Cartographer ROS Documentation

Release 1.0.0

The Cartographer Authors

Oct 27, 2017

Contents

CHAPTER 1

Configuration

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*.

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.

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.

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

CHAPTER 2

Tuning

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.

Two systems

Cartographer can be seen as two separate, but related systems. The first one is local SLAM (sometimes also called frontend). Its job is build a locally consistent set of submaps and tie them together, but it will drift over time. Most of its options can be found in trajectory_builder_2d.lua for 2D and trajectory_builder_3d.lua for 3D.

The other system 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 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 sparse_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.

Tuning local SLAM

For this example we'll start at `cartographer` commit ea7c39b and `cartographer_ros` commit 44459e1 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.

```
SPARSE_POSE_GRAPH.optimize_every_n_scans = 0
```

Correct size of submaps

Local SLAM drifts over time, only loop closure can 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 being distinct for loop closure to work properly. 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.

The choice of scan matchers

The idea behind local SLAM is to use sensor data of other sensors besides the range finder to predict where the next scan should be inserted into the submap. Then, the `CeresScanMatcher` takes this 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.

If you do not have other sensors or you do not trust them, Cartographer also provides a `RealTimeCorrelativeScanMatcher`. It uses an approach similar to how scans are matched against submaps in loop closure, 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.

Tuning the correlative scan matcher

TODO

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.

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.

CHAPTER 3

ROS API

Cartographer Node

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

Command-line Flags

TODO(hrapp): Should these not be removed? It seems duplicated efforts documenting them here and there.

–configuration_directory First directory in which configuration files are searched, second is always the Cartographer installation to allow including files from there.

–configuration_basename Basename (i.e. not containing any directory prefix) of the configuration file (e.g. backpack_3d.lua).

–map_filename A Cartographer state file that will be loaded from disk. This allows to add new trajectories SLAMing from an earlier state, but the loaded state is frozen.

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 *Configuration*, 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 *Configuration*, 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 *Configuration*, 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 *Configuration*, this topic will be used as input for SLAM.

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 *Configuration*.

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

Services

submap_query (cartographer_ros_msgs/SubmapQuery) Fetches the requested submap.

start_trajectory (cartographer_ros_msgs/StartTrajectory) Starts another trajectory by specifying its sensor topics and trajectory options as an binary-encoded proto. Returns an assigned trajectory ID.

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.

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*.

Provided tf Transforms

The transformation between the *configured* *map_frame* and *published_frame* is always provided.

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

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.

Occupancy grid Node

The occupancy_grid_node listens to the submaps published by SLAM and builds a ROS occupancy_grid and publishes it. This tool is 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.

Subscribed Topics

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

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.

CHAPTER 4

Assets writer

The purpose of SLAM is to compute the trajectory of a single sensor through a metric space. On a higher level, the input of SLAM is sensor data, its output is the best estimate of the trajectory up to this point in time. To be real-time and efficient, Cartographer throws most of the sensor data away immediately.

The trajectory alone is rarely of interest. But once the best trajectory is established, the full sensor data can be used to derive and visualize information about its surroundings.

Cartographer provides the assets writer for this. Its inputs are

1. the original sensor data fed to SLAM in a ROS bag file,
2. the cartographer state, which is contained in the `.pbstream` file that SLAM creates,
3. the sensor extrinsics (i.e. TF data from the bag or a URDF),
4. and a pipeline configuration, which is defined in a `.lua` file.

The assets writer runs through the sensor data in batches with a known trajectory. It can be used to color, filter and export SLAM point cloud data in a variety of formats. For more information on what the asset writer can be used for, refer to the examples below below and the header files in cartographer/io.

Sample usage

```
# Download the 3D backpack example bag.
wget -P ~/Downloads https://storage.googleapis.com/cartographer-public-data/bags/
↪backpack_3d/b3-2016-04-05-14-14-00.bag

# Launch the 3D offline demo.
roslaunch cartographer_ros offline_backpack_3d.launch bag_filenames:=${HOME}/
↪Downloads/b3-2016-04-05-14-14-00.bag
```

Watch the output on the commandline until the offline 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. You could have gotten the same state data by running the online node and calling:

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

# Ask Cartographer to serialize its current state.
rosservice call /write_state ${HOME}/Downloads/b3-2016-04-05-14-14-00.bag.pbstream
```

Now we run the assets writer with the sample configuration file 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 by `--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`.

Configuration

The assets writer is modeled as a pipeline. It consists of PointsProcessors and PointsBatchs flow through it. Data flows from the first processor to the next, each has the chance to modify the `PointsBatch` before passing it on.

For example the assets_writer_backpack_3d.lua uses `min_max_range_filter` to remove points that are either too close or too far from the sensor. After this, it writes X-Rays, then recolors the `PointsBatchs` depending on the sensor frame ids and writes another set of X-Rays using these new colors.

The individual `PointsProcessors` are all in the cartographer/io sub-directory and documented in their individual header files.

First-person visualization of point clouds

Generating a fly through of points is a two step approach: First, write a PLY file with the points you want to visualize, then use point_cloud_viewer.

The first step is usually accomplished by using IntensityToColorPointsProcessor to give the points a non-white color, then writing them to a PLY using PlyWritingPointsProcessor. An example 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 in the same repo.

CHAPTER 5

Public Data

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.

License

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 5.1 – 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 5.1 – 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 5.1 – 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	

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).

License

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
b3-2016-04-27-12-56-11.bag	2905 s	21 GB	no intensities
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

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.

License

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	

Continued on next page

Table 5.2 – continued from previous page

ROS Bag	Known Issues
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	
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	

Continued on next page

Table 5.2 – continued from previous page

ROS Bag	Known Issues
2011-09-13-13-44-21.bag	
2011-09-14-10-19-20.bag	
2011-09-15-08-32-46.bag	

CHAPTER 6

Frequently asked questions

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.scans_per_accumulation = 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.

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.

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.

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.

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 7

System Requirements

See Cartographer's system requirements.

The following ROS distributions are currently supported:

- Indigo
- Kinetic

CHAPTER 8

Building & Installation

We recommend using wstool and rosdep. For faster builds, we also recommend using Ninja.

```
# Install wstool and rosdep.
sudo apt-get update
sudo apt-get install -y python-wstool python-rosdep ninja-build

# Create a new workspace in 'catkin_ws'.
mkdir catkin_ws
cd catkin_ws
wstool init src

# Merge the cartographer_ros.rosinstall file and fetch code for dependencies.
wstool merge -t src https://raw.githubusercontent.com/googlecartographer/
↪cartographer_ros/master/cartographer_ros.rosinstall
wstool update -t src

# Install deb dependencies.
# 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

# Build and install.
catkin_make_isolated --install --use-ninja
source install_isolated/setup.bash
```

CHAPTER 9

Running the demos

Now that Cartographer and Cartographer's ROS integration are installed, download the 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:

```
# Download the 2D backpack example bag.
wget -P ~/Downloads https://storage.googleapis.com/cartographer-public-data/
↪bags/backpack_2d/cartographer_paper_deutsches_museum.bag

# Launch the 2D backpack demo.
roslaunch cartographer_ros demo_backpack_2d.launch bag_filename:=${HOME}/
↪Downloads/cartographer_paper_deutsches_museum.bag

# Pure localization demo: We use 2 different 2D bags from the Deutsche
# Museum. The first one is used to generate the map, the second to run
# pure localization.
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: Run the next command, wait until cartographer_offline_
↪node finishes.
roslaunch cartographer_ros offline_backpack_2d.launch bag_filenames:=${HOME}/
↪Downloads/b2-2016-04-05-14-44-52.bag
# Run pure localization:
roslaunch cartographer_ros demo_backpack_2d_localization.launch \
   bag_filename:=${HOME}/Downloads/b2-2016-04-27-12-31-41.bag \
   map_filename:=${HOME}/Downloads/b2-2016-04-05-14-44-52.bag.pbstream

# Download the 3D backpack example bag.
wget -P ~/Downloads https://storage.googleapis.com/cartographer-public-data/
↪bags/backpack_3d/b3-2016-04-05-14-14-00.bag

# Launch the 3D backpack demo.
roslaunch cartographer_ros demo_backpack_3d.launch bag_filename:=${HOME}/
↪Downloads/b3-2016-04-05-14-14-00.bag
```

```
# Download the Revo LDS example bag.
wget -P ~/Downloads https://storage.googleapis.com/cartographer-public-data/
↪bags/revo_lds/cartographer_paper_revo_lds.bag

# Launch the Revo LDS demo.
roslaunch cartographer_ros demo_revo_lds.launch bag_filename:=${HOME}/
↪Downloads/cartographer_paper_revo_lds.bag

# Download the PR2 example bag.
wget -P ~/Downloads https://storage.googleapis.com/cartographer-public-data/
↪bags/pr2/2011-09-15-08-32-46.bag

# Launch the PR2 demo.
roslaunch cartographer_ros demo_pr2.launch bag_filename:=${HOME}/Downloads/
↪2011-09-15-08-32-46.bag

# Download the Taurob Tracker example bag.
wget -P ~/Downloads https://storage.googleapis.com/cartographer-public-data/
↪bags/taurob_tracker/taurob_tracker_simulation.bag

# Launch the Taurob Tracker demo.
roslaunch cartographer_ros demo_taurob_tracker.launch bag_filename:=${HOME}/
↪Downloads/taurob_tracker_simulation.bag
```

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