MoveIt Tutorial Documentation

Release 0.1.1

Tokyo Opensource Robotics Kyokai Association

Oct 26, 2022

CONTENTS

1 はじめに **1**
- 1.1 ROS と MoveIt! . . . 1
- 1.2 チュートリアルの構成 . . . 1

2 ロボットシミュレータを使う **3**
- 2.1 シミュレータの種類 . . . 3
- 2.2 ソフトウェアのインストール . . . 3
  - 2.2.1 ROS のインストール . . . 4
  - 2.2.2 チュートリアルパッケージのインストール . . . 4
  - 2.2.3 NEXTAGE OPEN ソフトウェアのインストール . . . 5
  - 2.2.4 KHI duaro ソフトウェアのインストール . . . 5
  - 2.2.5 myCobot ソフトウェアのインストール . . . 5
  - 2.2.6 インストールの最後に . . . 5
- 2.3 ソースインストール . . . 6
  - 2.3.1 MINAS TRA1 ソフトウェアの取得とビルド . . . 6
  - 2.3.2 ワークスペースでビルドしたソフトウェアのインストール . . . 6
- 2.4 シミュレータと MoveIt! の起動 . . . 7
  - 2.4.1 NEXTAGE OPEN - hrpsys シミュレータ . . . 7
    - hrpsys シミュレータ・MoveIt! などの起動 . . . 7
    - シミュレータの終了 . . . 7
  - 2.4.2 NEXTAGE OPEN - Gazebo シミュレータ . . . 7
    - Gazebo シミュレータの起動 . . . 8
    - MoveIt! の起動 . . . 9
    - シミュレータの終了 . . . 10
  - 2.4.3 MINAS TRA1 - MoveIt! シミュレータ . . . 10
  - 2.4.4 KHI duaro - Gazebo シミュレータ . . . 10
    - Gazebo シミュレータの起動 . . . 10
    - MoveIt! の起動 . . . 11
    - シミュレータの終了 . . . 12
  - 2.4.5 myCobot - MoveIt! シミュレータ . . . 12
    - MoveIt! の起動 . . . 12
    - シミュレータの終了 . . . 13
  - 2.4.6 MoveIt! GUI での動作計画 . . . 13

3 プログラムでロボットを動かす **15**
- 3.1 プログラムを入力して実行する . . . 15
  - 3.1.1 NEXTAGE OPEN の場合 . . . 15
    - 特定の関節を動かす . . . 15
    - 腕全体の関節を動かす . . . 16
    - 手先の位置を指定して動かす . . . 17
    - 手先の姿勢を指定して動かす . . . 17
    - 手先の位置と姿勢を指定して動かす . . . 17
    - 直線補間軌道でロボットを動かす . . . 20
    - 連続した指令をロボットに送る . . . 21
    - 四角形や円に沿ってエンドエフェクタを動かす . . . 22

3.1.2 myCobot の場合 . . . 23
実機を動かす場合 . . . 23
特定の関節を動かす . . . 23
腕全体の関節を動かす . . . 25
手先の位置を指定して動かす . . . 25
手先の姿勢を指定して動かす . . . 25
手先の位置と姿勢を指定して動かす . . . 26
直線補間軌道でロボットを動かす . . . 29
連続した指令をロボットに送る . . . 29
四角形や円に沿ってエンドエフェクタを動かす . . . 32
3.2 プログラムファイルを実行する . . . 37
3.2.1 NEXTAGE OPEN の場合 . . . 38
3.2.2 MINAS TRA1 の場合 . . . 39
3.2.3 KHI duaro の場合 . . . 40
3.2.4 myCobot の場合 . . . 41
3.2.5 ROS や MoveIt! のメリット . . . 45

4 発展的なロボットプログラミング 47
4.1 プログラム制御フローツールとロボットプログラミング . . . 47
4.1.1 条件判断とロボット動作 - if . . . 47
4.1.2 繰り返しとロボット動作 - for . . . 48
4.2 プログラムのタイミングを図る . . . 49
4.2.1 プログラムの一時休止 - sleep . . . 49
4.2.2 プログラムループの一定時間間隔実行 - Rate . . . 49
4.2.3 関数の定期呼び出し - Timer . . . 50
4.2.4 ROS ノードを停止させずに待機 - spin . . . 50
4.2.5 定期的に動くロボットプログラム例 . . . 50
4.3 より複雑なロボット動作計画 . . . 52
4.3.1 動作アームを指定する . . . 52
もう一方の腕を動かす . . . 52
両腕を動かす . . . 53
4.3.2 姿勢の参照座標を指定する . . . 55
4.3.3 座標系フレーム間の相対姿勢を取得する - tf . . . 58
4.3.4 画像処理プログラム出力 tf の利用 . . . 61
4.3.5 障害物の設置と回避動作計画 . . . 68
4.3.6 障害物回避動作計画における拘束条件の付加 . . . 70

5 トラブルシューティング 75
5.1 動作計画が得られない . . . 75
5.1.1 動作姿勢に問題がある . . . 75
【対策】姿勢を変更する・姿勢の経由点を追加する . . . 75
5.1.2 計算がタイムアウトする . . . 75
【対策】動作計画のプランナを指定する . . . 75
【対策】動作計画の計算タイムアウト設定を長くする . . . 75
5.2 動作計画が実行されない . . . 76
5.2.1 動作計画軌道上に障害物が存在 . . . 76
【対策】動作の再計画やターゲット姿勢などの変更 . . . 76
5.3 tf が取得できない . . . 76
5.3.1 画像処理ノードとの関係上の問題 . . . 76
【対策】tf 変換の試行回数を多くする . . . 76
5.3.2 同期が取れていない - 実機ロボットの場合 . . . 76
【対策】NTP サーバと同期をとる . . . 76
5.4 Gazebo の問題 . . . 76
5.4.1 Gazebo を起動してもロボットが表示されない . . . 76
【対策】PC で初めて Gazebo を起動した場合は「しばらく待つ」 . . . 76
【対策】ROS 環境設定を確認する . . . 77
【対策】全ての ROS ノードを終了して再起動 . . . 77

6 クラス・関数リファレンス 79

6.1 MoveIt! Commander . . . 79
6.1.1 MoveGroupCommander クラス . . . 79
def **init**(self, name, robot_description= ”robot_description ”): . . . 79
def get_name(self): . . . 79
stop(self): . . . 80
def get_active_joints(self): . . . 80
def get_joints(self): . . . 80
def get_variable_count(self): . . . 80
has_end_effector_link(self): . . . 81
def get_end_effector_link(self): . . . 81
def set_end_effector_link(self, link_name): . . . 81
def get_pose_reference_frame(self): . . . 81
def set_pose_reference_frame(self, reference_frame): . . . 82
def get_planning_frame(self): . . . 82
def get_current_joint_values(self): . . . 82
def get_current_pose(self, end_effector_link = “ ”): . . . 82
def get_current_rpy(self, end_effector_link = “ ”): . . . 83
def get_random_joint_values(self): . . . 83
def get_random_pose(self, end_effector_link = “ ”): . . . 83
def set_start_state_to_current_state(self): . . . 83
def set_start_state(self, msg): . . . 84
def set_joint_value_target(self, arg1, arg2 = None, arg3 = None): . . . 84
def set_rpy_target(self, rpy, end_effector_link = “ ”): . . . 84
def set_orientation_target(self, q, end_effector_link = “ ”): . . . 85
def set_position_target(self, xyz, end_effector_link = “ ”): . . . 85
def set_pose_target(self, pose, end_effector_link = “ ”): . . . 85
def set_pose_targets(self, poses, end_effector_link = “ ”): . . . 86
def shift_pose_target(self, axis, value, end_effector_link = “ ”): . . . 86
def clear_pose_target(self, end_effector_link): . . . 86
def clear_pose_targets(self): . . . 86
def set_random_target(self): . . . 87
def set_named_target(self, name): . . . 87
def remember_joint_values(self, name, values = None): . . . 87
def get_remembered_joint_values(self): . . . 87
def forget_joint_values(self, name): . . . 88
def get_goal_tolerance(self): . . . 88
def get_goal_joint_tolerance(self): . . . 88
def get_goal_position_tolerance(self): . . . 88
def get_goal_orientation_tolerance(self): . . . 89
def set_goal_tolerance(self, value): . . . 89
def set_goal_joint_tolerance(self, value): . . . 89
def set_goal_position_tolerance(self, value): . . . 89
def set_goal_orientation_tolerance(self, value): . . . 90
def allow_looking(self, value): . . . 90
def allow_replanning(self, value): . . . 90
def get_known_constraints(self): . . . 90
def get_path_constraints(self): . . . 91
def set_path_constraints(self, value): . . . 91
def clear_path_constraints(self): . . . 91
def set_constraints_database(self, host, port): . . . 91
def set_planning_time(self, seconds): . . . 92
def get_planning_time(self): . . . 92
def set_planner_id(self, planner_id): . . . 92
def set_num_planning_attempts(self, num_planning_attempts): . . . 92
def set_workspace(self, ws): . . . 93
def set_max_velocity_scaling_factor(self, value): . . . 93
def go(self, joints = None, wait = True): . . . 93
def plan(self, joints = None): . . . 93

def compute_cartesian_path(self, waypoints, eef_step, jump_threshold, avoid_collisions = True): . . . 94
def execute(self, plan_msg, wait = True): . . . 94
def attach_object(self, object_name, link_name = “ ”, touch_links = []): . . . 94
def detach_object(self, name = “ ”): . . . 95
def pick(self, object_name, grasp = []): . . . 95
def place(self, object_name, location=None): . . . 95
def set_support_surface_name(self, value): . . . 95
def retime_trajectory(self, ref_state_in, traj_in, velocity_scaling_factor): . . . 96
6.1.2 RobotCommander クラス . . . 96
def **init**(self, robot_description= ”robot_description ”): . . . 96
def get_planning_frame(self): . . . 96
def get_root_link(self): . . . 97
def get_joint_names(self, group=None): . . . 97
def get_link_names(self, group=None): . . . 97
def get_current_state(self): . . . 97
def get_current_variable_values(self): . . . 98
def get_joint(self, name): . . . 98
def get_link(self, name): . . . 98
def get_group(self, name): . . . 98
def has_group(self, name): . . . 99
def get_default_owner_group(self, joint_name): . . . 99
6.1.3 Joint クラス（ RobotCommandeer 内クラス ） . . . 99
def **init**(self, robot, name): . . . 99
def name(self): . . . 99
def variable_count(self): . . . 100
def bounds(self): . . . 100
def min_bound(self): . . . 100
def max_bound(self): . . . 100
def value(self): . . . 101
def move(self, position, wait=True): . . . 101
def __get_joint_limits(self): . . . 101
6.1.4 Link クラス（ RobotCommandeer 内クラス ） . . . 101
def **init**(self, robot, name): . . . 101
def name(self): . . . 102
def pose(self): . . . 102
6.1.5 PlanningSceneInterface クラス . . . 102
def **init**(self): . . . 102
def __make_sphere(self, name, pose, radius): . . . 103
def add_sphere(self, name, pose, radius = 1): . . . 103
def __make_box(self, name, pose, size): . . . 103
def add_box(self, name, pose, size = (1, 1, 1)): . . . 104
def __make_mesh(self, name, pose, filename, scale = (1, 1, 1)): . . . 104
def add_mesh(self, name, pose, filename, size = (1, 1, 1)): . . . 104
def __make_existing(self, name): . . . 105
def add_plane(self, name, pose, normal = (0, 0, 1), offset = 0): . . . 105
def attach_mesh(self, link, name, pose = None, filename = ‘ ’, size = (1, 1, 1), touch_links = []): . . . 105
def attach_box(self, link, name, pose = None, size = (1, 1, 1), touch_links = []): . . . 106
def remove_world_object(self, name = None): . . . 106
def remove_attached_object(self, link, name = None): . . . 106
def get_known_object_names(self, with_type = False): . . . 107
def get_known_object_names_in_roi(self, minx, miny, minz, maxx, maxy, maxz, with_type = False): . . . 107
def get_objects(self, object_ids = []): . . . 107
def get_attached_objects(self, object_ids = []): . . . 108
6.1.6 conversions の関数 . . . 108
def msg_to_string(msg): . . . 108
def msg_from_string(msg, data): . . . 108

def pose_to_list(pose_msg): . . . 109
def list_to_pose(pose_list): . . . 109
def list_to_pose_stamped(pose_list, target_frame): . . . 109
def transform_to_list(trf_msg): . . . 109
def list_to_transform(trf_list): . . . 110
6.1.7 roscpp_initializer の関数 . . . 110
def roscpp_initialize(args): . . . 110
def roscpp_shutdown(): . . . 110
6.2 チュートリアルパッケージ . . . 111
6.2.1 moveit_tutorial_tools の関数 . . . 111
def init_node(node_name = “ commander_example ”): . . . 111
def question_yn(qmsg= ’Message ’, title= ’Question ’): . . . 111
def get_current_target_pose(target_frame_id, base_frame_id, timeout = 1.0): . . . 111
def make_waypoints_example(rtype= ”NEXTAGE ”): . . . 112
def make_waypoints_rectangular(dp_a=[0.25, 0.0, 0.1], dp_b=[0.45, -0.2, 0.1], rpy=[0.0,0.0,0.0]): . . . 112
def make_waypoints_circular(center=[0.3, -0.2, 0.1], radius=0.1 ,steps=12, rpy=[0.0,0.0,0.0]): . . . 112

7 Python チュートリアル 113
7.1 Python とは . . . 113
7.2 Python をはじめる . . . 113
7.2.1 IPython の起動 . . . 113
7.2.2 命令文の入力と実行 . . . 114
7.2.3 計算 . . . 115
7.2.4 変数 . . . 116
7.2.5 リスト . . . 116
7.2.6 文字列 . . . 117
7.3 Python プログラミング . . . 117
7.3.1 比較と真偽値 . . . 117
7.3.2 関数 . . . 117
7.3.3 クラス . . . 118
7.3.4 制御フローツール . . . 121
while 文 . . . 121
if 文 . . . 121
for 文 . . . 122
7.3.5 コメント文 . . . 123
1 行のコメントアウト . . . 123
複数行のコメントアウト . . . 124

8 実機の使い方 125
8.1 myCobot280 の場合 . . . 125
8.1.1 myCobot280 の固定 . . . 125
8.1.2 myCobot280 のファームウェア更新 . . . 125
myStudio のダウンロード . . . 125
USB ドライバのインストール . . . 126
ファームウェアの更新 . . . 126
8.1.3 myCobot280 を Transponder モードにする . . . 130
8.1.4 dialout グループへのユーザ追加 . . . 130
8.1.5 pymycobot (Python API) のインストール . . . 131

CHAPTER

ONE

はじめに

本チュートリアルは GUI（グラフィカル・ユーザ・インタフェース）からロボットやロボットシミュレーションを動かしたことはあるもののそれをプログラムから動かすことは行ったことがない人がプログラミングを始められる，そして発展的に様々なロボット動作プログラミングができるようになることを目標としています.

本チュートリアルの PDF 形式のファイル，および Markdown 形式のソースコードは以下の場所で公開されています．コマンド等の文字列は，ここからコピー&ペーストすると便利です.

- tork-a/tork_moveit_tutorial : https://github.com/tork-a/tork_moveit_tutorial

1.1 ROS と MoveIt!

ロボットソフトウェア開発で近年広く使われるようになっているのが **ROS (ロス / Robot Operating System)** です.

- ROS : http://wiki.ros.org/ja

  ROS (Robot Operating System) はソフトウェア開発者のロボット・アプリケーション作成を支援するライブラリとツールを提供しています. 具体的には, ハードウェア抽象化, デバイスドライバ，ライブラリ，視覚化ツール, メッセージ通信，パッケージ管理などが提供されています. ROS はオープンソースの一つ, BSD ライセンスにより, ライセンス化されています.

ROS でのロボット動作計画で頻繁に使用されるのが **MoveIt!** の動作計画機能です．MoveIt! は GUI での操作を提供していますが，それだけではなくそのプログラミングインタフェースである **MoveIt! Commander** が提供されていますので，プログラミング言語から MoveIt! の機能を利用してロボットを動かすこともできます.

MoveIt! Commander のプログラミングインタフェースには Python や C++ があります.

- MoveIt! : http://moveit.ros.org/
- MoveIt! Tutorials : http://docs.ros.org/melodic/api/moveit_tutorials/html/

1.2 チュートリアルの構成

本チュートリアルでは MoveIt! Commander を使ってロボットの動作プログラミングを行います.

チュートリアルの構成は次のようになっています.

- はじめに（本章）
- ロボットシミュレータを使う
- プログラムでロボットを動かす
- 発展的なロボットプログラミング
- トラブルシューティング
- クラス・関数リファレンス

- Python チュートリアル
- 実機の使用法

本チュートリアルのロボットプログラミングではプログラム言語 Python を使いますが，Python の知識がなくてもコマンドやプログラムを実行できるように構成されています.

また，ロボット動作ではシミュレーションでのプログラミング実行が記述されていますので，実機のロボットを扱える環境にない人やロボットプログラミングに慣れていないので先ずはシミュレーションからという人でもプログラミングを行えます.

CHAPTER

TWO

ロボットシミュレータを使う

シミュレータ上のロボットを動かしてみます．本チュートリアルでは下記のロボットのシミュレータの利用方法を紹介します．

- myCobot : 教育用単腕マニピュレータ
- NEXTAGE OPEN : 人型双腕ロボット
- MINAS TRA1 : 単腕マニピュレータ（「ソースインストール」の章にて）
- KHI duaro : スカラ型双腕ロボット
- Baxter Research Robot : 人型双腕ロボット（「ワークスペースの作成」の章にて）

2.1 シミュレータの種類

本チュートリアルで扱うシミュレータには次のような種類があります．

- ROS のシミュレータ
  - myCobot / NEXTAGE OPEN / MINAS TRA1 / KHI duaro
    - MoveIt! シミュレータ : 運動学のみの動作計画シミュレータ
    - Gazebo シミュレータ : 動力学を含む環境・物理シミュレータ
- hrpsys(RTM) シミュレータ
  - NEXTAGE OPEN のみ
    - 動力学を含む物理シミュレータ

本チュートリアルではどのシミュレータを使っても最終的には動作計画ソフトウェアの MoveIt! を起動してその動作計画機能を利用します．

また，本チュートリアルの構成として，NEXTAGE OPEN の Gazebo シミュレータと MoveIt! の組み合わせを基本として各ロボットへの応用を展開する形を採っています．

2.2 ソフトウェアのインストール

ソフトウェアのインストールでは主に次のインストール項目があります．

- ROS のインストール
- ロボットシミュレータなどのインストール

次のソフトウェアのインストールをします．

各ロボットソフトウェアは全てインストールしても，どれか 1 つでも大丈夫ですが，本チュートリアルは NEXTAGE OPEN を中心的な例として記述していますので少なくとも NEXTAGE OPEN ソフトウェアのインストールをお願いします．

NEXTAGE OPEN に加えて他のロボットのソフトウェアもインストールすると他のロボットへのプログラム応用方法についての理解が進みます.

- ROS とチュートリアルパッケージ
- ロボットソフトウェア
  - NEXTAGE OPEN ソフトウェア
  - KHI duaro ソフトウェア

また，システム構成は次のとおりです.

- Ubuntu 18.04
- ROS Melodic

ROS は Ubuntu の各バージョンに対応したものがあります．それぞれに対応した Ubuntu と ROS の組み合わせで利用する必要があります.

Ubuntu バージョン	ROS バージョン	サポート終了
20.04 (Focal)	Noetic Ninjemys	2025 年 5 月
18.04 (Bionic)	Melodic Morenia	2023 年 5 月
16.04 (Xenial)	Kinetic Kame	2021 年 4 月
14.04 (Trusty)	Indigo Igloo	2019 年 4 月

詳しくは下記の ROS Wiki で確認してください.

- ROS Wiki - Distributions
  - http://wiki.ros.org/Distributions

2.2.1 ROS のインストール

ターミナルから次のコマンドを実行して ROS ソフトウェアをインストールします．既に ROS がインストールされていれば，次のチュートリアルパッケージのインストールに進んでください.

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" >
 /etc/apt/sources.list.d/ros-latest.list'
sudo apt-get install curl
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-
↪key add -
sudo apt-get update
sudo apt-get install ros-melodic-desktop-full
```

rosdep の初期化を行います.

```
sudo rosdep init
rosdep update
```

rosinstall をインストールします.

```
sudo apt-get install python-rosinstall
```

- 参考 :
  - Ubuntu install of ROS melodic
  - ROS melodic の Ubuntu へのインストール

2.2.2 チュートリアルパッケージのインストール

ターミナルから次のコマンドを実行してチュートリアルパッケージのソフトウェアをインストールします.

```
sudo apt-get install ros-melodic-tork-moveit-tutorial
```

2.2.3 NEXTAGE OPEN ソフトウェアのインストール

ターミナルから次のコマンドを実行して NEXTAGE OPEN のソフトウェアをインストールします.

```
sudo apt-get update && sudo apt-get install ros-melodic-rtmros-nextage ros-melodic-
↪rtmros-hironx
```

2.2.4 KHI duaro ソフトウェアのインストール

ターミナルから次のコマンドを実行して KHI duaro のソフトウェアをインストールします.

```
sudo apt-get update && sudo apt-get install ros-melodic-khi-duaro-gazebo ros-
↪melodic-khi-duaro-description ros-melodic-khi-duaro-ikfast-plugin ros-melodic-
↪khi-duaro-moveit-config
```

2.2.5 myCobot ソフトウェアのインストール

myCobot のソフトウェアのインストールにはワークスペースの作成が必要です.

- catkin の workspace を作る
  - http://wiki.ros.org/ja/catkin/Tutorials/create_a_workspace

`catkin_ws` という名前のワークスペースを作成する手順は次のとおりです.

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
cd ~/catkin_ws/
catkin_make
source devel/setup.bash
```

次に，myCobot のソースコードの取得とビルドを行います.

```
cd ~/catkin_ws/src
git clone https://github.com/tork-a/tork_moveit_tutorial
rosdep install --from-paths . --ignore-src -y
cd ~/catkin_ws
catkin_make
source devel/setup.bash
```

2.2.6 インストールの最後に

インストールの最後に setup.bash を読み込み，ROS の環境を設定します.

```
source /opt/ros/melodic/setup.bash
```

これは新しくターミナルを立ち上げて ROS を使用する前に毎回必要になります．下記のように .bashrc ファイルに設定を加えてターミナル起動時に setup.bash を自動で実行し ROS 環境になるようにしておくと便利です.

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

- **注意**: 上記コマンドの >> を > にしてしまうと元々あった .bashrc 内の設定が消えてしまうので気をつけてください.

また，~/catkin_ws というワークスペースを作成した場合は，ワークスペース内の setup.bash を読み込むことで，ワークスペース内を含む ROS の環境を設定することができます。

```
source ~/catkin_ws/devel/setup.bash
```

同様に、これは新しくターミナルを立ち上げて ROS を使用する前には毎回必要になるので，下記のように .bashrc ファイルに設定を加えることでターミナル起動時に自動で実行するようにしておくと便利です.

```
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
```

.bashrc の設定ができていると以後のターミナルを起動するたびに行う source /opt/ros/melodic/setup.bash 及び source ~/catkin_ws/devel/setup.bash は不要です.

2.3 ソースインストール

apt でインストールできるようにバイナリ／リリースされていないロボットパッケージはワークスペース経由でインストールすることができます．標準では推奨されていない方法ですので十分に注意して実行してください.

まず，/tmp/catkin_ws という名前のワークスペースを作成する手順は次のとおりです.

```
source /opt/ros/melodic/setup.bash
mkdir -p /tmp/catkin_ws/src
cd /tmp/catkin_ws/src
catkin_init_workspace
```

2.3.1 MINAS TRA1 ソフトウェアの取得とビルド

次の手順で MINAS TRA1 のクローンとそれに必要なソフトウェアパッケージの取得，ビルドを行います.

```
cd /tmp/catkin_ws/src
git clone https://github.com/tork-a/minas.git
rosdep install --from-paths . --ignore-src -y
cd /tmp/catkin_ws
catkin_make
```

2.3.2 ワークスペースでビルドしたソフトウェアのインストール

ここまでの手順がエラー無く進んでいることを再度確認したら次の手順によりワークスペースでビルドしたパッケージを/opt/ros/melodic/へとインストールします.

繰り返しになりますが標準では推奨されない方法ですのでもしここまでの手順でエラーが出ていれば作業を中止してください.

```
cd /tmp/catkin_ws
sudo su
source /opt/ros/melodic/setup.bash
catkin_make_isolated --install --install-space /opt/ros/melodic -DCMAKE_BUILD_
↪TYPE=Release
```

2.4 シミュレータと MoveIt! の起動

2.4.1 NEXTAGE OPEN - hrpsys シミュレータ

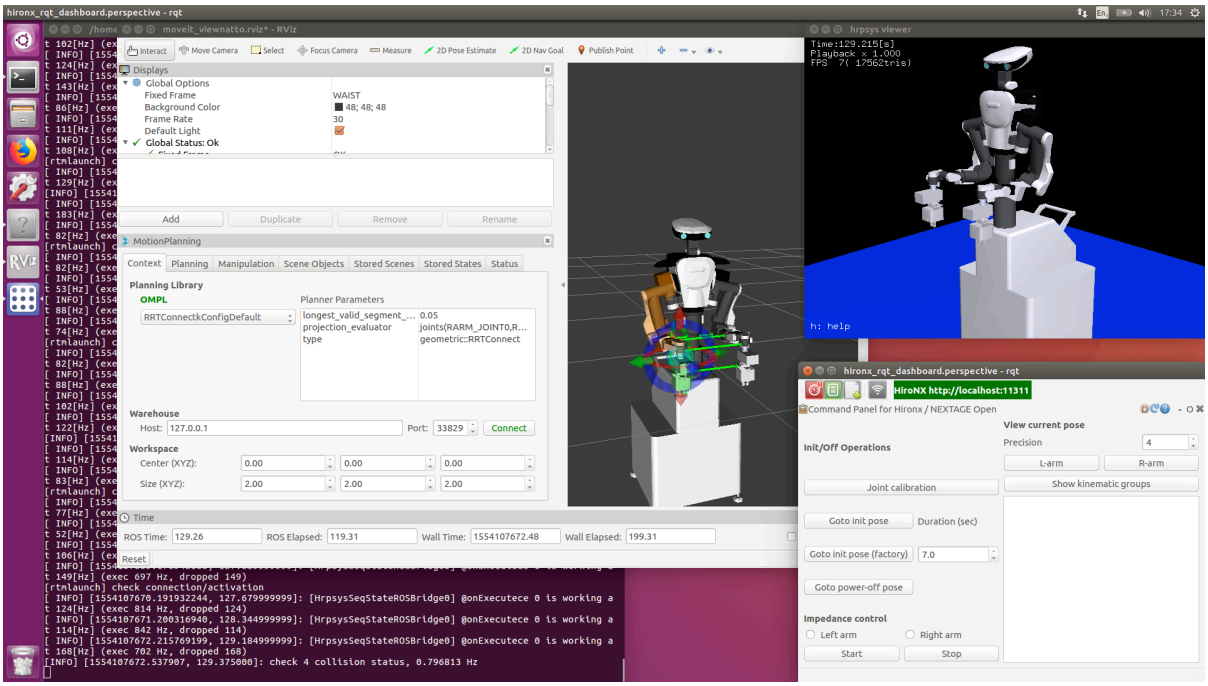
hrpsys シミュレータ・MoveIt! などの起動

NEXTAGE OPEN の動力学シミュレータの一つである NEXTAGE OPEN hrpsys(RTM) シミュレータを起動します．ターミナルを開いて次のコマンドを実行してください．

```
source /opt/ros/melodic/setup.bash
rtmlaunch nextage_moveit_config nextage_demo.launch
```

コマンドを実行すると次の 4 つのウィンドウが開きます．

- ターミナル
- hrpsys シミュレータ (hrpsys viewer)
- MoveIt! / RViz
- Hironx Dashboard (Command Panel for Hironx / NEXTAGE Open)



これで MoveIt! の動作計画機能が利用できる状態になっています．

シミュレータの終了

シミュレータを終了するには全体を起動したターミナル上で Ctrl-C を入力すると全体が終了します．

2.4.2 NEXTAGE OPEN - Gazebo シミュレータ

NEXTAGE OPEN のもうひとつの動力学シミュレータは ROS の動力学環境シミュレータ Gazebo 上で動きます．

ターミナルを 2 つ開きます．

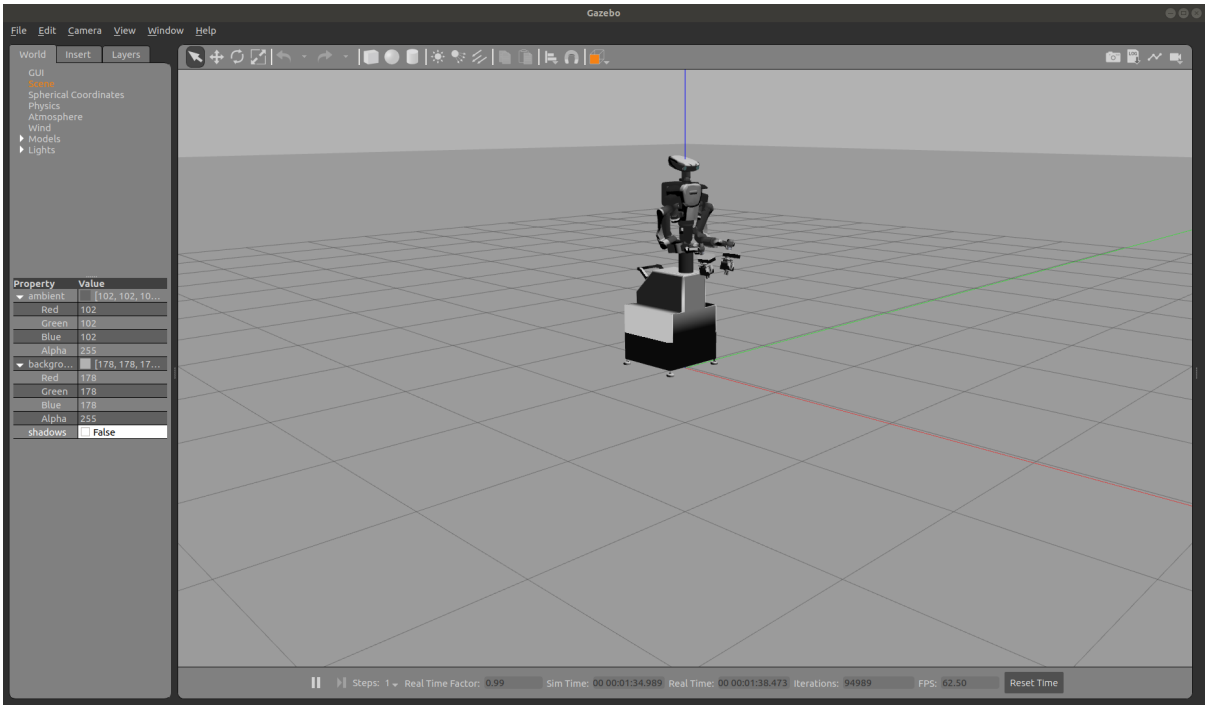
Gazebo シミュレータの起動

1 つ目のターミナルで次のコマンドを入力しして NEXTAGE OPEN Gazebo シミュレータを起動します.

ターミナル-1 : Gazebo シミュレータの起動

```
$ source /opt/ros/melodic/setup.bash
$ roslaunch nextage_gazebo nextage_world.launch
:
:
[go_initial-6] process has finished cleanly
log file: /home/robotuser/.ros/log/5d4ac8aa-baeb-11e7-af06-001c4284b313/go_initial-
↪6*.log
:
```

Gazebo が起動して上記のターミナルの出力が得られたら Gazebo シミュレータ内の NEXTAGE OPEN ロボットの準備が完了しています.



- 注意: 最初に gazebo を立ち上げる際にはモデルデータをダウンロードするために以下のように Warning や Error が表示され数秒から数分の時間がかかる場合が有ります.「トラブルシューティング (Gazebo を起動してもロボットが表示されない)」もご参照ください.

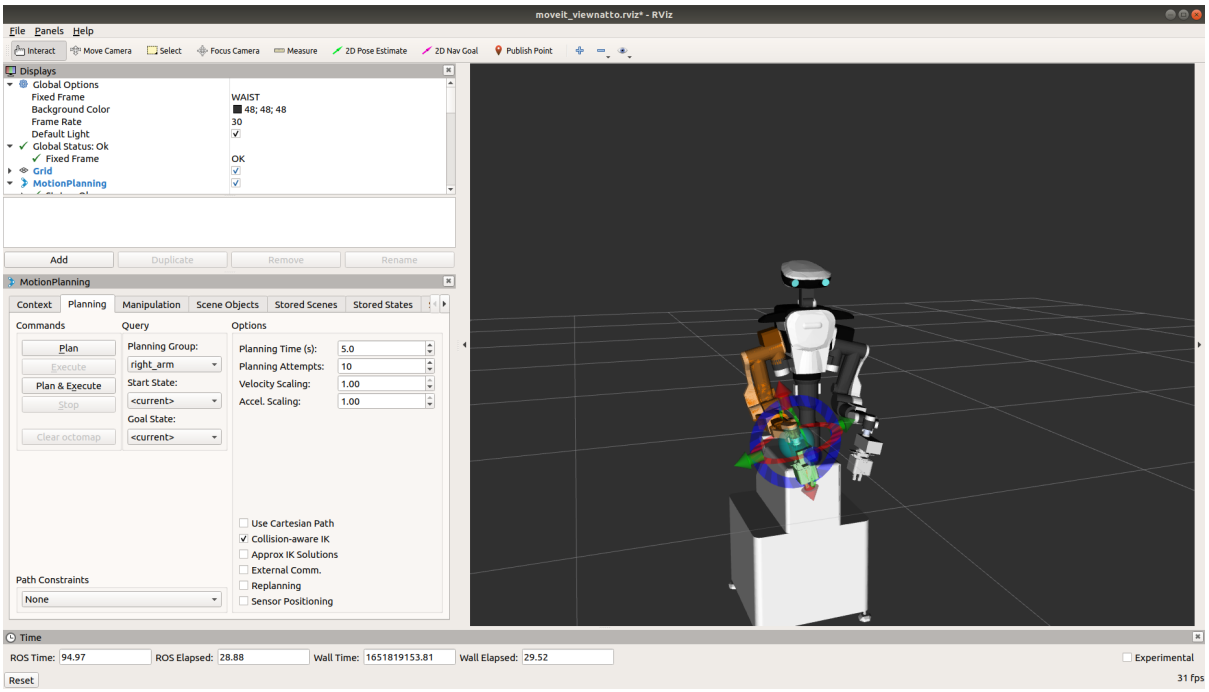
```
Gazebo multi-robot simulator, version 2.2.3
Copyright (C) 2012-2014 Open Source Robotics Foundation.
Released under the Apache 2 License.
http://gazebosim.org

Msg Waiting for master
Msg Connected to gazebo master @ http://127.0.0.1:11345
Msg Publicized address: 10.0.2.15
[ INFO] [1514250989.072029722]: Finished loading Gazebo ROS API Plugin.
Msg Waiting for master
[ INFO] [1514250989.074773534]: waitForService: Service [/gazebo/set_physics_pro
perties] has not been advertised, waiting...
Msg Connected to gazebo master @ http://127.0.0.1:11345
Msg Publicized address: 10.0.2.15
spawn_model script started
Warning [gazebo.cc:215] Waited 1seconds for namespaces.
Warning [ModelDatabase.cc:334] Getting models from[http://gazebosim.org/models/]
. This may take a few seconds.
[INFO] [WallTime: 1514250990.267662] [0.000000] Loading model xml from ros param
eter
[INFO] [WallTime: 1514250990.271217] [0.000000] Waiting for service /gazebo/spaw
n_urdf_model
Warning [gazebo.cc:215] Waited 1seconds for namespaces.
Warning [gazebo.cc:215] Waited 1seconds for namespaces.
Warning [gazebo.cc:215] Waited 1seconds for namespaces.
Warning [gazebo.cc:215] Waited 1seconds for namespaces.
Warning [gazebo.cc:215] Waited 1seconds for namespaces.
Error [gazebo.cc:220] Waited 11 seconds for namespaces. Giving up.
Error [Node.cc:90] No namespace found
Error [Node.cc:90] No namespace found
Error [Node.cc:90] No namespace found
Error [Node.cc:90] No namespace found
[WARN] [WallTime: 1514251018.700622] [0.000000] Controller Spawner couldn't find the expected controller_manager ROS interface.
[controller_spawner-5] process has finished cleanly
log file: /home/fujiimak/.ros/log/62fa17ec-e9da-11e7-8756-0800275ee806/controller_spawner-5*.log
[INFO] [WallTime: 1514251093.946457] [0.022000] Calling service /gazebo/spawn_ur
```

MoveIt! の起動

2 つ目のターミナルで次のコマンドを入力して MoveIt! を起動します.

ターミナル-2 : MoveIt! の起動

```
source /opt/ros/melodic/setup.bash
roslaunch nextage_moveit_config moveit_planning_execution.launch
```



これで MoveIt! の動作計画機能が利用できる状態になっています.

シミュレータの終了

シミュレータを終了するには各ターミナルで Ctrl-C を入力してください.

2.4.3 MINAS TRA1 - MoveIt! シミュレータ

ターミナルを 2 つ起動します.

1 つ目のターミナルでコントローラをシミュレーションモードで起動します.

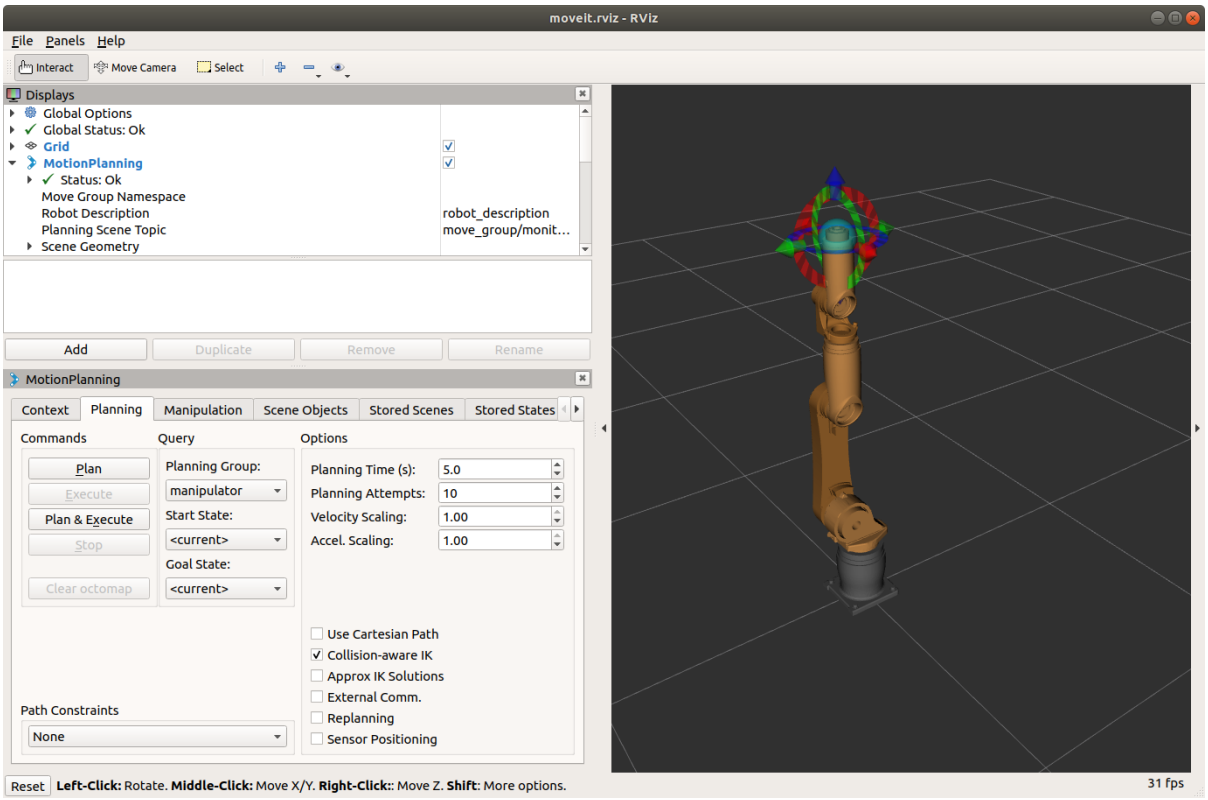
ターミナル-1

```
source /opt/ros/melodic/setup.bash
roslaunch tra1_bringup tra1_bringup.launch simulation:=true
```

2 つ目のターミナルで MoveIt! を起動します.

ターミナル-2

```
source /opt/ros/melodic/setup.bash
roslaunch tra1_bringup tra1_moveit.launch
```



シミュレータを終了するには各ターミナルで Ctrl-C を入力してください.

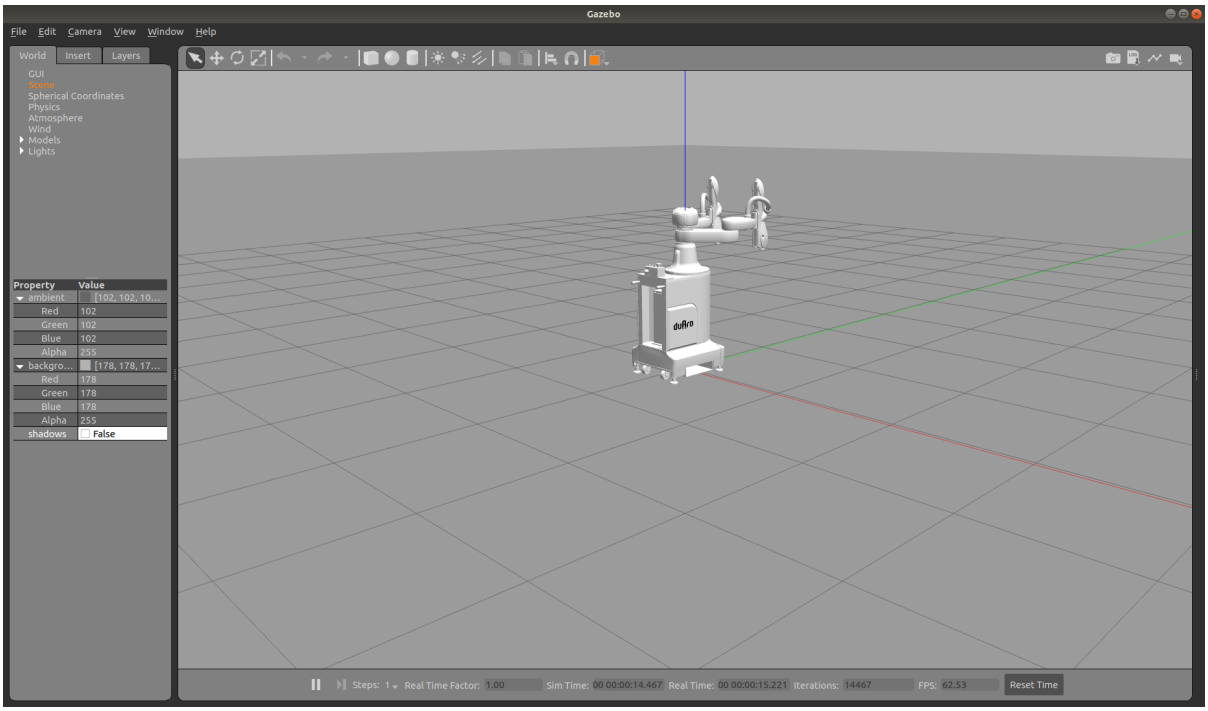
2.4.4 KHI duaro - Gazebo シミュレータ

Gazebo シミュレータの起動

ターミナルを 2 つ開きます.

ターミナル-1 : KHI duaro Gazebo シミュレータの起動

```
$ source /opt/ros/melodic/setup.bash
$ roslaunch khi_duaro_gazebo duaro_world.launch
```



しばらくすると次のようなメッセージが **ターミナル-1** に表示されます.

```
[INFO] [1557303124.764122, 0.426000]: Started controllers: joint_state_controller,␣
↪duaro_lower_arm_controller, duaro_upper_arm_controller
[go_initial-8] process has finished cleanly
log file: /home/robotuser/.ros/log/f5391a42-7168-11e9-931c-1c1bb5f26084/go_initial-
↪8*.log
```

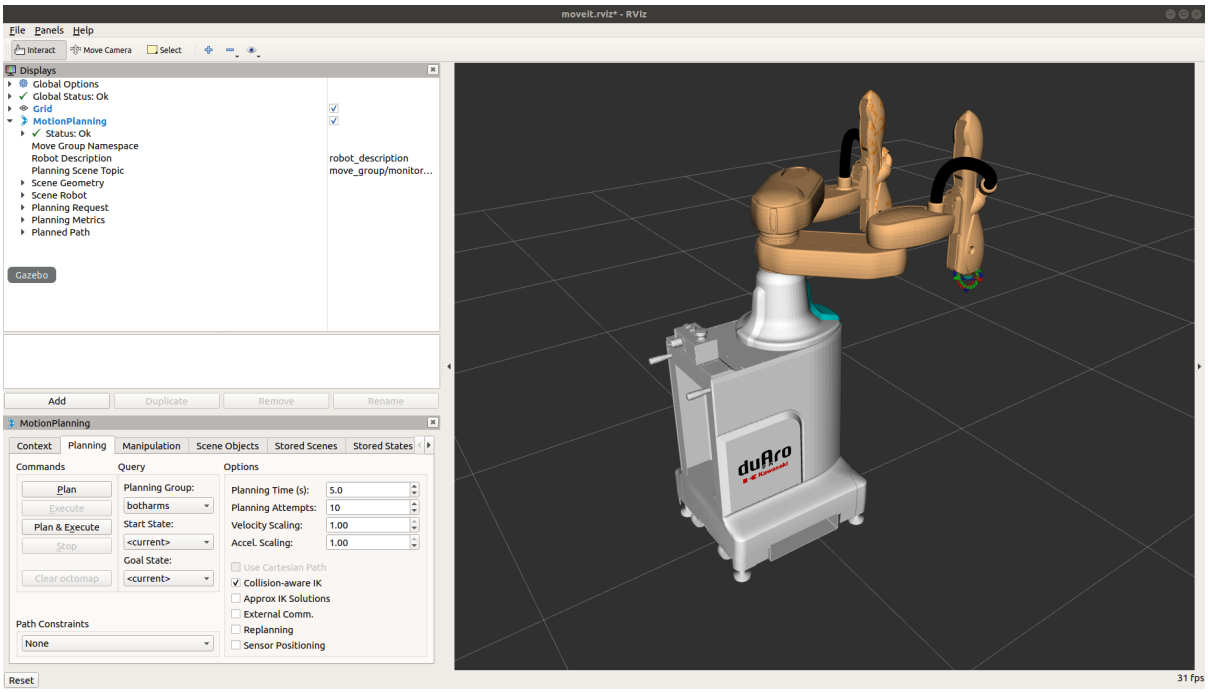
これで Gazebo シミュレータの準備は終了です.

MoveIt! の起動

2 つ目のターミナルで次のコマンドを実行して MoveIt! を起動します.

ターミナル-2 : MoveIt! の起動

```
source /opt/ros/melodic/setup.bash
roslaunch khi_duaro_moveit_config moveit_planning_execution.launch
```



これで MoveIt! の動作計画機能が利用できる状態になっています.

シミュレータの終了

シミュレータでの作業が終わりましたら全てのターミナルで Ctrl-C を入力することでシミュレータを終了します.

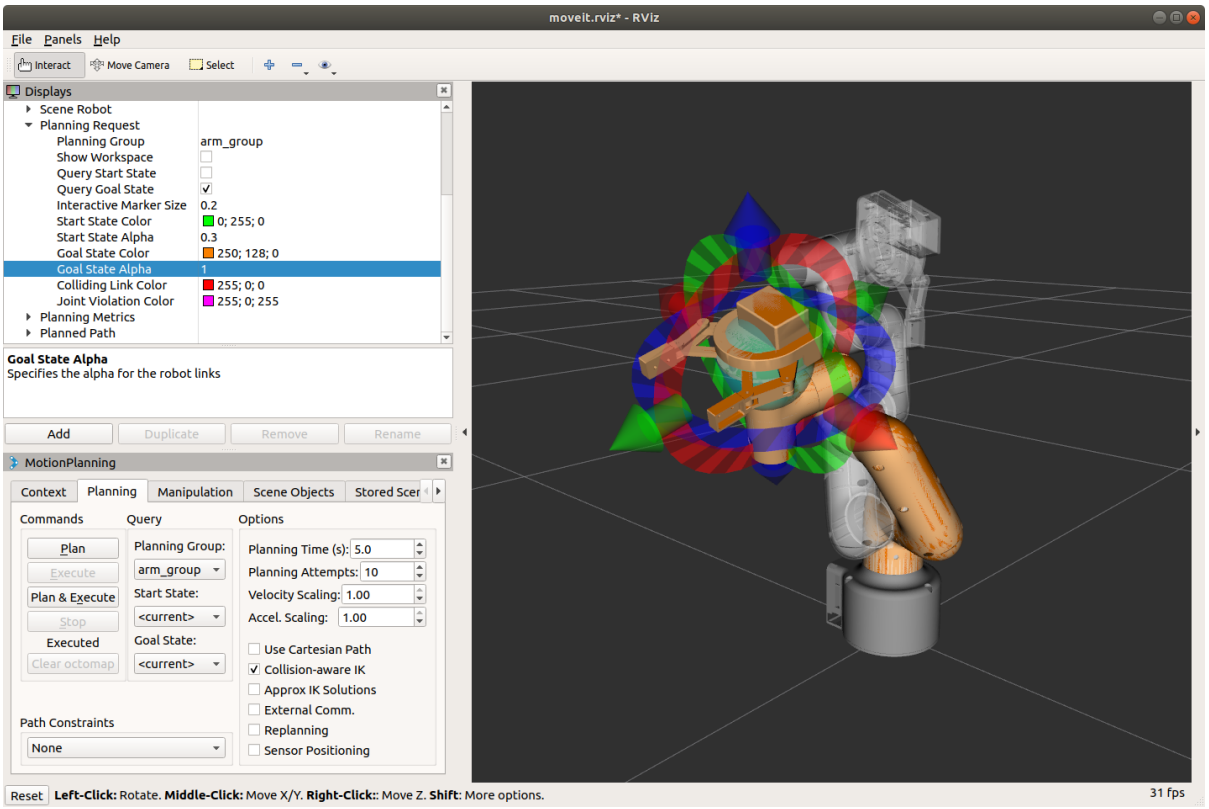
2.4.5 myCobot - MoveIt! シミュレータ

MoveIt! の起動

ターミナルで次のコマンドを実行して myCobot 280 用 MoveIt! を起動します.

ターミナル-1 : myCobot 280 用の MoveIt! の起動

```
source ~/catkin_ws/devel/setup.bash
roslaunch tork_moveit_tutorial demo.launch
```



これで MoveIt! の動作計画機能が利用できる状態になっています.

シミュレータの終了

シミュレータでの作業が終わりましたら全てのターミナルで Ctrl-C を入力することでシミュレータを終了します.

2.4.6 MoveIt! GUI での動作計画

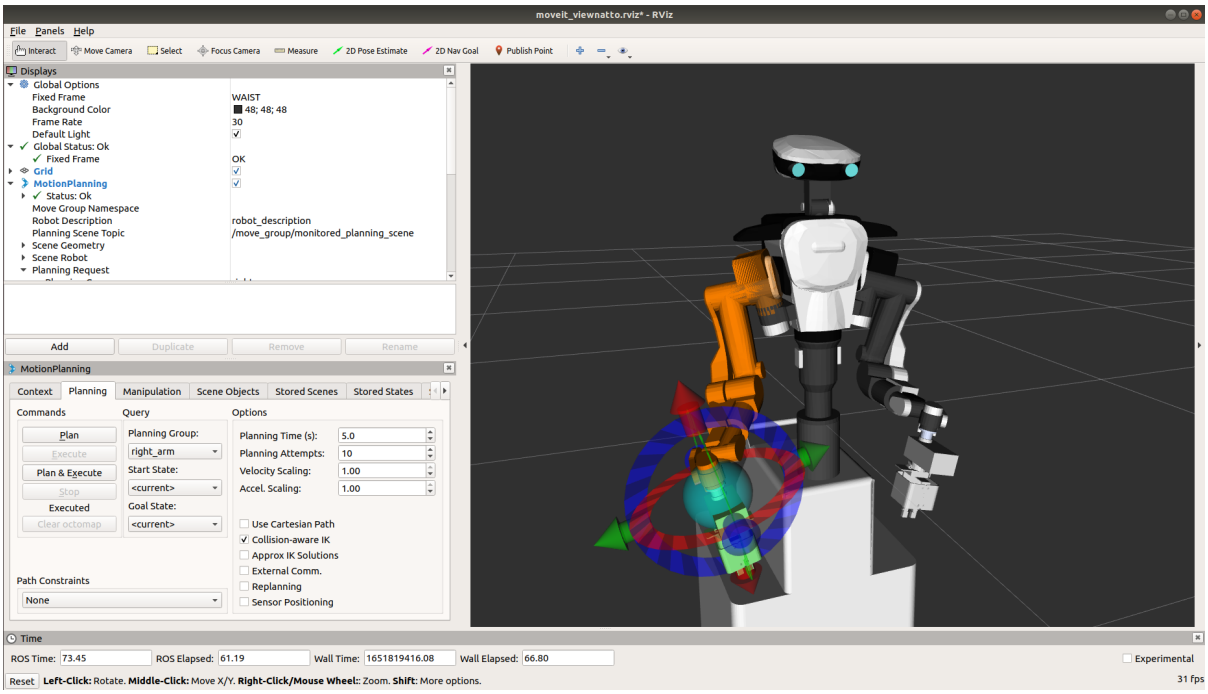
MoveIt! の動作計画機能を GUI から利用してみます.

MoveIt! / RViz (GUI) 上に表示されているロボットのエンドエフェクタのところに水色の球や赤緑青 (RGB) の矢印マークが表示されています. これは InteractiveMarker と呼ばれるもので MoveIt! の GUI でのマニピュレータ操作を行うためのものです.

- エンドエフェクタ: End Effector (EEF)
  - マニピュレータ先端に着けるハンド・グリッパ・工具などの機器

InteractiveMarker の球や矢印をマウスなどでドラッグ操作を行うとオレンジ色のマニピュレータがそのドラッグ操作に追従して動きます. これが目標姿勢となります. そこで MotionPlanning 子ウィンドウ内の Planning タブ内の `Plan and Execute` ボタンをクリックするとその目標姿勢に向かって MoveIt! が動作計画を行い, シミュレータのロボットが動作します.

動作計画だけを行いたい場合は `Plan` ボタンをクリックします.



このように MoveIt! の GUI 上で InteractiveMarker を動かして目標値を設定し，動作計画を行い実行するという操作は基本的にどのロボットでも共通です.

CHAPTER

THREE

プログラムでロボットを動かす

物理シミュレータと MoveIt! を使ってプログラムからロボットを操作します.

3.1 プログラムを入力して実行する

1 行もしくは数行ごとにプログラムを入力して実行し，各コマンドで何をしているのかを見てみます.

シミュレータや MoveIt! を起動し，それらを起動したターミナルとは別に新たにターミナルを起動して ROS の環境設定と対話的にプログラミングを行えるように準備します.

3.1.1 NEXTAGE OPEN の場合

「シミュレータと MoveIt! の起動」の「NEXTAGE OPEN - hrpsys シミュレータ」もしくは「NEXTAGE OPEN - Gazebo シミュレータ」を参照して hrpsys か Gazebo のどちらか一方のシミュレータと MoveIt! を起動します.

次に 1 行もしくは複数行ごとにプログラミングとその実行を行う対話的プログラミングコンソールを起動します.

ターミナル : 対話的プログラミングのコンソールの起動

```
$ source /opt/ros/melodic/setup.bash
$ rosrun tork_moveit_tutorial demo.py
```

特定の関節を動かす

プログラムから関節を動かすなどをするために「右腕」の `group` を作成します．次の `In[1]:` 以下につづくプログラムを入力して [Enter/Return] キーを押して 1 行ずつ実行してください.

- 補足 : `In [1]:` の `[ ]` の中の数字はプログラム入力するたびに更新されます．本チュートリアルにある入力例にある数字と一致しない場合がありますがそれは問題ありませんので，そのままチュートリアルを進めてください.

```
In [1]: group = MoveGroupCommander("right_arm")
[ INFO] [1511506815.441893962, 135.795000000]: TrajectoryExecution will use new␣
↪action capability.
[ INFO] [1511506815.442105792, 135.795000000]: Ready to take MoveGroup commands␣
↪for group right_arm.

In [2]:
```

- 注意：以下のように `TrajectoryExecution will use old action capability.` と表示される場合も有りますが問題ありません.

```
In [1]: group = MoveGroupCommander("right_arm")
[ INFO] [1511506815.441893962, 135.795000000]: TrajectoryExecution will use old␣
↪action capability.
[ INFO] [1511506815.442105792, 135.795000000]: Ready to take MoveGroup commands␣
↪for group right_arm.
```

- 注意 : MINAS TRA1 の場合はグループ名が異なります.

```
In [1]: group = MoveGroupCommander("manipulator")
```

- 注意 : myCobot の場合はグループ名が異なります.

```
In [1]: group = MoveGroupCommander("arm_group")
```

グループ group に含まれる関節の名前を get_joints() で調べます.

```
In [2]: group.get_joints()
Out[2]:
['RARM_JOINT0',
 'RARM_JOINT1',
 'RARM_JOINT2',
 'RARM_JOINT3',
 'RARM_JOINT4',
 'RARM_JOINT5']

In [3]:
```

上は NEXTAGE OPEN の右腕の場合の出力結果で RARM_JOINT0〜RARM_JOINT5 の 6 つの関節があることがわかります.

肘関節に相当する RARM_JOINT2 を動かしてみます.

set_joint_value_target() を使って関節の目標値を設定します. set_joint_value_target() に関節名 'RARM_JOINT2' と関節角度を -2.0（単位ラジアン [rad]）を渡します.

```
In [3]: group.set_joint_value_target( 'RARM_JOINT2', -2.0 )
```

関節角度目標を設定したので go() で動かします.

```
In [4]: group.go()
Out[4]: True
```

正常に動作が完了すると True が返ってきます. 肘関節 RARM_JOINT2 が少し屈曲したと思います.

同じ要領で他の右腕の関節もそれぞれ動かしてみます.

```
In [5]: group.set_joint_value_target( 'RARM_JOINT3', -0.78 )
In [6]: group.go()
Out[6]: True

In [7]: group.set_joint_value_target( 'RARM_JOINT4', 0.78 )
In [8]: group.go()
Out[8]: True

In [9]: group.set_joint_value_target( 'RARM_JOINT5', 0.78 )
In [10]: group.go()
Out[10]: True
```

腕全体の関節を動かす

set_joint_value_target() は 1 つの関節だけでなく腕全体の関節角度目標値のリストを渡すことで複数の関節を同時に動かすこともできます. group.get_joints() で調べたように NEXTAGE OPEN の右腕

には 6 つの関節があるので 6 つ分の関節角度目標値 [rad] を持ったリストを `set_joint_value_target()` に渡します.

```
In [11]: group.set_joint_value_target( [ 0.0, 0.0, -1.57, 0.0, 0.0, 0.0 ] )
In [12]: group.go()
Out[12]: True
```

NEXTAGE OPEN ロボットは初期姿勢に戻っていることと思います.

手先の位置を指定して動かす

手先（エンドエフェクタリンク）の位置を指定して腕を動かしてみます.

`set_position_target()` を使います. 手先の目標位置の座標 (x, y, z) をリスト `[ X, Y, Z ]` で `set_position_target()` に渡します.

位置座標の単位はメートル [m] です. 座標原点は NEXTAGE OPEN の場合, 腰部リンク (`/WAIST`) にあります. 各軸の方向は下記の右手座標系です.

- X 方向 : 正 = 前 / 負 = 後
- Y 方向 : 正 = 左 / 負 = 右
- Z 方向 : 正 = 上 / 負 = 下

```
In [21]: group.set_position_target( [ 0.3, -0.3, 0.3 ] )
```

`go()` を使って実行します.

```
In [22]: group.go()
Out[22]: True
```

手先の「位置」しか指定していないので手先の「姿勢」は思わぬ方を向いていることもあります.

手先の姿勢を指定して動かす

手先の姿勢を指定してロボットを動かしてみます.

`set_rpy_target()` を使います. 手先の目標姿勢の角度 (roll, pitch, yaw) をリスト `[ R, P, Y ]` で `set_rpy_target()` に渡します.

```
In [31]: group.set_rpy_target( [ 0.0, -2.36, 0.0 ] )
```

`go()` を使って実行します.

```
In [32]: group.go()
Out[32]: True
```

手先の「姿勢」しか指定していないので手先の「位置」は思わぬところにあることもあります.

- 注意：以下のようなエラーメッセージが表示された場合は, 再度 `group.go()` をお送りください.

```
[ INFO] [1515668193.145149146, 166.319999999]: ABORTED: Solution found but␣
↪controller failed during execution
```

手先の位置と姿勢を指定して動かす

手先の位置と姿勢を同時に指定して腕を動かすことができます.

`set_pose_target()` を使います. 次のいずれか 1 つを `set_pose_target()` に渡して手先の位置と姿勢を指定します.

- 位置座標と Roll/Pitch/Yaw 姿勢角の 6 つの数値のリスト `[x, y, z, rot_x, rot_y, rot_z]`

- 位置座標とクォータニオンの 7 つの数値のリスト `[x, y, z, qx, qy, qz, qw]`
- `Pose` 型
- `PoseStamped` 型

どれを渡しても `set_pose_target()` 内で判断して適切な処理がなされます.

- 参考 : GitHub - move_group.py 内の `set_pose_target()` の定義

https://github.com/ros-planning/moveit/blob/1.0.2/moveit_commander/src/moveit_commander/move_group.py#L252

それでは右手先の位置と姿勢を指定して腕を動かしてみます. まずは 位置と RPY 角 を `set_pose_target()` に渡して動作させます.

```
In [41]: group.set_pose_target( [ 0.4, -0.4, 0.15, 0.0, -1.57, 0.0 ] )
In [42]: group.go()
Out[42]: True
```

もう 1 つ位置・姿勢を指定して腕を動かしてみます.

```
In [43]: group.set_pose_target( [ 0.3, -0.3, 0.5, 0.0, -3.14, 0.0 ] )
In [44]: group.go()
Out[44]: True
```

今度は 位置とクォータニオン を `set_pose_target()` に渡して腕を動かしてみます.

```
In [45]: group.set_pose_target( [ 0.4, -0.4, 0.15, 0.0, -0.707, 0.0, 0.707 ] )
In [46]: group.go()
Out[46]: True
```

```
In [47]: group.set_pose_target( [ 0.3, -0.3, 0.5, 0.0, -1.0, 0.0, 0.0] )
In [48]: group.go()
Out[48]: True
```

- クォータニオン（四元数/しげんすう）
  - 空間上の姿勢を表現するための 4 つの成分をもつベクトル
    - 複素数から拡張された数体系
  - 回転の結合・補間の計算が容易
  - Roll/Pitch/Yaw のオイラー角により姿勢を表現した場合と比べて
    - 長所
      - ジンバルロックという特異点がない
      - 計算が速い
    - 短所
      - 直感的に分かりにくい
  - 比較的理解しやすい Roll/Pitch/Yaw からクォータニオンに変換することも可能
    - 3D Rotation Converter
      - https://www.andre-gaschler.com/rotationconverter/
    - ROS Wiki - Quaternion Basics
      - http://wiki.ros.org/tf2/Tutorials/Quaternions
  - ロボット以外では航空宇宙や 3D グラフィックスなどの分野で用いられる

更に `Pose` 型を `set_pose_target()` に渡して腕を動かしてみます.

```
In [49]: pose_target_1 = Pose()

In [50]: print( pose_target_1 )
position:
  x: 0.0
  y: 0.0
  z: 0.0
orientation:
  x: 0.0
  y: 0.0
  z: 0.0
  w: 0.0

In [51]: pose_target_1.position.x = 0.4
In [52]: pose_target_1.position.y = -0.4
In [53]: pose_target_1.position.z = 0.15
In [54]: pose_target_1.orientation.x = 0.0
In [55]: pose_target_1.orientation.y = -0.707
In [56]: pose_target_1.orientation.z = 0.0
In [57]: pose_target_1.orientation.w = 0.707

In [58]: print( pose_target_1 )
position:
  x: 0.4
  y: -0.4
  z: 0.15
orientation:
  x: 0.0
  y: -0.707
  z: 0.0
  w: 0.707

In [59]: group.set_pose_target( pose_target_1 )
In [60]: group.go()
Out[60]: True
```

ポーズをもう 1 つ指定して腕を動かします.

```
In [61]: pose_target_2 = Pose()

In [62]: print( pose_target_2 )
position:
  x: 0.0
  y: 0.0
  z: 0.0
orientation:
  x: 0.0
  y: 0.0
  z: 0.0
  w: 0.0

In [63]: pose_target_2.position.x = 0.3
In [64]: pose_target_2.position.y = -0.3
In [65]: pose_target_2.position.z = 0.5
In [66]: pose_target_2.orientation.y = -1.0

In [68]: print( pose_target_2 )
position:
  x: 0.3
  y: -0.3
  z: 0.5
orientation:
```

(continues on next page)

(continued from previous page)

```
  x: 0.0
  y: -1.0
  z: 0.0
  w: 0.0

In [69]: group.set_pose_target( pose_target_2 )
In [70]: group.go()
Out[70]: True
```

`set_pose_target()` に渡せるもう 1 つの型 `PoseStamped` 型については

- 発展的なロボットプログラミング - より複雑なロボット動作計画 / 姿勢の参照座標を指定する

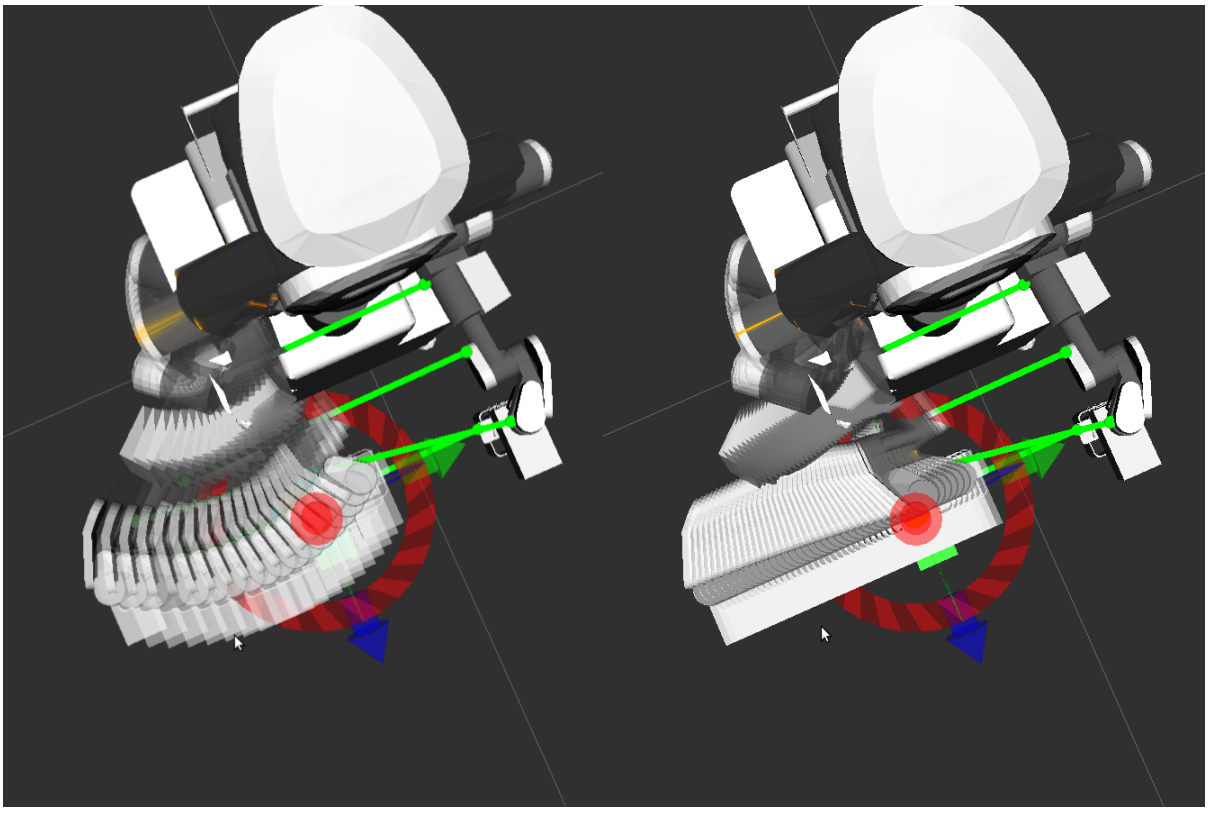
にて説明します.

直線補間軌道でロボットを動かす

`group.plan()` や `group.go()` を用いた動作計画では動作開始姿勢と目標姿勢の間の動作は各関節の開始角度と目標角度の間を補間した動作として計画されます. このことは開始姿勢や目標姿勢として指定した姿勢以外の動作途中におけるエンドエフェクタの姿勢は保証されないことを意味します.

エンドエフェクタを目標姿勢間で直線的に動作させたい場合は `group.compute_cartesian_path()` を用いて動作計画をします. `compute_cartesian_path()` はその名前のとおり, 直行座標（=デカルト座標: Cartesian Coordinates）における補間軌道（Path）を作成します.

`group.plan()` や `group.go()` で作成された動作計画（画像:左）と `group.compute_cartesian_path()` で作成された動作計画（画像:右）を比較すると次のようになります.



`compute_cartesian_path()` の具体的な使用方法は本項に続く項目の

- 連続した指令をロボットに送る
- 四角形や円に沿ってエンドエフェクタを動かす

を通して学習します.

連続した指令をロボットに送る

ロボットの複数の異なる姿勢を指示して動作計画と実行を行います.

複数の姿勢を指定した動作計画を行う場合も直線補間軌道でロボットを動かす場合と同じ `compute_cartesian_path()` を用います.

`compute_cartesian_path( self, waypoints, eef_step, jump_threshold, avoid_collisions = True )` には次のものを渡します.

- `waypoints` : エンドエフェクタが経由する姿勢のリスト
- `eef_step` : エンドエフェクタの姿勢を計算する間隔の距離
- `jump_threshold` : 軌道内の連続する点間の最大距離（`0.0` で無効）
- `avoid_collisions` : 干渉と運動学上の制約チェック（デフォルトは `True` でチェックする）

本チュートリアル手順に則って進めている場合，複数の姿勢のリスト `waypoints` が用意されているので内容を確かめてみます.

```
In [81]: print( waypoints )
[position:
  x: 0.4
  y: -0.4
  z: 0.15
orientation:
  x: 0.0
  y: -0.707
  z: 0.0
  w: 0.707, position:
  x: 0.4
  y: -0.2
  z: 0.15
orientation:
  x: 0.0
  y: -0.707
  z: 0.0
  w: 0.707, position:
  x: 0.3
  y: -0.3
  z: 0.5
orientation:
  x: 0.0
  y: -1.0
  z: 0.0
  w: 0.0, position:
  x: 0.3
  y: -0.5
  z: 0.5
orientation:
  x: 0.0
  y: -1.0
  z: 0.0
  w: 0.0]

In [82]:
```

姿勢のリスト `waypoints` を `compute_cartesian_path()` に渡して動作計画を作成します.

```
In [82]: ( plan, fraction ) = group.compute_cartesian_path( waypoints, 0.01, 0.0 )
```

compute_cartesian_path() で得られた計画 plan を group.execute() に渡してロボットで動作を実行します.

```
In [83]: group.execute( plan )
Out[83]: True
```

四角形や円に沿ってエンドエフェクタを動かす

エンドエフェクタを四角形や円に沿って動かすような場合も複数の異なる姿勢を指示して動作計画と実行を行います.

四角形や円に沿った複数の姿勢のリストをそれぞれ waypoints_rectangular と waypoints_circular として用意しているのでそれらを使います.

```
In [91]: print( waypoints_rectangular )
[position:
  x: 0.25
  y: 0.0
  z: 0.1
orientation:
  x: 0.0
  y: -0.707106781187
  z: 0.0
  w: 0.707106781187, position:
  x: 0.25
  y: -0.2
  z: 0.1
orientation:
  x: 0.0
  y: -0.707106781187
  z: 0.0
  w: 0.707106781187, position:
  x: 0.45
  y: -0.2
  z: 0.1
orientation:
  x: 0.0
  y: -0.707106781187
  z: 0.0
  w: 0.707106781187, position:
  x: 0.45
  y: 0.0
  z: 0.1
orientation:
  x: 0.0
  y: -0.707106781187
  z: 0.0
  w: 0.707106781187, position:
  x: 0.25
  y: 0.0
  z: 0.1
orientation:
  x: 0.0
  y: -0.707106781187
  z: 0.0
  w: 0.707106781187]

In [92]: ( plan, fraction ) = group.compute_cartesian_path( waypoints_rectangular,␣
↪0.01, 0.0 )

In [93]: group.execute( plan )
Out[93]: True
```

同様に円に沿った動作を行います.

```
In [94]: ( plan, fraction ) = group.compute_cartesian_path( waypoints_circular, 0.
↪01, 0.0 )

In [95]: group.execute( plan )
Out[95]: True
```

exit もしくは quit で終了します.

```
In [96]: exit
```

3.1.2 myCobot の場合

「シミュレータと MoveIt! の起動」の「myCobot - MoveIt! シミュレータ」を参照して MoveIt! を起動します.

次に 1 行もしくは複数行ごとにプログラミングとその実行を行う対話的プログラミングコンソールを起動します.

ターミナル : 対話的プログラミングのコンソールの起動

```
source /opt/ros/melodic/setup.bash
rosrun tork_moveit_tutorial demo.py
```

実機を動かす場合

先に「実機の使い方 – myCobot の場合」を参照してください.

そのあと，上に書かれたコマンドの代わりに，以下を実行してください.

ターミナル-1 : myCobot 280 とパソコンの接続プログラムの起動

```
source ~/catkin_ws/devel/setup.bash
roslaunch tork_moveit_tutorial mycobot_interface.launch
```

ターミナル-2 : myCobot 280 用の MoveIt! の起動

```
source ~/catkin_ws/devel/setup.bash
roslaunch tork_moveit_tutorial demo.launch mode:=real
```

ターミナル-3 : 対話的プログラミングのコンソールの起動

```
source /opt/ros/melodic/setup.bash
rosrun tork_moveit_tutorial demo.py
```

特定の関節を動かす

プログラムから関節を動かすなどをするために「右腕」の group を作成します．次の In[1]: 以下につづくプログラムを入力して [Enter/Return] キーを押して 1 行ずつ実行してください.

- 補足 : In [1]: の [ ] の中の数字はプログラム入力するたびに更新されます．本チュートリアルにある入力例にある数字と一致しない場合がありますがそれは問題ありませんので，そのままチュートリアルを進めてください.

```
In [1]: group = MoveGroupCommander("arm_group")
[ INFO] [1511506815.441893962, 135.795000000]: TrajectoryExecution will use new␣
↪action capability.
[ INFO] [1511506815.442105792, 135.795000000]: Ready to take MoveGroup commands␣
↪for group right_arm.
```

(continues on next page)

(continued from previous page)

```
In [2]:
```

- 注意：以下のように `TrajectoryExecution will use old action capability.` と表示される場合も有りますが問題ありません.

```
In [1]: group = MoveGroupCommander("arm_group")
[ INFO] [1511506815.441893962, 135.795000000]: TrajectoryExecution will use old␣
↪action capability.
[ INFO] [1511506815.442105792, 135.795000000]: Ready to take MoveGroup commands␣
↪for group right_arm.
```

グループ `group` に含まれる関節の名前を `get_joints()` で調べます.

```
In [2]: group.get_joints()
Out[2]:
['joint1', 'joint2', 'joint3', 'joint4', 'joint5', 'joint6']

In [3]:
```

上は myCobot 280 の場合の出力結果で `joint1`〜`joint6` の 6 つの関節があることがわかります.

肘関節に相当する `joint3` を動かしてみます.

`set_joint_value_target()` を使って関節の目標値を設定します. `set_joint_value_target()` に関節名 `'joint3'` と関節角度を `-2.0`（単位ラジアン [rad]）を渡します.

```
In [3]: group.set_joint_value_target( 'joint3', -2.0 )
```

関節角度目標を設定したので `go()` で動かします.

```
In [4]: group.go()
Out[4]: True
```

正常に動作が完了すると `True` が返ってきます. 肘関節 `joint3` が少し屈曲したと思います.

![myCobot - result of group.set_joint_value_target(‘ joint3 ’, -2.0)](images/melodic/3.1_In_[3]_set_joint_value_target(‘ joint3 ’, -2.0).jpg)

同じ要領で他の右腕の関節もそれぞれ動かしてみます.

```
In [5]: group.set_joint_value_target( 'joint4', -0.78 )
In [6]: group.go()
Out[6]: True
```

![myCobot - result of group.set_joint_value_target(‘ joint4 ’, -0.78)](images/melodic/3.1_In_[5]_set_joint_value_target(‘ joint4 ’, -0.78).jpg)

```
In [7]: group.set_joint_value_target( 'joint5', 0.78 )
In [8]: group.go()
Out[8]: True
```

![myCobot - result of group.set_joint_value_target(‘ joint5 ’, 0.78)](images/melodic/3.1_In_[7]_set_joint_value_target(‘ joint5 ’, 0.78).jpg)

```
In [9]: group.set_joint_value_target( 'joint6', 0.78 )
In [10]: group.go()
Out[10]: True
```

![myCobot - result of group.set_joint_value_target(‘ joint6 ’, 0.78)](images/melodic/3.1_In_[9]_set_joint_value_target(‘ joint6 ’, 0.78).jpg)

腕全体の関節を動かす

`set_joint_value_target()` は 1 つの関節だけでなく腕全体の関節角度目標値のリストを渡すことで複数の関節を同時に動かすこともできます．`group.get_joints()` で調べたように myCobot には 6 つの関節があるので 6 つ分の関節角度目標値 [rad] を持ったリストを `set_joint_value_target()` に渡します．

```
In [11]: group.set_joint_value_target( [ 0.0, 0.0, 0, 0.0, 0.0, 0.0 ] )
In [12]: group.go()
Out[12]: True
```

![myCobot - result of group.set_joint_value_target([0.0, 0.0, 0, 0.0, 0.0, 0.0])](images/melodic/3.1_In_[11]_set_joint_value_target([0.0, 0.0, 0.0, 0.0, 0.0, 0.0]).jpg)

myCobot ロボットは初期姿勢に戻っていることと思います．

手先の位置を指定して動かす

手先（エンドエフェクタリンク）の位置を指定して腕を動かしてみます．

`set_position_target()` を使います．手先の目標位置の座標 (x, y, z) をリスト `[ X, Y, Z ]` で `set_position_target()` に渡します．

位置座標の単位はメートル [m] です．座標原点は myCobot の場合，ベースリンク (`/base`) にあります．各軸の方向は下記の右手座標系です．

- X 方向 : 正 = 前 / 負 = 後
- Y 方向 : 正 = 左 / 負 = 右
- Z 方向 : 正 = 上 / 負 = 下

```
In [21]: group.set_position_target( [ 0.1, -0.1, 0.1 ] )
```

`go()` を使って実行します．

```
In [22]: group.go()
Out[22]: True
```

手先の「位置」しか指定していないので手先の「姿勢」は思わぬ方を向いていることもあります．

![myCobot - one result of group.set_position_target([0.1, -0.1, 0.1])](images/melodic/3.1_In_[21]_set_position_target([0.1, -0.1, 0.1]).jpg)

手先の姿勢を指定して動かす

手先の姿勢を指定してロボットを動かしてみます．

`set_rpy_target()` を使います．手先の目標姿勢の角度 (roll, pitch, yaw) をリスト `[ R, P, Y ]` で `set_rpy_target()` に渡します．

```
In [31]: group.set_rpy_target( [ 0.0, -2.36, 0.0 ] )
```

`go()` を使って実行します．

```
In [32]: group.go()
Out[32]: True
```

手先の「姿勢」しか指定していないので手先の「位置」は思わぬところにあることもあります．

![myCobot - one result of group.set_rpy_target([0.0, -2.36, 0.0])](images/melodic/3.1_In_[31]_set_rpy_target([0.0, -2.36, 0.0]).jpg)

- 注意：以下のようなエラーメッセージが表示された場合は，再度 `group.go()` をお送りください．

```
[ INFO] [1515668193.145149146, 166.319999999]: ABORTED: Solution found but␣
↪controller failed during execution
```

手先の位置と姿勢を指定して動かす

手先の位置と姿勢を同時に指定して腕を動かすことができます.

`set_pose_target()` を使います. 次のいずれか 1 つを `set_pose_target()` に渡して手先の位置と姿勢を指定します.

- 位置座標と Roll/Pitch/Yaw 姿勢角の 6 つの数値のリスト `[x, y, z, rot_x, rot_y, rot_z]`
- 位置座標とクォータニオンの 7 つの数値のリスト `[x, y, z, qx, qy, qz, qw]`
- `Pose` 型
- `PoseStamped` 型

どれを渡しても `set_pose_target()` 内で判断して適切な処理がなされます.

- 参考 : GitHub - move_group.py 内の `set_pose_target()` の定義

https://github.com/ros-planning/moveit/blob/1.0.2/moveit_commander/src/moveit_commander/move_group.py#L252

それでは右手先の位置と姿勢を指定して腕を動かしてみます. まずは 位置と RPY 角 を `set_pose_target()` に渡して動作させます.

```
In [41]: group.set_pose_target( [ 0.1, -0.1, 0.1, 0, -1.57, 0 ] )
In [42]: group.go()
Out[42]: True
```


もう 1 つ位置・姿勢を指定して腕を動かしてみます.

```
In [43]: group.set_pose_target( [ 0.1, -0.1, 0.2, 0, -3.14, 0 ] )
In [44]: group.go()
Out[44]: True
```


今度は 位置とクォータニオン を `set_pose_target()` に渡して腕を動かしてみます.

```
In [45]: group.set_pose_target( [ 0.1, -0.1, 0.1, 0.0, -0.707, 0.0, 0.707 ] )
In [46]: group.go()
Out[46]: True
```



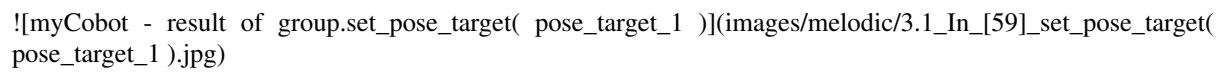
```
In [47]: group.set_pose_target( [ 0.1, -0.1, 0.2, 0.0, -1.0, 0.0, 0.0] )
In [48]: group.go()
Out[48]: True
```


- クォータニオン（四元数/しげんすう）
  - 空間上の姿勢を表現するための 4 つの成分をもつベクトル
    - 複素数から拡張された数体系
  - 回転の結合・補間の計算が容易

- Roll/Pitch/Yaw のオイラー角により姿勢を表現した場合と比べて
  - 長所
    - ジンバルロックという特異点がない
    - 計算が速い
  - 短所
    - 直感的に分かりにくい
- 比較的理解しやすい Roll/Pitch/Yaw からクォータニオンに変換することも可能
  - 3D Rotation Converter
    - https://www.andre-gaschler.com/rotationconverter/
  - ROS Wiki - Quaternion Basics
    - http://wiki.ros.org/tf2/Tutorials/Quaternions
- ロボット以外では航空宇宙や 3D グラフィックスなどの分野で用いられる

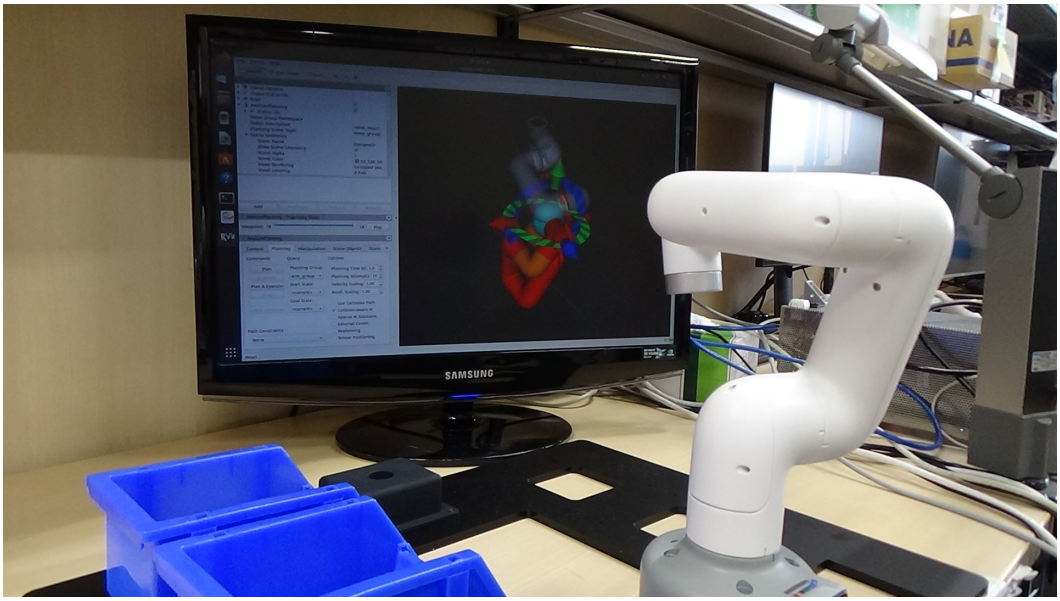
更に `Pose` 型を `set_pose_target()` に渡して腕を動かしてみます.

```
In [49]: pose_target_1 = Pose()

In [50]: print( pose_target_1 )
position:
  x: 0.0
  y: 0.0
  z: 0.0
orientation:
  x: 0.0
  y: 0.0
  z: 0.0
  w: 0.0

In [51]: pose_target_1.position.x = 0.1
In [52]: pose_target_1.position.y = -0.1
In [53]: pose_target_1.position.z = 0.1
In [54]: pose_target_1.orientation.x = 0.0
In [55]: pose_target_1.orientation.y = -0.707
In [56]: pose_target_1.orientation.z = 0.0
In [57]: pose_target_1.orientation.w = 0.707

In [58]: print( pose_target_1 )
position:
  x: 0.1
  y: -0.1
  z: 0.1
orientation:
  x: 0.0
  y: -0.707
  z: 0.0
  w: 0.707

In [59]: group.set_pose_target( pose_target_1 )
In [60]: group.go()
Out[60]: True
```



ポーズをもう 1 つ指定して腕を動かします.

```
In [61]: pose_target_2 = Pose()

In [62]: print( pose_target_2 )
position:
  x: 0.0
  y: 0.0
  z: 0.0
orientation:
  x: 0.0
  y: 0.0
  z: 0.0
  w: 0.0

In [63]: pose_target_2.position.x = 0.1
In [64]: pose_target_2.position.y = -0.1
In [65]: pose_target_2.position.z = 0.2
In [66]: pose_target_2.orientation.y = -1.0

In [68]: print( pose_target_2 )
position:
  x: 0.1
  y: -0.1
  z: 0.2
orientation:
  x: 0.0
  y: -1.0
  z: 0.0
  w: 0.0

In [69]: group.set_pose_target( pose_target_2 )
In [70]: group.go()
Out[70]: True
```



`set_pose_target()` に渡せるもう 1 つの型 `PoseStamped` 型については

- 発展的なロボットプログラミング - より複雑なロボット動作計画 / 姿勢の参照座標を指定する

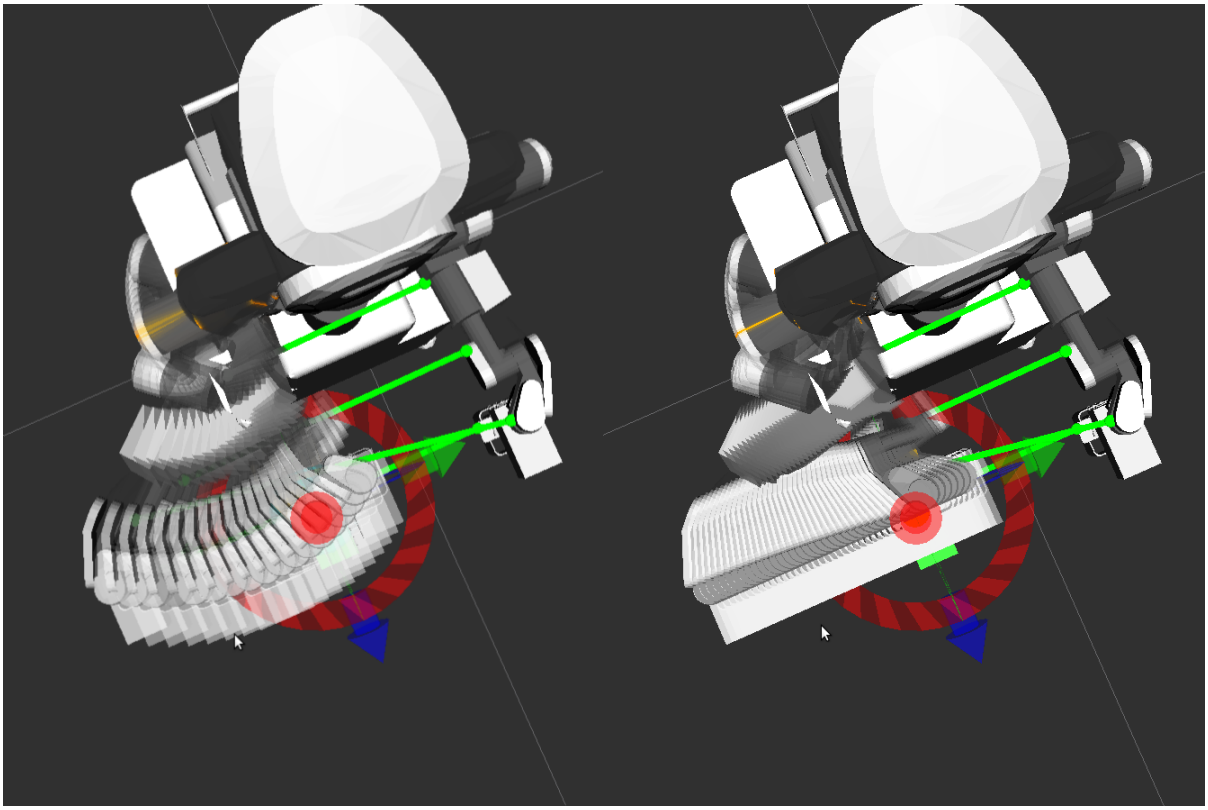
にて説明します.

直線補間軌道でロボットを動かす

`group.plan()` や `group.go()` を用いた動作計画では動作開始姿勢と目標姿勢の間の動作は各関節の開始角度と目標角度の間を補間した動作として計画されます．このことは開始姿勢や目標姿勢として指定した姿勢以外の動作途中におけるエンドエフェクタの姿勢は保証されないことを意味します．

エンドエフェクタを目標姿勢間で直線的に動作させたい場合は `group.compute_cartesian_path()` を用いて動作計画をします．`compute_cartesian_path()` はその名前のとおり，直行座標（=デカルト座標: Cartesian Coordinates）における補間軌道（Path）を作成します．

`group.plan()` や `group.go()` で作成された動作計画（画像:左）と `group.compute_cartesian_path()` で作成された動作計画（画像:右）を比較すると次のようになります．



`compute_cartesian_path()` の具体的な使用方法は本項に続く項目の

- 連続した指令をロボットに送る
- 四角形や円に沿ってエンドエフェクタを動かす

を通して学習します．

連続した指令をロボットに送る

ロボットの複数の異なる姿勢を指示して動作計画と実行を行います．

複数の姿勢を指定した動作計画を行う場合も直線補間軌道でロボットを動かす場合と同じ `compute_cartesian_path()` を用います．

`compute_cartesian_path( self, waypoints, eef_step, jump_threshold, avoid_collisions = True )` には次のものを渡します．

- `waypoints`：エンドエフェクタが経由する姿勢のリスト
- `eef_step`：エンドエフェクタの姿勢を計算する間隔の距離
- `jump_threshold`：軌道内の連続する点間の最大距離（`0.0` で無効）

- avoid_collisions : 干渉と運動学上の制約チェック（デフォルトは True でチェックする）

本チュートリアル手順に則って進めている場合，myCobot 用の複数の姿勢のリスト waypoints_mycobot が用意されているので内容を確かめてみます.

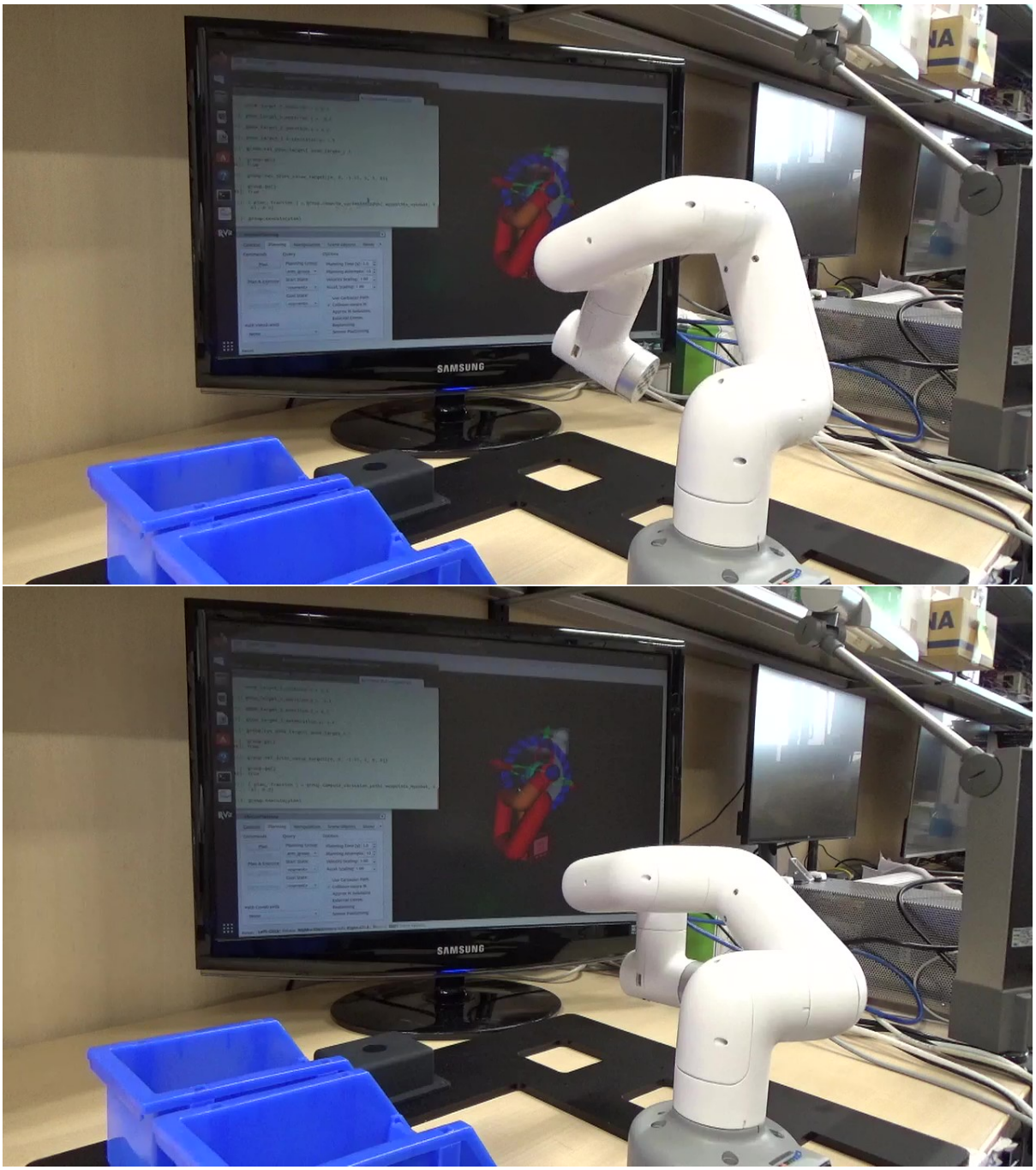
```
In [2]: print( waypoints_mycobot )
[position:
  x: 0.1
  y: -0.1
  z: 0.1
orientation:
  x: 0.0
  y: -0.707
  z: 0.0
  w: 0.707, position:
  x: 0.1
  y: -0.15
  z: 0.1
orientation:
  x: 0.0
  y: -0.707
  z: 0.0
  w: 0.707, position:
  x: 0.1
  y: -0.1
  z: 0.2
orientation:
  x: 0.0
  y: -1.0
  z: 0.0
  w: 0.0, position:
  x: 0.1
  y: -0.2
  z: 0.2
orientation:
  x: 0.0
  y: -1.0
  z: 0.0
  w: 0.0]
```

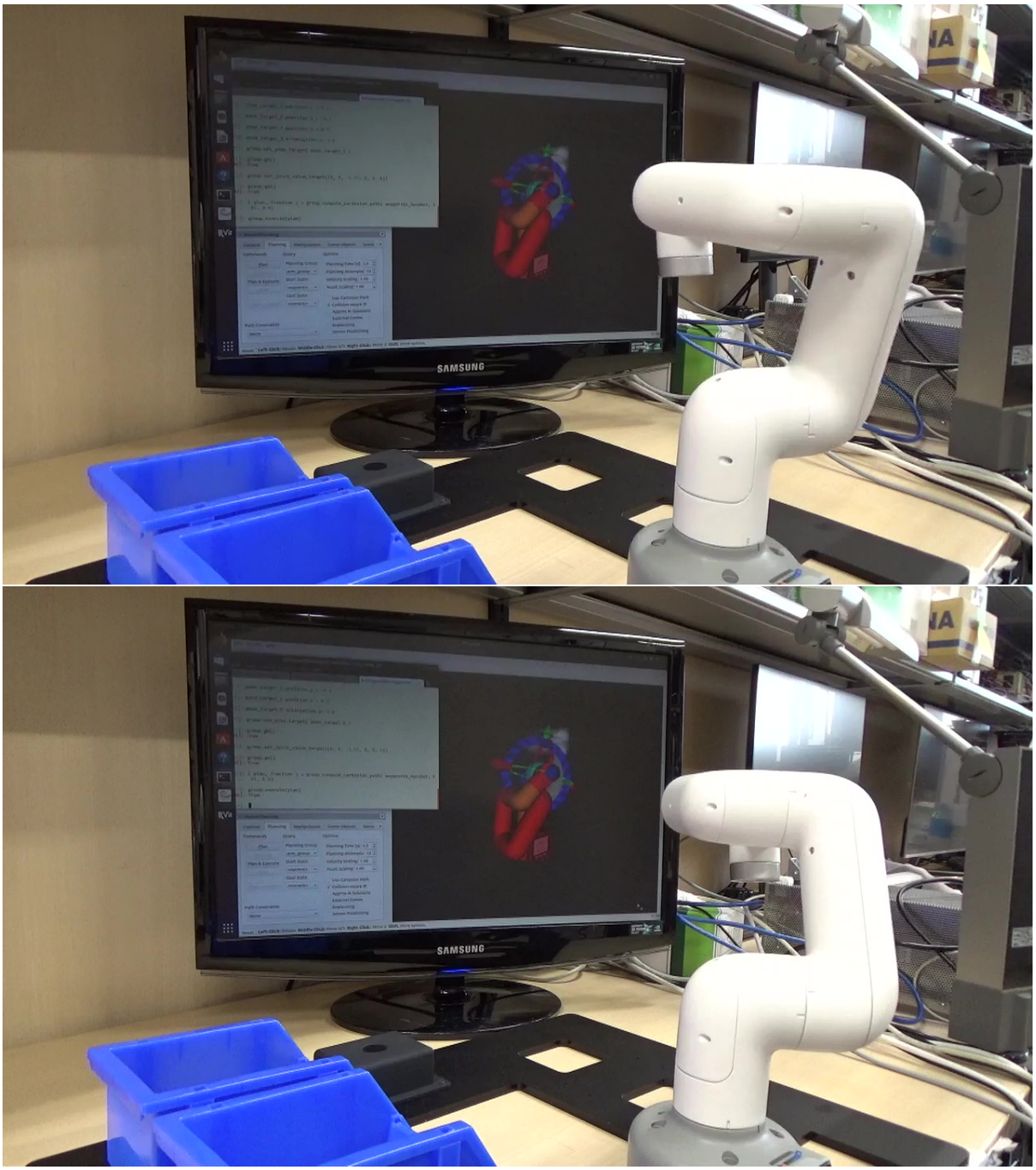
myCobot を特異姿勢でない姿勢にしてから，姿勢のリスト waypoints_mycobot を compute_cartesian_path() に渡して動作計画を作成します.

```
In [3]: group.set_joint_value_target([0, 0, -1.57, 0, 0, 0])

In [4]: group.go()
Out[4]: True

In [5]: ( plan, fraction ) = group.compute_cartesian_path( waypoints_mycobot, 0.01,
 0.0)
```

compute_cartesian_path() で得られた計画 plan を group.execute() に渡してロボットで動作を実行します.

```
In [6]: group.execute(plan)
Out[6]: True
```


SAMSUNG
SAMSUNG



四角形や円に沿ってエンドエフェクタを動かす

エンドエフェクタを四角形や円に沿って動かすような場合も複数の異なる姿勢を指示して動作計画と実行を行います.

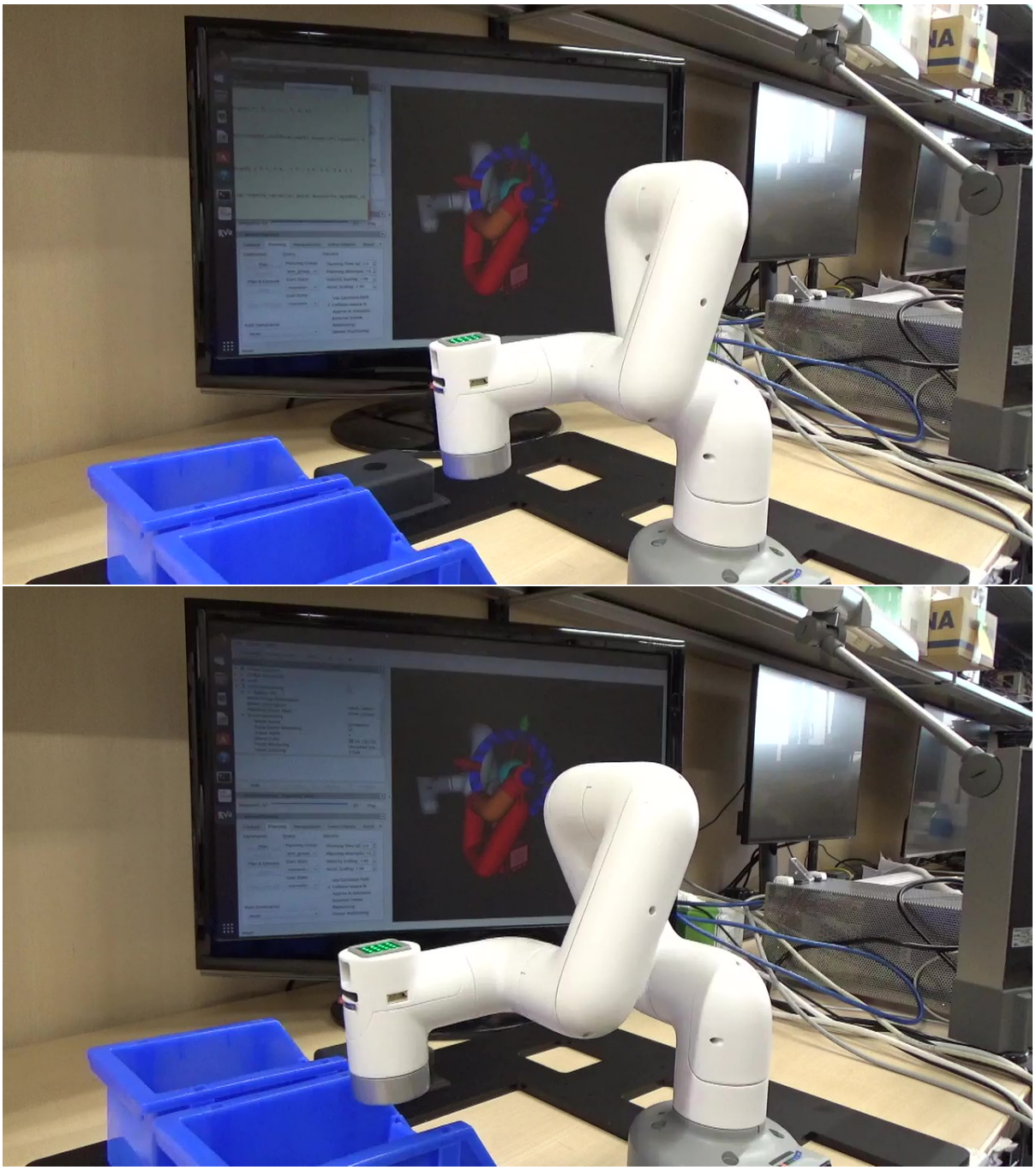
四角形や円に沿った複数の姿勢のリストをそれぞれ `waypoints_rectangular` と `waypoints_circular` として用意しているのでそれらを使います.

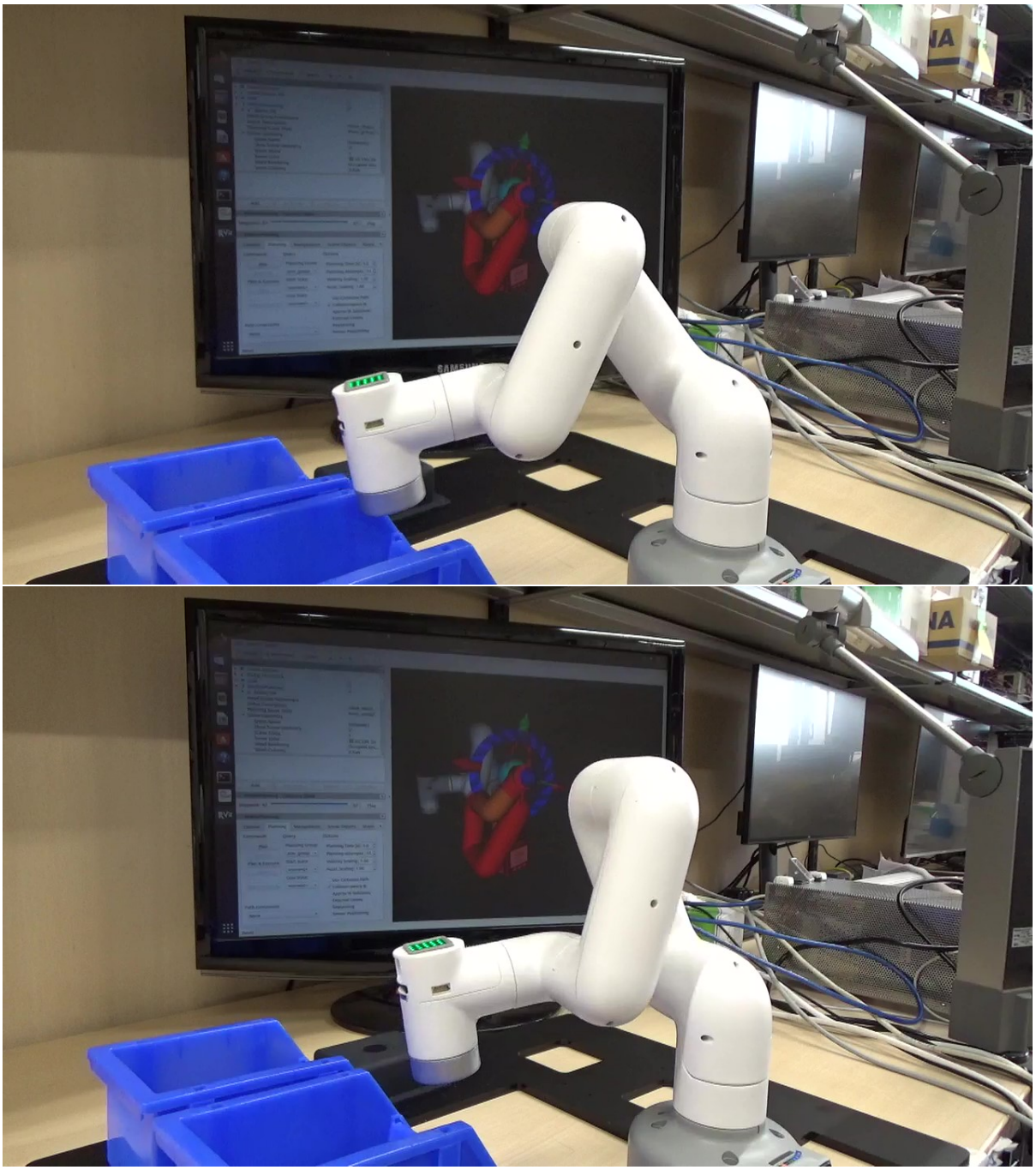
```
In [10]: print( waypoints_mycobot_rectangular )
[position:
  x: 0.1
  y: 0.1
  z: 0.1
orientation:
  x: 0.0
```

(continues on next page)

(continued from previous page)

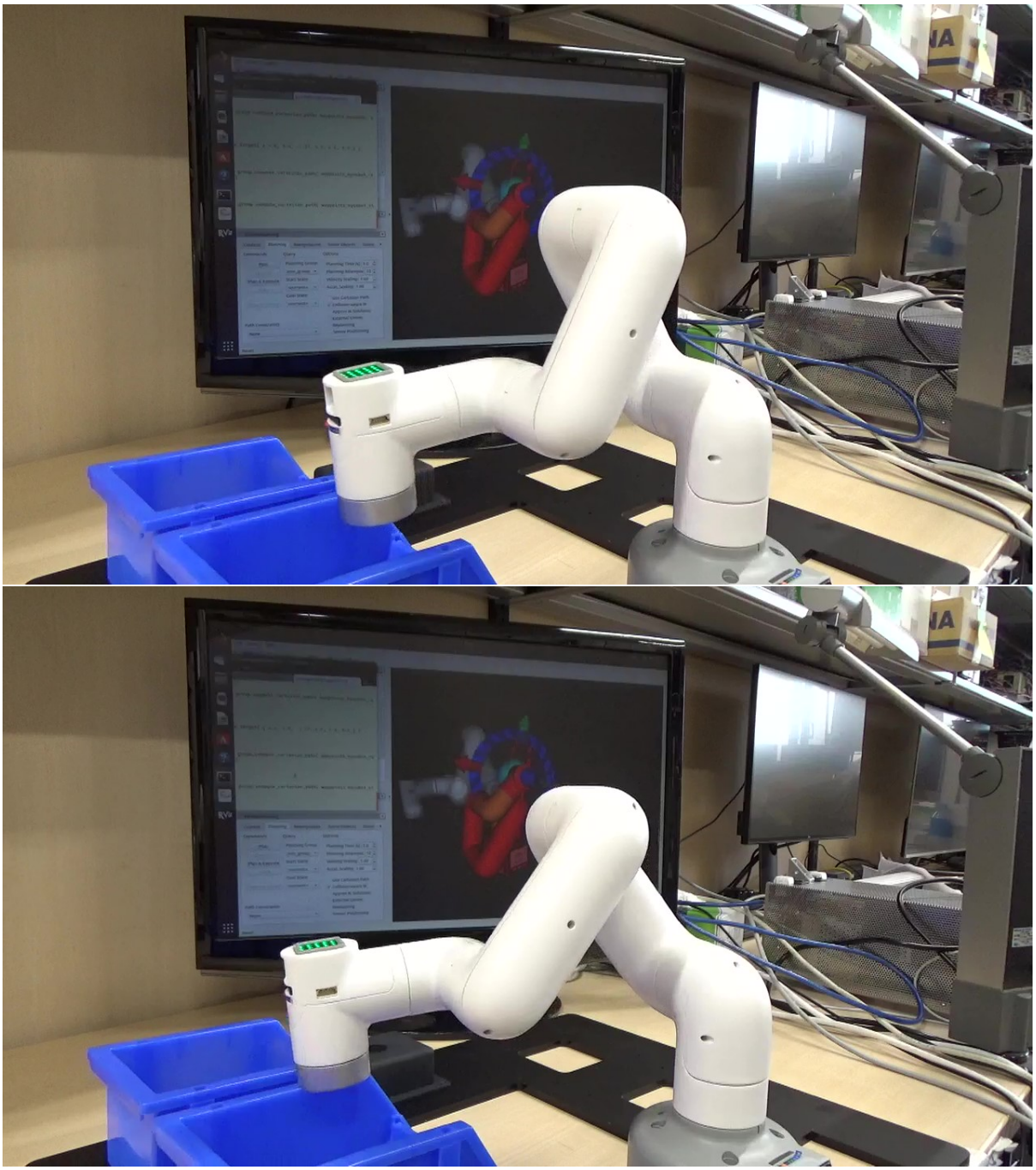
```
  y: -1.0
  z: 0.0
  w: 6.123233995736766e-17, position:
  x: 0.1
  y: 0.15
  z: 0.1
orientation:
  x: 0.0
  y: -1.0
  z: 0.0
  w: 6.123233995736766e-17, position:
  x: 0.15
  y: 0.15
  z: 0.1
orientation:
  x: 0.0
  y: -1.0
  z: 0.0
  w: 6.123233995736766e-17, position:
  x: 0.15
  y: 0.1
  z: 0.1
orientation:
  x: 0.0
  y: -1.0
  z: 0.0
  w: 6.123233995736766e-17, position:
  x: 0.1
  y: 0.1
  z: 0.1
orientation:
  x: 0.0
  y: -1.0
  z: 0.0
  w: 6.123233995736766e-17]

In [11]: group.set_joint_value_target( [ 0.0, 0.0, -1.57, 0.0, 0.0, 0.0 ] )

In [12]: group.go()
Out[12]: True

In [13]: ( plan, fraction ) = group.compute_cartesian_path( waypoints_mycobot_
↪rectangular, 0.01, 0.0)

In [14]: group.execute(plan)
Out[14]: True
```

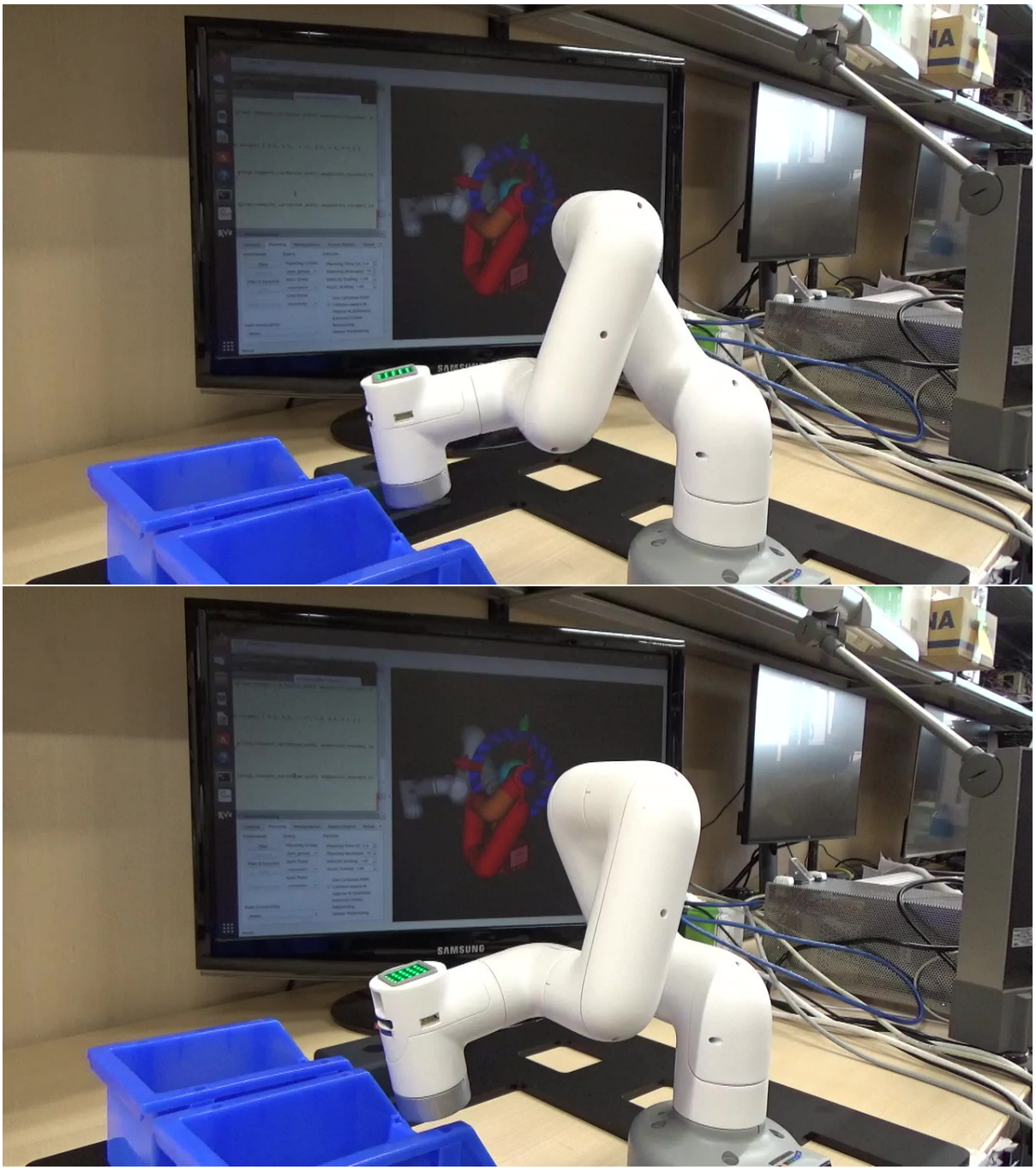




同様に円に沿った動作を行います.

```
In [13]: ( plan, fraction ) = group.compute_cartesian_path( waypoints_mycobot_
↪circular, 0.01, 0.0)

In [14]: group.execute(plan)
Out[14]: True
```





exit もしくは quit で終了します.

```
In [15]: exit
```

3.2 プログラムファイルを実行する

コンソールでのプログラム実行は学習や各コマンドの動作確認には良いのですが，毎回同じことを入力して実行するのは大変なので命令が書かれたプログラムファイルを実行します.

プログラムファイルを実行する前に物理シミュレータと MoveIt! を起動しておきます.

3.2.1 NEXTAGE OPEN の場合

ターミナル-1

```
$ source /opt/ros/melodic/setup.bash
$ roslaunch nextage_gazebo nextage_world.launch
```

- メモ : hrpsys (RTM) シミュレータでも可

ターミナル-2

```
$ source /opt/ros/melodic/setup.bash
$ roslaunch nextage_moveit_config moveit_planning_execution.launch
```

動作プログラムファイルを実行します.

ターミナル-3

```
$ source /opt/ros/melodic/setup.bash
$ rosrun tork_moveit_tutorial nextage_moveit_tutorial_poses.py
```

nextage_moveit_tutorial_poses.py

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

from tork_moveit_tutorial import *


if __name__ == '__main__':

    init_node()

    group = MoveGroupCommander("right_arm")

    # Pose Target 1
    rospy.loginfo( "Start Pose Target 1")
    pose_target_1 = Pose()

    pose_target_1.position.x = 0.4
    pose_target_1.position.y = -0.4
    pose_target_1.position.z = 0.15
    pose_target_1.orientation.x = 0.0
    pose_target_1.orientation.y = -0.707
    pose_target_1.orientation.z = 0.0
    pose_target_1.orientation.w = 0.707

    rospy.loginfo( "Set Target to Pose:n{}".format( pose_target_1 ) )
    group.set_pose_target( pose_target_1 )
    group.go()

    # Pose Target 2
    rospy.loginfo( "Start Pose Target 2")
    pose_target_2 = Pose()

    pose_target_2.position.x = 0.3
    pose_target_2.position.y = -0.3
    pose_target_2.position.z = 0.5
    pose_target_2.orientation.y = -1.0

    rospy.loginfo( "Set Target to Pose:n{}".format( pose_target_2 ) )
    group.set_pose_target( pose_target_2 )
    group.go()
```

nextage_moveit_tutorial_poses.py で実行している内容は

- **手先の位置と姿勢を指定して動かす** - `set_pose_target()` `go()`

と基本的に同じです．主に異なるのは下記の部分です．

- `print()` を ROS のログに出力する `rospy.loginfo()` に変更
- `rospy.loginfo()` の表示内容もどの箇所の実行ログかわかるように変更

3.2.2 MINAS TRA1 の場合

ターミナル-1

```
$ source /opt/ros/melodic/setup.bash
$ roslaunch tra1_bringup tra1_bringup.launch simulation:=true
```

ターミナル-2

```
$ source /opt/ros/melodic/setup.bash
$ roslaunch tra1_bringup tra1_moveit.launch
```

動作プログラムファイルを実行します．

ターミナル-3

```
$ source /opt/ros/melodic/setup.bash
$ rosrun tork_moveit_tutorial tra1_moveit_tutorial_poses.py
```

tra1_moveit_tutorial_poses.py

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

from tork_moveit_tutorial import *


if __name__ == '__main__':

    init_node()

    group = MoveGroupCommander("manipulator")

    # Pose Target 1
    rospy.loginfo( "Start Pose Target 1")
    pose_target_1 = Pose()

    pose_target_1.position.x = 0.0
    pose_target_1.position.y = -0.6
    pose_target_1.position.z = 0.3
    pose_target_1.orientation.x = 1.0
    pose_target_1.orientation.y = 0.0
    pose_target_1.orientation.z = 0.0
    pose_target_1.orientation.w = 0.0

    rospy.loginfo( "Set Target to Pose:n{}".format( pose_target_1 ) )
    group.set_pose_target( pose_target_1 )
    group.go()

    # Pose Target 2
    rospy.loginfo( "Start Pose Target 2")
    pose_target_2 = Pose()

    pose_target_2.position.x = 0.6
```

(continues on next page)

(continued from previous page)

```
    pose_target_2.position.y = 0.0
    pose_target_2.position.z = 0.3
    pose_target_2.orientation.x = -0.707
    pose_target_2.orientation.y = -0.707

    rospy.loginfo( "Set Target to Pose:n{}".format( pose_target_2 ) )
    group.set_pose_target( pose_target_2 )
    group.go()
```

他のロボットの動作計画・動作の実行ファイルとの相違点は次のとおりです.

- `group = MoveGroupCommander()` に渡すグループ名を `"manipulator"` に変更
- ターゲットポーズの位置・姿勢を MINAS TRA1 の機構に適したものに変更

3.2.3 KHI duaro の場合

ターミナル-1

```
$ source /opt/ros/melodic/setup.bash
$ roslaunch khi_duaro_gazebo duaro_world.launch
```

ターミナル-2

```
$ source /opt/ros/melodic/setup.bash
$ roslaunch khi_duaro_moveit_config moveit_planning_execution.launch
```

動作プログラムファイルを実行します.

ターミナル-3

```
$ source /opt/ros/melodic/setup.bash
$ rosrun tork_moveit_tutorial duaro_moveit_tutorial_poses.py
```

duaro_moveit_tutorial_poses.py

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

from tork_moveit_tutorial import *


if __name__ == '__main__':

    init_node()

    group = MoveGroupCommander("upper_arm")

    # Pose Target 1
    rospy.loginfo( "Start Pose Target 1")
    pose_target_1 = Pose()

    pose_target_1.position.x =  0.0
    pose_target_1.position.y =  0.55
    pose_target_1.position.z =  1.0
    pose_target_1.orientation.x =  0.0
    pose_target_1.orientation.y =  0.0
    pose_target_1.orientation.z =  0.0
    pose_target_1.orientation.w =  0.0

    rospy.loginfo( "Set Target to Pose:n{}".format( pose_target_1 ) )
```

(continues on next page)

(continued from previous page)

```
    group.set_pose_target( pose_target_1 )
    group.go()

    # Pose Target 2
    rospy.loginfo( "Start Pose Target 2")
    pose_target_2 = Pose()

    pose_target_2.position.x = -0.55
    pose_target_2.position.y = -0.0
    pose_target_2.position.z =  1.05
    pose_target_2.orientation.x =  0.0
    pose_target_2.orientation.y =  0.0
    pose_target_2.orientation.z =  0.707
    pose_target_2.orientation.w =  0.707

    rospy.loginfo( "Set Target to Pose:n{}".format( pose_target_2 ) )
    group.set_pose_target( pose_target_2 )
    group.go()
```

他のロボットの動作計画・動作の実行ファイルとの相違点は次のとおりです.

- `group = MoveGroupCommander()` に渡すグループ名を `"upper_arm"` に変更
- ターゲットポーズの位置・姿勢を KHI duaro の機構に適したものに変更

3.2.4 myCobot の場合

ターミナル-1

```
$ source ~/catkin_ws/devel/setup.bash
$ roslaunch tork_moveit_tutorial demo.launch
```

動作プログラムファイルを実行します.

ターミナル-2

```
$ source /opt/ros/melodic/setup.bash
$ rosrun tork_moveit_tutorial mycobot_moveit_tutorial_poses.py
```

実機を動かす場合

先に「実機の使い方 – myCobot の場合」を参照してください.

そのあと，上に書かれたコマンドの代わりに，以下を実行してください.

ターミナル-1 : myCobot 280 とパソコンの接続プログラムの起動

```
$ source ~/catkin_ws/devel/setup.bash
$ roslaunch tork_moveit_tutorial mycobot_interface.launch
```

ターミナル-2 : myCobot 280 用の MoveIt! の起動

```
$ source ~/catkin_ws/devel/setup.bash
$ roslaunch tork_moveit_tutorial demo.launch mode:=real
```

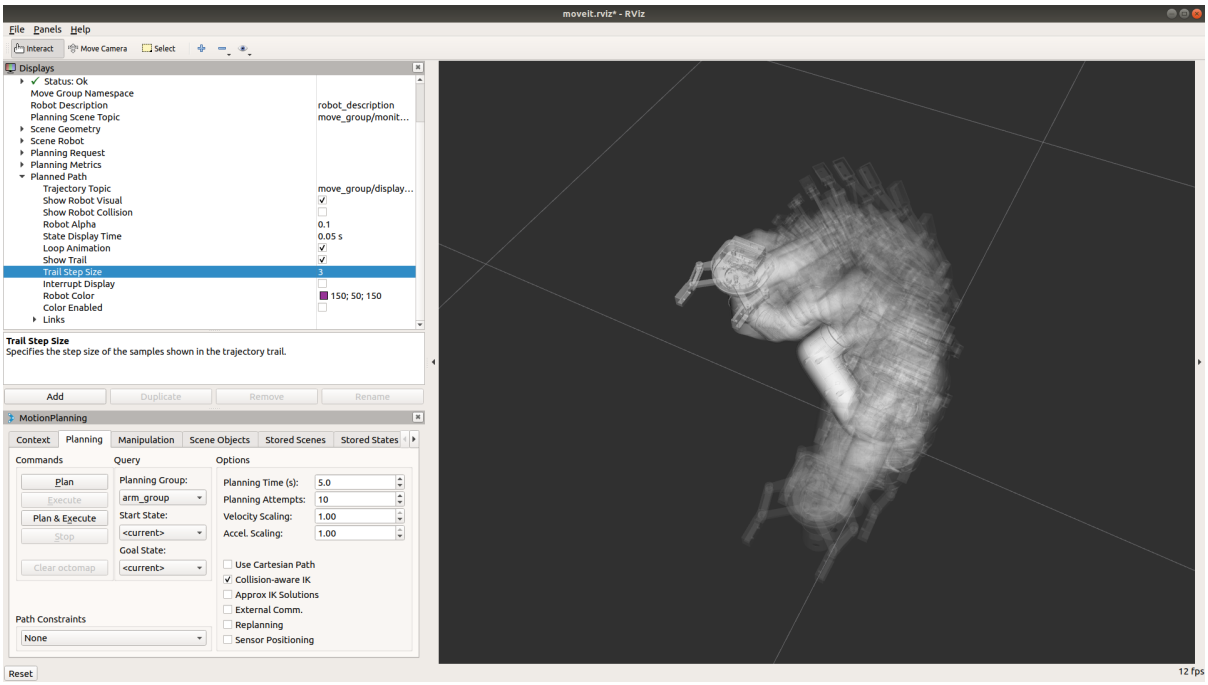
ターミナル-3

```
$ source /opt/ros/melodic/setup.bash
$ rosrun tork_moveit_tutorial mycobot_moveit_tutorial_poses.py
```

`set_pose_target` を用いて手先位置姿勢を指定し，腕を動かすプログラムの例が以下になります.

mycobot_moveit_tutorial_poses.py

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

from tork_moveit_tutorial import *
from tf.transformations import quaternion_from_euler

if __name__ == '__main__':

    init_node()

    group = MoveGroupCommander("arm_group")
    # Pose Target 1
    rospy.loginfo( "Start Pose Target 1")
    pose_target_1 = Pose()

    # quaternion_from_euler(0, 0, -1.57079)
    pose_target_1.position.x = 0.2
    pose_target_1.position.y = 0.0
    pose_target_1.position.z = 0.2
    pose_target_1.orientation.x = 0.0
    pose_target_1.orientation.y = 0.0
    pose_target_1.orientation.z = -0.7071
    pose_target_1.orientation.w =  0.7071

    rospy.loginfo( "Set Target to Pose:n{}".format( pose_target_1 ) )
    group.set_pose_target( pose_target_1 )
    group.go()

    # Pose Target 2
    rospy.loginfo( "Start Pose Target 2")
    pose_target_2 = Pose()

    pose_target_2.position.x = 0.0
    pose_target_2.position.y =-0.2
    pose_target_2.position.z = 0.2
    pose_target_2.orientation.z = -0.7071
    pose_target_2.orientation.w =  0.7071

    rospy.loginfo( "Set Target to Pose:n{}".format( pose_target_2 ) )
    group.set_pose_target( pose_target_2 )
    group.go()

    # Compute Cartesian path
    (plan, fraction) = group.compute_cartesian_path([pose_target_1, pose_target_2],
 0.01, 0.0)
    group.execute( plan )
```


moveit.rviz* - RViz
File Panels Help
Interact
Move Camera
Select
Displays
Status: Ok
Move Group Namespace
Robot Description
robot_description
Planning Scene Topic
move_group/monit...
Scene Geometry
Scene Robot
Planning Request
Planning Metrics
Planned Path
Trajectory Topic
move_group/display...
Show Robot Visual
Show Robot Collision
Robot Alpha
0.1
State Display Time
0.05 s
Loop Animation
Show Trail
Trail Step Size
3
Interrupt Display
Robot Color
150; 50; 150
Color Enabled
Links
Trail Step Size
Specifies the step size of the samples shown in the trajectory trail.
Add
Duplicate
Remove
Rename
MotionPlanning
Context
Planning
Manipulation
Scene Objects
Stored Scenes
Stored States
Commands
Plan
Execute
Plan & Execute
Stop
Clear octomap
Query
Planning Group:
arm_group
Start State:
<current>
Goal State:
<current>
Options
Planning Time (s):
5.0
Planning Attempts:
10
Velocity Scaling:
1.00
Accel. Scaling:
1.00
Use Cartesian Path
Collision-aware IK
Approx IK Solutions
External Comm.
Replanning
Sensor Positioning
Path Constraints
None
Reset
12 fps


SAMSUNG

![myCobot - (3) result of group.compute_cartesian_path([pose_target_1, pose_target_2], 0.01, 0.0)](images/melodic/3.2_3_compute_cartesian_path(pose_target_1, pose_target_2).jpg) ![myCobot - (4) result of group.compute_cartesian_path([pose_target_1, pose_target_2], 0.01, 0.0)](images/melodic/3.2_4_compute_cartesian_path(pose_target_1, pose_target_2).jpg)

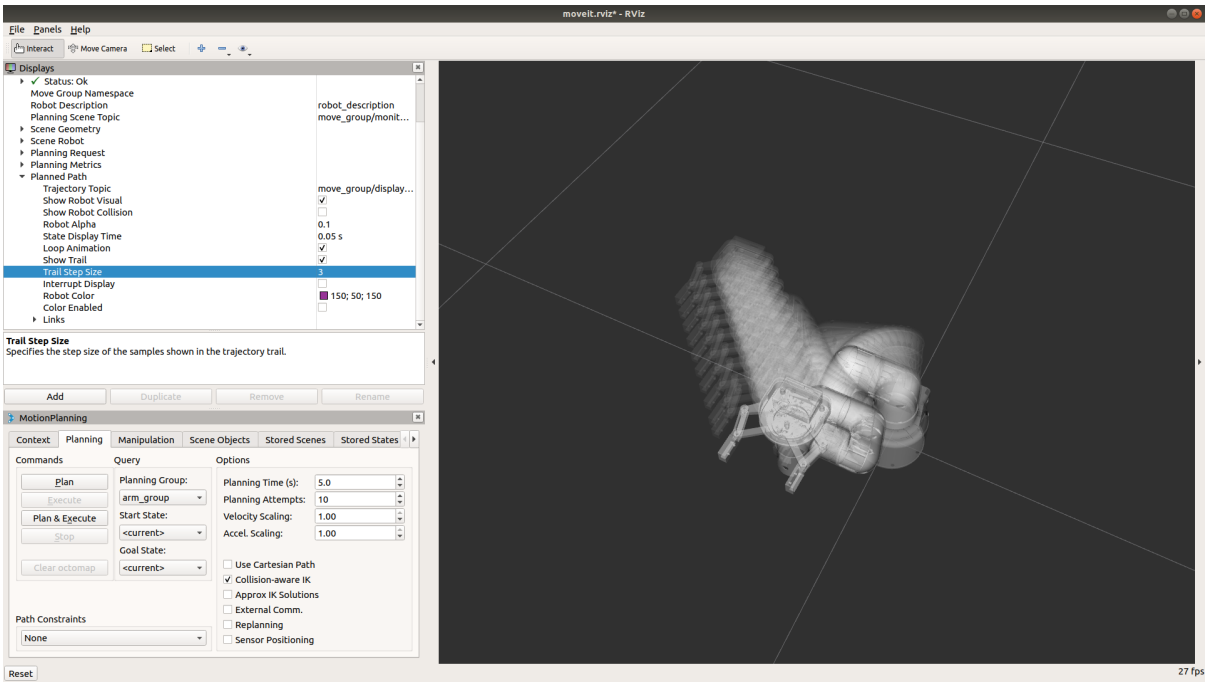
他のロボットの動作計画・動作の実行ファイルとの相違点は次のとおりです.

- `group = MoveGroupCommander()` に渡すグループ名を `"arm_group"` に変更
- ターゲットポーズの位置・姿勢を myCobot の機構に適したものに変更

`Pose()` に指定する姿勢情報のクオータニオン情報は，`from tf.transformations import quaternion_from_euler` で得られる，`quaternion_from_euler(0, 0, -1.57079)` 関数を用いて取得します.

また，このプログラムに続けて以下のように，`pose_target_1`, `pose_target_2` を用いて `group.compute_cartesian_path([pose_target_1, pose_target_2], 0.01, 0.0)` として直線補間軌道を計画し，`group.execute( plan )` で実行します.

```
    # Compute Cartesian path
    (plan, fraction) = group.compute_cartesian_path([pose_target_1, pose_target_2],
0.01, 0.0)
    group.execute( plan )
```



3.2.5 ROS や MoveIt! のメリット

NEXTAGE OPEN や MINAS TRA1，KHI duaro の動作計画・動作プログラムの相違点を見ると下記の 2ヶ所だけが各ロボットに対応しただけだということが分かります.

- group = MoveGroupCommander() に渡すグループ名を各ロボットアームに対応したものにする
- ターゲットポーズの位置・姿勢を各ロボットの機構に適したものにする

このように「**ロボットが異なっても基本的には同じプログラムが動く**」ということが ROS や MoveIt! のインタフェースを使用する最大のメリットの 1 つであるといえます.

CHAPTER

FOUR

発展的なロボットプログラミング

本章は NEXTAGE OPEN ロボットシミュレータを対象にしてチュートリアルを進めてゆきます.

NEXTAGE OPEN の下記ソフトウェアを既に起動した状態で本章の各プログラムを実行してください.

- NEXTAGE OPEN の Gazebo シミュレータもしくは hypsys(RTM) シミュレータ
- NEXTAGE OPEN の MoveIt!

4.1 プログラム制御フローツールとロボットプログラミング

プログラムの制御フローツールを用いることで，より簡潔なプログラムやより複雑なロボット動作プログラムを作成することができます.

4.1.1 条件判断とロボット動作 - if

条件判断を行う `if` 文とロボット動作計画を組み合わせて使ってみます.

下のプログラム例では動作の実行の可否を問うウィンドウを開いて [Yes] ボタンがクリックされた場合に `True` を返す関数 `question_yn()` を利用しています.

本文中の `if` 文の条件部分で関数 `question_yn()` を呼び出して返ってきた `True`/`False` により「動作の実行」と「スキップ」の分岐を行っています.

```
$ rosrun tork_moveit_tutorial nextage_moveit_tutorial_poses_ifqyn.py
```

nextage_moveit_tutorial_poses_ifqyn.py

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

import sys, copy
import rospy

from moveit_commander import MoveGroupCommander
from geometry_msgs.msg import Pose

from tork_moveit_tutorial import init_node, question_yn


if __name__ == '__main__':

    init_node()
    group = MoveGroupCommander("right_arm")

    # Pose Target 1
    rospy.loginfo( "Start Pose Target 1")
    pose_target_1 = Pose()
```

(continues on next page)

(continued from previous page)

```
    pose_target_1.position.x = 0.4
    pose_target_1.position.y = -0.4
    pose_target_1.position.z = 0.15
    pose_target_1.orientation.x = 0.0
    pose_target_1.orientation.y = -0.707
    pose_target_1.orientation.z = 0.0
    pose_target_1.orientation.w = 0.707

    rospy.loginfo( "Set Target to Pose:n{}".format( pose_target_1 ) )
    group.set_pose_target( pose_target_1 )

    if question_yn( "Start moving to target 1 ?" ):
        group.go()

    # Pose Target 2
    rospy.loginfo( "Start Pose Target 2")
    pose_target_2 = Pose()

    pose_target_2.position.x = 0.3
    pose_target_2.position.y = -0.3
    pose_target_2.position.z = 0.5
    pose_target_2.orientation.y = -1.0

    rospy.loginfo( "Set Target to Pose:n{}".format( pose_target_2 ) )
    group.set_pose_target( pose_target_2 )

    if question_yn( "Start moving to target 2 ?" ):
        group.go()
```

4.1.2 繰り返しとロボット動作 - for

指定回数繰り返しを行う `for` 文とロボット動作計画を組み合わせて使ってみます.

下のプログラム例では `for` 文で動作計画と実行を 5 回繰り返し，繰り返すごとに右腕の目標姿勢の Y 座標を `step` 変数で指定した長さ横に移動します.

```
$ rosrun tork_moveit_tutorial nextage_moveit_tutorial_poses_for.py
```

nextage_moveit_tutorial_poses_for.py

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

import sys, copy
import rospy

from moveit_commander import MoveGroupCommander
from geometry_msgs.msg import Pose

from tork_moveit_tutorial import init_node, question_yn


if __name__ == '__main__':

    init_node()
    group = MoveGroupCommander("right_arm")

    # Pose Target 1
    rospy.loginfo( "Start Pose Target 1")
    pose_target_1 = Pose()
```

(continues on next page)

(continued from previous page)

```
    pose_target_1.position.x = 0.3
    pose_target_1.position.y = 0.0
    pose_target_1.position.z = 0.2
    pose_target_1.orientation.x = 0.0
    pose_target_1.orientation.y = -0.707
    pose_target_1.orientation.z = 0.0
    pose_target_1.orientation.w = 0.707

    step = -0.1

    for i in range(5):
        pose_target_1.position.y = i * step
        rospy.loginfo( "Set Target to Pose No.{}:n{}".format( i, pose_target_1 ) )

        group.set_pose_target( pose_target_1 )

        if question_yn( "Start moving to target No.{} ?".format( i ) ):
            group.go()
        else:
            rospy.loginfo( "Skipped Pose No.{}".format( i ) )
```

4.2 プログラムのタイミングを図る

4.2.1 プログラムの一時休止 - sleep

プログラムの一時休止には `rospy.sleep(duration)` を使います.

`rospy.sleep()` には浮動小数点型の数値（単位:秒）か `Duration` 型を渡します. 渡した時間の長さだけプログラムを休止させます.

`rospy.sleep()` の基本的な記法を下に示します.

```
# sleep for 10 seconds
rospy.sleep(10.)

# sleep for duration
d = rospy.Duration(10, 0)
rospy.sleep(d)
```

`Duration( secs, nsecs )` に渡す数値は各々下のようになっています.

- secs : 秒（整数 / デフォルト=0 ）
- nsecs : ナノ秒（整数 / デフォルト=0 ）

4.2.2 プログラムループの一定時間間隔実行 - Rate

プログラムのループを一定時間間隔で実行するには `rospy.Rate(hz)` を使います.

`rospy.Rate(hz)` には周波数（単位:Hz）を浮動小数点型で渡します.

使用方法は `rospy.Rate(hz)` を周波数を設定してプログラムループ内で `Rate.sleep()` を行うと次の周期までプログラムを停止します.

`rospy.Rate()` の基本的な記法を下に示します.

```
r = rospy.Rate(10) # 10hz
while not rospy.is_shutdown():
    print( "Hello 10Hz" )
    r.sleep()
```

- 参考 : ROS Wiki - Sleeping and Rates
  - http://wiki.ros.org/rospy/Overview/Time#Sleeping_and_Rates

4.2.3 関数の定期呼び出し - Timer

プログラムの関数を一定時間間隔で呼び出して実行するには `rospy.Timer()` を用いることができます.

- rospy.Timer(period, callback, oneshot=False)
  - period
    - 呼び出し間隔
    - `Duration` 型
  - callback
    - 呼び出される関数
  - oneshot
    - 1 回だけの実行か
    - `True` / `False`
    - デフォルト : `False`（1 回だけの実行ではない）

下記の例では関数 `my_callback()` を 2 秒ごとに呼び出しています.

```
def my_callback(event):
    print( 'Timer called at {}'.format( event.current_real ) )

rospy.Timer( rospy.Duration(2), my_callback )
```

- 参考 : ROS Wiki - Timer
  - http://wiki.ros.org/rospy/Overview/Time#Timer

4.2.4 ROS ノードを停止させずに待機 - spin

`Timer` などで呼び出し関数を設定してもそのプログラム（ROS ノード）が終了してしまっては関数も呼び出されないので待機状態にする必要があります.

ROS ノードを終了させずにそこで待機状態にするのが `rospy.spin()` です.

`rospy.spin()` の記法を下に示します.

```
rospy.spin()
```

4.2.5 定期的に動くロボットプログラム例

`rospy.Rate()` を使ってロボットが動作を実行する時間の間隔を指定します.

下記プログラムでは次のことを行っています.

- `pose_target_1` にロボット右腕の初期姿勢を定義
- ログメッセージを表示してから `rospy.sleep(5.0)` で 5 秒休止
- `rate = rospy.Rate(0.2)` で 0.2 [Hz] つまり 5 秒に 1 回の時間間隔を設定

- while not rospy.is_shutdown(): ノードが終了指定ない限り次のループを反復
  - group.set_pose_target(pose_target_1) で目標姿勢を設定
  - group.go() で動作計画・実行
  - 目標姿勢の位置の Y 座標を step だけずらす
  - 目標姿勢の位置の Y 座標が -0.4 [m] よりも小さくなったら 0.0 [m] を代入
  - rate.sleep() で次のタイミングが来るまで休止

```
$ rosrun tork_moveit_tutorial nextage_moveit_tutorial_poses_rate.py
```

nextage_moveit_tutorial_poses_rate.py

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

import sys, copy
import rospy

from moveit_commander import MoveGroupCommander
from geometry_msgs.msg import Pose

from tork_moveit_tutorial import init_node


if __name__ == '__main__':

    init_node()
    group = MoveGroupCommander("right_arm")

    # Pose Target 1
    pose_target_1 = Pose()

    pose_target_1.position.x = 0.3
    pose_target_1.position.y = 0.0
    pose_target_1.position.z = 0.2
    pose_target_1.orientation.x = 0.0
    pose_target_1.orientation.y = -0.707
    pose_target_1.orientation.z = 0.0
    pose_target_1.orientation.w = 0.707

    rospy.loginfo( "Start Move Loop / Ctrl-C to Stop nWaiting 5 seconds" )
    rospy.sleep(5.0)

    step = -0.1

    rate = rospy.Rate(0.2)
    while not rospy.is_shutdown():

        rospy.loginfo( "Set Target to Pose:n{}".format( pose_target_1 ) )
        group.set_pose_target( pose_target_1 )
        group.plan()
        group.go()

        pose_target_1.position.y += step
        if pose_target_1.position.y < -0.4:
            pose_target_1.position.y = 0.0

        rospy.loginfo( "Waiting Next... / Ctrl-C to Stop" )
        rate.sleep()
```

4.3 より複雑なロボット動作計画

4.3.1 動作アームを指定する

これまで NEXTAGE OPEN や duaro の動作プログラムの例では NEXTAGE OPEN では「右腕」(right_arm) を，duaro では「上側の腕」(upper_arm) を動かしていました．これらのロボットは双腕ですから「右腕」や「上側の腕」だけでなく「左腕」や「下側の腕」,「両腕」を動かすこともできます．

もう一方の腕を動かす

NEXTAGE OPEN の「左腕」を動かすには次の変更を行います．

- 動作グループを `left_arm` に変更

```
group = MoveGroupCommander("right_arm")     # 変更前（右腕）
↓
group = MoveGroupCommander("left_arm")      # 変更後（左腕）
```

KHI duaro の「下側の腕」を動かすには次の変更を行います．

- 動作グループを `lower_arm` に変更

```
group = MoveGroupCommander("upper_arm")     # 変更前（上側の腕）
↓
group = MoveGroupCommander("lower_arm")     # 変更後（下側の腕）
```

下記プログラムは NEXTAGE OPEN の左腕を動かすプログラムの例です．右腕のターゲット姿勢のままでは左腕の動作にはきつくなるのでターゲット位置の Y 座標の正負（左右）を反転しています．

```
$ rosrun tork_moveit_tutorial nextage_moveit_tutorial_poses_left_arm.py
```

nextage_moveit_tutorial_poses_left_arm.py

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

import sys, copy
import rospy

from moveit_commander import MoveGroupCommander
from geometry_msgs.msg import Pose

from tork_moveit_tutorial import init_node, question_yn


if __name__ == '__main__':

    init_node()

    group = MoveGroupCommander("left_arm")

    # Pose Target 1
    rospy.loginfo( "Start Pose Target 1")
    pose_target_1 = Pose()

    pose_target_1.position.x = 0.4
    pose_target_1.position.y = 0.4
    pose_target_1.position.z = 0.15
    pose_target_1.orientation.x = 0.0
    pose_target_1.orientation.y = -0.707
    pose_target_1.orientation.z = 0.0
    pose_target_1.orientation.w = 0.707
```

(continues on next page)

(continued from previous page)

```
    rospy.loginfo( "Set Target to Pose:n{}".format( pose_target_1 ) )
    group.set_pose_target( pose_target_1 )
    group.go()

    # Pose Target 2
    rospy.loginfo( "Start Pose Target 2")
    pose_target_2 = Pose()

    pose_target_2.position.x = 0.3
    pose_target_2.position.y = 0.3
    pose_target_2.position.z = 0.5
    pose_target_2.orientation.y = -1.0

    rospy.loginfo( "Set Target to Pose:n{}".format( pose_target_2 ) )
    group.set_pose_target( pose_target_2 )
    group.go()
```

両腕を動かす

前項目で動作グループに `left_arm` を指定して左腕を動かしました．他に指定できる「動作グループ」に何があるのかは動作プログラムの例を実行したときに表示されるようにしてあります．

下の出力例は NEXTAGE OPEN の場合のもので `right_arm` や `left_arm` の他に `botharms` や `head`，`torso` などがあるのが分かります．

```
$ rosrun tork_moveit_tutorial nextage_moveit_tutorial_poses_left_arm.py
[ INFO] [1511612715.981903861]: Loading robot model 'NextageOpen'...
[INFO] [WallTime: 1511612716.988247] [0.000000] Move Groups defined in the robot :
botharms
head
left_arm
left_arm_torso
left_hand
right_arm
right_arm_torso
right_hand
torso
upperbody
[ INFO] [1511612718.120724861, 43.933000000]: TrajectoryExecution will use new␣
↪action capability.
[ INFO] [1511612718.120966861, 43.933000000]: Ready to take MoveGroup commands for␣
↪group left_arm.
```

双腕のロボットで「両腕」を動かすには「両腕のグループ」を使います．上記の出力結果から NEXTAGE OPEN では `botharms` の動作グループがあります．

また，`group` が両腕になるので `set_pose_target()` にターゲットポーズに加えてエンドエフェクタのリンク名を渡して明示的に「右腕」と「左腕」を区別します．

```
group.set_pose_target( pose_target_rarm_1, 'RARM_JOINT5_Link' )
group.set_pose_target( pose_target_larm_1, 'LARM_JOINT5_Link' )
```

あとは片腕の動作計画と実行の手順と同じです．

```
group.go()
```

両腕を同時に動作させるプログラム例を下に示します．

```
$ rosrun tork_moveit_tutorial nextage_moveit_tutorial_poses_botharms.py
```

nextage_moveit_tutorial_poses_botharms.py

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

import sys, copy
import rospy

from moveit_commander import MoveGroupCommander
from geometry_msgs.msg import Pose

from tork_moveit_tutorial import init_node, question_yn


if __name__ == '__main__':

    init_node()

    group = MoveGroupCommander("botharms")

    # Pose Target 1
    pose_target_rarm_1 = Pose()
    pose_target_rarm_1.position.x = 0.4
    pose_target_rarm_1.position.y = -0.4
    pose_target_rarm_1.position.z = 0.15
    pose_target_rarm_1.orientation.x = 0.0
    pose_target_rarm_1.orientation.y = -0.707
    pose_target_rarm_1.orientation.z = 0.0
    pose_target_rarm_1.orientation.w = 0.707

    rospy.loginfo( "Right Arm Pose Target 1:n{}".format( pose_target_rarm_1 ) )

    pose_target_larm_1 = Pose()
    pose_target_larm_1.position.x = pose_target_rarm_1.position.x
    pose_target_larm_1.position.y = pose_target_rarm_1.position.y * -1.0
    pose_target_larm_1.position.z = pose_target_rarm_1.position.z
    pose_target_larm_1.orientation.x = pose_target_rarm_1.orientation.x
    pose_target_larm_1.orientation.y = pose_target_rarm_1.orientation.y
    pose_target_larm_1.orientation.z = pose_target_rarm_1.orientation.z
    pose_target_larm_1.orientation.w = pose_target_rarm_1.orientation.w

    rospy.loginfo( "Left Arm Pose Target 1:n{}".format( pose_target_larm_1 ) )

    group.set_pose_target( pose_target_rarm_1, 'RARM_JOINT5_Link' )
    group.set_pose_target( pose_target_larm_1, 'LARM_JOINT5_Link' )
    group.plan()
    group.go()

    # Pose Target 2
    pose_target_rarm_2 = Pose()
    pose_target_rarm_2.position.x = 0.3
    pose_target_rarm_2.position.y = -0.3
    pose_target_rarm_2.position.z = 0.5
    pose_target_rarm_2.orientation.y = -1.0

    rospy.loginfo( "Right Arm Pose Target 2:n{}".format( pose_target_rarm_2 ) )

    pose_target_larm_2 = Pose()
    pose_target_larm_2.position.x = pose_target_rarm_2.position.x
    pose_target_larm_2.position.y = pose_target_rarm_2.position.y * -1.0
    pose_target_larm_2.position.z = pose_target_rarm_2.position.z
```

(continues on next page)

(continued from previous page)

```
    pose_target_larm_2.orientation.x = pose_target_rarm_2.orientation.x
    pose_target_larm_2.orientation.y = pose_target_rarm_2.orientation.y
    pose_target_larm_2.orientation.z = pose_target_rarm_2.orientation.z
    pose_target_larm_2.orientation.w = pose_target_rarm_2.orientation.w

    rospy.loginfo( "Left Arm Pose Target 2:n{}".format( pose_target_larm_2 ) )

    # Move to Pose Target 1
    rospy.loginfo( "Move to Pose Target 1" )
    group.set_pose_target( pose_target_rarm_2, 'RARM_JOINT5_Link' )
    group.set_pose_target( pose_target_larm_2, 'LARM_JOINT5_Link' )
    group.go()

    # Move to Pose Target 2
    rospy.loginfo( "Move to Pose Target 2" )
    group.set_pose_target( pose_target_rarm_2, 'RARM_JOINT5_Link' )
    group.set_pose_target( pose_target_larm_2, 'LARM_JOINT5_Link' )
    group.go()

    # Pose Target 1 & 2 Mixture
    rospy.loginfo( "Move to Pose Target Right:1 Left:2" )
    group.set_pose_target( pose_target_rarm_1, 'RARM_JOINT5_Link' )
    group.set_pose_target( pose_target_larm_2, 'LARM_JOINT5_Link' )
    group.go()

    rospy.loginfo( "Move to Pose Target Right:2 Left:1" )
    group.set_pose_target( pose_target_rarm_2, 'RARM_JOINT5_Link' )
    group.set_pose_target( pose_target_larm_1, 'LARM_JOINT5_Link' )
    group.go()

    # Back to Pose Target 1
    rospy.loginfo( "Go Back to Pose Target 1" )
    group.set_pose_target( pose_target_rarm_1, 'RARM_JOINT5_Link' )
    group.set_pose_target( pose_target_larm_1, 'LARM_JOINT5_Link' )
    group.go()
```

4.3.2 姿勢の参照座標を指定する

参照リンクを指定して参照リンク座標基準で目標姿勢を設定することで，リンクフレーム間の相対的な位置・姿勢の関係において動作計画を行うことができます．

参照リンクを指定するには 2 つの方法があります．

- `PoseStamped` を用いる
  - 参照フレーム情報を含む `PoseStamped` 型のデータを `set_pose_target()` の姿勢に渡して動作を計画する．
- `set_pose_reference_frame()` を用いる
  - `set_pose_reference_frame()` に参照フレーム名を渡して実行することで動作計画の参照フレーム設定を変更する．

下方にある動作プログラム例 **nextage_moveit_tutorial_poses_relative.py** では左腕のエンドエフェクタリンクである `LARM_JOINT5_Link` を参照リンクとしています．`LARM_JOINT5_Link` のリンク座標系上で

- 位置 : `[ 0.4, 0.0, 0.0 ]`
- 姿勢 : クォータニオン表現で `[ 0.0, 0.0, 0.0, 1.0 ]`（＝回転無し）

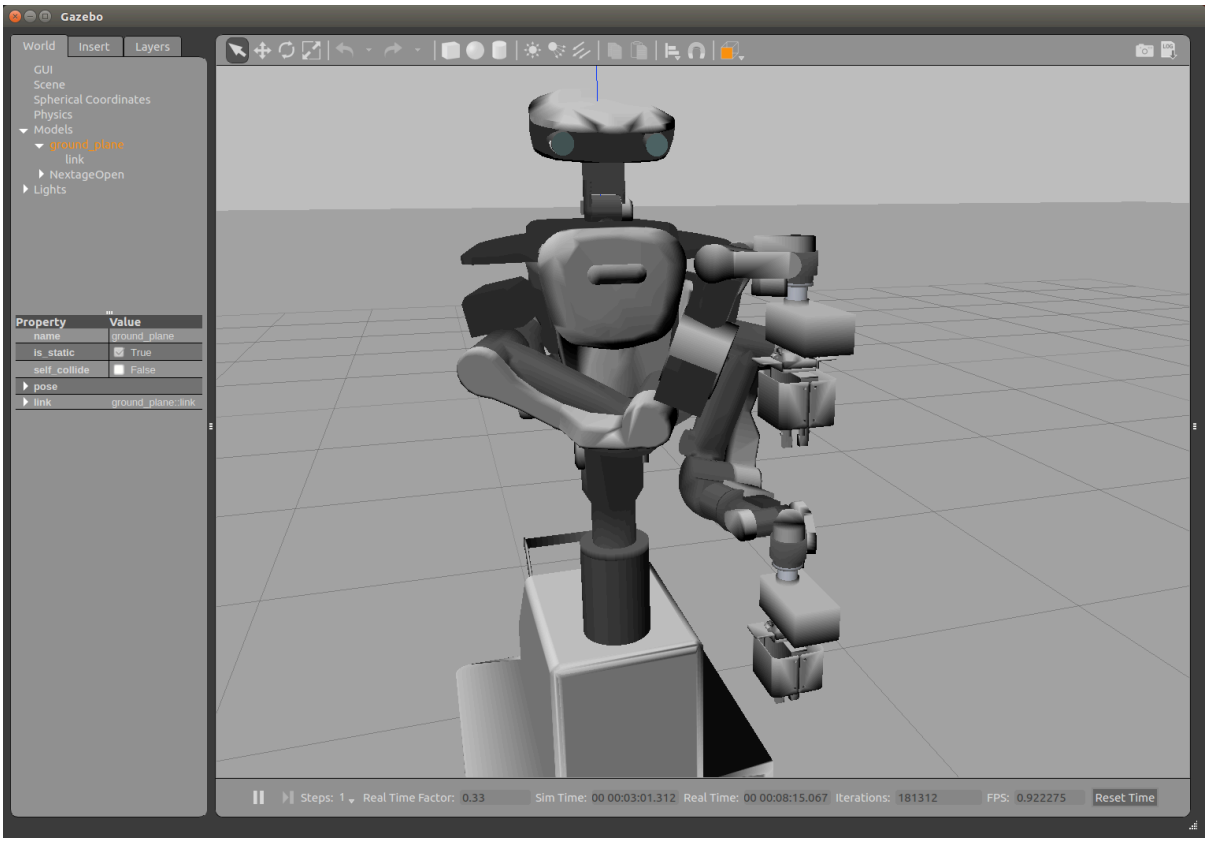
になるように目標姿勢を設定し，2 つの方法について動作計画と実行を行っています．

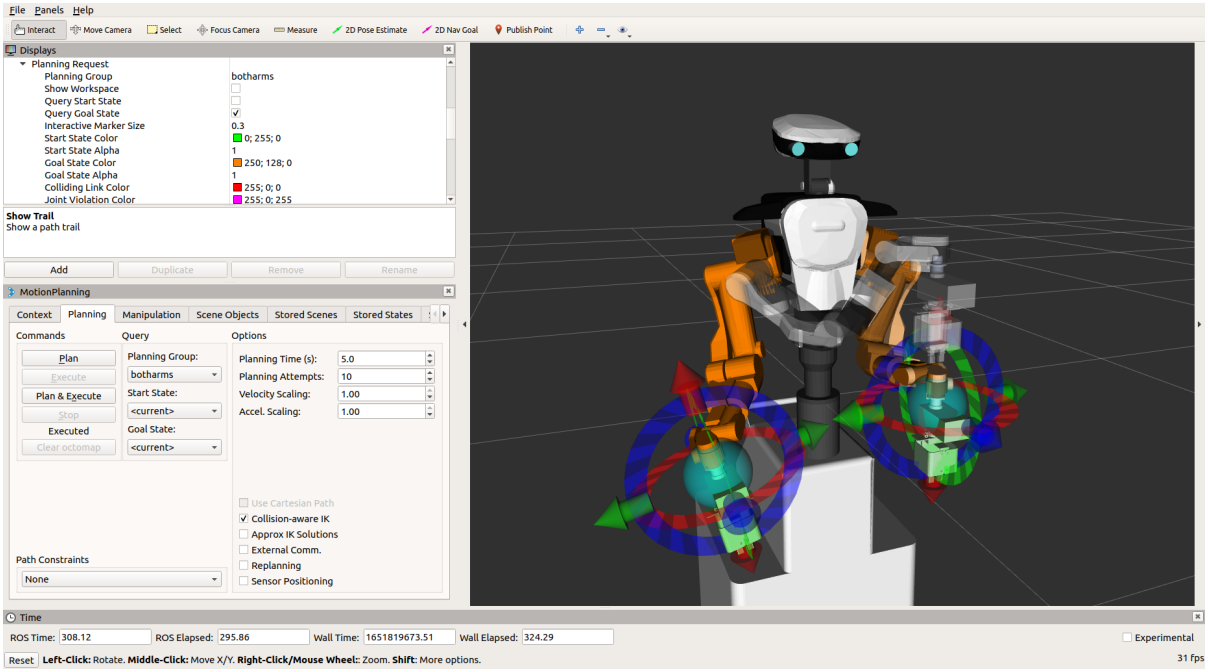
まず PoseStamped 型を利用して参照リンク /LARM_JOINT5_Link の情報を一緒に変数 target_posestamped で渡すことでリンクに相対的な位置・姿勢の目標を指定し動作計画と実行を行います.

次に set_pose_reference_frame() で参照リンクを /LARM_JOINT5_Link に設定して Pose 型の target_pose を渡すだけでリンクに相対的な位置・姿勢の目標を指定し動作計画と実行を行います.

また逐次 group.get_pose_reference_frame() で現状の参照リンクフレームを取得してログ表示しています.

プログラムの最後に右腕の動作開始前の姿勢 initial_pose に戻ります. 初期姿勢 initial_pose は右腕の動作をはじめる前に get_current_pose() を利用して initial_pose = group.get_current_pose() として姿勢の取得をしています.






```
rosrun tork_moveit_tutorial nextage_moveit_tutorial_poses_relative.py
```

nextage_moveit_tutorial_poses_relative.py

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

import sys, copy
import rospy

from moveit_commander import MoveGroupCommander
from geometry_msgs.msg import Pose, PoseStamped

from tork_moveit_tutorial import init_node


if __name__ == '__main__':

    init_node()

    # Preparing Left Arm
    rospy.loginfo( "Preparing Left Arm..." )
    larmg = MoveGroupCommander("left_arm")
    larm_init_pose = Pose()
    larm_init_pose.position.x = 0.325
    larm_init_pose.position.y = 0.182
    larm_init_pose.position.z = 0.067
    larm_init_pose.orientation.x = 0.0
    larm_init_pose.orientation.y = -0.707
    larm_init_pose.orientation.z = 0.0
    larm_init_pose.orientation.w = 0.707
    larmg.set_pose_target(larm_init_pose)
    larmg.go()


    # Right Arm
    group = MoveGroupCommander("right_arm")

    initial_reference_frame = group.get_pose_reference_frame()
    rospy.loginfo( "Initial Reference Frame: {}".format( initial_reference_frame )␣
↪)
```

(continues on next page)

(continued from previous page)

```
    initial_pose = group.get_current_pose()
    rospy.loginfo( "Initial Pose:n{}".format( initial_pose ) )

    # Relative Target Pose
    target_pose = Pose()
    target_pose.position.x = 0.4
    target_pose.orientation.w = 1.0

    # Relative Target with PoseStamped
    rospy.loginfo( "Using PoseStamped" )

    target_posestamped = PoseStamped()
    target_posestamped.pose.position.x = 0.4
    target_posestamped.pose.orientation.w = 1.0
    target_posestamped.header.frame_id = '/LARM_JOINT5_Link'
    target_posestamped.header.stamp = rospy.Time.now()

    rospy.loginfo( "Target Pose:n{}".format( target_posestamped ) )
    group.set_pose_target( target_posestamped )
    group.go()

    rospy.loginfo( "Current Reference Frame: {}".format( group.get_pose_reference_
↪frame() ) )

    # Go Back to Initial Pose
    group.set_pose_target( initial_pose.pose )
    group.go()

    # Relative Target with set_pose_reference_frame
    rospy.loginfo( "Using set_pose_refercence_frame() and Pose" )

    group.set_pose_reference_frame( '/LARM_JOINT5_Link' )
    rospy.loginfo( "Current Reference Frame: {}".format( group.get_pose_reference_
↪frame() ) )
    rospy.loginfo( "Target Pose:n{}".format( target_pose ) )

    group.set_pose_target( target_pose )
    group.go()

    # Reset Pose Reference Frame
    group.set_pose_reference_frame( initial_reference_frame )
    rospy.loginfo( "Current Reference Frame: {}".format( group.get_pose_reference_
↪frame() ) )

    # Go Back to Initial Pose
    rospy.loginfo( "Go Back to Initial Pose..." )
    group.set_pose_target( initial_pose )
    group.go()
```

4.3.3 座標系フレーム間の相対姿勢を取得する - tf

ロボットシステムは多くの変化する座標系フレームで構成されています.

- ワールドフレーム
- ベースフレーム（ロボットフレーム）
- グリッパフレーム
- ヘッドフレーム

- …

ROS の **tf** は常にこれらのフレームを追跡・記憶して各フレーム間の姿勢（同時・非同時）の情報を発信・提供します.

- ROS Wiki
  - tf : http://wiki.ros.org/ja/tf
  - tf チュートリアル : http://wiki.ros.org/ja/tf/Tutorials

項目「姿勢の参照座標を指定する」で行った左手の上に右手を持ってくる動作と同じようなことを tf を用いてロボットに行わせてみます.

ここでは tf を利用した `get_current_target_pose()` という関数を用意していて，ターゲットとするフレームのベースとする任意のフレームに対する相対座標（相対姿勢）を `PoseStamped` 型で取得出来るようになっています.

- `get_current_target_pose( target_frame_id, base_frame_id, timeout = 1.0 )`
  - 引数（関数に渡す値）
    - `target_frame_id` : 文字列型 (string) - ターゲットとするフレーム ID
    - `base_frame_id` : 文字列型 (string) - ベースとするフレーム ID
    - `timeout` : 浮動小数点型 (float) - tf 変換の制限時間 [sec] / デフォルト `1.0`
  - 戻り値（関数から返ってくる値）
    - `PoseStamped` 型 - ベースフレーム座標上のターゲットフレームの姿勢

動作プログラム例 **nextage_moveit_tutorial_poses_tf.py** を実行すると左腕のエンドエフェクタフレーム / `LARM_JOINT5_Link` の姿勢を動作計画基準座標フレーム（NEXTAGE OPEN の場合は `/WAIST`）を参照フレームとして `get_current_target_pose()` で取得します．取得したターゲット姿勢の Z 座標を 0.4 [m] 高くして動作計画をして実行しています.

```
$ rosrun tork_moveit_tutorial nextage_moveit_tutorial_poses_tf.py
```

nextage_moveit_tutorial_poses_tf.py

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

import sys, copy, math
import rospy

from moveit_commander import MoveGroupCommander
from geometry_msgs.msg import Pose

from tork_moveit_tutorial import init_node, get_current_target_pose


def main():

    init_node()

    # Preparing Left Arm
    rospy.loginfo( "Preparing Left Arm..." )
    larmg = MoveGroupCommander("left_arm")
    larmg.set_pose_target( [ 0.325, 0.182, 0.067, 0.0, -math.pi/2, 0.0 ] )
    larmg.go()


    # Right Arm
    group = MoveGroupCommander("right_arm")
```

(continues on next page)

(continued from previous page)

```
    # Frame ID Definitoins
    planning_frame_id = group.get_planning_frame()
    tgt_frame_id = '/LARM_JOINT5_Link'

    # Get a target pose
    pose_target = get_current_target_pose( tgt_frame_id, planning_frame_id )

    # Move to a point above target
    if pose_target:
        pose_target.pose.position.z += 0.4
        rospy.loginfo( "Set Target To: n{}".format( pose_target ) )
        group.set_pose_target( pose_target )
        ret = group.go()
        rospy.loginfo( "Executed ... {}".format( ret ) )
    else:
        rospy.logwarn( "Pose Error: {}".format( pose_target ) )


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

ちなみに関数 `get_current_target_pose()` は **moveit_tutorial_tools.py** 内で定義していて次のようになっています.

```
def get_current_target_pose( target_frame_id, base_frame_id, timeout = 1.0 ):
    '''
    Get current pose TF between a target frame and a base frame.

    @type  target_frame_id : str
    @param target_frame_id : Target frame ID for aquiring TF
    @type  base_frame_id   : str
    @param base_frame_id   : Base frame ID for aquiring TF
    @type  timeout         : float
    @param timeout         : Time length for TF translation timeout [s]
    '''

    endtime = rospy.get_time()
    rospy.loginfo( "Waiting Clock: {}".format( endtime ) )
    while not endtime:
        endtime = rospy.get_time()

    endtime += timeout

    target_pose = None
    listener = tf.TransformListener()
    rate = rospy.Rate(10.0)

    while not rospy.is_shutdown():
        try:
            now = rospy.Time(0)
            (trans,quat) = listener.lookupTransform( base_frame_id, target_frame_
↪id, now )
            target_pose = PoseStamped()
            target_pose.pose.position.x = trans[0]
            target_pose.pose.position.y = trans[1]
            target_pose.pose.position.z = trans[2]
            target_pose.pose.orientation.x = quat[0]
            target_pose.pose.orientation.y = quat[1]
```

(continues on next page)

(continued from previous page)

```
            target_pose.pose.orientation.z = quat[2]
            target_pose.pose.orientation.w = quat[3]
            target_pose.header.frame_id = base_frame_id
            target_pose.header.stamp = now
            break
        except (tf.LookupException, tf.ConnectivityException, tf.
↪ExtrapolationException) as e:
            rospy.logwarn(e)

        now_float = rospy.get_time()
        if endtime < now_float:
            rospy.logwarn( "Time Out: {} [sec] at Clock: {} [sec]".format( timeout,
 now_float ) )
            break

        rate.sleep()

    return target_pose
```

4.3.4 画像処理プログラム出力 tf の利用

前項目で左手のエンドエフェクタのベースフレームに対する姿勢を取得してその値を右手の目標姿勢の算出に使用しました．これを応用すると様々なことに利用できるのですが，その 1 例として画像処理プログラムが出力している tf を取得して対象物に合わせた動作計画とその実行を行ってみます．

ここでは NEXTAGE OPEN の頭部左眼カメラ画像から，そこに映った AR マーカ 姿勢の tf を出力するプログラムを利用して AR マーカ の真上の位置に右腕を動かしてみます．

AR マーカ の動作を行うための準備をします．

Gazebo シミュレータを起動します．

ターミナル-1

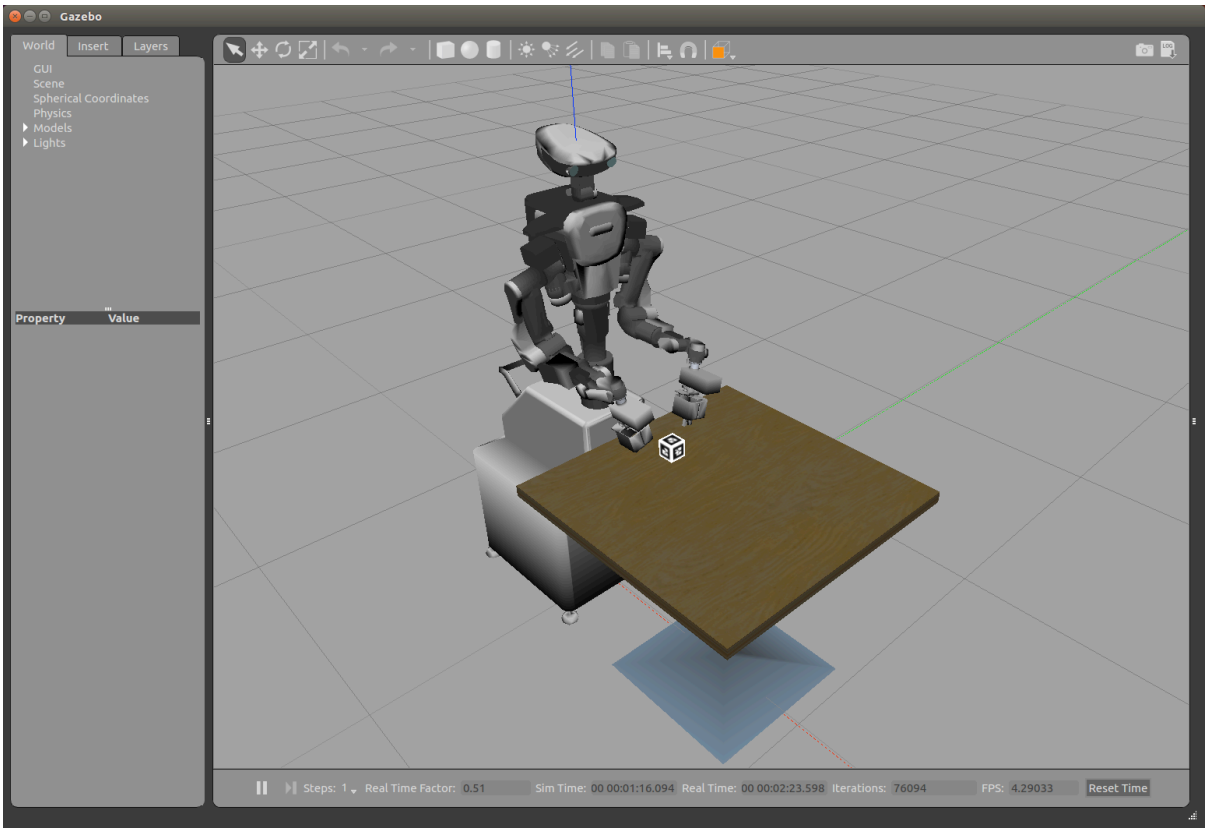
```
$ source /opt/ros/melodic/setup.bash
$ roslaunch nextage_gazebo nextage_world.launch
```

Gazebo 内の NEXTAGE OPEN ロボットが一通りの初期動作を終えたら AR マーカ と テーブル のモデルを Gazebo 内に設置して AR マーカ認識プログラムを起動します．2 つ目のターミナルで次のコマンドを実行してください．

ターミナル-2

```
$ source /opt/ros/melodic/setup.bash
$ roslaunch nextage_ros_bridge ar_headcamera.launch sim:=true
```

`roslaunch nextage_ros_bridge ar_headcamera.launch sim:=true` が正常に起動すると Gazebo は次のような状態になります．



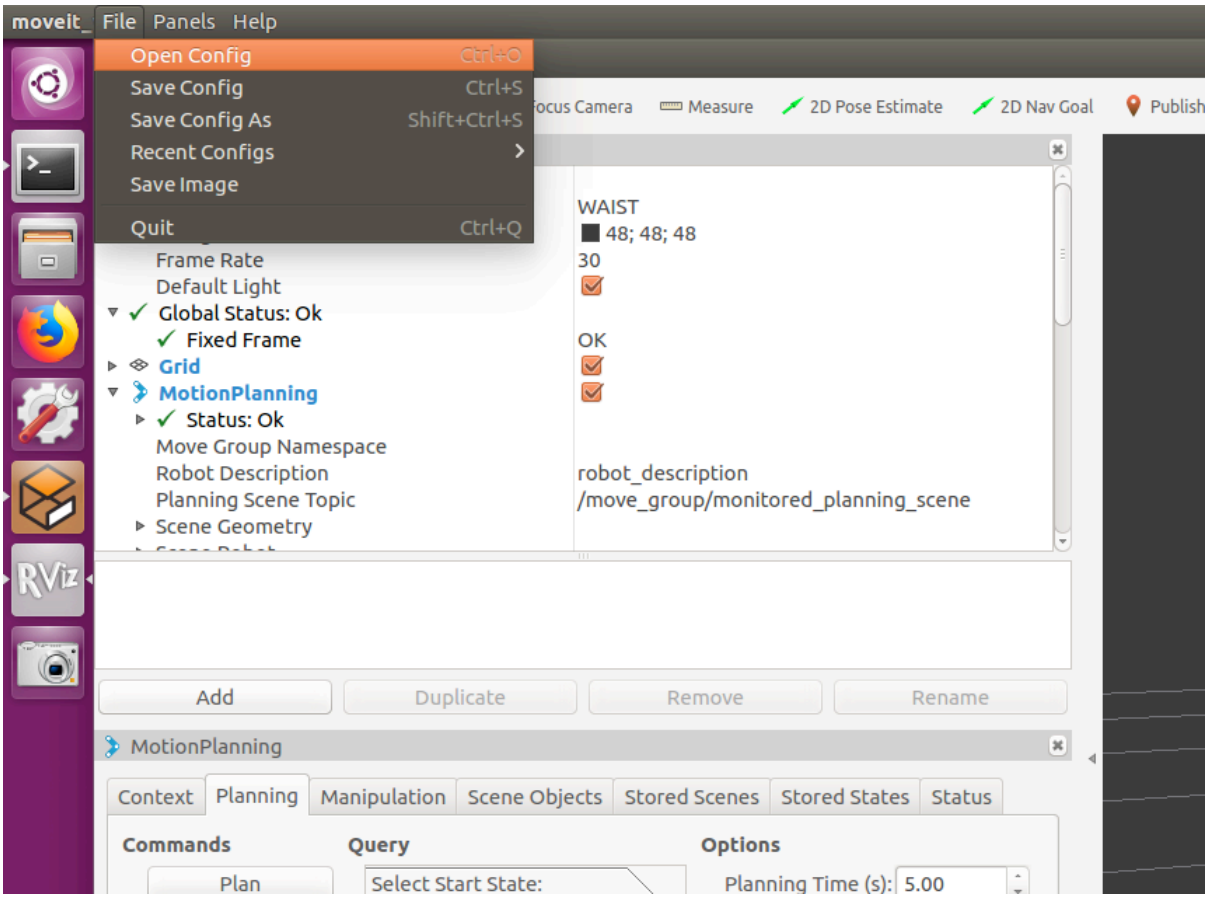
次に MoveIt! を起動します.

ターミナル-3

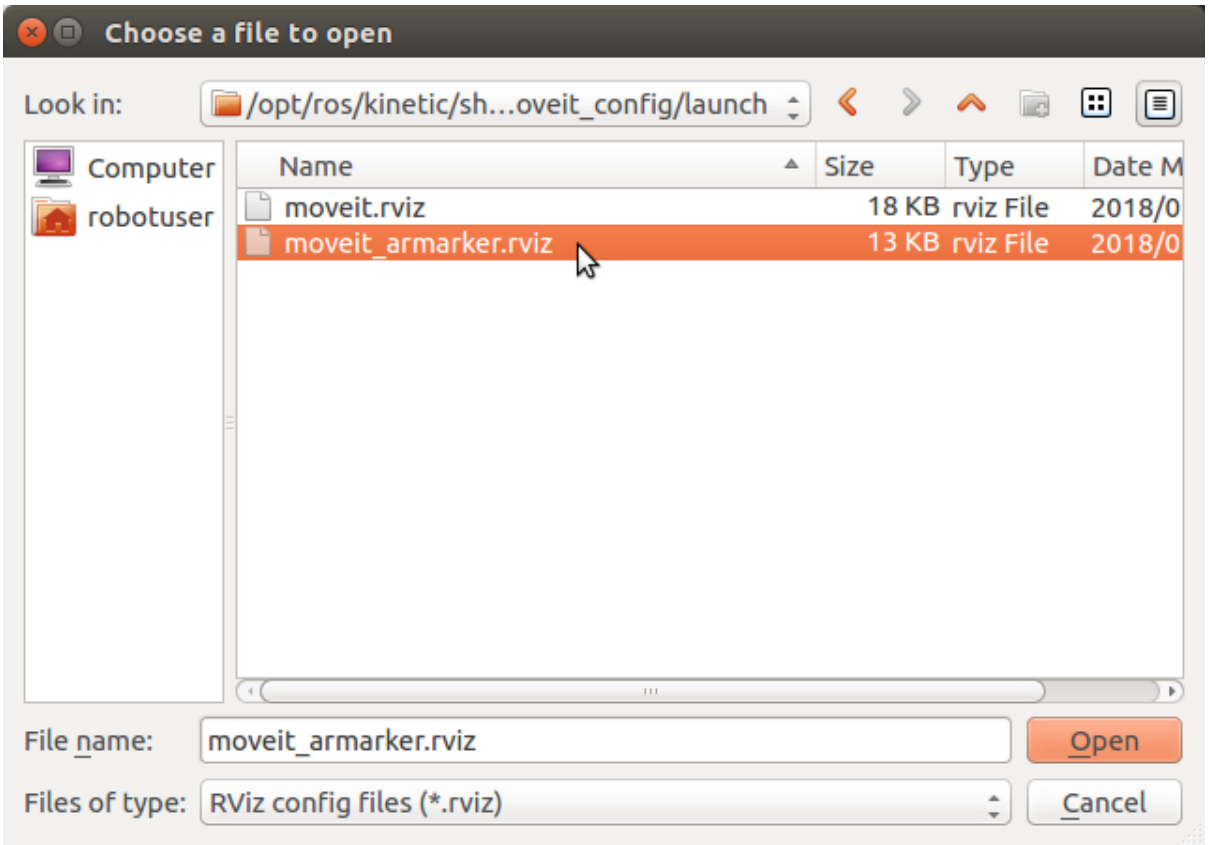
```
$ source /opt/ros/melodic/setup.bash
$ roslaunch nextage_moveit_config moveit_planning_execution.launch
```

MoveIt! ではカメラ画像や AR マーカ の情報を表示したいので RViz の設定ファイルを読み込みます.

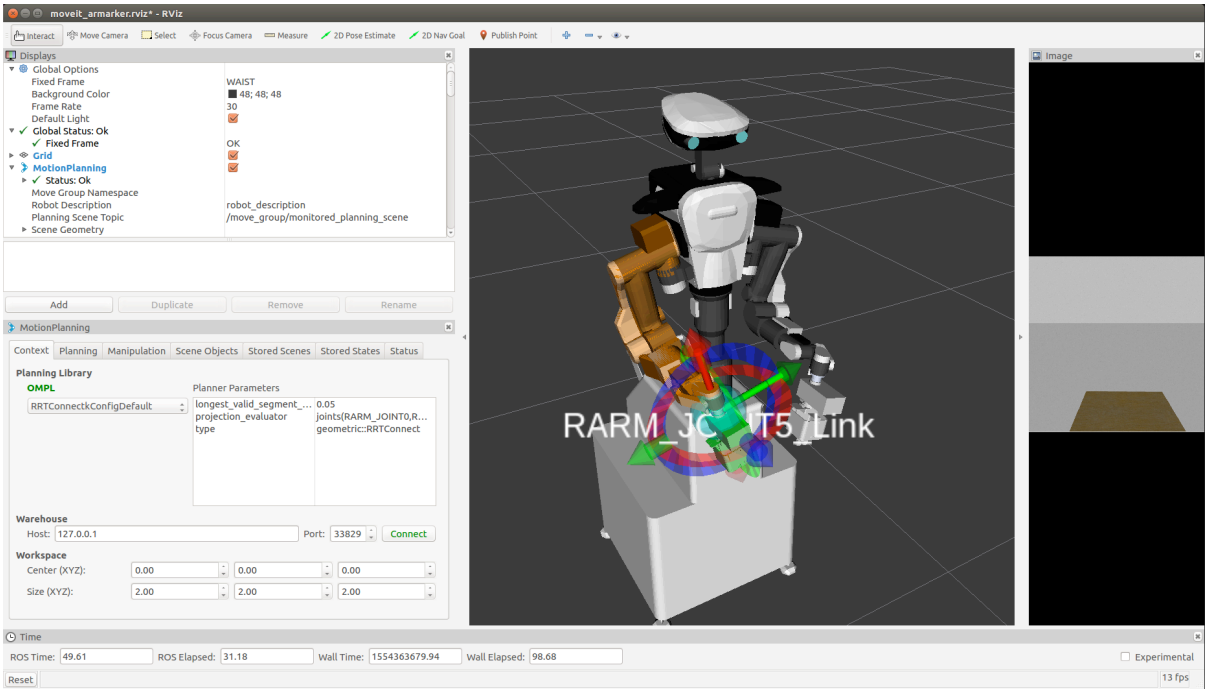
画面左上の File から Open Config をクリックします.



設定ファイル moveit_armarker.rviz を選択して開きます. `/opt/ros/melodic/share/nextage_moveit_config/launch/moveit_armarker.rviz`



設定ファイルが読み込まれると MoveIt! は次のように表示されます.

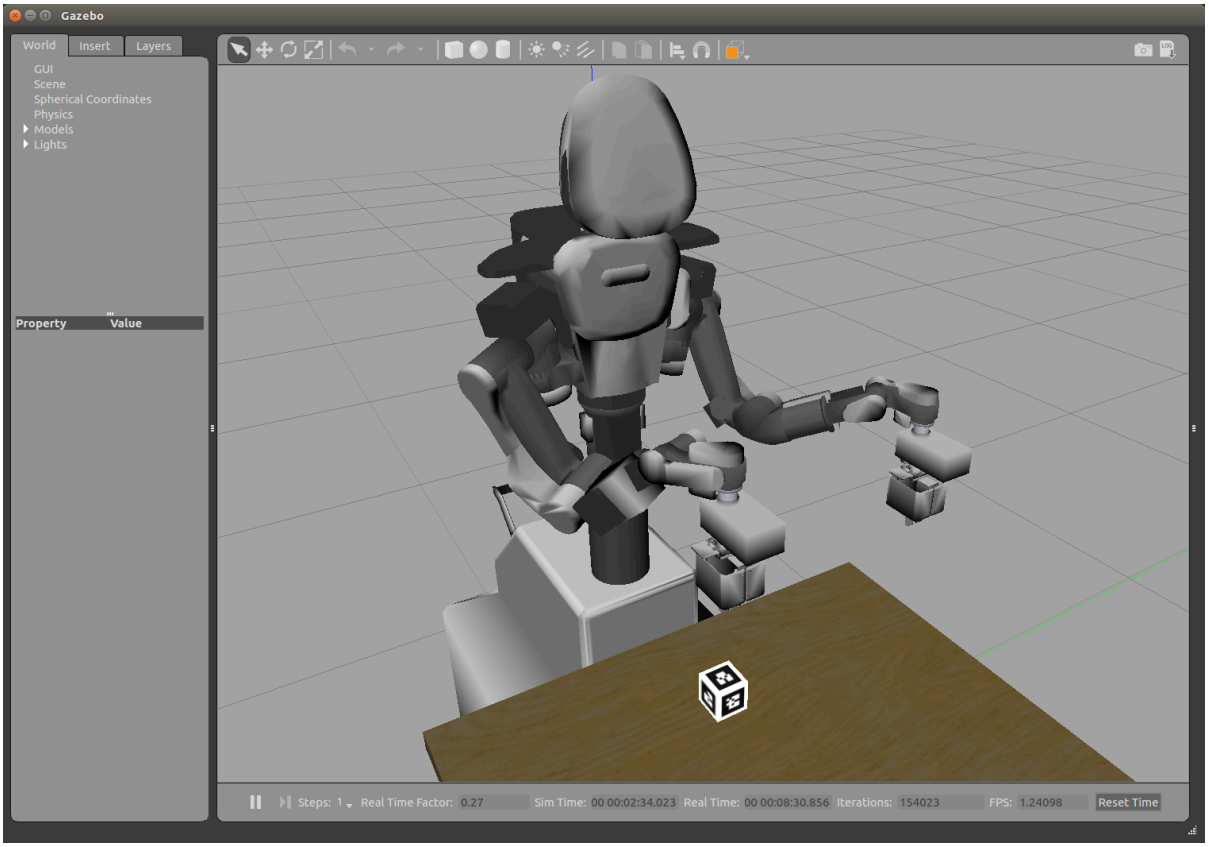


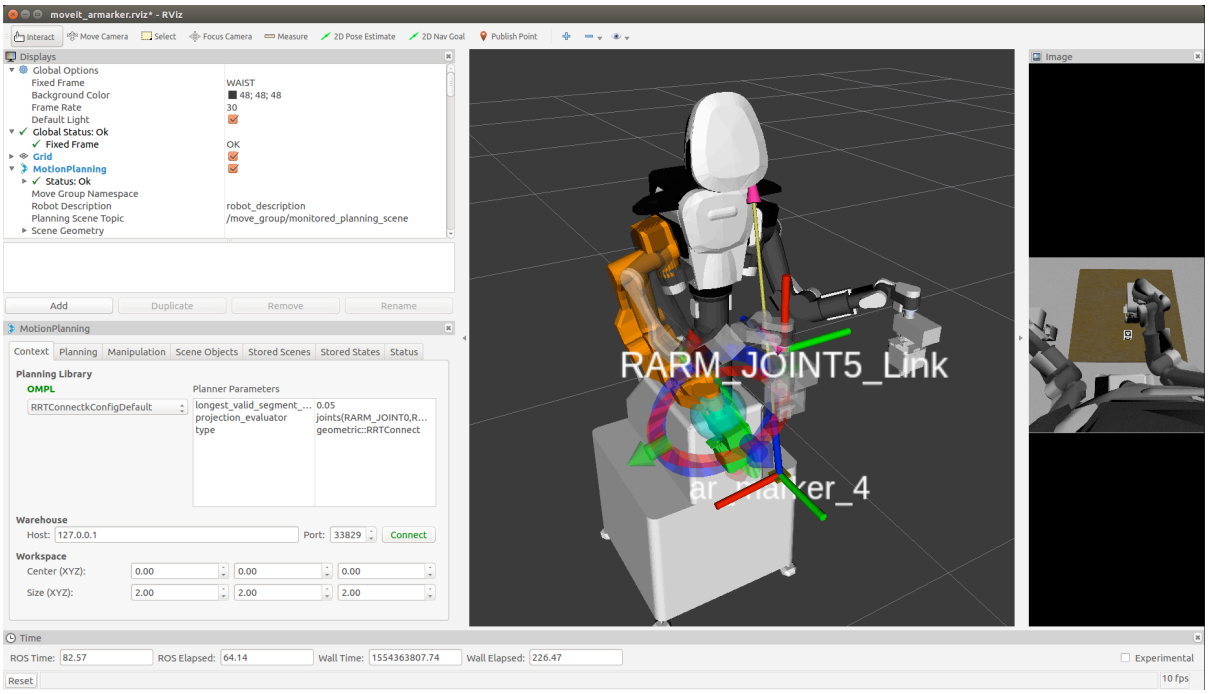
準備が整いましたので AR マーカ の上に右手を動かすプログラムを実行します.

ターミナル-4

```
$ source /opt/ros/melodic/setup.bash
$ rosrun tork_moveit_tutorial nextage_moveit_tutorial_poses_ar.py
```

正常にこのプログラムが動作すると次の画像のように AR マーカ の上に右手が位置していると思います.





この AR マーカ の上に右手を動かすプログラムは下記のようになっています．その主要な部分について続いて説明します．

nextage_moveit_tutorial_poses_ar.py

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

import sys, copy, math
import rospy, tf

from moveit_commander import MoveGroupCommander
from geometry_msgs.msg import Pose
from tf.transformations import quaternion_multiply, quaternion_about_axis

from tork_moveit_tutorial import init_node, get_current_target_pose


def main():

    init_node()

    # Preparing Head
    rospy.loginfo( "Preparing Head..." )
    headg = MoveGroupCommander("head")
    headg.set_joint_value_target( [ 0.0, 60.0/180.0*math.pi ] )
    headg.go()

    # Preparing Both Arms
    rospy.loginfo( "Preparing Left Arm..." )
    barmg = MoveGroupCommander("botharms")
    barmg.set_pose_target( [ 0.325,  0.482, 0.167, 0.0, -math.pi/2, 0.0 ], 'LARM_
↪JOINT5_Link'  )
    barmg.set_pose_target( [ 0.325, -0.482, 0.167, 0.0, -math.pi/2, 0.0 ], 'RARM_
↪JOINT5_Link'  )
    barmg.go()
    rospy.sleep(2.0)

    # Right Arm
    group = MoveGroupCommander("right_arm")
```

(continues on next page)

(continued from previous page)

```
    # Frame ID Definitoins
    planning_frame_id = group.get_planning_frame()
    tgt_frame_id = '/ar_marker_4'

    # Get a target pose
    pose_target = get_current_target_pose( tgt_frame_id, planning_frame_id, 5.0 )

    # Move to a point above target
    if pose_target:

        # Rotate Pose for Right Hand
        quat = []
        quat.append( pose_target.pose.orientation.x )
        quat.append( pose_target.pose.orientation.y )
        quat.append( pose_target.pose.orientation.z )
        quat.append( pose_target.pose.orientation.w )
        quat = quaternion_multiply( quat, quaternion_about_axis( math.pi/2, (1,0,
↪0) ) )
        quat = quaternion_multiply( quat, quaternion_about_axis( math.pi/2, (0,0,
↪1) ) )
        pose_target.pose.orientation.x = quat[0]
        pose_target.pose.orientation.y = quat[1]
        pose_target.pose.orientation.z = quat[2]
        pose_target.pose.orientation.w = quat[3]

        pose_target.pose.position.z += 0.4
        rospy.loginfo( "Set Target To: n{}".format( pose_target ) )
        group.set_pose_target( pose_target )
        ret = group.go()
        rospy.loginfo( "Executed ... {}".format( ret ) )
    else:
        rospy.logwarn( "Pose Error: {}".format( pose_target ) )


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

AR マーカ 4 番 のフレーム /ar_marker_4 の姿勢を動作計画基準座標フレーム（NEXTAGE OPEN の場合は /WAIST）を参照フレームとして get_current_target_pose() で取得して pose_target とします.

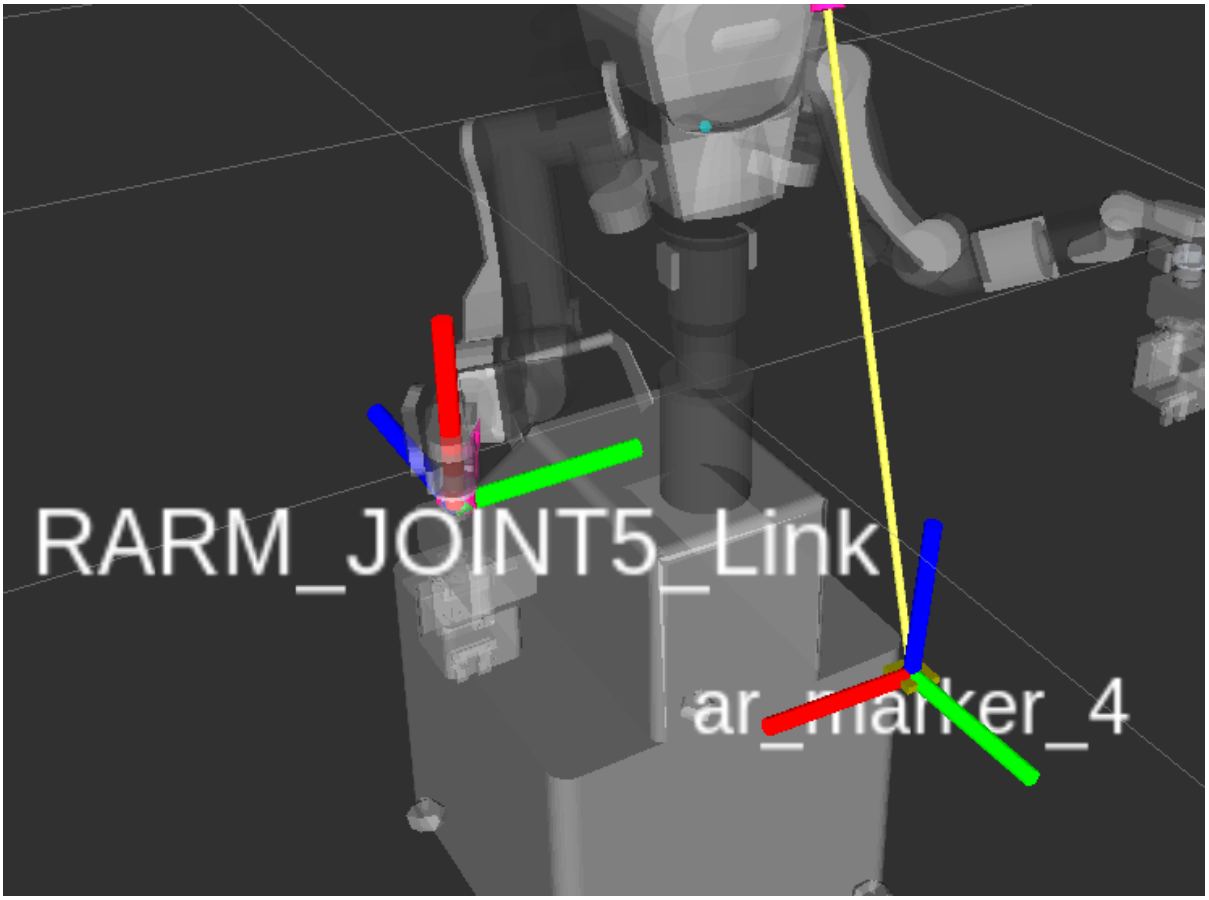
```
# Frame ID Definitoins
planning_frame_id = group.get_planning_frame()
tgt_frame_id = '/ar_marker_4'

# Get a target pose
pose_target = get_current_target_pose( tgt_frame_id, planning_frame_id, 5.0 )
```

この際 get_current_target_pose() 内での tf の取得にある程度時間がかかるので timeout を 5.0 秒に設定しました.

取得したターゲットの姿勢をもとに右腕の動作目標姿勢を作成するのですが右手のフレーム RARM_JOINT5_Link と AR マーカ 4 番 ar_marker_4 の姿勢を MoveIt!(RViz) 上で比較すると次の図のように方向が異なることが分かります.

- メモ: 各フレームの TF を表示しています.
  - X 軸 : 赤 (Red)

- Y 軸 : 緑 (Green)
- Z 軸 : 青 (Blue)



この ar_marker_4 の姿勢をそのまま NEXTAGE OPEN の右腕のエンドエフェクタ RARM_JOINT5_Link の目標姿勢として適用させてしまうと非常に無理のある姿勢となってしまいます.

そのため取得したターゲット姿勢を NEXTAGE OPEN に適した姿勢に回転させます．それを行っているのが下に書き出した部分です.

姿勢を X 軸回りに 90[deg] 回してから Z 軸回りに 90[deg] 回しています.

```
# Rotate Pose for Right Hand
quat = []
quat.append( pose_target.pose.orientation.x )
quat.append( pose_target.pose.orientation.y )
quat.append( pose_target.pose.orientation.z )
quat.append( pose_target.pose.orientation.w )
quat = quaternion_multiply( quat, quaternion_about_axis( math.pi/2, (1,0,0) ) )
quat = quaternion_multiply( quat, quaternion_about_axis( math.pi/2, (0,0,1) ) )
pose_target.pose.orientation.x = quat[0]
pose_target.pose.orientation.y = quat[1]
pose_target.pose.orientation.z = quat[2]
pose_target.pose.orientation.w = quat[3]
```

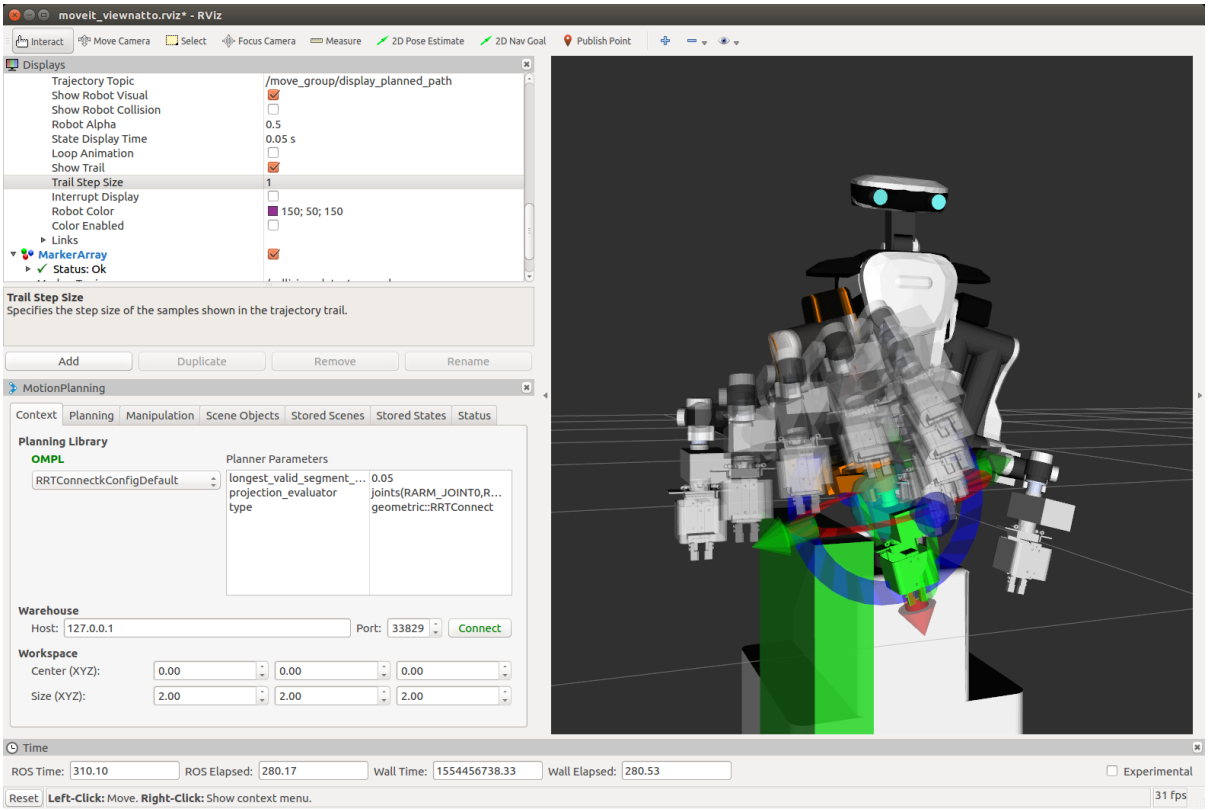
あとは Z 座標を 0.4 [m] 高くして動作計画をし，実行しています.

```
pose_target.pose.position.z += 0.4
rospy.loginfo( "Set Target To: n{}".format( pose_target ) )
group.set_pose_target( pose_target )
ret = group.go()
rospy.loginfo( "Executed ... {}".format( ret ) )
```

4.3.5 障害物の設置と回避動作計画

障害物を動作計画空間に設置することは MoveIt! の RViz/GUI 操作でもできますがプログラムからも行うことができます.

障害物を動作計画空間に設置してしまえば障害物が無い場合の動作計画と全く同じ手順で MoveIt! が自動で障害物を回避する動作計画を作成します.



動作計画空間への障害物を設置して回避動作計画をするプログラムを下方に掲載します. そのうち動作計画空間への障害物を設置する部分は次の箇所です.

```
# Add Object to Planning Scene
rospy.loginfo( "Planning Scene Settings")
scene = PlanningSceneInterface()

rospy.sleep(2)    # Waiting for PlanningSceneInterface

box_pose = PoseStamped()
box_pose.header.frame_id = group.get_planning_frame()
box_pose.pose.position.x = 0.35
box_pose.pose.position.y = -0.3
box_pose.pose.position.z = -0.2
box_pose.pose.orientation.w = 1.0
scene.add_box( 'box_object', box_pose, ( 0.3, 0.1, 0.5 ) )
```

この部分で行っていることは次のとおりです.

- `PlanningSceneInterface` を作成して `scene` とする
- `PlanningSceneInterface` の作成に時間が必要なので `rospy.sleep(2)` で待つ
- 設置する障害物の位置・姿勢を `PoseStamped` 型で定義
- `scene` の `add_box()` で動作計画空間に「箱」を設置

```
$ rosrun tork_moveit_tutorial nextage_moveit_tutorial_poses_object.py
```

nextage_moveit_tutorial_poses_object.py

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

import sys, copy, math
import rospy, tf

from moveit_commander import MoveGroupCommander, PlanningSceneInterface
from geometry_msgs.msg import Pose, PoseStamped

from tork_moveit_tutorial import init_node


if __name__ == '__main__':

    init_node()

    group = MoveGroupCommander("right_arm")

    # Pose Target 1
    rospy.loginfo( "Start Pose Target 1")
    pose_target_1 = Pose()

    pose_target_1.position.x = 0.3
    pose_target_1.position.y = -0.1
    pose_target_1.position.z = 0.15
    pose_target_1.orientation.x = 0.0
    pose_target_1.orientation.y = -0.707
    pose_target_1.orientation.z = 0.0
    pose_target_1.orientation.w = 0.707

    rospy.loginfo( "Set Target to Pose:n{}".format( pose_target_1 ) )

    group.set_pose_target( pose_target_1 )
    group.go()

    # Add Object to Planning Scene
    rospy.loginfo( "Planning Scene Settings")
    scene = PlanningSceneInterface()

    rospy.sleep(2)    # Waiting for PlanningSceneInterface

    box_pose = PoseStamped()
    box_pose.header.frame_id = group.get_planning_frame()
    box_pose.pose.position.x = 0.35
    box_pose.pose.position.y = -0.3
    box_pose.pose.position.z = -0.2
    box_pose.pose.orientation.w = 1.0
    scene.add_box( 'box_object', box_pose, ( 0.3, 0.1, 0.5 ) )

    rospy.loginfo( "Scene Objects : {}".format( scene.get_known_object_names() ) )

    # Pose Target 2
    rospy.loginfo( "Start Pose Target 2")
    pose_target_2 = Pose()
    pose_target_2.position.x = 0.3
    pose_target_2.position.y = -0.5
    pose_target_2.position.z = 0.15
    pose_target_2.orientation.x = 0.0
    pose_target_2.orientation.y = -0.707
    pose_target_2.orientation.z = 0.0
    pose_target_2.orientation.w = 0.707
```

(continues on next page)

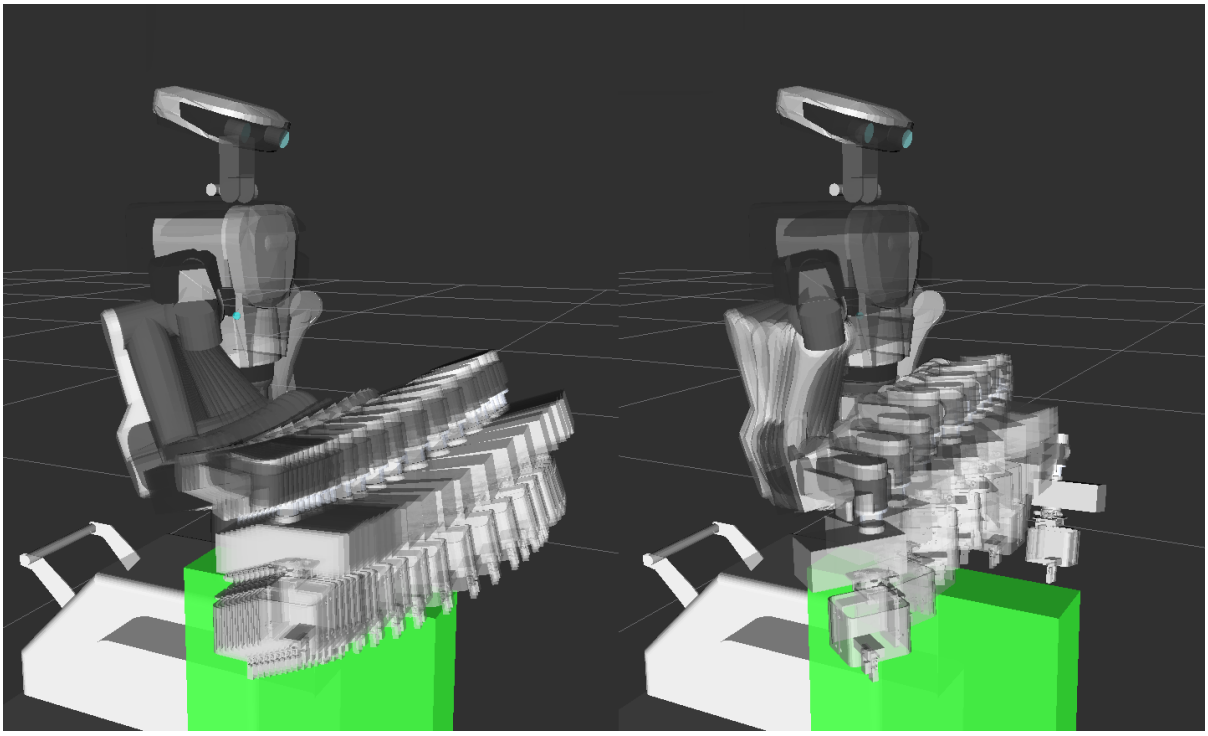
(continued from previous page)

```
    rospy.loginfo( "Set Target to Pose:n{}".format( pose_target_2 ) )

    group.set_pose_target( pose_target_2 )
    group.go()
```

4.3.6 障害物回避動作計画における拘束条件の付加

手先に溢れるようなものを持たせた場合などで障害物回避をするときに姿勢が傾かないようにエンドエフェクタの姿勢に拘束条件を付加して動作計画を行います.

次の図のように障害物回避動作時に拘束条件を付加しない場合に大きく傾いてしまう動作（左）をエンドエフェクタの水平方向を維持したまま動作（右）するようにします.



障害物回避動作計画に姿勢の拘束条件を付加した動作を行うプログラム **nextage_moveit_tutorial_poses_object_constraint.py** を下方に掲載します. そのうち拘束条件を付加する部分は次の箇所です.

```
# Set Path Constraint
constraints = Constraints()
constraints.name = "down"

orientation_constraint = OrientationConstraint()
orientation_constraint.header.frame_id = group.get_planning_frame()
orientation_constraint.link_name = group.get_end_effector_link()
orientation_constraint.orientation = pose_target_1.orientation
orientation_constraint.absolute_x_axis_tolerance = 3.1415
orientation_constraint.absolute_y_axis_tolerance = 0.05
orientation_constraint.absolute_z_axis_tolerance = 0.05
orientation_constraint.weight = 1.0

constraints.orientation_constraints.append( orientation_constraint )

group.set_path_constraints( constraints )
```

エンドエフェクタを現在の状態での水平を維持しながら動くように，エンドエフェクタ座標系で Y 軸 と Z 軸 の動作範囲を 0.05 [rad] 以内とし，X 軸 は 3.1415 として無視されるように拘束条件を設定しています.

また，拘束条件を付加した動作計画ではデフォルトのプランナのままだと動作計画の算出に時間がかかるのでプランナを RRTConnectkConfigDefault に set_planner_id() で変更し，group.allow_replanning(True) で動作計画の再計算を許可しています.

```
group.set_planner_id( "RRTConnectkConfigDefault" )
group.allow_replanning( True )
```

障害物回避動作目標設定では姿勢は拘束条件で設定しているので set_pose_target() は使わずに set_position_target() を使います.

```
xyz =  [ pose_target_2.position.x,
         pose_target_2.position.y,
         pose_target_2.position.z ]
group.set_position_target( xyz )
result_p = group.go()
```

そして位置の移動が正常に終了したのを確認して姿勢の修正を行います.

```
if result_p:
    group.set_pose_target( pose_target_2 )
    result_o = group.go()
```

```
$ rosrun tork_moveit_tutorial nextage_moveit_tutorial_poses_object_constraint.py
```

nextage_moveit_tutorial_poses_object_constraint.py

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

import sys, copy, math
import rospy, tf

from moveit_commander import MoveGroupCommander, PlanningSceneInterface
from geometry_msgs.msg import Pose, PoseStamped
from moveit_msgs.msg import Constraints, OrientationConstraint

from tork_moveit_tutorial import init_node


if __name__ == '__main__':

    init_node()

    group = MoveGroupCommander("right_arm")


    # Initialize the Planning Scene
    rospy.loginfo( "Setting the Planning Scene...")
    scene = PlanningSceneInterface()
    rospy.sleep(2)

    scene.remove_world_object()     # Remove all objects first
    rospy.sleep(2)

    rospy.loginfo( "All objects Removed : {}".format( scene.get_known_object_
↪names() ) )


    # Pose Target 1
    rospy.loginfo( "Start Pose Target 1")
    pose_target_1 = Pose()
```

(continues on next page)

(continued from previous page)

```
    pose_target_1.position.x = 0.3
    pose_target_1.position.y = -0.1
    pose_target_1.position.z = 0.15
    pose_target_1.orientation.x = 0.0
    pose_target_1.orientation.y = -0.707
    pose_target_1.orientation.z = 0.0
    pose_target_1.orientation.w = 0.707

    rospy.loginfo( "Set Target to Pose:n{}".format( pose_target_1 ) )

    group.set_pose_target( pose_target_1 )
    group.go()


    # Add Object to the Planning Scene
    rospy.loginfo( "Add Objects to the Planning Scene..." )
    box_pose = PoseStamped()
    box_pose.header.frame_id = group.get_planning_frame()
    box_pose.pose.position.x = 0.3
    box_pose.pose.position.y = -0.3
    box_pose.pose.position.z = -0.25
    box_pose.pose.orientation.w = 1.0

    scene.add_box( 'box_object', box_pose, ( 0.4, 0.1, 0.5 ) )
    rospy.sleep(2)

    rospy.loginfo( "Scene Objects : {}".format( scene.get_known_object_names() ) )


    # Set Path Constraint
    constraints = Constraints()
    constraints.name = "down"

    orientation_constraint = OrientationConstraint()
    orientation_constraint.header.frame_id = group.get_planning_frame()
    orientation_constraint.link_name = group.get_end_effector_link()
    orientation_constraint.orientation = pose_target_1.orientation
    orientation_constraint.absolute_x_axis_tolerance = 3.1415
    orientation_constraint.absolute_y_axis_tolerance = 0.05
    orientation_constraint.absolute_z_axis_tolerance = 0.05
    orientation_constraint.weight = 1.0

    constraints.orientation_constraints.append( orientation_constraint )

    group.set_path_constraints( constraints )
    rospy.loginfo( "Get Path Constraints:n{}".format( group.get_path_constraints()␣
↪) )


    # Pose Target 2
    rospy.loginfo( "Start Pose Target 2")
    pose_target_2 = Pose()
    pose_target_2.position.x = 0.3
    pose_target_2.position.y = -0.5
    pose_target_2.position.z = 0.15
    pose_target_2.orientation.x = 0.0
    pose_target_2.orientation.y = -0.707
    pose_target_2.orientation.z = 0.0
    pose_target_2.orientation.w = 0.707

    group.set_planner_id( "RRTConnectkConfigDefault" )
```

(continues on next page)

(continued from previous page)

```
    group.allow_replanning( True )

    rospy.loginfo( "Set Target to Pose:n{}".format( pose_target_2 ) )

    xyz =  [ pose_target_2.position.x,
             pose_target_2.position.y,
             pose_target_2.position.z ]
    group.set_position_target( xyz )
    result_p = group.go()

    rospy.loginfo( "Moving to the Position Executed... {}".format( result_p ) )

    group.clear_path_constraints()

    if result_p:
        group.set_pose_target( pose_target_2 )
        result_o = group.go()
        rospy.loginfo( "Adjusting the Orientation Executed... {}".format( result_o␣
↪) )
```

CHAPTER

FIVE

トラブルシューティング

5.1 動作計画が得られない

5.1.1 動作姿勢に問題がある

動作計画時にロボットの姿勢に問題がある場合は動作計画が得られません.

- ターゲットの姿勢がロボットの機構として可能な姿勢ではない
  - 位置が近すぎる・遠すぎる
    - 方向がロボットの構造的に無理がある
- ターゲットの姿勢間の動作計画軌道内に不可能な姿勢が存在する

【対策】姿勢を変更する・姿勢の経由点を追加する

ターゲット姿勢を変更したり，無理のない姿勢の経由点を追加するなどの対策を行ってみてください.

5.1.2 計算がタイムアウトする

動作計画の計算に要する時間が長くなる場合にはタイムアウトして結果が得られない場合があります.

【対策】動作計画のプランナを指定する

動作計画のプランナが「デフォルト」の設定の場合に計算時間が長くなることがあります．その場合にはプランナをデフォルト以外のもの，例えば `RRTConnectkConfigDefault` に変更すると短時間で計算が終了する場合があります.

```
group.set_planner_id( "RRTConnectkConfigDefault" )
```

【対策】動作計画の計算タイムアウト設定を長くする

動作計画に多くの計算が必要な場合は設定時間内に計算が終わらない場合があります．その場合は `set_planning_time()` でタイムアウトの設定を変更します.

```
group.set_planning_time( 30.0 )
```

5.2 動作計画が実行されない

5.2.1 動作計画軌道上に障害物が存在

動作計画の解が得られても実行時のチェックで障害物との干渉が発見されると動作が実行されません.

【対策】動作の再計画やターゲット姿勢などの変更

動作の再計画やターゲット姿勢の変更を行ってください.

5.3 tf が取得できない

5.3.1 画像処理ノードとの関係上の問題

画像処理には多くの計算リソースを使用することが多かったり，姿勢などのタイミングで画像処理の対象がカメラに映っていなかったりと，動作計画をする瞬間は適切に tf の発行がされていない場合があります.

【対策】tf 変換の試行回数を多くする

tf 変換の試行を長く，多くして結果を得られる機会を増やしてみてください.

本チュートリアルで準備している tork_moveit_tools にある `get_current_target_pose()` を使用する場合は引数の `timeout` を長めにします.

```
pose_target = get_current_target_pose( tgt_frame_id, planning_frame_id, 5.0 )
```

5.3.2 同期が取れていない - 実機ロボットの場合

シミュレータを使用している場合は大体は同一の PC 上で実行されていることが多いので tf の同期に問題が出ることはほぼありませんが，実機ロボットとの通信下で様々なノードを実行している場合は制御コンピュータやクライアント PC の間で同期が取れずに tf 変換がされない場合があります.

【対策】NTP サーバと同期をとる

NTP サーバとの同期をとる作業を数回行ってください.

```
$ sudo ntpdate -bv <ntp_server_address>
```

目安として時間のずれが 10 [msec] 以下になるようにします.

NTP サーバはロボットである場合や他のサーバである場合があります．ロボット管理者に NTP サーバの設定について問い合わせてください.

5.4 Gazebo の問題

5.4.1 Gazebo を起動してもロボットが表示されない

【対策】PC で初めて Gazebo を起動した場合は「しばらく待つ」

作業を行う PC で初めて Gazebo を起動する場合はモデルの読み込みなどに時間がかかることがあり，単に時間がかかっているだけということがあります.

【対策】ROS 環境設定を確認する

ROS の環境設定に問題がある場合があります．ターミナルで ROS の環境設定がされているか確認します．

```
$ env | grep ROS
```

正常な場合の出力例

```
ROS_ROOT=/opt/ros/melodic/share/ros
ROS_PACKAGE_PATH=/opt/ros/melodic/share:/opt/ros/melodic/stacks
ROS_MASTER_URI=http://localhost:11311
ROSLISP_PACKAGE_DIRECTORIES=
ROS_DISTRO=melodic
ROS_ETC_DIR=/opt/ros/melodic/etc/ros
```

ROS の環境設定に問題がある場合は設定し直します．

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

【対策】全ての ROS ノードを終了して再起動

何度か Gazebo などを利用して終了を繰り返していると ROS のノードが終了しきれていないなどの理由で新しい ROS ノードの起動に不具合が生じることがあります．

たまに Gazebo が正常に起動しなくなるような場合はこれに相当することが多いです．

Gazebo を Ctrl-C で終了して他の ROS ノードも全て終了していることを確認してから Gazebo を再起動してみてください．

CHAPTER

SIX

クラス・関数リファレンス

6.1 MoveIt! Commander

6.1.1 MoveGroupCommander クラス

GitHub - moveit_commander/src/moveit_commander/move_group.py

GitHub - moveit/moveit_ros/planning_interface/move_group_interface/src/wrap_python_move_group.cpp

---

def init(self, name, robot_description= ”robot_description ”):

- 機能
  - コンストラクタ
- 引数
  - `name` - str : グループ名
  - `robot_description` - str : ロボット名
- 戻り値
  - なし

---

def get_name(self):

- 機能
  - グループ名の取得
- 引数
  - なし
- 戻り値
  - str : グループ名

---

stop(self):

- 機能
  - 実行の停止
- 引数
  - なし
- 戻り値
  - なし

---

def get_active_joints(self):

- 機能
  - グループの機能している関節名の取得
- 引数
  - なし
- 戻り値
  - str : 機能している関節名のリスト

---

def get_joints(self):

- 機能
  - グループ内の関節名の取得
- 引数
  - なし
- 戻り値
  - str : 関節名のリスト

---

def get_variable_count(self):

- 機能
  - グループの状態をパラメータ化するために使用される変数の数を返す（DOF の数より大きいか等しい）
- 引数
  - なし
- 戻り値
  - int : 変数の数

---

has_end_effector_link(self):

- 機能
  - グループのエンドエフェクタリンクの有無のチェック
- 引数
  - なし
- 戻り値
  - bool : 有無

---

def get_end_effector_link(self):

- 機能
  - エンドエフェクタリンク名の取得
- 引数
  - なし
- 戻り値
  - str : リンク名（ 空の文字列ならエンドエフェクタリンク設定無し ）

---

def set_end_effector_link(self, link_name):

- 機能
  - エンドエフェクタとするリンク名の設定
- 引数
  - `link_name` - str : リンク名
- 戻り値
  - なし

---

def get_pose_reference_frame(self):

- 機能
  - エンドエフェクタの姿勢定義参照リンク名の取得
- 引数
  - なし
- 戻り値
  - str : リンク名

---

def set_pose_reference_frame(self, reference_frame):

- 機能
  - エンドエフェクタの姿勢定義参照リンク名の設定
- 引数
  - `link_name` - str : リンク名
- 戻り値
  - なし

---

def get_planning_frame(self):

- 機能
  - 動作計画で使用されるフレーム名の取得
- 引数
  - なし
- 戻り値
  - str : リンク名

---

def get_current_joint_values(self):

- 機能
  - 現在の関節角度値リストの取得
- 引数
  - なし
- 戻り値
  - [float, float, float, ⋯] : 関節角度値のリスト

---

def get_current_pose(self, end_effector_link = “ ”):

- 機能
  - グループのエンドエフェクタの姿勢の取得
- 引数
  - `end_effector_link` - str : エンドエフェクタの名前
- 戻り値
  - PoseStamped : エンドエフェクタの姿勢

---

def get_current_rpy(self, end_effector_link = “ ”):

- 機能
  - グループのエンドエフェクタの Roll, Pitch, Yaw で定義される姿勢の取得
- 引数
  - `end_effector_link` - str : エンドエフェクタの名前
- 戻り値
  - [float, float, float] : エンドエフェクタの姿勢 [Roll, Pitch, Yaw] [rad]

---

def get_random_joint_values(self):

- 機能
  - ランダムな関節角度値の取得
- 引数
  - なし
- 戻り値
  - [float, float, float, ⋯] : 関節角度値のリスト

---

def get_random_pose(self, end_effector_link = “ ”):

- 機能
  - ランダムなエンドエフェクタ姿勢の取得
- 引数
  - `end_effector_link` - str : エンドエフェクタの名前
- 戻り値
  - PoseStamped : エンドエフェクタの姿勢

---

def set_start_state_to_current_state(self):

- 機能
  - ロボットの現在の状態を動作計画の開始姿勢に設定
- 引数
  - なし
- 戻り値
  - なし

---

def set_start_state(self, msg):

- 機能
  - ロボットの現在の状態の代わりに異なる開始状態の設定
- 引数
  - `msg` - RobotState : ロボットの状態（ moveit_msgs/RobotState.msg ）
- 戻り値
  - なし

---

def set_joint_value_target(self, arg1, arg2 = None, arg3 = None):

- 機能
  - グループの関節角度目標値の設定
- 引数
  - `arg1` - dict or list or JointState : 関節角度目標値データセット [rad]
    - `arg2`, `arg3` = None : 設定の必要なし
  - `arg1` - string : 特定のジョイント名
    - `arg2` - float or [float, float, ⋯] : 特定関節の角度目標値 [rad]
  - `arg1` - Pose or PoseStamped
    - `arg2`, `arg3` - str or bool - str : 姿勢が指定されたエンドエフェクタ- bool : 指定された姿勢が近似値か否かを決定 / デフォルト False（厳密解）
    - 逆運動学計算を行いグループの関節角度目標値を設定
- 戻り値
  - なし

---

def set_rpy_target(self, rpy, end_effector_link = “ ”):

- 機能
  - エンドエフェクタの目標姿勢 Roll, Pitch, Yaw 角度を設定
- 引数
  - `rpy` - [float, float, float] : 目標姿勢角度 Roll, Pitch, Yaw [rad]
  - `end_effector_link` - str : エンドエフェクタの名前
- 戻り値
  - なし

---

def set_orientation_target(self, q, end_effector_link = “ ”):

- 機能
  - エンドエフェクタの目標姿勢クォータニオンを設定
- 引数
  - `q` - [float, float, float, float] : 目標姿勢クオータニオン
  - `end_effector_link` - str : エンドエフェクタの名前
- 戻り値
  - なし

---

def set_position_target(self, xyz, end_effector_link = “ ”):

- 機能
  - エンドエフェクタの目標位置を設定
- 引数
  - `xyz` - [float, float, float] : 目標位置 X, Y, Z [m]
  - `end_effector_link` - str : エンドエフェクタの名前
- 戻り値
  - なし

---

def set_pose_target(self, pose, end_effector_link = “ ”):

- 機能
  - エンドエフェクタの目標姿勢（位置・方向）の設定
- 引数
  - `pose` : 次のいずれか
    - [x, y, z, rot_x, rot_y, rot_z] : 位置座標と Roll/Pitch/Yaw 姿勢角の 6 つの数値のリスト
    - [x, y, z, qx, qy, qz, qw] : 位置座標とクォータニオンの 7 つの数値のリスト
    - Pose
    - PoseStamped
  - `end_effector_link` - str : エンドエフェクタ名 / デフォルト“ ”
- 戻り値
  - なし

---

def set_pose_targets(self, poses, end_effector_link = “ ”):

- 機能
  - エンドエフェクタの複数の目標姿勢（位置・方向）の設定
- 引数
  - `poses` - [Pose, Pose, Pose, ···] : 目標姿勢 Pose のリスト
  - `end_effector_link` - str : エンドエフェクタ名 / デフォルト“ ”
- 戻り値
  - なし

---

def shift_pose_target(self, axis, value, end_effector_link = “ ”):

- 機能
  - エンドエフェクタの現在の姿勢を取得し，対応する軸に値を加えてそれを目標姿勢として設定
- 引数
  - `axis` - int : 0〜5 がそれぞれ X, Y, Z, R, P, Y 軸に対応
  - `value` - float : 加算する値 [m] or [rad]
  - `end_effector_link` - str : エンドエフェクタ名 / デフォルト“ ”
- 戻り値
  - なし

---

def clear_pose_target(self, end_effector_link):

- 機能
  - 指定エンドエフェクタの目標姿勢の消去
- 引数
  - `end_effector_link` - str : エンドエフェクタ名
- 戻り値
  - なし

---

def clear_pose_targets(self):

- 機能
  - 全ての目標姿勢の消去
- 引数
  - なし
- 戻り値
  - なし

---

def set_random_target(self):

- 機能
  - ランダムな関節角度目標値の設定
- 引数
  - なし
- 戻り値
  - なし

---

def set_named_target(self, name):

- 機能
  - 下記により名称が設定されている関節角度目標値の設定
    - remember_joint_values() で記録
    - SRDF で設定
- 引数
  - `name` - str : 関節角度値の名称
- 戻り値
  - なし

---

def remember_joint_values(self, name, values = None):

- 機能
  - グループの関節角度値構成を名前をつけて記録
- 引数
  - `name` - str : 関節角度値構成の記録名称
  - `values` - [float, float, float, ⋯] : 関節角度値のリスト/ デフォルト None → get_current_joint_values() で現在の関節角度値を取得
- 戻り値
  - なし

---

def get_remembered_joint_values(self):

- 機能
  - グループで記録されている関節角度値のディクショナリを取得
- 引数
  - なし
- 戻り値
  - dict : 関節角度値構成のディクショナリ

---

def forget_joint_values(self, name):

- 機能
  - 指定した名称の関節角度値の破棄
- 引数
  - `name` - str : 関節角度値の名称
- 戻り値
  - なし

---

def get_goal_tolerance(self):

- 機能
  - 関節角度，位置，方向の目標に対する許容値の取得
- 引数
  - なし
- 戻り値
  - tuple : 関節角度，位置，姿勢の目標に対する許容値

---

def get_goal_joint_tolerance(self):

- 機能
  - 関節角度の目標に対する許容値の取得
- 引数
  - なし
- 戻り値
  - float : 関節角度の目標に対する許容値

---

def get_goal_position_tolerance(self):

- 機能
  - エンドエフェクタ位置の目標に対する許容値の取得
    - 許容値は目標位置を中心とした球として設定
- 引数
  - なし
- 戻り値
  - float : 位置の目標に対する許容値

---

def get_goal_orientation_tolerance(self):

- 機能
  - エンドエフェクタの姿勢（方向）の目標に対する許容値の取得
- 引数
  - なし
- 戻り値
  - float : 姿勢（方向）の目標に対する許容値

---

def set_goal_tolerance(self, value):

- 機能
  - 関節角度，位置および方向の目標に対する許容値を同時に設定
- 引数
  - `value` - float : 関節角度，位置および方向の目標に対する許容値
- 戻り値
  - なし

---

def set_goal_joint_tolerance(self, value):

- 機能
  - 関節角度の目標に対する許容値を設定
- 引数
  - `value` - float : 関節角度の目標に対する許容値
- 戻り値
  - なし

---

def set_goal_position_tolerance(self, value):

- 機能
  - 位置の目標に対する許容値を設定
- 引数
  - `value` - float : 位置の目標に対する許容値
- 戻り値
  - なし

---

def set_goal_orientation_tolerance(self, value):

- 機能
  - 方向の目標に対する許容値を設定
- 引数
  - `value` - float : 方向の目標に対する許容値
- 戻り値
  - なし

---

def allow_looking(self, value):

- 機能
  - ロボットが移動する前に見回すことができるかどうかを指定
- 引数
  - `value` - bool : 見回すか否か / デフォルト True
- 戻り値
  - なし

---

def allow_replanning(self, value):

- 機能
  - 環境の変化を検出したときに動作の再計画をするか否かを指定
- 引数
  - `value` - bool : 再計画するか否か
- 戻り値
  - なし

---

def get_known_constraints(self):

- 機能
  - このグループに固有の拘束条件の名前のリストをデータベースから取得
- 引数
  - なし
- 戻り値
  - [str, str, …] : 名前のリスト

---

def get_path_constraints(self):

- 機能
  - moveit_msgs.msgs.Constraints 形式で実際の軌道拘束条件を取得
- 引数
  - なし
- 戻り値
  - Constraints : 軌道拘束条件

---

def set_path_constraints(self, value):

- 機能
  - 軌道拘束条件の設定
- 引数
  - `value` - Constraints : 軌道拘束条件
- 戻り値
  - なし

---

def clear_path_constraints(self):

- 機能
  - 軌道拘束条件の消去
- 引数
  - なし
- 戻り値
  - なし

---

def set_constraints_database(self, host, port):

- 機能
  - 既知の拘束条件を保持しているデータベースサーバが存在する場所を設定
- 引数
  - `host` - string : ホスト名
    - `port` - int : ポート番号
- 戻り値
  - なし

---

def set_planning_time(self, seconds):

- 機能
  - 動作計画に使用する時間の設定
- 引数
  - `seconds` - float : 時間 [s]
- 戻り値
  - なし

---

def get_planning_time(self):

- 機能
  - 動作計画に使用する時間の取得
- 引数
  - なし
- 戻り値
  - float : 時間 [s]

---

def set_planner_id(self, planner_id):

- 機能
  - 動作計画に使用するプランナの設定
- 引数
  - `planner_id` - str : プランナの名称
- 戻り値
  - なし

---

def set_num_planning_attempts(self, num_planning_attempts):

- 機能
  - 最短解が返される前に動作計画を計算する回数の設定
- 引数
  - `num_planning_attempts` - int : 計算回数 / デフォルト 1
- 戻り値
  - なし

---

def set_workspace(self, ws):

- 機能
  - ロボットのワークスペースの設定
- 引数
  - `ws` - [float, float, ···] : ワークスペース
    - [minX, minY, maxX, maxY] or
    - [minX, minY, minZ, maxX, maxY, maxZ]
- 戻り値
- なし

---

def set_max_velocity_scaling_factor(self, value):

- 機能
  - 最大関節角速度減少係数の設定
- 引数
  - `value` - float : 速度減少係数 0.0〜1.0
- 戻り値
  - なし

---

def go(self, joints = None, wait = True):

- 機能
  - 目標を設定してグループを動かす
- 引数
  - `joints` - bool, JointState, Pose : グループの動作目標（ bool の場合は wait に代入) / デフォルト None
    - `wait` - bool : 動作の終了を待つか否か / デフォルト True
- 戻り値
  - MoveItErroeCode : MoveIt! エラーコード

---

def plan(self, joints = None):

- 機能
  - 目標を設定し，動作計画を計算して返す
- 引数
  - `joints` - JointState, Pose, str, [float, float, ···] : グループの動作目標/ デフォルト None
- 戻り値
  - RobotTrajectory : 動作計画軌道

---

def compute_cartesian_path(self, waypoints, eef_step, jump_threshold, avoid_collisions = True):

- 機能
  - ウェイポイントとして定義された複数の経路上の位置・姿勢に向かってエンドエフェクタを線形補間にて動作させる軌道を作成して返す
- 引数
  - `waypoints` - [Pose, Pose, ⋯] : エンドエフェクタが経由する姿勢のリスト
  - `eef_step` - float : エンドエフェクタの姿勢を計算する間隔の距離
  - `jump_threshold` - float : 軌道内の連続する点間の最大距離（`0.0` で無効）
  - `avoid_collisions` bool : 干渉と運動学上の制約チェック（デフォルトは `True` でチェックする）
- 戻り値
  - tuple : (plan, fraction)
    - `plan` - RobotTrajectory : 動作計画軌道
    - `fraction` - float : ウェイポイントによって記述されたパスの割合 0.0〜1.0 / エラーの場合は -1.0

---

def execute(self, plan_msg, wait = True):

- 機能
  - 事前に動作計画された動作を実行
- 引数
  - `plan_msg` - Plan : 動作計画軌道
  - `wait` - bool : 動作の終了を待つか否か / デフォルト True
- 戻り値
  - MoveItErroeCode : MoveIt! エラーコード

---

def attach_object(self, object_name, link_name = “ ”, touch_links = []):

- 機能
  - 動作計画シーン内にあるオブジェクトをロボットのリンクに接続
- 引数
  - `object_name` - str : ロボットに接続するオブジェクト名
  - `link_name` -str : オブジェクトを接続するロボットのリンク名 / デフォルト エンドエフェクタ
  - `touch_links` - [str, str, ⋯] : オブジェクトが干渉を考慮せずに接触することが許されるリンク群
- 戻り値
  - bool : Move Group ノードにオブジェクトの接続リクエストが送信されたときに True を返す/ これは接続リクエストの実行が成功したか否かは意味しない

---

def detach_object(self, name = “ ”):

- 機能
  - 指定したリンクからオブジェクトを切り離す
- 引数
  - `name` - str : オブジェクトを切り離すリンク名
    - リンク名が存在しない場合はオブジェクト名として解釈
    - 名前が指定されていない場合はグループ内のリンクに接続されている全てのオブジェクトを切り離そうとする
- 戻り値
  - bool : 切り離すオブジェクトが特定できない場合はエラーを生成

---

def pick(self, object_name, grasp = []):

- 機能
  - 名称のあるオブジェクトの持ち上げ
- 引数
  - `object_name` - str : 掴む対象オブジェクトの名称
  - `grasp` - Grasp or [Grasp, Grasp, ···] : 掴む動作の指定
- 戻り値
  - MoveItErroeCode : MoveIt! エラーコード

---

def place(self, object_name, location=None):

- 機能
  - オブジェクトを置く
- 引数
  - `object_name` - str : 置くオブジェクトの名称
  - `location` - PoseStamped, Pose, PlaceLocation : オブジェクトを置く場所/ デフォルト None （環境内の安全な場所を検出してそこへ置く）
- 戻り値
  - MoveItErroeCode : MoveIt! エラーコード

---

def set_support_surface_name(self, value):

- 機能
  - place() 動作用の支持サーフェス名を設定
- 引数
  - value - str : サーフェス名
- 戻り値

  - なし

---

def retime_trajectory(self, ref_state_in, traj_in, velocity_scaling_factor):

- 機能
  - 軌道のタイムスタンプを速度と加速度の制約を考慮して変更
- 引数
  - `ref_state_in` - RobotState : 現在のロボット全体の状態
  - `traj_in` - RobotTrajectory : タイムスタンプの再設定する動作計画軌道
  - `velocity_scaling_factor` - float : 速度係数
- 戻り値
  - RobotTrajectory : 動作計画軌道

6.1.2 RobotCommander クラス

GitHub - moveit_commander/src/moveit_commander/robot.py

GitHub - moveit/moveit_ros/planning_interface/robot_interface/src/wrap_python_robot_interface.cpp

---

def init(self, robot_description= ”robot_description ”):

- 機能
  - コンストラクタ
- 引数
  - `robot_description` - str : ロボット名
- 戻り値
  - なし

---

def get_planning_frame(self):

- 機能
  - 動作計画で使用された参照フレーム名の取得
- 引数
  - なし
- 戻り値
  - str : リンク名

---

def get_root_link(self):

- 機能
  - ロボットモデルのルートリンク名の取得
- 引数
  - なし
- 戻り値
  - str : リンク名

---

def get_joint_names(self, group=None):

- 機能
  - 関節名の取得
- 引数
  - `group` - str : グループ名 / デフォルト None（ロボットの全関節名を返す）
- 戻り値
  - [str, str, …] : 関節名のリスト

---

def get_link_names(self, group=None):

- 機能
  - リンク名の取得
- 引数
  - `group` - str : グループ名 / デフォルト None（ロボットの全リンク名を返す）
- 戻り値
  - [str, str, …] : リンク名のリスト

---

def get_current_state(self):

- 機能
  - ロボットの現在の状態の RobotState 型メッセージの取得
- 引数
  - なし
- 戻り値
  - RobotStage : ロボットの現在の状態

---

def get_current_variable_values(self):

- 機能
  - ジョイント名とジョイント状態値のディクショナリとして現在の状態を取得
- 引数
  - なし
- 戻り値
  - dictionary : ジョイント名とジョイント状態値のディクショナリ

---

def get_joint(self, name):

- 機能
  - 関節の Joint クラスの取得
- 引数
  - `name` - str : 関節名
- 戻り値
  - Joint : Joint クラス

---

def get_link(self, name):

- 機能
  - リンクの Link クラスの取得
- 引数
  - `name` - str : 関節名
- 戻り値
  - Link : Link クラス

---

def get_group(self, name):

- 機能
  - グループ名を指定してその MoveGroupCommander クラスを取得
- 引数
  - `name` - str : グループ名
- 戻り値
  - MoveGroupCommander クラス :

---

def has_group(self, name):

- 機能
  - グループ名のグループが存在するかを確認
- 引数
  - `name` - str : グループ名
- 戻り値
  - bool : グループが存在するか否か

---

def get_default_owner_group(self, joint_name):

- 機能
  - 引数指定された関節名を含む最小のグループの名前の取得
- 引数
  - `joint_name` - str : 関節名
- 戻り値
  - str : グループ名

6.1.3 Joint クラス（ RobotCommandeer 内クラス ）

---

def init(self, robot, name):

- 機能
  - コンストラクタ
- 引数
  - `robot` - RobotCommander : RobotCommander クラスインスタンス
  - `name` - str : 関節名
- 戻り値
  - なし

---

def name(self):

- 機能
  - 関節名の取得
- 引数
  - なし
- 戻り値
  - str : 関節名

---

def variable_count(self):

- 機能
  - 関節を記述する変数の数の取得
- 引数
  - なし
- 戻り値
  - int : 変数の数

---

def bounds(self):

- 機能
  - 関節リミットの最小値と最大値のリストもしくはそれらのセットリストの取得
- 引数
  - なし
- 戻り値
  - [float, float] or [[float, float], ···] : 関節リミットのリスト

---

def min_bound(self):

- 関節リミットの最小値もしくはそれらのセットリストの取得
- 引数
  - なし
- 戻り値
  - float or [[float, float], ···] : 関節リミットの最小値（のリスト）

---

def max_bound(self):

- 関節リミットの最大値もしくはそれらのセットリストの取得
- 引数
  - なし
- 戻り値
  - float or [[float, float], ···] : 関節リミットの最大値（のリスト）

---

def value(self):

- 機能
  - 関節の現在の値の取得
- 引数
  - なし
- 戻り値
  - float : 関節の現在の値

---

def move(self, position, wait=True):

- 機能
  - 関節を目標値に動かす
- 引数
  - `position` - float : 目標値
  - `wait` - bool : 動作完了を待つか否か / デフォルト True
- 戻り値
- MoveItErroeCode or bool : MoveIt! エラーコード もしくは False

---

def __get_joint_limits(self):

- 機能
  - 関節リミットの最大値と最小値のリストのセットリストの取得
- 引数
  - なし
- 戻り値
  - [float, float] or [[float, float], ···] : 関節リミットのリスト

6.1.4 Link クラス（ RobotCommandeer 内クラス ）

---

def init(self, robot, name):

- 機能
  - コンストラクタ
- 引数
  - `robot` - RobotCommander : RobotCommander クラスインスタンス
  - `name` - リンク名
- 戻り値
  - なし

---

def name(self):

- 機能
  - リンク名の取得
- 引数
  - なし
- 戻り値
  - str : リンク名

def pose(self):

- 機能
  - リンクの姿勢の取得
- 引数
  - なし
- 戻り値
  - PoseStamped : リンクの姿勢

6.1.5 PlanningSceneInterface クラス

GitHub - moveit_commander/src/moveit_commander/planning_scene_interface.py

GitHub - moveit/moveit_ros/planning_interface/planning_scene_interface/src/wrap_python_planning_scene_interface.cpp

---

def init(self):

- 機能
  - コンストラクタ
- 引数
  - なし
- 戻り値
  - なし

---

def __make_sphere(self, name, pose, radius):

- 機能
  - 球形状オブジェクトの作成
- 引数
  - `name` - str : オブジェクトの名前
  - `pose` - PoseStamped : オブジェクトの姿勢
  - `radius` - float : 球の半径 [m]
- 戻り値
  - ColllisionObject : オブジェクト

---

def add_sphere(self, name, pose, radius = 1):

- 機能
  - 球形状オブジェクトを作成して動作計画空間に設置
- 引数
  - `name` - str : オブジェクトの名前
  - `pose` - PoseStamped : オブジェクトの姿勢
  - `radius` - float : 球の半径 [m]
- 戻り値
  - なし

---

def __make_box(self, name, pose, size):

- 機能
  - 箱形状オブジェクトの作成
- 引数
  - `name` - str : オブジェクトの名前
  - `pose` - PoseStamped : オブジェクトの姿勢
  - `size` - (float, float, float) : 三辺の各長さのタプル [m]
- 戻り値
  - ColllisionObject : オブジェクト

---

def add_box(self, name, pose, size = (1, 1, 1)):

- 機能
  - 箱形状オブジェクトを作成して動作計画空間に設置
- 引数
  - `name` - str : オブジェクトの名前
  - `pose` - PoseStamped : オブジェクトの姿勢
  - `size` - (float, float, float) : 三辺の各長さのタプル [m]
- 戻り値
  - なし

---

def __make_mesh(self, name, pose, filename, scale = (1, 1, 1)):

- 機能
  - メッシュデータからオブジェクトを作成
- 引数
  - `name` - str : オブジェクトの名前
  - `pose` - PoseStamped : オブジェクトの姿勢
  - `filename` - str : メッシュデータファイル名
  - `scale` - (float, float, float) : スケール / デフォルト (1, 1, 1)
- 戻り値
  - ColllisionObject : オブジェクト

---

def add_mesh(self, name, pose, filename, size = (1, 1, 1)):

- 機能
  - メッシュデータからオブジェクトを作成して動作計画空間に設置
- 引数
  - `name` - str : オブジェクトの名前
  - `pose` - PoseStamped : オブジェクトの姿勢
  - `filename` - str : メッシュデータファイル名
  - `scale` - (float, float, float) : スケール / デフォルト (1, 1, 1)
- 戻り値
  - なし

---

def __make_existing(self, name):

- 機能
  - オブジェクトが既に存在するときに使用される空のオブジェクトの作成
- 引数
  - `name` - str : オブジェクト
- 戻り値
  - ColllisionObject : オブジェクト

---

def add_plane(self, name, pose, normal = (0, 0, 1), offset = 0):

- 機能
  - 平面オブジェクトを作成して動作計画空間に設置
- 引数
  - `name` - str : オブジェクトの名前
  - `pose` - PoseStamped : オブジェクトの姿勢
  - `normal` - (float, float, float) : 面の法線ベクトル / デフォルト (0, 0, 1)
  - `offset` - float : オフセット量 [m] / デフォルト 0
- 戻り値
  - なし

---

def attach_mesh(self, link, name, pose = None, filename = ‘ ’, size = (1, 1, 1), touch_links = []):

- 機能
  - メッシュデータをロボットリンクに接続
    - 指定があればメッシュオブジェクトを作成
- 引数
  - `link` - str : 接続するロボットリンク名
  - `name` - str : 接続するオブジェクトの名前
  - `pose` - PoseStamped : オブジェクトの姿勢
  - `filename` - str : メッシュデータファイル名
  - `size` - (float, float, float) : サイズスケール / デフォルト (1, 1, 1)
  - `touch_links` - - [str, str, ···] : オブジェクトが干渉を考慮せずに接触することが許されるリンク群
- 戻り値
  - なし

---

def attach_box(self, link, name, pose = None, size = (1, 1, 1), touch_links = []):

- 機能
  - 箱オブジェクトをロボットリンクに接続
    - 指定があれば箱オブジェクトを作成
- 引数
  - `link` - str : 接続するロボットリンク名
  - `name` - str : 接続するオブジェクトの名前
  - `pose` - PoseStamped : オブジェクトの姿勢
  - `size` - (float, float, float) : サイズ / デフォルト (1, 1, 1)
  - `touch_links` - - [str, str, ···] : オブジェクトが干渉を考慮せずに接触することが許されるリンク群
- 戻り値
  - なし

---

def remove_world_object(self, name = None):

- 機能
  - 動作計画空間からオブジェクトを削除
    - オブジェクト名の指定がなければ全オブジェクトを削除
- 引数
  - `name` - str : 削除するオブジェクト名
- 戻り値
  - なし

---

def remove_attached_object(self, link, name = None):

- 機能
  - リンクに接続している動作計画空間内のオブジェクトを削除
    - オブジェクト名の指定がなければリンクに接続している全オブジェクトを削除
- 引数
  - `link` - str : 削除するオブジェクトの接続しているリンク名
  - `name` - str : 削除するオブジェクト名
- 戻り値
  - なし

---

def get_known_object_names(self, with_type = False):

- 機能
  - 既知のオブジェクトの全名称の取得
- 引数
  - `with_type` - bool : 既知の型を持つオブジェクトのみを返すか否か / デフォルト False
- 戻り値
  - [str, str, ···] : オブジェクトの名称リスト

---

def get_known_object_names_in_roi(self, minx, miny, minz, maxx, maxy, maxz, with_type = False):

- 機能
  - 領域内にある既知のオブジェクトの全名称の取得
- 引数
  - `minx` - float : 領域の X 方向最小値
  - `miny` - float : 領域の Y 方向最小値
  - `minz` - float : 領域の Z 方向最小値
  - `maxx` - float : 領域の X 方向最大値
  - `maxy` - float : 領域の Y 方向最大値
  - `maxz` - float : 領域の Z 方向最大値
  - `with_type` - bool : 既知の型を持つオブジェクトのみを返すか否か / デフォルト False
- 戻り値
  - [str, str, ···] : オブジェクトの名称リスト

---

def get_objects(self, object_ids = []):

- 機能
  - リストで指定したオブジェクトデータの取得
- 引数
  - `object_ids` - [str, str, ···] : オブジェクト名のリスト（指定がなければ全て）
- 戻り値
  - { str:CollisionObject, str:CollisionObject, ··· } : オブジェクトのディクショナリ

---

def get_attached_objects(self, object_ids = []):

- 機能
  - リンク接続しているリストで指定したオブジェクトデータの取得
- 引数
  - `object_ids` - [str, str, …] : オブジェクト名のリスト（指定がなければ全て）
- 戻り値
  - { str:CollisionObject, str:CollisionObject, … } : オブジェクトのディクショナリ

6.1.6 conversions の関数

- GitHub - moveit_commander/src/moveit_commander/conversions.py

---

def msg_to_string(msg):

- 機能
  - ROS メッセージから文字列への変換
- 引数
  - `msg` - ROS Message : ROS メッセージ
- 戻り値
  - str : 文字列

---

def msg_from_string(msg, data):

- 機能
  - 文字列から ROS メッセージへの変換
- 引数
  - `msg` - ROS Message : ROS メッセージ
  - `data` - str : 文字列
- 戻り値
  - なし

---

def pose_to_list(pose_msg):

- 機能
  - Pose からリストへの変換
- 引数
  - `pose_msg` - Pose : Pose メッセージ
- 戻り値
  - [float, float, ···] : [x, y, z, qx, qy, qz, qw]

---

def list_to_pose(pose_list):

- 機能
  - リストから Pose への変換
- 引数
  - `pose_list` - [float, float, ···] : [x, y, z, qx, qy, qz, qw]
- 戻り値
  - Pose : Pose メッセージ

---

def list_to_pose_stamped(pose_list, target_frame):

- 機能
  - リストから PoseStamped への変換
- 引数
  - `pose_list` - [float, float, ···] : [x, y, z, qx, qy, qz, qw]
  - `target_frame` - str : 基準フレーム
- 戻り値
  - Pose : Pose メッセージ

---

def transform_to_list(trf_msg):

- 機能
  - Transform からリストへの変換
- 引数
  - `trf_msg` - Transform : Transform メッセージ
- 戻り値
  - [float, float, ···] : [x, y, z, qx, qy, qz, qw]

---

def list_to_transform(trf_list):

- 機能
  - リストから Transform への変換
- 引数
  - `trf_list` - [float, float, ⋯] : [x, y, z, qx, qy, qz, qw]
- 戻り値
  - Transform : Transform メッセージ

6.1.7 roscpp_initializer の関数

- GitHub - moveit_commander/src/moveit_commander/roscpp_initializer.py

---

def roscpp_initialize(args):

- 機能
  - moveit_commander プロセスの初期化
- 引数
  - `args` - [str, str, ⋯] : スクリプトの名前とコマンドライン引数のリスト
- 戻り値
  - なし

---

def roscpp_shutdown():

- 機能
  - moveit_commander プロセスの終了
- 引数
  - なし
- 戻り値
  - なし

6.2 チュートリアルパッケージ

6.2.1 moveit_tutorial_tools の関数

---

def init_node(node_name = “ commander_example ”):

- 機能
  - ROS と Qt アプリケーション の初期化
  - ロボットの動作グループ（ Move Group ）の表示
- 引数
  - `node_name` : str - 初期化する ROS ノードにつける名称 / デフォルト“ commander_example ”

---

def question_yn(qmsg= ’Message ’, title= ’Question ’):

- 機能
  - Yes / No を問い合わせて Yes が選択された場合にのみ `True` を返す
  - PyQt の QMessageBox() を利用
- 引数
  - `qmsg` - str : 問い合わせメッセージの文字列 / デフォルト‘ Message ’
  - `title` - str : メッセージボックスウィンドウのタイトル文字列 / デフォルト‘ Question ’
- 戻り値
  - bool - Yes が選択された場合のみ `True` を返す / その他は `False`

---

def get_current_target_pose(target_frame_id, base_frame_id, timeout = 1.0):

- 機能
  - 引数で指定されたのフレーム間の TF を返す
- 引数
  - `target_frame_id` - str : TF の値を得たいフレームの名称
    - `base_frame_id` - str : TF 変換の基準とするフレームの名称
    - `timeout` - float : TF 変換のタイムアウト時間 [sec] / デフォルト 1.0

---

def make_waypoints_example(rtype= ”NEXTAGE ”):

- 機能
  - 動作例用の複数の姿勢の作成
- 引数
  - `rtype` - str 型 : ロボットのタイプ名 / デフォルト“ NEXTAGE ”
- 戻り値
  - [Pose, Pose, ···] : 動作例用の複数のポーズのリスト

---

def make_waypoints_rectangular(dp_a=[0.25, 0.0, 0.1], dp_b=[0.45, -0.2, 0.1], rpy=[0.0,0.0,0.0]):

- 機能
  - 四角形を描く動作の各頂点の姿勢の作成
- 引数
  - `dp_a` - [float, float, float] : 1 つ目の四角形の対角点 / デフォルト [0.25, 0.0, 0.1]
  - `dp_b` - [float, float, float] : 2 つ目の四角形の対角点 / デフォルト [0.45, -0.2, 0.1]
  - `rpy` - [float, float, float] : 四角形を描く動作時の姿勢 Roll,Pitch,Yaw [rad] / デフォルト [0.0,0.0,0.0]
- 戻り値
  - [Pose, Pose, ···] : 四角形を描く動作の各頂点の姿勢

---

def make_waypoints_circular(center=[0.3, -0.2, 0.1], radius=0.1 ,steps=12, rpy=[0.0,0.0,0.0]):

- 機能
  - 円を描く動作の円周上の姿勢の作成
- 引数
  - `center` - [float, float, float] : 円の中心の位置 / デフォルト [0.3, -0.2, 0.1]
  - `radius` - float : 円の半径 [m] / デフォルト 0.1
    - `steps` - int : 円周の分割数 / デフォルト 12
  - `rpy` - [float, float, float] : 円を描く動作時の姿勢 Roll,Pitch,Yaw [rad] / デフォルト [0.0,0.0,0.0]
- 戻り値
  - [Pose, Pose, ···] : 円を描く動作の各頂点の姿勢

CHAPTER

SEVEN

PYTHON チュートリアル

7.1 Python とは

本チュートリアルではロボットの操作にプログラミング言語 Python を使用しています．Python はロボットに限らず広く利用されているコンピュータプログラミング言語です．

Python はプログラムの文を書いたらすぐに実行できるので非常に便利です．プログラムを書いては実行して試して修正して…を頻繁に繰り返せるのでプログラミングの習得に適していますし，ソフトウェア開発作業でも不具合修正に非常に適しています．

7.2 Python をはじめる

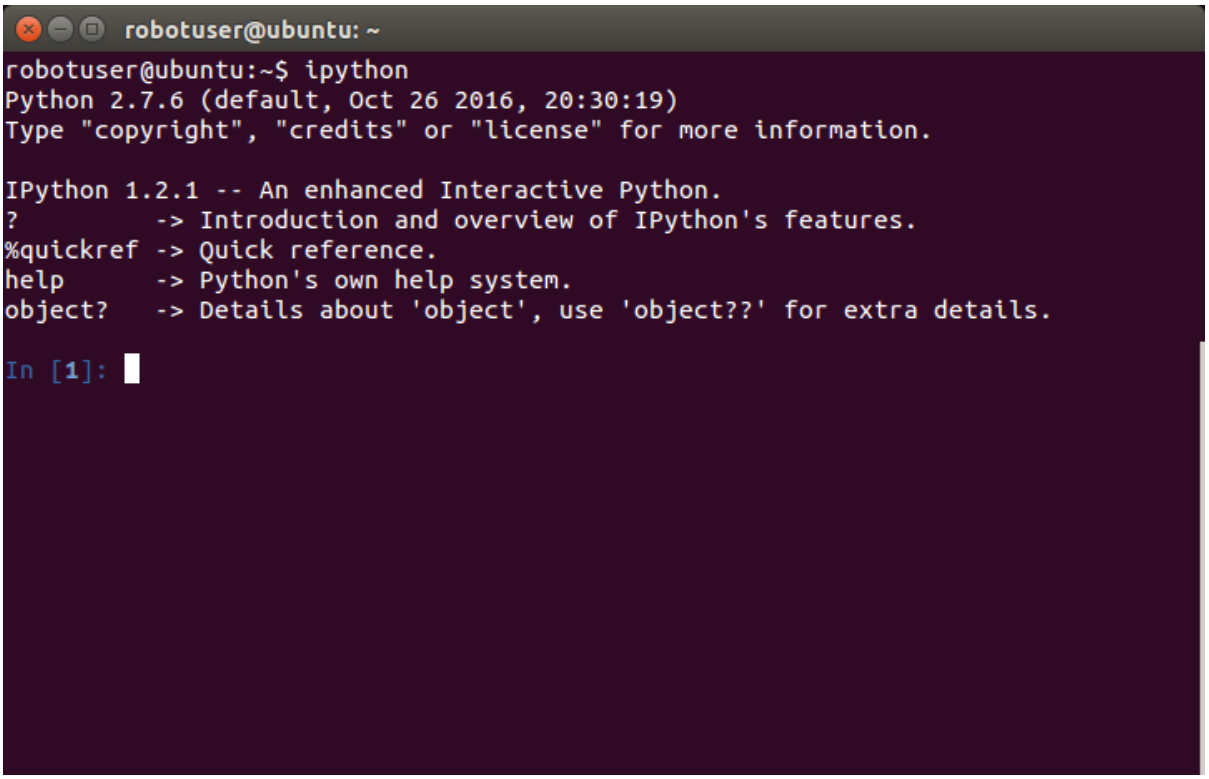
Python プログラムの実行には次の 2 通りの方法があります．

- Python コンソールを使って 1 行から数行のプログラムを入力・実行
- プログラム文を書いたテキストファイルを保存してそれを実行

本章では Python コンソールを使ってプログラムを 1 行ずつ入力・実行することで，Python がどのように実行されるのかを見てみます．

7.2.1 IPython の起動

Python コンソールとして IPython を使います．ターミナル上 `ipython` （全て小文字）と入力して `Enter` キーを押すと IPython コンソールが起動します．



- メモ : IPython がインストールされていない場合はインストールしてください.
  - 方法-1

```
$ pip install ipython
```

  - 方法-2

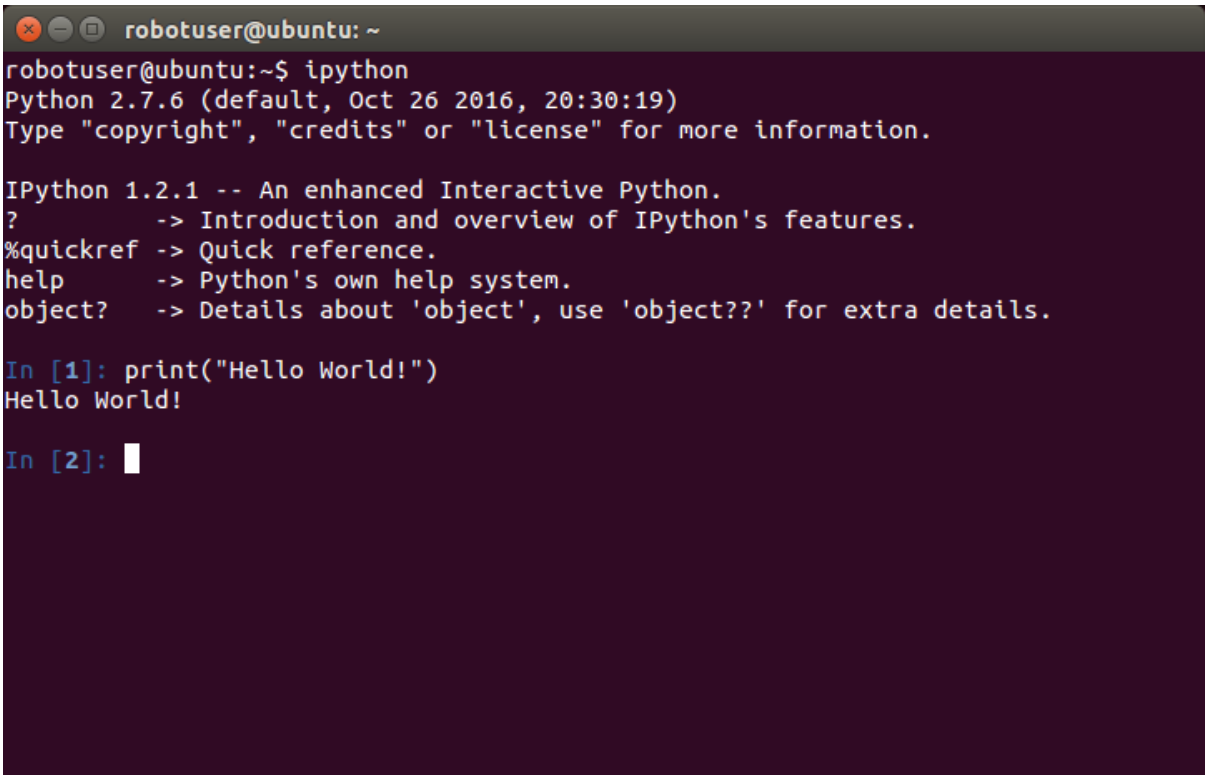
```
$ sudo apt-get update
$ sudo apt-get dist-upgrade
$ suoo apt-get install ipython
```

    * 注意: 方法-2 には Ubuntu PC の管理者パスワードが必要

7.2.2 命令文の入力と実行

IPython コンソールに次の行の `print` 以降を入力して `Enter` キーを押します.

```
In [1]: print("Hello World!")
```




```
In [1]: print("Hello World!")
Hello World!

In [2]:
```

`Hello World!` と出力されたと思います.

7.2.3 計算

Python で数の計算をしてみます. `1 + 1` と入力して `Enter` キーを押します.

```
In [2]: 1 + 1
Out[2]: 2
```

掛け算です.

```
In [3]: 2 * 3
Out[3]: 6
```

割り算です. Python 3.x での結果です.

```
In [4]: 3 / 2
Out[4]: 1.5
```

Python 2.x では整数の割り算の結果が Python 3.x とは異なります.

```
In [4]: 3 / 2
Out[4]: 1
```

Python 2.x では整数の割り算は `3 / 2` 答えが `1` （余りが `1` ）と出力されます. Python 2.x で小数点以下も記述して再び計算してみます.

```
In [5]: 3.0 / 2.0
Out[5]: 1.5
```

Python 2.x の計算では整数（整数型）を扱っているのか，小数点を含んだ数（浮動小数点型）を扱っているのかを意識する必要があります．

7.2.4 変数

次はプログラムらしく「変数」を使って計算してみます．変数へ値を入れるのに = を使って「代入」します．

```
In [6]: a = 4
In [7]: b = 5
In [8]: a * b
Out[8]: 20
```

```
In [9]: x = 4.0
In [10]: y = 5.0
In [11]: z = x * y
In [12]: print(z)
20.0
```

7.2.5 リスト

変数は 1 つの値しか扱いませんが多くの数値をまとまって扱いたい場合に利用するものに「リスト」や「マトリクス」などがあります．

リストは `[ ]` で定義します．注意が必要なのはリストの 1 つ目の入れ物の番号（インデックス）が `1` ではなく `0` であることです．

```
In [41]: list = [ 10, 20, 30, 40, 50 ]
In [42]: list[0]
Out[42]: 10
In [43]: list[1]
Out[43]: 20
In [44]: list[2]
Out[44]: 30
In [45]: list[5]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-5-48bc095b4faf> in <module>()
----> 1 list[5]
IndexError: list index out of range
In [46]: list[4]
Out[46]: 50
In [47]: list[-1]
Out[47]: 50
In [48]: print(list)
[10, 20, 30, 40, 50]
```

上記のリストの例では `list` に 5 つ値のあるリストを定義しました．インデックスが `0` から始まるので 5 つ目のインデックスは `4` になります．インデックス `5` の入れ物はこのリストには存在しないので `list[5]` を実行すると `IndexError` が返ってきます．この例の場合は最後に格納されている値を読むには `list[4]` または `list[-1]` とすれば値 `50` を読むことができます．

リストのリストがマトリクスになります．

```
In [51]: matrix = [ [ 1, 2, 3, 4, 5 ], [ 6, 7, 8, 9, 10 ], [ 11, 12, 13, 14, 15] ]
In [52]: matrix[0]
Out[52]: [1, 2, 3, 4, 5]
In [53]: matrix[1]
Out[53]: [6, 7, 8, 9, 10]
In [54]: matrix[2]
Out[54]: [11, 12, 13, 14, 15]
```

7.2.6 文字列

文字列も少し扱ってみます.

```
In [61]: p = 'robot '
In [62]: q = 'programmer'
In [63]: print(p)
robot
In [64]: print(q)
programmer
In [65]: r = p + q
In [66]: print(r)
robot programmer
In [67]: r = p * 3 + q
In [68]: print(r)
robot robot robot programmer
```

7.3 Python プログラミング

7.3.1 比較と真偽値

IPython 上で数値を比較してみます. ロボットでは状態を数値で見て基準値や制限値などとの比較をしたりします.

比較が正しければ `True` を正しくなければ `False` の真偽値（ブーリアン）を返してきます.

```
In [21]: c = 40
In [22]: d = 50
In [23]: c < d
Out[23]: True
In [24]: c > d
Out[24]: False
In [25]: c == d
Out[25]: False
In [26]: c != d
Out[26]: True
```

```
In [27]: e = c < d
In [28]: f = c > d
In [29]: g = c == d
In [30]: h = c != d
In [31]: print( e, f, g, h )
(True, False, False, True)
```

`g = c == d` は少しややこしいですが= と == の違いを知ってしまえば何のことはありません. = は「代入」でした.「代入演算子」と呼ばれています. == は「比較」をしています.「比較演算子」の「等しい」です. つまり `c == d` の比較結果 `True` もしくは `False` を `g` に代入しているわけです.

!= は比較演算子の 1 つで「等しくない」を意味します. `c != d` では `c` と `d` がこの場合 `40` と `50` で「等しくない」が「正しい」ので `True` が返ってきます.

7.3.2 関数

プログラムで非常に便利なのが「関数」です. 関数は何か値を渡すと決まった処理をした値を返してくるものです.

Python に限らずプログラミングにおける関数の最も大きな利点に次の 2 つが挙げられます.

- 同じ処理を何度もプログラム上に記述しなくて済む（関数を呼び出すだけで済む）

- 各処理機能をそれぞれのまとまりにするのでプログラムが読みやすくなる

簡単な関数を定義してみます．ここでは数学の関数 $y = x2$ を Python の関数で定義します．

```
In [71]: def square_number(x):     # Ctrl + Enter
    ...:     return x ** 2         # Ctrl + Enter
    ...:                           # Enter
```

`def square_number(x):` と入力してから `Ctrl` キーを押しながら `Enter` を押します．`Ctrl` + `Enter` キーでプログラムを実行せずに改行がされます．次に `return x ** 2` と入力して `Ctrl` + `Enter` します．そして `Enter` を押して関数の定義は終了です．

- **注意** : Python ではインデントでプログラムのまとまり（**ブロック**）を表現します．本例では `return x ** 2` が `def square_number(x):` の中身になるので `return x ** 2` のインデントを 1 つ下げています．関数の中だけではなく後ほど出てくる条件文でもインデントを用いたブロックで内外を区別します．

ここで定義した関数の内容をまとめると次のようになります．

- 関数名 : `square_number`
- 引数　 : `x`（入力値を入れる変数）
- 戻り値 : `x ** 2`（x の 2 乗）

それでは定義した関数を使ってみます．

```
In [72]: y = square_number(2)
In [73]: print(y)
4

In [74]: z = square_number(5)
In [75]: print(z)
25
```

関数 `square_number()` に与えた値の 2 乗の値が返ってくることと思います．

まず例として数値の処理を関数で定義してみましたが関数ができることは数値の処理だけではありません．文字列の操作などプログラムで書ける処理であれば何でもできます．

```
In [81]: def print_words( p, q ):
    ...:     r = p + ' ' + q
    ...:     print(r)
    ...:     return
    ...:

In [82]: a = "robot"
In [83]: b = "programmer"

In [84]: print_words( a, b )
robot programmer

In [85]: print_words( b, a )
programmer robot
```

- **注意** : 入力時に `def print_words( p, q ):` 内のインデントが揃うように気をつけてください．

7.3.3 クラス

ここまでの Python チュートリアルでプログラム命令を使って数値や文字列，変数，リスト，関数の操作を行いました．それらの変数や関数をまとめて 1 つの機能のまとまりにしたのが「クラス」です．

Python や ROS のライブラリの多くは「クラス」の形式で提供されています．

Python におけるクラスの基本的な構造は次のような構成になっています．

```
class MyClass:

    def __init__( self, x = 3.0, y = 2.0, name = 'Name' ):
        self.x = x
        self.y = y
        self.name = name

    def function( self ):
        return self.x + self.y

    def print_result( self ):
        print( '%8s - x: %5.2f  y: %5.2f  => Result: %5.2f'
              % ( self.name, self.x, self.y, self.function() ) )
```

クラス内で定義された関数はそのクラスの「メソッド」と呼びます．また `__init__()` はクラスの初期化を行っている「コンストラクタ」です．

クラスを利用する側では次のようにクラスを「インスタンス」化して使用します．

```
    a = MyClass()
    b = MyClass( 5.0, 4.0, 'B' )

    a.print_result()

    a.x = 6.0
    a.y = 7.0
    a.name = 'A'
    a.print_result()

    b.print_result()
```

上記の例では `a` や `b` が「`MyClass` クラスのインスタンス」と呼ばれます．

IPython で実行すると次のようになります．

- **注意**：入力時にインデントに気をつけてください．

```
In [1]: class MyClass:
   ...:     def __init__( self, x = 3.0, y = 2.0, name = 'Name' ):
   ...:         self.x = x
   ...:         self.y = y
   ...:         self.name = name
   ...:     def function( self ):
   ...:         return self.x + self.y
   ...:     def print_result( self ):
   ...:         print( '%8s - x: %5.2f  y: %5.2f  => Result: %5.2f'
   ...:               % ( self.name, self.x, self.y, self.function() ) )
   ...:

In [2]: a = MyClass()
In [3]: b = MyClass( 5.0, 4.0, 'B' )

In [4]: a.print_result()
    Name - x:  3.00  y:  2.00  => Result:  5.00

In [5]: a.x = 6.0
In [6]: a.y = 7.0
In [7]: a.name = 'A'
In [8]: a.print_result()
       A - x:  6.00  y:  7.00  => Result: 13.00

In [9]: b.print_result()
       B - x:  5.00  y:  4.00  => Result:  9.00
```

＜参考＞ 同じことを Python プログラムファイルに書くと次のようになります.

pyclass_example.py

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

class MyClass:

    def __init__( self, x = 3.0, y = 2.0, name = 'Name' ):
        self.x = x
        self.y = y
        self.name = name

    def function( self ):
        return self.x + self.y

    def print_result( self ):
        print( '%8s - x: %5.2f  y: %5.2f  => Result: %5.2f'
              % ( self.name, self.x, self.y, self.function() ) )


def main():

    a = MyClass()
    b = MyClass( 5.0, 4.0, 'B' )

    a.print_result()

    a.x = 6.0
    a.y = 7.0
    a.name = 'A'
    a.print_result()

    b.print_result()


if __name__ == '__main__':

    main()
```

Python プログラム（Python スクリプト）ファイルの実行方法には次の方法があります. いずれもターミナルから入力して実行します.

- python に続けてファイル名を入力

```
$ python pyclass_example.py
```

- Python スクリプトファイルに実行権限を付与（後の章で詳述）して ./ファイル名 で実行

```
$ ./pyclass_example.py
```

pyclass_example.py の実行結果です.

```
$ python pyclass_example.py
    Name - x:  3.00  y:  2.00  => Result:  5.00
       A - x:  6.00  y:  7.00  => Result: 13.00
       B - x:  5.00  y:  4.00  => Result:  9.00
```

7.3.4 制御フローツール

while 文

条件判断により繰り返し実行に使われるのが while 文です．while 文の基本的な構造は次のようになっています．

```
while 条件:
    処理
```

条件 が True である限り 処理 を繰り返します．

次の例では i が 10 未満である限り while の中を繰り返します．while の中では 1 回実行される毎に i に 1 ずつ足され i が 10 になると次の while の条件判断で i < 10 が False になるのでループを抜けます．

py_while.py

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

i = 0
while i < 10:
    print( i )
    i += 1

print( "while loop ended at i=%d" % i )
```

py_while.py の実行結果

```
$ python py_while.py
0
1
2
3
4
5
6
7
8
9
while loop ended at i=10
```

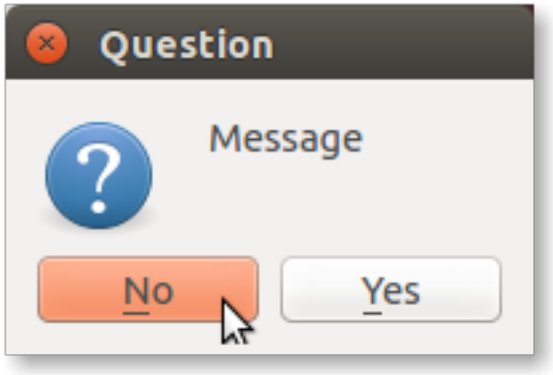
if 文

条件分岐などに使われるのが if 文です．if 文の基本的な構造は次のようになっています．

```
if 条件 1:
    処理 1
elif 条件 2:
    処理 2
else:
    処理 3
```

もし 条件 1 が True なら 処理 1 を実行し条件 1 が False で 条件 2 が True なら 処理 2 を実行し条件 1，条件 2 ともに False であれば 処理 3 が実行されます．

次の例では [No] [Yes] の選択をするウィンドウを表示します．



[Yes] ボタンが押された場合は `if` 内の条件評価が `True` になるので関数 `qestion_yn()` が `True` を返します．それ以外は `else` 内の処理が実行されて `qestion_yn()` が `False` を返します．

pyqt_question.py

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

import sys
from PyQt4 import QtGui

def question_yn( qmsg='Message', title='Question' ):

    msgbox = QtGui.QMessageBox()
    result = msgbox.question( msgbox, title, qmsg, msgbox.Yes | msgbox.No, msgbox.
↪No )

    if result == msgbox.Yes:
        return True
    else:
        return False


if __name__ == '__main__':

    app = QtGui.QApplication(sys.argv)

    print( question_yn() )
    print( question_yn("No/Yes") )
    print( question_yn("Question", "ROS Question") )
```

pyqt_quesiton.py の実行結果例（ Yes / No の選択による ）

```
$ python pyqt_question.py
False
True
False
```

for 文

特定の回数の繰り返し実行に用いられるのが for 文です．Python の for 文はリストまたは文字列の任意のシーケンス型にわたって反復処理を行います．

```
for 要素 in シーケンス型:
    処理
```

シーケンス型 の各 要素 について 処理 を行います．

py_for.py

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

num_list = [ 10.0, 20.0, 30.0, 40.0, 50.0 ]

for x in num_list:
    print( x )

for i in range(5):
    print( i )

for n in range( len(num_list) ):
    print( "{0:2d} : {1:5.1f}".format(n, num_list[n]) )
```

py_for.py の実行結果

```
$ ./py_for.py
10.0
20.0
30.0
40.0
50.0
0
1
2
3
4
 0 : 10.00
 1 : 20.00
 2 : 30.00
 3 : 40.00
 4 : 50.00
```

7.3.5 コメント文

プログラミング言語一般にコメントアウト部分は実行が行われません．実行されないものをプログラムに書く理由はいくつかあります．

- 命令文やクラス，関数，変数などの説明
- 変更や修正などのときに既存の文をコメントとして仮に残す
- …

他の人がプログラムを見ても分かる，また後々に自分がプログラムを見返すときも分かるように説明文を入れます．

1 行のコメントアウト

Python で 1 行だけコメントアウトする場合は # を書けば# 以降，行末までがコメント文となります．

```
# 行全体のコメント
print( "Comment Test" )     # 行の途中から行末までのコメント
# print( "Done" )
```

実行結果

```
Comment Test
```

複数行のコメントアウト

Python で複数行コメントアウトする場合は 3 つのシングルクォーテーション ''' もしくは 3 つのダブルクォーテーション """ で囲みます．ただし，周りのインデントと揃える必要があります．

```
for i in range(5):
    '''
    複数行のコメント文
    print( "Comment Test" ) を試します．この行はコメント内で実行されません．
    '''
    print( "Comment Test" )
```

実行結果

```
Comment Test
Comment Test
Comment Test
Comment Test
Comment Test
```

CHAPTER

EIGHT

実機の使い方

実機と MoveIt! を使ってプログラムからロボットを操作する際の実機の使い方を説明します.

8.1 myCobot280 の場合

8.1.1 myCobot280 の固定

myCobot280 がぐらつかないように，固定をしてください.

- 土台となる板を用意し，myCobot をネジで固定.
- 土台となる板自体がぐらつかないように，机と固定．または，大きな板を使用してください.

8.1.2 myCobot280 のファームウェア更新

myStudio のダウンロード

Windows PC が必要です.

MyStudio の リリースページにアクセスし，最新の exe ファイルをダウンロード・実行してください.

USB ドライバのインストール

SILICON LABS CP210x USB - UART ブリッジ VCP ドライバにアクセスし，CP210x VCP Windows をダウンロードしてください.

ファームウェアの更新

アーム先端の M5Stack Atom と胴体の M5Stack Basic のファームウェアを更新します.

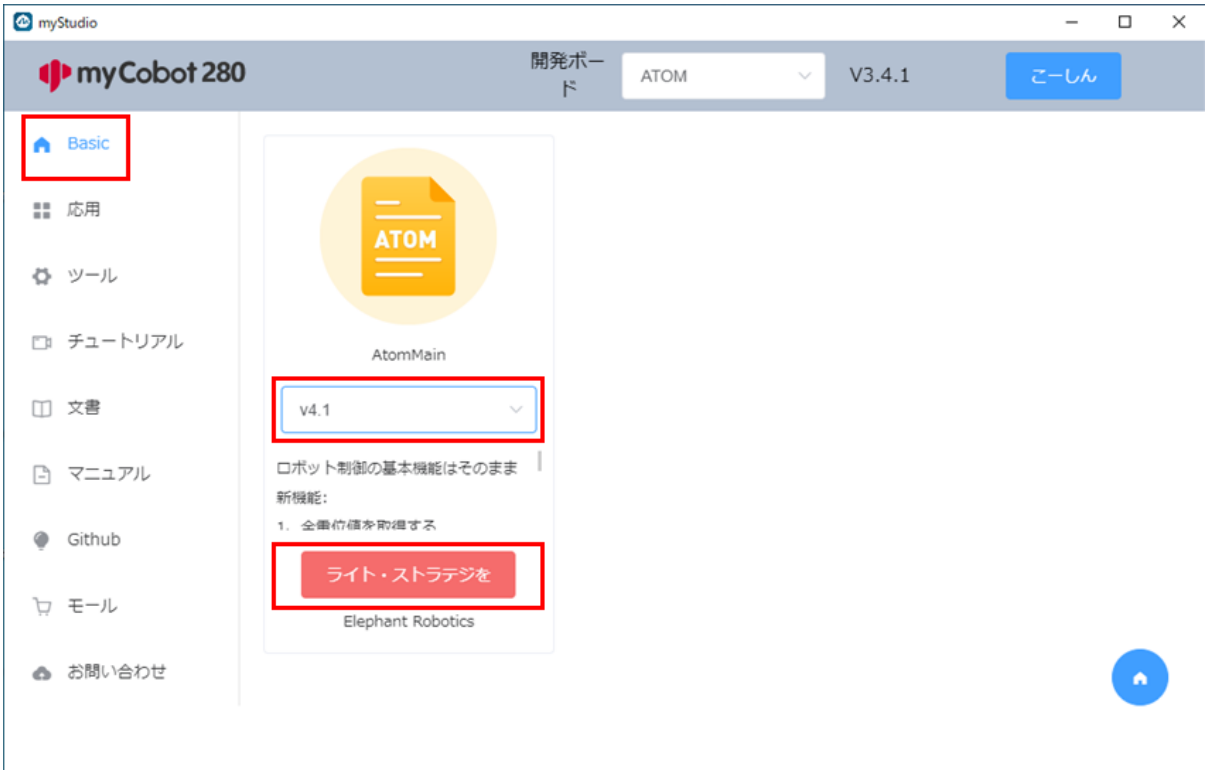
M5Stack Atom のファームウェア更新

今回は，AtomMain の v4.1 をインストールします.

MyStudio を立ち上げた後，アーム先端の Atom とパソコンを接続してください.

MyStudio の画面で USB Port の欄が ATOM になったはずです.

その状態で接続した後，Basic タブをクリックし，AtomMain の v4.1 を選択してインストールします.



M5Stack Basic のファームウェア更新

今回は，minirobot の v1.0 をインストールします.

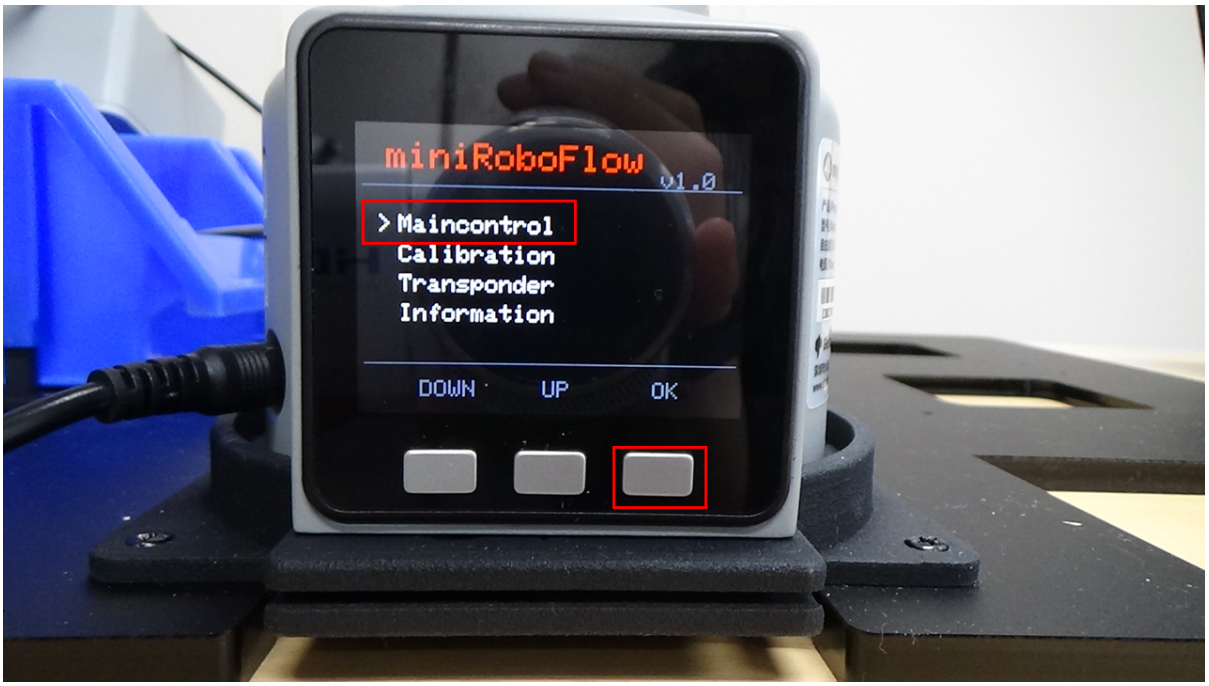
MyStudio を立ち上げた後，胴体の Basic とパソコンを接続してください.

MyStudio の画面で USB Port の欄が BASIC になったはずです.

その状態で接続した後，Basic タブをクリックし，minirobot の v1.0 を選択してインストールします．

動作確認（ティーチングデモ）

Basic とパソコンを接続し，myCobot の画面 (miniroboFlow) から Maincontrol を選択してください．



動作のティーチングを行うために，Record を選択してください．

Recording to Ram/Flash? と聞かれるので，Ram を選択してください．

同様に，Play で教えた動作を再生します.

8.1.3 myCobot280 を Transponder モードにする

以降，パソコンから myCobot に指令を送る際には，myCobot の画面 (miniroboFlow) から Transponder を選択してください.

8.1.4 dialout グループへのユーザ追加

シリアルポートにアクセス権を持つ，dialout グループにユーザを追加します.

ターミナル

```
sudo adduser $USER dialout
```

8.1.5 pymycobot (Python API) のインストール

myCobot を Python で動かすための API をインストールします.

ターミナル

```
pip install pymycobot --user
```

動作確認

```
In [1]: from pymycobot.mycobot import MyCobot
In [2]: mycobot=MyCobot('/dev/ttyUSB0')
```

```
In [3]: mycobot.set_color(0,0,255)
```

```
In [4]: mycobot.set_color(0,255,255)
```

```
In [5]: from pymycobot.genre import Angle
In [6]: mycobot.send_angles([0,0,0,0,0,0], 80)
```

Creative Commons License

3. 可能な限りにおいて、本ライセンサーは、ライセンス対象権利の行使の対価としてロイヤリティを受け取る権利を放棄し、当該受け取りについて、直接的なものであるか、あるいは任意の、もしくは放棄可能な、法定の、もしくは強制的なライセンス制度に基づく徴収団体を介するものであるかを問わない。ライセンス対象資料が、非営利目的以外で使用された場合を含め、その他すべての場合において、本ライセンサーは、当該ロイヤリティを受け取る権利を明確に留保する。

第 3 条 – ライセンス条件

ライセンス対象権利のあなたの行使には、以下の条件が明確に適用される。

a. クレジット表示.

1. あなたがライセンス対象資料（変更された形式によるものを含む）を共有する場合、あなたは、以下の規定に従わなければならない。

A. 本ライセンサーがライセンス対象資料とともに以下のものを提供した場合、それを維持すること。

A. 本ライセンサーにより求められた合理的な方法による、ライセンス対象資料の創作者およびクレジット表示を受けるために指定されたその他のものの識別（ペンネームを指定された場合、ペンネームによる場合を含む）。

B. 著作権表示。

C. このパブリック・ライセンスに言及する表示。

D. 保証の否認に言及する表示。

E. 合理的に可能な範囲で、ライセンス対象資料への URI またはハイパーリンク。

B. あなたがライセンス対象資料を改変したか否かを示し、また過去の改変がある場合、その表示を維持すること。および、

C. ライセンス対象資料がこのパブリック・ライセンスに基づきライセンスされたものであることを示し、またこのパブリック・ライセンスの文章またはこのパブリック・ライセンスへの URI またはハイパーリンクを盛り込むこと。

2. あなたは、あなたがその中でライセンス対象資料を共有するメディア、手段および状況に基づく合理的な方法で条項 3(a)(1) の条件を満たすことができる。たとえば、必要な情報を含む情報源への URI またはハイパーリンクを提供することにより条件を満たすことが合理的な場合がある。

3. 本ライセンサーにより求められた場合、あなたは、合理的に可能な範囲で、条項 3(a)(1)(A) により求められるいずれかの情報を削除しなければならない。

4. あなたが、自己が作成した翻案資料を共有する場合、あなたが適用する翻案者のライセンスにより、翻案資料の受領者によるこのパブリック・ライセンスの順守を妨げてはならない。

b. 継承. 条項 3(a) の条件に加えて、あなたが、自己が作成した翻案資料を共有する場合、以下の条件も適用される。

1. あなたが適用する翻案者のライセンスは、ライセンス要素が同じであるクリエイティブ・コモンズのライセンス（このバージョンもしくはその後のバージョン）または BY-NC-SA 互換ライセンスでなければならない。

2. あなたは、自己が適用する翻案者のライセンスの文章、または当該翻案者のライセンスへの URI またはハイパーリンクを盛り込まなければならない。あなたは、あなたがその中で翻案資料を共有するメディア、手段および状況に基づく合理的な方法でこの条件を満たすことができる。

3. あなたは、翻案資料について、自己が適用する翻案者のライセンスに基づき許諾された権利の行使を制限する、追加的な条件や異なる条件を申込むこと、もしくはそうした条件を課すこと、またはそうした効果的な技術的手段を適用することはできない。

第 4 条 – 独自のデータベース権

ライセンス対象権利に、独自のデータベース権が含まれ、ライセンス対象資料のあなたの使用に適用される場合、以下の規定に従う。

a. 誤解を避けるための確認事項として、あなたは条項 2(a)(1) により、データベースの内容の全部または実質的な部分について、非営利目的に限定して抜粋、再使用、複製および共有を行う権利を許諾される。

b. あなたが、データベースの内容の全部または実質的な部分を、自己が独自のデータベース権を有するデータベースに盛り込む場合、あなたが独自のデータベース権を有するデータベースは（その個々の内容を除く）、条項 3(b) に関する場合を含め、翻案資料である。および、

c. あなたがデータベースの内容の全部または実質的な部分を共有する場合、あなたは、条項 3(a) の条件を順守しなければならない。誤解を避けるための確認事項として、ライセンス対象権利にその他の著作権および類似の権利が含まれる場合、本第 4 条は、このパブリック・ライセンスに基づくあなたの義務を補足するものであり、その義務に取って代わるものではない。誤解を避けるための確認事項として、ライセンス対象権利にその他の著作権および類似の権利が含まれる場合、本第 4 条は、このパブリック・ライセンスに基づくあなたの義務を補足するものであり、その義務に取って代わるものではない。

第 5 条 – 保証の否認および責任の制限

a. 本ライセンサーが別段の保証を行った場合を除き、可能な限りにおいて、本ライセンサーは、無保証で、かつ提供可能な範囲で、ライセンス対象資料を提供し、明示的、黙示的、法定またはその他のいずれのものであるかを問わず、ライセンス対象資料に関していかなる表明も保証も行わない。これには、所有権、商品性、特定の目的に対する適合性、権利不侵害、隠れた欠陥などの欠陥の不存在、正確性、またはエラー（既知のものであるか否か、または発見可能なものであるか否かを問わない）の有無に関する保証などを含む。保証の否認が、全部または一部、認められない場合、あなたに対してこの否認を適用することはできない。

b. 可能な限りにおいて、いかなる場合においても、本ライセンサーはあなたに対して、このパブリック・ライセンスまたはライセンス対象資料の使用に起因する、直接的、特別、間接的、付随的、派生的、懲罰的、制裁的またはその他の、損失、費用、経費または損害賠償について、責任の根拠が法的理論（過失などを含む）であるか否かを問わず、責任を負わない。このことは、本ライセンサーが、上記の損失、費用、経費または損害賠償の可能性について知らされていた場合も同様とする。責任の制限が、全部または一部、認められない場合、あなたに対してこの制限を適用することはできない。上記の保証の否認と責任の制限は、可能な限りにおいて、すべての責任の無条件の否認および放棄に最も近い方法で解釈する。

第 6 条 – 期間および終了

a. このパブリック・ライセンスは、このパブリック・ライセンスによりライセンスされる著作権および類似の権利の期間中、適用される。ただし、あなたがこのパブリック・ライセンスを順守しない場合、このパブリック・ライセンスに基づくあなたの権利は、自動的に終了する。ライセンス対象資料を使用するあなたの権利が、条項 6(a) に基づき終了した場合においても、その権利は、以下の規定に従い復帰する。

b. あなたの違反の発覚から 30 日以内に違反が是正された場合、違反を是正した日の時点で自動的に復帰する。または、本ライセンサーによる明確な復帰次第、復帰する。誤解を避けるための確認事項として、このパブリック・ライセンスのあなたの違反について救済を求める権利であって、本ライセンサーが有する可能性がある権利は、本条項 6(b) によって左右されない。

c. 誤解を避けるための確認事項として、本ライセンサーは、別個の条件に基づきライセンス対象資料を提供し、または時期を問わずライセンス対象資料の配布を停止することもできる。ただし、そのことによりこのパブリック・ライセンスは終了しない。

d. 第 1 条、第 5 条、第 6 条、第 7 条および第 8 条は、このパブリック・ライセンスの終了後も存続する。

第 7 条 – その他の条件

a. 本ライセンサーは、あなたにより追加的な条件や異なる条件を伝えられた場合においても、明確に合意しない限り、そうした条件により拘束されない。

b. このパブリック・ライセンスに明記されていないライセンス対象資料に関する取り決め、了解または合意がある場合、それらは、このパブリック・ライセンスの条件とは別個の独立したものである。

第 8 条 – 解釈

a. 誤解を避けるための確認事項として、このパブリック・ライセンスは、このパブリック・ライセンスに基づく許可を得ることなく適法に行うことができるライセンス対象資料の使用について、その縮減、制限もしくは禁止を行い、またはその使用について条件を課すものではなく、またそのように解釈しないものとする。

b. 可能な限りにおいて、このパブリック・ライセンスのいずれかの規定が、強制不能と判断された場合、その規定を強制可能にするために必要な最小限の範囲で、その規定は自動的に改定される。その規定を改定することができない場合、残りの条件の強制可能性に影響を及ぼすことなく、その規定はこのパブリック・ライセンスから分離する。

c. 本ライセンサーが明確に合意した場合を除き、このパブリック・ライセンスのいずれの規定も放棄されず、またいずれの不順守についても承認されない。

d. このパブリック・ライセンスのいずれの規定も、本ライセンサーやあなたに適用される特権および免除（ある管轄や機関の法的手続きの特権と免除を含む）の制限や放棄ではなく、またそのように解釈することはできない。

クリエイティブ・コモンズは、自己のパブリック・ライセンスの当事者ではない。上記にかかわらず、クリエイティブ・コモンズは、自己が公表する資料について、自己のパブリック・ライセンスの 1 つを適用する選択を行うことが可能であり、その場合、「本ライセンサー」とみなされるクリエイティブ・コモンズ・パブリック・ライセンスに基づき、またはウェブサイト（creativecommons.org/policies）で公表されているクリエイティブ・コモンズの方針によるその他の許可に従い、資料が共有されていることを示すという限定的な目的を除き、クリエイティブ・コモンズは、「Creative Commons」という商標またはクリエイティブ・コモンズのその他の商標やロゴの使用について、クリエイティブ・コモンズの書面による事前の承認を得ない限り、その許可を行わない。上記の使用には、クリエイティブ・コモンズのいずれかのパブリック・ライセンスに対する、またはライセンス対象資料の使用に関するその他の取り決め、了解または合意に対する、無許可の修正に関連して使用する場合などを含む。誤解を避けるための確認事項として、この文章は、パブリック・ライセンスの一部ではない。

問合せ先：creativecommons.org