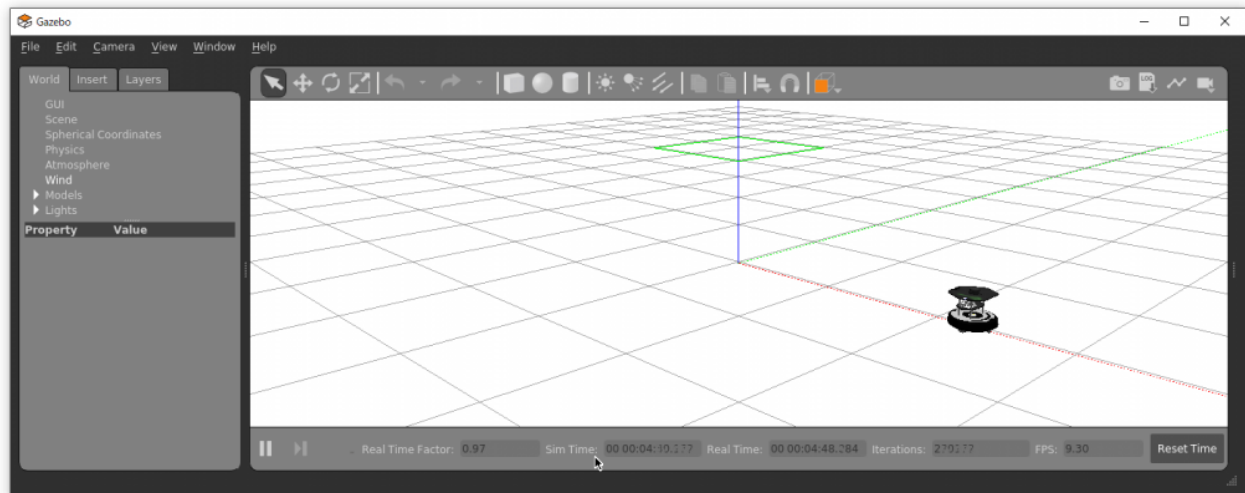


# HARD2021:ルンバの位置をPythonプログラムで知ろう！

 [demura.net/robot/hard/20085.html](http://demura.net/robot/hard/20085.html)

2021.03.11



このページではルンバの位置を知るためのROSを使ったPythonプログラムを学びます。ここでいう位置とはROSのcreate\_autonomyパッケージで計算した**odometry**(オドメトリ)情報です。オドメトリはロボットの車輪回転速度から移動量を求め自分の位置を計算する方法のことで、**デッドレコニング**ともいいます。

Pythonプログラムを読む前にROSの基本を簡単に説明します。これらの知識はプログラムを読むうえで頭に入れておく必要があります。まず、ROSでは**ノード**(実行中のプログラム)間でいろいろなデータを通信することで、ロボットを動作させています。

ROSの通信方式には**トピック通信**と**サービス通信**があり、**トピック通信**は1対多の非同期通信で、**メッセージ通信**は1体1の同期通信です。ここではトピック通信を学びます。

**トピック通信**では送り手と受け手が**Topic**(トピック)とよばれる名前付きのバス(伝送路)を介してデータをやり取りしています。送り手のことを**Publisher**(配信者)、受け手のことを**Subscriber**(購読者)、データのことを**メッセージ**とよびます。

オドメトリのトピックは/odomが一般的ですが、HARDワークショップで使っているcreate\_autonomyパッケージでは/create1/odomとなっています。

## ソースコード

では、さっそくソースコードを紹介します。たったの37行です。

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import rospy
from nav_msgs.msg import Odometry
import tf
from tf.transformations import euler_from_quaternion

_odom_x, _odom_y, _odom_theta = 0.0, 0.0, 0.0

def callback_odom(msg):
    global _odom_x, _odom_y, _odom_theta
    _odom_x = msg.pose.pose.position.x
    _odom_y = msg.pose.pose.position.y
    qx = msg.pose.pose.orientation.x
    qy = msg.pose.pose.orientation.y
    qz = msg.pose.pose.orientation.z
    qw = msg.pose.pose.orientation.w
    q = (qx, qy, qz, qw)
    e = euler_from_quaternion(q)
    _odom_theta = e[2]
    rospy.loginfo("Odometry: x=%s y=%s theta=%s", _odom_x, _odom_y, _odom_theta)

def odometry():
    rospy.init_node('odometry')
    odom_subscriber = rospy.Subscriber('/create1/odom', Odometry, callback_odom)
    rospy.spin()

if __name__ == '__main__':
    odometry()
```

## 説明付きソースコード

次に説明付きソースコードを読んでプログラムを理解しよう。なお、このプログラムは上で説明した**Subscriber** (購読者)のプログラム例となっています。

```

#!/usr/bin/env python
# -*- coding: utf-8 -*- # 日本語を使うためのおまじない。
import rospy # ROSでpython使う場合に必要
from nav_msgs.msg import Odometry # オドメトリを使う場合は必要
import tf
from tf.transformations import euler_from_quaternion

# グローバル変数。先頭のアンダーバーはグローバル変数の目印。
# 0.0で初期化している。
_odom_x, _odom_y, _odom_theta = 0.0, 0.0, 0.0

# callback_odom関数
def callback_odom(msg):
    global _odom_x, _odom_y, _odom_theta # グローバル変数
    _odom_x = msg.pose.pose.position.x # x座標 [m]
    _odom_y = msg.pose.pose.position.y # y座標 [m]
    # クォーターニオン(4次元数) 姿勢を表す手法の一つ
    qx = msg.pose.pose.orientation.x
    qy = msg.pose.pose.orientation.y
    qz = msg.pose.pose.orientation.z
    qw = msg.pose.pose.orientation.w
    q = (qx, qy, qz, qw)
    # クォータニオンからオイラー角を取得
    e = euler_from_quaternion(q)
    _odom_theta = e[2] # e[0]:ロール角, e[1]:ピッチ角, e[2]:ヨー角 [rad]

    rospy.loginfo("Odomery: x=%s y=%s theta=%s", _odom_x, _odom_y, _odom_theta)

# odometry関数
def odometry():
    # ノードの初期化。引数はノード名。ROSでは同じノード名のノードを複数作ることはいできない。
    rospy.init_node('odometry')

    # サブスクライバー(購読者)の生成。一番目の引数はトピック名、2番目の引数はメッセージの型、
    # オドメトリのメッセージはOdometry型。3番目の引数はコールバック関数。
    # 新しいメッセージが来るたびにコールバック関数が自動的に呼び出される。
    # これは以下のrospy.spin()とは別に並列で実行される。
    odom_subscriber = rospy.Subscriber('/create1/odom', Odometry, callback_odom)

    # spin()はノードが終了するまで、このプログラムを終わらせないようにしている。
    # これがないとプログラムはすぐ終了しノードも死んでしまう。
    rospy.spin()

# このプログラムをモジュールとしてimportできるようにするおまじない。
# なお、モジュールとは他のプログラムから再利用できるようにしたファイルのこと。
if __name__ == '__main__':
    odometry()

```

## 実行

## • パッケージの作成

- パッケージは次のcatkin\_create\_pkgコマンドで作ります。依存パッケージはそのパッケージを作るために必要なパッケージです。

```
catkin_create_pkg <パッケージ名> [依存パッケージ1] [依存パッケージ2] [依存パッケージ3]
```

- では、odometryパッケージを作っていきます！

- まず、odometryパッケージを格納するhard2021ディレクトリを作成。

```
$ mkdir -p ~/catkin_ws/src/hard2021
```

- hard2021ディレクトリに移動。

```
$ cd ~/catkin_ws/src/hard2021
```

- odometryパッケージの作成。依存パッケージとしてroscpp、rospy、std\_msgsを指定します。

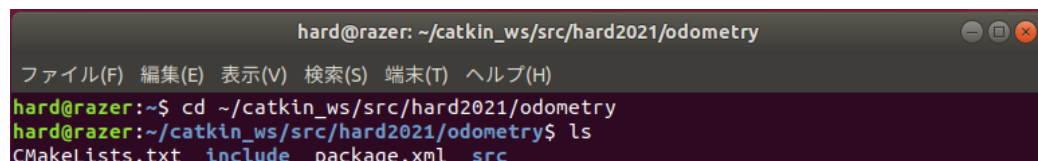
```
$ catkin_create_pkg odometry roscpp rospy std_msgs
```

- 上手く作成できたか確認しましょう。

- ```
$ cd ~/catkin_ws/src/hard2021/odometry
```

- ```
$ ls
```

- 下図のようにCMakeLists.txt, include, package.xml, srcができていれば成功。



```
hard@razer: ~/catkin_ws/src/hard2021/odometry
ファイル(F) 編集(E) 表示(V) 検索(S) 端末(T) ヘルプ(H)
hard@razer:~$ cd ~/catkin_ws/src/hard2021/odometry
hard@razer:~/catkin_ws/src/hard2021/odometry$ ls
CMakeLists.txt  include  package.xml  src
```

## • ソースコードの作成

エディタを開き、上の説明のついていない方のソースコードをコピーして保存する。以下のコマンドはエディタにgeditを使う場合。

- ```
$ cd ~/catkin_ws/src/hard2021/odometry/src
```

- ```
$ gedit odometry.py
```

## • ビルド

- Pythonはビルドする必要はないが、ROSで使用する場合は必要になる場合があるので、以下のコマンドを実行する。

- ```
$ cd ~/catkin_ws
```

- ```
$ catkin build odometry
```

- 作成したpythonプログラムに実行権を与える。実行権を与えないとroslaunchコマンドでノードを実行できない。

- ```
$ source ~/.bashrc
```

- ```
$ roscd odometry/src
```

- chmodはファイルのアクセス権を変更するコマンド。ここでは、このファイルをユーザが実行できるように変更している。

```
$ chmod u+x odometry.py
```

## ・実 行

### ◦ シミュレータの起動

端末を開き、以下のコマンドを実行してシミュレータGazeboを起動する。  
一番上図のルンバをベースにしたロボットが現れる。

```
$ roslaunch ca_gazebo create_empty_world.launch
```

### ◦ turtlebot\_teleop\_keyboardノードの実行

端末をもう一つ開き、以下のコマンドを実行してGazebo上のルンバをキー入力により遠隔制御する。

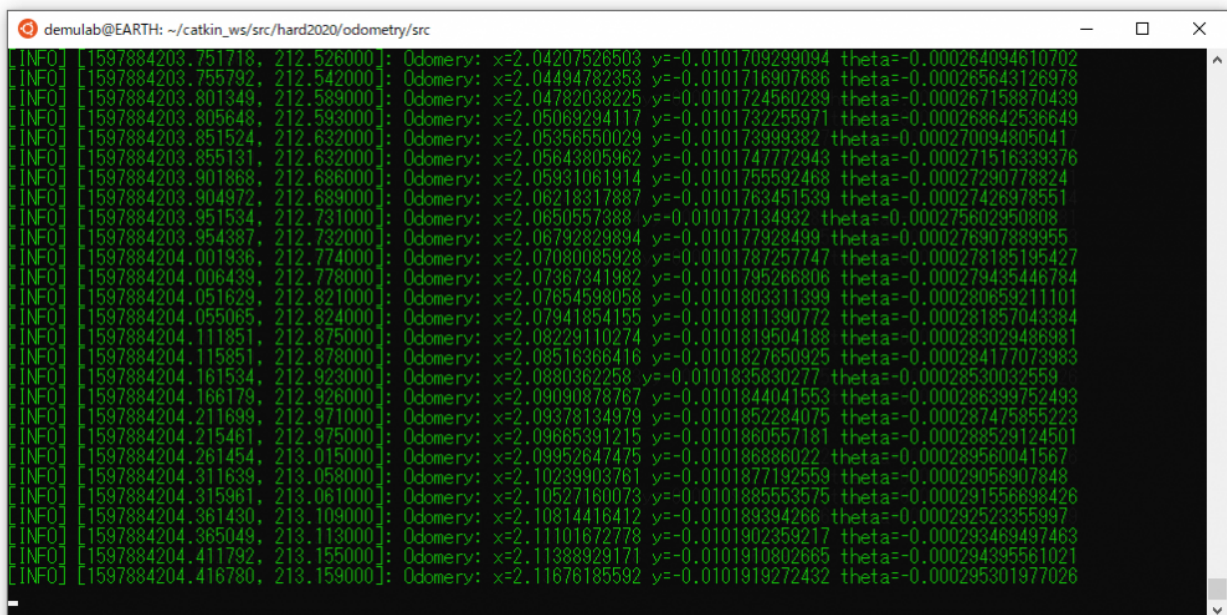
```
$ roslaunch ca_tools keyboard_teleop.launch
```

### ◦ odometryノードの実行

- 端末をさらに一つ開き、以下のroslaunchコマンドによりodometryノードを実行する。roslaunchコマンドの1番目の引数はノード名でディレクトリ名にもなっている。2番目の引数は実行するPythonプログラム。

```
$ roslaunch odometry odometry.py
```

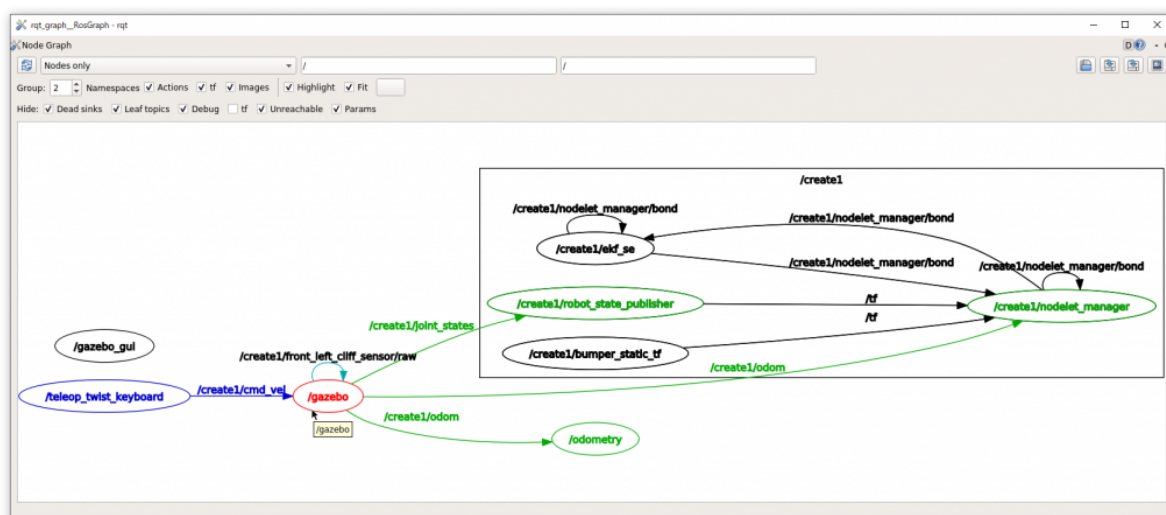
- キーボードでロボットを動かすと、以下のようにオドメトリの値が変化していることがわかる。最上図で、赤線がx軸、緑線がy軸、青線がz軸となっており、碁盤の目の1目盛りが1mとなっている。オドメトリのthetaはロボットの姿勢で単位はradian。



```
demulab@EARTH: ~/catkin_ws/src/hard2020/odometry/src
[INFO] [1597884203.751718]: Odometry: x=2.04207526503 y=-0.0101709239094 theta=-0.000264094610702
[INFO] [1597884203.755792]: Odometry: x=2.04494782353 y=-0.0101716907686 theta=-0.000265643126978
[INFO] [1597884203.801349]: Odometry: x=2.04782038225 y=-0.0101724560289 theta=-0.000267158870439
[INFO] [1597884203.805648]: Odometry: x=2.05069294117 y=-0.0101732255971 theta=-0.000268642536649
[INFO] [1597884203.851524]: Odometry: x=2.05356550029 y=-0.010173999382 theta=-0.000270094905041
[INFO] [1597884203.855131]: Odometry: x=2.05643805962 y=-0.0101747772943 theta=-0.000271516339376
[INFO] [1597884203.901868]: Odometry: x=2.05931061914 y=-0.0101755592468 theta=-0.00027290778824
[INFO] [1597884203.904972]: Odometry: x=2.06218317887 y=-0.0101763451539 theta=-0.00027426978551
[INFO] [1597884203.951534]: Odometry: x=2.06505573988 y=-0.010177134932 theta=-0.000275602950808
[INFO] [1597884203.954387]: Odometry: x=2.06792829894 y=-0.010177928499 theta=-0.000276907889955
[INFO] [1597884204.001936]: Odometry: x=2.07080085928 y=-0.0101787257747 theta=-0.000278185195427
[INFO] [1597884204.006439]: Odometry: x=2.07367341982 y=-0.0101795266806 theta=-0.000279435446784
[INFO] [1597884204.051629]: Odometry: x=2.07654598058 y=-0.0101803311399 theta=-0.000280659211101
[INFO] [1597884204.055065]: Odometry: x=2.07941854155 y=-0.0101811390772 theta=-0.000281857043384
[INFO] [1597884204.111851]: Odometry: x=2.08229110274 y=-0.0101819504188 theta=-0.000283029486981
[INFO] [1597884204.115851]: Odometry: x=2.08516366416 y=-0.0101827650925 theta=-0.000284177073983
[INFO] [1597884204.161534]: Odometry: x=2.0880362258 y=-0.0101835830277 theta=-0.00028530032559
[INFO] [1597884204.166179]: Odometry: x=2.09090878767 y=-0.0101844041553 theta=-0.000286399752493
[INFO] [1597884204.211699]: Odometry: x=2.09378134979 y=-0.0101852284075 theta=-0.000287475855223
[INFO] [1597884204.215461]: Odometry: x=2.09665391215 y=-0.0101860557181 theta=-0.000288529124501
[INFO] [1597884204.261454]: Odometry: x=2.09952647475 y=-0.010186886022 theta=-0.000289560041567
[INFO] [1597884204.311639]: Odometry: x=2.10239903761 y=-0.0101877192559 theta=-0.00029056907848
[INFO] [1597884204.315961]: Odometry: x=2.10527160073 y=-0.0101885553575 theta=-0.000291556698426
[INFO] [1597884204.361430]: Odometry: x=2.10814416412 y=-0.010189394266 theta=-0.000292523355997
[INFO] [1597884204.365049]: Odometry: x=2.11101672778 y=-0.0101902359217 theta=-0.000293469497463
[INFO] [1597884204.411792]: Odometry: x=2.11388929171 y=-0.0101910802665 theta=-0.000294395561021
[INFO] [1597884204.416780]: Odometry: x=2.11676185592 y=-0.0101919272432 theta=-0.000295301977026
```

## ノードとトピックの可視化

- では、以下のコマンドでノードやトピックを見てみよう。ノードは楕円で囲まれ中の文字列がノード名、矢印はデータの流れる方向で、その上の文字列がノード名です。



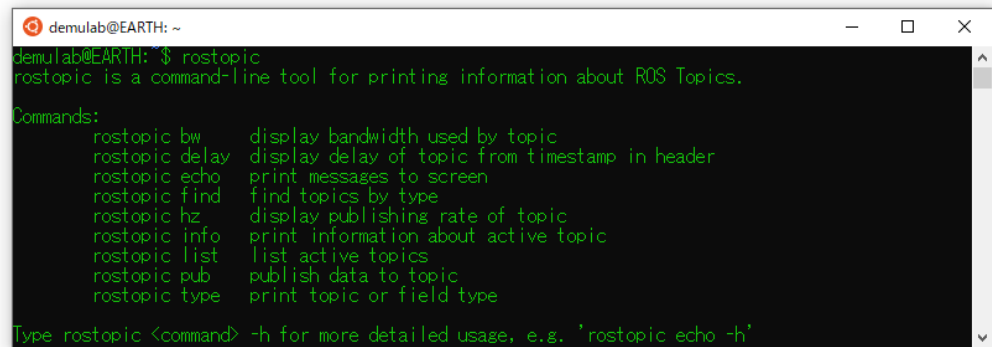
- ここでは、/create1と書かれた四角枠の外の、楕円で囲まれたteleop\_twist\_keyboardノードとgazeboノードとodometryノードだけに注目します。gazeboノードはシミュレータで、odometryノードへ/create1/odomトピックを送り、teleop\_twist\_keyboardノードから/create1/cmd\_velトピックを受け取っていることがわかります。

## トピックをコマンドで調べる

次のトピックを調べるコマンドを紹介します。

- rostopicコマンドの説明

- `$ rostopic`

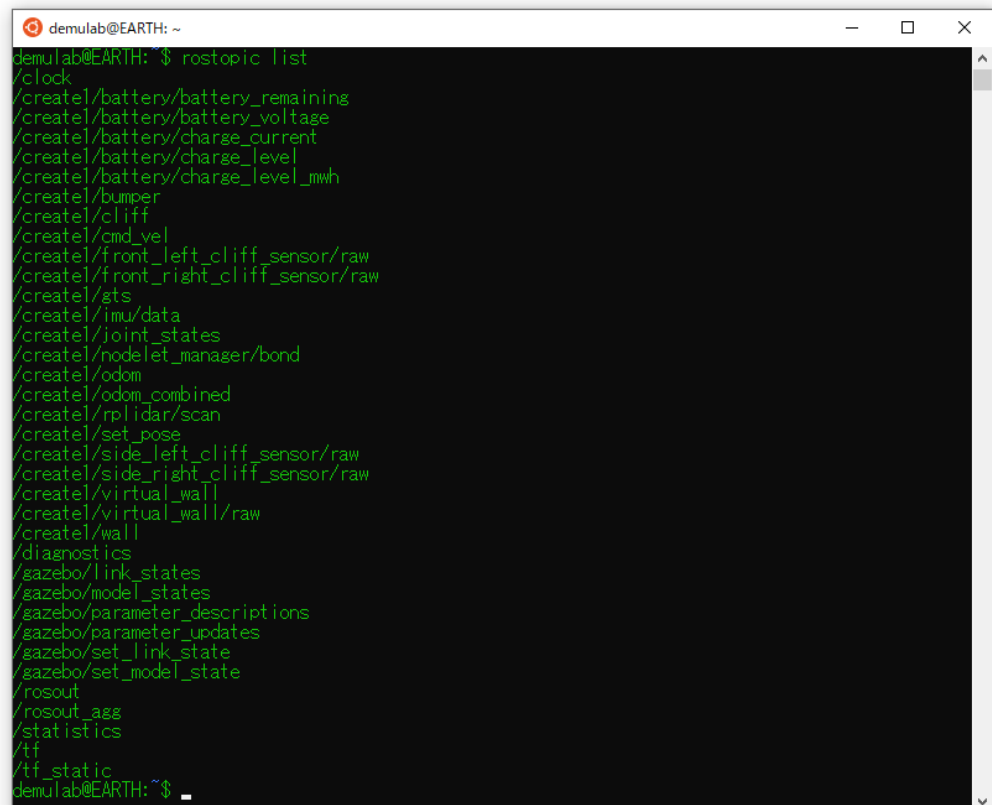


```
demulab@EARTH: ~  
demulab@EARTH: $ rostopic  
rostopic is a command-line tool for printing information about ROS Topics.  
  
Commands:  
  rostopic bw      display bandwidth used by topic  
  rostopic delay   display delay of topic from timestamp in header  
  rostopic echo    print messages to screen  
  rostopic find    find topics by type  
  rostopic hz      display publishing rate of topic  
  rostopic info    print information about active topic  
  rostopic list    list active topics  
  rostopic pub     publish data to topic  
  rostopic type    print topic or field type  
  
Type rostopic <command> -h for more detailed usage, e.g. 'rostopic echo -h'
```

- rostopicコマンドを実行すると使い方がわかります。その中で良く使うのは次のrostopic listです。使われているトピックがわかります。

- トピックの表示

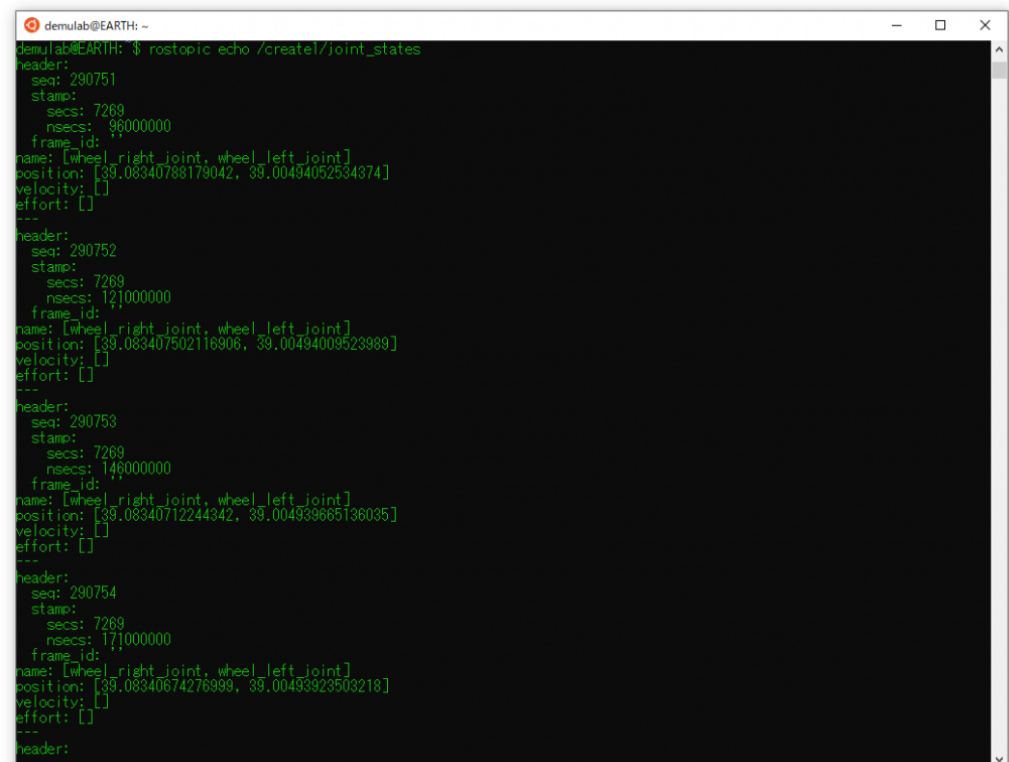
- `$ rostopic list`



```
demulab@EARTH: ~  
demulab@EARTH: $ rostopic list  
/clock  
/create1/battery/battery_remaining  
/create1/battery/battery_voltage  
/create1/battery/charge_current  
/create1/battery/charge_level  
/create1/battery/charge_level_mwh  
/create1/bumper  
/create1/cliff  
/create1/cmd_vel  
/create1/front_left_cliff_sensor/raw  
/create1/front_right_cliff_sensor/raw  
/create1/gts  
/create1/imu/data  
/create1/joint_states  
/create1/nodelet_manager/bond  
/create1/odom  
/create1/odom_combined  
/create1/rplidar/scan  
/create1/set_pose  
/create1/side_left_cliff_sensor/raw  
/create1/side_right_cliff_sensor/raw  
/create1/virtual_wall  
/create1/virtual_wall/raw  
/create1/wall  
/diagnostics  
/gazebo/link_states  
/gazebo/model_states  
/gazebo/parameter_descriptions  
/gazebo/parameter_updates  
/gazebo/set_link_state  
/gazebo/set_model_state  
/rosout  
/rosout_agg  
/statistics  
/tf  
/tf_static  
demulab@EARTH: ~$
```

- メッセージの表示

- `$ rostopic echo トピック名`
- 例として gazebo ノードからは `/create1/joint_states` トピックも出ているので、そのメッセージを見てみましょう。

A terminal window titled 'demulab@EARTH: ~' showing the output of the command 'rostopic echo /create1/joint\_states'. The output displays four consecutive messages. Each message has a 'header' section with 'seq', 'stamp' (secs and nsecs), and 'frame\_id'. The 'data' section contains 'name' (a list of joint names), 'position' (a list of two floating-point values), 'velocity' (an empty list), and 'effort' (an empty list). The position values for the right and left joints are shown in increasing order across the four messages.

```
demulab@EARTH: ~$ rostopic echo /create1/joint_states
header:
  seq: 290751
  stamp:
    secs: 7269
    nsecs: 96000000
  frame_id: ''
name: [wheel_right_joint, wheel_left_joint]
position: [39.08340788179042, 39.00494052534374]
velocity: []
effort: []
---
header:
  seq: 290752
  stamp:
    secs: 7269
    nsecs: 121000000
  frame_id: ''
name: [wheel_right_joint, wheel_left_joint]
position: [39.083407502116906, 39.00494009523989]
velocity: []
effort: []
---
header:
  seq: 290753
  stamp:
    secs: 7269
    nsecs: 146000000
  frame_id: ''
name: [wheel_right_joint, wheel_left_joint]
position: [39.08340712244342, 39.004939665136035]
velocity: []
effort: []
---
header:
  seq: 290754
  stamp:
    secs: 7269
    nsecs: 171000000
  frame_id: ''
name: [wheel_right_joint, wheel_left_joint]
position: [39.08340674276999, 39.00493923503218]
velocity: []
effort: []
---
header:
```

- `/create1/joint_states` トピックは左右のロボット車輪の回転量[rad]がわかります。これを使うとロボットのオドメトリを計算できそうですね。なお、各トピックには連番と時間情報がついています。

終わり