

ロボットの作り方～移動ロボットの制御とROSによる動作計画実習～

セミナーに必要な物品の準備

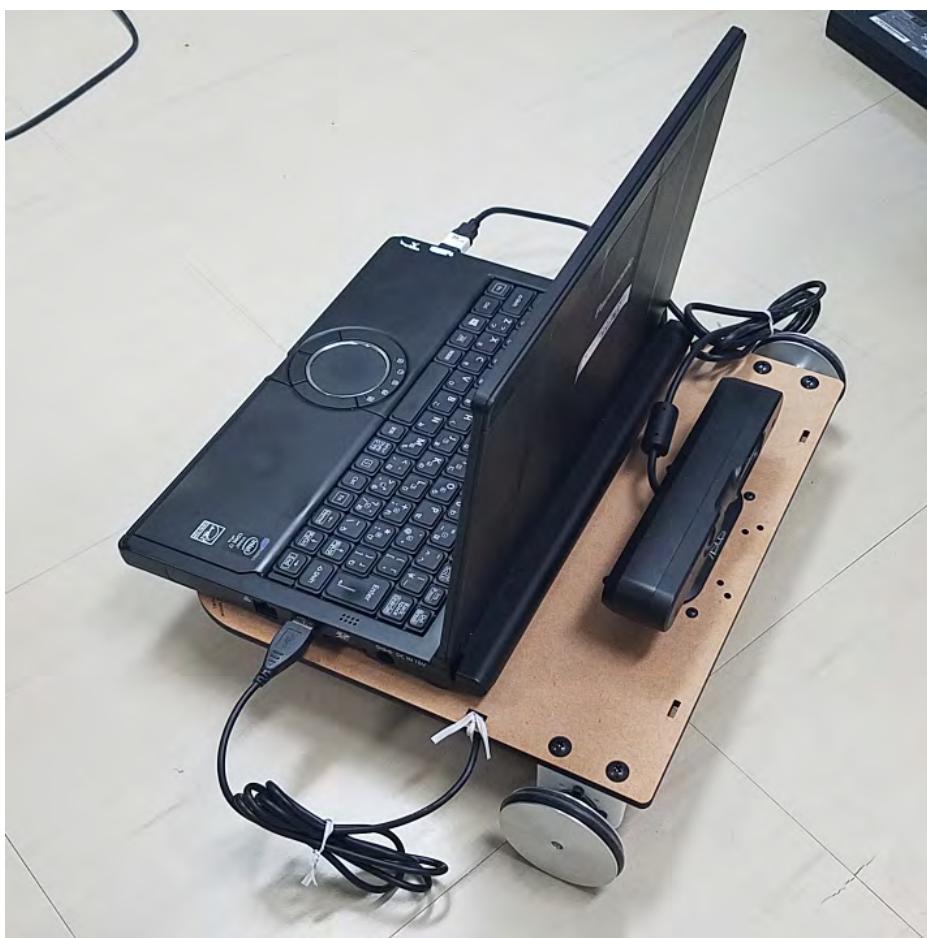
下記3点を必ず持参してください。

- 実習に使用するノートPC
 - Ubuntu Linux(64bit必須)とROSを事前にインストールしてください
 - インストール方法は以下の[事前準備](#)を参考にしてください
 - USBポートを2つ以上持つタイプか、ポート数が足りない場合はUSBハブをご用意ください
 - LANポートを1つ持つタイプか、ない場合はUSB-LANアダプタをご用意ください
- USB A-miniBケーブル(長さ50cm程度)2本
- 単三乾電池 8本(充電池も可)
 - たくさん動かすと2日間持たない場合がありますので、予備の電池を追加で8本程度ご用意頂くと安心です。なお、会場付近に電気店がありますので、当日そちらで購入することも可能です。

下記3点は、可能であれば持参してください。(用意できない場合は貸し出します。)

- プラスドライバ(M3～M5のネジを締められるもの)
- 六角レンチセット
- モバイルルーターなどのインターネット接続機器(会場でもWiMAX回線を用意しますが、速度が遅くなる可能性があるため、可能であれば持参をお願いします。)

教材として用いるロボットは*i-Cart edu*です。PCが載るスペースとしては幅約32cm 奥行き約20cm程度となります。もちろんこの大きさ以上のPCでも構いませんが選定の目安としてください。[Lets' note CF-SX4](#)とXtion PRO Liveを搭載した様子は以下のようになります。



不明な点や、事前準備がうまく行かない点については、オーガナイザまでお問い合わせ下さい。

事前準備

- [Ubuntu LinuxとROSのインストール](#)
- [Linuxの基本操作](#)

スケジュール

1日目 (6/16)

09:30	受付開始
10:00- 12:00	実習1-1 ロボットの組み立てと動作テスト

12:00-	昼休み
13:00-	講義1 「モータ制御・ロボット動作制御の理論」講師：渡辺敦志
14:00-	(SEQSENSE)
14:00-	実習1-2 ROSプログラミングの基本
15:00-	
16:30	実習1-3 ROSを用いた点群取得
16:00-	実習1-4 ROSの便利機能、三次元センサの利用 YVT-35LXの場合、
17:30	Xtion Pro Liveの場合

2日目 (6/17)

08:30	開場
10:00-	講義2 「ROSを用いた自律移動ロボットのシステム構築」講師：渡辺敦志 (SEQSENSE)
11:00	
11:00-	実習2-1 ROS Navigationパッケージの利用
12:00	
12:00-	昼休み
13:00-	
14:00-	実習2-2 3次元点群の処理、PointCloudに対するフィルタ
15:00-	
14:00-	実習2-3 点群処理とロボットナビゲーションの統合
15:00-	
15:00-	課題と質疑
17:30	

スケジュールは、演習の進行に応じて多少変更する場合があります。

テキスト

配布資料の[正誤表](#)

参考情報

- [ROS Japan UG](#) (日本のユーザ会)
- [ROSのメッセージボード](#)
- [ROS Answers](#) (日本語でも大丈夫です。)
- [Programming Robots with ROS](#)

Ubuntu LinuxとROSのインストール

セミナー中に開発環境としてUbuntu Linuxを利用します。そして、Ubuntu Linux上でROSを利用します。本ページではUbuntu LinuxとROSのインストール方法を示します。

用意するもの

- ノート型パソコン - 本手順によりパソコンの既存のOS（Windows等）及び保存されているデータやソフトウェアは完全に削除されます。予めバックアップを行ってください。
- 容量4GB以上の空のUSBメモリ
- インターネット接続

手順

Ubuntu Linuxのダウンロード

1. 下記URLからUbuntu Linux 16.04(64bit)のインストールイメージをダウンロードします。

[Ubuntu ダウンロード](#)

The screenshot shows the Ubuntu 16.04.4 LTS (Xenial Xerus) download page. At the top right, the word "ubuntu" is written in white on an orange background. Below it, the text "Ubuntu 16.04.4 LTS (Xenial Xerus)" is displayed in large, bold, black font. The main heading "Select an image" is centered above a list of download options. A red box highlights the "64-bit PC (AMD64) desktop image" link, which is followed by the text "クリックしてダウンロード開始".

Select an image

Ubuntu is distributed on two types of images described below.

Desktop image

The desktop image allows you to try Ubuntu without changing your computer at all, and at your option to install it permanently later. This type of image is what most people will want to use. You will need at least 384MiB of RAM to install from this image.

There are two images available, each for a different type of computer:

64-bit PC (AMD64) desktop image クリックしてダウンロード開始

Choose this to take full advantage of computers based on the AMD64 or EM64T architecture (e.g., Athlon64, Opteron, EM64T Xeon, Core 2). If you have a non-64-bit processor made by AMD, or if you need full support for 32-bit code, use the i386 images instead. Choose this if you are at all unsure.

32-bit PC (i386) desktop image

For almost all PCs. This includes most machines with Intel/AMD/etc type processors and almost all computers that run Microsoft Windows, as well as newer Apple Macintosh systems based on Intel processors.

Live USBの作成

1. 下記URLから、Live USB作成ソフトをダウンロードします。
 - Windows、Mac OS Xの場合：<https://unetbootin.github.io/>

ページ中の、Live USB作成に使用しているPCのOSを選択してください。

UNetbootin

[Features](#) [Using](#) [Supported Distributions](#) [FAQs](#) [License](#) [Wiki](#)



[Download
\(Windows\)](#)



[Download
\(Linux\)](#)



[Download
\(Mac OS X\)](#)



[Donate via Paypal](#)



[Donate via Bitcoin](#)



[Donate via Venmo](#)

- Linux (Debian/Ubuntu) の場合: 下記コマンドを実行

1

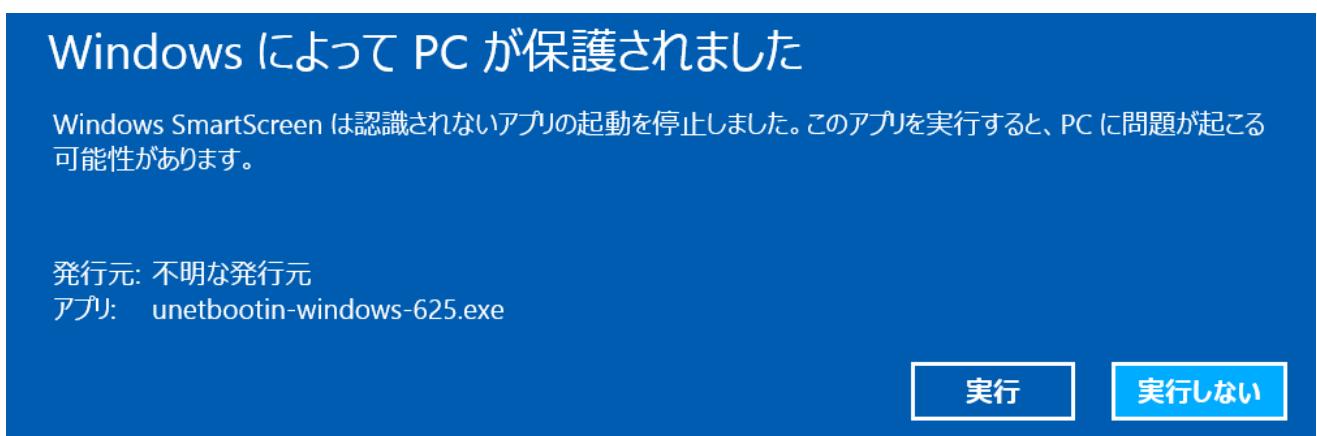
```
$ sudo apt-get install unetbootin
```

2. 事故防止のため、使用しないUSBメモリやメモリーカードを取り外し、使用するUSBメモリのみを接続します。使用するUSBメモリは、ファイルが入っていない、空の状態にして下さい。

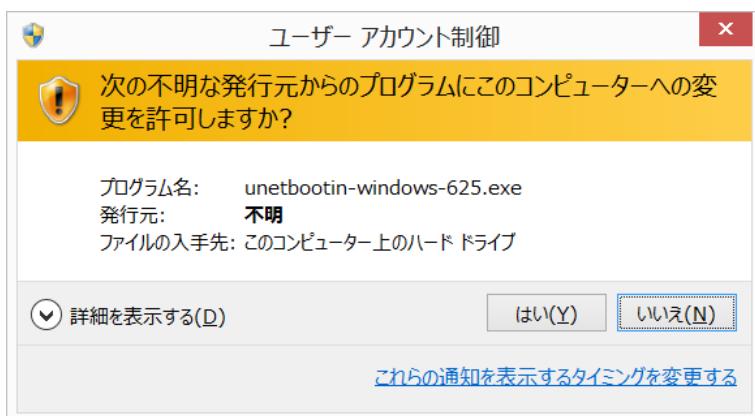
3. Live USBを作成するPCとUbuntuをインストールするPCが別でも構いません。

4. ダウンロードした unetbootin-windows-???.exe (Windowsの場合) を実行します。

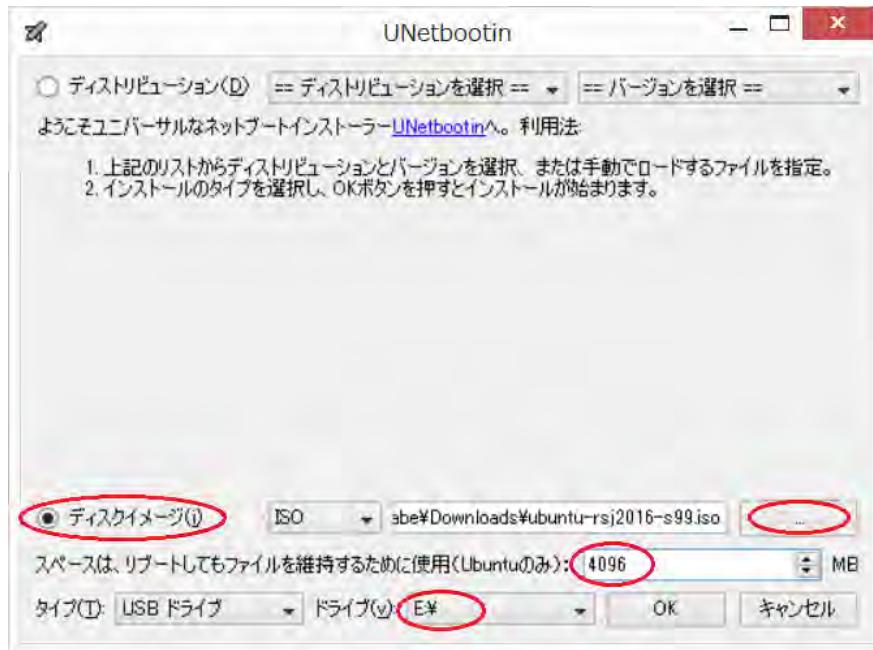
下記、「WindowsによってPCが保護されました」画面が現れた場合は、「実行」ボタンをクリックしてください。



また、下記のユーザーアカウント制御画面が現れた場合、「はい」をクリックしてください。



5. UNetbootinの画面で「ディスクイメージ」を選択し、「...」ボタンをクリックして先ほどダウンロードした、ubuntu-16.04.2-desktop-amd64.isoファイルを選択します。また、「スペースは、リブートしてもファイルを維持するために使用」欄に「4096」と入力し、「ドライブ」欄で、使用するUSBメモリのドライブ名を選択します。内容を確認後、「OK」をクリックしてください。



完了まで、しばらく待機します。（USB2.0の場合10分以上、書き込み速度の遅いメモリだと30分程度かかる場合があります。）下記の「永続性を設定する」画面で、応答なしと表示される場合がありますが、正常に動作していますので、そのまま待機してください。

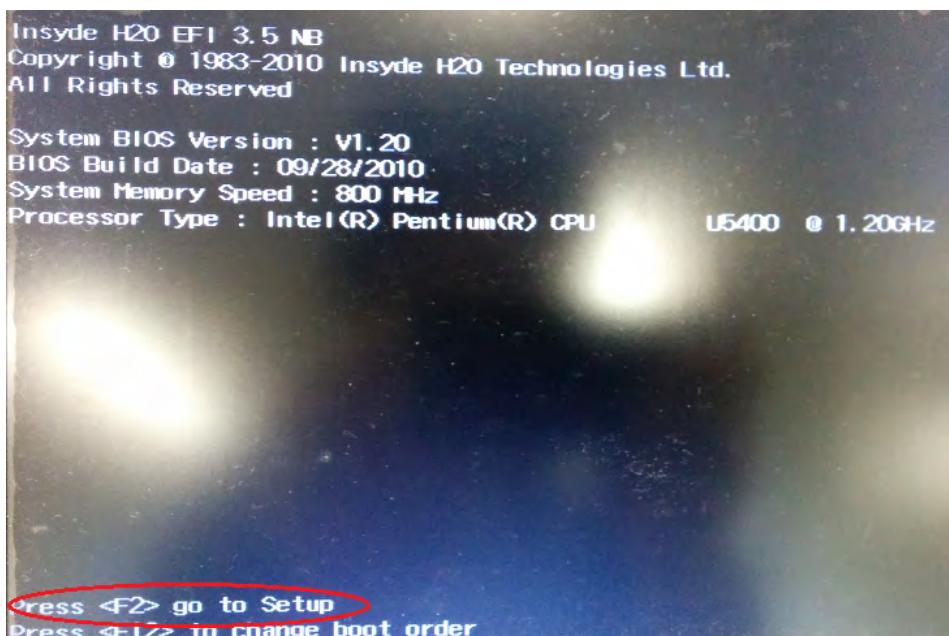


下記画面が表示されれば、「Live USB」の作成は完了です。終了をクリックして下さい。

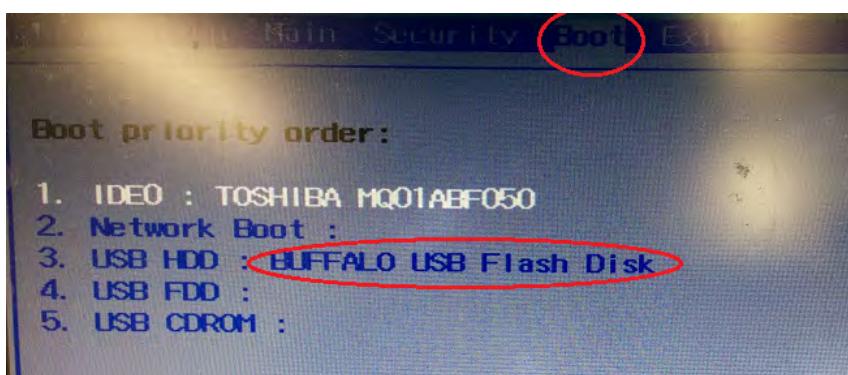


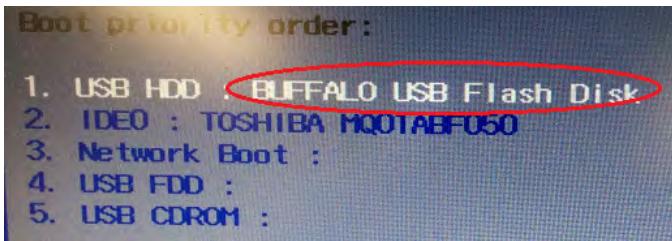
6. Live USBから起動するためのBIOSの設定を行います。

セミナーで使用するPCの電源を切り、下記の手順で作成したLive USBを接続した状態で起動します。起動時に、BIOS設定画面に入ります。PCのメーカー毎に入り方が異なりますので、マニュアル等で確認してください。図はAcerの例です。

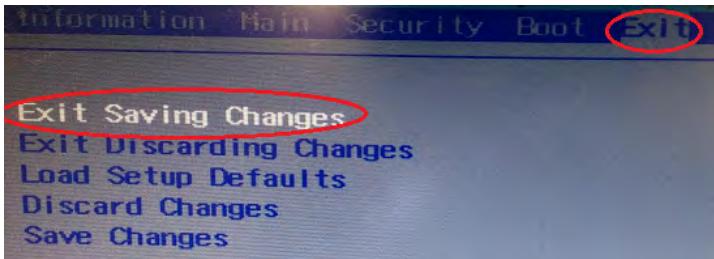


BIOS設定画面に入ったら、起動順(Boot order, Boot priority)の設定で、USBメモリが最優先になるように設定します。(表示は使用しているPCおよびUSBメモリのメーカーによって異なります。)





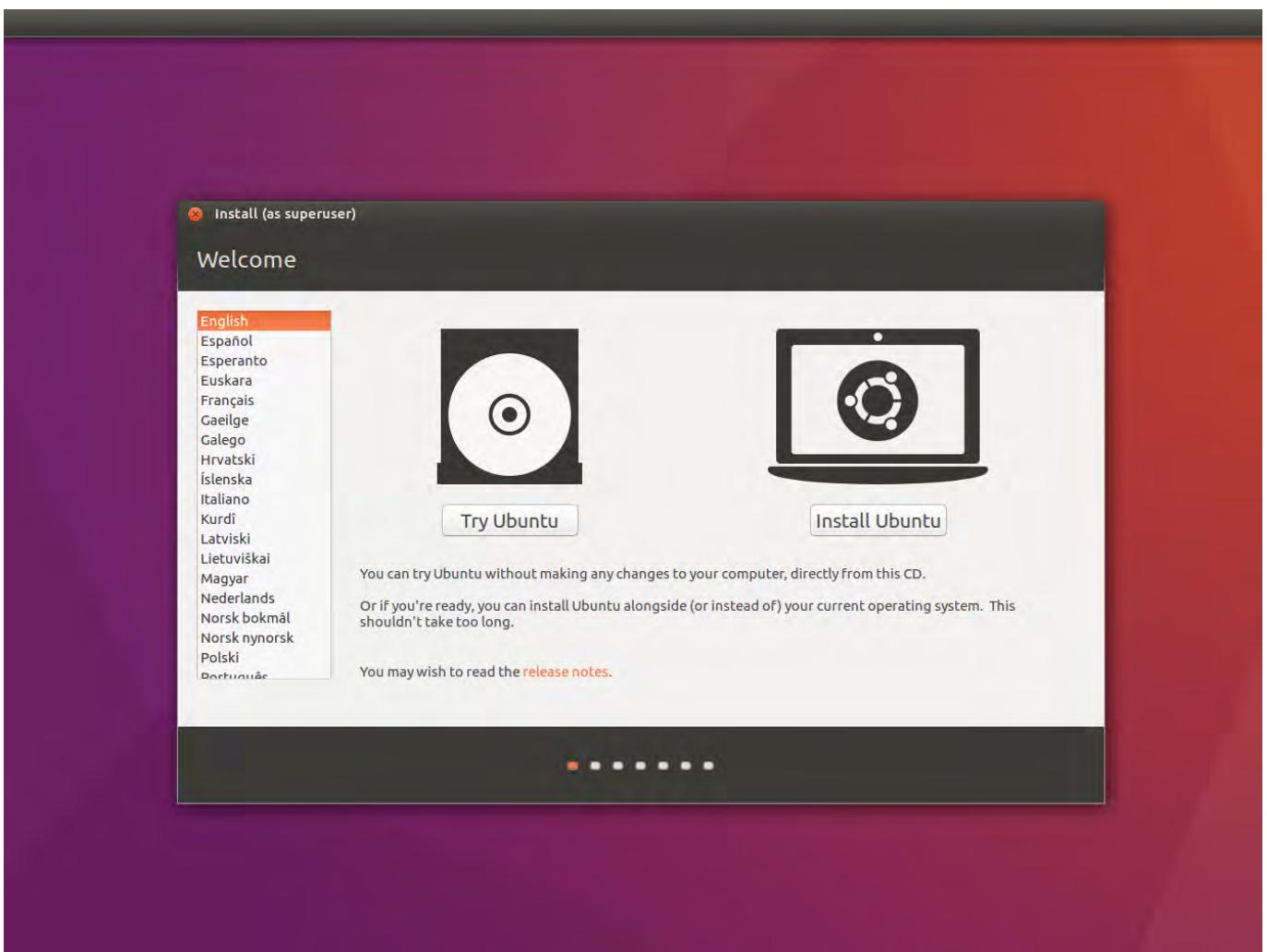
設定を保存して再起動します。



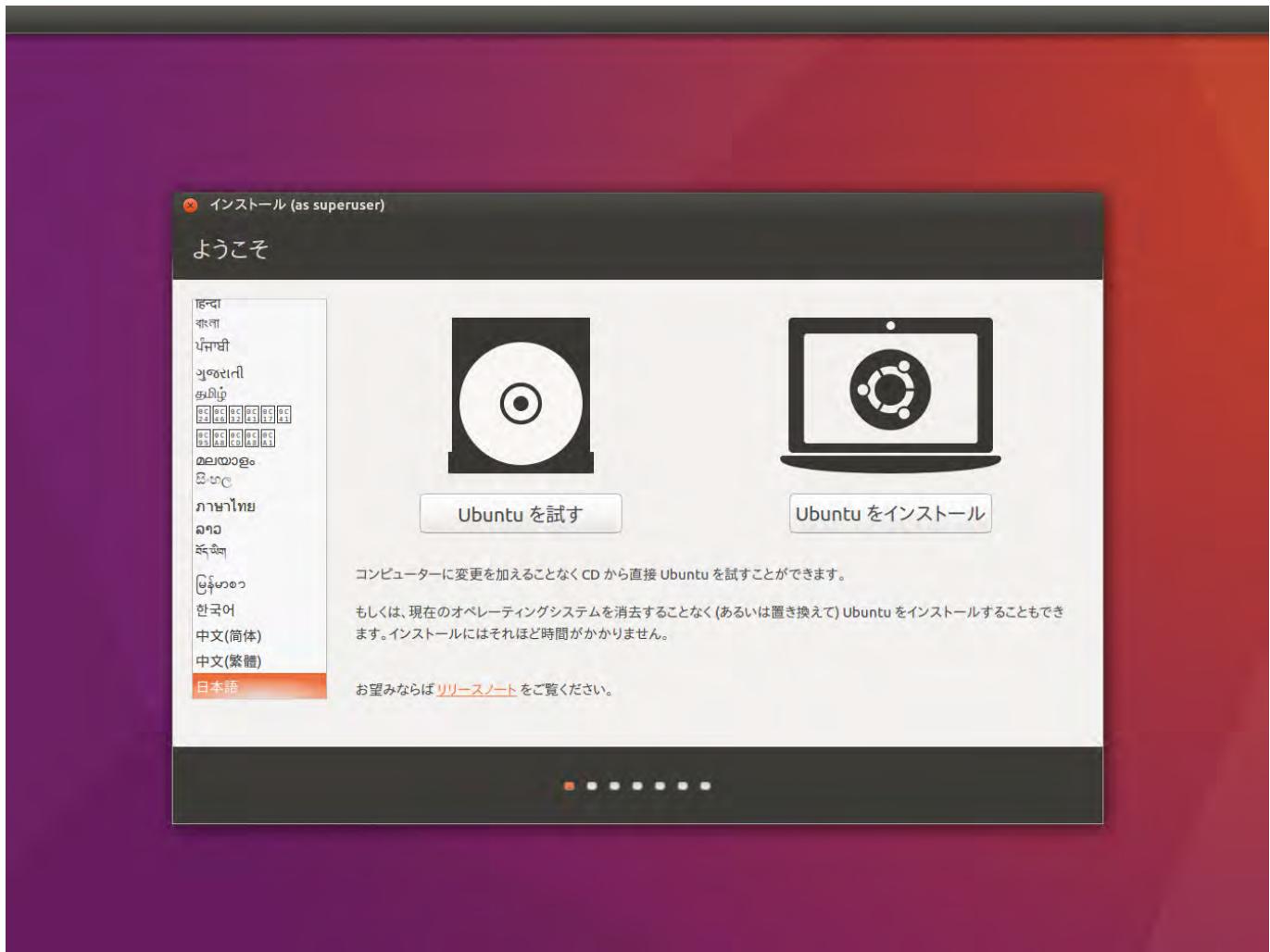
Ubuntu Linuxのインストール

1. Live USBをパソコンに接続し、パソコンの電源を入れます。

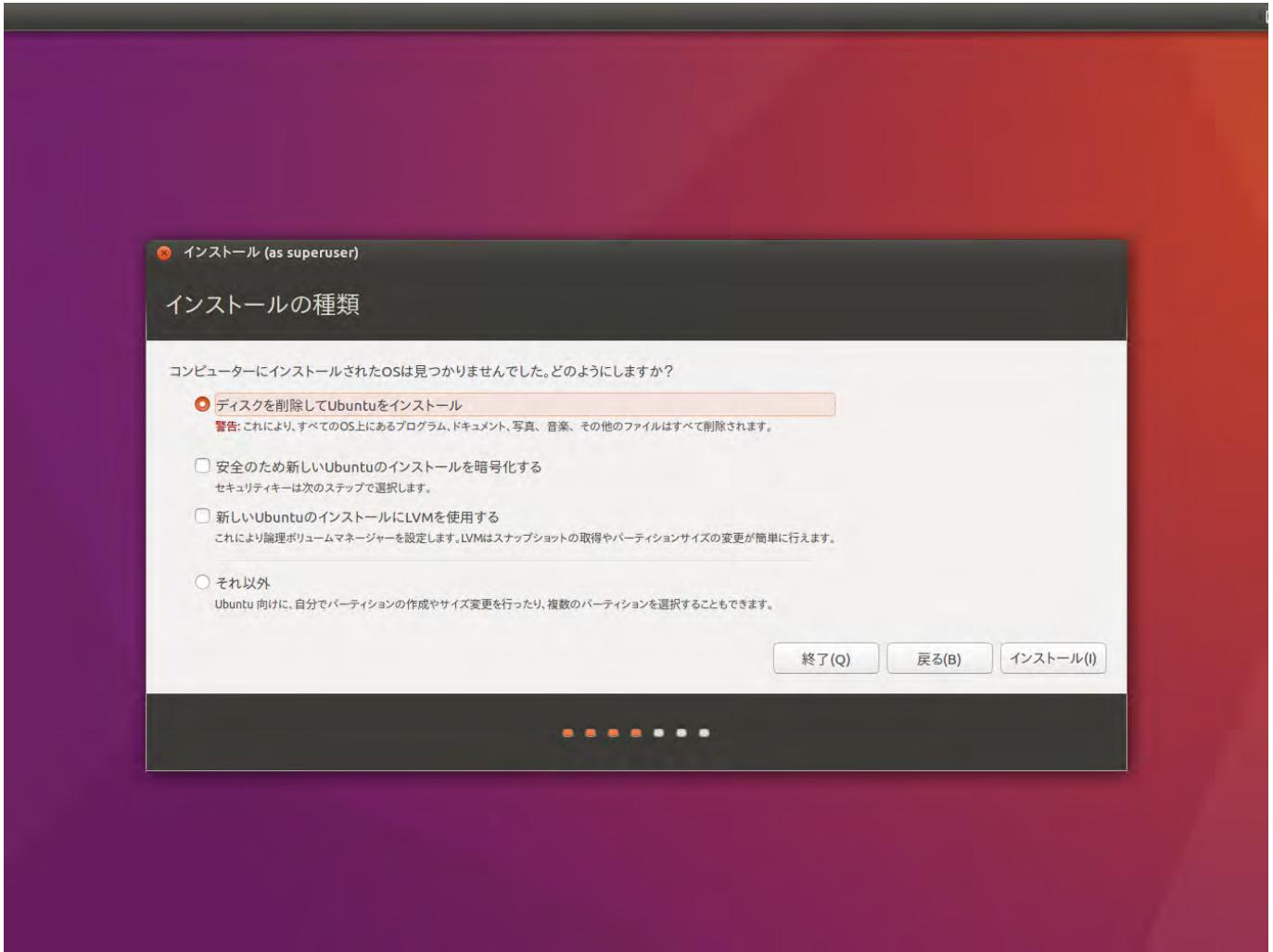
2. 以下の画面が表示されます。言語を選択してください。



3. 「Ubuntuをインストール」を選択しインストール手順を開始します。画面に出る説明に従ってインストール手順を続けてください。



4. 以下の画面に届いたら、「ディスクを削除してUbuntuをインストールする」を選択してください。



5. インストール後、LiveUSBを外してパソコンを再起動すると以下の画面が現れます。これでUbuntu Linuxのインストールが完了です。



ROSのインストール

1. *ROS Kinetic Kame*をインストールします。

ROSをインストールするためには、以下のURLで書いてある手順に従ってください。すべてのデスクトップ環境のインストールを行ってください。

BOS KineticのUbuntuへのインストール

以下はインストールコマンドの概要だけです。上記のページをご参照ください。

```
1 $ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
2 $ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 \
3   --recv-key 421C365BD9FF1F717815A3895523BAE8B01FA116
4 $ sudo apt-get update
5 $ sudo apt-get install ros-kinetic-desktop-full
6 $ sudo rosdep init
7 $ rosdep update
8 $ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
9 $ source ~/.bashrc
10 $ sudo apt-get install python-rosinstall
```

2. 確認のため、新しい端末を起動して下記を実行してください。

```
1 $ printenv | grep ROS
```

下記が出力されたら、ROSのインストールが完了しました。

```
1 ROS_ROOT=/opt/ros/kinetic/share/ros  
2 ROS_PACKAGE_PATH=/opt/ros/kinetic/share  
3 ROS_MASTER_URI=http://localhost:11311  
4 ROSLISP_PACKAGE_DIRECTORIES=  
5 ROS_DISTRO=kinetic  
6 ROS_ETC_DIR=/opt/ros/kinetic/etc/ros
```

必要なパッケージのインストール

最後に、本セミナーに必要なパッケージをインストールします。以下のコマンドを実行しインストールしてください。

```
1 $ sudo apt update
2 $ sudo apt install ros-kinetic-slam-gmapping ros-kinetic-mouse-teleop
3 $ sudo apt install ros-kinetic-openni2-camera
4 $ sudo apt install ros-kinetic-openni2-launch
5 $ sudo apt install ros-kinetic-urg-node
6 $ sudo apt install ros-kinetic-hokuyo03d
7 $ sudo apt install ros-kinetic-pointcloud-to-laserscan
8 $ sudo apt install ros-kinetic-map-server ros-kinetic-move-base
9 $ sudo apt install ros-kinetic-amcl ros-kinetic-dwa-local-planner
10 $ sudo apt install ros-kinetic-navigation
```

ModemManagerを削除

```
1 sudo apt-get purge modemmanager
```

これは、ロボット・センサのUSBが接続されたときに、ModemManagerが起動していると1~2分間ほどモデムとしての通信試行処理が実行され、その間デバイスが利用できない問題があるためです。そのため、削除、起動させない、または、USBデバイスをModemManagerのblacklistに追加する、といった処置が必要になります。

以上、開発環境の構築が完了しました。

ROSの基本操作

基本的なROS上で動くプログラムの書き方とビルド方法を学習します。

基本的な用語

パッケージ

ノードや設定ファイル、コンパイル方法などをまとめたもの

ノード

ROSの枠組みを使用する、実行ファイル

メッセージ

ノード間でやりとりするデータ

トピック

ノード間でメッセージをやりとりする際に、メッセージを置く場所

ノード、メッセージ、トピックの関係は以下の図のように表せます。



基本的には、ソフトウェアとしての ROS は、ノード間のデータのやりとりをサポートするための枠組みです。加えて、使い回しがきく汎用的なノードを世界中の ROS 利用者で共有するコミュニティも、大きな意味でのROSの一部となっています。

ソースコードを置く場所

ROS ではプログラムをビルドする際に、catkinというシステムを使用しています。また、catkinはcmakeというシステムを使っており、ROS 用のプログラムのパッケージ毎にcmakeの設定ファイルを作成することで、ビルドに必要な設定を行います。

以下の手順で本作業用の新しいワークスペースを作ります。

```

1 $ mkdir -p ~/catkin_ws/src
2 $ cd ~/catkin_ws/src
3 $ catkin_init_workspace
4 Creating symlink "/home/[ユーザディレクトリ]/catkin/src/CMakeLists.txt"
5   pointing to "/opt/ros/kinetic/share/catkin/cmake/toplevel.cmake"
6 $ ls
7 CMakeLists.txt
8 $ cd ..
9 $ ls
10 src
11 $ catkin_make
12 Base path: /home/【ユーザ名】/catkin_tmp
13 Source space: /home/【ユーザ名】/catkin_tmp/src
14 Build space: /home/【ユーザ名】/catkin_tmp/build
15 Devel space: /home/【ユーザ名】/catkin_tmp/devel
16 Install space: /home/【ユーザ名】/catkin_tmp/install
17 .
18 $ ls
19 build devel src
  
```

catkin_wsディレクトリ内にある、build、develは、catkinシステムがプログラムをビルドする際に使用するものなので、ユーザが触る必要はありません。

catkin_ws/srcディレクトリはROS パッケージのソースコードを置く場所で、中にあるCMakeLists.txt はワークスペース全体をビルドするためのルールが書かれているファイルです。

このディレクトリにypspur-coordinatorをROSに接続するためのパッケージypspur_rosをダウンロードします。（今回は説明のためypspur_rosのソースコード入手しましたが、aptを利用してバイナリーのみのインストールも可能です。）

```

1 $ cd ~/catkin_ws/src
2 $ git clone https://github.com/openspur/ypspur_ros.git
3 $ ls
4 CMakeLists.txt ypspur_ros
  
```

git はソースコードなどの変更履歴を記録して管理する、分散型バージョン管理システムと呼ばれるものです。今回のセミナーでは詳細は触れませんが、研究開発を行う上では非常に有用なシステムですので利用をお勧めします。公式の解説書、[Pro Git](#)などを参考にして下さい。

GitHub はソースコードなどのリポジトリーサービスです。オープンソースソフトウェアの開発、共同作業及び配布を行うためによく利用されていて、ROS ではソースコードの保存と配布する場所としてもっとも人気なサービスです。バイナリーパッケージとして配布されている ROS パッケージ以外を利用する場合、GitHub を使います。URL が分かれば上の手順だけで簡単に ROS のパッケージを自分のワークスペースにインポートし利用することができます。

では、次にパッケージのディレクトリ構成を確認します。ダウンロードしているパッケージがバージョンアップされている場合などには、下記の実行例とファイル名が異なったり、ファイルが追加・削除されている場合があります。

```

1 $ cd ~/catkin_ws/src/ypspur_ros/
2 $ ls
3 CMakeLists.txt msg package.xml src
4 $ ls msg/
5 ControlMode.msg DigitalInput.msg DigitalOutput.msg JointPositionControl.msg
6 $ ls src/
7 getID.sh joint_position_to_joint_trajectory.cpp joint_tf_publisher.cpp ypspur_ros.cpp

```

CMakeLists.txtとpackage.xmlには、使っているライブラリの一覧や、生成する実行ファイルとC++のソースコードの対応など、このパッケージをビルドするために必要な情報が書かれています。msgディレクトリには、このパッケージ独自のデータ形式の定義が、srcディレクトリには、このパッケージに含まれるプログラム(ノード)のソースコードが含まれています。

次にcatkin_makeコマンドで、ダウンロードしたypspur_rosパッケージを含む、ワークスペース全体をビルドします。catkin_makeは、ワークスペースの最上位ディレクトリ(~/.catkin_ws/)で行います。

```

1 $ cd ~/.catkin_ws/
2 $ catkin_make
... 実行結果 ...

```

ROSノードの理解とビルド・実行

端末を開き、ひな形をダウンロードします。

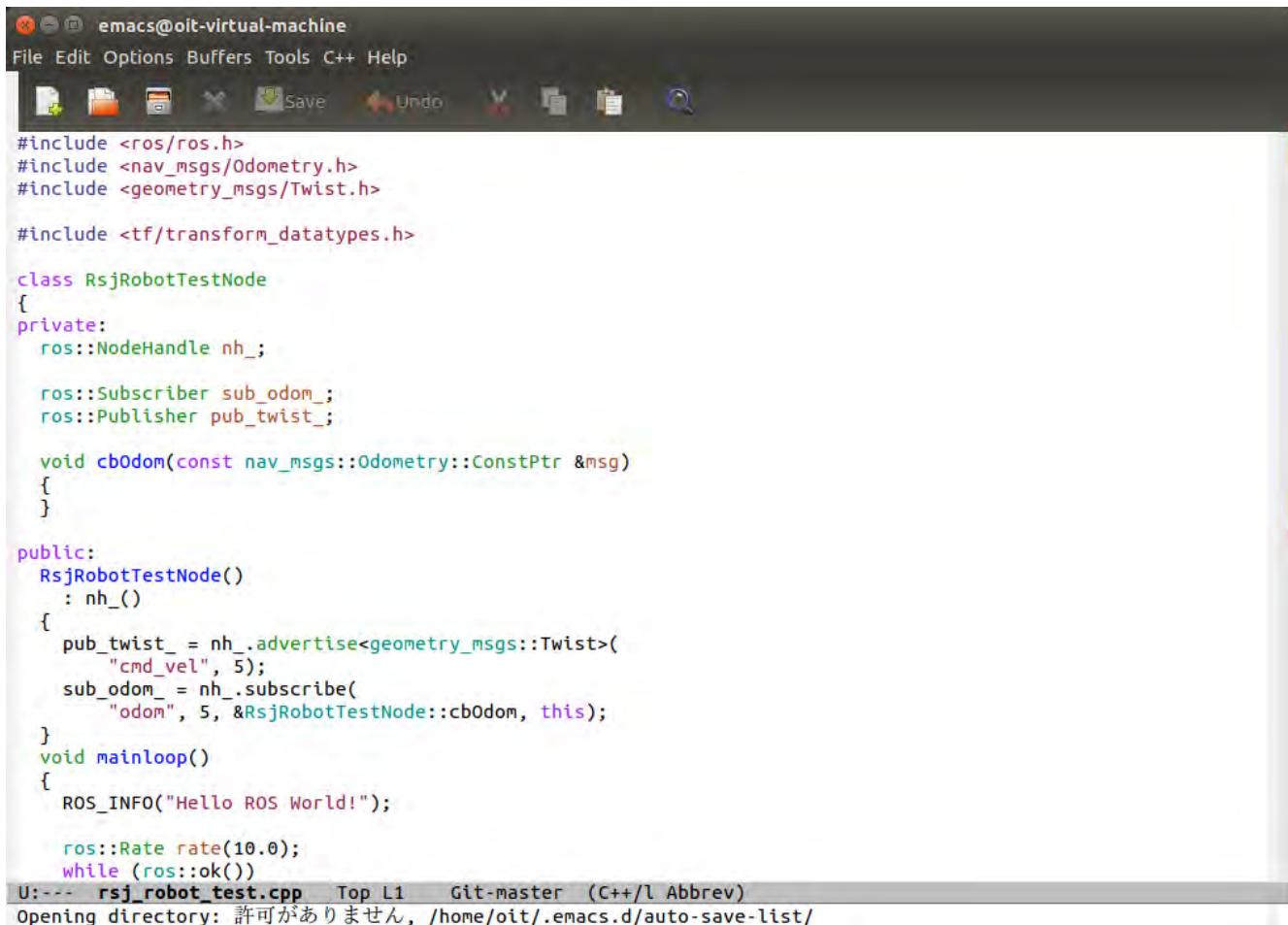
```

1 $ cd ~/.catkin_ws/src/
2 $ git clone https://github.com/BND-tc/rsj_robot_test.git

```

ソースファイルの編集にはお好みのテキストエディターが利用可能です。本セミナーの説明ではメジャーなテキストエディタであるemacsの画面で例を示します。Linuxがはじめての方にはgeditもおすすめです。

お好みのテキストエディターで~/.catkin_ws/src/rsj_robot_test/src/rsj_robot_test.cppを開きます。



```

emacs@oit-virtual-machine
File Edit Options Buffers Tools C++ Help
Save Undo

#include <ros/ros.h>
#include <nav_msgs/Odometry.h>
#include <geometry_msgs/Twist.h>

#include <tf/transform_datatypes.h>

class RsjRobotTestNode
{
private:
    ros::NodeHandle nh_;

    ros::Subscriber sub_odom_;
    ros::Publisher pub_twist_;

    void cbOdom(const nav_msgs::Odometry::ConstPtr &msg)
    {

public:
    RsjRobotTestNode()
        : nh_()
    {
        pub_twist_ = nh_.advertise<geometry_msgs::Twist>(
            "cmd_vel", 5);
        sub_odom_ = nh_.subscribe(
            "odom", 5, &RsjRobotTestNode::cbOdom, this);
    }
    void mainloop()
    {
        ROS_INFO("Hello ROS World!");

        ros::Rate rate(10.0);
        while (ros::ok())
    }
U:--- rsj_robot_test.cpp Top L1 Git-master (C++/l Abbrev)
Opening directory: 許可がありません, /home/oit/.emacs.d/auto-save-list/

```

基本的なコードを読み解く

このコードが実行されたときの流れを確認しましょう。

まず、先頭部分では必要なヘッダファイルをインクルードしています。

```
1 #include <ros/ros.h>
```

続いて、`RsjRobotTestNode`クラスを定義しています。ROS プログラミングの際には、基本的にノードの持つ機能をクラスとして定義し、これを呼び出す形式を取ることが標準的です。クラスを使用せずに書くことも可能ですが、気をつけなければならない点が多くなるため、本セミナーではクラスでの書き方のみを解説します。

```
1 class RsjRobotTestNode
2 {
3     // (略)
4     public:
5         // (略)
6         void mainloop()
7         {
8             ROS_INFO("Hello ROS World!");
9
10            ros::Rate rate(10.0);
11            while(ros::ok())
12            {
13                ros::spinOnce();
14                // ここに速度指令の出力コード
15                rate.sleep();
16            }
17            // ここに終了処理のコード
18        }
19    };
```

`RsjRobotTestNode`クラスのメンバ関数である`mainloop`関数の中では、ROS で情報を画面などに出力する際に用いる`ROS_INFO`関数を呼び出して、"Hello ROS World!"と表示しています。他にも、`ROS_DEBUG`、`ROS_WARN`、`ROS_ERROR`、`ROS_FATAL`関数が用意されています。

`ros::Rate rate(10.0)`で、周期実行のためのクラスを初期化しています。初期化時の引数で実行周波数(この例では10Hz)を指定します。

`while(ros::ok())`で、メインの無限ループを回します。`ros::ok()`を`while`の条件にすることで、ノードの終了指示が与えられたとき(`Ctrl+c`が押された場合も含む)には、ループを抜けて終了処理などが行えるようになっています。

ループ中では、まず`ros::spinOnce()`を呼び出して、ROS のメッセージを受け取るといった処理を行います。`ros::spinOnce()`はその時点で届いているメッセージの受け取り処理を済ませた後、すぐに処理を返します。`rate.sleep()`は、先ほど初期化した実行周波数を維持するように`sleep`します。

なお、ここではクラスを定義しただけなので、中身が呼び出されることはできません。後ほど実体化されたときに初めて中身が実行されます。

続いて、C++ の`main`関数が定義されています。ノードの実行時にはここから処理がスタートします。

```
1 int main(int argc, char **argv) {
2     ros::init(argc, argv, "rsj_robot_test_node");
3
4     RsjRobotTestNode robot_test;
5
6     robot_test.mainloop();
7 }
```

はじめに`ros::init`関数を呼び出して、ROS ノードの初期化を行います。1、2番目の引数には`main`関数の引数をそのまま渡し、3番目の引数にはこのノードの名前(この例では"rsj_robot_test_node")を与えます。

次に`RsjRobotTestNode`クラスの実体を作成します。ここでは`robot_test`と名前をつけています。

最後に実体化した`robot_test`のメンバ関数、`mainloop`を呼び出します。`mainloop`関数の中は無限ループになっているため、終了するまでの間`ros::spinOnce()`、`rate.sleep()`が呼び出され続けます。

つまり、`rsj_robot_test`は特に仕事をせず"Hello ROS World!"と画面に表示します。

ビルド&実行

ROS 上でこのパッケージをビルドするためには、`catkin_make`コマンドを用います。

```
1 $ cd ~/catkin_ws/
2 $ catkin_make
```

端末で実行してみましょう。

ROS システムの実行の際、ROS を通してノード同士がデータをやりとりするために用いる、`roscore`を起動しておく必要があります。2つの端末を開き、それぞれで以下を実行して下さい。

1つ目の端末：

```
1 $ roscore
```

ROSでワークスペースを利用するとき、端末でそのワークスペースをアクティベートすることが必要です。そのためにワークスペースの最上のディレクトリでsource devel/setup.bashを実行します。このコマンドはワークスペースの情報を利用中の端末に読み込みます。しかし、これは一時的にしか効果がないので新しい端末でワークスペースを利用し始めるときは必ずまずはsource devel/setup.bashを実行しなければなりません。一つの端末で一回だけ実行すれば十分です。その端末を閉じるまで有効です。

2つ目の端末で下記を実行します。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun rsj_robot_test rsj_robot_test_node
[ INFO] [1466002781.136800000]: Hello ROS World!
```

Hello ROS World!と表示されれば成功です。

以上の手順で、ROSパッケージに含まれるノードのソースコードを編集し、ビルドして実行できるようになりました。

両方の端末でCtrl+cでノードとroscoreを終了します。

ロボットに速度指令を与える

まず、ロボットに速度指令(目標並進速度・角速度)を与えるコードを追加します。ひな形には、既に速度指令値が入ったメッセージを出力するための初期化コードが含まれていますので、この部分の意味を確認します。

```
1 RsjRobotTestNode()
2 {
3     nh_();
4     pub_twist_ = nh_.advertise<geometry_msgs::Twist>(
5         "cmd_vel", 5);
6     sub_odom_ = nh_.subscribe(
7         "odom", 5, &RsjRobotTestNode::cbOdom, this);
8 }
```

ソースコード中の、RsjRobotTestNodeクラスのRsjRobotTestNode関数は、クラスのコンストラクタと呼ばれるもので、クラスが初期化されるときに自動的に呼び出されます。2行目の：nh_()の部分では、クラスのメンバ変数であるnh_を、引数なしで初期化しています。このコンストラクタの中で、

```
1 nh_.advertise<geometry_msgs::Twist>("cmd_vel", 5);
```

の部分で、このノードが、これからメッセージを出力することを宣言しています。advertise関数に与えている引数は以下のようない意味を持ちます。

"cmd_vel"
 出力するメッセージを置く場所(トピックと呼ぶ)を指定
5
 メッセージのバッファリング量を指定(大きくすると、処理が一時的に重くなったときなどに受け取り側の読み飛ばしを減らせる)

先頭のnh_はros::NodeHandle型のメンバ変数で、トピックやパラメータといったROSノードの基本的な機能を初期化するために使用します。nh_()のように引数無しで初期化することでグローバル名前空間(/)を使う設定になるので、/という名前空間の下のcmd_velという名前のトピック、つまり/cmd_velというトピックにメッセージを出力することを意味します。

advertise関数についている<geometry_msgs::Twist>の部分は、メッセージの型を指定しています。これは、幾何的・運動学的な値を扱うメッセージを定義しているgeometry_msgs/パッケージの、並進・回転速度を表すTwist型です。この指定方法は、C++のテンプレートという機能を利用してますが、ここでは「advertiseのときはメッセージの型指定を<>の中に書く」とだけ覚えておけば問題ありません。

mainloop関数中の「ここに速度指令の出力コード」の部分を以下のように編集することで、速度指令のメッセージを出力(publish)します。

```
1 void mainloop()
2 {
3     ROS_INFO("Hello ROS World!");
4     ros::Rate rate(10.0);
5     while(ros::ok())
6     {
7         ros::spinOnce();
8         // ここに速度指令の出力コード
9         geometry_msgs::Twist cmd_vel;
10        cmd_vel.linear.x = 0.05;
11        cmd_vel.angular.z = 0.0;
```

```

13     pub_twist_.publish(cmd_vel);
14
15     rate.sleep();
16 }
17 // ここに終了処理のコード
18 }
```

ビルド&実行

```

1 $ cd ~/catkin_ws/
2 $ catkin_make
```

この際、ビルドエラーが出ていないか良く確認して下さい。エラーが出ている場合は、ソースコードの該当箇所を確認・修正して下さい。

実行の際、まずroscoreとypspur_rosを起動します。ypspur_rosの中では、ロボットの動作テストの際に使用したypspur-coordinatorが動いています。なお、roscoreは前のものを実行し続けている場合はそのまま使用できます。コマンド入力の際はタブ補完を活用しましょう。

```
1 $ roscore
```

2番目の端末を開いて、下記を実行します。

```

1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun ypspur_ros ypspur_ros _param_file:=/home/【ユーザ名】/params/rsj-seminar20???.param \
4   _port:=/dev/serial/by-id/usb-T-frog_project_T-frog_Driver-if00 \
5   _compatible:=1
```

続いて、別の端末でrsj_robot_test_nodeノードを実行します。まずは、ロボットのホイールを浮かせて走り出さない状態にして実行してみましょう。

```

1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun rsj_robot_test rsj_robot_test_node
4 Hello ROS World!
```

ゆっくりとホイールが回れば正しく動作しています。それぞれの端末で **Ctrl+c** を押し、終了します。

小課題(1)

速度、角速度を変更して動作を確認してみましょう。

ロボットの状態を表示する

ロボットの状態を表示するコードを追加

まず、ロボットの動作したときの移動量やオドメトリ座標を取得、表示するコードを追加します。ひな形には、既に移動量や座標が入ったメッセージを受け取るコードが含まれていますので、この部分の意味を確認します。

```

1 RsjRobotTestNode()
2   : nh_()
3 {
4   pub_twist_ = nh_.advertise<geometry_msgs::Twist>(
5     "cmd_vel", 5);
6   sub_odom_ = nh_.subscribe(
7     "odom", 5, &RsjRobotTestNode::cbodom, this);
8 }
```

この中で

```
1 sub_odom_ = nh_.subscribe("odom", 5, &RsjRobotTestNode::cbodom, this);
```

の部分で、このノードがこれからメッセージを受け取ることを宣言しています。subscribe関数に与えている引数は以下ののような意味を持ちます。

```

"odom"
    受け取るメッセージが置かれている場所(トピック)を指定
5
    メッセージのバッファリング量を指定(大きくすると、処理が一時に重くなったときなどに受け取り側の読み飛ばしを減らせる)
&RsjRobotTestNode::cbOdom
    メッセージを受け取ったときに呼び出す関数を指定(RsjRobotTestNodeクラスの中にある、cbOdom関数)
this
    メッセージを受け取ったときに呼び出す関数がクラスの中にある場合にクラスの実体を指定(とりあえず、おまじないと思って構いません。)

```

こちらもグローバル名前空間(/)を使う設定で初期化されているnh_を使っているので、/下のodom、つまり/odomという名前のトピックからメッセージを取得することを意味します。これにより、rsj_robot_test_nodeノードが/odomトピックからメッセージをうけとると、cbOdom関数が呼び出されるようになります。

続いてcbOdom関数の中身を確認しましょう。

```

1 void cbOdom(const nav_msgs::Odometry::ConstPtr &msg)
2 {
3 }

```

const nav_msgs::Odometry::ConstPtrは、constな(内容を書き換えられない)nav_msgsパッケージに含まれるOdometry型のメッセージの、const型ポインタを表しています。&msgの&は、参照型(内容を書き換えられるように変数を渡すことができる)という意味ですが、(const型なので)ここでは特に気にする必要はありません。

cbOdom関数に、以下のコードを追加してみましょう。これにより、受け取ったメッセージの中から、ロボットの並進速度を取り出して表示できます。

```

1 void cbOdom(const nav_msgs::Odometry::ConstPtr &msg)
2 {
3     ROS_INFO("vel %f", msg->twist.twist.linear.x);
4 }

```

ここで、msg->twist.twist.linear.xの意味を確認します。nav_msgs::Odometryメッセージには、下記のように入れ子状にメッセージが入っています。

- std_msgs/Header header
- string child_frame_id
- geometry_msgs/PoseWithCovariance pose
- geometry_msgs/TwistWithCovariance twist

全て展開すると、以下の構成になります。

- std_msgs/Header header
 - uint32 seq
 - time stamp
 - string frame_id
- string child_frame_id
- geometry_msgs/PoseWithCovariance pose
 - geometry_msgs/Pose pose
 - geometry_msgs/Point position
 - float64 x
 - float64 y
 - float64 z
 - geometry_msgs/Quaternion orientation
 - float64 x
 - float64 y
 - float64 z
 - float64 w
 - float64[36] covariance
- geometry_msgs/TwistWithCovariance twist
 - geometry_msgs/Twist twist
 - geometry_msgs/Vector3 linear
 - float64 x ロボット並進速度
 - float64 y
 - float64 z
 - geometry_msgs/Vector3 angular
 - float64 x
 - float64 y
 - float64 z ロボット角速度
 - float64[36] covariance

読みたいデータであるロボット並進速度を取り出すためには、これを順にたどっていけば良く、msg->twist.twist.linear.xとなります。msgはクラスへのポインタなので「->」を用い、以降はクラスのメンバ変数へのアクセスなので「.」を用いてアクセスしています。

ビルド&実行

```
1 $ cd ~/catkin_ws/
2 $ catkin_make
```

この際、ビルドエラーが出ていないか、良く確認して下さい。エラーが出ている場合は、ソースコードの該当箇所を確認・修正して下さい。

まず、先ほどと同様にroscoreと、ypspur_rosを起動します(以降、この手順の記載は省略します)。

```
1 $ roscore
```

2番目の端末を開いて、下記を実行します。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ roslaunch ypspur_ros ypspur_ros _param_file:=~/home/【ユーザ名】/params/rsj-seminar20???.param \
4   _port:=/dev/serial/by-id/usb-T-frog_project_T-frog_Driver-if00 \
5   _compatible:=1
```

続いて、rsj_robot_test_nodeノードを実行します。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ roslaunch rsj_robot_test rsj_robot_test_node
Hello ROS World!
vel: 0.0500
vel: 0.0500
vel: 0.0500
vel: 0.0500
```

ロボットのホイールが回転し、先ほどの小課題で設定した走行指令の値と近い値が表示されれば正しく動作しています。

小課題(2)

同様に、ロボットの角速度を表示してみましょう。

シーケンス制御

時間で動作を変える

メインループを以下のように変更してみましょう。

```
1 void mainloop()
2 {
3     ROS_INFO("Hello ROS World!");
4
5     ros::Rate rate(10.0);
6     ros::Time start = ros::Time::now();
7     while(ros::ok())
8     {
9         ros::spinOnce();
10        ros::Time now = ros::Time::now();
11
12        geometry_msgs::Twist cmd_vel;
13        if(now - start > ros::Duration(3.0))
14        {
15            cmd_vel.linear.x = 0.05;
16            cmd_vel.angular.z = 0.0;
17        }
18        pub_twist_.publish(cmd_vel);
19
20        rate.sleep();
21    }
22 }
```

これは、メインループ開始時刻から3.0秒後に、並進速度0.05m/sの指令を与えるコードです。ros::Time型(時刻を表す)同士の減算結果はros::Duration型(時間を表す)になります。比較演算子で比較できます。したがって、now - start > ros::Duration(3.0)の部分は、開始から3秒後にtrueになります。

先ほどと同様にビルドし、ypspur_rosとrsj_robot_test_nodeを起動して動作を確認します。

センシング結果で動作を変える

cbodomで取得したオドメトリのデータを保存しておくように、以下のように変更してみましょう。

```

1 void cbodom(const nav_msgs::Odometry::ConstPtr &msg)
2 {
3     ROS_INFO("vel %f", msg->twist.twist.linear.x);
4     odom_ = *msg; // 追記
5 }
```

また、class RsjRobotTestNodeの先頭に下記の変数定義を追加します。

```

1 class RsjRobotTestNode
2 {
3     private:
4         nav_msgs::Odometry odom_;
```

また、odom_の中で方位を表すクオータニオンをコンストラクタ(RsjRobotTestNode()関数)の最後で初期化しておきます。

```

1 RsjRobotTestNode():
2 {
3     (略)
4     odom_.pose.pose.orientation.w = 1.0;
5 }
```

mainloop()を以下のように変更してみましょう。

```

1 void mainloop()
2 {
3     ROS_INFO("Hello ROS World!");
4
5     ros::Rate rate(10.0);
6     while(ros::ok())
7     {
8         ros::spinOnce();
9
10        geometry_msgs::Twist cmd_vel;
11        if(tf::getYaw(odom_.pose.pose.orientation) > 1.57)
12        {
13            cmd_vel.linear.x = 0.0;
14            cmd_vel.angular.z = 0.0;
15        }
16        else
17        {
18            cmd_vel.linear.x = 0.0;
19            cmd_vel.angular.z = 0.1;
20        }
21        pub_twist_.publish(cmd_vel);
22
23        rate.sleep();
24    }
25 }
```

これは、オドメトリのYaw角度(旋回角度)が1.57ラジアン(90度)を超えるまで、正方向に旋回する動作を表しています。

先ほどと同様にビルドし、yppspur_rosとrsj_robot_test_nodeを起動して動作を確認します。

小課題(3)

1m 前方に走行し、その後で帰ってくるコードを作成してみましょう。(1m 前方に走行し 180 度旋回して 1m 前方に走行するか、1m 前方に走行し 1m 後方に走行すればよい。)

余裕があれば、四角形を描いて走行するコードを作成してみましょう。

ロボットの組立てと動作テスト

教材のロボットハードウェアを組み立て、簡単な動作テストを行います。

モータドライバのバスパワー化

本セミナーの教材として使用するモータドライバは、初期状態ではセルフパワー(USBケーブルの他に、別途電源を供給する必要のある)USBデバイスとして動作します。本セミナーでは、簡易的な使用のため、モータドライバに配線を追加してバスパワーUSBデバイスとして使用します。

1. 被覆電線を約80mmの長さで切断
2. ワイヤストリッパを用い、電線の両端5mmの被覆を剥ぐ
3. モータドライバ上の「+5Vin」と「USB5V」の穴に電線を差し込み、半田付け



4. 半田付けがうまく行えているか確認(オーガナイザーまたはTAに声をかけてください)

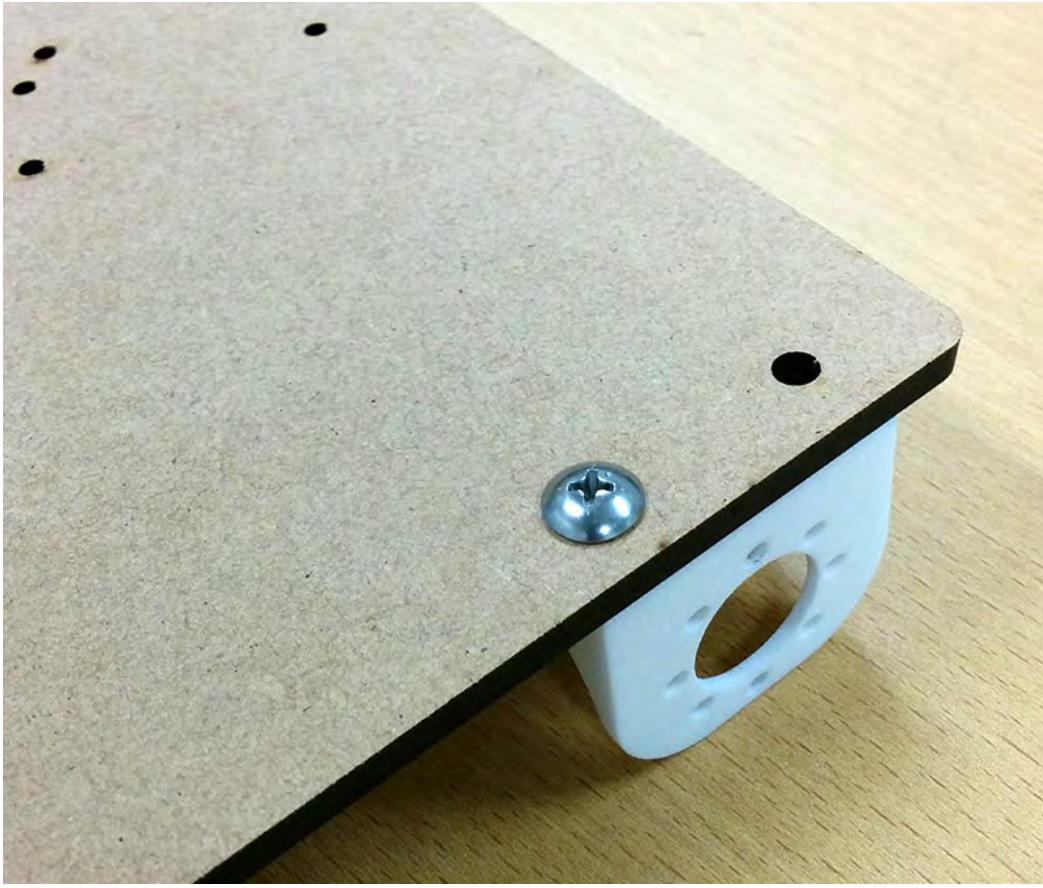
注意：「CON1」から5Vを供給して利用する場合には、ここで半田付けした電線を切除してください。そのままで「CON1」から5Vを供給すると、モータドライバおよびUSBで接続しているPCが破損する可能性があります。

ロボットの組立て

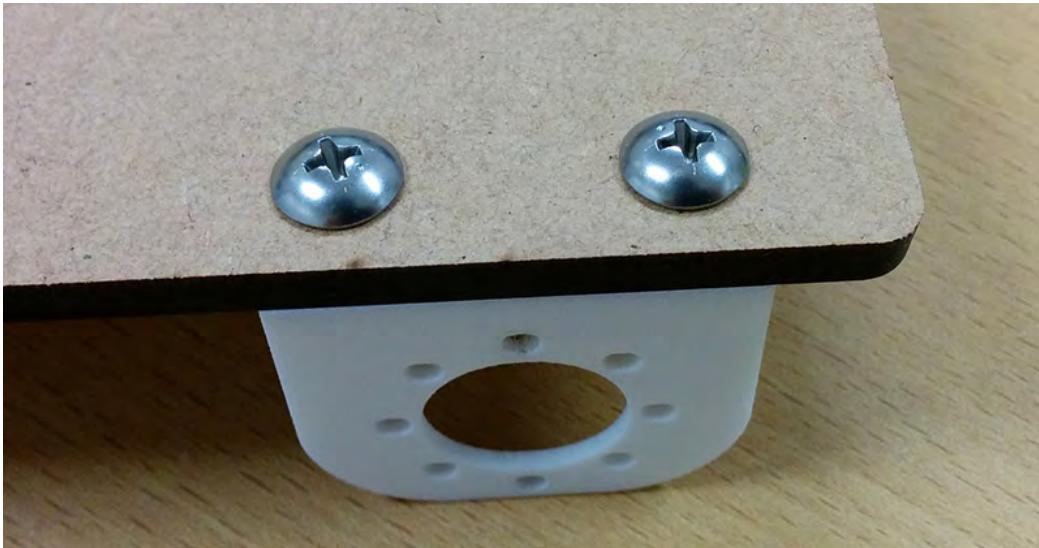
1. 天板にモータマウントをネジ止め
表からM5のネジを軽く留める



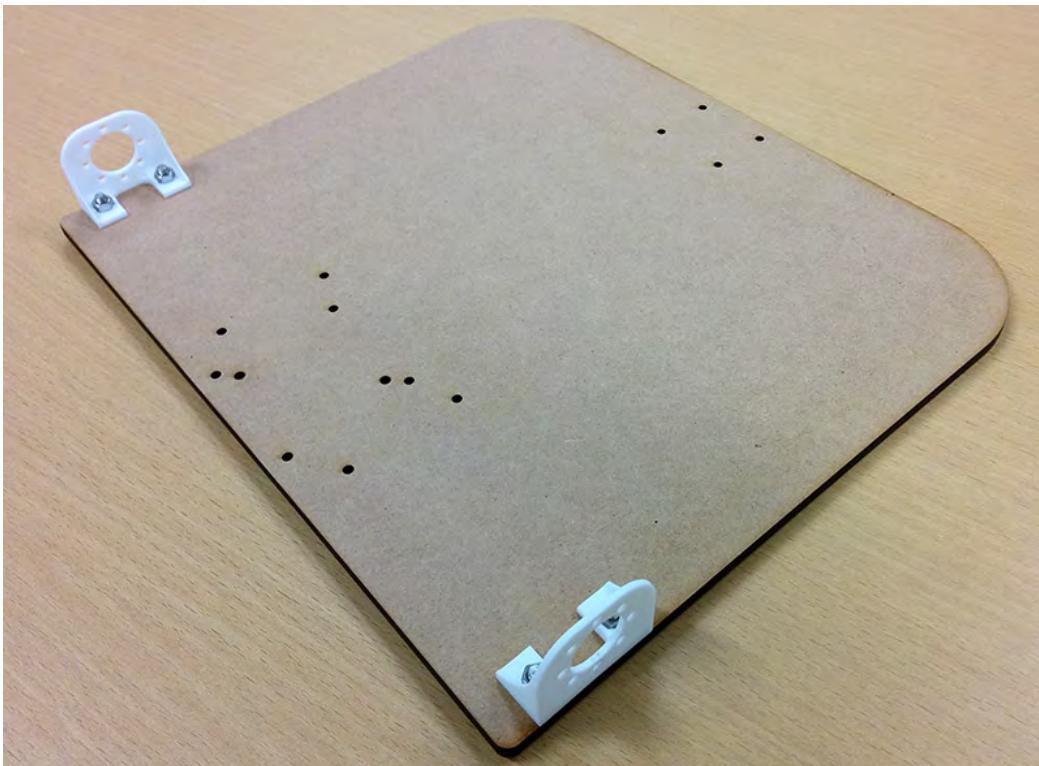
同様にもう一本もネジ留め



反対側のモータマウントも同様に取り付け



モータマウント裏側の溝に、M5のナットをはめ込み



2. モータマウントにモータをネジ止め

モータをモータマウントにはめ込み



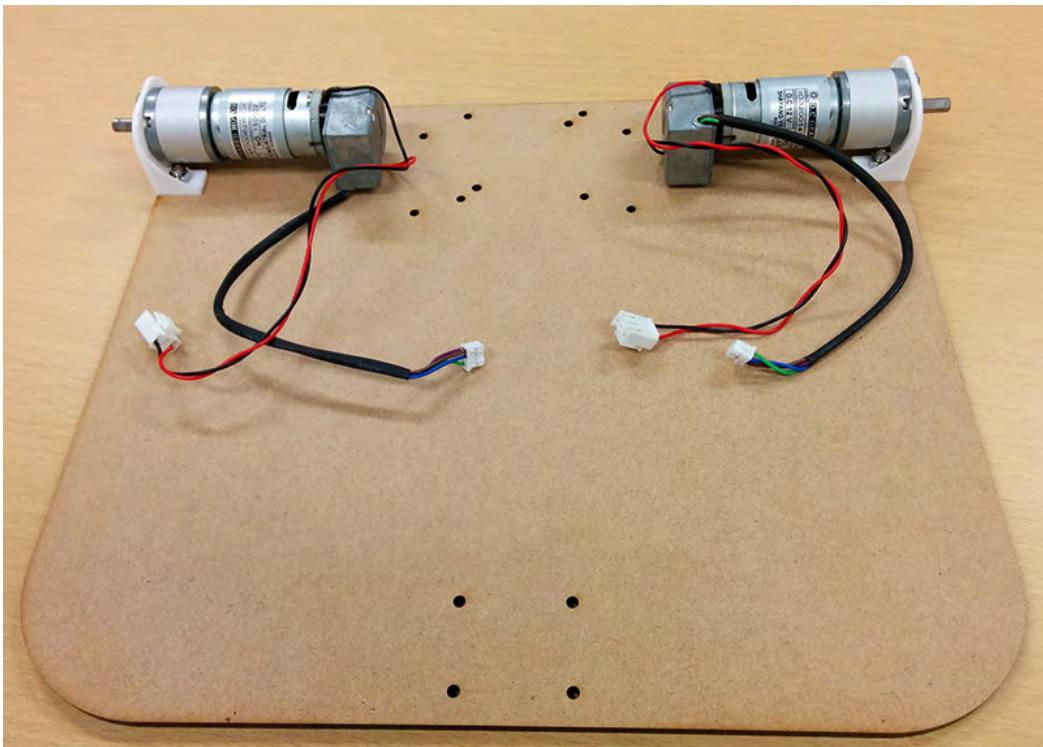
2018/11/15

移動ロボットの制御とROSによる動作計画実習 by BND-tc

モータマウント側面の穴から、M3のネジ（写真ではプラスネジですが、キットでは六角穴付きネジになります）4本でモータを固定



反対側のモータも同様に取り付け

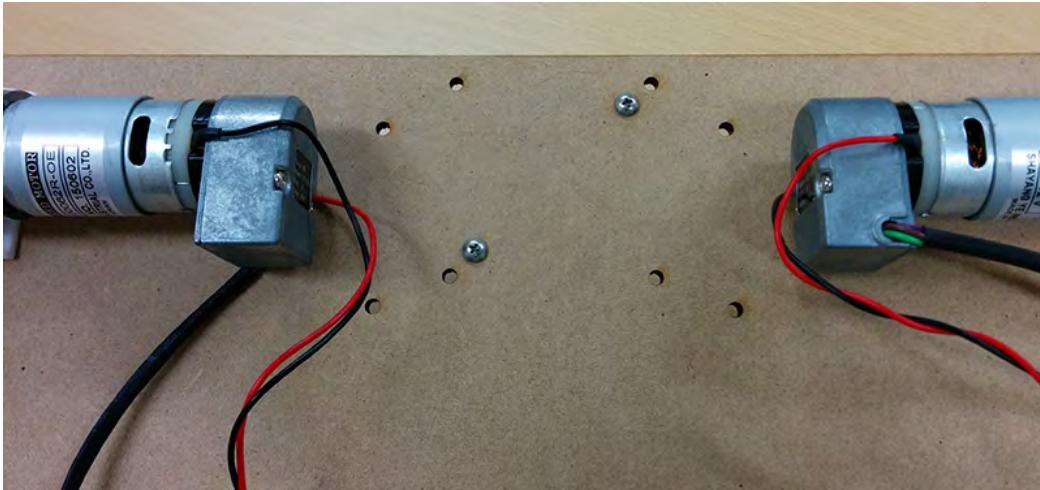


3. 天板に測域センサをネジ止め

天板の表側に、写真の向きにURGを載せる

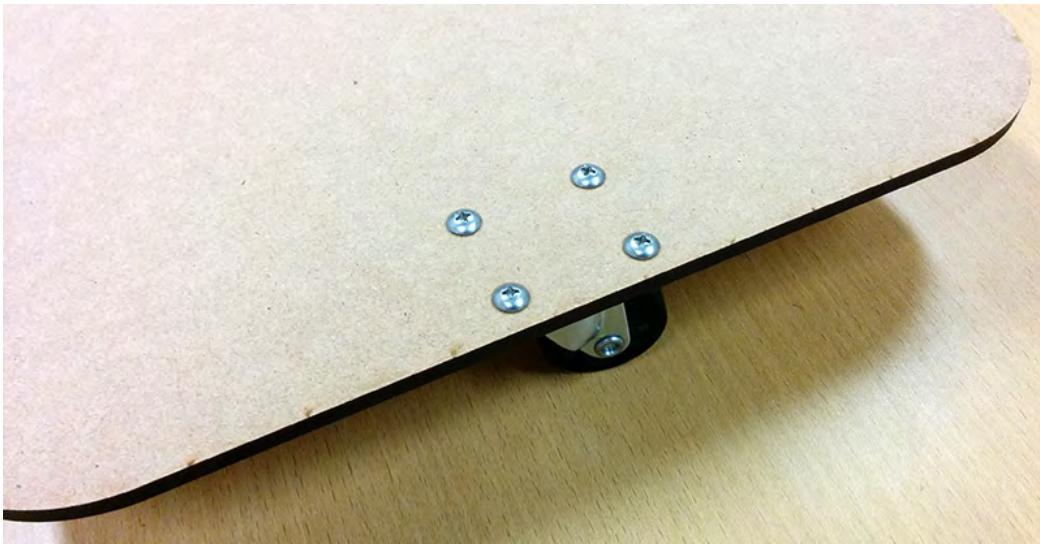


裏面からM3のネジ2本で固定



4. 天板にキャスターをネジ止め

天板の表側からM4のネジを挿入し



裏面からM4のナットで固定



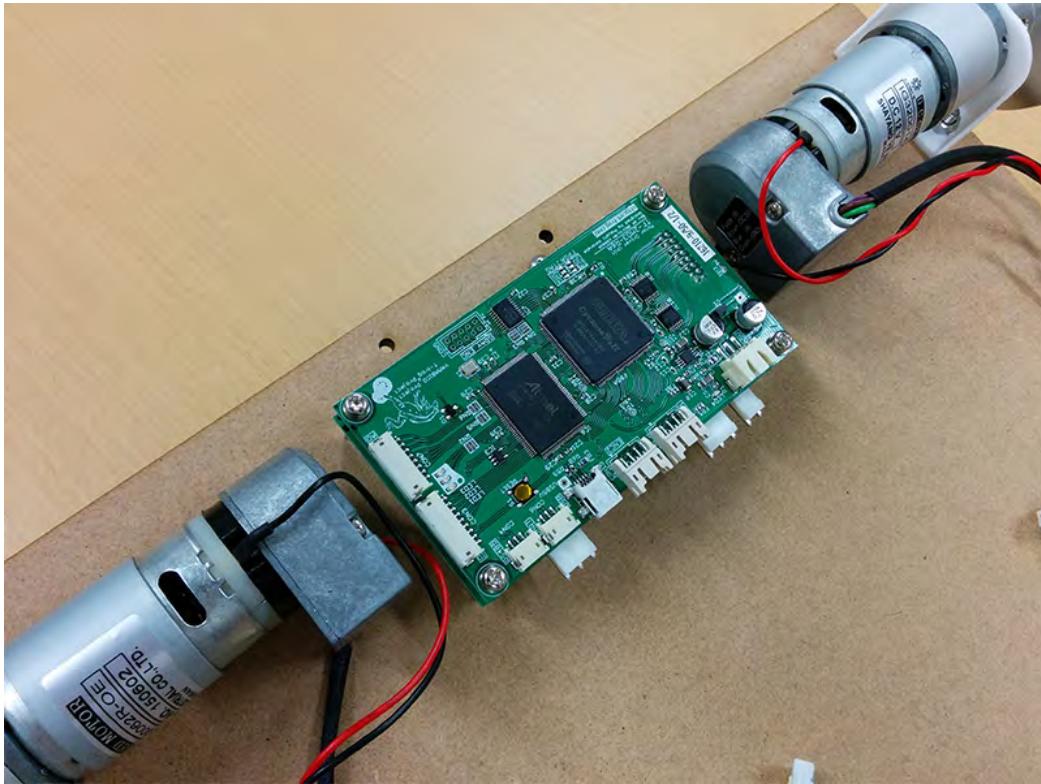
5. モータにホイールをイモネジ止め

モータ軸のDカット(平らな面)に、ホイールのイモネジが入っているねじ穴を合わせ、六角レンチでイモネジを締めて固定



6. 天板にモータドライバをネジ止め

天板の裏面に、写真の向きでモータドライバを載せ

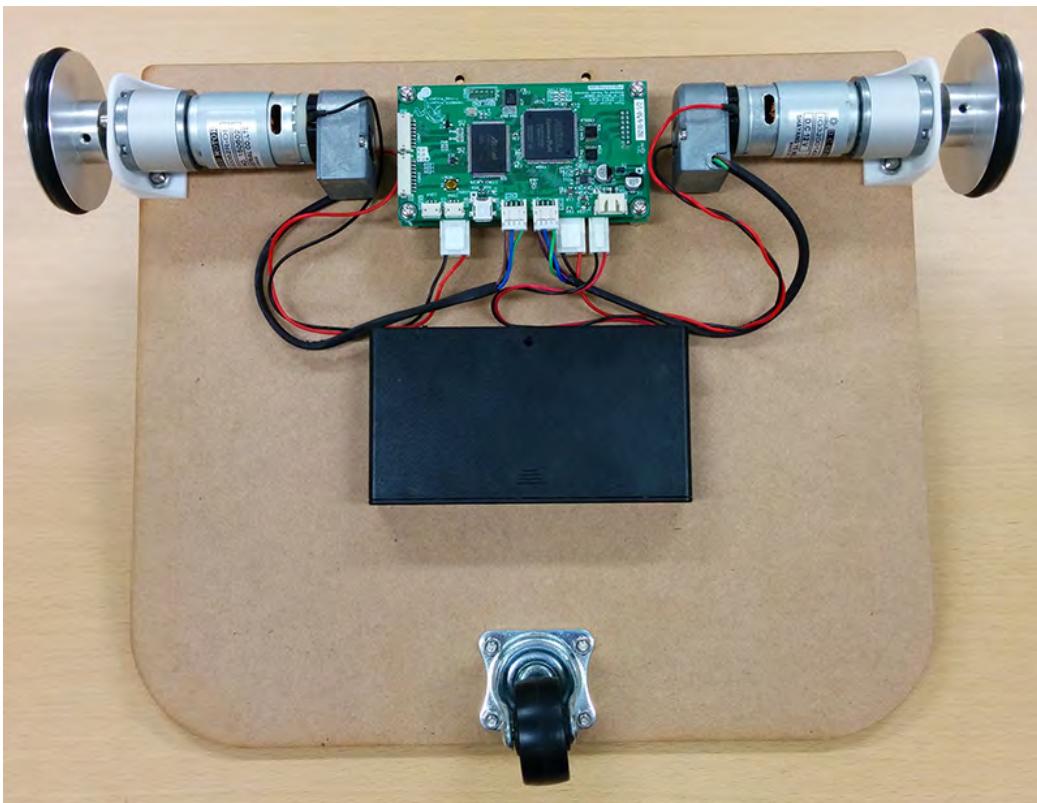


表面からM3のネジで固定



7. 天板に電池ボックスを貼り付け

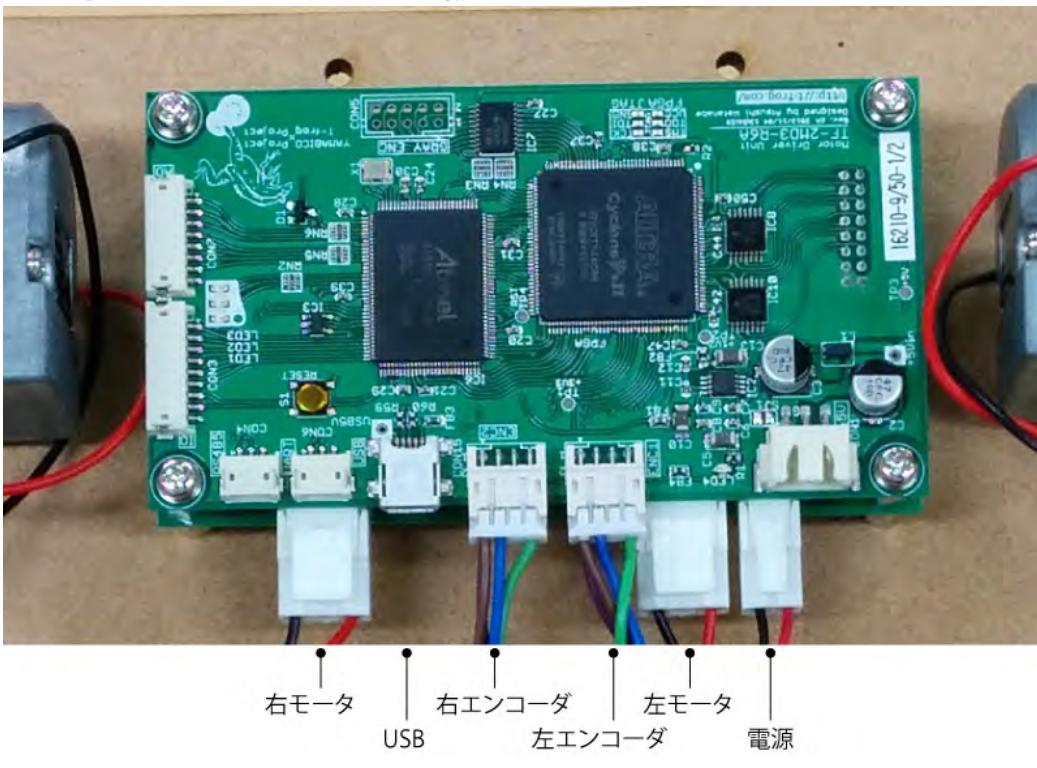
天板の裏面に、写真の向きで、両面テープを用いて電池ボックスを固定



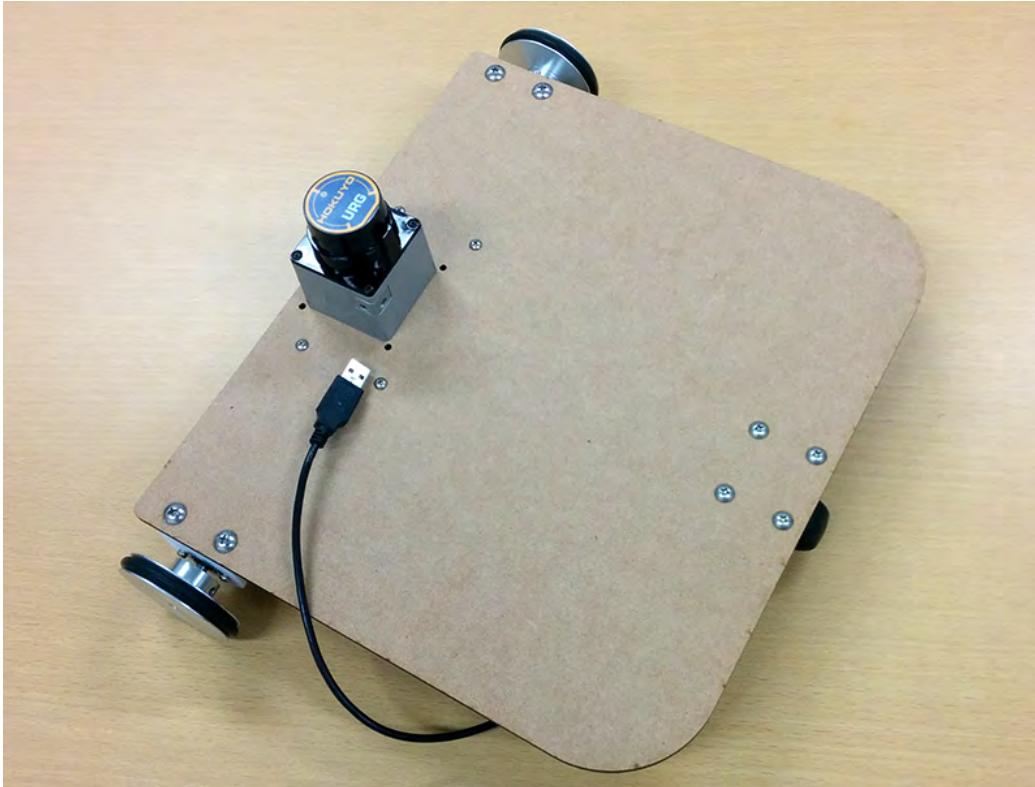
貼り合わせ後、よく押しつけて、接着面同士を十分に密着させます。接着力を発揮させるため、貼り付け後5分間は、電池ボックスが上側にある状態を維持して下さい。

8. 配線

モータ・電池ボックスのケーブルをモータドライバに接続



以上でロボットの組み立ては完了です。



昨年以前のロボットのメンテナンス

2014年、2015年、2016年のロボットを使用する場合、ネジ類が緩んでいないか確認しましょう。特に、ホイールをモータに固定しているイモネジの緩みに注意して下さい。

初期設定

1. ノートPCでUbuntuを起動
2. USBデバイスを開けるように、使用しているユーザをdialoutグループに追加

```
1 $ sudo adduser [ YOUR_USER_NAME ] dialout
```

3. 上記設定を反映するため、画面右上のアイコンから、ログアウトを選択してログアウト
4. ログイン画面から再度ログイン

制御ソフトのインストールと動作テスト

1. 画面左のランチャーから「端末」を起動し、下記コマンドを実行して制御ソフトをインストール（日本語環境の場合はdownloadsをダウンロードに読み替えてください）

```
1 $ cd ~/Downloads/
2 $ git clone https://github.com/openspur/yp-spur.git
3 $ cd yp-spur
4 $ mkdir build
5 $ cd build
6 $ cmake ..
7 $ make
8 $ sudo make install
9 $ sudo ldconfig
```

ロボットパラメータファイルをダウンロードします。（[参考資料: ロボットの見分け方](#)）

- 2015年度・2016年度・2018年度のロボットの場合

```
1 $ mkdir ~ /params
2 $ cd ~ /params /
3 $ wget https://at-wat.github.io/ROS-quick-start-up/files/rsj-seminar2016.param
```

- 2014年度のロボットの場合

```

1 $ mkdir ~/params
2 $ cd ~/params/
3 $ wget https://at-wat.github.io/ROS-quick-start-up/files/rsj-seminar2014.param

```

2. ロボットの電池ボックスに電池を挿入
3. 電池ボックスの側面にあるスライドスイッチをON
4. モータドライバのUSBをPCに接続
5. 端末を開いて、下記コマンドで制御ソフトを起動

```

1 $ ypspur-coordinator -p ~/params/rsj-seminar20?.param 【該当するものに置き換えること】 \
2   -d /dev/serial/by-id/usb-T-frog_project_T-frog_Driver-if00

```

6. ホイールを持ち上げて走り出さない状態、もしくは、ひっくり返してホイールが浮いている状態にしてサンプルプログラムを起動(端末をもう1つ開いて実行します。)

```

1 $ cd ~/Downloads/yp-spur/build
2 $ ./samples/run-test

```

ホイールが回転して、回転方向を何度か変え、最終的に止まることを確認したら、**Ctrl+c**で停止させます。このとき、ypspur-coordinatorは、まだ停止させないで下さい。(停止させた場合はもう一度起動)

7. 床の広い場所でサンプルプログラムを起動

```

1 $ ./samples/run-test

```

1m x 0.1m の四角形を描いてロボットが移動することを確認します。最後に、サンプルプログラムと、ypspur-coordinatorを**Ctrl+c**で停止させます。

8. 電池ボックスのスライドスイッチをOFF(待機電力で電池の電力を消費してしまいます)

補足

今回のセミナではYP-Spurをソースコードからインストールしましたが、下記コマンドでバイナリーのみをインストールすることも可能です。

```

1 $ sudo apt-get install ros-kinetic-ypspur

```

ROSの基本操作

基本的なROS上で動くプログラムの書き方とビルド方法を学習します。

基本的な用語

パッケージ

ノードや設定ファイル、コンパイル方法などをまとめたもの

ノード

ROSの枠組みを使用する、実行ファイル

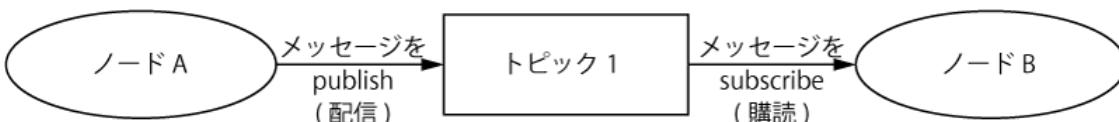
メッセージ

ノード間でやりとりするデータ

トピック

ノード間でメッセージをやりとりする際に、メッセージを置く場所

ノード、メッセージ、トピックの関係は以下の図のように表せます。



基本的には、ソフトウェアとしての ROS は、ノード間のデータのやりとりをサポートするための枠組みです。加えて、使い回しがきく汎用的なノードを世界中の ROS 利用者で共有するコミュニティも、大きな意味でのROSの一部となっています。

ソースコードを置く場所

ROS ではプログラムをビルドする際に、catkinというシステムを使用しています。また、catkinはcmakeというシステムを使っており、ROS 用のプログラムのパッケージ毎にcmakeの設定ファイルを作成することで、ビルドに必要な設定を行います。

以下の手順で本作業用の新しいワークスペースを作ります。

```
1 $ mkdir -p ~/catkin_ws/src
2 $ cd ~/catkin_ws/src
3 $ catkin_init_workspace
4 Creating symlink "/home/[ユーザディレクトリ]/catkin/src/CMakeLists.txt"
5   pointing to "/opt/ros/kinetic/share/catkin/cmake/toplevel.cmake"
6 $ ls
7 CMakeLists.txt
8 $ cd ..
9 $ ls
10 src
11 $ catkin_make
12 Base path: /home/[ユーザ名]/catkin_tmp
13 Source space: /home/[ユーザ名]/catkin_tmp/src
14 Build space: /home/[ユーザ名]/catkin_tmp/build
15 Devel space: /home/[ユーザ名]/catkin_tmp/devel
16 Install space: /home/[ユーザ名]/catkin_tmp/install
17 ...
18 $ ls
19 build devel src
```

catkin_wsディレクトリ内にある、build、develは、catkinシステムがプログラムをビルドする際に使用するものなので、ユーザが触る必要はありません。catkin_ws/srcディレクトリはROS パッケージのソースコードを置く場所で、中にあるCMakeLists.txtはワークスペース全体をビルドするためのルールが書かれているファイルです。

このディレクトリにypspur-coordinatorをROSに接続するためのパッケージypspur_rosをダウンロードします。（今回は説明のためypspur_rosのソースコードを入手しましたが、aptを利用してバイナリーのみのインストールも可能です。）

```
1 $ cd ~/catkin_ws/src
2 $ git clone https://github.com/openspur/ypspur_ros.git
3 $ ls
4 CMakeLists.txt ypspur_ros
```

git はソースコードなどの変更履歴を記録して管理する、分散型バージョン管理システムと呼ばれるものです。今回のセミナーでは詳細は触れませんが、研究開発を行う上では非常に有用なシステムですので利用をお勧めします。公式の解説書、[Pro Git](#)などを参考にして下さい。

GitHub はソースコードなどのリポジトリーサービスです。オープンソースソフトウェアの開発、共同作業及び配布を行うためによく利用されていて、ROS ではソースコードの保存と配布する場所としても人気なサービスです。バイナリーパッケージとして配布されている ROS パッケージ以外を利用する場合、GitHub を使います。URL が分かれば上の手順だけで簡単に ROS のパッケージを自分のワークスペースにインポートし利用することができます。

では、次にパッケージのディレクトリ構成を確認します。ダウンロードしているパッケージがバージョンアップされている場合などには、下記の実行例とファイル名が異なったり、ファイルが追加・削除されている場合があります。

```
1 $ cd ~/catkin_ws/src/ypspur_ros/
2 $ ls
3 CMakeLists.txt msg package.xml src
4 $ ls msg/
5 ControlMode.msg DigitalInput.msg DigitalOutput.msg JointPositionControl.msg
6 $ ls src/
7 getID.sh joint_position_to_joint_trajectory.cpp joint_tf_publisher.cpp ypspur_ros.cpp
```

CMakeLists.txt と package.xml には、使っているライブラリの一覧や、生成する実行ファイルと C++ のソースコードの対応など、このパッケージをビルドするために必要な情報が書かれています。msg ディレクトリには、このパッケージ独自のデータ形式の定義が、src ディレクトリには、このパッケージに含まれるプログラム(ノード)のソースコードが含まれています。

次に catkin_make コマンドで、ダウンロードした ypspur_ros パッケージを含む、ワークスペース全体をビルドします。catkin_make は、ワークスペースの最上位ディレクトリ(~/catkin_ws/)で行います。

```
1 $ cd ~/catkin_ws/
2 $ catkin_make
3     実行結果 . . .
```

ROSノードの理解とビルド・実行

端末を開き、ひな形をダウンロードします。

```
1 $ cd ~/catkin_ws/src/
2 $ git clone https://github.com/BND-tc/rsj_robot_test.git
```

ソースファイルの編集にはお好みのテキストエディターが利用可能です。本セミナーの説明ではメジャーなテキストエディタである emacs の画面で例を示します。Linux がはじめての方には gedit もおすすめです。

お好みのテキストエディターで ~/catkin_ws/src/rsj_robot_test/src/rsj_robot_test.cpp を開きます。

The screenshot shows an Emacs window with a dark theme, displaying C++ code for a ROS node. The code defines a class `RsjRobotTestNode` with private members for a node handle and subscribers/publishers, and a public constructor and main loop. The code uses ROS message types like `Odometry` and `Twist`. The status bar at the bottom indicates the file is `rsj_robot_test.cpp`, at line 1, in Git master, with C++ abbreviations enabled.

```
#include <ros/ros.h>
#include <nav_msgs/Odometry.h>
#include <geometry_msgs/Twist.h>

#include <tf/transform_datatypes.h>

class RsjRobotTestNode
{
private:
    ros::NodeHandle nh_;

    ros::Subscriber sub_odom_;
    ros::Publisher pub_twist_;

    void cbodom(const nav_msgs::Odometry::ConstPtr &msg)
    {

public:
    RsjRobotTestNode()
        : nh_()
    {
        pub_twist_ = nh_.advertise<geometry_msgs::Twist>(
            "cmd_vel", 5);
        sub_odom_ = nh_.subscribe(
            "odom", 5, &RsjRobotTestNode::cbodom, this);
    }
    void mainloop()
    {
        ROS_INFO("Hello ROS World!");

        ros::Rate rate(10.0);
        while (ros::ok())
    }

U:--- rsj_robot_test.cpp Top L1 Git-master (C++/l Abbrev)
Opening directory: 許可がありません, /home/oit/.emacs.d/auto-save-list/
```

基本的なコードを読み解く

このコードが実行されたときの流れを確認しましょう。

まず、先頭部分では必要なヘッダファイルをインクルードしています。

```
1 #include <ros/ros.h>
```

続いて、`RsjRobotTestNode`クラスを定義しています。ROSプログラミングの際には、基本的にノードの持つ機能をクラスとして定義し、これを呼び出す形式を取ることが標準的です。クラスを使用せずに書くことも可能ですが、気をつけなければならない点が多くなるため、本セミナーではクラスでの書き方のみを解説します。

```
1 class RsjRobotTestNode
2 {
3     // (略)
4     public:
5         // (略)
6         void mainloop()
7         {
8             ROS_INFO("Hello ROS World!");

9             ros::Rate rate(10.0);
10            while(ros::ok())
11            {
12                ros::spinOnce();
13                // ここに速度指令の出力コード
14                rate.sleep();
15            }
16            // ここに終了処理のコード
17        }
18    };
19};
```

`RsjRobotTestNode`クラスのメンバ関数である`mainloop`関数の中では、ROSで情報を画面などに出力する際に用いる`ROS_INFO`関数を呼び出して、"Hello ROS World!"と表示しています。他にも、`ROS_DEBUG`、`ROS_WARN`、`ROS_ERROR`、`ROS_FATAL`関数が用意されています。

`ros::Rate rate(10.0)`で、周期実行のためのクラスを初期化しています。初期化時の引数で実行周波数(この例では10Hz)を指定します。

`while(ros::ok())`で、メインの無限ループを回します。`ros::ok()`を`while`の条件にすることで、ノードの終了指示が与えられたとき(**Ctrl+c**が押された場合も含む)には、ループを抜けて終了処理などが行えるようになっています。

ループ中では、まず`ros::spinOnce()`を呼び出して、ROSのメッセージを受け取るといった処理を行います。`ros::spinOnce()`はその時点で届いているメッセージの受け取り処理を済ませた後、すぐに処理を返します。`rate.sleep()`は、先ほど初期化した実行周波数を維持するように`sleep`します。

なお、ここではクラスを定義しただけなので、中身が呼び出されることはできません。後ほど実体化されたときに初めて中身が実行されます。

続いて、C++の`main`関数が定義されています。ノードの実行時にはここから処理がスタートします。

```
1 int main(int argc, char **argv) {  
2     ros::init(argc, argv, "rsj_robot_test_node");  
3  
4     RsjRobotTestNode robot_test;  
5  
6     robot_test.mainloop();  
7 }
```

はじめに`ros::init`関数を呼び出して、ROSノードの初期化を行います。1、2番目の引数には`main`関数の引数をそのまま渡し、3番目の引数にはこのノードの名前(この例では"rsj_robot_test_node")を与えます。

次に`RsjRobotTestNode`クラスの実体を作成します。ここでは`robot_test`と名前をつけています。

最後に実体化した`robot_test`のメンバ関数、`mainloop`を呼び出します。`mainloop`関数の中は無限ループになっているため、終了するまでの間`ros::spinOnce()`、`rate.sleep()`が呼び出され続けます。

つまり、`rsj_robot_test`は特に仕事をせず"Hello ROS World!"と画面に表示します。

ビルド&実行

ROS上でこのパッケージをビルドするためには、`catkin_make`コマンドを用います。

```
1 $ cd ~/catkin_ws/  
2 $ catkin_make
```

端末で実行してみましょう。

ROSシステムの実行の際、ROSを通してノード同士がデータをやりとりするために用いる、`roscore`を起動しておく必要があります。2つの端末を開き、それぞれ以下を実行して下さい。

1つ目の端末：

```
1 $ roscore
```

ROSでワークスペースを利用するとき、端末でそのワークスペースをアクティベートすることが必要です。そのためにワークスペースの最上のディレクトリで`source devel/setup.bash`を実行します。このコマンドはワークスペースの情報を利用中の端末に読み込みます。しかし、これは一時的にしか効果がないので新しい端末でワークスペースを利用し始めると必ずまずは`source devel/setup.bash`を実行しなければなりません。一つの端末で一回だけ実行すれば十分です。その端末を閉じるまで有効です。

2つ目の端末で下記を実行します。

```
1 $ cd ~/catkin_ws/  
2 $ source devel/setup.bash  
3 $ rosrun rsj_robot_test rsj_robot_test_node  
4 [ INFO] [1466002781.136800000]: Hello ROS World!
```

Hello ROS World!と表示されれば成功です。

以上の手順で、ROSパッケージに含まれるノードのソースコードを編集し、ビルドして実行できるようになりました。

両方の端末で **Ctrl+c** でノードとroscoreを終了します。

ロボットに速度指令を与える

まず、ロボットに速度指令(目標並進速度・角速度)を与えるコードを追加します。ひな形には、既に速度指令値が入ったメッセージを出力するための初期化コードが含まれていますので、この部分の意味を確認します。

```
1 RsjRobotTestNode()
2   : nh_()
3 {
4   pub_twist_ = nh_.advertise<geometry_msgs::Twist>(
5     "cmd_vel", 5);
6   sub_odom_ = nh_.subscribe(
7     "odom", 5, &RsjRobotTestNode::cbOdom, this);
8 }
```

ソースコード中の、`RsjRobotTestNode`クラスの`RsjRobotTestNode`関数は、クラスのコンストラクタと呼ばれるもので、クラスが初期化されるときに自動的に呼び出されます。2行目の: `nh_()`の部分では、クラスのメンバ変数である`nh_`を、引数なしで初期化しています。このコンストラクタの中で、

```
1 nh_.advertise<geometry_msgs::Twist>("cmd_vel", 5);
```

の部分で、このノードが、これからメッセージを出力することを宣言しています。`advertise`関数に与えている引数は以下のようない意味を持ちます。

"cmd_vel"
出力するメッセージを置く場所(トピックと呼ぶ)を指定
5 メッセージのバッファリング量を指定(大きくすると、処理が一時的に重くなったときなどに受け取り側の読み飛ばしを減らせる)

先頭の`nh_`は`ros::NodeHandle`型のメンバ変数で、トピックやパラメータといったROSノードの基本的な機能を初期化するために使用します。`nh_()`のように引数無しで初期化することでグローバル名前空間(/)を使う設定になるので、/という名前空間の下の`cmd_vel`という名前のトピック、つまり`/cmd_vel`というトピックにメッセージを出力することを意味します。

`advertise`関数についている`<geometry_msgs::Twist>`の部分は、メッセージの型を指定しています。これは、幾何的・運動学的な値を扱うメッセージを定義している`geometry_msgs`パッケージの、並進・回転速度を表す`Twist`型です。この指定方法は、C++のテンプレートという機能を利用してますが、ここでは「`advertise`のときはメッセージの型指定を<>の中に書く」とだけ覚えておけば問題ありません。

`mainloop`関数中の「ここに速度指令の出力コード」の部分を以下のように編集することで、速度指令のメッセージを出力(`publish`)します。

```
1 void mainloop()
2 {
3   ROS_INFO("Hello ROS World!");
4
5   ros::Rate rate(10.0);
6   while(ros::ok())
7   {
8     ros::spinOnce();
9     // ここに速度指令の出力コード
10    geometry_msgs::Twist cmd_vel;
11    cmd_vel.linear.x = 0.05;
12    cmd_vel.angular.z = 0.0;
13    pub_twist_.publish(cmd_vel);
14
15    rate.sleep();
16  }
17  // ここに終了処理のコード
18 }
```

ビルド&実行

```
1 $ cd ~/catkin_ws/
2 $ catkin_make
```

この際、ビルドエラーが出ていないか良く確認して下さい。エラーが出ている場合は、ソースコードの該当箇所を確認・修正して下さい。

実行の際、まずroscoreとypspur_rosを起動します。ypspur_rosの中では、ロボットの動作テストの際に使用したypspur-coordinatorが動いています。なお、roscoreは前のものを実行し続けている場合はそのまま使用できます。コマンド入力の際はタブ補完を活用しましょう。

```
1 $ roscore
```

2番目の端末を開いて、下記を実行します。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun ypspur_ros ypspur_ros _param_file:=/home/【ユーザ名】/params/rsj-seminar20???.param \
4   _port:=/dev/serial/by-id/usb-T-frog_project_T-frog_Driver-if00 \
5   _compatible:=1
```

続いて、別の端末でrsj_robot_test_nodeノードを実行します。まずは、ロボットのホイールを浮かせて走り出さない状態にして実行してみましょう。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun rsj_robot_test rsj_robot_test_node
Hello ROS World!
```

ゆっくりとホイールが回れば正しく動作しています。それぞれの端末で`Ctrl+C`を押し、終了します。

小課題(1)

速度、角速度を変更して動作を確認してみましょう。

ロボットの状態を表示する

ロボットの状態を表示するコードを追加

まず、ロボットの動作したときの移動量やオドメトリ座標を取得、表示するコードを追加します。ひな形には、既に移動量や座標が入ったメッセージを受け取るコードが含まれていますので、この部分の意味を確認します。

```
1 RsjRobotTestNode()
2   : nh_()
3 {
4   pub_twist_ = nh_.advertise<geometry_msgs::Twist>(
5     "cmd_vel", 5);
6   sub_odom_ = nh_.subscribe(
7     "odom", 5, &RsjRobotTestNode::cbOdom, this);
8 }
```

この中で

```
1 sub_odom_ = nh_.subscribe("odom", 5, &RsjRobotTestNode::cbOdom, this);
```

の部分で、このノードがこれからメッセージを受け取ることを宣言しています。subscribe関数に与えている引数は以下ののような意味を持ちます。

"odom"
受け取るメッセージが置かれている場所(トピック)を指定
5 メッセージのバッファリング量を指定(大きくすると、処理が一時的に重くなったときなどに受け取り側の読み飛ばしを減らせる)
&RsjRobotTestNode::cbOdom
メッセージを受け取ったときに呼び出す関数を指定(RsjRobotTestNodeクラスの中にある、cbOdom関数)
this メッセージを受け取ったときに呼び出す関数がクラスの中にある場合にクラスの実体を指定(とりあえず、おまじないと思って構いません。)

こちらもグローバル名前空間(/)を使う設定で初期化されているnh_を使っているので、/下のodom、つまり/odomという名前のトピックからメッセージを取得することを意味します。これにより、rsj_robot_test_nodeノードが/odomトピックからメッセージをうけとると、cbOdom関数が呼び出されるようになります。

続いてcbodom関数の中身を確認しましょう。

```
1 void cbodom(const nav_msgs::Odometry::ConstPtr &msg)
2 {
3 }
```

const nav_msgs::Odometry::ConstPtrは、constな(内容を書き換えられない)nav_msgsパッケージに含まれるOdometry型のメッセージの、const型ポインタを表しています。&msgの&は、参照型(内容を書き換えられるように変数を渡すことができる)という意味ですが、(const型なので)ここでは特に気にする必要はありません。

cbodom関数に、以下のコードを追加してみましょう。これにより、受け取ったメッセージの中から、ロボットの並進速度を取り出して表示できます。

```
1 void cbodom(const nav_msgs::Odometry::ConstPtr &msg)
2 {
3     ROS_INFO("vel %f", msg->twist.twist.linear.x);
4 }
```

ここで、msg->twist.twist.linear.xの意味を確認します。nav_msgs::Odometryメッセージには、下記のように入れ子状にメッセージが入っています。

- std_msgs/Header header
- string child_frame_id
- geometry_msgs/PoseWithCovariance pose
- geometry_msgs/TwistWithCovariance twist

全て展開すると、以下の構成になります。

- std_msgs/Header header
 - uint32 seq
 - time stamp
 - string frame_id
- string child_frame_id
- geometry_msgs/PoseWithCovariance pose
 - geometry_msgs/Pose pose
 - geometry_msgs/Point position
 - float64 x
 - float64 y
 - float64 z
 - geometry_msgs/Quaternion orientation
 - float64 x
 - float64 y
 - float64 z
 - float64 w
 - float64[36] covariance
- geometry_msgs/TwistWithCovariance twist
 - geometry_msgs/Twist twist
 - geometry_msgs/Vector3 linear
 - float64 x ロボット並進速度
 - float64 y
 - float64 z
 - geometry_msgs/Vector3 angular
 - float64 x
 - float64 y
 - float64 z ロボット角速度
 - float64[36] covariance

読みたいデータであるロボット並進速度を取り出すためには、これを順にたどっていけば良く、msg->twist.twist.linear.xとなります。msgはクラスへのポインタなので「->」を用い、以降はクラスのメンバ変数へのアクセスなので「.」を用いてアクセスしています。

ビルド&実行

```
1 $ cd ~/catkin_ws/
2 $ catkin_make
```

この際、ビルドエラーが出ていないか、良く確認して下さい。エラーが出ている場合は、ソースコードの該当箇所を確認・修正して下さい。

まず、先ほどと同様にroscoreと、ypspur_rosを起動します(以降、この手順の記載は省略します)。

```
1 $ roscore
```

2番目の端末を開いて、下記を実行します。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun ypspur_ros ypspur_ros _param_file:=/home/【ユーザー名】/params/rsj-seminar20???.param \
4   _port:=/dev/serial/by-id/usb-T-frog_project_T-frog_Driver-if00 \
5   _compatible:=1
```

続いて、rsj_robot_test_nodeノードを実行します。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun rsj_robot_test rsj_robot_test_node
Hello ROS World!
4 vel: 0.0500
5 vel: 0.0500
6 vel: 0.0500
7 vel: 0.0500
8 vel: 0.0500
```

ロボットのホイールが回転し、先ほどの小課題で設定した走行指令の値と近い値が表示されれば正しく動作しています。

小課題(2)

同様に、ロボットの角速度を表示してみましょう。

シーケンス制御

時間で動作を変える

メインループを以下のように変更してみましょう。

```
1 void mainloop()
2 {
3     ROS_INFO("Hello ROS World!");
4
5     ros::Rate rate(10.0);
6     ros::Time start = ros::Time::now();
7     while(ros::ok())
8     {
9         ros::spinOnce();
10        ros::Time now = ros::Time::now();
11
12        geometry_msgs::Twist cmd_vel;
13        if(now - start > ros::Duration(3.0))
14        {
15            cmd_vel.linear.x = 0.05;
16            cmd_vel.angular.z = 0.0;
17        }
18        pub_twist_.publish(cmd_vel);
19
20        rate.sleep();
21    }
22 }
```

これは、メインループ開始時刻から3.0秒後に、並進速度0.05m/sの指令を与えるコードです。ros::Time型(時刻を表す)同士の減算結果はros::Duration型(時間を表す)になります。比較演算子で比較できます。したがって、now - start > ros::Duration(3.0)の部分は、開始から3秒後にtrueになります。

先ほどと同様にビルドし、ypspur_rosとrsj_robot_test_nodeを起動して動作を確認します。

センシング結果で動作を変える

cbodomで取得したオドメトリのデータを保存しておくように、以下のように変更してみましょう。

```
1 void cbodom(const nav_msgs::Odometry::ConstPtr &msg)
2 {
3     ROS_INFO("vel %f", msg->twist.twist.linear.x);
4     odom_ = *msg; // 追記
5 }
```

また、class RsjRobotTestNodeの先頭に下記の変数定義を追加します。

```
1 class RsjRobotTestNode
2 {
3     private:
4         nav_msgs::Odometry odom_;
```

また、odom_の中で方位を表すクオータニオンをコンストラクタ(RsjRobotTestNode()関数)の最後で初期化しておきます。

```
1 RsjRobotTestNode():
2 {
3     (略)
4     odom_.pose.pose.orientation.w = 1.0;
5 }
```

mainloop()を以下のように変更してみましょう。

```
1 void mainloop()
2 {
3     ROS_INFO("Hello ROS World!");
4
5     ros::Rate rate(10.0);
6     while(ros::ok())
7     {
8         ros::spinOnce();
9
10        geometry_msgs::Twist cmd_vel;
11        if(tf::getYaw(odom_.pose.pose.orientation) > 1.57)
12        {
13            cmd_vel.linear.x = 0.0;
14            cmd_vel.angular.z = 0.0;
15        }
16        else
17        {
18            cmd_vel.linear.x = 0.0;
19            cmd_vel.angular.z = 0.1;
20        }
21        pub_twist_.publish(cmd_vel);
22
23        rate.sleep();
24    }
25 }
```

これは、オドメトリのYaw角度(旋回角度)が1.57ラジアン(90度)を超えるまで、正方向に旋回する動作を表しています。

先ほどと同様にビルドし、yppspur_rosとrsj_robot_test_nodeを起動して動作を確認します。

小課題(3)

1m 前方に走行し、その後で帰ってくるコードを作成してみましょう。(1m 前方に走行し 180 度旋回して 1m 前方に走行するか、1m 前方に走行し1m後方に走行すればよい。)

余裕があれば、四角形を描いて走行するコードを作成してみましょう。

ROSを用いた点群取得

ここではURG-04LX-UG01の場合を例に、点群を取得してロボットの動作に反映する方法を習得します。

urg_nodeをインストール

```
1 $ sudo apt-get update
2 $ sudo apt-get install ros-kinetic-urg-node
```

rsj_robot_test_nodeでURGのデータ取得

ソースコードの変更

ソースコードの先頭部分で、スキャンデータのメッセージ型をincludeします。

```
1 (略)
2 #include <geometry_msgs/Twist.h>
3 #include <sensor_msgs/LaserScan.h> // <- スキャンデータのメッセージ型をinclude
```

RsjRobotTestNodeクラス内で、sub_odom_(サブスクライバ)を定義しているところに、URG用のサブスクライバを追加します。

```
1 class RsjRobotTestNode
2 {
3     private:
4         (略)
5         ros::Subscriber sub_odom_;
6         ros::Subscriber sub_scan_; // <- URG用のサブスクライバを追加
```

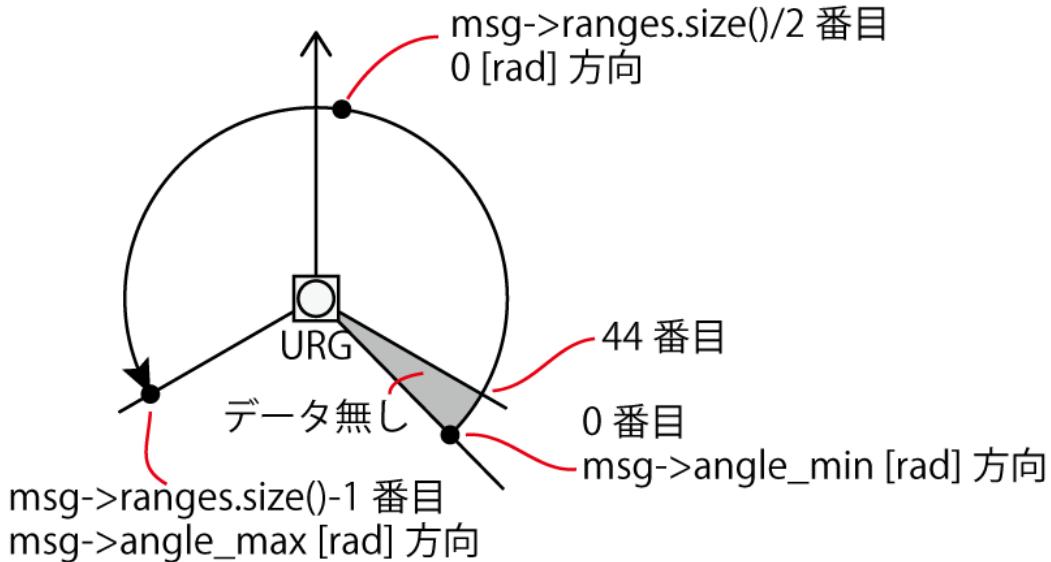
RsjRobotTestNodeのコンストラクタに、URG用のサブスクライバ初期化コードを追加します。

```
1 RsjRobotTestNode()
2     : nh_()
3 {
4     pub_twist_ = nh_.advertise<geometry_msgs::Twist>(
5         "cmd_vel", 5);
6     sub_odom_ = nh_.subscribe(
7         "odom", 5, &RsjRobotTestNode::cbOdom, this);
8     sub_scan_ = nh_.subscribe(
9         "scan", 5, &RsjRobotTestNode::cbScan, this); // <- URG用のサブスクライバ初期化コードを追加
```

更に、RsjRobotTestNodeクラスに、URG用のコールバック関数を追加します。(cbodomの後の位置など)

```
1 void cbScan(const sensor_msgs::LaserScan::ConstPtr &msg)
2 {
3     int i = msg->ranges.size() / 2;
4     if (msg->ranges[i] < msg->range_min || // エラー値の場合
5         msg->ranges[i] > msg->range_max || // 測定範囲外の場合
6         std::isnan(msg->ranges[i])) // 無限遠の場合
7     {
8         ROS_INFO("front-range: measurement error");
9     }
10    else
11    {
12        ROS_INFO("front-range: %0.3f",
13                 msg->ranges[msg->ranges.size() / 2]);
14    }
15 }
```

このコールバック関数中では、距離データ配列のうち、配列の中央の距離を表示しています。すなわち、URGの場合、正面方向の距離データ(m単位)が表示されます。また、msg->rangesの値がmsg->range_minより小さい場合は、測定エラー(遠すぎて測定できない、など)を意味しています。なお、msg->ranges[0]はmsg->angle_min方向(rad単位)、msg->ranges[msg->size()-1]はmsg->angle_max方向(rad単位)を表します。



ビルド&実行

まず、catkin_wsでcatkin_makeを実行して、追加したコードをビルドします。roscore、ypspur_ros、urg_node、rsj_robot_test_nodeを実行したいので、端末を4つ用意して、下記それぞれを実行します。

URGとロボットのUSBケーブルを接続しておきます。また、ロボットが走り出さないように、電池ボックスのスイッチをOFFにしておくとよいでしょう。

1つ目の端末でroscoreを起動

```
1 $ roscore
```

2つ目の端末でypspur_rosを起動

```
1 $ rospack find ypspur_ros
2 $ rospack depends ypspur_ros
3 $ rospack param -l ypspur_ros
```

3つ目の端末でurg_nodeを起動

```
1 $ rospack find urg_node
2 $ rospack depends urg_node
```

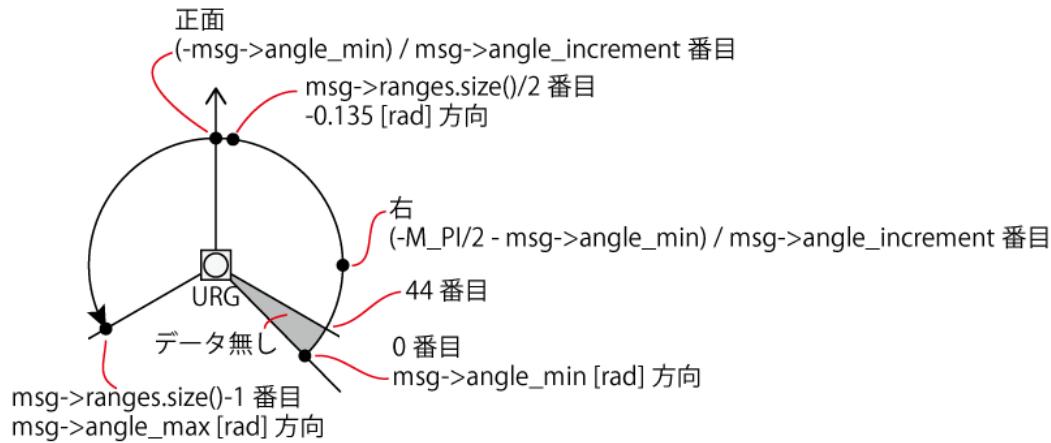
4つ目の端末で作成したプログラムを実行

```
1 $ rospack find rsj_robot_test
2 $ rospack depends rsj_robot_test
3 $ rospack build rsj_robot_test
```

URGの正面方向に手などを置いて、距離の値が変わることを確認して下さい。

小課題

URGの真横方向の距離を表示してみましょう。



ROSの便利機能

ROSには多くの便利な機能が用意されています。ここでは、そのうちのよく利用するコマンドなどの一部を紹介します。

roslaunch

[ROSを用いた点群取得](#)では、端末を4つ起動して、それぞれroscoreとrosrunコマンドでノードを起動していました。複雑なロボットシステムではこれが、100個以上のノードになる場合もあり、手作業ではやっていられません。

そこでROSには、launchファイルに、起動するノードやそのパラメータを書いておき、roslaunchコマンドで一括起動する方法が用意されています。下記の操作を行い、launchファイルを追加してみましょう。

1. rsj_robot_testパッケージに、launchディレクトリを作成

```
1 $ mkdir ~/catkin_ws/src/rsj_robot_test/launch
```

2. robot_test.launchファイルを作成して開く

```
1 $ gedit ~/catkin_ws/src/rsj_robot_test/launch/robot_test.launch
```

3. 下記内容を記入

```
1 <launch>
2   <node pkg="ypspur_ros" type="ypspur_ros" name="ypspur_ros">
3     <param name="port" value="/dev/serial/by-id/usb-T-frog_project_T-frog_Driver-if00" />
4     <param name="param_file"
5       value="/home/${env USER}/params/rsj-seminar20?.param該当するものに置き換えること" />
6     <param name="compatible" value="1" />
7   </node>
8   <node pkg="urg_node" type="urg_node" name="urg_node">
9     <param name="serial_port"
10       value="/dev/serial/by-id/usb-Hokuyo_Data_Flex_for_USB_URG-Series_USB_Driver-if00" />
11   </node>
12   <node pkg="rsj_robot_test" type="rsj_robot_test_node" name="robot_test" output="screen">
13     </node>
14   </launch>
```

作成したlaunchファイルは下記のコマンドで実行できます。

```
1 $ rosrun rsj_robot_test robot_test.launch
```

これは、rsj_robot_testパッケージ中の、robot_test.launchを実行する、という指示を表しています。下記に代表的なタグの説明を示します。

nodeタグ

動するノードを指定します。nodeタグの各属性の意味は下記の通りです。

name	ノードインスタンスの名
pkg	ノードを定義するパッケージ名
type	ノードの実行ファイル名（バイナリーやPythonスクリプト）
output	ノードのstdoutの先：定義しないとstdout（ROS_INFOやstd::coutへの出力等）は端末で表示されず、~/.ros/log/に保存されるログファイルだけに出力される。端末で表示したい場合はscreenにします。

paramタグ

パラメータサーバーにパラメータを設定します。起動されるノードはこのパラメータが利用できます。

paramは<launch>、</launch>の間に入れるとグローバルパラメータに、<node>、</node>の間に入れるとプライベートパラメータになります。

グローバルとプライベートでは最終的に展開される名前が異なってきます。例えば、同じbarというパラメータでも次のような違いがあります。

- グローバルパラメータの場合、ROSのノードからは/barという名前で参照される。
- ノード名fooのプライベートパラメータの場合、ROSのノードからは/foo/barという名前で参照される。

各属性は下記の通りです。

name

パラメータ名
value パラメータの値
type double, int, string, bool など(一意に決まるときは省略可能)

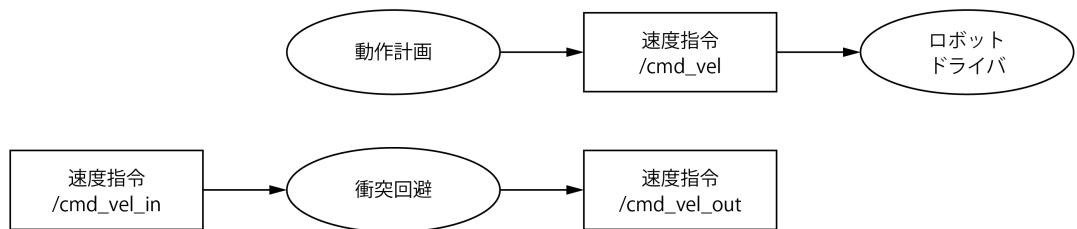
remapタグ

ノードとトピックをつなぎ変えます。

remapタグの各属性の意味は下記の通りです。

from 変更前のトピック名
to 変更後のトピック名

これを使うことで、ノードとトピックをつなぎ変えることができます。たとえば、下記のような、動作計画のノードと、ロボットのドライバノードがつながっている状態から、新たに衝突回避のノードを加えたいとします。



remapを用いることで、各ノードのソースコードを変更することなく、ノードとトピックの接続だけ切り替えて、動作計画とロボットドライバの間に、衝突回避を追加することができます。

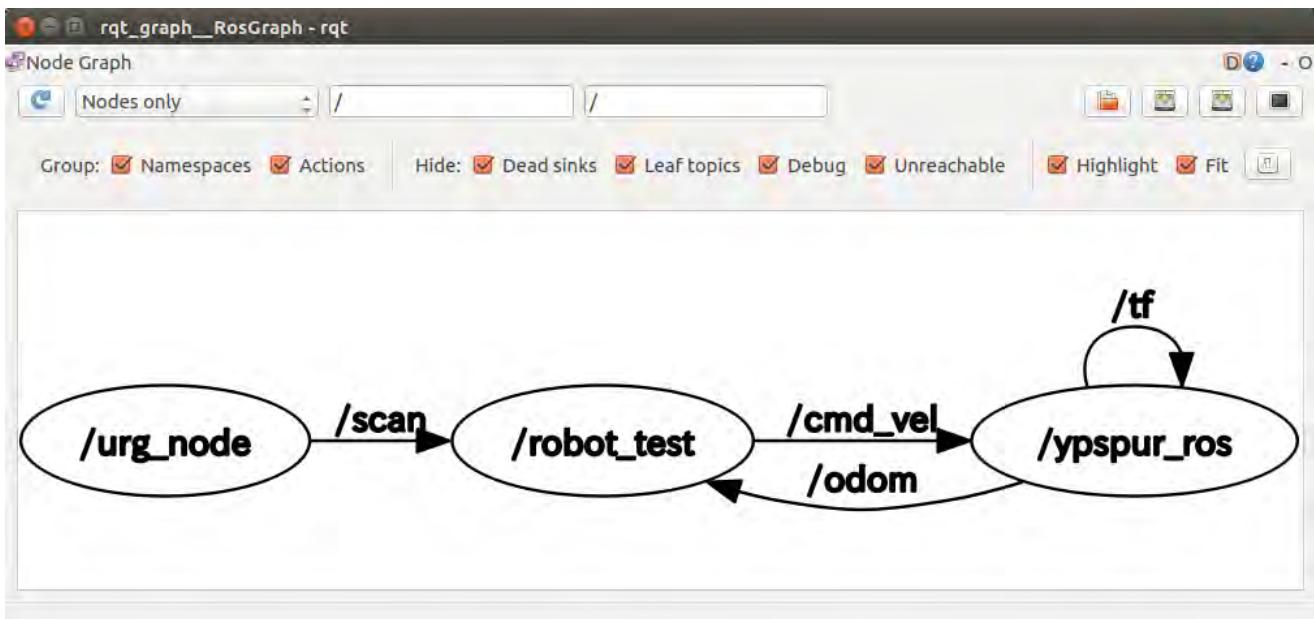


rqt_graph

トピックとノードの接続状態を可視化することができます。ロボットとURGをPCに接続して、roslaunchの項で説明したようにrobot_test.launchを実行し、その状態で下記コマンドを実行してみましょう。

1 \$ rqt_graph

以下の画像のように、ノードとトピックの接続グラフが表示されます。



rostopic

デバッグなどのため、ROSのトピックに流れているメッセージを確認したいときや、試しにメッセージを送信したいときに、コマンドラインのツールでこれらの処理を行うことができます。

- 存在するトピックを確認する

```

1 $ rostopic list
2 /ad/ad0
3 /ad/ad1
4 /ad/ad2
5 /ad/ad3
6 /ad/ad4
7 /ad/ad5
8 /ad/ad6
9 /ad/ad7
10 /cmd_vel
11 /control_mode
12 /diagnostics
13 /laser_status
14 /odom
15 /rosout
16 /rosout_agg
17 /scan
18 /tf
19 /tf_static
20 /urg_node/parameter_descriptions
21 /urg_node/parameter_updates
22 /wrench

```

- 1つのトピックに流れているメッセージを確認する

```
1 $ rostopic echo /odom
```

- 1つのトピックにメッセージを送信する

Tabでトピック名、データ型及びメッセージのテンプレートが出せます。

```

1 $ rostopic pub -1 /cmd_vel geometry_msgs/Twist [Tab補完で表示]"linear
2   x: 0.0
3   y: 0.0
4   z: 0.0
5   angular:
6     x: 0.0
7     y: 0.0
8     z: 0.0"

```

-1を利用すると一回のみ送信します。 -1を削除するとrostopicはこのメッセージを1度送信し、**Ctrl+c**を入力するまで待機して、あとから起動したノードがメッセージを受け取れる状態を維持します。

rosbag

ROSで提供されているrosbagツールを用いると、ROS上で送信、受信されているデータ(メッセージ)を記録・再生することができます。

- データを記録(URGのデータ /scan と、オドメトリ /odom を記録する例)

```
1 $ rosbag record /scan /odom
```

記録の終了は、**Ctrl+c**で行います。記録されたデータは、「日付時刻.bag」のファイル名で保存されています。

- データを再生する

```
1 $ rosbag play ファイル名.bag
```

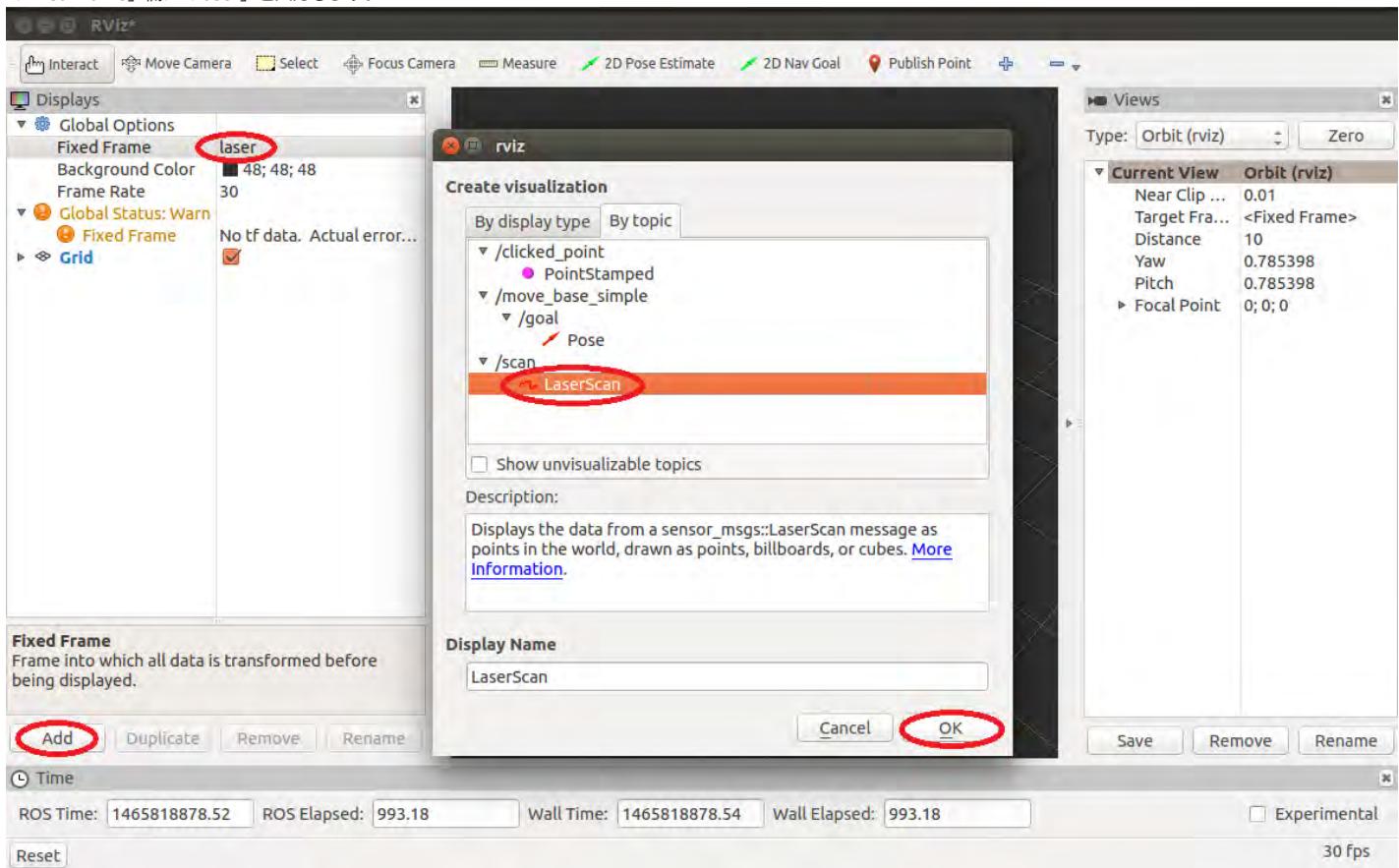
RViz

ROSでは、rvizという、データ可視化ツール(ビューワ)が提供されています。今回のセミナーの環境にも、インストールされており、URGのデータやオドメトリを表示することができます。ロボットとURGをPCに接続して、roslaunchの項で説明したようにrobot_test.launchを実行し、その状態で下記コマンドを実行してみましょう。

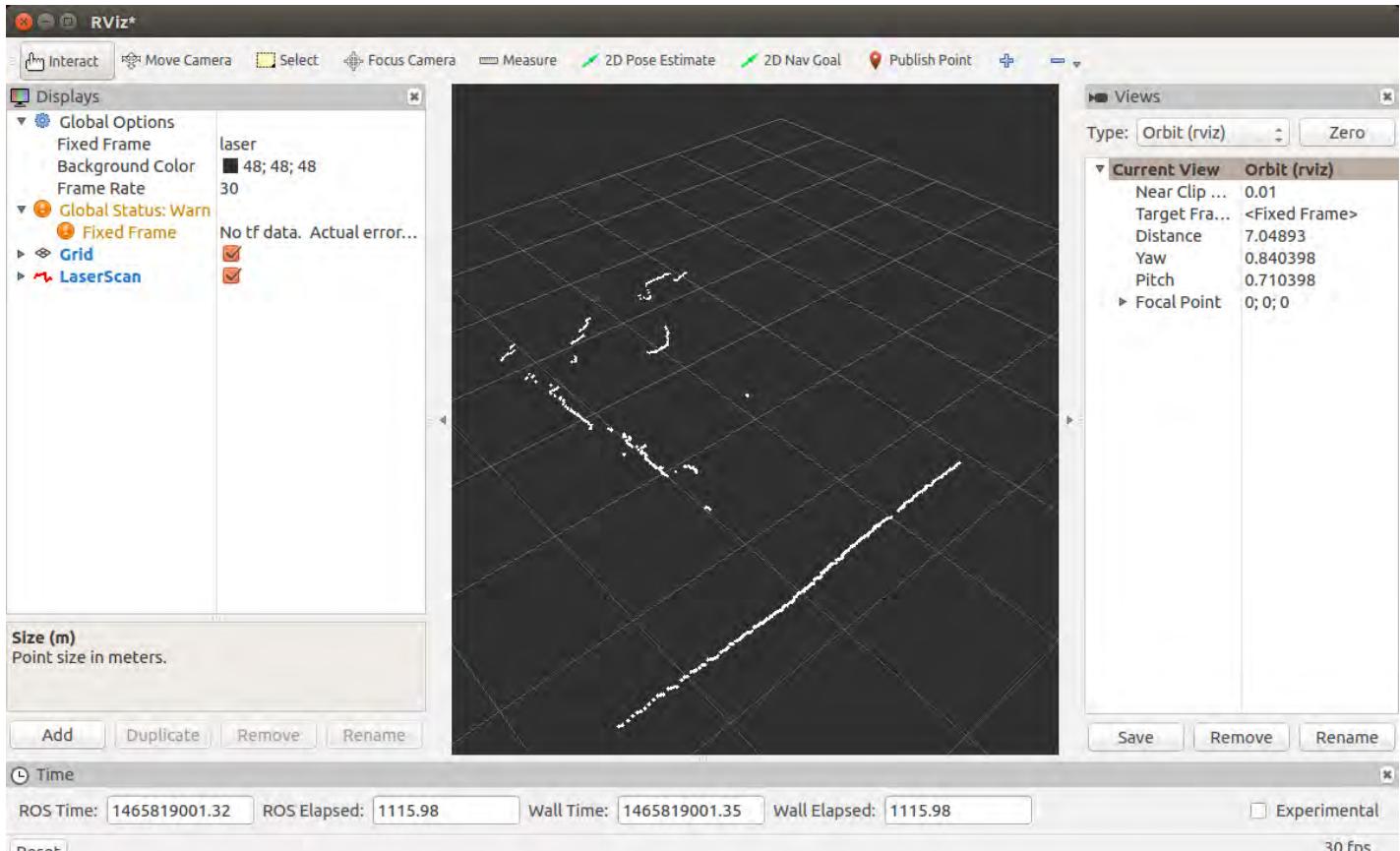
1

\$ rosrun rviz rviz

rviz画面中の、「add」ボタンをクリックし、開いた選択ウインドウ内で、「By topic」タブから、「/scan」中の「LaserScan」を選択します。また、「Global Option」の「Fixed Frame」欄に「laser」と入力します。



センサデータが取得できていれば、図のように、距離データがプロットされます。



3Dセンサからのデータ取得に関する実習

配布された3次元距離センサのデータについて、`roslaunch`を利用して2次元データに変換し、`RViz`で結果を確認してみましょう。お手持ちのセンサに応じて、以下を実施してください。

- [YVT-35LXの場合](#)
- [Xtion Pro Liveの場合](#)

YVT-35LX からのデータ取得

YVT-35LX からデータを取得し、2次元点群に変換する方法を説明します。

準備

パッケージのインストール

YVT-35LX を利用するためのパッケージと3次元点群から2次元データに変換するためのパッケージをインストールします。

```
1 $ sudo apt install ros-kinetic-hokuyo3d
```

launchファイルの入手

点群を3次元から2次元データに変換するパッケージの launch ファイル等をダウンロード

```
1 $ cd ~/catkin_ws/src
2 $ git clone https://github.com/KMiyawaki/rsj_pointcloud_to_laserscan.git
```

動作確認

ネットワークの設定

1. PC の LAN ポートに YVT-35LX を接続し、バッテリーのスイッチを入れる。

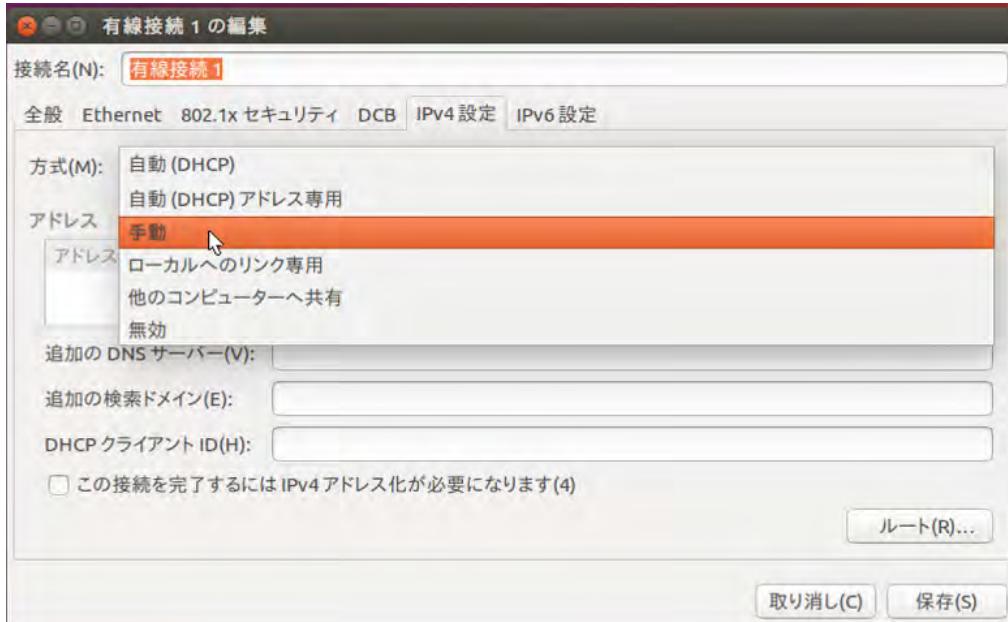
2. 画面右上のネットワークアイコンをクリック



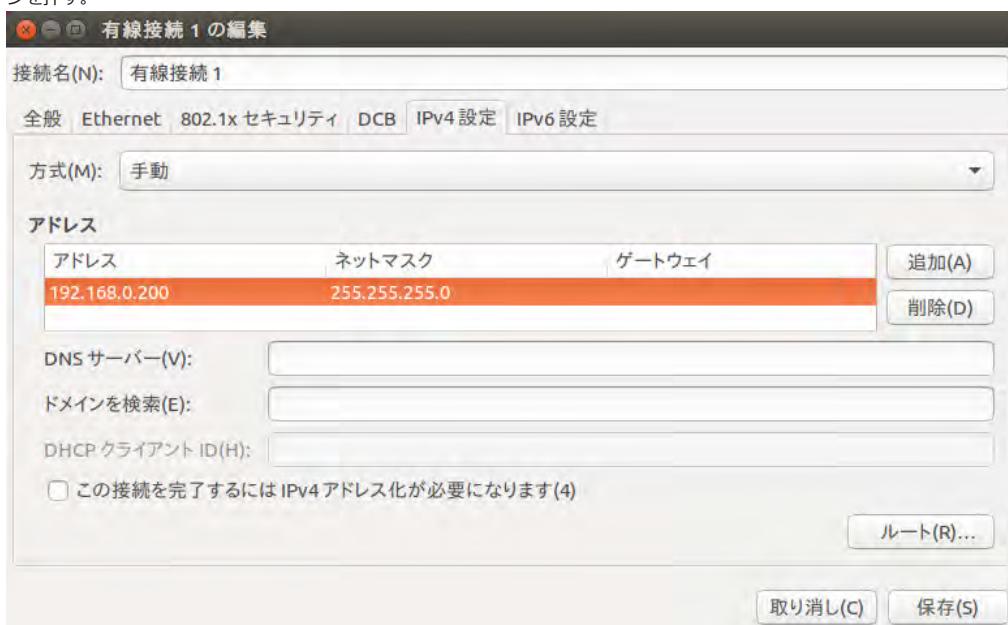
3. 有線接続の項目を選択し、「編集」ボタンをクリック



4. 「IPv4設定」のタブをクリックし、「方式」を手動に設定



5. 「アドレス」の項目で「追加」ボタンをクリックし、アドレス「192.168.0.*」（*は10以外の1から255の間の数字）とネットマスクを入力し、最後に「保存」ボタンを押す。



画面に「有線接続完了」の文字が表示されれば完了です。

注意：無線LANで既に192.168.0.系のネットワークに接続している場合、通信が不通となります。その場合はセンサを接続する前に既存のネットワークから切断してから上記の作業を行ってください。

launchファイルの実行

次のコマンドを実行します。

```
1 $ cd ~/catkin_ws/
2 $ catkin_make
3 $ source devel/setup.bash
4 $ roslaunch rsj_pointcloud_to_laserscan rsj_pointcloud_to_laserscan_3durg.launch
```

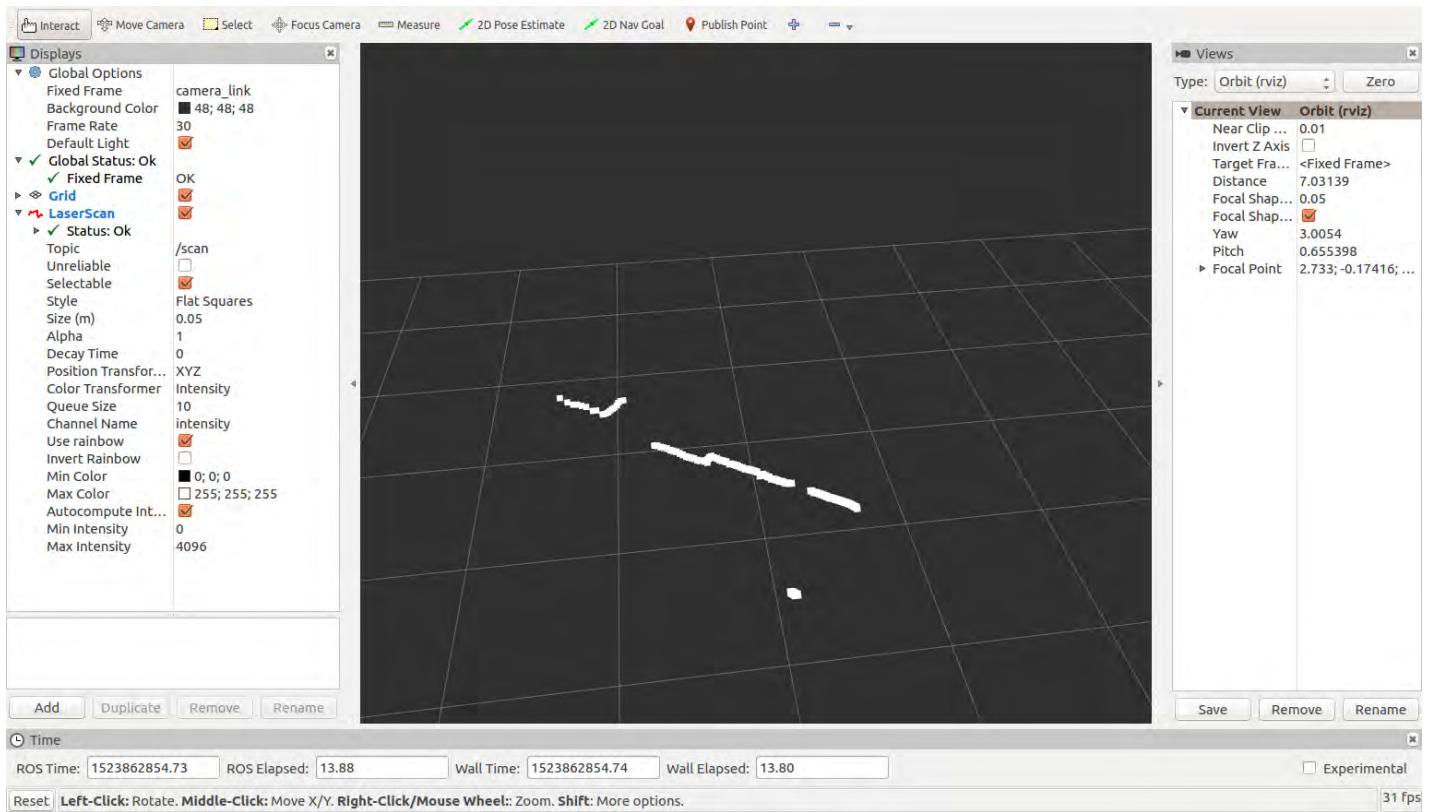
コンソールに赤字でエラーメッセージが出ていないかどうか確認してください。もしエラーメッセージが出ていたら、プログラムを **Ctrl+c** で終了し、もう一度上記を実行してみてください。

データの表示

別のコマンドターミナルを開き次を実行してください。

```
1 $ cd ~/catkin_ws/src/rsj_pointcloud_to_laserscan/config/rviz
2 $ rosrun rviz rviz -d view_scan.rviz
```

下図のように2次元の点群が表示されれば成功です。



この2次元点群は3次元センサが出力している3次元点群からpointcloud_to_laserscanノードを使って、高さの範囲指定で切り出して生成しているものです。3次元点群そのものの処理については後に続く実習で説明します。

Xtion PRO Live からのデータ取得

Xtion PRO Live からデータを取得し、2次元点群に変換する方法を説明します。

Xtion PRO Live は深度（物体までの距離）およびRGBカラー画像を取得できます。USBバスパワーで駆動可能なデブスセンサで、非常に扱いやすいのですが現在は廃番となつており入手することはできません。

ROSで利用可能な類似のセンサとしてOrbbec Astraがあります。またXtion PRO Liveの後継機としてXtion2がありますが、こちらはまだROS対応はしていません（2018/5/23時点）。

準備

パッケージのインストール

Xtion PRO Liveを利用するためのパッケージと3次元点群から2次元データに変換するためのパッケージをインストールします。[準備のページ](#)に同様の手順を書いていますので、すでにインストールされている方はこの手順は不要です。

```
1 $ sudo apt install ros-kinetic-openni2-camera
2 $ sudo apt install ros-kinetic-openni2-launch
3 $ sudo apt install ros-kinetic-pointcloud-to-laserscan
```

launchファイルの入手

点群を3次元から2次元データに変換するパッケージのlaunchファイル等をダウンロード

```
1 $ cd ~/catkin_ws/src
2 $ git clone https://github.com/kmiyawaki/rsj_pointcloud_to_laserscan.git
```

動作確認

launchファイルの実行

PCのUSBポートにXtion PRO Liveを接続し、次のコマンドを実行します。

```
1 $ cd ~/catkin_ws/
2 $ catkin_make
3 $ source devel/setup.bash
4 $ roslaunch rsj_pointcloud_to_laserscan rsj_pointcloud_to_laserscan.launch
```

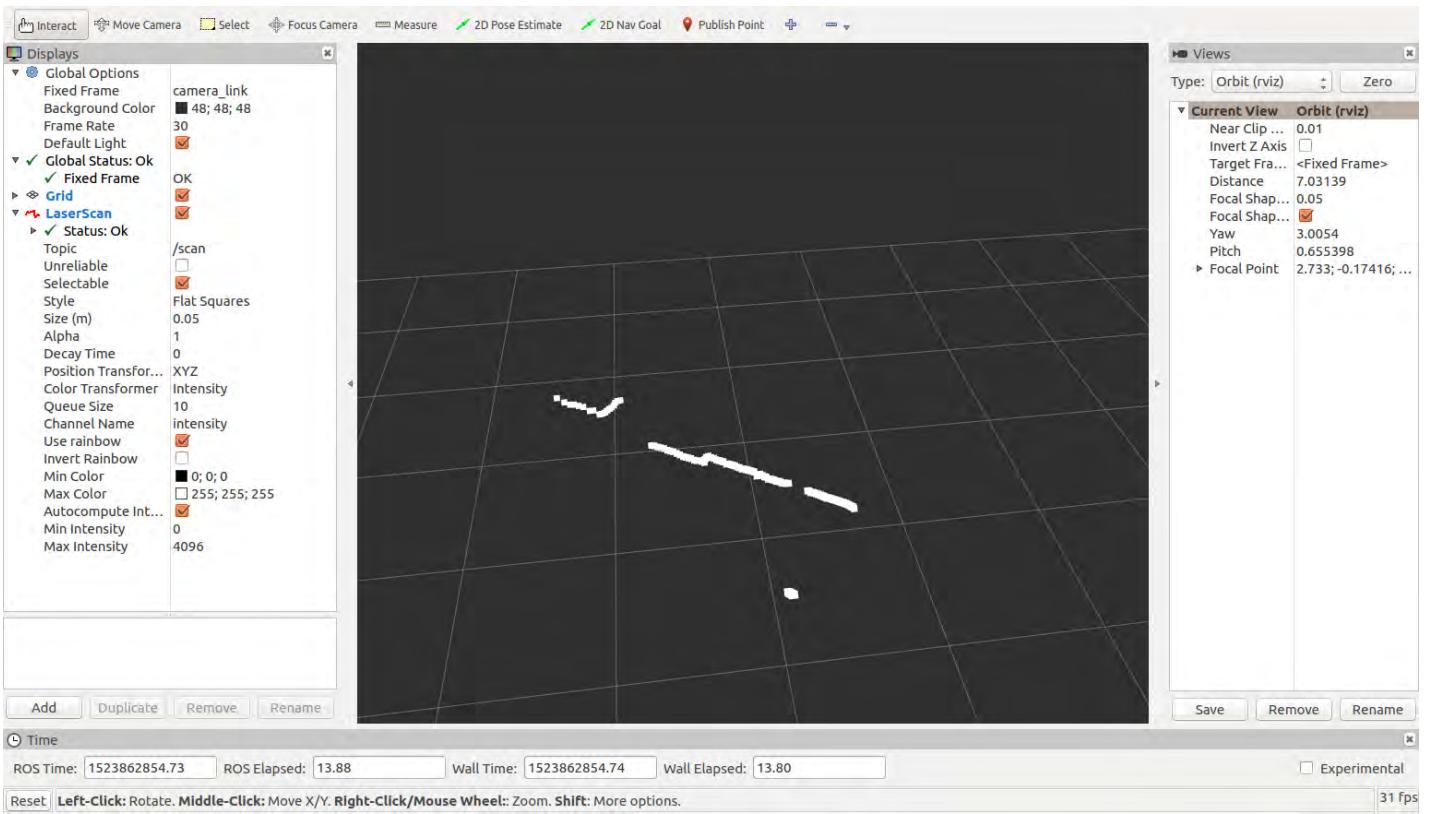
コンソールに赤字でエラーメッセージが出ていないかどうか確認してください。もしエラーメッセージが出ていたら、プログラムを[Ctrl+c](#)で終了し、Xtion PRO LiveをUSBポートから一旦抜いて再接続してからもう一度上記を実行してみてください。

データの表示

別のコマンドターミナルを開き次を実行してください。

```
1 $ cd ~/catkin_ws/src/rsj_pointcloud_to_laserscan/config/rviz
2 $ rosrn rviz rviz -d view_scan.rviz
```

下図のように2次元の点群が表示されれば成功です。



この2次元点群は3次元センサが出力している3次元点群からpointcloud_to_laserscanノードを使って、高さの範囲指定で切り出して生成しているものです。3次元点群そのものの処理については後に続く実習で説明します。

ROS navigation メタパッケージの利用

地図作成と、簡易的に調整済みのnavigationメタパッケージを利用して、自律ナビゲーションを体験します。

ROS navigation メタパッケージとは

navigationメタパッケージは、自己位置推定、大域的な経路計画、局所的な動作計画など、地図ベースの自律走行に必要なノードを含むパッケージ群で、世界中のROS開発者によって作成・管理されています。

世界中の英知の塊とも言えるnavigationメタパッケージですが、含まれるソフトウェアの内ではアドホックな構造やパラメータが多数あり、まともに動く(例えばつくばチャレンジを走破する)ようにチューニングするのは(これら全てを自分で実装できるくらいの)知識と経験が必要です。

本セミナーでは、地図作成と自律ナビゲーションを簡易的に調整済みのlaunchファイルを利用して動作させます。

必要なパッケージのインストール

下記コマンドを用いて地図生成に用いるslam_gmappingパッケージと、マウスでロボットを操縦するmouse_teleopパッケージをインストールします。

navigationメタパッケージはサイズが非常に大きいため、ダウンロードに時間がかかることがあります。 [準備のページ](#)に同様の手順を書いていますので、すでにインストールされている方はこの手順は不要です。

```
1 $ sudo apt-get install ros-kinetic-slam-gmapping ros-kinetic-mouse-teleop
2 $ sudo apt-get install ros-kinetic-map-server ros-kinetic-move-base
3 $ sudo apt-get install ros-kinetic-amcl ros-kinetic-dwa-local-planner
```

また、セミナー教材用にパラメータを調整してある地図生成や自律ナビゲーションのlaunchファイルが入ったパッケージをダウンロードします。

```
1 $ cd ~/catkin_ws/src/
2 $ git clone https://github.com/BND-tc/rsj_seminar_navigation.git
```

ダウンロードしたrsj_seminar_navigationパッケージをcatkin_makeでビルドします。

```
1 $ cd ~/catkin_ws/
2 $ catkin_make
```

地図生成

まず、地図ベースの自律ナビゲーションを実現するためにslam_gmappingパッケージを用いて地図を作成します。

PCにロボットとURGのUSBケーブルを接続し、地図を生成したい場所にロボットを置いて、下記のコマンドを実行します。ただし利用するセンサに応じてコマンドが異なりますのでご注意ください。

URG-04LX-UG01の場合

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ roslaunch rsj_seminar_navigation mapping.launch \
4   robot_param:=~/home/【ユーザ名】/params/rsj-seminar20???.param 【該当するものに置き換えること】
```

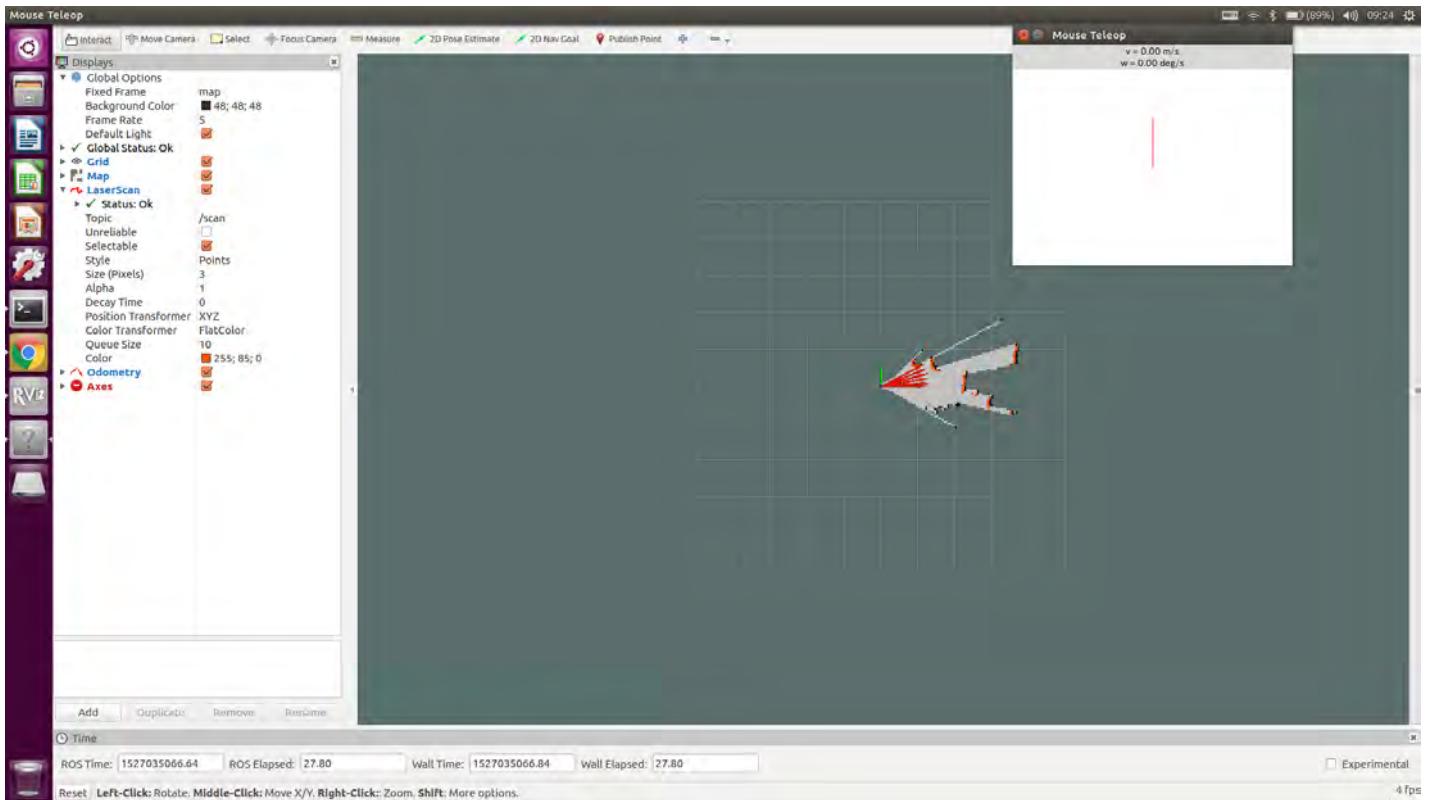
YVT-35LXの場合

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ roslaunch rsj_seminar_navigation 3durg_mapping.launch \
4   robot_param:=~/home/【ユーザ名】/params/rsj-seminar20???.param 【該当するものに置き換えること】
```

Xtion PRO Liveの場合

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ roslaunch rsj_seminar_navigation xtion_mapping.launch \
4   robot_param:=~/home/【ユーザ名】/params/rsj-seminar20???.param 【該当するものに置き換えること】
```

RViz が起動し、下記のように URG もしくは Xtion の複数のスキャンデータをつなげて、大きな占有格子地図を生成し始めます。

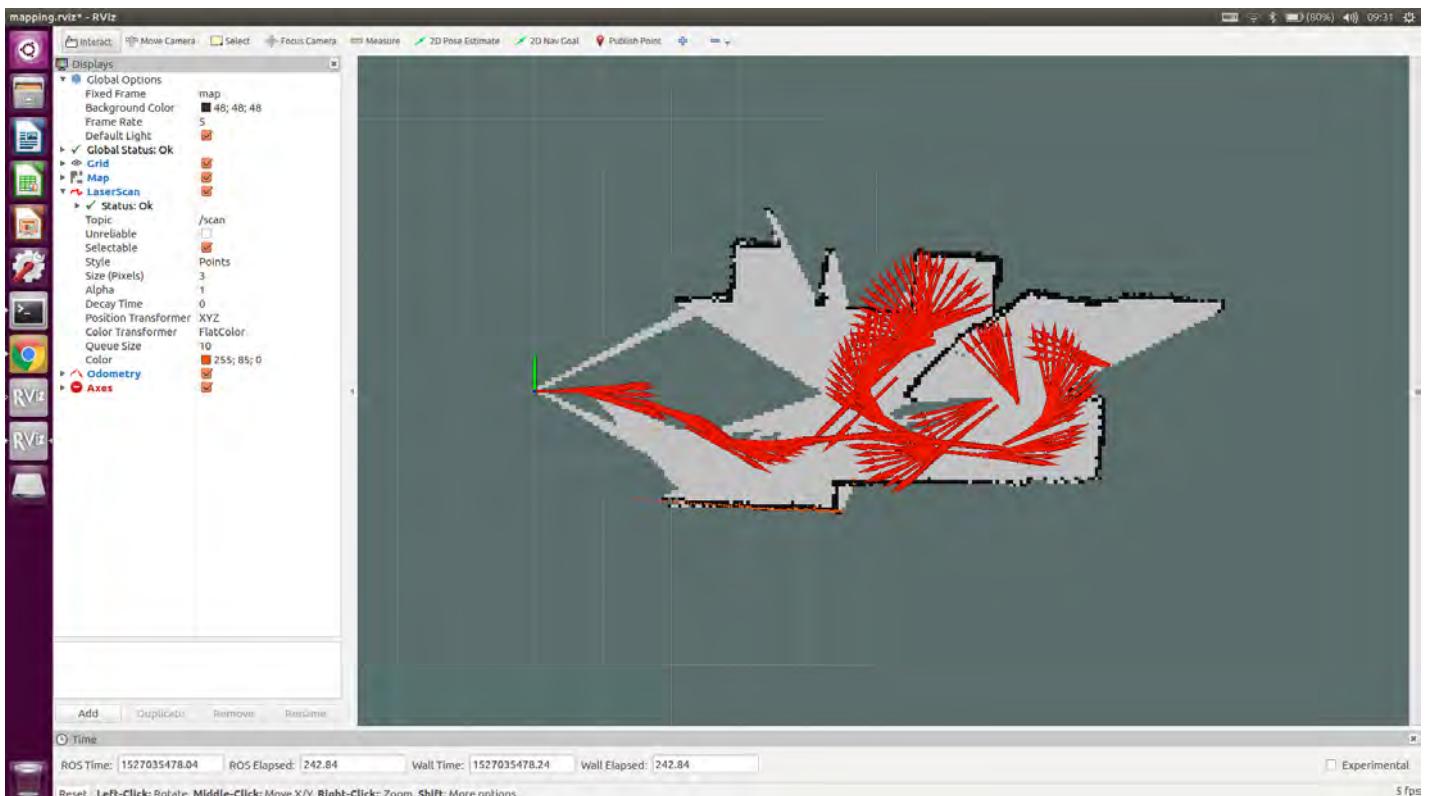


「Mouse Teleop」のウインドウ内をマウスでドラッグ操作することでロボットの動作を制御できますので、地図を作りたい範囲を移動させてみましょう。「Mouse Teleop」のウインドウが隠れている場合は、左の「?」アイコンをクリックすることで最前面に表示できます。

この地図生成は、URG や Xtion のスキャンデータを逐次つなげていく仕組みのため、ロボットの位置姿勢を大きく変化させた場合やセンサに見えているものが少ない場合には、地図が破綻する場合があります。

その際は、`roslaunch`を実行した端末で、**Ctrl+c** を押して終了し、もう一度実行しなおします。

ロボットを走行させていくと、図のように走行させた範囲の地図が表示されます。



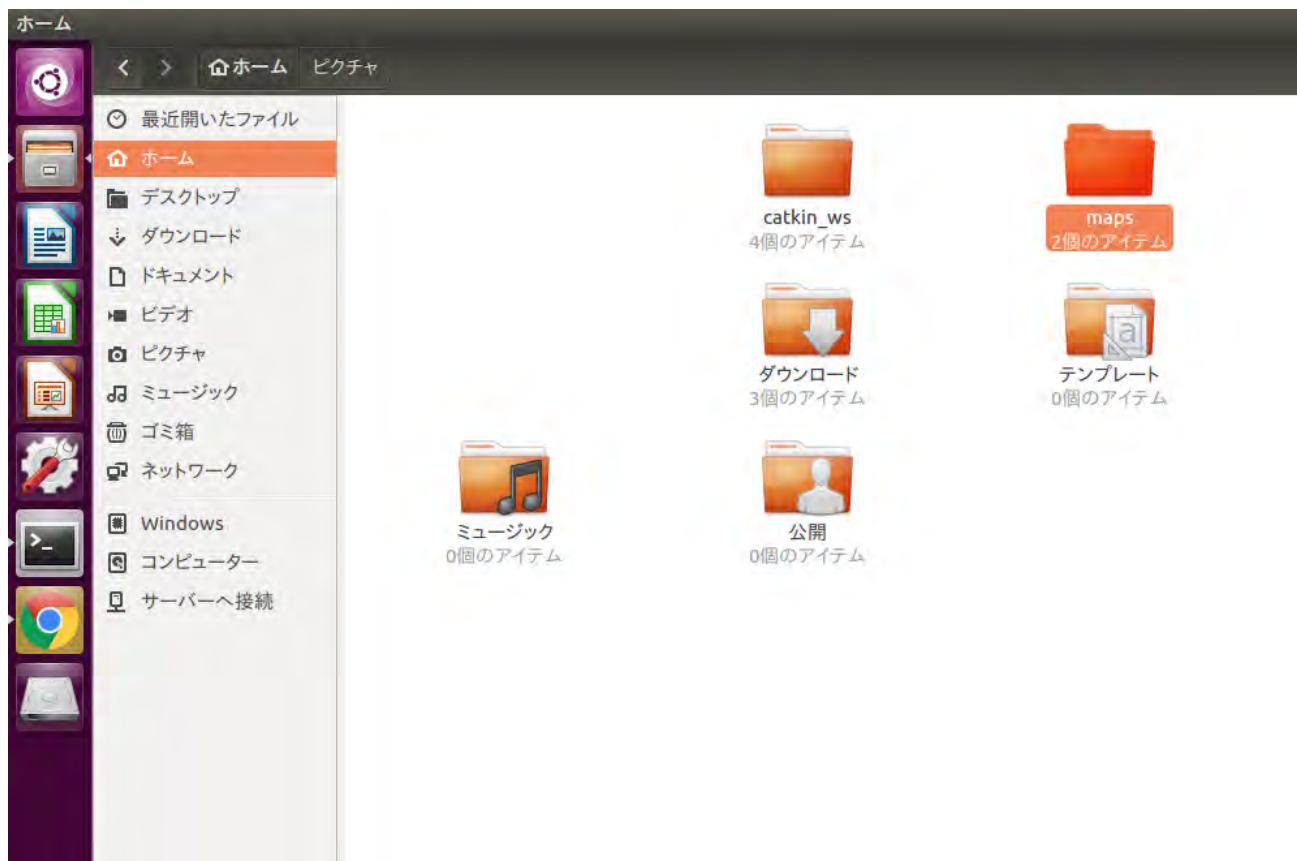
必要な範囲の地図ができあがったら端末をもう一つ開き、下記のコマンドで保存するディレクトリを作成し、地図データをファイルに出力します。

```

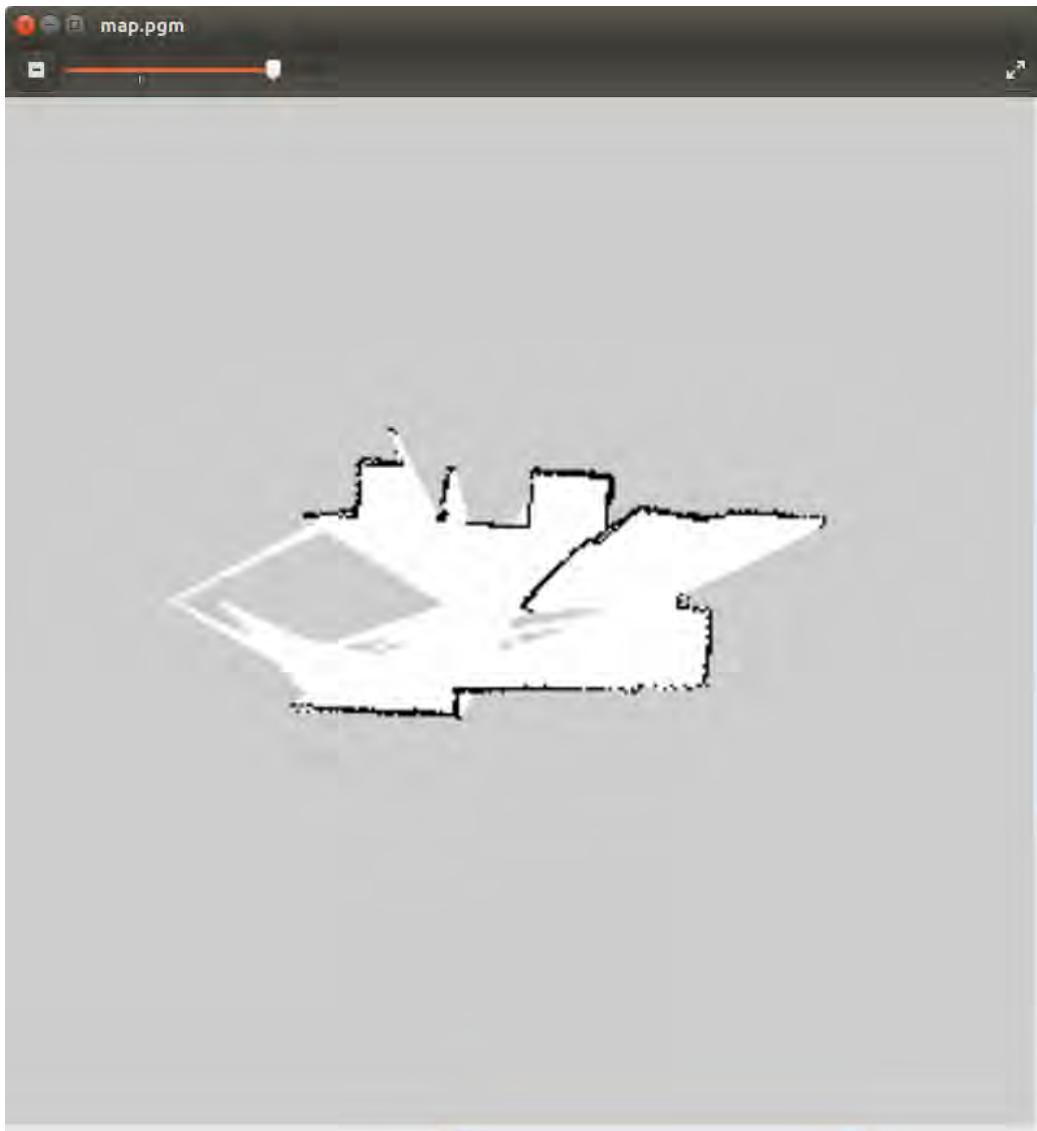
1 $ mkdir ~/maps
2 $ cd ~/maps/
3 $ rosrun map_server map_saver

```

画面左の「ファイル」アイコンから、ホームディレクトリ下の、「maps」ディレクトリを見ると、map.yamlと、map.pgmファイルが生成されていることが確認できます。



map.yamlをダブルクリックして開くと、地図の解像度や原点の情報が書かれています。また、map.pgmを開くと地図データが画像ファイルとして保存されていることがわかります。



地図の画像ファイルが正しく出力できていることを確認したら、rosLaunchを実行した端末のウインドウを選択して、**Ctrl+d** を押して終了します。

ナビゲーション

先ほど作成した地図を用いて、自律ナビゲーションを試してみましょう。

PCにロボットとURGやXtionのUSBケーブルを接続し、地図を作成した際の初期位置・姿勢と同じようにロボットを置いて、下記のコマンドを実行します。

URG-04LX-UG01の場合

```

1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun rsj_seminar_navigation navigation.launch \
4   robot_param:=~/home/【ユーザ名】/params/rsj-seminar20???.param該当するものに置き換えること

```

YVT-35LXの場合

```

1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun rsj_seminar_navigation 3durg_navigation.launch \
4   robot_param:=~/home/【ユーザ名】/params/rsj-seminar20???.param該当するものに置き換えること

```

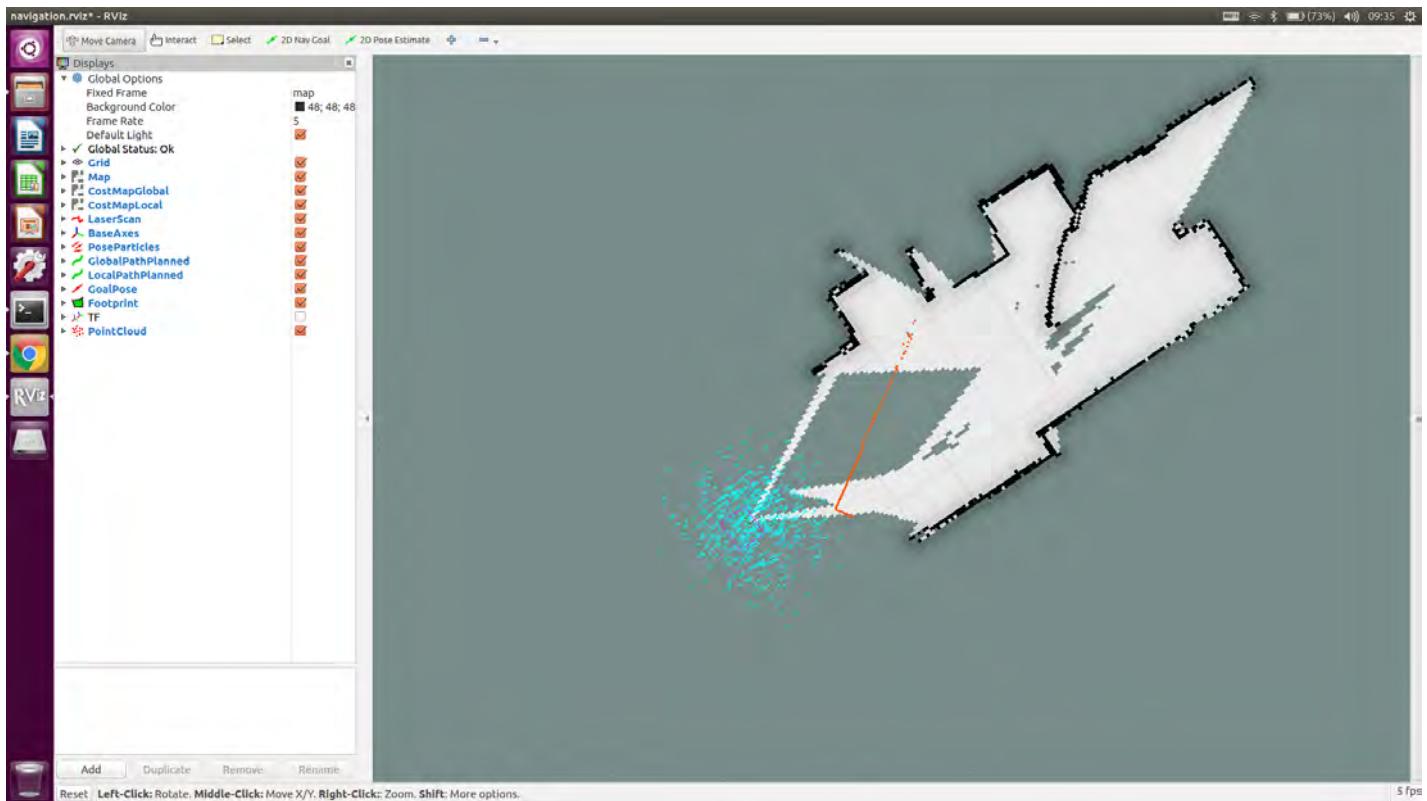
Xtion PRO Liveの場合

```

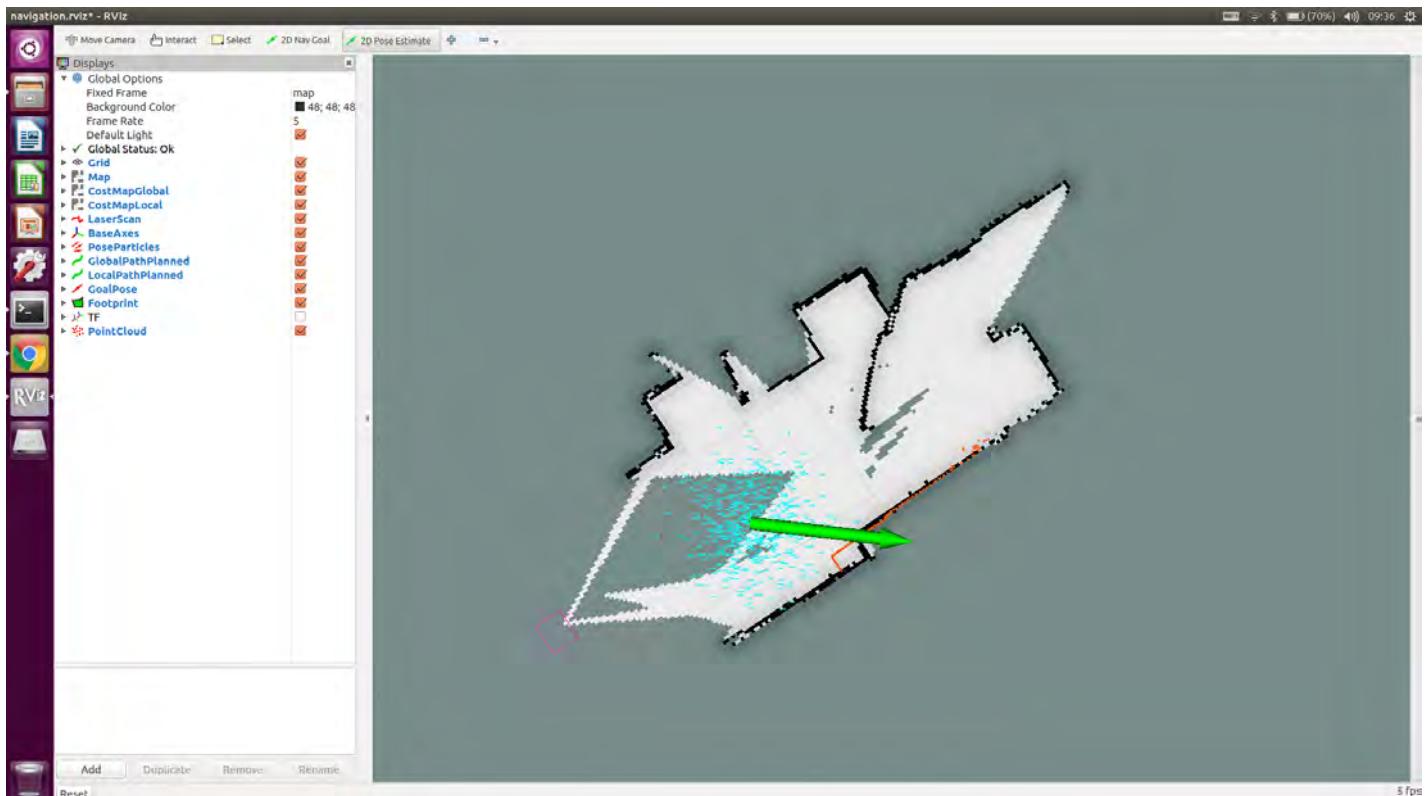
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun rsj_seminar_navigation xtion_navigation.launch \
4   robot_param:=~/home/【ユーザ名】/params/rsj-seminar20???.param該当するものに置き換えること

```

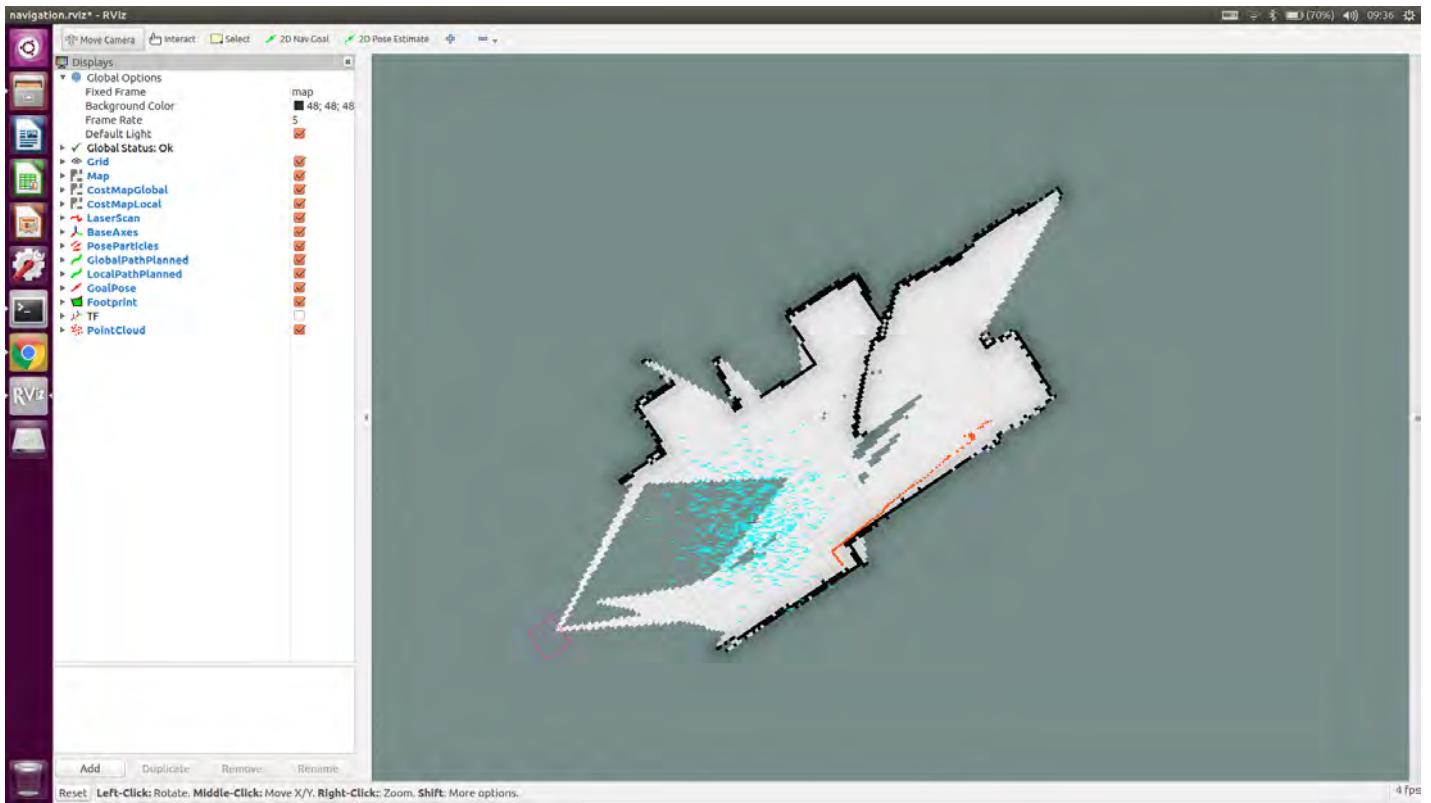
RViz が起動し、下記のように先ほど作成した地図が表示されます。この例では、地図と、オレンジ色でプロットされている URG や Xtion の現在のデータがずれており、自己位置がずれていることがわかります。また、水色のたくさんの矢印は、推定している自己位置の候補を表しています。



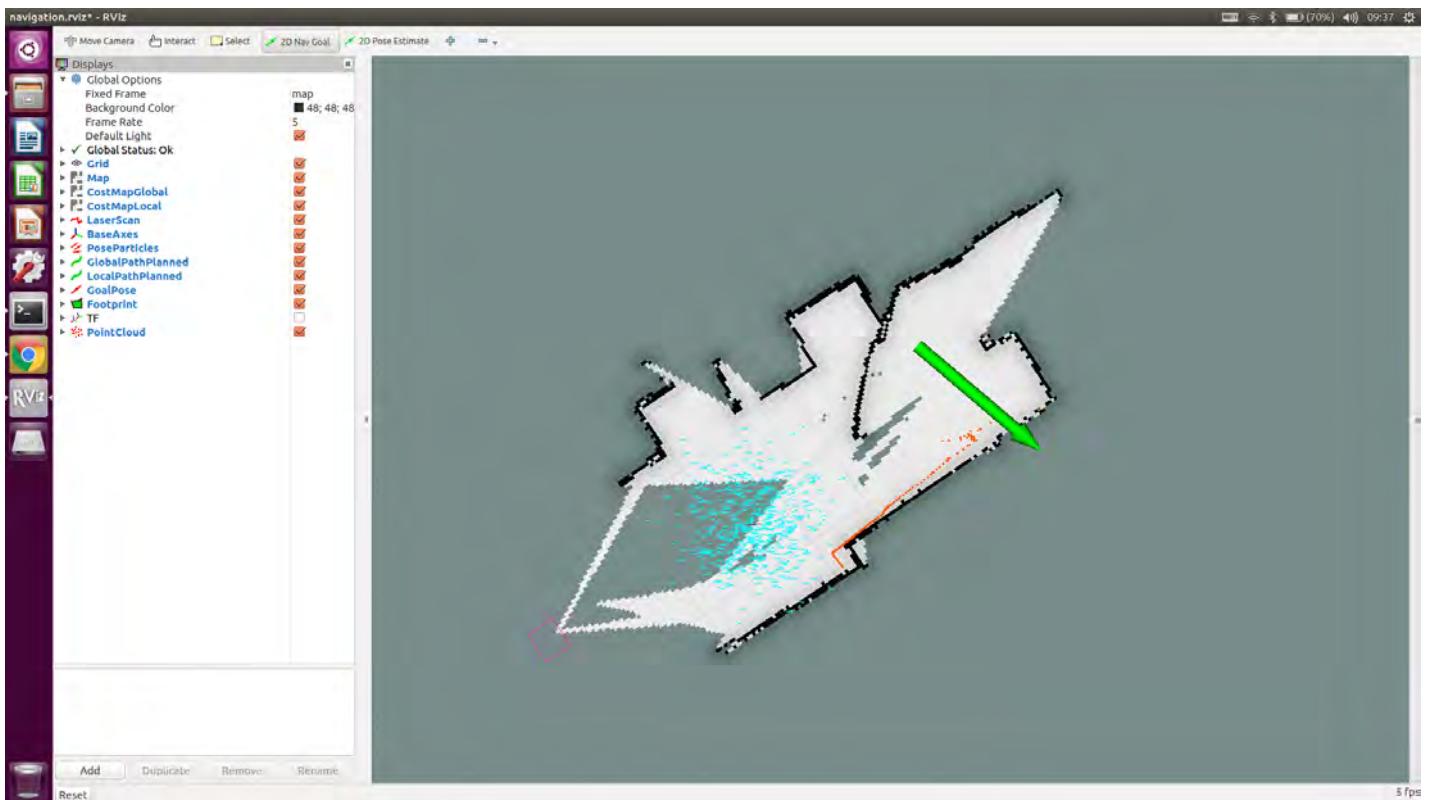
RViz のウインドウ上中央にある、「2D Pose Estimate」ボタンを押し、ロボットの本来の位置からロボットの向いている方向に向かってドラッグすると、緑色の矢印が現れ自己位置推定ノードに位置姿勢の修正を指示することができます。



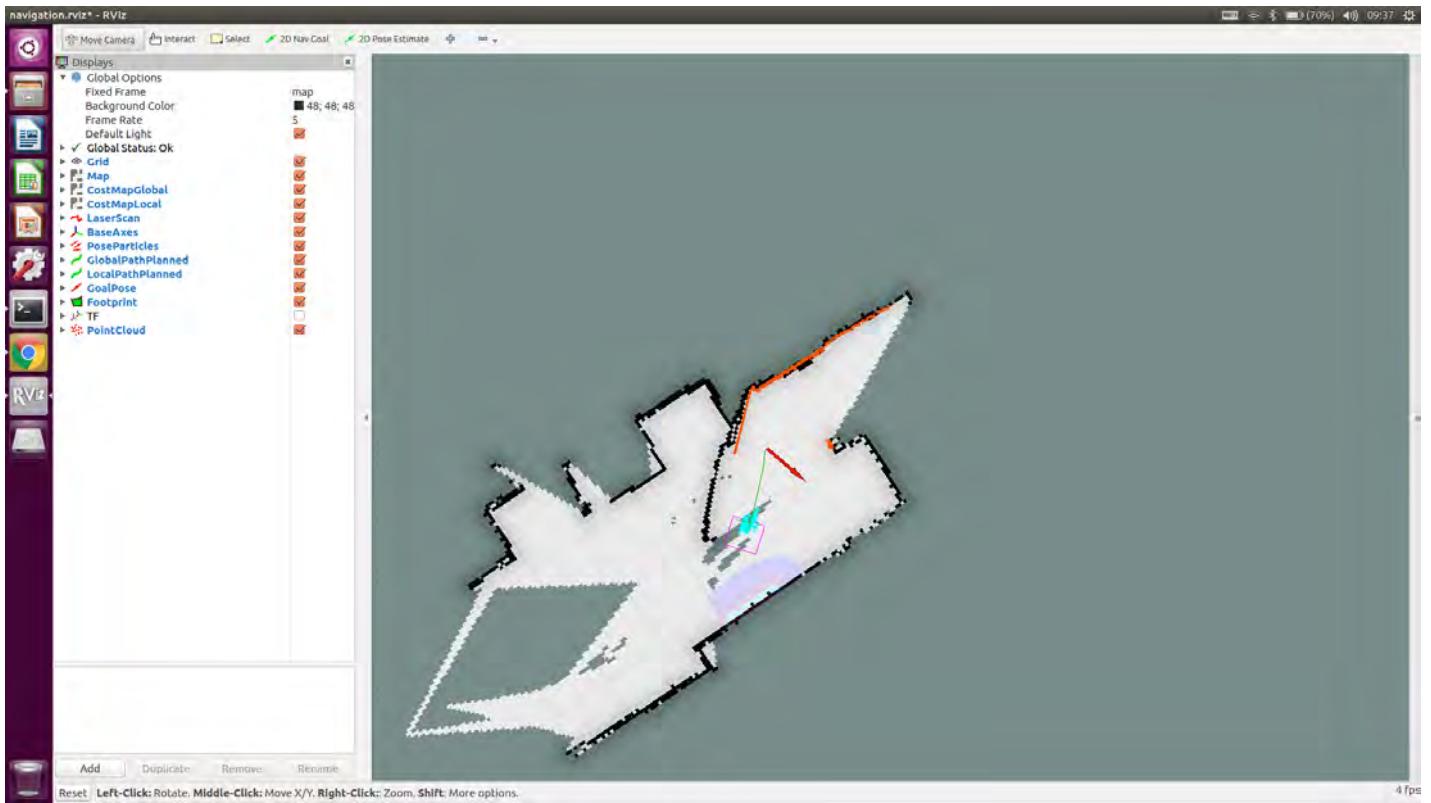
地図と、URG や Xtion のデータが概ね一致しました。



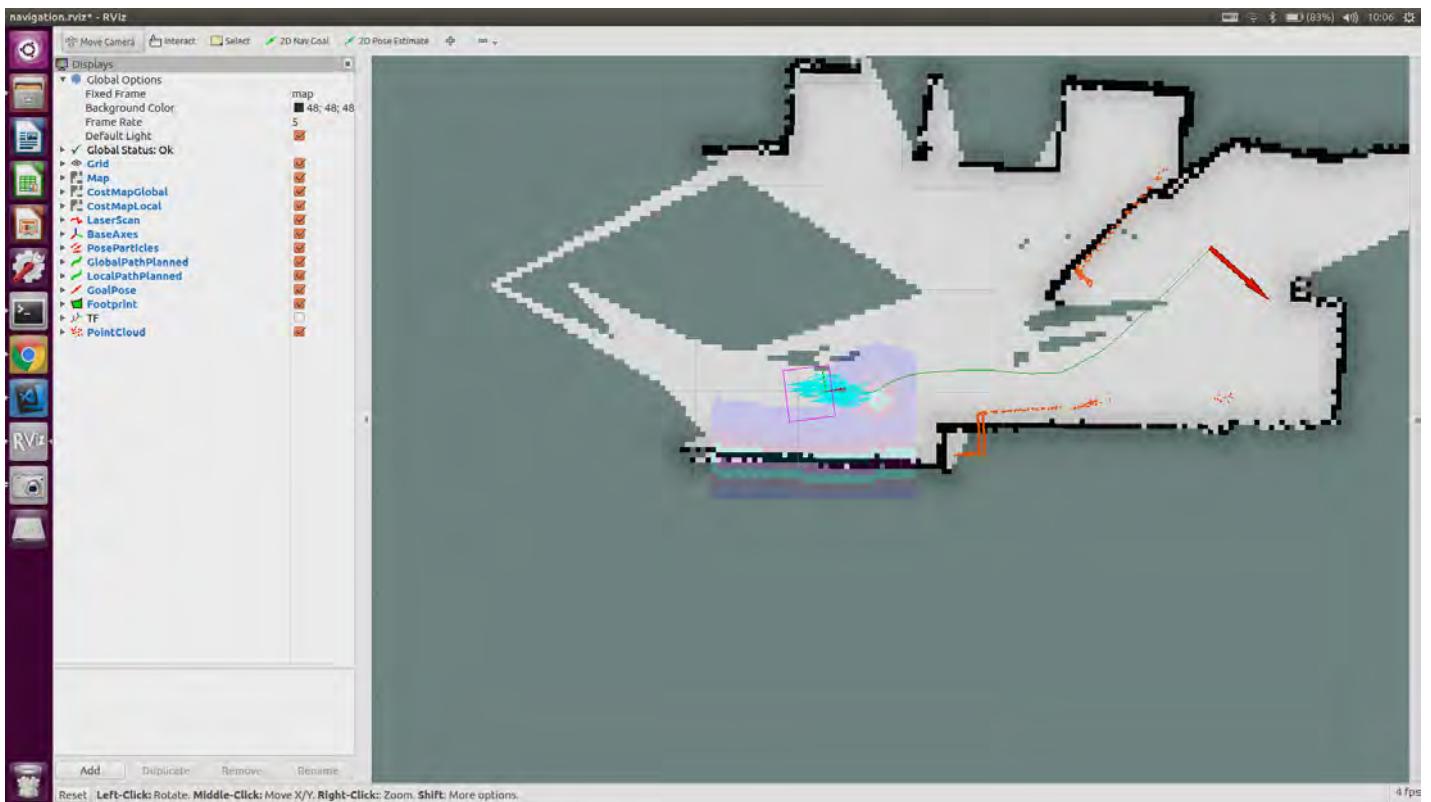
次に、ウインドウ上中央の「2D Nav Goal」ボタンでナビゲーションのゴールを指定します。同様に、与えたいゴールの位置から目標姿勢の方向に向かってドラッグすると、緑色の矢印が現れナビゲーションのノードに目標位置姿勢の指示を与えることができます。

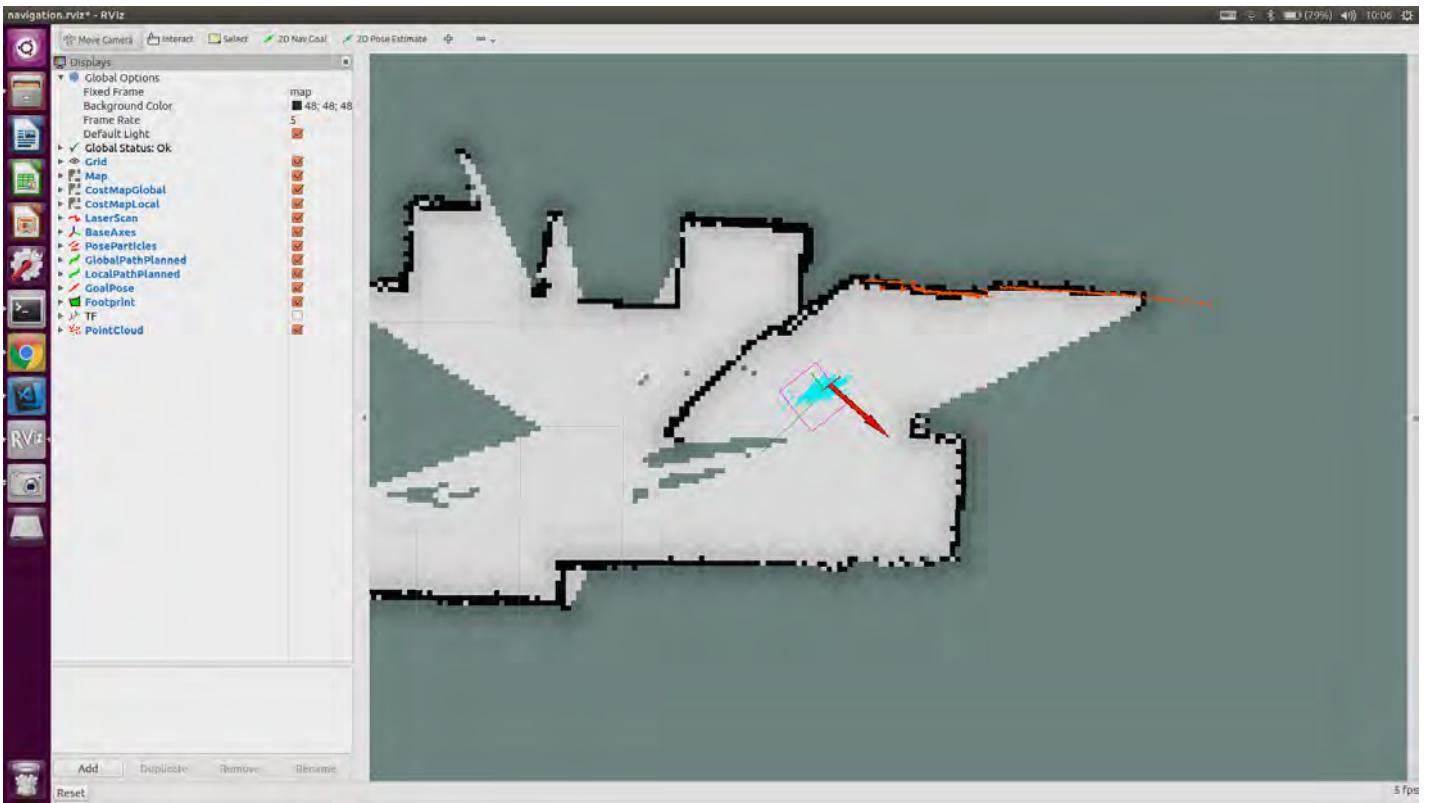


すると、緑色の線でグローバルプランナーが生成したパス、赤色の線でローカルプランナーが生成したパスが表示され、ロボットが走行を開始します。

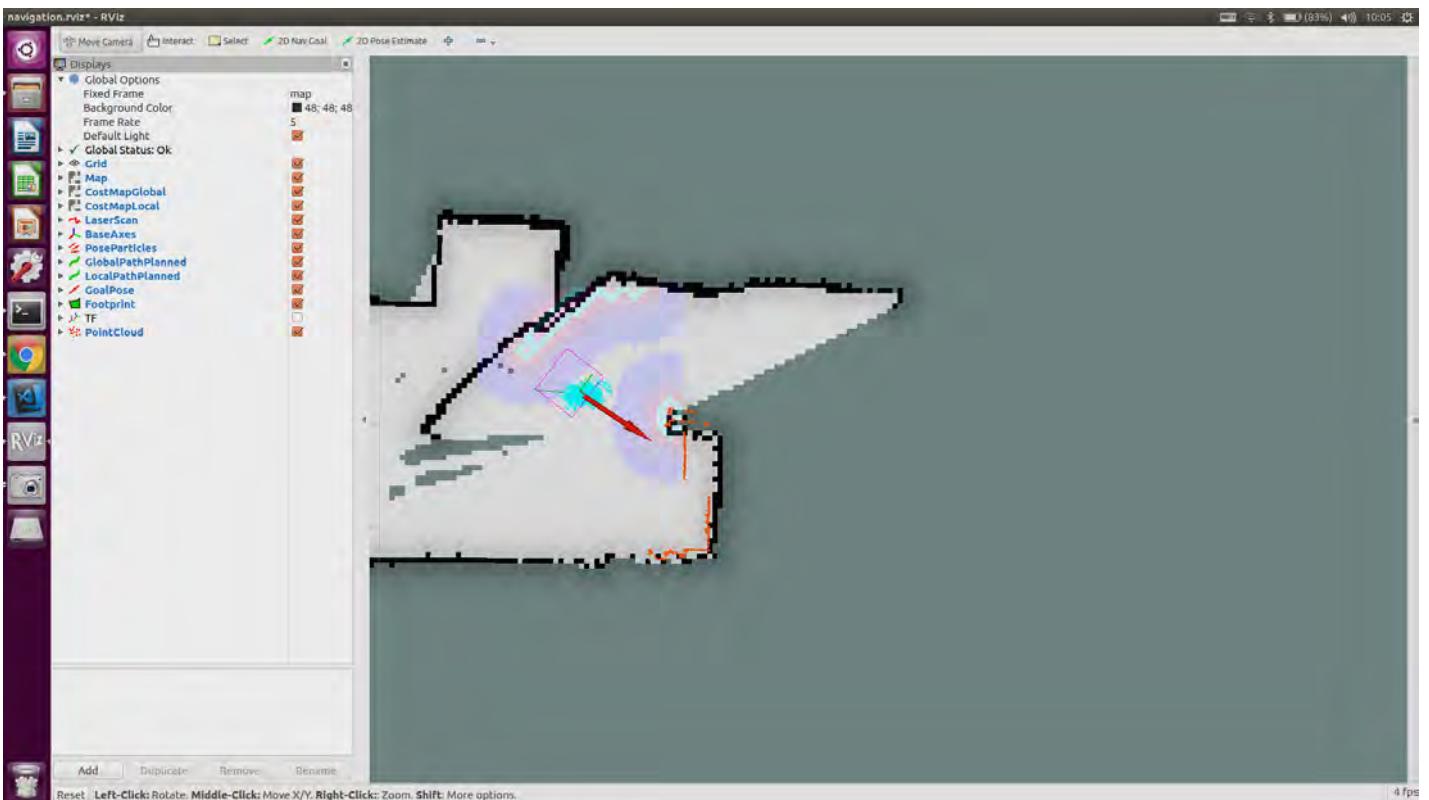


走行していくと、水色の矢印で表された自己位置の候補の分布が小さくなり、自己位置推定の確度が高くなつたことが確認できます。





ゴールに到着し、姿勢を目標の方向に向けると動作が終了します。

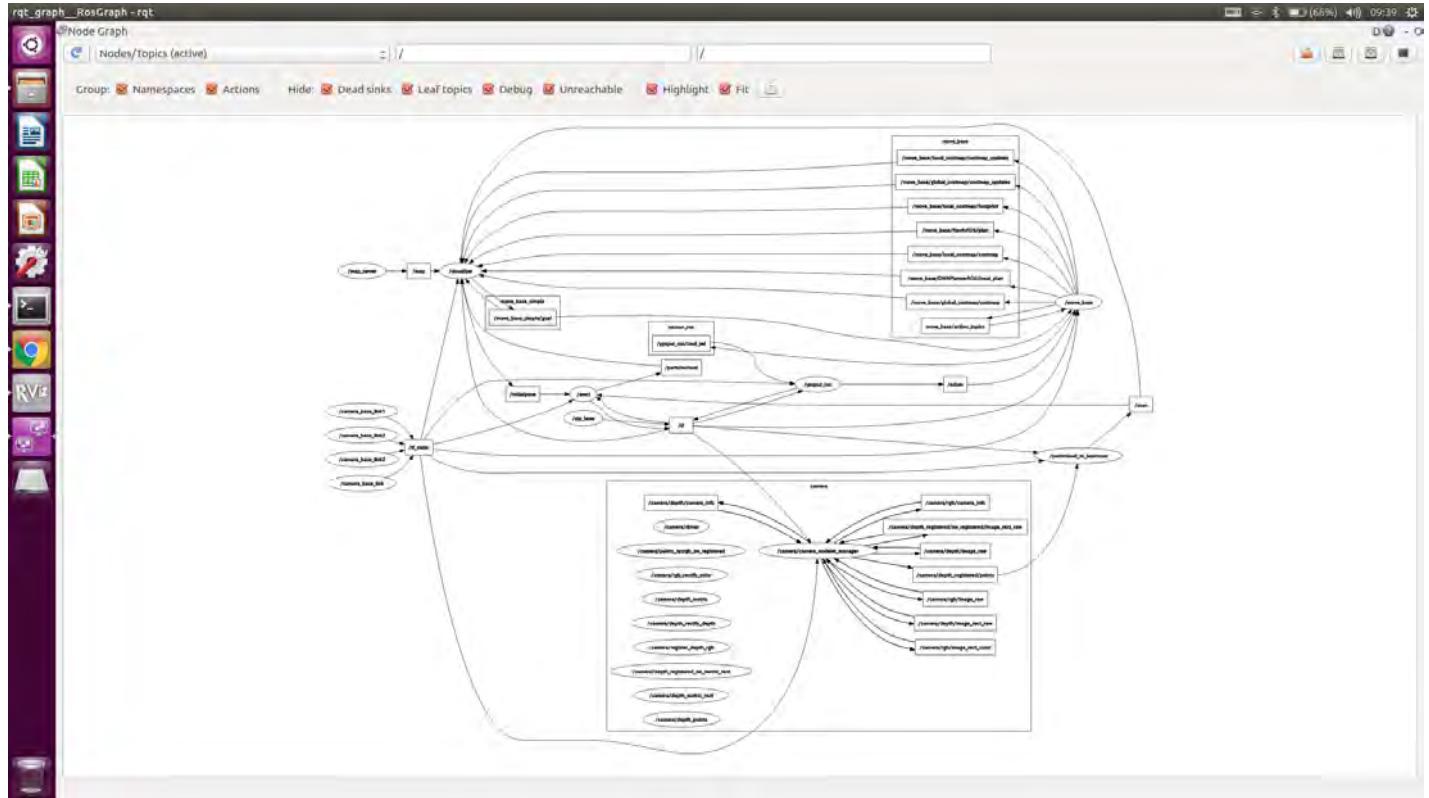


この状態で、どのようなノードが立ち上がっているのか確認してみましょう。新しい端末を開き、`rqt_graph`を実行します。

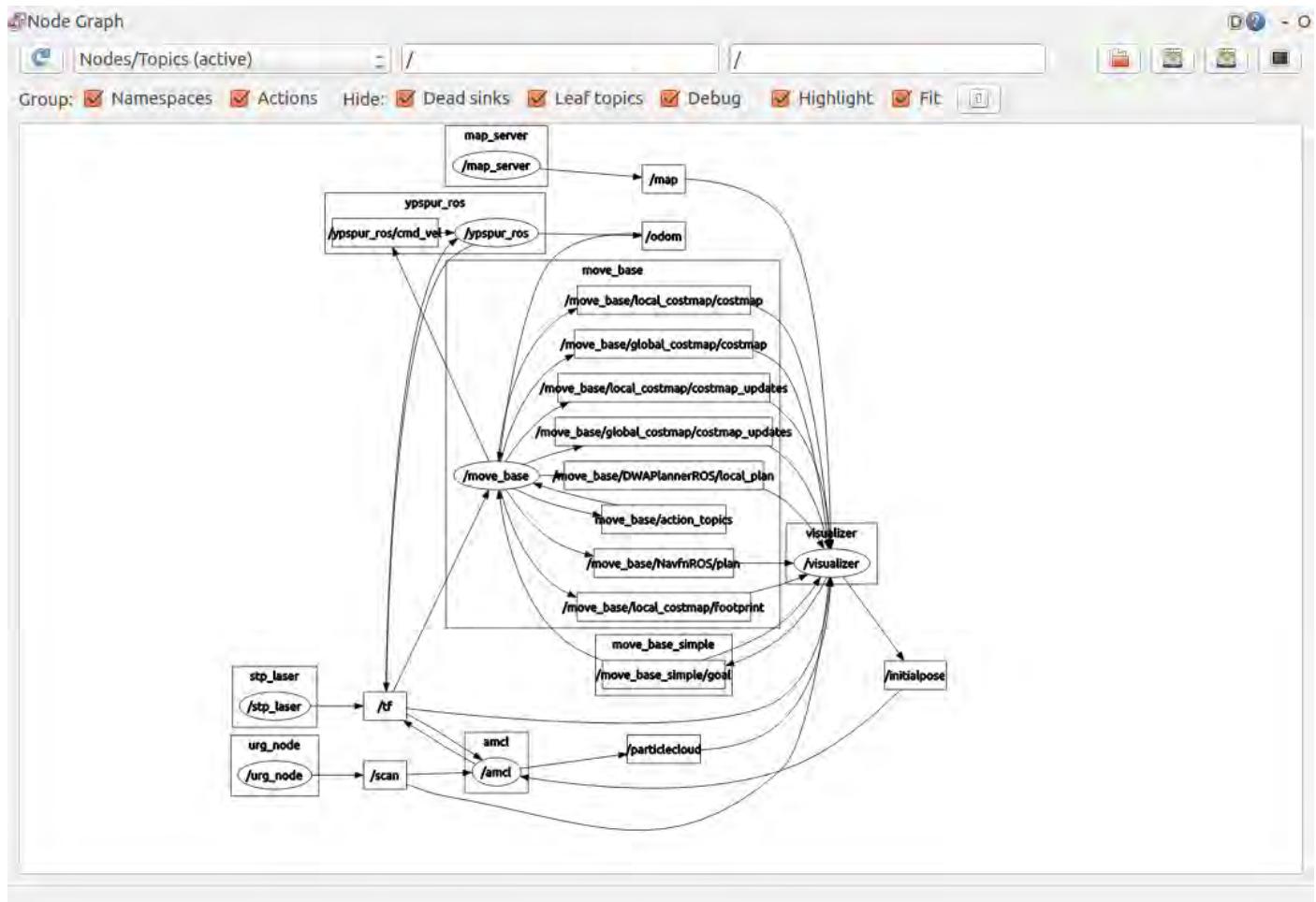
```
1 $ rqt_graph
```

左上の「Nodes only」の部分で「Nodes/Topics (active)」を選択します。

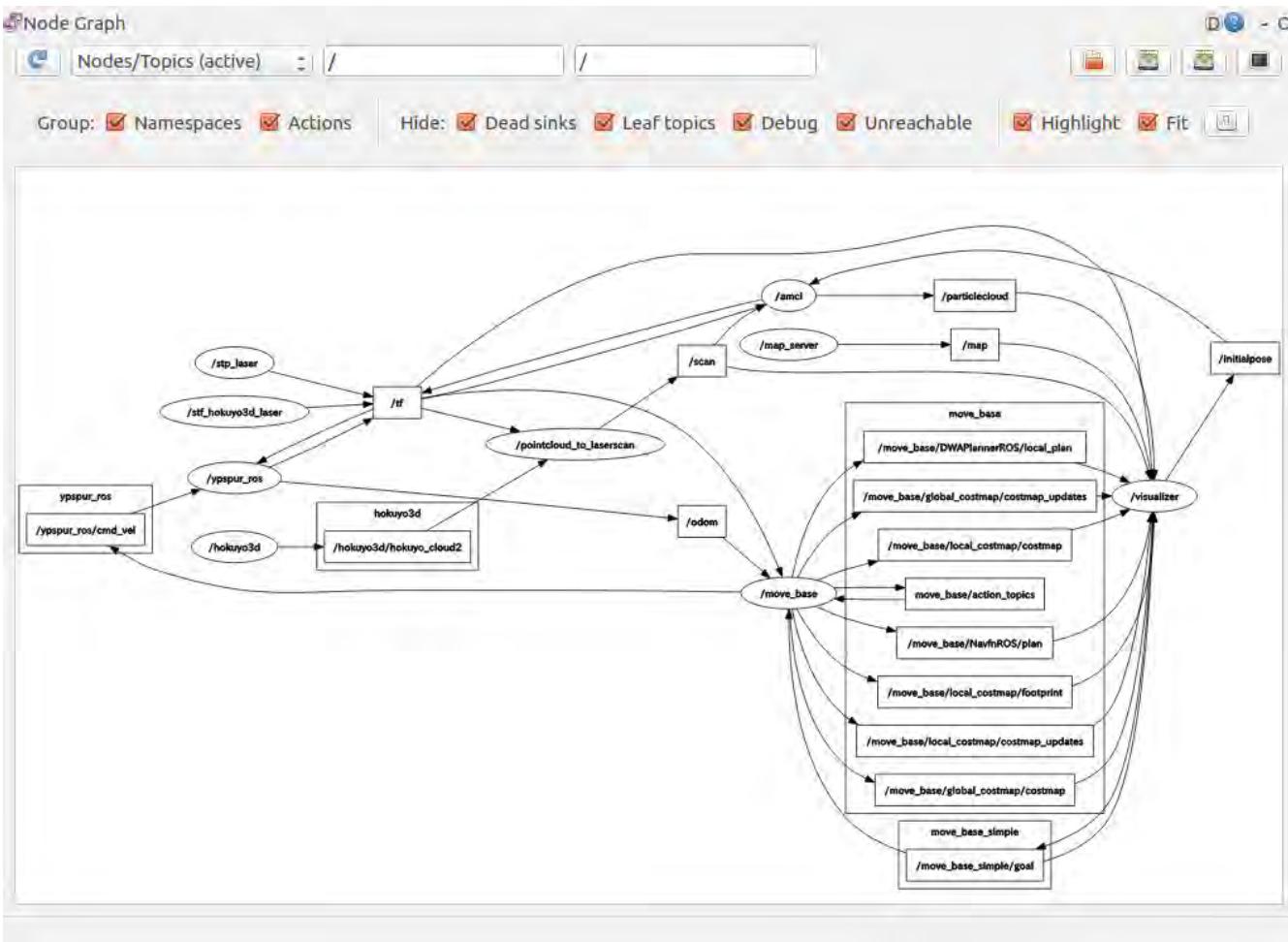
Xtion を使っていた場合は次のような図が表示されます。



URG を使っていた場合は次のような図が表示されます。



3DURG を使っていた場合は次のような図が表示されます。



四角枠で書かれた名前はトピック、もしくは複数のノードをグループ化したもの（nodelet）を表しています。丸枠で書かれた名前はノードを表しており、下記の仕事をしています。

```

ypspur_ros          ロボットの制御
urg_node            URG データの取り込み
"/camera"で始まるノード群
  Xtion データの取り込み
pointcloud_to_laserscan   3次元点群を2次元データに変換する
map_server           地図ファイルの読み取り
amcl                自己位置推定(モンテカルロローカライゼーション、いわゆる、パーティクルフィルター)
move_base            ナビゲーション(プラグインで、ダイクストラ法のグローバルプランナーと、ダイナミックウインドウアプローチのローカルプランナー、2Dコストマップの処理を実行)
visualizer          データの可視化( RViz )
stp_laser           ロボットの座標原点とURGの座標原点の座標変換の定義

```

rsj_seminar_navigationパッケージのlaunchファイルの中身を開き、どのようなノードが実行されているのか講義の内容と照らし合わせて確認してみましょう。

3次元点群の処理

Xtion PRO Live や YVT-35LX から得られる3次元点群PointCloudに対する基本的な処理を実習します。

Point Cloud の表示

各センサから得られる点群を RViz によって可視化します。センサごとに実行するコマンドが異なりますので、お手持ちのセンサに応じて次のコマンドを実行してください。

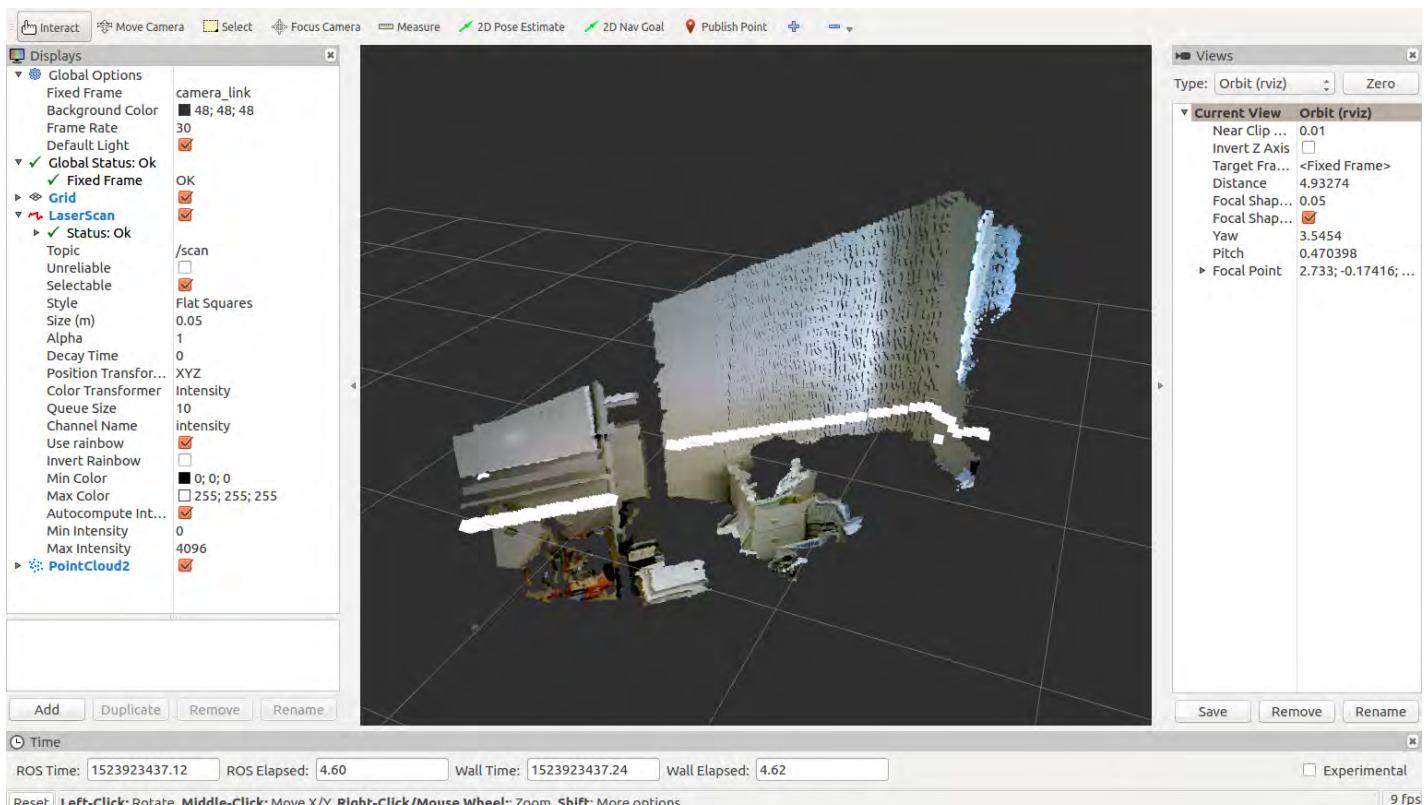
Xtion PRO Live の場合

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ roslaunch rsj_pointcloud_to_laserscan rsj_pointcloud_to_laserscan.launch
```

別のターミナルを開き

```
1 $ cd ~/catkin_ws/src/rsj_pointcloud_to_laserscan/config/rviz
2 $ rosrun rviz rviz -d view_points.rviz
```

図のようにPointCloudが表示されれば成功です。



起動した2つのターミナルを **Ctrl+c** で終了してください。

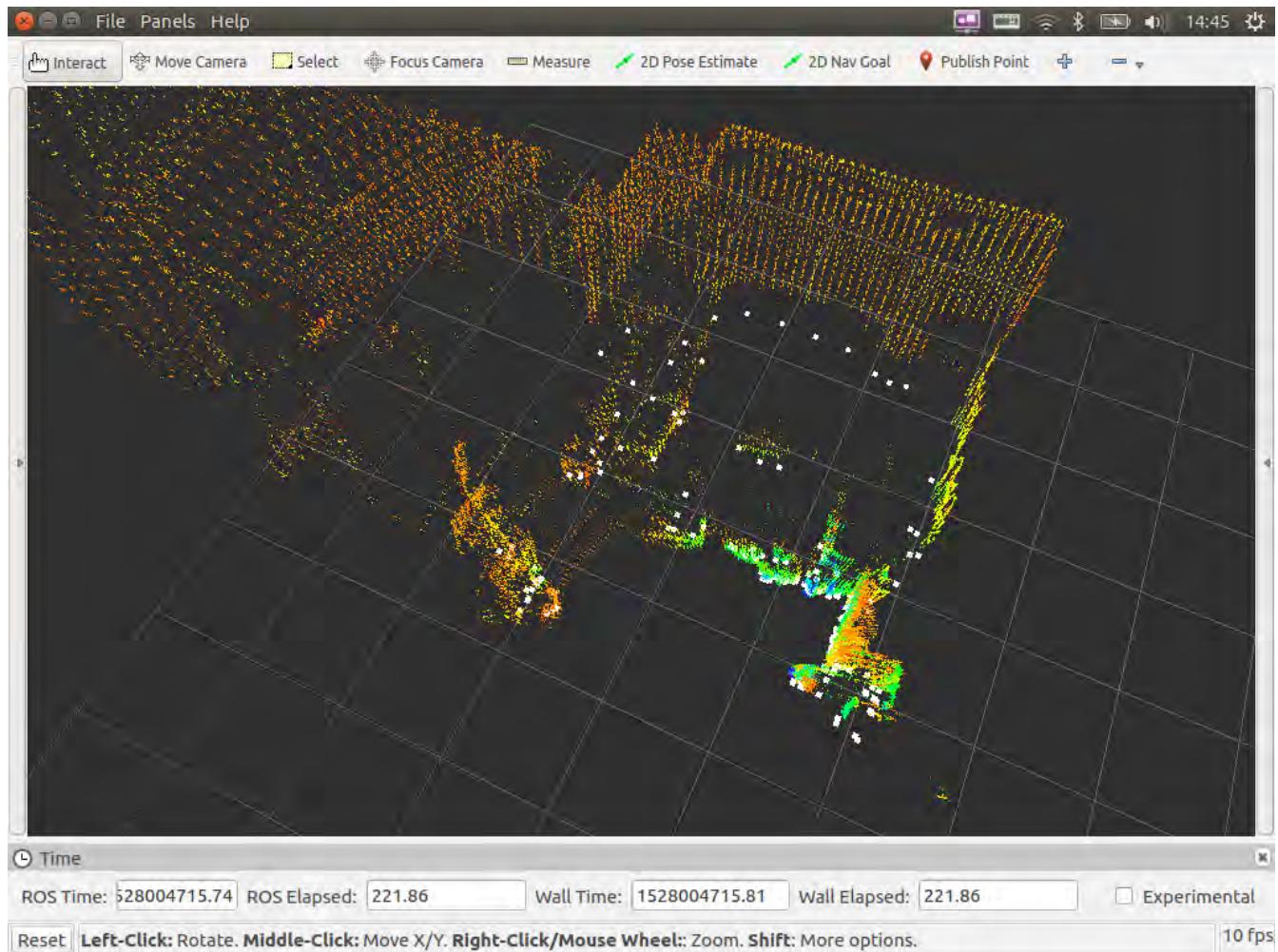
YVT-35LX の場合

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ roslaunch rsj_pointcloud_to_laserscan rsj_pointcloud_to_laserscan_3durg.launch
```

別のターミナルを開き

```
1 $ cd ~/catkin_ws/src/rsj_pointcloud_to_laserscan/config/rviz
2 $ rosrun rviz rviz -d view_points_3durg.rviz
```

図のようにPointCloudが表示されれば成功です。



起動した2つのターミナルを **Ctrl+c** で終了してください。

PCL (Point Cloud Library) による3次元点群処理

ロボットでオドメトリのデータを利用したときと同じように3次元センサが出力したPointCloudのデータを受け取るプログラムを作成しましょう。PointCloudの処理を独自に書こうとすると大変な労力が必要です。2次元画像処理用にOpenCVというライブラリがあるように、3次元点群処理にはPCL(Point Cloud Library)があります。PCLを利用するプログラムを作成してみましょう。

端末を開き雛形をダウンロードしてください。

```
1 $ cd ~/catkin_ws/src
2 $ git clone https://github.com/KMiyawaki/rsj_pointcloud_test.git
```

テキストエディタでrsj_pointcloud_test_node.cppを開いてください。

```
1 $ cd ~/catkin_ws/src/rsj_pointcloud_test/src
2 任意のテキストエディタで rsj_pointcloud_test_node.cpp を開く
```

RsjPointCloudTestNodeクラスにある、PointCloud用のコールバック関数cbPointsを編集します。

```
1 void cbPoints(const PointCloud::ConstPtr &msg)
2 {
3     try
4     {
5         (略)
6         // ここに cloud_src に対するフィルタ処理を書く
7         ROS_INFO("width: %u, height: %u", cloud_src->width, cloud_src->height);
8     }
9     catch (std::exception &e)
10    {
11        ROS_ERROR("%s", e.what());
12    }
13 }
```

cbPoints関数はセンサからPointCloudを受け取るたびに呼び出される関数です。

なお、今回の実習で利用している三次元センサのドライバノードから出力される点群データは、元々はROSのsensor_msgs/PointCloud2という型ですが、後述するpcl_rosというライブラリがPCLのPointCloud型にsensor_msgs/PointCloud2型と互換性を持たせる実装を追加することで、特別にそのままROS Message型と同様に使えるようにしています。サブスクリーブの初期化コードで指定するコールバック関数が任意の型を受け取れるわけではないことに注意してください。

ファイルを保存してエディタを閉じます。

ビルド&実行

まず、catkin_wsでcatkin_makeを実行して、追加したコードをビルドします。

```
1 $ cd ~/catkin_ws
2 $ catkin_make
```

次にお手持ちの3次元センサごとに次のようにノードを起動します。

Xtion PRO Live の場合

ターミナルでセンサドライバノードを起動します。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ roslaunch rsj_pointcloud_to_laserscan rsj_pointcloud_to_laserscan.launch
```

新しいターミナルを開き、rsj_pointcloud_test_nodeを起動します。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun rsj_pointcloud_test rsj_pointcloud_test_node \
4   _target_frame:=camera_link _topic_name:=/camera/depth_registered/points
5 [ INFO] [1524039160.481736901]: target_frame='camera_link'
6 [ INFO] [1524039160.481783905]: topic_name='/camera/depth_registered/points'
7 [ INFO] [1524039160.485222004]: Hello Point Cloud!
8 [ INFO] [1524039161.311438819]: width: 640, height: 480
```

YVT-35LX の場合

ターミナルでセンサドライバノードを起動します。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ roslaunch rsj_pointcloud_to_laserscan rsj_pointcloud_to_laserscan_3durg.launch
```

新しいターミナルを開き、rsj_pointcloud_test_nodeを起動します。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun rsj_pointcloud_test rsj_pointcloud_test_node \
4   _target_frame:= _topic_name:=/hokuyo3d/hokuyo_cloud2
5 [ INFO] [1528006896.162315502]: target_frame=''
6 [ INFO] [1528006896.162660037]: topic_name='/hokuyo3d/hokuyo_cloud2'
7 [ INFO] [1528006896.178381795]: Hello Point Cloud!
8 [ INFO] [1528006896.378943557]: width: 2228, height: 1
9 [ INFO] [1528006896.412205921]: width: 2670, height: 1
10 [ INFO] [1528006896.478866703]: width: 2674, height: 1
```

このようにwidth: xxx, height: xxxというメッセージが表示されればPointCloudは受信できています。

補足 rsj_pointcloud_test_node.cppについて

プログラムの先頭にはROS内でPCLを扱うためのヘッダファイルに関するinclude文と実習で使うPointCloudの型宣言が記述されています。

```

1 #include <pcl_ros/point_cloud.h>
2 #include <pcl/point_types.h>
3 #include <visualization_msgs/MarkerArray.h>
4
5 typedef pcl::PointXYZ PointT
6 typedef pcl::PointCloud<PointT> Pointcloud;

```

PCL では `pcl::PointCloud<T>` という C++ のテンプレートで点群を扱うデータ型を表現しています。Tの部分には座標と色情報を持つ `pcl::PointXYZRGB` など様々な点の型を与えることが可能です。今回は色情報のない、位置だけの点 `pcl::PointXYZ` を使います。最終行の `typedef` 声明では `pcl::PointCloud<PointT>` に対し今回の実習で利用する点群の型 `PointCloud` という別名をつけ、プログラムが書きやすくなるようにしています。

なお、`#include <visualization_msgs/MarkerArray.h>` は点群処理結果を可視化するために必要となる ROS に含まれるヘッダファイルです。

`RsjPointCloudTestNode` クラスの冒頭には、`sub_odom_` と同じように `PointCloud` 用のサブスクリーバを宣言しています。

```

1 class RsjPointCloudTestNode
2 {
3     private:
4         (略)
5         ros::Subscriber sub_points_;

```

`RsjPointCloudTestNode` のコンストラクタには、このノードが必要としているパラメータの取得や `PointCloud` 用のサブスクリーバ初期化コードが記述されています。

```

1 RsjPointCloudTestNode()
2     : nh_()
3     , pnh_("~")
4 {
5     std::string topic_name;
6     pnh_.param("target_frame", target_frame_, std::string(""));
7     pnh_.param("topic_name", topic_name, std::string("/camera/depth_registered/points"));
8     ROS_INFO("target_frame = '%s'", target_frame_.c_str());
9     ROS_INFO("topic_name = '%s'", topic_name.c_str());
10    sub_points_ = nh_.subscribe(topic_name, 5, &RsjPointCloudTestNode::cbPoints, this);

```

`pnh_.param` 関数は、実行時に与えられたパラメータを読み込みます。ここで `pnh_` は、`rsj_robot_test` のソースコードで解説した `nh_` と同様の `ros::NodeHandle` です。ただし `pnh_` は、"~"を引数に初期化することで、このノードのプライベートな名前空間を使う設定になっています。この1つ目の例では、`/rsj_pointcloud_test_node/target_frame` というパラメータを `target_frame_` 変数に読み込み、もし指定されていなければ `std::string("")` をデフォルト値として用いることを意味します。

`topic_name` はセンサが送出する `PointCloud` のトピック名を、`target_frame_` は得られた点群を処理しやすい座標系に変換する際の座標系の名前を示しています。特に Xtion PRO Live の場合、点群の座標系はロボットのローカル座標系と異なっているため、`cbPoints` 関数の冒頭で座標変換をしています。

この座標変換は ROS の `tf` によるものです。 `tf` によって座標系の情報がノード間で共有され、時刻と座標系の名前に基づいて様々なデータの座標変換を実現することができます。

`target_frame_` が空白の場合は座標変換を行いません (YVT-35LX の場合)。

`CMakeLists.txt` では PCL を ROS で扱えるようにしています。

```

1 find_package(catkin REQUIRED COMPONENTS
2   pcl_ros # 注: pcl_ros を追加している
3   roscpp
4   rospy
5   sensor_msgs
6   std_msgs
7   visualization_msgs
8 )

```

終了したら PCL を使った点群処理に関する実習に進んでください。

PointCloud に対するフィルタ

PointCloudに対するフィルタ

3次元点群に対して様々なフィルタを施し、移動ロボットの追跡対象など意味のある情報を抽出します。

PassThrough フィルタ

PassThroughフィルタは得られた点群のうち、一定の範囲内にある点群のみを抽出します。テキストエディタでrsj_pointcloud_test_node.cppを開いてください。

```
1 $ cd ~/catkin_ws/src/rsj_pointcloud_test/src
2 任意のテキストエディタで rsj_pointcloud_test_node.cpp を開く
```

プログラム冒頭にinclude文を追記してください。

```
1 #include <pcl/point_types.h>
2 #include <pcl/filters/passthrough.h> // 追記
3 #include <visualization_msgs/MarkerArray.h>
4
5 typedef pcl::PointXYZ PointT;
6 typedef pcl::PointCloud<PointT> Pointcloud;
```

RsjPointCloudTestNodeクラスの冒頭に、pcl::PassThroughフィルタのインスタンスを追加します。また、フィルタの結果を格納するためのPointCloud型変数cloud_passthrough_、および処理結果をpublishするためのパブリッシャpub_passthrough_を追加します。

```
1 class RsjPointCloudTestNode
2 {
3     private:
4         (略)
5         PointCloud::Ptr cloud_transformed_;
6         // 以下を追記
7         pcl::PassThrough<PointT> pass_;
8         PointCloud::Ptr cloud_passthrough_;
9         ros::Publisher pub_passthrough_;
```

RsjPointCloudTestNodeクラスのコンストラクタでPassThroughフィルタの設定、cloud_passthrough_およびpub_passthrough_を初期化します。

```
1 RsjPointCloudTestNode()
2     : nh_()
3     , pnh_("~")
4 {
5     (略)
6     cloud_transformed_.reset(new PointCloud());
7     // 以下を追記
8     pass_.setFilterFieldName("z"); // Z軸(高さ)の値でフィルタをかける
9     pass_.setFilterLimits(0.1, 1.0); // 0.1 ~ 1.0 m の間にある点群を抽出
10    cloud_passthrough_.reset(new PointCloud());
11    pub_passthrough_ = nh_.advertise<PointCloud>("passthrough", 1);
12 }
```

cbPoints関数を次のように変更します。

```
1 void cbPoints(const PointCloud::ConstPtr &msg)
2 {
3     try
4     {
5         (略)
6         // ここに cloud_src に対するフィルタ処理を書く
7         pass_. setInputCloud(cloud_src);
8         pass_.filter(*cloud_passthrough_);
9         pub_passthrough_.publish(cloud_passthrough_);
10        // ROS_INFO("width: %u, height: %u", cloud_src->width, cloud_src->height); // 削除
11        ROS_INFO("points (src: %zu, paththrough: %zu)", 
12                 cloud_src->size(), cloud_passthrough_->size()); // 追記
13    }
14    catch (std::exception &e)
15    {
16        ROS_ERROR("%s", e.what());
17    }
18 }
```

ビルド&実行

まず、catkin_wsでcatkin_makeを実行して、追加したコードをビルドします。

```
1 $ cd ~/catkin_ws
2 $ catkin_make
```

次にお手持ちの3次元センサごとに次のようにノードを起動します。

Xtion PRO Live の場合

ターミナルでセンサドライバノードを起動します。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun rsj_pointcloud_to_laserscan rsj_pointcloud_to_laserscan.launch
```

新しいターミナルを開き、rsj_pointcloud_test_nodeを起動します。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun rsj_pointcloud_test rsj_pointcloud_test_node \
4   _target_frame:=camera_link _topic_name:=/camera/depth_registered/points
5 [ INFO] [1524040063.315596383]: target_frame='camera_link'
6 [ INFO] [1524040063.315656650]: topic_name='/camera/depth_registered/points'
7 [ INFO] [1524040063.320448185]: Hello Point Cloud!
8 [ INFO] [1524040064.148595331]: points (src: 307200, paththrough: 34350)
```

YVT-35LX の場合

ターミナルでセンサドライバノードを起動します。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun rsj_pointcloud_to_laserscan rsj_pointcloud_to_laserscan_3durg.launch
```

新しいターミナルを開き、rsj_pointcloud_test_nodeを起動します。

```
1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun rsj_pointcloud_test rsj_pointcloud_test_node \
4   _target_frame:= _topic_name:=/hokuyo3d/hokuyo_cloud2
5 [ INFO] [1528008816.751100536]: points (src: 2674, paththrough: 1019)
```

実行した際にpoints (src: xxxx, paththrough: xxx)というメッセージが表示されれば成功です。src、paththroughに続けて表示されている値はセンサから得られた、もとのPointCloudにおける点の個数とPassThroughフィルタ実行後の点の個数を示しています。フィルタ実行後の点の個数がゼロの場合はpass_.setFilterLimits(0.1, 1.0);の引数を調節してみてください。

フィルタ実行結果の可視化

RVizでフィルタ実行後の点群の様子を可視化します。rsj_pointcloud_test_nodeを起動したまま、新しいターミナルを開き、RVizを起動します。

Xtion PRO Live の場合

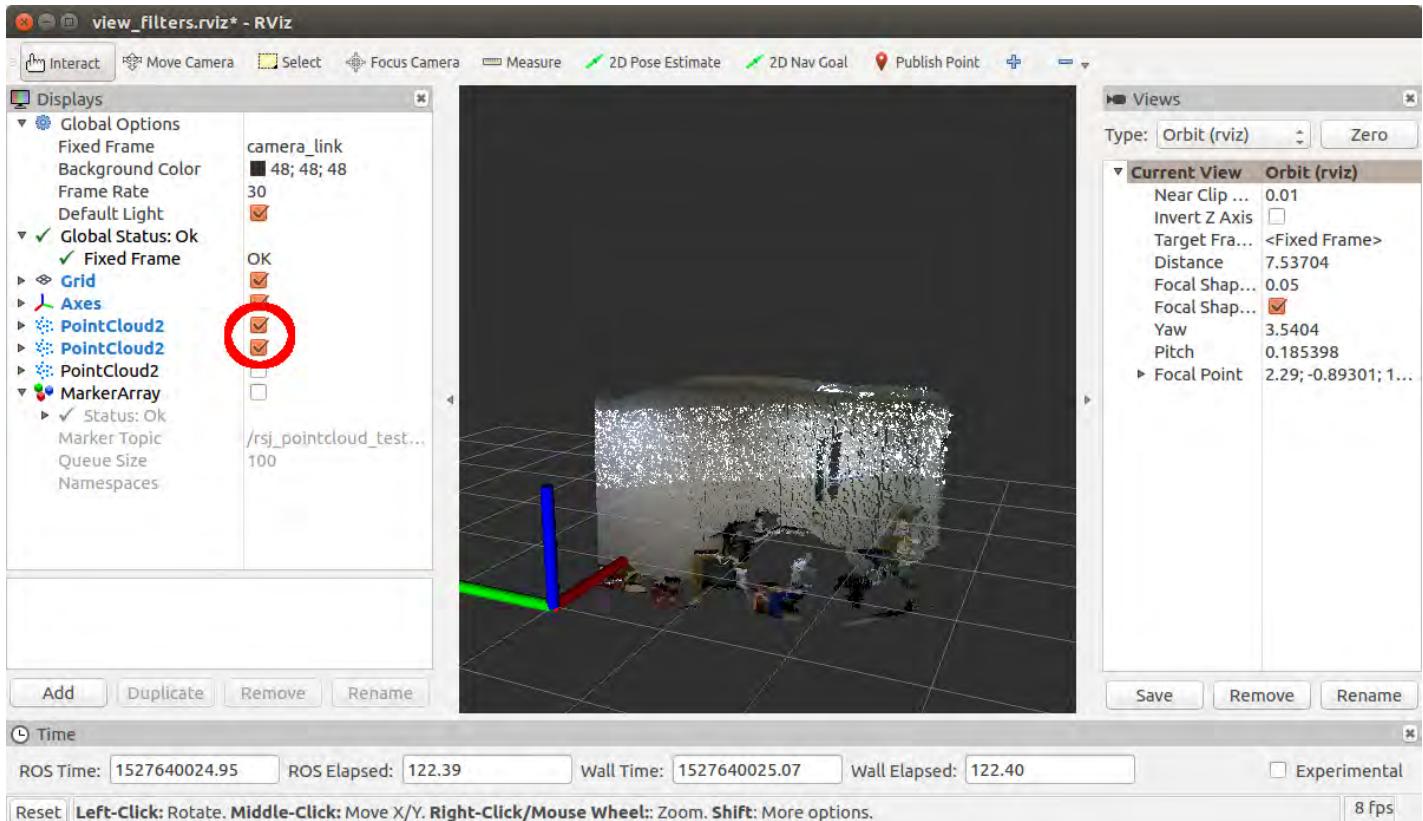
```
1 $ cd ~/catkin_ws/src/rsj_pointcloud_test/config/rviz
2 $ rviz -d view_filters.rviz
```

YVT-35LX の場合 の場合

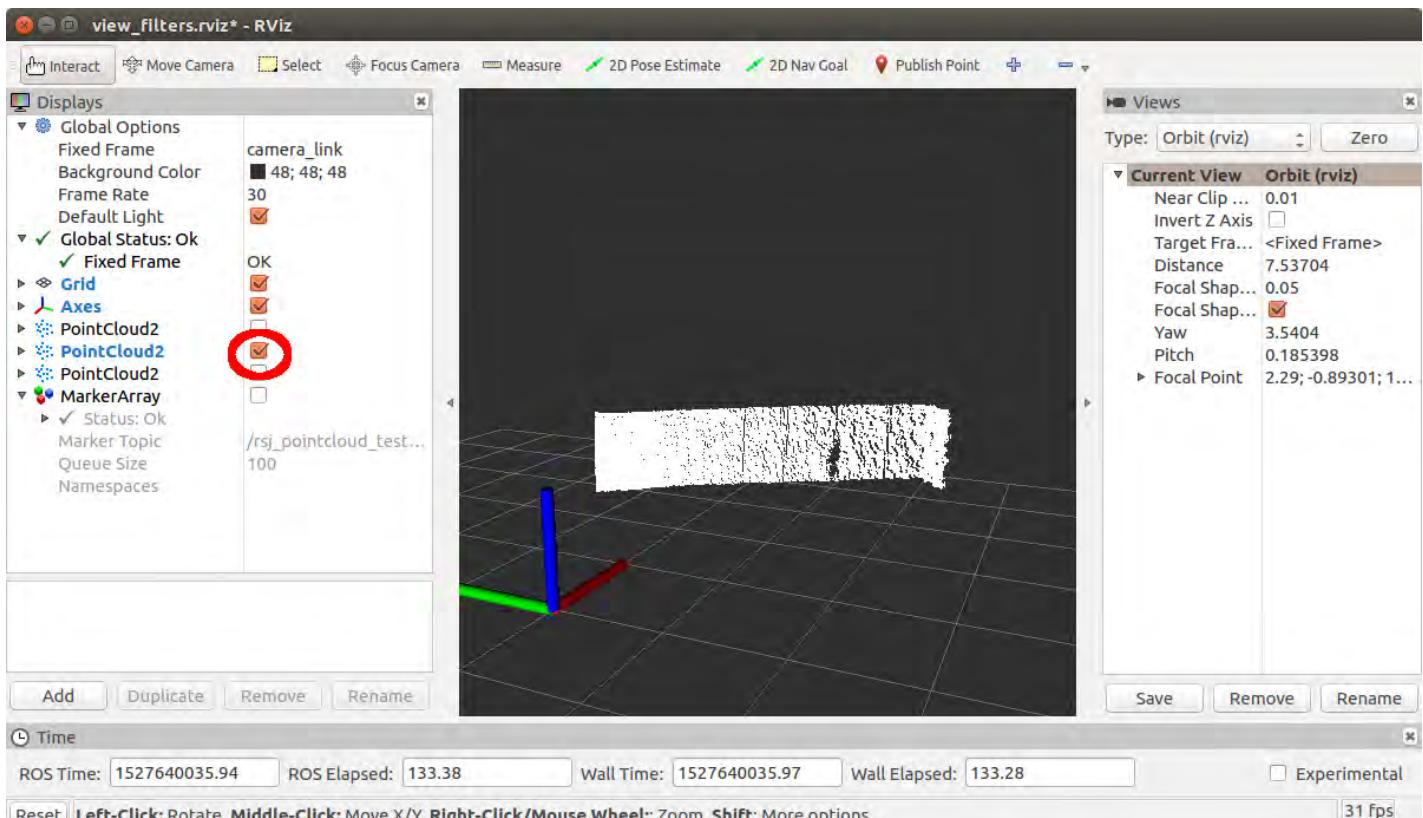
```
1 $ cd ~/catkin_ws/src/rsj_pointcloud_test/config/rviz
2 $ rviz -d view_filters_3durg.rviz
```

なお、RVizはroscoreが起動していれば、上記のようにrvizとタイプするだけでも実行可能です。

図のようにRvizの左方にあるPointCloud2のチェックボックスのうち、上2つだけにチェックを入れてください。フィルタ実行前と実行後の点群が重なって表示されています。



PointCloud2のチェックボックスのうち上から2番めのチェックだけをチェックするとフィルタ実行後の点群だけが表示されます。



VoxelGrid フィルタ

3次元点群の処理には時間がかかることが多いため、低スペックのPCの場合はある程度点を間引いておいた方が都合が良いことがあります。VoxelGridフィルタは等間隔に点群をダウンサンプリングします。引き続きrsj_pointcloud_test_node.cppを編集します。プログラム冒頭にinclude文を追記してください。

```

1 #include <pcl/filters/passthrough.h>
2 #include <pcl/filters/voxel_grid.h> // 追記
3 #include <visualization_msgs/MarkerArray.h>
```

```
4
5     typedef pcl::PointXYZ PointT;
```

RsjPointCloudTestNodeクラスの冒頭に、pcl::VoxelGridフィルタのインスタンスを追加します。また、フィルタの結果を格納するためのPointCloud型変数cloud_passthrough_、および処理結果をpublishするためのパブリッシャpub_voxel_を追加します。

```
1 class RsjPointCloudTestNode
2 {
3     private:
4         (略)
5         ros::Publisher pub_passthrough_;
6         // 以下を追記
7         pcl::VoxelGrid<PointT> voxel_;
8         PointCloud::Ptr cloud_voxel_;
9         ros::Publisher pub_voxel_;
10    }
```

RsjPointCloudTestNodeクラスのコンストラクタでVoxelGridフィルタの設定、cloud_voxel_およびpub_voxel_を初期化します。

```
1 RsjPointCloudTestNode()
2     : nh_()
3     , pnh_("~")
4 {
5     (略)
6     pub_passthrough_ = nh_.advertise<PointCloud>("passthrough", 1);
7     // 以下を追記
8     voxel_.setLeafsize(0.025f, 0.025f, 0.025f); // 0.025 m 間隔でダウンサンプリング
9     cloud_voxel_.reset(new PointCloud());
10    pub_voxel_ = nh_.advertise<PointCloud>("voxel", 1);
11 }
```

cbPoints関数を次のように変更します。

```
1 void cbPoints(const PointCloud::ConstPtr &msg)
2 {
3     try
4     {
5         (略)
6         pub_passthrough_.publish(cloud_passthrough_);
7         // 以下のように追記・修正
8         voxel_.setInputCloud(cloud_passthrough_);
9         voxel_.filter(*cloud_voxel_);
10        pub_voxel_.publish(cloud_voxel_);
11        ROS_INFO("points (src: %zu, paththrough: %zu, voxelgrid: %zu)",
12                 msg->size(), cloud_passthrough_->size(), cloud_voxel_->size());
13        // 追記・修正箇所ここまで
14    }
15    catch (std::exception &e)
16    {
17        ROS_ERROR("%s", e.what());
18    }
19 }
```

ビルド&実行

PassThroughフィルタのときと同様にビルドして実行してください。

フィルタ実行結果の可視化

RVizでフィルタ実行後の点群の様子を可視化します。rsj_pointcloud_test_nodeを起動したまま、新しいターミナルを開き、RVizを起動します。

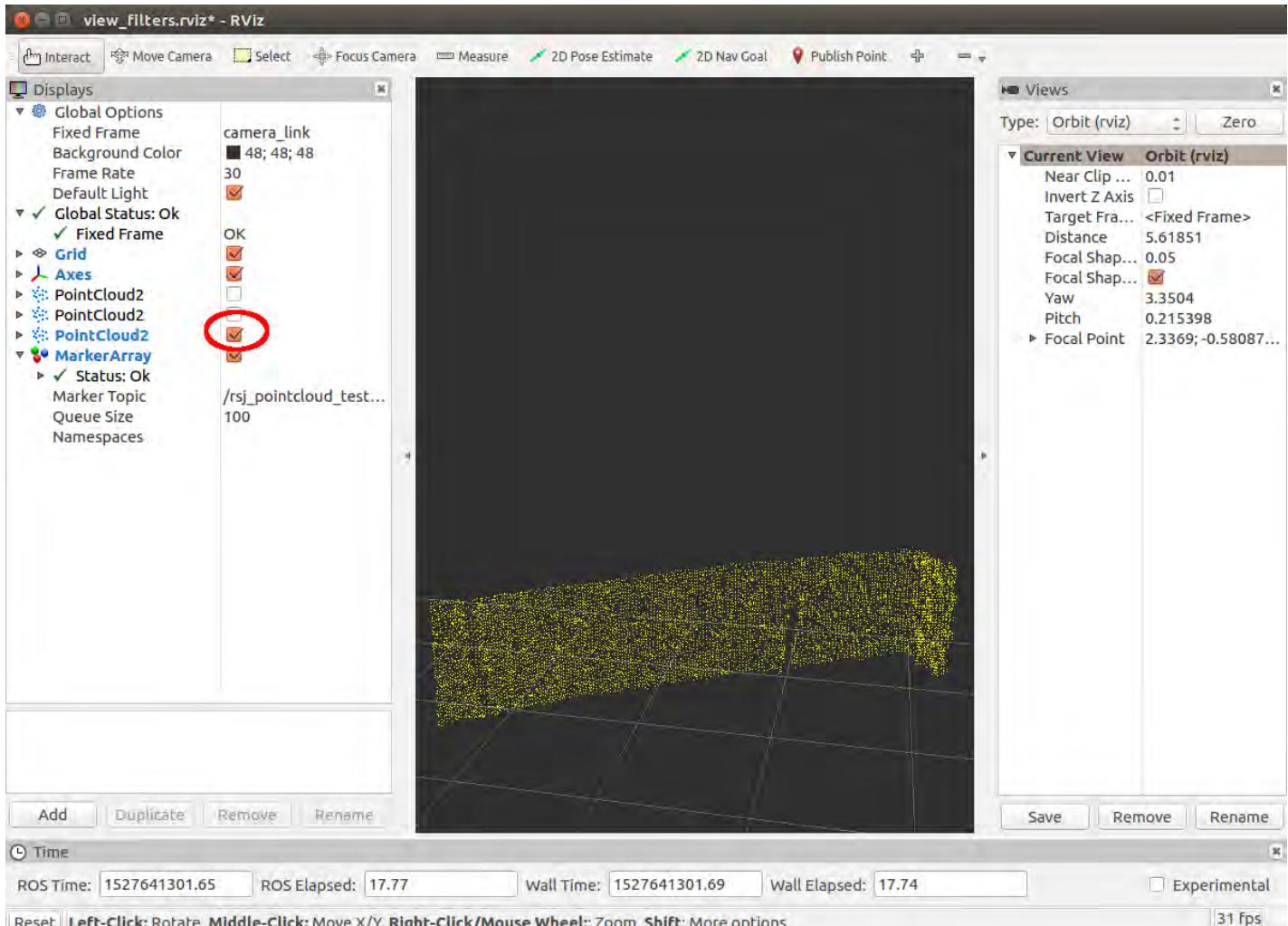
Xtion PRO Live の場合

```
1 $ cd ~/catkin_ws/src/rsj_pointcloud_test/config/rviz
2 $ rviz -d view_filters.rviz
```

YVT-35LX の場合

```
1 $ cd ~/catkin_ws/src/rsj_pointcloud_test/config/rviz
2 $ rviz -d view_filters_3durg.rviz
```

RViz の左にあるPointCloud2の一番下のチェックだけをONにするとvoxelGridフィルタ実行後の点群だけが表示されます。



PassThrough実行後の結果と比較すると点がまばらになっていることが分かると思います。もし違いがわかりにくい場合はsetLeafSize関数の引数を

```

1 RsjPointcloudTestNode()
2   : nh_()
3   , pnh_("~")
4 {
5   (略)
6   /* Xtion の場合 */
7   voxel_.setLeafSize(0.05f, 0.05f, 0.05f); // LeafSize 変更
8   /* YV7-35LX の場合 */
9   voxel_.setLeafSize(0.3f, 0.3f, 0.3f); // LeafSize 変更

```

のように大きくしてみてください（確認後は元の値に戻しておいてください）。

VoxelGridフィルタはRGBDカメラ（Xtion）のような得られる点群の数が非常に多いセンサに特に有効です。

クラスタリング

点群のクラスタリング（いくつかの塊に分離すること）により物体認識などをする際の物体領域候補が検出できます。プログラム冒頭にinclude文を追記してください。

```

1 #include <pcl/filters/voxel_grid.h>
2 #include <pcl/common/common.h> // 追記
3 #include <pcl/kdtree/kdtree.h> // 追記
4 #include <pcl/segmentation/extract_clusters.h> // 追記
5 #include <visualization_msgs/MarkerArray.h>
6
7 typedef pcl::PointXYZ PointT;

```

RsjPointcloudTestNodeクラスの冒頭に、`pcl::search::KdTree`クラスのポインタ、`pcl::EuclideanClusterExtraction`クラスのインスタンス、検出されたクラスタの可視化情報をパブリッシュする`pub_cluster`を追加します。

```
1 class RsjPointcloudTestNode
```

```

2
3     {
4         private:
5             // (略)
6             ros::Publisher pub_voxel_;
7             // 以下を追記
8             pcl::search::KdTree<PointT>::Ptr tree_;
9             pcl::EuclideanClusterExtraction<PointT> ec_;
10            ros::Publisher pub_clusters_;
11
12
13
14

```

RsjPointCloudTestNodeクラスのコンストラクタでpcl::EuclideanClusterExtractionの設定、tree_、pub_clusters_の初期化をします。

Xtion PRO Live の場合

```

1     RsjPointCloudTestNode()
2         : nh_()
3             , pnh_("~")
4     {
5         // (略)
6         pub_voxel_ = nh_.advertise<PointCloud>("voxel", 1);
7         // 以下を追記
8         tree_.reset(new pcl::search::KdTree<PointT>());
9         ec_.setClusterTolerance(0.15);
10        ec_.setMinClusterSize(100);
11        ec_.setMaxClusterSize(5000);
12        ec_.setSearchMethod(tree_);
13        pub_clusters_ = nh_.advertise<visualization_msgs::MarkerArray>("clusters", 1);
14

```

YVT-35LX の場合

```

1     RsjPointCloudTestNode()
2         : nh_()
3             , pnh_("~")
4     {
5         // (略)
6         pub_voxel_ = nh_.advertise<PointCloud>("voxel", 1);
7         // 以下を追記
8         tree_.reset(new pcl::search::KdTree<PointT>());
9         ec_.setClusterTolerance(0.15);
10        ec_.setMinClusterSize(5);
11        ec_.setMaxClusterSize(5000);
12        ec_.setSearchMethod(tree_);
13        pub_clusters_ = nh_.advertise<visualization_msgs::MarkerArray>("clusters", 1);
14

```

pcl::EuclideanClusterExtractionの設定部分のプログラムは次のとおりです。

```

ec_.setClusterTolerance(0.15);
    15cm以上離れていれば別のクラスタだとみなす
ec_.setMinClusterSize(100); ec_.setMaxClusterSize(5000);
    クラスタを構成する点の数は最低でも100個、最高で5000個
ec_.setSearchMethod(tree_);
    ある点とクラスタを形成可能な点の探索方法としてKD木を使用する。

```

cbPoints関数を次のように変更します。

```

1     void cbPoints(const PointCloud::ConstPtr &msg)
2     {
3         try
4         {
5             // (略)
6             pub_voxel_.publish(cloud_voxel_);
7             // 以下のように追記・修正
8             std::vector<pcl::PointIndices> cluster_indices;
9             tree_->setInputCloud(cloud_voxel_);
10            ec_. setInputCloud(cloud_voxel_);
11            ec_. extract(cluster_indices);
12            visualization_msgs::MarkerArray marker_array;
13            int marker_id = 0;
14            for (std::vector<pcl::PointIndices>::const_iterator
15                 it = cluster_indices.begin(),
16                 it_end = cluster_indices.end();
17                 it != it_end; ++it, ++marker_id)
18            {
19                Eigen::Vector4f min_pt, max_pt;
20                pcl::getMinMax3D(*cloud_voxel_, *it, min_pt, max_pt);
21                Eigen::Vector4f cluster_size = max_pt - min_pt;
22                if (cluster_size.x() > 0 && cluster_size.y() > 0 && cluster_size.z() > 0)
23                {
24                    marker_array.markers.push_back(
25                        makeMarker(
26                            frame_id, "cluster", marker_id, min_pt, max_pt, 0.0f, 1.0f, 0.0f, 0.5f));
27                }

```

```

28
29     if (marker_array.markers.empty() == false)
30     {
31         pub_clusters_.publish(marker_array);
32     }
33     ROS_INFO("points (src: %zu, paththrough: %zu, voxelgrid: %zu, cluster: %zu)",
34             msg->size(), cloud_passthrough_->size(), cloud_voxel_->size(),
35             cluster_indices.size());
36     // 追記・修正箇所ここまで
37 }
38 catch (std::exception &e)
39 {
40     ROS_ERROR("%s", e.what());
41 }
42 }
```

ビルド&実行

VoxelGridフィルタのときと同様にビルドして実行してください。

フィルタ実行結果の可視化

RVizでフィルタ実行後の点群の様子を可視化します。rsj_pointcloud_test_nodeを起動したまま新しいターミナルを開き、RVizを起動します。

Xtion PRO Live の場合

```

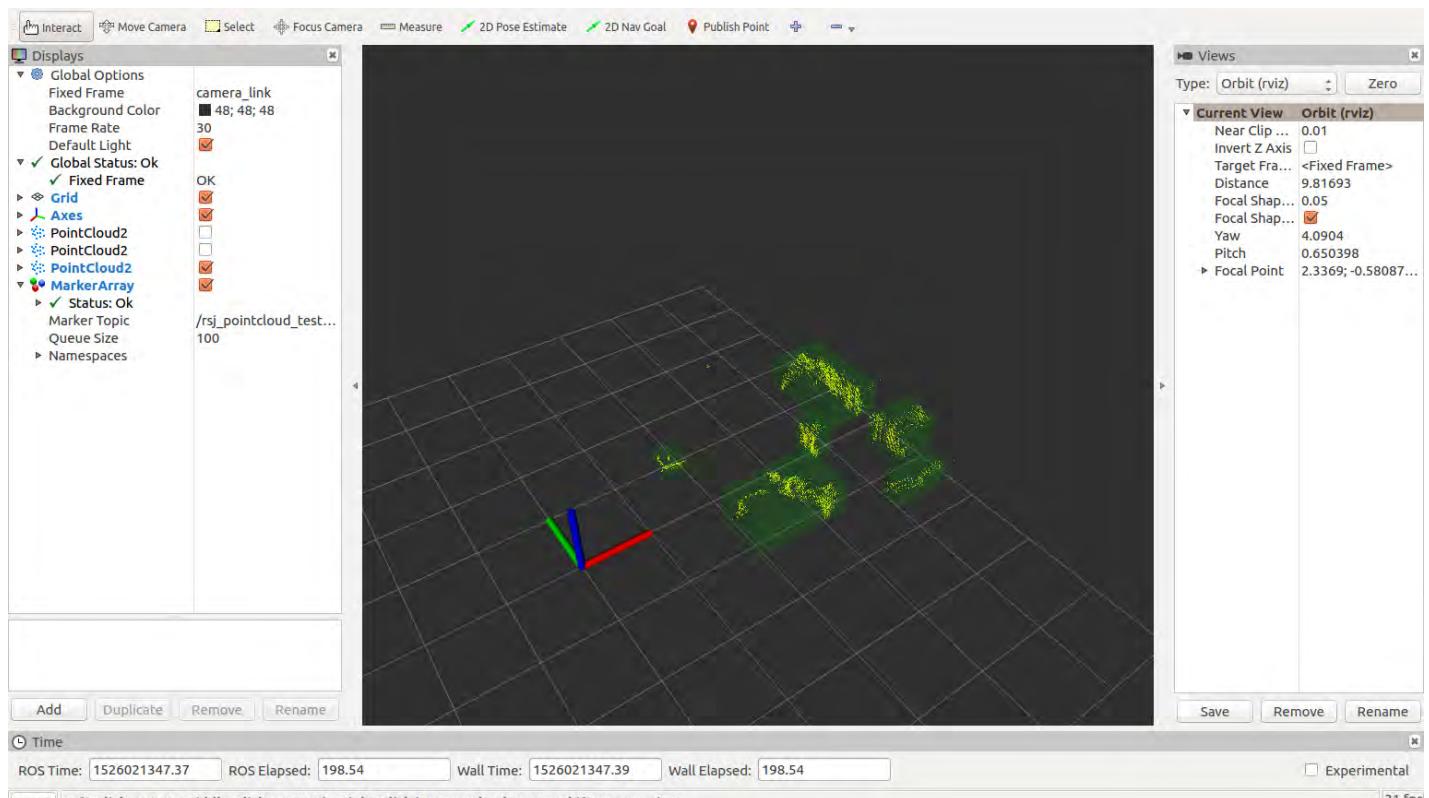
1 $ cd ~/catkin_ws/src/rsj_pointcloud_test/config/rviz
2 $ rviz -d view_filters.rviz
```

YVT-35LX の場合 の場合

```

1 $ cd ~/catkin_ws/src/rsj_pointcloud_test/config/rviz
2 $ rviz -d view_filters_3durg.rviz
```

RVizの左にあるPointCloud2の一番下のチェックだけをONにするとvoxelGridフィルタ実行後の点群だけが表示されます。さらにクラスタリング結果が半透明の緑のBOXで表示されているのが分かります。これはプログラム内でクラスタリング結果をRVizが可視化可能な型である visualization_msgs::MarkerArrayに変換してパブリッシュしているからです。



特定の条件に合致するクラスタを検出する

検出したクラスタのうち、一定の大きさをもつものだけを抽出するようにしましょう。最終的にはゴミ箱や人間の足など、特定の大きさなど何らかの条件を満たすクラスタに向かって走行するように制御します。

cbPoints関数を次のように変更します。

```

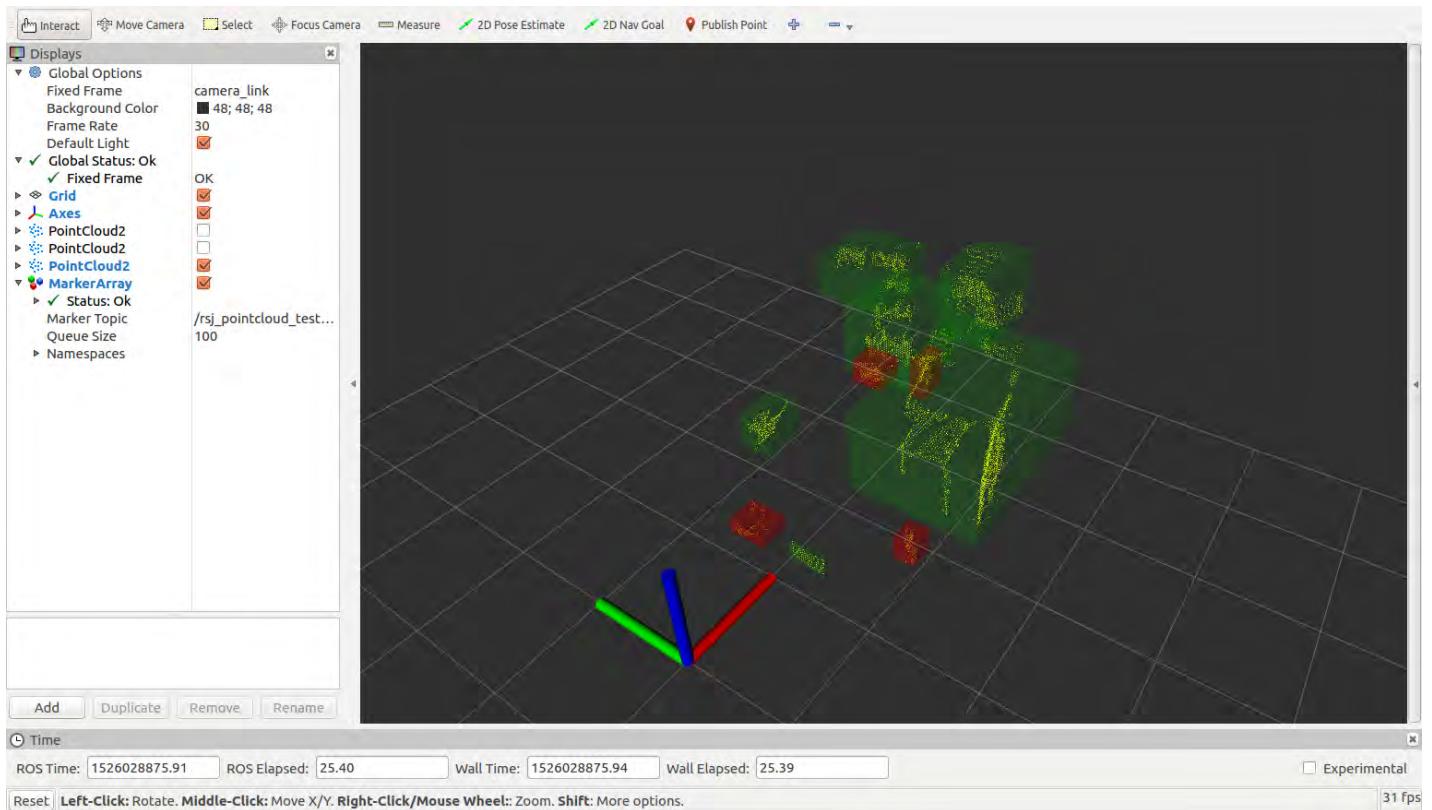
1 void cbPoints(const PointCloud::ConstPtr &msg)
2 {
3     try
4     {
5         (略)
6         pub_voxel_.publish(cloud_voxel_);
7         std::vector<pcl::PointIndices> cluster_indices;
8         tree_->setInputCloud(cloud_voxel_);
9         ec_.setInputCloud(cloud_voxel_);
10        ec_.extract(cluster_indices);
11        visualization_msgs::MarkerArray marker_array;
12        int marker_id = 0;
13        /* */
14        size_t ok = 0; // 追記
15        /* */
16        for (std::vector<pcl::PointIndices>::const_iterator
17              it = cluster_indices.begin(),
18              it_end = cluster_indices.end();
19              it != it_end; ++it, ++marker_id)
20        {
21            Eigen::Vector4f min_pt, max_pt;
22            pcl::getMinMax3D(*cloud_voxel_, *it, min_pt, max_pt);
23            Eigen::Vector4f cluster_size = max_pt - min_pt;
24            if (cluster_size.x() > 0 && cluster_size.y() > 0 && cluster_size.z() > 0)
25            {
26                // 以下を追記・修正
27                bool is_ok = true;
28                if (cluster_size.x() < 0.05 || cluster_size.x() > 0.4)
29                {
30                    is_ok = false;
31                }
32                else if (cluster_size.y() < 0.05 || cluster_size.y() > 0.6)
33                {
34                    is_ok = false;
35                }
36                else if (cluster_size.z() < 0.05 || cluster_size.z() > 0.5)
37                {
38                    is_ok = false;
39                }
40                visualization_msgs::Marker marker =
41                    makeMarker(
42                        frame_id, "cluster", marker_id, min_pt, max_pt, 0.0f, 1.0f, 0.0f, 0.5f);
43                if (is_ok)
44                {
45                    marker.ns = "ok_cluster";
46                    marker.color.r = 1.0f;
47                    marker.color.g = 0.0f;
48                    marker.color.b = 0.0f;
49                    marker.color.a = 0.5f;
50                    ok++;
51                }
52                marker_array.markers.push_back(marker);
53                // 追記・修正箇所ここまで
54            }
55        }
56        if (marker_array.markers.empty() == false)
57        {
58            pub_clusters_.publish(marker_array);
59        }
60        /*** 修正 ***/
61        ROS_INFO("points (src: %zu, paththrough: %zu, voxelgrid: %zu,"
62                  " cluster: %zu, ok_cluster: %zu)",
63                  msg->size(), cloud_passthrough_->size(), cloud_voxel_->size(),
64                  cluster_indices.size(), ok);
65        /*** 修正 ***/
66    }
67    catch (std::exception &e)
68    {
69        ROS_ERROR("%s", e.what());
70    }
71 }
```

ビルド&実行

クラスタリングのときと同様にビルドして実行してください。

フィルタ実行結果の可視化

クラスタリングのときと同様に RViz で可視化してください。ある一定の大きさのクラスタだけを赤く表示しているのが分かります。



最も近いクラスタを検出する

前項で抽出したクラスタのうち、センサに最も近いクラスタを選択するようしましょう。

`cbPoints`関数を次のように変更します。

```

1 void cbPoints(const PointCloud::ConstPtr &msg)
2 {
3     try
4     {
5         (略)
6         pub_voxel_.publish(cloud_voxel_);
7         std::vector<pcl::PointIndices> cluster_indices;
8         tree_->setInputCloud(cloud_voxel_);
9         ec_.setInputCloud(cloud_voxel_);
10        ec_.extract(cluster_indices);
11        visualization_msgs::MarkerArray marker_array;
12        /* */
13        int target_index = -1; // 追記
14        /* */
15        int marker_id = 0;
16        size_t ok = 0;
17        for (std::vector<pcl::PointIndices>::const_iterator
18              it = cluster_indices.begin(),
19              it_end = cluster_indices.end();
20              it != it_end; ++it, ++marker_id)
21        {
22            (略)
23            if (is_ok)
24            {
25                marker.ns = "ok_cluster";
26                marker.color.r = 1.0f;
27                marker.color.g = 0.0f;
28                marker.color.b = 0.0f;
29                marker.color.a = 0.5f;
30                ok++;
31                // 以下のように追記
32                if(target_index < 0){
33                    target_index = marker_array.markers.size();
34                }else{
35                    float d1 = ::hypot(marker_array.markers[target_index].pose.position.x,
36                                         marker_array.markers[target_index].pose.position.y);
37                    float d2 = ::hypot(marker.pose.position.x, marker.pose.position.y);
38                    if(d2 < d1){
39                        target_index = marker_array.markers.size();
40                    }
41                }
42            }
43            // 追記箇所ここまで
44        }
45        marker_array.markers.push_back(marker);
46    }
47    if (marker_array.markers.empty() == false)
48    {
49        // 以下のように追記
50        if(target_index >= 0){

```

```

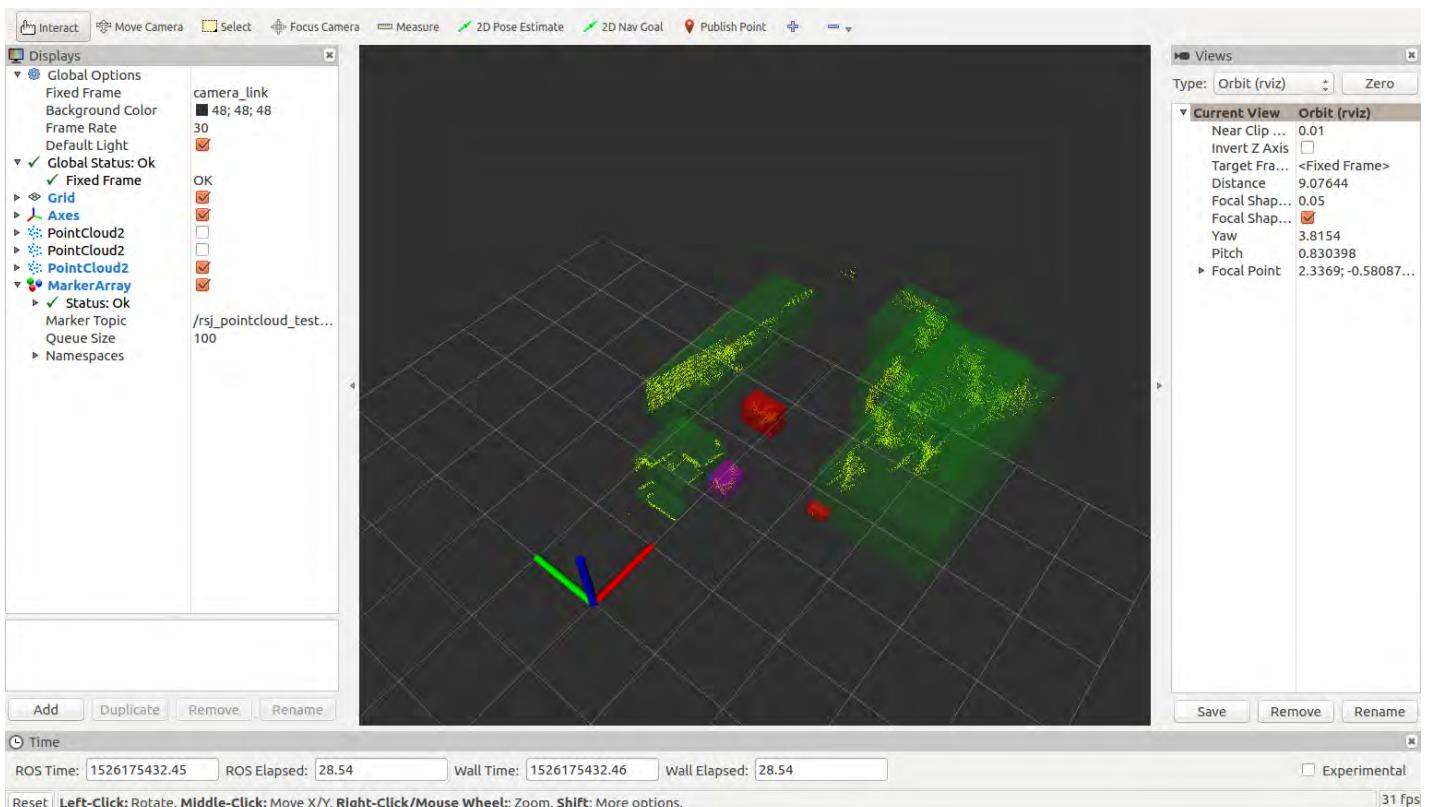
51     marker_array.markers[target_index].ns = "target_cluster";
52     marker_array.markers[target_index].color.r = 1.0f;
53     marker_array.markers[target_index].color.g = 0.0f;
54     marker_array.markers[target_index].color.b = 1.0f;
55     marker_array.markers[target_index].color.a = 0.5f;
56   }
57   // 追記箇所ここまで
58   pub_clusters_.publish(marker_array);
59 }
60 ROS_INFO("points (src: %zu, paththrough: %zu, voxelgrid: %zu,"
61           " cluster: %zu, ok_cluster: %zu)",
62           msg->size(), cloud_passthrough_->size(), cloud_voxel_->size(),
63           cluster_indices.size(), ok);
64 }
65 catch (std::exception &e)
66 {
67   ROS_ERROR("%s", e.what());
68 }
69 }
```

ビルド&実行

前項と同様にビルドして実行してください。

フィルタ実行結果の可視化

前項と同様に RViz で可視化してください。ある一定の大きさのクラスタだけを赤く表示し、その中にセンサに最も近いクラスタを紫で表示しているのが分かります。



点群処理とロボットナビゲーションの統合

PCL のクラスタリング結果を rsj_robot_test_node で利用してみましょう。

rsj_robot_test.cpp の編集

テキストエディタで rsj_robot_test.cpp を開いてください。

```
1 $ cd ~/catkin_ws/src/rsj_robot_test/src  
2 任意のテキストエディタで rsj_robot_test.cpp を開く
```

先頭に visualization_msgs::MarkerArray を扱うための include 文を加えます。

```
1 #include <tf/transform_datatypes.h>  
2 #include <visualization_msgs/MarkerArray.h> // 追記
```

CMakeLists.txt を編集します。

```
1 $ cd ~/catkin_ws/src/rsj_robot_test  
2 任意のテキストエディタで CMakeLists.txt を開く
```

find_package と catkin_package の最後に visualization_msgs を追記します。

```
1 find_package(catkin REQUIRED COMPONENTS  
2   (略)  
3   tf  
4   visualization_msgs # 追記  
5 )  
6 find_package(Boost 1.53 REQUIRED system serialization)  
7  
8 ## Declare a catkin package  
9 catkin_package(DEPENDS  
10   (略)  
11   tf  
12   visualization_msgs # 追記  
13 )
```

また、 package.xml の <build_depend> <run_depend> にも visualization_msgs を追記します。

```
1 (略)  
2 <build_depend>tf</build_depend>  
3 <build_depend>visualization_msgs</build_depend> <!--追記-->  
4  
5 (略)  
6 <run_depend>tf</run_depend>  
7 <run_depend>visualization_msgs</run_depend> <!--追記-->
```

一旦ビルドし、コンパイルエラーがないことを確認してください。

```
1 $ cd ~/catkin_ws  
2 $ catkin_make
```

クラスタリング結果の受信

RsjRobotTestNode クラスで、 sub_odom_ や sub_scan_ を定義しているところに、 visualization_msgs::MarkerArray 用のサブスクライバを追加します。

```
1 ros::Subscriber sub_scan_;  
2 ros::Subscriber sub_clusters_; // 追記
```

RsjRobotTestNode のコンストラクタに、 visualization_msgs::MarkerArray 用のサブスクライバ初期化コードを追加します。

```

1 RsjRobotTestNode()
2 {
3   (略)
4   sub_scan_ = nh_.subscribe(
5     "scan", 5, &RsjRobotTestNode::cbScan, this);
6   sub_clusters_ = nh_.subscribe(
7     "clusters", 5, &RsjRobotTestNode::cbCluster, this); // 追記

```

更に、RsjRobotTestNodeクラスに、visualization_msgs::MarkerArray用のコールバック関数を追加します。(cbScanの後の位置など)

```

1 void cbCluster(const visualization_msgs::MarkerArray::ConstPtr &msg)
2 {
3   ROS_INFO("clusters: %zu", msg->markers.size());
4 }

```

編集が終了したらエディタを閉じてください。

ビルト&実行

ターミナルで次のコマンドを実行してください。

```

1 $ cd ~/catkin_ws
2 $ catkin_make

```

お手持ちの3Dセンサ、およびロボットのUSBケーブルをPCに接続しておきます。

また、ロボットが走り出さないように、電池ボックスのスイッチをOFFにしておいてください。

まずナビゲーションシステムを起動します。地図を作成した際の初期位置・姿勢と同じようにロボットを置いて、下記のコマンドを実行します。

Xtion PRO Live の場合

```

1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ roslaunch rsj_seminar_navigation xtion_integration.launch \
4   robot_param:=~/home/【ユーザ名】/params/rsj-seminar20???.param 【該当するものに置き換えること】

```

YVT-35LX の場合

```

1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ roslaunch rsj_seminar_navigation 3durg_integration.launch \
4   robot_param:=~/home/【ユーザ名】/params/rsj-seminar20???.param 【該当するものに置き換えること】

```

rsj_pointcloud_test_node と rsj_robot_test_node の起動

3次元センサをお持ちの場合は新しいターミナルを開きrsj_pointcloud_test_nodeを起動します。

Xtion PRO Live の場合

```

1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun rsj_pointcloud_test rsj_pointcloud_test_node \
4   _target_frame:=camera_link _topic_name:=/camera/depth_registered/points
5 [ INFO] [1524040063.315596383]: target_frame='camera_link'
6 [ INFO] [1524040063.315656650]: topic_name='/camera/depth_registered/points'
7 [ INFO] [1524040063.320448185]: Hello Point Cloud!
8 [ INFO] [1524040064.148595331]: points (src: 307200, paththrough: 34350)

```

YVT-35LX の場合

```

1 $ cd ~/catkin_ws/

```

```

2 $ source devel/setup.bash
3 $ rosrun rsj_pointcloud_test rsj_pointcloud_test_node \
4   _target_frame:= _topic_name:=/hokuyo3d/hokuyo_cloud2
5 [ INFO] [1528008816.751100536]: points (src: 2674, paththrough: 1019)

```

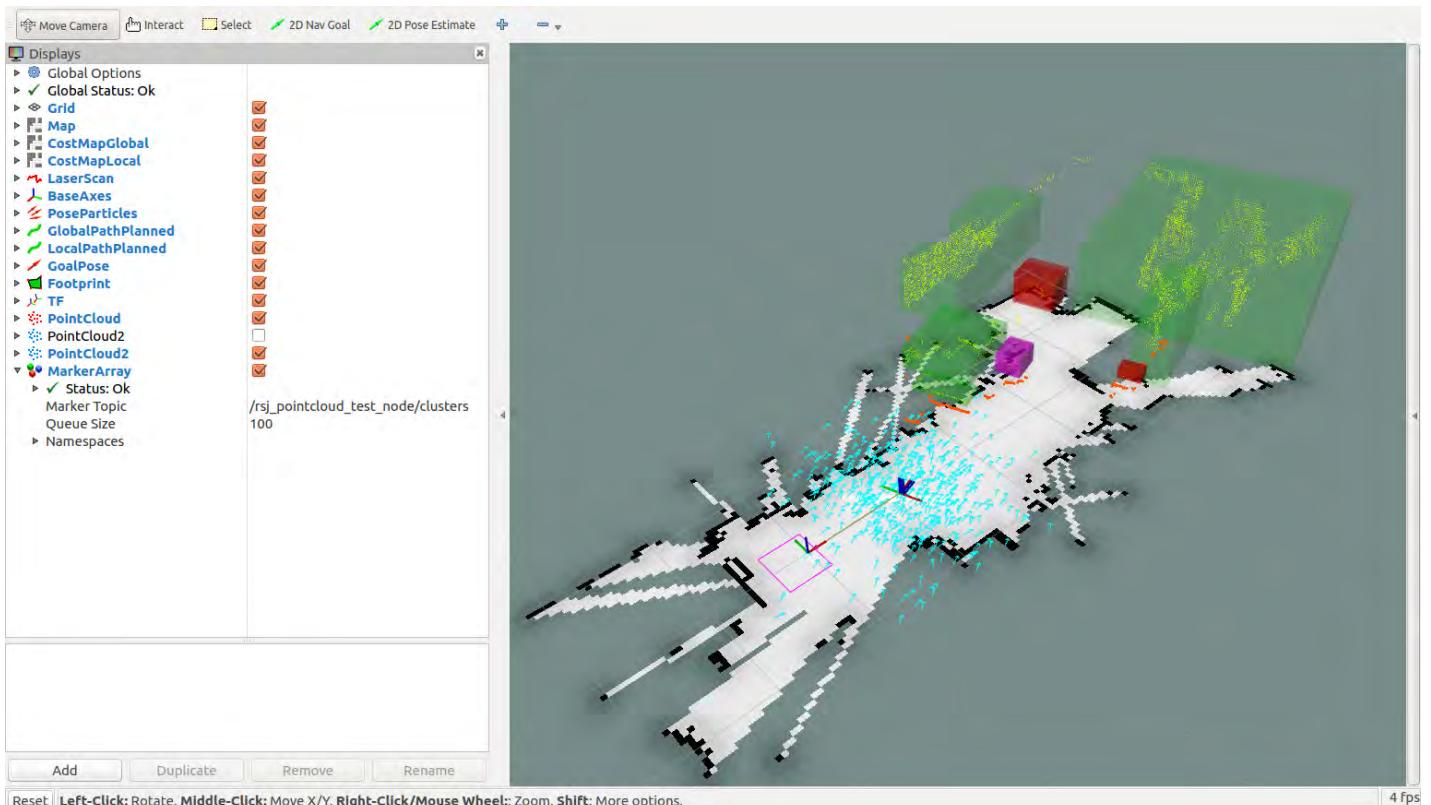
さらに別のターミナルでrsj_robot_test_nodeを起動します。

```

1 $ cd ~/catkin_ws/
2 $ source devel/setup.bash
3 $ rosrun rsj_robot_test rsj_robot_test_node
4 [ INFO] [1523957582.691740639]: Hello ROS World!
5 [ INFO] [1523957582.991946984]: clusters: 6
6 [ INFO] [1523957583.091959056]: clusters: 7
7 [ INFO] [1523957583.191939063]: front-range: 3.178
8 [ INFO] [1523957583.192001041]: front-range: 3.178

```

rsj_robot_test_node側の端末で「[INFO] [1523957583.091959056]: clusters: 7」のようにPCLで処理したクラスタを受信し、その個数を表示できていることが分かります。またRViz上ではnavigation用のマップ上に重畳してPointCloudのクラスタと、クラスタを囲む直方体が表示されています。センサに最も近いクラスタは紫で表示されています。



センサに最も近いクラスタの位置情報を取得する

RViz上で紫で表示されている、センサに最も近いクラスタの位置を取得しましょう。

テキストエディタでrsj_robot_test.cppを開いてください。

```

1 $ cd ~/catkin_ws/src/rsj_robot_test/src
2 任意のテキストエディタで rsj_robot_test.cpp を開く

```

cbCluster関数を編集します。

```

1 void cbCluster(const visualization_msgs::MarkerArray::ConstPtr &msg)
2 {
3     const visualization_msgs::Marker *target = NULL;
4     for (visualization_msgs::MarkerArray::const_iterator
5          it = msg->markers.cbegin(),
6          it_end = msg->markers.cend();
7          it != it_end; ++it)
8     {
9         const visualization_msgs::Marker &marker = *it;
10        if (marker.ns == "target_cluster")
11        {

```

```
12     target = &marker;
13 }
14 }
15 ROS_INFO("clusters: %zu", msg->markers.size());
16 if (target != NULL)
17 {
18     float dx = target->pose.position.x;
19     float dy = target->pose.position.y;
20     ROS_INFO("target: %f, %f", dx, dy);
21 }
22 }
```

編集が終了したらエディタを閉じてください。

ビルド&実行

前項と同じようにビルドして実行してください。rsj_robot_test_node側の端末で「[INFO] [1526342853.141823400]: target: 2.579500, 0.063012」のようにPCLで処理した最も近いクラスタの座標が表示できています。なおここで表示されている座標はロボットの中心を原点とし、正面をX軸プラス方向とするローカル座標系です。

終了したら[課題](#)に進んでください。

点群処理とロボットナビゲーションの統合

これまでにセミナーで取り扱った内容のおさらいに演習を用意していますので、取り組んでみてください。

PointCloudのデータを使ったロボット制御

RViz 上で紫で表示されている、センサに最も近いクラスタに対してロボットが近づくように制御してください。ただし一定の距離まで近づいたら停止するようにしてください。

ヒント：ロボットローカル座標系における紫のクラスタの位置はすでに計算されています。ロボットがその方向を向くように角速度を算出します。直進速度は一定の値でも良いですし、角速度が大きくなる場合はそれに応じて小さくするなどすれば小回りが効く制御になります。

URG-04LX-UG01(2DURG)

2DURG のデータを利用して、ロボットが人に追従するように制御してみましょう。基本的な方法は上記の課題と同じです。

ROS navigation メタパッケージの利用

RViz 上で紫で表示されている、センサに最も近いクラスタに対してロボットが近づくように制御してください。ただし、ROS navigation メタパッケージを利用します。これにより、経路上の障害物を回避しながら近づくようにします。

ヒント：

- /move_base_simple/goal トピックにgeometry_msgs::PoseStamped型のメッセージをパブリッシュし、目標地点の位置姿勢を与えることで、ナビゲーションの指示を与えられます。
- geometry_msgs::PoseStamped型のメッセージの作成方法の例は次のとおりです。

```

1  geometry_msgs::PoseStamped goal;
2  goal.header.frame_id = "base_link"; // ロボットローカル座標におけるゴール位置を指定したい場合のコード。
3
4  goal.pose.position.x = 目標の x 座標;
5  goal.pose.position.y = 目標の y 座標;
6
7  // 目標地点に到達した際にロボットを向かせたい方向を指定する。
8  goal.pose.orientation = tf::createQuaternionMsgFromYaw(ロボットから見た目標の方向をラジアンで指定);
9  // 良くわからなければゼロ度でも構わない。

```

- geometry_msgs::PoseStampedのパブリッシュは目標発見時に1回行えば十分です。したがって、目標探索中か追跡中かといった状態を管理するのが簡単です。