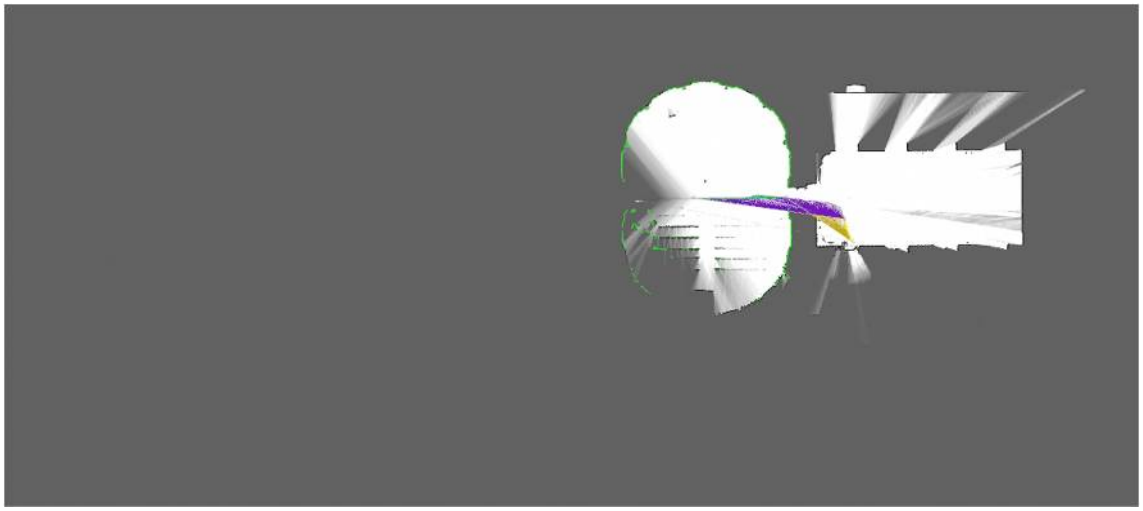
Ouster OS-1 ライダーと Google Cartographer の統合

本投稿は、Ouster OS-1 ライダーで取得したデータと Google Cartographer を統合して、ある環境下で 2D あるいは 3D マップを作成していく過程を記述しています。我々は社内で、我々自身の HD マッピング・ソリューションを構築する一方で、以下の投稿でユーザの方々が、Google Cartographer のようなオープンソースのプログラムを使って、どのように基本的なマッピングを作り上げていくべきか、そのスタートになれればと思います。

Cartographer とは、リアルタイムで、複数のプラットフォームやセンサーを統合化し、2D あるいは 3D で自己位置推定とマップ作成を同時に行う（SLAM）を提供するシステムです。SLAM アルゴリズムは、様々なセンサー（例えば、ライダー、IMU、カメラ）からのデータを統合して、センサーの位置推定とセンサー周辺のマップ作成を同時に行ないます。SLAM は、自動走行車、倉庫の自動フォークリフト、電気掃除ロボット、UAV のような自律走行車両にとって必要不可欠な構成部品になっています。Cartographer の 2D アルゴリズムの詳細に関する記述は、かれらの ICRA 2016 の論文をご覧ください。



Google Cartographer でマッピングを作成する

Cartographer は 2016 年 10 月に、オープンソースプロジェクトとしてリリースされました。Google は、既存の ROS に Cartographer を統合して使用できる幾つかの ROS パッケージを含んだ cartogrpaher_ros repository（カートグラファーROS 貯蔵庫）もリリースし、ROS のサポートを打ち出しています。

Cartographer のインストール

Cartographer は以下の ROS ディストリビューションをサポートしています。

- Indigo
- Kinetic
- Lunar
- Melodic

Google は、ROS 用に Cartographer を構築し、インストールする詳細な説明書をソースから提供しています。あるいは、ROS パッケージは、debian リポジトリからダウンロードできます。以下のコマンドは、ROS Melodic 配布用（ROS Melodic distribution）に必要となる package をインストールできます：

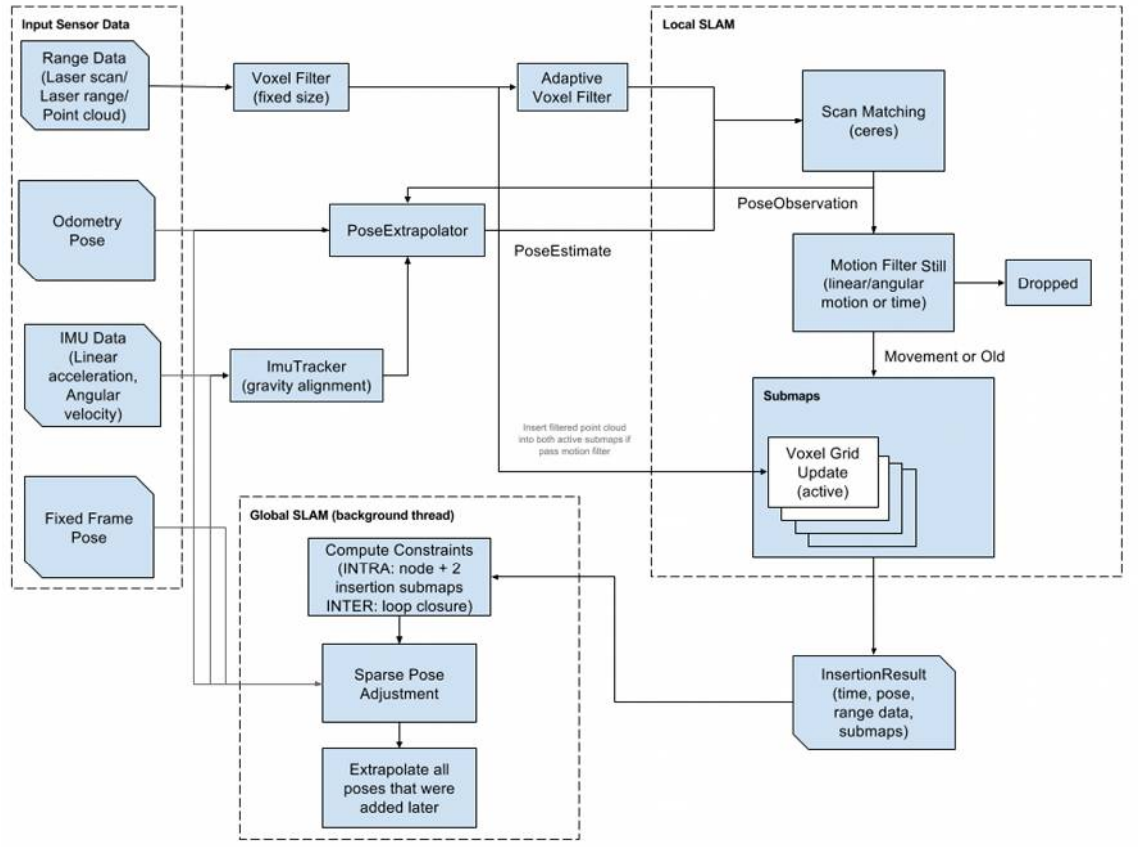
```
$ sudo apt-get install ros-melodic-cartographer ros-melodic-cartographer-ros
ros-melodic-cartographer-ros-msgs ros-melodic-cartographer-rviz
```

Google は、2D、3D マップ作成機能のデモを行うことができる、サンプルデータと説明書を提供しています。

Cartographer プロセスの概要

Cartographer は、主に 2 つのサブシステム、即ち、グローバル SLAM、ローカル SLAM から構成されています。ローカル SLAM は、高品質な領域の部分マップを作成するために使用され、グローバル SLAM は、それら部分マップを可能な限り一致するように繋ぎ合わせるために使用します。

ローカル SLAM は、局所的に一致することを目的とし、しかし時間経過と共にドリフト（ズレ）を持つ一連の部分マップ群を生成します。ローカル SLAM は、局所的な軌跡も作成します。グローバル SLAM は、別の並列スレッドで実行され、ローカル SLAM により作成された部分マップ間のループクロージャ（loop closure）の拘束箇所を見つけ出します。そして、部分マップに対してセンサーのスキャンデータのマッチングを行い、ループクロージャを実行します。グローバル SLAM は、最大限一致するグローバルソリューションを提供するために、他のセンサーも組み込んでいます。



Google Cartographer のシステム概要

Cartographer SLAM アルゴリズムへの主な入力は、ライダーやレーダーのような工学視差式距離計（レンジファインダー）センサーから計算される距離計測の結果です。これらのセンサーは、もともとノイズを持っているため、最初のデータ処理ステップは、バンドパスフィルタを通して、最大及び最小の閾値（range threshold）の範囲外の計測結果を除去することになります。その閾値は、使用される特定のセンサーの物理特性から導かれます。

OS-1-64 のような高解像度センサーは、大量の計測結果を出力するため、計算の時間がかかります。点密度の問題に対処するため、Cartographer は、ボクセルフィルターを使用し、生のポイントを一定サイズのキューブにダウンサンプル化し、各キューブの重心情報のみ保持します。Cartographer は、アダプティブボクセルフィルターを適用し、ターゲットポイント数に到達するように、最適なボクセルサイズを決めます。

データがフィルタにかけられると、ローカル SLAM アルゴリズムは、スキャンマッチングにより、スキャンをカレントの部分マップに挿入します。このプロセスは、初期推定用に pose extrapolator アルゴリズムを使用し、スキャンが部分マップのどこに挿入されるか初期段階で推定するために、他のセンサーを使用します。

Cartographer は、CeresScanMatcher と RealTimeCorrelativeScanMatcher とい

うスキャンマッチングモジュールを提供します。通常は、CeresScanMatcher が推奨されます。これは高速ですが、部分マップの解像度よりもずっと大きな誤差を修正することが出来ません。RealTimeCorrelativeScanMatcher は、非常に高価で、基本的に、レンジファインダー以外の他のセンサーからのいかなる信号も上書きしてしまいます。しかし、特徴に富んだ環境下では強靭性に富んでいます。

スキャンマッチングが行われると、モーションフィルターは、距離、角度、時間に基づくずっと大きな動きから得られたスキャンのみが部分マップに含まれるように働きます。部分マップは、ローカル SLAM が与えられた量の範囲データを受け取ったときに、完結したと見なされます。ローカル SLAM アルゴリズムは、確率グリッドとして知られるデータ構造に部分マップとその範囲データを保存します。

グローバル SLAM アルゴリズムは、部分マップを取り込み、整合性の取れたグローバルなマップになるように、それらを再構成します。これは GraphSLAM の一種で、ノード（頂点）と部分マップ間に拘束を構築し、その結果 作成される拘束グラフを最適化する pose graph optimization です。

ノードと部分マップにより拘束を構築する場合、それらは、FastCorrelativeScanMatcher と呼ばれる最初のスキャンマッチャーに送られます。FastCorrelativeScanMatcher が十分な評価を行えば（最低限のマッチングスコア以上）、位置姿勢を修正するため、次に Ceres Scan Matcher に入力されます。

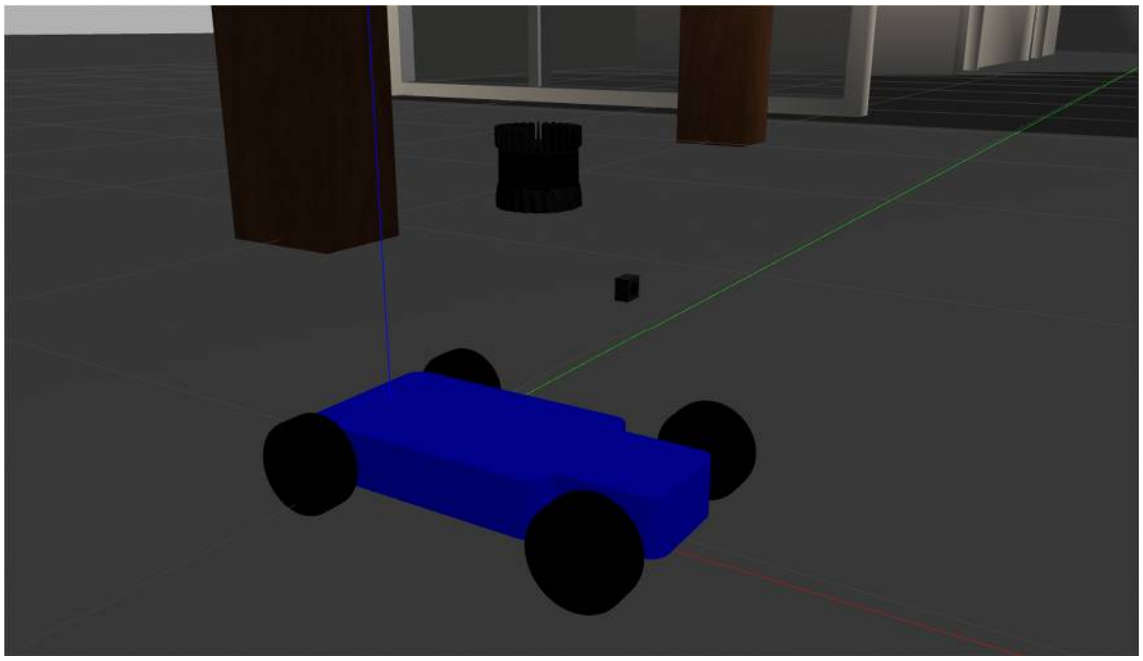
関連するアルゴリズム詳細に関しては、調整のためのアルゴリズム概要をご覧ください。

Cartographer によるシミュレーション実行

Cartographer により提供される ROS ノードを使用すれば、ROS Gazebo により、シミュレーションされたロボットとセンサー一式を構成して、シミュレーション環境でロボットを動かすことが出来ます。そして、Cartographer で作成されたマップや軌跡の閲覧に、ROS の RViz ツールを使用することが出来ます。Cartographer ROS パッケージは、diy_driverless_car_ROS レポジトリに統合されています。

このプロセスに対して、シミュレーションされた OS-1-64 ライダーセンサーを用いて、IMU および距離計測を取得します。シミュレーションされたセンサーの開発詳細は、こちらから入手できます。OS-1-64 は、MIT レースカープロジェクトで提供されたリモコンカーに搭載されます。最後に、WillowGarage world をシミュレーションした環境として使用し、マップを作成します。WillowGarage Gazebo の環境下で、レースカーの車両に

搭載した OS-1-64 のサンプル画像を以下に示します：



OS-1 リモコン カー Gazebo シミュレーション

Lua 設定ファイル

定義される最初のファイルは、.lua 設定ファイルです。ロボットの設定は、スクリプトから定義されなければならない options データ構造から読み込まれます。.lua ファイルは、そのロボットに特有のファイルです。本シミュレーションでは、このロボットに対する Cartographer 設定を定義するために racer_2d.lua が使用されます。

最初に、環境とロボットの TF フレーム ID を定義します。これらの座標フレームは、REP 105 に定義されます。

```
map_frame = ″map″,
tracking_frame = ″os1_imu″,
published_frame = ″base_link″,
odom_frame = ″odom″,
```

map_frame は、部分マップを発行するのに用いられる ROS フレーム ID です。

tracking_frame は、SLAM アルゴリズムにより追跡される ROS フレーム ID で、通常、使用される場合は、IMU フレームです。

published_frame は、位置姿勢を発行するのに用いられる ROS フレーム ID です。Cartographer は走行距離測定を行うように設定するため、この値は、“base link”に設定します。

odom_frame は、Cartographer が走行距離測定を行うよう設定される場合、“odom”に設定します。このフレームは、非ループ閉合のローカル SLAM の結果を発行します。

次に、Cartographer を odom_frame として、ローカルで、非ループ閉合(non-loop-closed local SLAM)で、連続の位置姿勢として設定します。

provide_odom_frame = true,

use_odometry, use_nav_sat, と use_landmarks の設定では、すべて false へ設定します。これは、我々は odometry, GPS, landmark の入力が無いからです。

我々は OS1-64 のデータを 2D laser scan として入力します。また、num_laser_scans の入力欄では Cartographer を非有効に設定し、/scan topic の点データの情報を処理します。そして、使用する sensor_msgs/LaserScan topics の数を定義します。

num_laser_scans = 1,

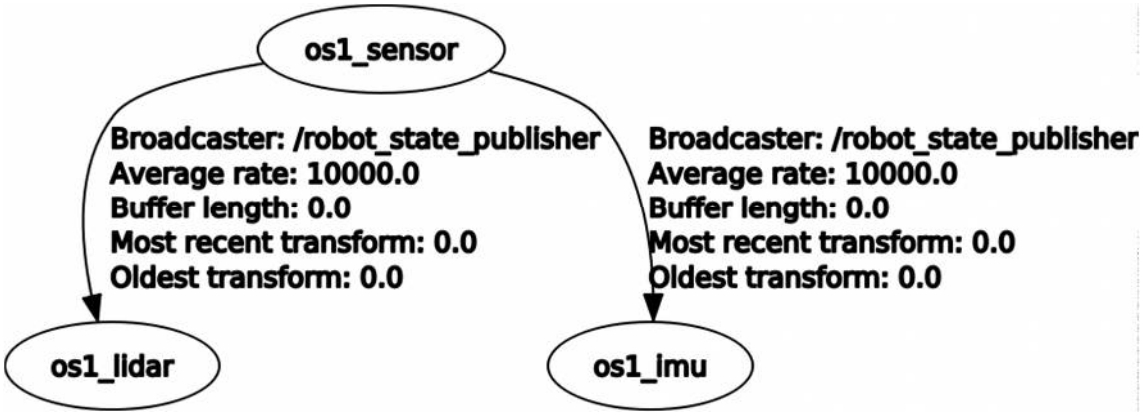
センサー特性に基づき設定する必要がある変数が 1 つあります。TRAJECTORY_BUILDER_2D.num_accumulated_range_data 変数は、フルスキャンを構成するのに要するメッセージの数を定義します（通常は、フル回転です）。この入力欄のデフォルト値は、ベースになる trajectory_builder_2d.lua ファイルでは、1 です。OS-1-64 は、1 回転で 1 つのメッセージを出力するため、これは正しいことです。

残りの値の説明は、Lua configuration reference documentation にあります。

URDF ファイル

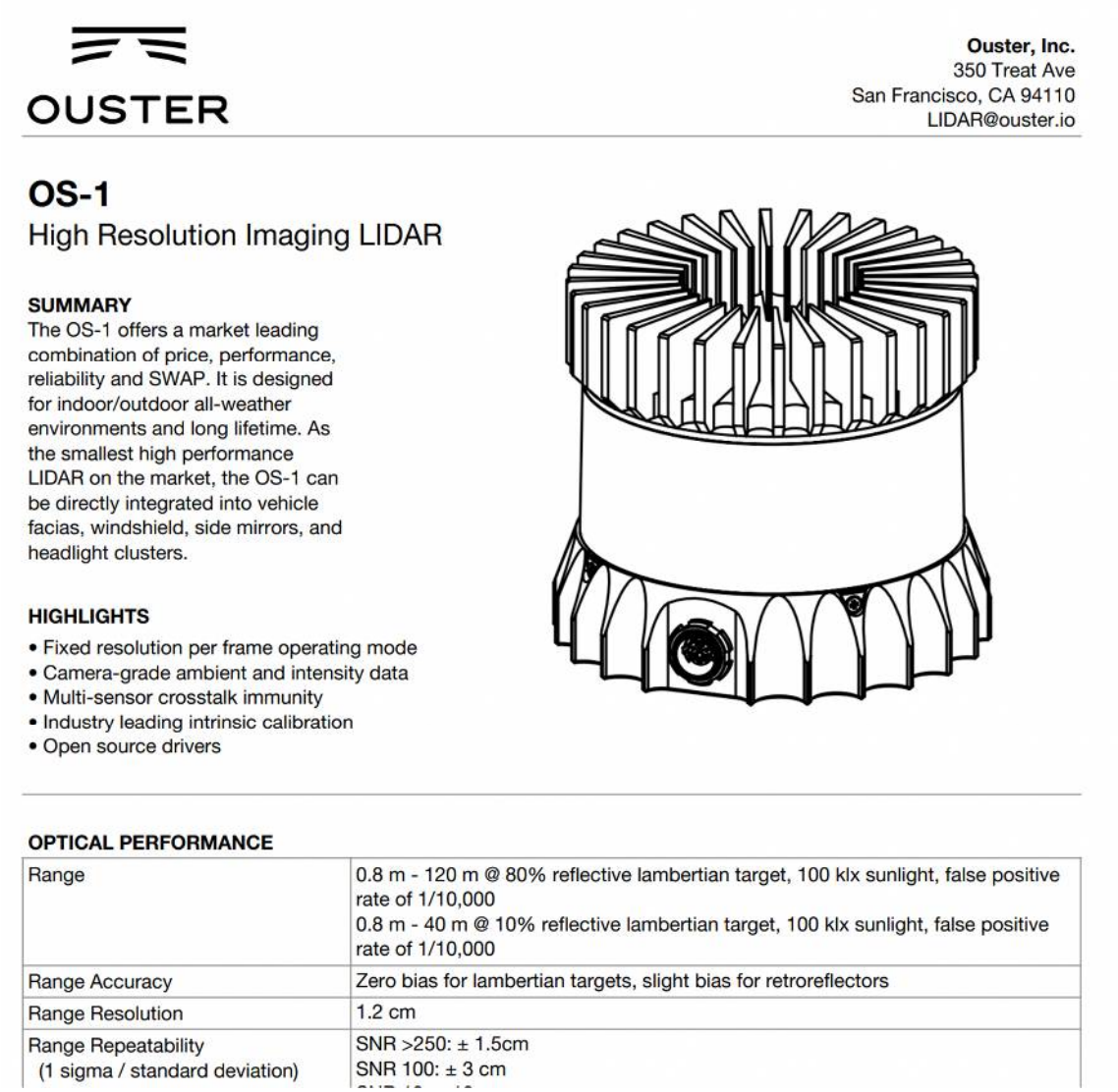
ロボットの TF ツリーを、/tf のトピックから頒布するか、あるいは、.urdf ロボット定義に定義することが可能です。os1_sensor.urdf ファイルを使用して、IMU、レーザーアパーチャー、センサー筐体間の変換を定義することが出来ます。

os1_sensor.urdf ファイルは、センサー筐体のリンク“os1_sensor”を、ベースリンクとして定義します。このリンクは 2 つの子リンクがあります。“os1_imu”リンクは、IMU の位置を表し、“os1_lidar”リンクは、レーザーアパーチャーの位置を表します。これにより、以下の TF ツリーが作成され、Ouster ROS sensor client で提供される TF ツリーと正確に一致します。



OS-1 変換ツリー

部品間の特定の回転と並進（平行移動）は、.urdf ファイルでも定義されます。これらの寸法は、OS-1 Datasheet から取得できます。



OUSTER

Ouster, Inc.
350 Treat Ave
San Francisco, CA 94110
LIDAR@ouster.io

OS-1

High Resolution Imaging LIDAR

SUMMARY

The OS-1 offers a market leading combination of price, performance, reliability and SWAP. It is designed for indoor/outdoor all-weather environments and long lifetime. As the smallest high performance LIDAR on the market, the OS-1 can be directly integrated into vehicle facias, windshield, side mirrors, and headlight clusters.

HIGHLIGHTS

- Fixed resolution per frame operating mode
- Camera-grade ambient and intensity data
- Multi-sensor crosstalk immunity
- Industry leading intrinsic calibration
- Open source drivers

OPTICAL PERFORMANCE

Range	0.8 m - 120 m @ 80% reflective lambertian target, 100 klx sunlight, false positive rate of 1/10,000 0.8 m - 40 m @ 10% reflective lambertian target, 100 klx sunlight, false positive rate of 1/10,000
Range Accuracy	Zero bias for lambertian targets, slight bias for retroreflectors
Range Resolution	1.2 cm
Range Repeatability (1 sigma / standard deviation)	SNR >250: ± 1.5cm SNR 100: ± 3 cm

OS-1 データシート

起動ファイル（Launch File）

Cartographer で推奨する使用法は、ロボットや SLAM のタイプごとにカスタマイズされた.launch ファイルを作成することです。このシミュレーションでは、入力データとして、シミュレーションされた IMU とレーザースキャンデータを用いて、2D SLAM を行っていきます。Cartographer の ROS の設定は、racer_2d_cartographer.launch ファイルに設定されます。

os1_sensor.urdf ファイルはロードされ、robot_state_publisher ノードは、URDF ファイルで定義された状態を、/TF トピックに対して発行します。

cartographer_ros ノードも初期化され、racer_2d.lua ファイルを読み込んで設定が行われます。IMU とレーザースキャンデータも、それらのデフォルトのトピックから、シミュレーションされたロボットと OS-1 センサーにより出力されるトピックへ、リマップされます。

<remap from="scan" to="/scan_sim" />
<remap from="imu" to="/os1_cloud_node/imu" />

最後に、cartographer_occupancy_grid_node が実行されます。occupancy_grid_node は SLAM により発行された部分マップを聞き取り、それらから ROS occupancy_grid を構築し発行します。マップの作成は高価で遅いため、マップの更新は秒のオーダーとします。

この起動ファイル（launch file）は、より幅広いシステムシミュレーション起動ファイルに統合されます。

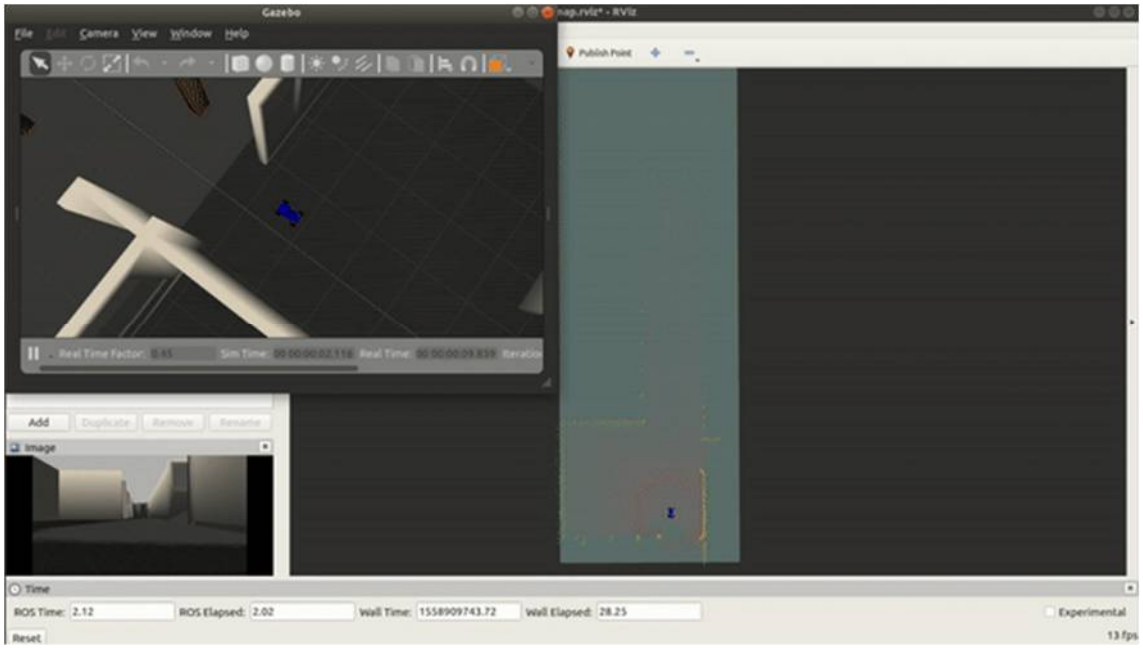
シミュレーションの実行

シミュレーション全体は、rc_laser_map.launch ファイルで定義されます。このファイルは、シミュレーションと同様に、全ての関連するサポート ROS ノードをロードし、Cartographer が機能するために必要なデータを生成します。

ファイルの最初の部分では、幾つかのパラメータのデフォルト値を定義し、シミュレーションのカスタマイズに使用することが出来ます。これらの値は、シミュレーション起動時にコマンドラインから上書きすることが出来ます。

次に、適切なワールドファイルを読み込んで、Gazebo シミュレーションが開始され、センサーと統合したロボットの誕生となります。そして、幾つかのノードがロードされ、ユーザは、キーボードやジョイスティックでロボットを制御することが出来るようになります。そして、pointcloud_to_laserscan ノードがロードされ、3D 点群が 2D レーザースキャンに変換されます。これにより、計算が更にシンプルになり、システムの柔軟性も増加します。ROS ノードには、レーザースキャンの入力のみ対応し、PointCloud2 の入力に対応していないものもあります。最後に、racer_2d_cartographer.launch ファイルが実行され、前述の特定の設定により Cartographer の起動が実行されます。

以下のビデオは、車両が環境内をナビゲートする様子を示したものです。Cartographer は RViz プラグインを提供し、部分マップ、軌跡、拘束の可視化を可能にしています。

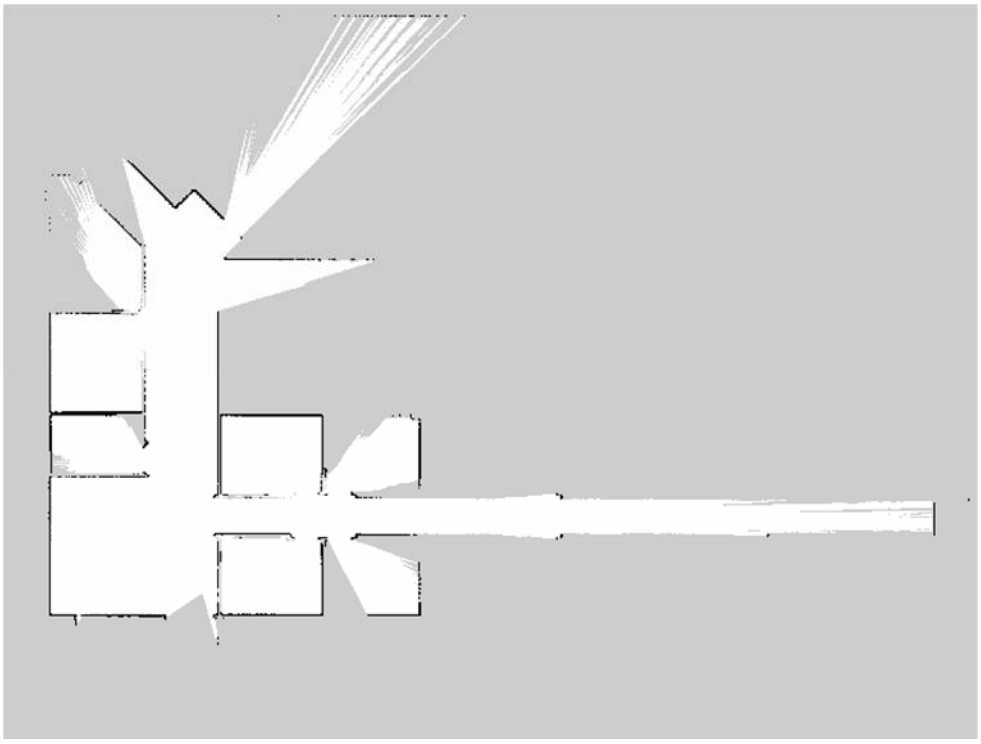


OS-1 リモコン車両 Cartographer ROS Gazebo シミュレーション

環境が十分にマップ化されると、マップファイルは保存できるようになり、後で、ロードできるようになります。これは、map_serverROS パッケージにより実行できます。

```
rosrun map_server map_saver -f /tmp/my_map
```

これにより 2 つのファイルが生成されます。YAML ファイルは、マップのメタデータを記述し、画像ファイルの名前付けを行います。画像ファイルは、占有データをエンコード化します。占有グリッドマップのサンプル画像を以下に示します。



OS-1 リモコン車両 ROS シミュレーション Cartographer マップ

実世界データ上を走る

次のステップは、シミュレーション環境から実世界の環境下での走行です。Cartographer のシステム設定は、シミュレーションで有効でしたが、実世界で収集されたデータに基づいて作業を実行することは極めて容易なことです。実世界の環境を反映する小さな変更やパラメータ調整は必要となるでしょう。手押し車やリモコンカーに搭載された OS-1-64 から収集されたデータに基づき 2D、3D モードの両方で Cartographer を実行していきます。

インドアでのデータ収集

最初のステップは、自分の環境からデータを収集することです。統合リモコンカーからすべてを実行する前に、より単純でより制御された環境から、データを収集しましょう。最初の例では、以下の写真に示すように、OS-1-64 が PC と共にカートに乗っています。

データ収集のためにカートに搭載された OS-1

カートはマニュアルでオフィス内を押して移動しました。標準の ouster_ros センサーパッケージを、センサーの設定や ROS とのインターフェースとして使用しました。

os1_cloud_node/points および os1_cloud_node/imu topics が記録されました。

ROS .bag ファイルの検証

Cartographer ROS は、cartographer_rosbag_validate という名前のツールを提供し、自分の bag の中に現在あるデータを自動的に解析していきます。正しくないデータを修正するためには、Cartographer をチューニング（調整）する前に、このツールを実行することが、一般的に良いと言えます。

Cartographer の開発者の経験を基に、bag 内にありがちな様々な間違いを検出することが出来ます。このツールは、データの品質を改善するコツを提供しています。

ツールは以下のコマンドで実行することが出来ます。

rosrun cartographer_ros cartographer_rosbag_validate -bag_filename <bag filename>

これにより、以下の出力が得られます：

```
I0526 13:11:07.732892 16925 rosbag_validate_main.cc:398] Time delta histogram for consecutive messages on t
opic "/os1_node/points" (frame_id: "os1"):
Count: 1084  Min: 0.050000  Max: 0.050000  Mean: 0.050000
I0526 13:11:07.732954 16925 rosbag_validate_main.cc:398] Time delta histogram for consecutive messages on t
opic "/os1_node/imu" (frame_id: "os1_imu"):
Count: 5424  Min: 0.009776  Max: 0.010224  Mean: 0.009999
[0.009776, 0.009821)                          Count: 6 (0.110619%)    Total: 6 (0.110619%)
[0.009821, 0.009866)                          Count: 16 (0.294985%)   Total: 22 (0.405605%)
[0.009866, 0.009910)                          Count: 27 (0.497788%)   Total: 49 (0.903392%)
[0.009910, 0.009955)                      #   Count: 349 (6.434366%)  Total: 398 (7.337758%)
[0.009955, 0.010000)                  #########   Count: 2417 (44.561211%)        Total: 2815 (51.898968%)
[0.010000, 0.010045)                  ########    Count: 2217 (40.873894%)        Total: 5032 (92.772858%)
[0.010045, 0.010089)                      #   Count: 345 (6.360620%)  Total: 5377 (99.133484%)
[0.010089, 0.010134)                          Count: 27 (0.497788%)   Total: 5404 (99.631271%)
[0.010134, 0.010179)                          Count: 14 (0.258112%)   Total: 5418 (99.889381%)
[0.010179, 0.010224]                          Count: 6 (0.110619%)    Total: 5424 (100.000000%)
```

Cartographer ROSbag 検証出力

Cartographer の実行

demo_cart_2d.launch ファイルを使って、収集されたデータを再生し、Cartographer を実行します。このファイルは、cart_2d.launch ファイルを通して、Cartographer を読み込み、demo_2d.rviz 設定ファイルで記述された設定を事前に設定して RViz を起動し、コマンドラインからの bag_filename パラメータで指定する bag ファイルを再生します。

cart_2d.launch ファイルは、os1_sensor.urdf ファイルを読み込んで、Cartographer から要求される IMU とセンサー部品の間の変換（transformations）を行います。そして、cart_2d.lua 設定ファイルを読み込んで、Cartographer ノードを起動します。シミュレーションで使用される設定ファイルからは、幾つかの変更点が存在します。:

- カートには走行距離計測（odometry）のソースが存在しない
  - use_odometry = false
- LaserScan 信号（message）ではなく、すべて PointCloud2 信号を使用
  - num_laser_scans = 0,
  - num_point_clouds = 1,

以前と同様、デフォルトの入力トピックは、cart_2d.launch ファイル中にシステムが出力するトピック名を反映してリマップされます。今我々は PointCloud2 トピックを読んでいるため、“スキャン”トピックの代わりに、“ポイント”トピックがリマップされます。

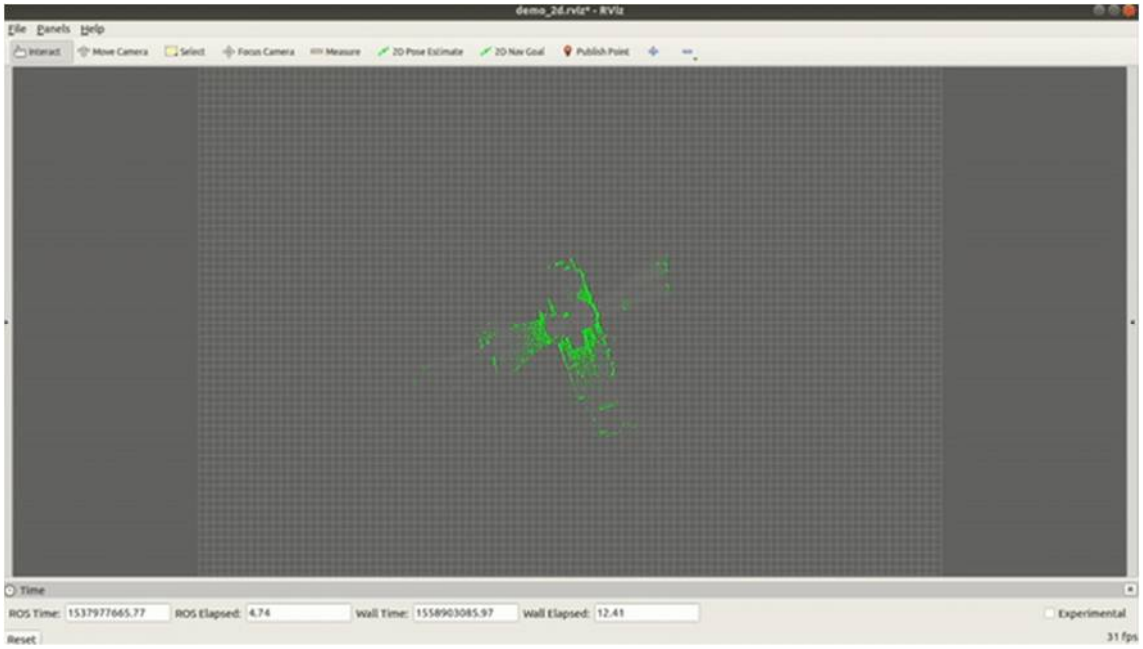
```
<remap from="points2" to="/os1_cloud_node/points" />
<remap from="imu" to="/os1_cloud_node/imu" />
```

最後に、cartographer_occupancy_grid_node を起動して、マップを作成します。

一連のプロセス一式を以下のコマンドで実行できます。:

$ roslaunch rover_2dnav demo_cart_2d.launch bag_filename:=cart.bag

以下の画像に示すように RViz によりマップ生成プロセスをモニターすることが出来ます。



RViz 上での OS-1Cartographer マップ生成

以下に完成されたマップの最終版を示します。

Cartographer により作成された OS-1 マップ

RC（リモコン）カー上で Cartographer を実行する

Cartographer は実世界のデータ上で実行できるため、Cartographer を自律型ロボティクス・システムと統合できます。この例では、OS-1-64 を RC カープラットフォームに搭載します。システムには OpenMV カメラも搭載していますが、Cartographer システムでは使用しません。車両は、X ボックスコントローラーで遠隔操作が可能です。全ての処理は、Arduino Uno 付き fitlet2 で、モーターに対してステアリングとスロットルの制御コマンドを追加すれば可能です。本セットアップは、以前のシミュレーションシステムをほぼまねしたものです。

以下は完成したシステムの画像です。

OusterOS-1 と OpenMV カメラの搭載された RC カー

RC カーを扱う際は、[diy_driverless_car_ROS]レポジトリ、特に、[rover_2dnav]パッケージが使用されます。RC カーをマニュアル操作する際は、[rc_dbw_cam.launch]起動ファイルを使用してシステムをスタートします。

2DCartographer をオンラインで実行する

rc_dbw_cam.launch ファイルは、ジョイスティックノードを読み込んで、X ボックスコントローラーから、ステアリングおよびスロットルコマンドを処理します。そして、これらは、L298N_node に送られて、Arduino Uno との交信が行われます。rc_control.launch ファイルが読み込まれ、幾つかの制御信号の変換と拡張カルマンフィルターの起動を、robot_localization を通して行われます。EKF ノードは、走行距離情報（odometry information）を計算し、その後、Cartographer にその情報が送られます。OpenMV カメラドライバーノードも、OS-1 ROS 同様に起動されます。

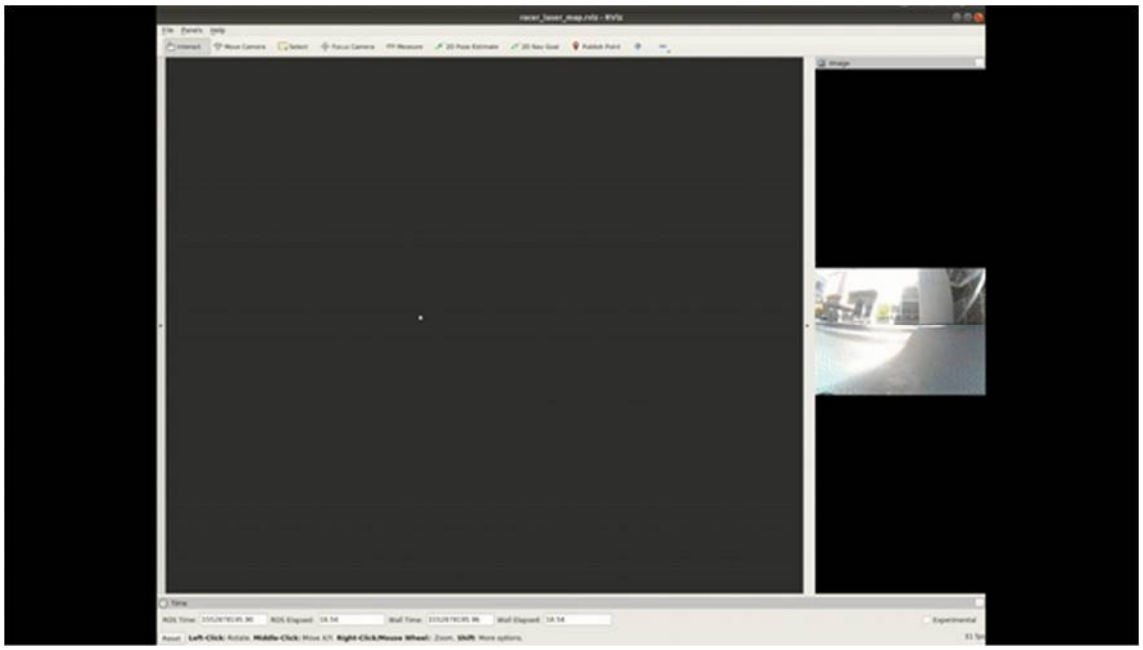
Cartographer は、rc_dbw_cam.launch ファイルの起動時は、コマンドラインから、map の記述（map argument）を True に設定することでも実行できます。これにより、racer_2d_cartographer.launch ファイルを通じて、Cartographer 同様、pointcloud_to_laserscan ノードも起動します。

前述のとおり、os1_sensor.urdf ファイルは、racer_2d.lua ファイルから、部品変換（component transforms）と設定を読み込みます。

システムは、以下のコマンドを実行することで起動できます。

```
$ roslaunch rover_2dnav racer_map.launch rviz:=true map:=true cartographer:=true
```

以下に RC カーのマニュアル操作時のオンラインマップ生成の画像を示します。



RViz 上での OS-1RC カーCartographer マップ生成

オフラインで Cartographer3D パイプラインを実行する

RC カーを操作する時、マッピングに利用されるリソースの処理量に制限を設けて、これらのリソースを、走行中であろう他の認識手段、経路決定、制御機能にも使用できるようにしたいと考えます。しかし、収集されたデータをオフラインで処理できれば、ランタイム（実行に要する時間）の制約から、より解放されます。

offline_node はセンサーデータのバッグ（bag）に対して SLAM を実行する最速の方法です。トピックは、まったく聞かずに、その代わりに、TF やコマンドラインから供給される bag 群のセンサーデータを読みに行きます。それ以外の点に関しては、cartographer_node のように機能します。

Cartographer は 2D モードでも、3D モードでも稼働します。2D パイプラインは、2D スキャンを 2D サブマップにマッチングすることで、3DoF（3自由度：x、y、ヨー）姿勢の軌跡を推定します。それに較べて、3D パイプラインは、3D スキャンを 3D サブマップにマッチングすることで、6DoF（6自由度：x、y、z、ロール、ピッチ、ヨー）姿勢を推定します。3D では、RViz は、3D ハイブリッド確率グリッドの 2D 投影のみを表示します（グレイスケールで）。

2D SLAM を使用する場合、レンジ（範囲）データは、追加の情報ソースが無くてもリアルタイムで処理できます。従って、IMU はオプションとなります。3DSLAM では、IMU を装備する必要があります。なぜならば、IMU はスキャン方向の初期推定に使用され、スキャンマッチングの煩雑さを大幅に低減してくれるからです。

RC カーデータをオフラインで起動するには、以下を実行してください：

```
$ roslaunch rover_2dnav offline_rc_3d.launch
bag_filenames:=patio_16mar2.bag rviz:=true
```

この起動ファイルは、racer_3d.lua ファイルから設定を読み込みます。この設定は、LaserScan 信号の代わりに、PointCloud2 トピックを聞きに（subscribe）いきます。

```
num_laser_scans = 0,
num_subdivisions_per_laser_scan = 1,
num_point_clouds = 1,
```

offline_rc_3d.lunach ファイルは、points2 および imu トピックもリマップするので、オフラインノードは、レーザースキャンだけでなく、点群全般を処理できます。

```
<remap from="points2" to="/os1_cloud_node/points" />
<remap from="imu" to="/os1_cloud_node/imu" />
```

Cartographer Assets Writer パイプラインの実行

Cartographer は、SLAM アルゴリズムを起動しているので、常に、ロボットの軌跡と環境は、最新・最良の推定ステータスが維持されます。これにより、最終的な軌跡とマップは、Cartographer が提供できる最も高精度な結果となります。

Cartographer は、その内部ステータスを、幾つかの.pbstream ファイルフォーマットにシリアル化し、そこに含まれるデータ構造は、Cartographer 内部で使用されます。Cartographer は、効率的に稼働できるように、処理したセンサーデータのほとんどを無視（スキップ）します。しかし、Cartographer では、cartographer_assets_writer を使用して、.pbstream ファイルに保存された軌跡情報と、.bag ファイルに保存されたオリジナルのセンサーデータを組み合わせてマッピングを行い、高解像度マップを作成することが出来ます。

Cartographer をオフラインノードで実行する場合、.pbstream ファイルが自動的に保存されます。これは、前述の RC カーの例でも確認できます。:

$ roslaunch rover_2dnav offline_rc_3d.launch
bag_filenames:=patio_16mar2.bag rviz:=true

端末上の出力では、.pbstream ファイルの保存が確認できます。

```
[ INFO] [1558995224.810463242, 1552764404.998767479]: I0527 15:13:44.000000 26867 offline_node.cc:335]
] Elapsed wall clock time: 399.675 s
[ INFO] [1558995224.810559515, 1552764404.998767479]: I0527 15:13:44.000000 26867 offline_node.cc:339]
] Elapsed CPU time: 469.331 s
[ INFO] [1558995224.810623651, 1552764404.998767479]: I0527 15:13:44.000000 26867 offline_node.cc:343]
] Peak memory usage: 96396 KiB
[ INFO] [1558995224.810678316, 1552764404.998767479]: I0527 15:13:44.000000 26867 offline_node.cc:350]
] Writing state to '/home/ouster/Downloads/patio_16mar2.bag.pbstream'...
Optimizing: Done.
[ INFO] [1558995225.327851514, 1552764404.998767479]: I0527 15:13:45.000000 26877 constraint_builder_2
2d.cc:281] 0 computations resulted in 0 additional constraints.
[ INFO] [1558995225.328047523, 1552764404.998767479]: I0527 15:13:45.000000 26877 constraint_builder_2
2d.cc:283] Score histogram:
Count: 70  Min: 0.610816  Max: 0.744464  Mean: 0.656487
[0.610816, 0.624180)                    ###    Count: 12 (17.142857%)  Total: 12 (17.142857%)
[0.624180, 0.637545)                   ####    Count: 15 (21.428572%)  Total: 27 (38.571430%)
[0.637545, 0.650910)                    ###    Count: 11 (15.714286%)  Total: 38 (54.285713%)
[0.650910, 0.664275)                   ####    Count: 13 (18.571428%)  Total: 51 (72.857140%)
[0.664275, 0.677640)                      #    Count: 4 (5.714286%)    Total: 55 (78.571426%)
[0.677640, 0.691004)                      #    Count: 3 (4.285714%)    Total: 58 (82.857140%)
[0.691004, 0.704369)                      #    Count: 2 (2.857143%)    Total: 60 (85.714287%)
[0.704369, 0.717734)                      #    Count: 2 (2.857143%)    Total: 62 (88.571426%)
[0.717734, 0.731099)                      #    Count: 4 (5.714286%)    Total: 66 (94.285713%)
[0.731099, 0.744464]                      #    Count: 4 (5.714286%)    Total: 70 (100.000000%)
[cartographer_offline_node-2] process has finished cleanly
log file: /home/ouster/.ros/log/c1388274-80cb-11e9-8d5a-b808cfa08836/cartographer_offline_node-2*.log
```

OS-1RC データをオフラインで処理する Cartographer

あるいは、オンラインモードでの実行時では、通常の Cartographer サービスを使って、現在（カレント）の軌道を明確に終了することで、Cartographer にカレントのステータスをシリアル化させます。

前述の通り、以下のコマンドで Cartographer をオンラインで実行できます。:

$ roslaunch rover_2dnav racer_map.launch rviz:=true map:=true cartographer:=true

まず、軌跡を終了します。

$ rosservice call /finish_trajectory 0

Cartographer に、カレントのステータスを、.pbstream ファイルにシリアライズするように指示します。

$ rosservice call /write_state xyz_your_file_name.pbstream

これにより、.pbstream ファイルも作成されます。

.pbstream ファイルができれば、パイプライン（主要）のコンフィグレーションは、.lua ファイルで設定して、データ処理を制御できます。このパイプラインを使用して、SLAM 点群データを、色付け、フィルター処理、様々なフォーマットで出力できます。詳細は、Exploiting the map generated by Cartographer ROS を参照してください。

assets writer は、PointsProcessors のパイプラインとして作られています。PointsBatchs は各プロセッサを流れ、それぞれすべてが、プロセッサを通り過ぎる前に、PointsBatch を変更できる機会があります。利用可能な PointsProcessors は、全て、cartographer/io サブディレクトリに定義され、各ヘッダファイルに記録されています。

パイプラインを作成するため、以下のコンポーネントを使用します。

- min_max_range_filter：センサーから近すぎるか、遠すぎるポイントを除去
    - action = ″min_max_range_filter″,
    - min_range = 1.,
    - max_range = 60.,
- voxel_filter_and_remove_moving_objects：データにボクセルフィルターをかけ、動いていないと思われるオブジェクトのポイントのみを通過させる
    - action = ″voxel_filter_and_remove_moving_objects″,
    - voxel_size = VOXEL_SIZE,
- write_xray_image：半透明の“X-線”マップビューを作成します。ここでは、‘ボクセルサイズ’の大きさの「ピクセル付きポイント」が用いられ、マップの X および Y 平面のみ閲覧できます。
    - action = ″write_xray_image″,

    - voxel_size = VOXEL_SIZE,
    - filename = ″xray_xy_all″,
    - transform = XY_TRANSFORM,
- intensity_to_color：ポイントの反射強度を使い、ポイントをカラー化できます。線形変換を適用して、強度値を[0, 255]に落とし込み、そして、この値をポイントの RGB として使用します。以降のパイプラインの全ステージは、カラーのポイントとなります。

    - write_ply to stream a PLY file to disk.
    - action = ″write_ply″,
    - filename = ″points.ply″,
- write_probability_grid：指定の解像度（resolution）で確率グリッド（probability grid）を作成

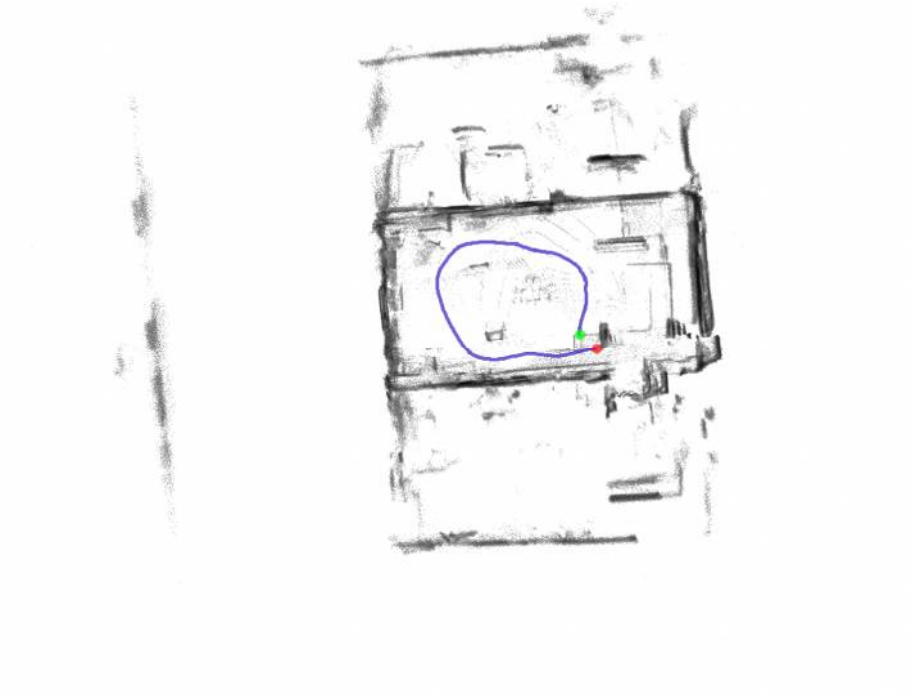
    - draw_trajectories = true,
    - resolution = 0.05,
    - range_data_inserter = {
        - insert_free_space = true,
        - hit_probability = 0.55,
        - miss_probability = 0.49,},
    - filename = ″probability_grid″

完成されたパイプライン一式を、assets_writer_rc_3d.lua 設定ファイルから閲覧することが出来ます。

assets_writer_rc_3d.launch 起動ファイルでパイプラインを実行することが出来ます。この起動ファイルのパスとして、.pbstream ファイルを記述することにご注意ください。

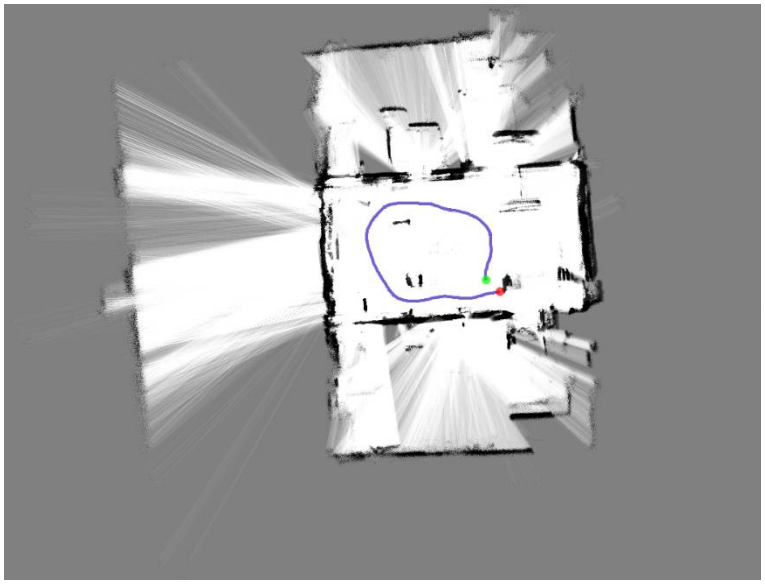
$roslaunch rover_2dnav assets_writer_rc_3d.launch
bag_filenames:=patio_16mar2.bag
pose_graph_filename:=patio_16mar2.bag.pbstream

assets writer パイプラインは、以下のような X 線画像を作成します。



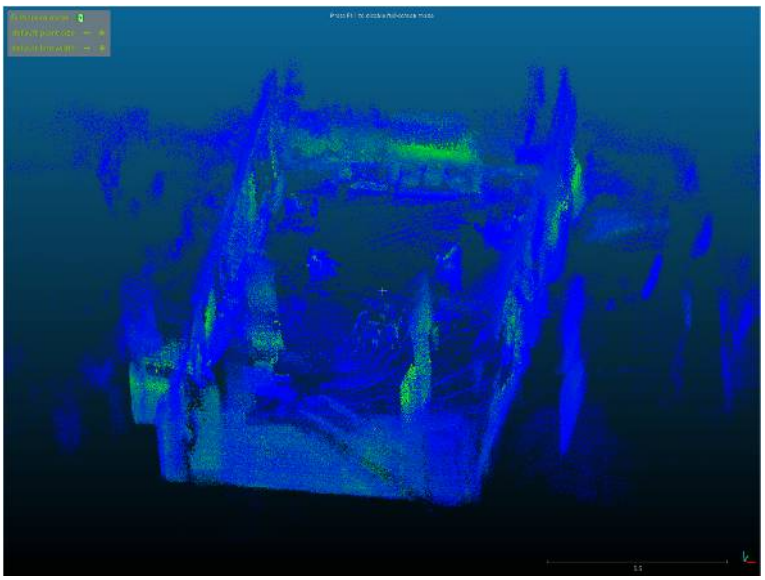
OS-1RC カーからの CartographerX線画像

assets writer パイプラインは、確率グリッドも作成できます。



OS-1RC カーからの Cartographer 確率グリッド

そして、CloudCompare のような点群ビューワから、.ply ファイルを閲覧できます。



OS-1RC カーの Cartographer .ply ファイルを CloudCompare で見る

自己位置推定のみ

localization-only（自己位置推定のみ）モードで Cartographer を使用して、計算時間を削減することが出来ます。これには、環境マップがすでに出来上がっていることが前提となります。Cartographer は既存のマップに対して、ＳＬＡＭアルゴリズムを実行しますが、新しいマップは作成できません。

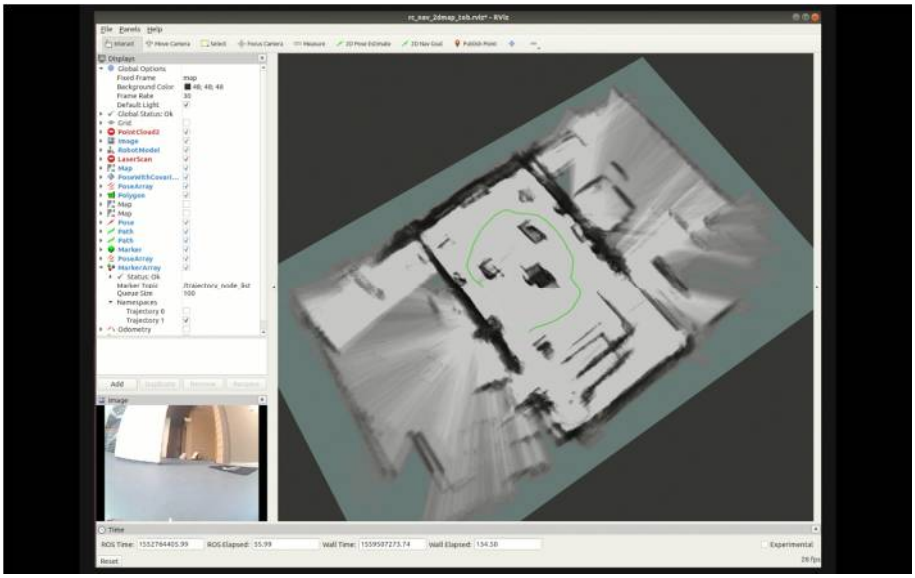
このモードを有効化するには、以下の設定の追加が必要になります。:

```
TRAJECTORY_BUILDER.pure_localization = true
POSE_GRAPH.optimize_every_n_nodes = 20
```

これは、racer_2d_localization.lua 設定ファイルに反映されます。

そして、racer_2d_cartographer_localization.launch 起動ファイルを使用して、新しい設定で Cartographer を実行します。この起動ファイルは、load_state_filename パラメータを設定する必要があります。このファイル名は、前述で作成された.pbstream ファイルです。

自己位置推定は、車両の自律走行を有効化するために有益なことです。以下の画像では、RC カーが環境内を移動した時の自己位置推定の軌跡を緑色で表示し、Cartographer が OS-1 ライダーデータを使用して車両の軌跡を推定しています。



RViz 上における Cartographer の OS-1 と RC カーの自己位置推定

Docker Container での Cartographer の実行

Docker のインストール説明書は、Docker documentation から取得出来ます。オプションではユーザは、Docker を Linux Ubuntu プラットフォームに、提供された install-docker.sh スクリプトを使ってインストールすることができます。このスクリプトは以下のコマンドで実行します：

```
$ ./install-docker.sh
```

Docker のインストールバージョンは、以下でチェック出来ます：

```
$ docker --version
```

コンテナを実行する前に、コンピュータ上でローカルに画像を構築する代わりに、Docker Hub から最新画像を取得することも出来ます。run-docker.sh が提供され、これにより、最新画像を取得でき、スクリプト経由でコンテナを実行できます。run-docker.sh スクリプトの実行により、最新の Docker コンテナを引き出し、実行できます。

```
$ ./run_docker
```

コンテナ内部に入ると、利用できる起動ファイルが全て実行できます。まず、OusterOS-1-64 データのサンプルを Docker コンテナの/root/bags/ディレクトリにコピーします。

```
$ cd /root/bags
```

$ curl -O https://data.ouster.io/downloads/office_demo_9_25_19.bag

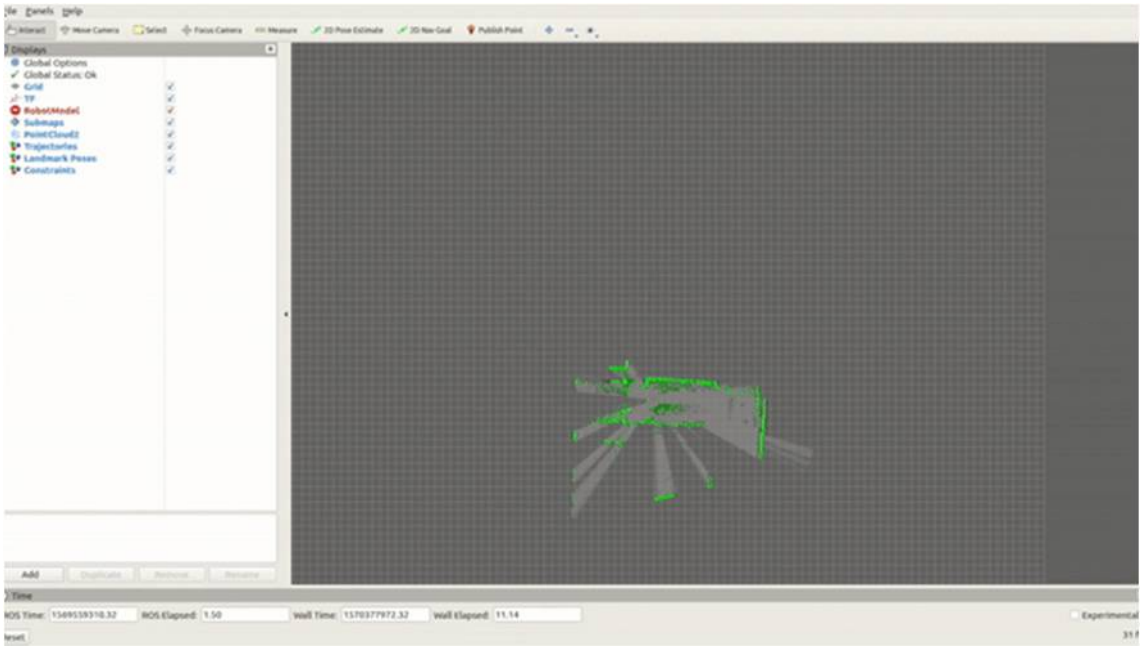
最初に、cartographer_rosbag_validate ノードを使い、bag の品質を調べます。

$ rosrun cartographer_ros cartographer_rosbag_validate -bag_filename /root/bags/office_demo_9_25_19.bag

そして、Cartographer をオフラインで実行して、.pbstream ファイルを作成します。

$ cd /root/catkin_ws/src/ouster_example/cartographer_ros/launch

$ roslaunch offline_cart_2d.launch bag_filenames:=/root/bags/office_demo_9_25_19.bag



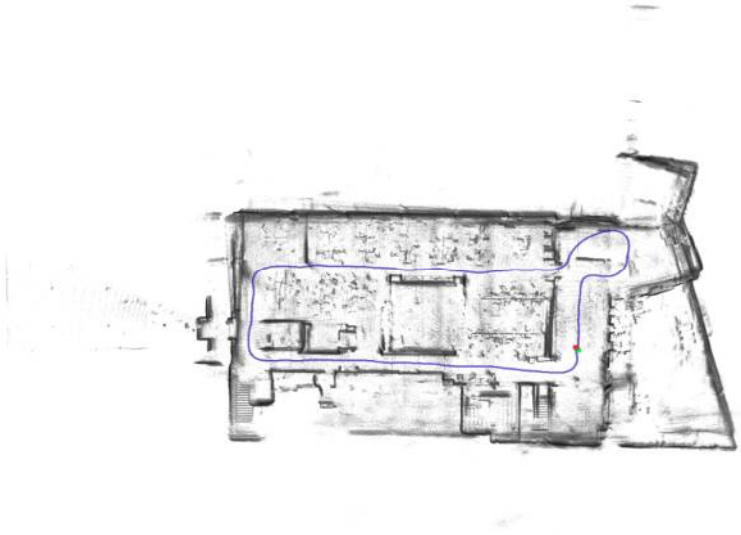
RViz 上で OS-1 Cartographer マップを作成する

最後に、assets_writer_cart_2d.launch ファイルを使って、マップ画像を作成します。

$ roslaunch assets_writer_cart_2d.launch bag_filenames:=/root/bags/office_demo_9_25_19.bag pose_graph_filename:=/root/bags/office_demo_9_25_19.bag.pbstream

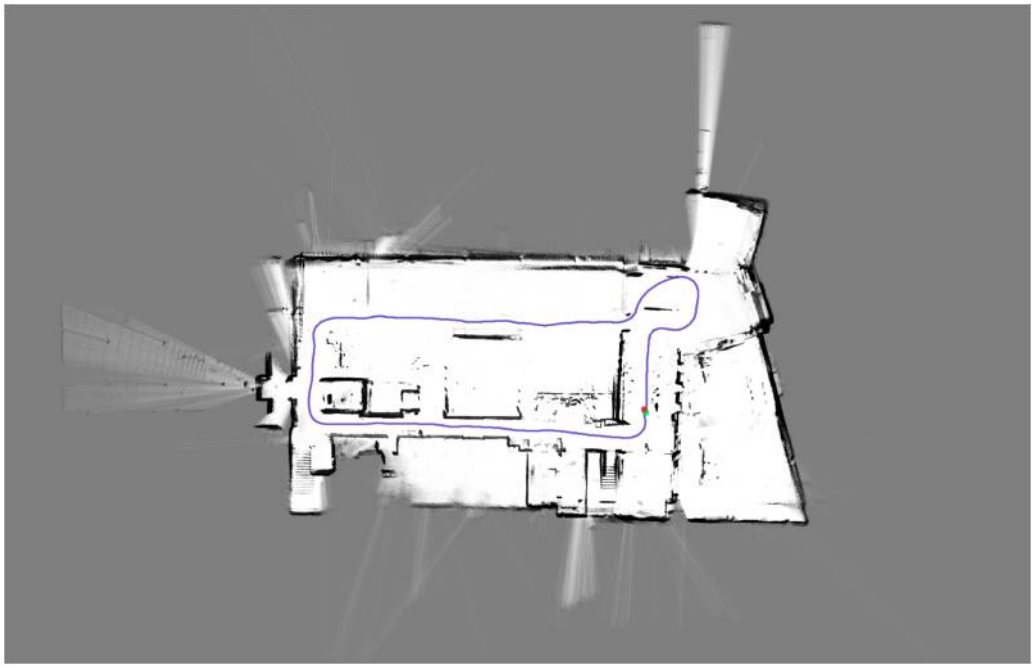
これで、出力 png ファイルが閲覧できるようになります。

$ xdg-open office_demo_9_25_19.bag_xray_xy_all.png



OS-1-64 からの CartographerX線画像

$ xdg-open office_demo_9_25_19.bag_probability_grid.png



OS-1-64 からの Cartographer 確率グリッド

最後となりますが、「OS1 と Google Cartographer」に関する.上記のガイダンスが、皆様に有益な情報になればと思います。今後、Ouster 社のフォーラム forum.ouster.at や、オンラインのリソース resources をチェックして下さい。