2024 機械情報夏学期 ソフトウェア第二

担当：岡田 慧 (k-okada@jsk.t.u-tokyo.ac.jp)

12. 分散プログラミング応用

1　オープンソースロボティクス

ソースコードを誰でも自由に入手・改変しながら開発を行うオープンソースソフトウェアは，Linux や Android 等の OS をはじめとして，gcc, Java, Python 等のプログラミング環境，さらには Firefox や OpenOffice (LibreOffice) 等のアプリケーションまで，コンピュータサイエンスの分野では広く受け入れられている．これはソースコードをオープンにして誰でも入手できるようにすることで，世界中から共同開発者とベータテスタを呼び込み短い期間で質の高いソフトウェアを構築するための方法論であり，OS，プログラム環境，アプリケーション等コンピュータサイエンスの分野では広く受け入れられている．

重要なのは共同開発者とベータテスタからなるコミュニティの構築であり，そのなかで活発に議論をしながらスピード感をもってバージョンアップを繰り返しソフトウェアの質を高めさらなるユーザの獲得を目指す戦略であり，ソースコードの公開はそのための手段であることに注意したい[1]．

一方，ロボット研究分野ではオープンソースの OS や開発環境を活用することはあっても，その研究のコアとなる知的な情報処理モジュールに関してはオープンになることは無かった．しかしここ数年，ロボット研究分野においてもオープンソースソフトウェアが，特に海外を中心に急速に拡がり従来のロボット分野の研究者だけでなくこれまでロボットを利用してこなかった層を巻き込んだ大きなムーブメントになりつつある，特に ROS(Robot Operating System) とよばれるロボットソフトウェアのプロジェクトが欧米を中心に急速にユーザを惹き付け注目を集めている[2]

[1] http://www.catb.org/~esr/writings/cathedral-bazaar/, 日本語訳山形浩生 伽藍とバザール http://cruel.org/freeware/cathedral.html

[2] ROS のユーザ統計は http://wiki.ros.org/Metrics,ROS の開発意図については http://www.willowgarage.com/blog/2010/04/27/reinventing-wheel

2 ROS(Robot Operating System)

ROS は近年急速にユーザを増やしているロボット用の OS であり，それは以下の様に通信機構 (plumbing) + ツール (tools) + ライブラリ群 (capabilities) + コミュニティ (ecosystem) から構成されるプロジェクトである．

```
I usually explain ROS in the following way:

ROS = plumbing + tools + capabilities + ecosystem

   1. plumbing: ROS provides publish-subscribe messaging
      infrastructure designed to support the quick and easy
      construction of distributed computing systems.

   2. tools: ROS provides an extensive set of tools for configuring,
      starting, introspecting, debugging, visualizing, logging,
      testing, and stopping distributed computing systems.

   3. capabilities: ROS provides a broad collection of libraries that
      implement useful robot functionality, with a focus on mobility,
      manipulation, and perception.

   4. ecosystem: ROS is supported and improved by a large community,
      with a strong focus on integration and documentation. ros.org is
      a one-stop-shop for finding and learning about the thousands of
      ROS packages that are available from developers around the
      world.

In the early days, the plumbing, tools, and capabilities were tightly
coupled, which has both advantages and disadvantages. On the one hand,
by making strong assumptions about how a particular component will be
used, developers are able to quickly and easily build and test complex
integrated systems. On the other hand, users are given an "all or
nothing" choice: to use an interesting ROS component, you pretty much
had to jump in to using all of ROS.

Four years in, the core system has matured considerably, and we're
hard at work refactoring code to separate plumbing from tools from
capabilities, so that each may be used in isolation. In particular,
we're aiming for important libraries that were developed within ROS to
become available to non-ROS users in a minimal-dependency fashion (as
has already happened with OMPL and PCL).

posted Dec 06
Brian Gerkey
```

ROS の大きな特徴はソースコードをオープンにするだけでなく，質問サイトも積極的に運営し，さらには公式 Wiki の情報や機能提案も万人が参加できる様になっている．

ソースコード `https://github.com/ros`

質問サイト `http://answers.ros.org/questions/`

Wiki `http://www.ros.org/wiki/`

機能提案 `http://ros.org/reps/rep-0000.html`

本講義受講者は是非積極的にこれらのコミュニティに参加して欲しい．英語が不得手な者は日本語のメーリングリスト[3] が存在するので活用できる．

[3] `https://groups.google.com/forum/#!forum/ros-japan-users`

2.1 ROS ディストリビューション

ROS はディストリビューションと呼ばれるバージョンが存在し，現在は 2016 年 5 月にリリースされた Kinetic, 2018 年 5 月にリリースされた Melodic, 2020 年 5 月にリリースされた Noetic, が利用可能である[4].

本講義では学科 PC でには既にインストールされている noetic を前提にすすめることにする．

まずは以下の apt コマンドにより全てのパッケージが最新にする．ただし 15 分ほど時間がかかるので注意．

```
$ sudo apt-get update
$ sudo apt-get dist-upgrade
```

`sudo apt-get dist-upgrade` はインストール済みのプログラム全体を更新する．通常は問題が出ないが，`apt` 以外の `sudo make install` などで個別にプログラムをインストールしてしまっていたり，自ら PPA を追加しているような場合は，

```
$ dpkg --get-selections | grep ros-noetic | cut -f 1 -d$'\t' | xargs sudo apt upgrade -y
```

と，`ros-noetic` から始まるパッケージだけを最新版に更新する．こちらのほうが，余計なトラブルがない可能性が高いかもしれない．

このとき，

```
パッケージリストを読み込んでいます... 完了
E: ロック /var/lib/apt/lists/lock が取得できませんでした - open (11: リソースが一時的に利用できません)
E: ディレクトリ /var/lib/apt/lists/ をロックできません
```

と表示され場合,「パッケージの更新プログラム」がバックグラウンドで走っている可能性が高い．数分待って再度コマンドを実施すればよい．

また，`sudo apt-get update` のコマンドを実行した後，

```
取得:7 http://jp.archive.ubuntu.com/ubuntu focal-updates InRelease [88.7 kB]
取得:3 http://packages.ros.org/ros/ubuntu focal InRelease [4,680 B]
エラー:3 http://packages.ros.org/ros/ubuntu focal InRelease
  以下の署名が無効です: EXPKEYSIG F42ED6FBAB17C654 Open Robotics <info@osrfoundation.org>
取得:8 http://dl.google.com/linux/chrome/deb stable/main amd64 Packages [1,104 B]
ヒット:9 http://apt.insync.io/ubuntu focal InRelease
.....
456 MB を 2 分 52 秒 で取得しました (2,647 kB/s)
パッケージリストを読み込んでいます... 完了
W: 署名照合中にエラーが発生しました。リポジトリは更新されず、過去のインデックスファイルが使われます。GPG エラー: http://packages.ros.org/ros/ubuntu focal InRelease: 以下の署名が無効です: EXPKEYSIG F42ED6FBAB17C654 Open Robotics <info@osrfoundation.org>
W: http://packages.ros.org/ros/ubuntu/dists/focal/InRelease の取得に失敗しました 以下の署名が無効です: EXPKEYSIG F42ED6FBAB17C654 Open Robotics <info@osrfoundation.org>
W: いくつかのインデックスファイルのダウンロードに失敗しました。これらは無視されるか、古いものが代わりに使われます。
```

というエラーが出た場合は以下のコマンドで公開鍵を更新し，再度 `sudo apt-get update` からやり直す必要がある．

```
$ wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

[4] http://ros.org/reps/rep-0003.html, http://wiki.ros.org/Distributions

2.2 2023 年度受講生用 PC セットアップ

学科 PC を利用しているものは，資料を読み進める前に以下を実行すること．

```
$ sudo apt-get update
$ sudo apt-get install ros-noetic-desktop
$ sudo apt-get install python3-rosdep python3-catkin-tools python3-wstool
$ rosdep update
```

`rosdep update` や `rosdep install` を実行した際に

```
$ rosdep update

Command 'rosdep' not found, but can be installed with:

sudo apt install python3-rosdep2
```

と表示される場合があるが，このまま実行してはいけない．万が一実行すると，

```
$ sudo apt install python3-rosdep2
パッケージリストを読み込んでいます... 完了
依存関係ツリーを作成しています
状態情報を読み取っています... 完了
これを削除するには 'sudo apt autoremove' を利用してください。
以下の追加パッケージがインストールされます:
  python3-rosdistro
以下のパッケージは「削除」されます:
  python3-rosdep-modules ros-noetic-actionlib ros-noetic-actionlib-tutorials ros-noetic-amcl
  ros-noetic-bondpy ros-noetic-camera-info-manager ros-noetic-carrot-planner ros-noetic-clea
  ros-noetic-costmap-2d ros-noetic-desktop
....
以下のパッケージが新たにインストールされます:
  python3-rosdep2 python3-rosdistro
アップグレード: 0 個、新規インストール: 2 個、削除: 142 個、保留: 106 個。
59.2 kB のアーカイブを取得する必要があります。
この操作後に 116 MB のディスク容量が解放されます。
続行しますか? [Y/n]
```

として大量のファイルを削除してしまう．この場合は「続行しますか」の問に「n」を打ち込むこと．

2.3 ROS の体験

ROS のコンセプトやツールを学ぶ前に簡単な ROS の体験をしてみよう．まずは

```
$ roscore
```

として ROS を使うのに必要な基本プログラムであるネームサーバプログラムを立ち上げる．

roscore と打ち込んだ時に

```
Command 'roscore' not found, but can be installed with:

sudo apt install python-roslaunch
```

と表示される場合は、

```
$ source /opt/ros/noetic/setup.bash
```

として ROS を利用するのに必要な重要な環境変数を設定してから roscore を立ち上げる．この処理はターミナルを開く度に実行する必要がある．

それでも command `roscore` not found が表示される場合は

```
$ sudo apt-get update
$ sudo apt-get install ros-noetic-desktop
```

としてインストールする．

ターミナルを立ち上げるたびに source /opt/ros/noetic/setup.bash を実行するのが面倒な場合は

```
$ echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

として環境設定がシェル起動時に反映されるようにする．

この時，>>を>と間違えると，必要な設定が消えてしまうので，最新の注意を払うこと．

2.3.1 分散メッセージ通信の体験

サンプルプログラムがインストールされることを rospack find roscpp_tutorials を実行し確認する．

既に起ち上げたターミナルは roscore が起動している状態なので，

```
$ rospack find roscpp_tutorials
/opt/ros/noetic/share/roscpp_tutorials
```

となればインストールされている．command `rospack` not found が表示される場合は

```
$ source /opt/ros/noetic/setup.bash
```

とする．

```
[rospack] Error: package 'roscpp_tutorials' not found
```

となれば、インストールされていないので、以下のようにしてインストールしておく.

```
$ sudo apt-get install ros-noetic-roscpp-tutorials
```

分散メッセージ通信の簡単なサンプルプログラムを実行するには、まず、以下のようにしてデータの送信側（パブリッシャ）を起動する.

```
$ rosrun roscpp_tutorials  talker
[ INFO] [1380637716.614071446]: hello world 0
[ INFO] [1380637716.714164961]: hello world 1
[ INFO] [1380637716.814151914]: hello world 2
[ INFO] [1380637716.914144863]: hello world 3
```

これで ROS ネットワーク上にデータが送出されていることになる．例えば,

```
$ rosnode  list
/rosout
/talker
```

として，２つの ROS ノード（プログラム）が存在することが分かる．rosout は roscore が立ち上げたノードである.

また，次にようにして talker ノードの情報を得ることが出来る.

```
$ rosnode info talker
--------------------------------------------------------------------------------
Node [/talker]
Publications:
 * /chatter [std_msgs/String]
 * /rosout [rosgraph_msgs/Log]

Subscriptions: None

Services:
 * /talker/set_logger_level
 * /talker/get_loggers
```

また次のようにして talker ノードが送出するするデータ（メッセージ）を確認することが出来る.

```
$ rostopic echo /chatter
data: hello world 428
---
data: hello world 429
```

talker に対応するサンプルプログラムは listener である．これは以下のようにして起動できる．この時/chatter というトピックを介して talker から listener にメッセージが送出されている.

```
$ rosrun  roscpp_tutorials  listener
[ INFO] [1380638754.914750213]: I heard: [hello world 10383]
[ INFO] [1380638755.014553211]: I heard: [hello world 10384]
[ INFO] [1380638755.114557256]: I heard: [hello world 10385]
[ INFO] [1380638755.214551899]: I heard: [hello world 10386]
```

また rostpic info コマンドを用いて以下のように通信路を確認することが出来る．

```
$ rostopic info  /chatter
Type: std_msgs/String

Publishers:
 * /talker (http://kokada-t430s:60652/)

Subscribers:
 * /listener (http://kokada-t430s:40825/)
```

また，Python のサンプルプログラムは rospy_tutorials パッケージにある．

```
$ rosrun rospy_tutorials listener.py
```

で実行することができる．

http://wiki.ros.org/ROS/Tutorials に情報がまとまっているので一読すると良い．

2.4 ROS サンプルプログラム

http://wiki.ros.org/ja/ROS/Tutorials/CreatingPackage に従い ROS のサンプルプログラムを作るための作業ディレクトリを作成する．ここでは~/catkin_ws を作業ディレクトリとする．

```
$ source /opt/ros/noetic/setup.bash
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
```

としてワークスペース（作業用ディレクトリ）を初期化する．以降，このワークスペース以下にパッケージを作成していく．

次に，

```
$ catkin create pkg beginner_tutorials --catkin-deps std_msgs \
   geometry_msgs rospy roscpp message_generation message_runtime
$ cd ~/catkin_ws/
$ catkin build
```

として ROS のパッケージ[5]を作成する．

catkin create の部分の std_msgs と geomeotry_msgs 以降は一つの行として扱うこと．

これでいくつかのファイルが自動で作成される．例えば cat beginner_tutorials/package.xml として，message_runtime という文字がファイル中に書かれていれば，成功している可能性が高い．

ここで catkin build に失敗する場合は，パッケージの作成時にコマンド (catkin create pkg) の打ち間違いの可能性が高い．再度よく確認すること．

間違いを見つけたときは一度パッケージ (src/beginner_tutorials ディレクトリ) を消去して再度 catkin create pkg コマンドを実行すると良い．

```
$ source ~/catkin_ws/devel/setup.bash
$ roscd beginner_tutorials
```

とすると，roscd コマンドでこのディレクトリに移動できるようになる．新しくターミナルを開いた

[5]ros ではあるまとまったプログラム群を管理するディレクトリをパッケージと呼ぶ

場合は必ず **source catrkin_ws/devel/setup.bash** をやり直す必要があるので注意すること．

catkin というコマンドがなければ `sudo apt-get install python-catkin-tools` としてみよう．

簡単な通信プログラムの例として Python の例，C++の例を見てみよう．それぞれ、`http://wiki.ros.org/ja/ROS/Tutorials/WritingPublisherSubscriber%28python%29`、`http://wiki.ros.org/ja/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29` に詳細な説明がある．

2.4.1 Python を使ったパブリッシャ

まずは Python を使ってパブリッシャを記述する．そのための準備として

```
$ roscd beginner_tutorials
$ mkdir scripts
$ cd scripts
```

としてディレクトリを作成し，この `script` ディレクトリの下に以下のプログラムを `talker.py` として保存する．

```
#!/usr/bin/env python

import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=1)
    rospy.init_node('talker')
    while not rospy.is_shutdown():
        str = "hello world %s"%rospy.get_time()
        rospy.loginfo(str)
        pub.publish(String(str))
        rospy.sleep(1.0)
if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException: pass
```

保存したら，

```
$ python ./talker.py
```

として，プログラムを起動[6]して，別のターミナルから

```
$ rostopic echo /chatter
```

としてデータを受信できる．

/chatter はトピック名と呼ばれる通信路につけられた名前である．`rostopic info /chatter` とすると，通信路で流れる型 (メッセージ) 等を知ることができる．同じ型（メッセージ）を別の通信路（トピック）で送受信することができる．`rosmsg show std_msgs/String` とすると型の詳細情報が得られる．

ここで，

[6]予め別ターミナルで roscore としてネームサーバを立ち上げておくことを忘れないように

```
$ chmod u+x ./talker.py
```

として実行ビットをつけると，
beginner_tutorials/script ディレクトリにいる場合は，

```
$ ./talker.py
```

として，またそれ以外のときでも，

```
$ rosrun beginner_tutorials talker.py
```

として実行することが出来る．

2.4.2 C++を使ったサブスクライバ

クライアント側のプログラムを c++で書いたのが次の<beginner_tutorials>/src/listener.cpp になる．

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
  ros::spin();
  return 0;
}
```

これができたら<beginner_tutorials>/CMakeLists.txt の最後に

```
add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
```

とかいて，

```
$ (cd ~/catkin_ws; catkin build)
```

とするとコンパイルされ

```
$ rosrun beginner_tutorials listener
```

とすると受信側のプログラムを実行できる．

Python と C++という異なる言語で書かれているプログラム同士であるが，それぞれはお互いを意識せずにプログラムを記述することができている点に注目してほしい．

2.4.3 オリジナルメッセージの追加

独自の型を追加したければ

```
$ roscd beginner_tutorials
$ mkdir msg
$ cd msg
```

としてディレクトリを作成し，この msg ディレクトリの下に以下のファイルを NamedPoint.msg として保存する．

```
Header header
string name
geometry_msgs/Vector3 point
```

これで，CMakeLists.txt ファイルに catkin_package() と書いてある行より前に以下の行を加えて catkin build するとこのメッセージに対応した C++, Python 等のコードが自動生成される．

```
add_message_files(FILES NamedPoint.msg)
generate_messages(DEPENDENCIES geometry_msgs)
```

さらに，scripts ディレクトリ以下の listener.py, src ディレクトリ以下に talker.cpp を作成すれば，独自に定義したメッセージをつかった通信が可能になる．

```
#!/usr/bin/env python

import rospy
from beginner_tutorials.msg import NamedPoint

def callback(msg):
    rospy.loginfo("I heard {} x:{}, y:{}, z:{}".format(msg.name,
                                msg.point.x, msg.point.y, msg.point.z))

def listener():
    rospy.init_node('listener')
    rospy.Subscriber('chatter', NamedPoint, callback)
    rospy.spin()

if __name__ == '__main__':
    try:
        listener()
    except rospy.ROSInterruptException: pass
```

```
#include <ros/ros.h>
#include <beginner_tutorials/NamedPoint.h>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
            n.advertise<beginner_tutorials::NamedPoint>("chatter", 1000);

  ros::Rate loop_rate(10);

  int count = 0;
  while (ros::ok())
  {
    beginner_tutorials::NamedPoint msg;

    msg.name = "Hello world";
    msg.point.x = 1;
    msg.point.y = 2;
    msg.point.z = 3;

    ROS_INFO_STREAM(msg);
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

listener.cpp に加えて talker.cpp が追加されたので

```
add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
```

を CMakeLists.txt に追加し catkin build しなおす必要がある.

3 ROS サンプルプログラム

3.1 ROS 自律移動

ナビゲーション (自律移動) プログラムのサンプルは以下のようにして実行します．https://www.youtube.com/watch?v=qziUJcUDfBc に解説ビデオがあります．

3.1.1 自律移動デモプログラム実行

自律移動デモプログラムは以下のようにしてインストールする．

2022 年 7 月時点では自律移動デモプログラムは Ubuntu の deb ファイルとしてリリースされていないので，ソースコードからインストールする必要がある．リリースの手続きについては https://github.com/ros-gbp/navigation_tutorials-release/issues/2 で議論を始めている．

ソースコードのダウンロード，並びにコンパイルは以下のように行う．B 節とは wstool を実行するディレクトリが異なるが，-t src を付けている．

```
$ mkdir ~/navigation_ws
$ cd  ~/navigation_ws
$ source /opt/ros/noetic/setup.bash
$ wstool init src
$ wstool set -y -t src navigation_tutorials --git http://github.com/ros-planning/navigation_tutorials
$ wstool update -t src navigation_tutorials
$ rosdep install --from-paths src/navigation_tutorials --ignore-src -y
$ catkin build navigation_stage
$ source devel/setup.bash
```

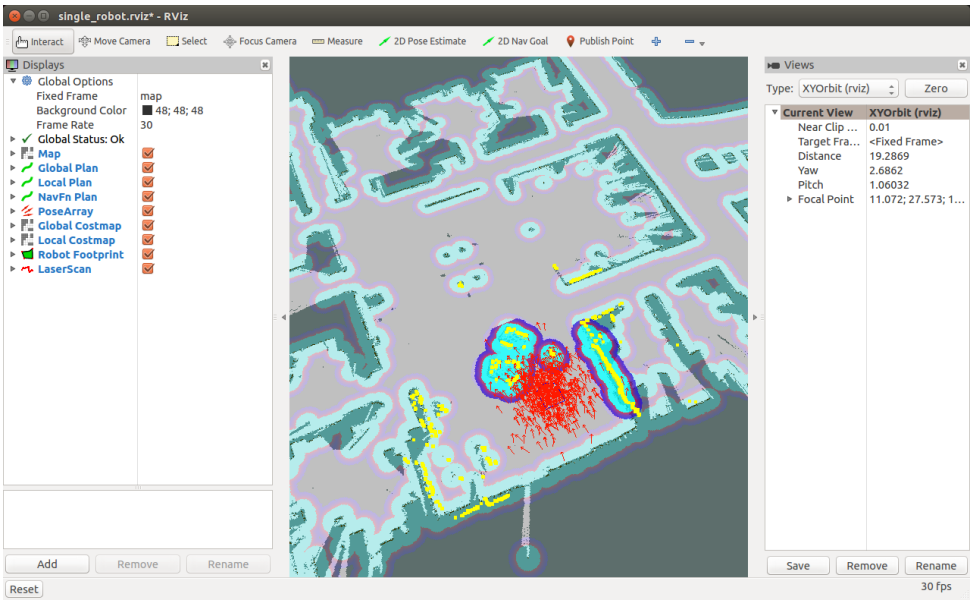
Ubuntu の deb ファイルとしてリリースされれば以下の 1 行でよい．

```
$ sudo apt-get install ros-noetic-navigation-stage
```

サンプルプログラムの起ち上げは以下のように行う．

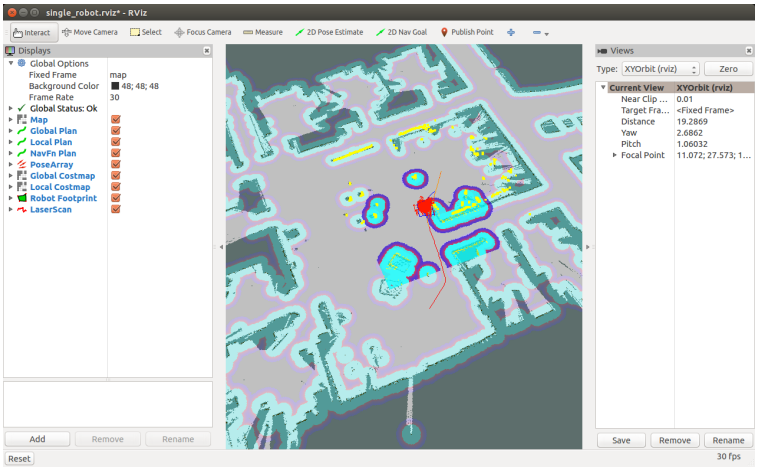
```
$ roslaunch navigation_stage move_base_amcl_2.5cm.launch
```

サンプルプログラムを立ち上げると以下のような表示になります．

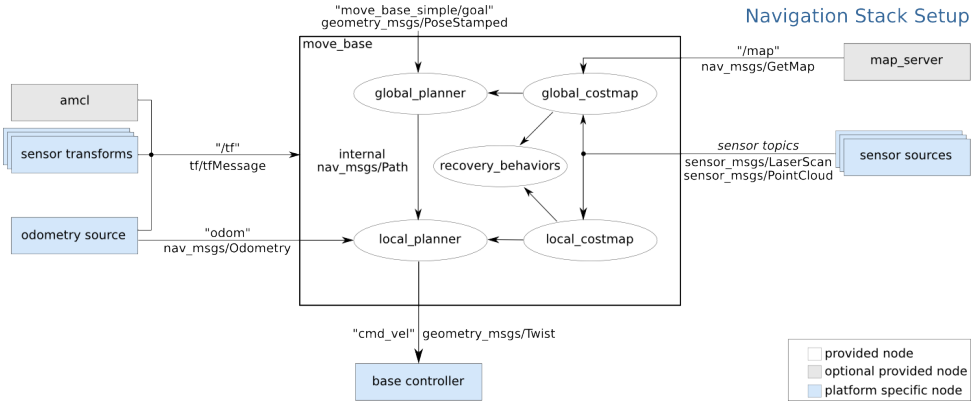


赤い矢印群がロボットの位置姿勢の確率的な表現（パーティクル）になり，赤色の中心にはロボットの姿勢を示す青い線が描かれています．黄色い点群がロボットのレーザセンサの表示，薄い水色の部分があらかじめ与えられた地図と，その地図から計算される障害物を示すコストマップ，濃い水色はセンサ情報から動的に計算されるコストマップになる．

自律移動プログラムに対して目標位置姿勢を設定するには，上部の「2D Nav Goal」と書かれたアイコンをクリックした後，地図上の目標位置をクリックし，さらにドラッグすることで目標姿勢を指示する．経路，あるいは，目的地に障害物がある場合はプランニングが失敗すますが，それ以外の場合は以下のようにロボットが動き出す．



この時，赤い線がロボットの大域的な経路を示し，青い線でロボットの局所的な移動経路を表示している．また，ロボットが移動するに従って，赤色の矢印群が収束し，より正確な位置姿勢が推定されている事がわかる．



出典 http://wiki.ros.org/navigation/Tutorials/RobotSetup

この自律移動システムは主に amcl と move_base という 2 つのプログラムから構成される．このシステムの概要を上の図に示した．amcl はロボットのセンサ情報と地図情報からロボットの位置姿勢を推定するプログラムであり，move_base は与えられた目標位置姿勢と地図を用いて障害物を回避しながら目標位置へ到達する経路とそのための制御目標値を計算する．

目標位置姿勢は geometry_msgs/PoseStamped 型の move_base_simple/goal トピックで与えられ，地図は nav_msgs/OccupancyGrid 型の /map トピック，または，nav_msgs/GetMap 型の /static_map サービスで与えられる．

制御目標値は geometry_msgs/Twist 型の cmd_vel という名前のトピックで速度指令と出力される．

3.1.2 ロボット速度指令プログラム

cmd_vel は ROS で動く移動ロボットの標準的なトピック名であり，ほぼ全ての ROS 対応ロボットはここに geometry_msgs/Twist 型で速度指令を送ってロボットが動くように構成されています．

キーボードから cmd_vel を出力するプログラムは teleop_twist_keyboard パッケージで提供されおり，以下のように実行できる．

```
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

ここでは，'i'，','キーで前進後進，'l'，'j'キーで回転移動，'L'，'J'キーで左右移動を指示できる．またプログラムを終了するには C-c キーを押す．

プログラムソースコードを確認したい場合は以下のようなコマンドを利用できる．

```
$ roscat teleop_twist_keyboard teleop_twist_keyboard.py
```

また，出力されているトピックの内容は以下のように確認できる．

```
$ rostopic echo /cmd_vel
```

teleop_twist_keyboard パッケージでインストールされていない場合は以下のようにしてインストールできる．

```
$ sudo apt-get install ros-noetic-teleop-twist-keyboard
```

3.1.3 ロボット目標位置指令プログラム (topic 版)

まずは GUI でロボットの目標位置姿勢を与えた際にどのようなトピックがパブリッシュされているか確認する．

rviz 上部の「2D Nav Goal」と書かれたアイコンをクリックすると，move_base_simple/goal トピック出力される．これで，ロボットに対して目標位置姿勢を与えらている．実際にパブリッシュされているトピックは以下のように確認できる．

```
$  rostopic echo move_base_simple/goal
header:
  seq: 4
  stamp:
    secs: 2830
    nsecs: 100000000
  frame_id: "map"
pose:
  position:
    x: 18.5897636414
    y: 11.2394886017
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: -0.674089865744
    w: 0.738649343668
```

pose でロボットの目標姿勢を指定している．frame_id はこの目標姿勢の座標原点となるフレームを表している．

例えば，Rviz の左側の Displays の欄の Fixed Frame の map を base_link に変更して，上部の「2D Nav Goal」と書かれたアイコンをクリックして目標位置を指定すると，base_link 相対の目標位置姿勢がパブリッシュされていることがわかる．

ROS ではフレームは tf と呼ばれる仕組みで管理されている．現在パブリッシュされているフレームを確認するには Displays の下の Add ボタンを押し，TF を選択する．これで rviz 上に tf が表示される．

tf を使った座標系の扱いは ROS の基本部分なので，以下の情報を参考に是非マスターして欲しい．

http://wiki.ros.org/tf

http://wiki.ros.org/tf/Tutorials

move_base_simple/goal をパブリッシュするプログラム (send_goal_topic.py) は以下のようにかける．

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import rospy
from geometry_msgs.msg import PoseStamped

if __name__ == '__main__':
    try:
        rospy.init_node('send_goal_topic', anonymous=True)
        pub = rospy.Publisher('move_base_simple/goal', PoseStamped, queue_size=1)
        rospy.sleep(1)

        goal = PoseStamped()
        goal.header.frame_id = 'map' # map フレーム相対で目標位置を指定
        goal.pose.position.x=41;
        goal.pose.position.y=17;
        goal.pose.orientation.w = 1
        rospy.loginfo("send goal :")
        rospy.loginfo(goal)
        pub.publish(goal)
    except rospy.ROSInterruptException: pass
```

ROS では回転の表現に Quaternion(四元数) が多様される．Quaternion の計算は直感的ではないが，以下のような変換ツールが利用できる．

```
$ python
>>> import math
>>> from tf import transformations
>>> transformations.quaternion_from_euler(0,0,0)
array([ 0.,  0.,  0.,  1.])
>>> transformations.quaternion_from_euler(0,0,math.pi/2)
array([ 0.        ,  0.        ,  0.70710678,  0.70710678])
```

Quaternion 関係の関数は以下にまとめられているので，活用すると良い．

`/opt/ros/noetic/lib/python3/dist-packages/tf/transformations.py`

3.1.4 ロボット目標位置指令プログラム (ActionLib 版)

自律移動の目標位置を送る actionlib プログラム (`send_goal_actionlib.py`) は以下になる．

Actionlib の説明や具体的なプログラムの書き方は `http://wiki.ros.org/actionlib` を参照されたい．

```
#!/usr/bin/env python

import rospy, actionlib
from move_base_msgs.msg import *

if __name__ == '__main__':
    try:
        rospy.init_node('send_goal', anonymous=True)
        client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
        client.wait_for_server() # ActionLib のサーバと通信が接続されることを確認
        goal = MoveBaseGoal()
        goal.target_pose.header.stamp = rospy.Time.now()
        goal.target_pose.header.frame_id = 'map'
        goal.target_pose.pose.position.x=34;
        goal.target_pose.pose.position.y=32;
        goal.target_pose.pose.orientation.w = 1
        rospy.loginfo("send goal")
        rospy.loginfo(goal)
        client.send_goal(goal)  # 目標位置姿勢を goal として送信
        rospy.loginfo("wait for goal ...")
        ret = client.wait_for_result() # ロボットが目標位置姿勢に到達するまで待つ
        rospy.loginfo("done")
    except rospy.ROSInterruptException: pass
```

3.2 ROS 人型ロボット制御

3.2.1 等身大ロボットデモプログラム実行

等身大ロボットのシミュレーションプログラムは以下のようにしてインストールする.

```
$ mkdir ~/gundam_ws
$ cd  ~/gundam_ws
$ source /opt/ros/noetic/setup.bash
$ wstool init src
$ wstool set -y -t src gundam_robot --git http://github.com/gundam-global-challenge/gundam_robot
$ wstool update -t src gundam_robot
$ rosdep install --from-paths src/gundam_robot --ignore-src -y
$ catkin build gundam_robot
$ source devel/setup.bash
```

また,

```
$ rosrun rqt_joint_trajectory_controller rqt_joint_trajectory_controller
  [rospack] Error: package 'rqt_joint_trajectory_controller' not found
```

と，エラーが出る場合は

```
$ sudo apt install ros-noetic-rqt-joint-trajectory-controller
```

としてインストールすること.

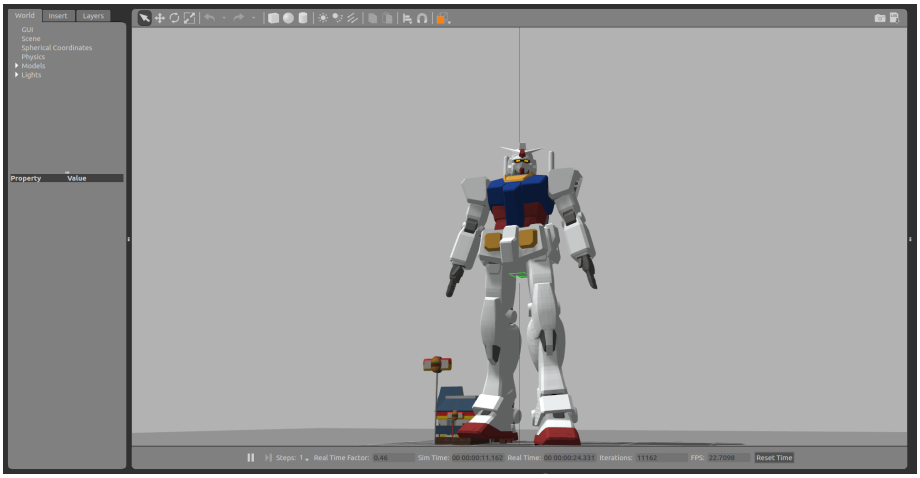
Ubuntu の deb ファイルとしてリリースされれば以下の 1 行でよい.

```
$ sudo apt-get install ros-noetic-gundam-robot
```

サンプルプログラムの起ち上げは以下のように行う.

```
$ roslaunch gundam_rx78_gazebo gundam_rx78_world.launch
```

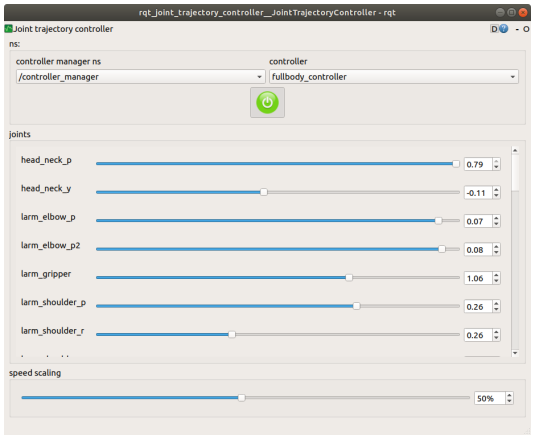
サンプルプログラムを立ち上げると以下のような表示になる.



ロボットは Gazebo と呼ばれる物理計算を含んだシミュレータ上で稼働している．すなわち，バランスの悪い姿勢を取ると，倒れてしまう．

```
$ rosrun rqt_joint_trajectory_controller rqt_joint_trajectory_controller
```

としてロボットの関節角度を指令する GUI プログラムを立ち上げ，controller manager ns，controller を適切に選択し，赤色のボタンを押すと緑となり，各関節の位置を指令するスライドバーが表示される．これを制御することでロボットの関節位置を変更することができる．



3.2.2 ロボットの関節位置指令プログラム (topic 版)

まずは GUI でロボットの各関節の目標位置を与えた際にどのようなトピックがパブリッシュされているか確認する．

実際にパブリッシュされているトピックは以下のように確認できる．

```
$ rostopic echo /fullbody_controller/command
---
header:
  seq: 3200
  stamp:
    secs: 0
    nsecs:         0
  frame_id: ''
joint_names: [head_neck_p, head_neck_y, larm_elbow_p, larm_elbow_p2, larm_gripper, larm_shoulder_p,
  larm_shoulder_r, larm_shoulder_y, larm_wrist_r, larm_wrist_y, lleg_ankle_p, lleg_ankle_r,
  lleg_crotch_p, lleg_crotch_r, lleg_crotch_y, lleg_knee_p, lleg_knee_p2, rarm_elbow_p,
  rarm_elbow_p2, rarm_gripper, rarm_shoulder_p, rarm_shoulder_r, rarm_shoulder_y,
  rarm_wrist_r, rarm_wrist_y, rleg_ankle_p, rleg_ankle_r, rleg_crotch_p, rleg_crotch_r,
  rleg_crotch_y, rleg_knee_p, rleg_knee_p2, torso_lthrust_p, torso_lthrust_r, torso_rthrust_p,
  torso_rthrust_r, torso_waist_p, torso_waist_p2, torso_waist_y]
points:
  -
    positions: [0.03612831551605, 0.15707963267899983, 0.07, 0.08, 1.06, 0.26, 0.26, -0.24, 0.18,
       -0.04, -0.07, 0.01, -0.48, 0.2, -0.2, -0.12, 1.05, -0.06, 0.13, 1.05, 0.51, -0.08, -0.03,
        -0.26, 0.01, -0.37, 0.12, 0.52, 0.15, -0.5, -0.12, 0.44, 0.0, -0.0, 0.01, -0.0, 0.52, -0.75, -0.26]
    velocities: []
    accelerations: []
    effort: []
    time_from_start:
      secs: 1
      nsecs:         0
---
```

ここでは，joint_names に関節の名前を，positions に関節の角度を，time_from_start のその関節位置まで移動する時間を指定している．

/fullbody_controller/command をパブリッシュするプログラムは以下のようにかける．

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import rospy
from trajectory_msgs.msg import JointTrajectory, JointTrajectoryPoint

if __name__ == '__main__':
    try:
        rospy.init_node('joint_trajectory_example', anonymous=True)
        pub = rospy.Publisher('/fullbody_controller/command', JointTrajectory, queue_size=1)
        rospy.sleep(1)

        joint = JointTrajectory()
        joint.joint_names = ['head_neck_p', 'head_neck_y']
        print(joint)
        print(joint.points)
        joint.points.append(JointTrajectoryPoint(positions=[-1, 1], time_from_start=rospy.Duration(1.0)))
        joint.points.append(JointTrajectoryPoint(positions=[0, -1], time_from_start=rospy.Duration(2.0)))

        rospy.loginfo("send pose :")
        rospy.loginfo(joint)
        pub.publish(joint)

    except rospy.ROSInterruptException: pass
```

rqt_joint_trajectory_controller が立ち上がっているとその出力もパブリッシュされており，自ら書いたプログラムの効果がわかりづらいので，rqt_joint_trajectory_controller を停止しておくこと．

3.2.3 ロボットの関節位置指令プログラム (ActionLib 版)

ロボットの関節目標位置を送る actionlib プログラムは以下のように書けるが，プログラム自身は roscd gundam_rx78_control/sample/として移動できるディレクトリの中の joint_trajectory_client_example.py にある．

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import rospy
import actionlib
import sys
import time

from control_msgs.msg import (
    FollowJointTrajectoryAction,
    FollowJointTrajectoryGoal,
)
from trajectory_msgs.msg import (
    JointTrajectoryPoint,
)


def main():
    print("Initializing node... ")
    rospy.init_node("joint_trajectory_client_example")
    rospy.sleep(1)
    print("Running. Ctrl-c to quit")
    positions = {
        'head_neck_p': 0.0, 'head_neck_y': 0.2,
        'larm_shoulder_p': 0.1, 'larm_shoulder_r':  0.3, 'larm_shoulder_y': -0.1,
        'larm_elbow_p': -0.2, 'larm_elbow_p2': -0.2,
        'larm_wrist_r': 0, 'larm_wrist_y': 0, 'larm_gripper': 1.1,
        'rarm_shoulder_p': 0.1, 'rarm_shoulder_r': -0.3, 'rarm_shoulder_y': 0.1,
        'rarm_elbow_p': -0.2, 'rarm_elbow_p2': -0.2,
        'rarm_wrist_r': 0, 'rarm_wrist_y': 0, 'rarm_gripper': 1.1,
        'torso_waist_p': -0.05, 'torso_waist_y': 0.0, 'torso_rthrust_p': 0.0,
        'torso_rthrust_r': 0.0, 'torso_lthrust_p': 0.0, 'torso_lthrust_r': 0.0,
        'lleg_crotch_p': -0.35, 'lleg_crotch_r':  0.2, 'lleg_crotch_y':   0.35,
        'lleg_knee_p':  0.20, 'lleg_knee_p2':  0.20, 'lleg_ankle_p':  0.05, 'lleg_ankle_r': -0.05,
        'rleg_crotch_p':  0.20, 'rleg_crotch_r': -0.1, 'rleg_crotch_y': -0.15,
        'rleg_knee_p':  0.05, 'rleg_knee_p2':  0.05, 'rleg_ankle_p': -0.2, 'rleg_ankle_r':  0.1,
    }
    client = actionlib.SimpleActionClient(
        '/fullbody_controller/follow_joint_trajectory',
        FollowJointTrajectoryAction,
    )

    if not client.wait_for_server(timeout=rospy.Duration(10)):
        rospy.logerr("Timed out waiting for Action Server")
        rospy.signal_shutdown("Timed out waiting for Action Server")
        sys.exit(1)
```

```
    # init goal
    goal = FollowJointTrajectoryGoal()
    goal.goal_time_tolerance = rospy.Time(1)
    goal.trajectory.joint_names = positions.keys()

    # points
    point = JointTrajectoryPoint()
    goal.trajectory.joint_names = positions.keys()
    point.positions = positions.values()
    point.time_from_start = rospy.Duration(10)
    goal.trajectory.points.append(point)

    point = JointTrajectoryPoint()
    positions['torso_waist_p']   +=  0.2
    positions['torso_waist_y']   +=  0.2
    positions['head_neck_p']      =  0.05
    positions['head_neck_y']      =  0.15
    positions['lleg_crotch_p']   += -0.1
    positions['rleg_crotch_p']   += -0.1
    positions['rarm_shoulder_p'] +=  0.2
    positions['larm_shoulder_p'] +=  0.2
    positions['rarm_elbow_p']    += -0.2
    positions['rarm_elbow_p2']    += -0.2
    positions['larm_elbow_p']    += -0.2
    positions['larm_elbow_p2']    += -0.2
    point.positions  = positions.values()
    point.time_from_start = rospy.Duration(12)
    goal.trajectory.points.append(point)

    # send goal
    goal.trajectory.header.stamp = rospy.Time.now()
    client.send_goal(goal)
    print(goal)
    print("waiting...")
    if not client.wait_for_result(timeout=rospy.Duration(20)):
        rospy.logerr("Timed out waiting for JTA")
    rospy.loginfo("Exitting...")

if __name__ == "__main__":
    main()
```

より詳細なプログラムは https://github.com/gundam-global-challenge/gundam_robot も参照されたい.

4 宿題

提出先：ITC-LMS を用いて提出すること

提出内容：以下の問題の実行結果の画面をキャプチャしファイル名は「問題番号.png」とし，また講義中にでてきたキーワードについて知らなかったもの, 興味のあるものを調べ「学籍番号.txt」としててアップロードすること．テキストファイルはワードファイルなどだと確認出来ないことがあるため，emacs/vi 等のテキストエディタを使って書こう．プログラムが長くなりキャプチャ画面に入り切らなくなってきたらプログラムファイルと実行結果を「問題番号.txt」にまとめてアップロードしてよい．
画像で提出する場合は，各自のマシンの Mac アドレスが分かるようにすること．例えば画面中に ifconfig というコマンドを打ち込んだターミナルを表示すればよい.

ITC-LMS にアップロードする際には講義・宿題の感想を必ずコメントに記すこと．また授業中に質問した者はその旨を記すこと．質問は成績評価時の加点対象となる．

キーワード : ソフトウェアライセンス（GNU と BSD), OSI によるオープンソスの定義，GitHub の Issue と Pull Request

1. ROS サンプルプログラムで紹介されている自律移動ロボットか等身大ロボットのサンプルを実行し画面をキャプチャせよ．あるいは他のロボットのシミュレータを動かせたものはその画面もキャプチャせよ．
2. （発展課題）自分で msg ファイルを新たに定義し，これを用いてデータを送受信するプログラムを作成せよ．プログラム言語は何でも良い．

質問の仕方のガイドラインは http://wiki.ros.org/Support も参考になる．

A GitHub

github はプログラムの共同開発用の web サービスであり，git と呼ばれる分散ソースコード管理システムのホスティングを提供している．その一番の特徴は Pull Request という仕組みであろう．これはソースコードのメンテナ以外のユーザが，オリジナルのソースコードに対する差分を upload し，メンテナがその内容をレビューし，オリジナルのソースコードに取り込むべきものであると判断した場合にこれをマージするものである

github の使い方は `https://guides.github.com/activities/hello-world/` や `http://rogerdudler.github.io/git-guide/index.ja.html` 等に基礎的な情報がある．また、`https://help.github.com/` にドキュメントがまとまっている

さらに A4 にまとめたものが `https://services.github.com/kit/downloads/ja/github-git-cheat-sheet.pdf` や `https://github.com/tork-a/cheatsheet/releases/download/20160116/GITcheatsheet.pdf` にある．

最初はわからないことも多いと思うが、何回も練習して必ず身につけよう．

B 演習用ロボットプログラム環境の構築

機械情報冬学期の演習用ロボットプログラム環境の構築は以下のように行う．

```
$ sudo apt-get update
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ source /opt/ros/noetic/setup.bash
$ git clone https://github.com/jsk-enshu/robot-programming
$ wstool init .
$ wstool merge robot-programming/.rosinstall.noetic
$ wstool update
$ rosdep update
$ rosdep install --from-paths . --ignore-src -y -r
$ cd ..
$ catkin build # 最初の 1 回は必ず~/catkin_ws あるいは src/ が置かれているディレクトリで実行すること．
```

としてワークスペースを作る．

また、以下のようにして自分のフォークしたレポジトリを登録しておく

```
$ cd ~/catkin_ws/src/robot-programming
$ git remote add <yourname> http://github.com/<yourname>/robot-programming
# <yourname> は自分の github のアカウント.
# 例えば git remote add k-okada http://github.com/k-okada/robot-programming
$ git fetch --all
```

最後に以下のようにして演習用ロボットのシミュレーションが可能になる.

```
$ source ~/catkin_ws/devel/setup.bash
$ roslaunch dxl_armed_turtlebot dxl_armed_turtlebot_gazebo.launch
```

これで，gazebo と呼ばれる動力学シミュレータが起ち上がり，演習ロボットが表示される.

また、以下のようにして rqt を使ったロボットの制御ツールを立ち上げる.

```
$ roscd dxl_armed_turltebot/launch
$ rqt --perspective-file enshu.perspective
```

ここで、ロボットの台車の速度指令、アームの位置（角度）指令が可能になる．アームを司令する場合は controller で fullbody_controller を司令し，更に赤い電源ボタンを押して緑になることを確認してから，スライダを動かしてみよ.

次に、

```
$ roslaunch opencv_apps camshift.launch image:=/camera/rgb/image_raw
```

として、画像処理ノードを立ち上げる．赤いボールが見えたら Window で追跡したい領域を囲えば良い.

結果は、

```
$ rostopic echo /camshift/track_box
```

として見ることができる．さらに、結果を視覚化するために

```
$ rosrun image_view2 image_view2 image:=/camera/rgb/image_raw
         ~image_transport:=compressed
```

として viewer を立ち上げて、以下のプログラムを実行すれば良い.

```
#!/usr/bin/env python

import rospy
from opencv_apps.msg import RotatedRectStamped
from image_view2.msg import ImageMarker2
from geometry_msgs.msg import Point

def cb(msg):
    print msg.rect
    marker = ImageMarker2()
    marker.type = 0
    marker.position = Point(msg.rect.center.x, msg.rect.center.y, 0)
    pub.publish(marker)

rospy.init_node('client')
rospy.Subscriber('/camshift/track_box', RotatedRectStamped, cb)
pub = rospy.Publisher('/image_marker', ImageMarker2)
rospy.spin()
```

ソースコード自体は https://github.com/k-okada/robot-programming/tree/add_answers/dxl_armed_turtlebot/scripts の./track_box_to_image_marker.py にプログラムがある．

k-okada をリモートリポジトリに追加していれば，

```
$ git checkout add_answers
$ roscd dxl_armed_turtlebot/scripts
```

として，ファイルのある場所に移動できる．

B.1 ロボット台車の移動制御

ロボットの台車の移動速度の指令は/cmd_vel トピックが使われることが多い，このトピックは geometry_msgs/Twist メッセージであることが多いため，以下のようにして現在の geometry_msgs/Twist メッセージを利用しているトピック一覧を知ることができる．

```
$ rostopic find geometry_msgs/Twist
```

rqt の画面でロボットの台車の移動速度を指令し，それにより以下のトピックが変化することを確認することで，ロボットの制御方法を確認することができる．

```
$ rostopic echo /cmd_vel
```

また，その時のロボットの画像認識の結果は以下のトピックが変化することを確認することで，確認することができる．

```
$ rostopic echo /camshift/track_box
```

CamShift を使った画像処理は指定されたヒストグラムに対応する画像領域を出力する．ヒストグラムそのものは Histogram ウィンドウで確認できる．また，このときの結果は CamShift Demo ウィンドウで確認できる．CamShift Demo ウィンドウでマウスクリック＋マウスドラッグすることでトラッキングしたい領域を設定できる．CamShift Demo ウィンドウで'b' キーを押すことで，

指定したヒストグラムに対応する画像領域が確認できる．楕円形の出力はこの二値画像をラベリングし，最大領域を楕円で表していることになる．

対象物の方向に必ず向くようなロボットの制御プログラムの書き方は幾つか考えられる．

1つめは/camshift/track_box が届くたびに/cmd_vel を出力する方式である．

```
def cb(msg):
    cmd_vel = Twist()
    ## 画像処理の結果 (msg) に応じて cmd_vel を計算
    (省略)
    pub.publish(cmd_vel)

if __name__ == '__main__':
    try:
        rospy.init_node('client')
        rospy.Subscriber('/camshift/track_box', RotatedRectStamped, cb)
        pub = rospy.Publisher('/cmd_vel', Twist)
        ## メインループに入る
        rospy.spin()
    except rospy.ROSInterruptException: pass # エラーハンドリング
```

画像処理プログラムの仕様に依存するが，周期的に結果が得られない場合，あるいは対象物が認識出来ない／存在しない場合に結果が Publish されない場合，cmd_vel を出力しなくなる．例えば対象物が存在しない場合はロボットは停止して欲しい，あるいは，対象物の探索行動をして欲しい場合などは都合が悪い事がある．

次に考えられるのは，メイン文内でループを回し，サブスクライバとメイン文内のループの間での情報共有を大域変数を介して行う，以下のような方法である．

```
rect = RotatedRectStamped() ## 大域変数として定義
def cb(msg):
    global rect  ## 大域変数の利用を宣言
    ## 画像処理の結果 (msg) を大域変数 rect に登録
    rect = msg

if __name__ == '__main__':
    try:
        rospy.init_node('client')
        rospy.Subscriber('/camshift/track_box', RotatedRectStamped, cb)
        pub = rospy.Publisher('/cmd_vel', Twist)
        rate = rospy.Rate(10)
        while not rospy.is_shutdown(): ## 無限ループに入る
            cmd_vel = Twist()
            ## 大域変数 rect に応じて cmd_vel を計算.
            ## 認識結果の領域の中心の x 座標が画像の半分より左であれば左回転する
            ## ロボットの回転指令は cmd_vel.angular.z で与えられる
            ##
            ## 考えられるエラーチェック例
            ## ・大域変数 rect に古い画像処理結果が入っている場合
            ## ・対象物が見つかっていない場合 など
            pub.publish(cmd_vel)
            rate.sleep()
    except rospy.ROSInterruptException: pass # エラーハンドリング
```

このように大域変数を使う場合はクラスを定義して実装することが可能になる．

```
class track_box_to_cmd_vel:
    rect = None  ## メンバ変数として定義
    pub = None
    rate = None
    def __init__(self):
        self.rect = RotatedRectStamped()
        rospy.init_node('client')
        rospy.Subscriber('/camshift/track_box', RotatedRectStamped, self.cb)
        self.pub = rospy.Publisher('/cmd_vel', Twist)

    def cb(self, msg):
        self.rect = msg       ## 画像処理の結果 (msg) を大域変数 rect に登録

    def loop(self):
        rate = rospy.Rate(10)
        while not rospy.is_shutdown():
            cmd_vel = Twist() ## 大域変数 rect に応じて cmd_vel を計算.
            (省略)
            self.pub.publish(cmd_vel)
            rate.sleep()

# track_box_to_cmd_vel オブジェクトを生成
obj = track_box_to_cmd_vel()
# obj.loop() メンバ関数内で無限ループとなる.
obj.loop()
# あるいは,
# rate = rospy.Rate(10)
# while not rospy.is_shutdown():
#     obj.loopOnce()
#     rate.sleep()
# のような書き方も考えられる
```

また rospy.Timer(http://wiki.ros.org/rospy/Overview/Time#Timer) を使って定期的に関数を呼び出すことも可能である．

```
class track_box_to_cmd_vel:
    (省略)
    def __init__(self):
        (省略)
        rospy.Timer(rospy.Duration(0.1), self.loopOnce)

    def cb(self, msg):
        self.rect = msg    ## 画像処理の結果 (msg) を大域変数 rect に登録

    def loopOnce(self, event):
        cmd_vel = Twist()  ## 大域変数 rect に応じて cmd_vel を計算.
        (省略)
        self.pub.publish(cmd_vel)

# track_box_to_cmd_vel インスタンスを生成
obj = track_box_to_cmd_vel()
# イベントドリブン型のプログラムにおけるイベントループ
```

何れの場合も以下のようなコードをいれて，プログラムの振る舞いを確認できるようにしておく

ことが重要である．

```
rospy.loginfo("area = {}, center = ({}, {})".format(area, msg.rect.center.x, msg.rect.center.y))
rospy.loginfo("\t\t\t\t\tpublish {}".format(cmd_vel.angular.z))
# \t（タブ）により'area = ' と'publish ..' の表示が混在しても見やすくしている．
```

実際のプログラムは https://github.com/k-okada/robot-programming/tree/add_answers/dxl_armed_turtlebot/scripts 以下にある track_box_to_cmd_vel_1.py, track_box_to_cmd_vel_2.py, track_box_to_cmd_vel_3.py を参考にすることができる．

B.2 ロボットアームの角度制御

ロボットのアームの角度指令の方法は幾つかあるが，rqt からの指令では fullbody_controller を利用している．これは以下のように確認できる．

```
rostopic echo /fullbody_controlr/command
```

このトピックのメッセージ型は以下のように調べることが出来る．

```
$ rostopic type /fullbody_controller/command
trajectory_msgs/JointTrajectory
$ rostopic type /fullbody_controller/command | rosmsg show
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string[] joint_names  # 関節名のリスト
trajectory_msgs/JointTrajectoryPoint[] points
  float64[] positions      # 各関節角度の位置指令
  float64[] velocities     # 各関節角度の速度指令
  float64[] accelerations  # 各関節角度の加速度指令
  float64[] effort         # 各関節角度のトルク指令
  duration time_from_start # 上記指令に到達する時刻の指令
```

関節名を知る方法は，以下のようにロボットの関節情報を確認すると良い．/joint_states も標準的なトピックの一つであり，殆どのロボットでパブリッシュされていると期待して良い．

```
rostopic echo /joint_states
```

また，以下のようにして joint_states の中の name 要素だけを echo する事が出来る．

```
rostopic echo /joint_states/name
```

関節名が分かれば，これを使って JointTrajectory を作成する．joint_trajectory.points は 各時間の指令値のリストになる．append(point) では point の値はコピーされないので，

```
    point = JointTrajectoryPoint()
    for i in range(5):
        point.positions = [math.pi/2, 0, math.pi/4*(i%2), 0, math.pi/2, math.pi/2]
        point.time_from_start = rospy.Duration(1.0+i)
        joint_trajectory.points.append(point)
```

のように値を変えて append() しても，希望通りの動きにはならない．

以下のプログラムを実行すればロボットのアームが動く様子が gazebo で観察できるはずである．

なお，rqt の電源ボタンが緑になっている場合は rqt から定期的に関節指令が送られるため，これを停止させる必要がある．ボタンをおして赤になることを確認すること．

無理な姿勢を指示すると動力学計算が破綻することがあり，ロボットが暴走する．その場合は gazebo シミュレータの起ち上げから行うこと．

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import math
import rospy
from trajectory_msgs.msg import JointTrajectory
from trajectory_msgs.msg import JointTrajectoryPoint

def send_joint_position():
    # rosnode の初期化
    rospy.init_node('send_joint_position')
    # トピック名，メッセージ型を使ってパブリッシャを定義
    pub = rospy.Publisher('/fullbody_controller/command', JointTrajectory, queue_size=1)

    ## publihser がサブスクライバと接続するのを待つ
    ## これをしないと接続する前に pub.pusblish() が呼ばれる
    ## 簡単には rospy.sleep(1) などでもよい．
    rospy.sleep(1)

    ## 出力用メッセージを作成
    joint_trajectory = JointTrajectory()
    joint_trajectory.header.stamp = rospy.Time.now()
    joint_trajectory.joint_names = ['arm_joint1', 'arm_joint2', 'arm_joint3',
                                    'arm_joint4', 'arm_joint5', 'arm_joint6']
    for i in range(5):
        point = JointTrajectoryPoint()
        point.positions = [math.pi/2, 0, math.pi/4*(i%2), 0, math.pi/2, math.pi/2]
        point.time_from_start = rospy.Duration(1.0+i)
        joint_trajectory.points.append(point)

    ## メッセージをパブリッシュ
    pub.publish(joint_trajectory)

    ## 動作終了を待つ
    rospy.sleep(5)

if __name__ == '__main__': # メイン文．
    try:
        send_joint_position()
    except rospy.ROSInterruptException: pass # エラーハンドリング
```

B.3 ActionLib を用いたロボットアームの角度制御

ActionLib を用いたロボットのアームの角度指令は，control_msgs/FollowJointTrajectoryActionGoal メッセージ型を利用することが一般的である．トピック名はロボット（システム）に依存するが，演習用ロボットでは/fullbody_controller/follow_joint_trajectory/goal になっている．

このメッセージ型を使っているトピックの一覧を知りたければ

```
$ rostopic find control_msgs/FollowJointTrajectoryActionGoal
```

とすると良い．

このメッセージ型の仕様の概要は以下になっている．基本的には B.2 節のトピックを用いたロボットアーム角度制御で利用した trajectory_msgs/JointTrajectory で司令する．

```
$ rosmsg show control_msgs/FollowJointTrajectoryGoal
trajectory_msgs/JointTrajectory trajectory
  std_msgs/Header header
  string[] joint_names
  trajectory_msgs/JointTrajectoryPoint[] points
control_msgs/JointTolerance[] path_tolerance
control_msgs/JointTolerance[] goal_tolerance
duration goal_time_tolerance
```

ロボットの関節目標位置を送る actionlib プログラム (send_joint_position_actionlib.py) は以下になる．

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import math
import rospy
import actionlib
from trajectory_msgs.msg import JointTrajectory
from trajectory_msgs.msg import JointTrajectoryPoint
from control_msgs.msg import FollowJointTrajectoryAction, FollowJointTrajectoryGoal

def send_joint_position_actionlib():
    # rosnode の初期化
    rospy.init_node('send_joint_position')
    # トピック名，メッセージ型を使って ActionLib client を定義
    client = actionlib.SimpleActionClient('/fullbody_controller/follow_joint_trajectory',
                                          FollowJointTrajectoryAction)
    client.wait_for_server() # ActionLib のサーバと通信が接続されることを確認

    # ActionLib client の goal を指定
    # http://wiki.ros.org/actionlib_tutorials/Tutorials の
    #   Writing a Simple Action Client (Python) を参照
    # __TOPIC_PREFIX__Action で actionlib.SimpleActionClient を初期化
    # ゴールオブジェクトは __TOPIC_PREFIX__Goal を使って生成
    goal = FollowJointTrajectoryGoal()
    goal.trajectory = JointTrajectory()
    goal.trajectory.header.stamp = rospy.Time.now()
    goal.trajectory.joint_names = ['arm_joint1', 'arm_joint2', 'arm_joint3',
                                   'arm_joint4', 'arm_joint5', 'arm_joint6']
    for i in range(5):
        point = JointTrajectoryPoint()
        point.positions = [math.pi/2, 0, math.pi/4*(i%2), 0, math.pi/2, math.pi/2]
        point.time_from_start = rospy.Duration(1.0+i)
        goal.trajectory.points.append(point)
```

```
    ## 目標姿勢をゴールとして送信
    client.send_goal(goal)
    rospy.loginfo("wait for goal ...")
    client.wait_for_result() # ロボットの動作が終わるまで待つ.
    rospy.loginfo("done")

if __name__ == '__main__': # メイン文.
    try:
        send_joint_position_actionlib()
    except rospy.ROSInterruptException: pass # エラーハンドリング
```

Actionlib の説明や具体的なプログラムの書き方は http://wiki.ros.org/actionlib を参照されたい.