack at the Deutsches Museum. Each bag contains data from an IMU and from two Velodyne VLP-16 LIDARs, one mounted horizontally (i.e. spin axis up) and one vertically (i.e. push broom).

11.2.1 License

11.2.2 Data

ROS Bag	Duration	Size	Known Issues
b3-2015-12-10-12-41-07.bag	1466 s	7.3 GB	1 large gap in data, no intensities
b3-2015-12-10-13-10-17.bag	718 s	5.5 GB	1 gap in data, no intensities
b3-2015-12-10-13-31-28.bag	720 s	5.2 GB	2 large gaps in data, no intensities
b3-2015-12-10-13-55-20.bag	429 s	3.3 GB	
b3-2015-12-14-15-13-53.bag	916 s	7.1 GB	no intensities
b3-2016-01-19-13-26-24.bag	1098 s	8.1 GB	no intensities
b3-2016-01-19-13-50-11.bag	318 s	2.5 GB	no intensities
b3-2016-02-02-13-32-01.bag	47 s	366 MB	no intensities
b3-2016-02-02-13-33-30.bag	1176 s	9.0 GB	no intensities
b3-2016-02-09-13-17-39.bag	529 s	4.0 GB	
b3-2016-02-09-13-31-50.bag	801 s	6.1 GB	no intensities
b3-2016-02-10-08-08-26.bag	3371 s	25 GB	
b3-2016-03-01-13-39-41.bag	382 s	2.9 GB	
b3-2016-03-01-15-42-37.bag	3483 s	17 GB	6 large gaps in data, no intensities
b3-2016-03-01-16-42-00.bag	313 s	2.4 GB	no intensities
b3-2016-03-02-10-09-32.bag	1150 s	6.6 GB	3 large gaps in data, no intensities
b3-2016-04-05-13-54-42.bag	829 s	6.1 GB	no intensities
b3-2016-04-05-14-14-00.bag	1221 s	9.1 GB	
b3-2016-04-05-15-51-36.bag	30 s	231 MB	
b3-2016-04-05-15-52-20.bag	377 s	2.7 GB	no intensities
b3-2016-04-05-16-00-55.bag	940 s	6.9 GB	no intensities
b3-2016-04-27-12-25-00.bag	2793 s	23 GB	
b3-2016-04-27-12-56-11.bag	2905 s	21 GB	
b3-2016-05-10-12-56-33.bag	1767 s	13 GB	
b3-2016-06-07-12-42-49.bag	596 s	3.9 GB	3 gaps in horizontal laser data, no intensities

11.3 MiR

This data was collected using MiR100. An additional Logitech Webcam C930e Full HD camera was attached on top to collect images for landmark detection.

11.3.1 License

11.3.2 Data

ROS Bag	Duration	Size
landmarks_demo_uncalibrated.bag	180 s	41.7 MB

11.4 PR2 – Willow Garage

This is the Willow Garage data set, described in:

- "An Object-Based Semantic World Model for Long-Term Change Detection and Semantic Querying.", by Julian Mason and Bhaskara Marthi, IROS 2012.

More details about these data can be found in:

- "Unsupervised Discovery of Object Classes with a Mobile Robot", by Julian Mason, Bhaskara Marthi, and Ronald Parr. ICRA 2014.
- "Object Discovery with a Mobile Robot" by Julian Mason. PhD Thesis, 2013.

11.4.1 License

11.4.2 Data

ROS Bag	Known Issues
2011-08-03-16-16-43.bag	Missing base laser data
2011-08-03-20-03-22.bag	
2011-08-04-12-16-23.bag	
2011-08-04-14-27-40.bag	
2011-08-04-23-46-28.bag	
2011-08-05-09-27-53.bag	
2011-08-05-12-58-41.bag	
2011-08-05-23-19-43.bag	
2011-08-08-09-48-17.bag	
2011-08-08-14-26-55.bag	
2011-08-08-23-29-37.bag	
2011-08-09-08-49-52.bag	
2011-08-09-14-32-35.bag	
2011-08-09-22-31-30.bag	
2011-08-10-09-36-26.bag	
2011-08-10-14-48-32.bag	
2011-08-11-01-31-15.bag	
2011-08-11-08-36-01.bag	
2011-08-11-14-27-41.bag	
2011-08-11-22-03-37.bag	
2011-08-12-09-06-48.bag	
2011-08-12-16-39-48.bag	
2011-08-12-22-46-34.bag	
2011-08-15-17-22-26.bag	
2011-08-15-21-26-26.bag	
2011-08-16-09-20-08.bag	
2011-08-16-18-40-52.bag	
2011-08-16-20-59-00.bag	
2011-08-17-15-51-51.bag	
2011-08-17-21-17-05.bag	
2011-08-18-20-33-16.bag	
2011-08-18-20-52-30.bag	
2011-08-19-10-12-20.bag	
2011-08-19-14-17-55.bag	
2011-08-19-21-35-17.bag	
2011-08-22-10-02-27.bag	
2011-08-22-14-53-33.bag	

Continued on next page

Table 2 – continued from previous page

ROS Bag	Known Issues
2011-08-23-01-11-53.bag	
2011-08-23-09-21-17.bag	
2011-08-24-09-52-14.bag	
2011-08-24-15-01-39.bag	
2011-08-24-19-47-10.bag	
2011-08-25-09-31-05.bag	
2011-08-25-20-14-56.bag	
2011-08-25-20-38-39.bag	
2011-08-26-09-58-19.bag	
2011-08-29-15-48-07.bag	
2011-08-29-21-14-07.bag	
2011-08-30-08-55-28.bag	
2011-08-30-20-49-42.bag	
2011-08-30-21-17-56.bag	
2011-08-31-20-29-19.bag	
2011-08-31-20-44-19.bag	
2011-09-01-08-21-33.bag	
2011-09-02-09-20-25.bag	
2011-09-06-09-04-41.bag	
2011-09-06-13-20-36.bag	
2011-09-08-13-14-39.bag	
2011-09-09-13-22-57.bag	
2011-09-11-07-34-22.bag	
2011-09-11-09-43-46.bag	
2011-09-12-14-18-56.bag	
2011-09-12-14-47-01.bag	
2011-09-13-10-23-31.bag	
2011-09-13-13-44-21.bag	
2011-09-14-10-19-20.bag	
2011-09-15-08-32-46.bag	

11.5 Magazino

Datasets recorded on Magazino robots.

See the cartographer_magazino repository for an integration of Magazino robot data for Cartographer.

See the `LICENSE` file in `cartographer_magazino` for details on the dataset license.

11.5.1 Data

ROS Bag	Duration	Size	Known Issues
hallway_return.bag	350 s	102.8 MB	
hallway_localization.bag	137 s	40.4 MB	

CHAPTER 12

Frequently asked questions

12.1 Why is laser data rate in the 3D bags higher than the maximum reported 20 Hz rotation speed of the VLP-16?

The VLP-16 in the example bags is configured to rotate at 20 Hz. However, the frequency of UDP packets the VLP-16 sends is much higher and independent of the rotation frequency. The example bags contain a sensor_msgs/PointCloud2 per UDP packet, not one per revolution.

In the corresponding Cartographer configuration file you see *TRAJECTORY_BUILDER_3D.num_accumulated_range_data = 160* which means we accumulate 160 per-UDP-packet point clouds into one larger point cloud, which incorporates motion estimation by combining constant velocity and IMU measurements, for matching. Since there are two VLP-16s, 160 UDP packets is enough for roughly 2 revolutions, one per VLP-16.

12.2 Why is IMU data required for 3D SLAM but not for 2D?

In 2D, Cartographer supports running the correlative scan matcher, which is normally used for finding loop closure constraints, for local SLAM. It is computationally expensive but can often render the incorporation of odometry or IMU data unnecessary. 2D also has the benefit of assuming a flat world, i.e. up is implicitly defined.

In 3D, an IMU is required mainly for measuring gravity. Gravity is an attractive quantity to measure since it does not drift and is a very strong signal and typically comprises most of any measured accelerations. Gravity is needed for two reasons:

1. There are no assumptions about the world in 3D. To properly world align the resulting trajectory and map, gravity is used to define the z-direction.

2. Roll and pitch can be derived quite well from IMU readings once the direction of gravity has been established. This saves work for the scan matcher by reducing the search window in these dimensions.

12.3 How do I build cartographer_ros without rviz support?

The simplest solution is to create an empty file named CATKIN_IGNORE in the *cartographer_rviz* package directory.

12.4 How do I fix the "You called InitGoogleLogging() twice!" error?

Building *rosconsole* with the *glog* back end can lead to this error. Use the *log4cxx* or *print* back end, selectable via the *ROSCONSOLE_BACKEND* CMake argument, to avoid this issue.