SEED Solutions
Smart RT System for Service Robots

NEDO講座
第４部
マニピュレーションの基礎

THK株式会社
産業機器統括本部 技術本部
事業開発統括部 永塚BU

第４部の概要

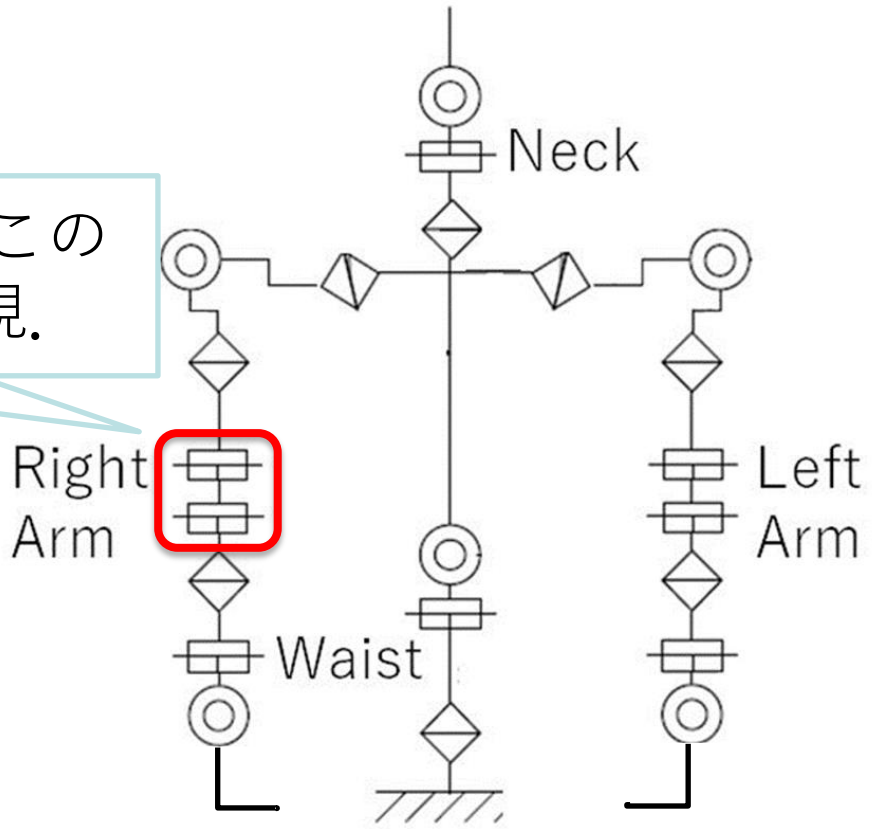
- SEED-Noidの上半身を駆使してマニピュレーションを行う手順を示す.
- 第４部では，習うより慣れろの観点から，演習を主体としています.

第４部をこなすことで，SEED-Noidの上半身を活用したマニピュレーションの手順を学べます.

SEED-Noidの関節構造の理解

SEED-Noidの上半身は以下の図に示すような関節構造を持つ

直動関節が2個並んだ形だが，この形で，仮想的に回転関節を実現.



実際に動かしながら，各関節について理解を深めましょう

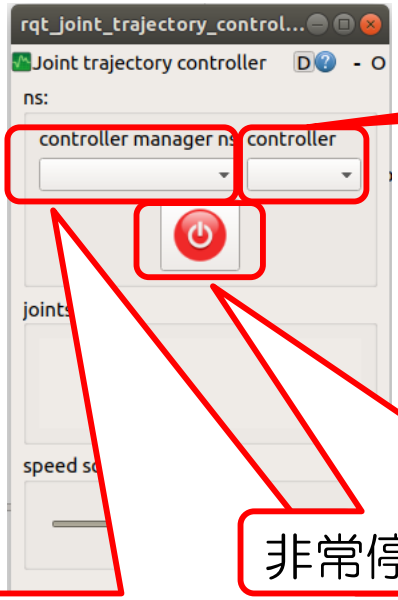
SEED-Noidの関節構造の理解

ターミナル1

```
$roslaunch seed_r7_bringup moveit.launch
```

ターミナル2

```
$rosrun rqt_joint_trajectory_controller rqt_joint_trajectory_controller
```



グループの選択．いろいろ選んで確認してください

下記のコントローラを用いることで，指定したグループの関節を１つづつコントロールできる．
このツールを用いて，関節角度ベースで，どこの関節がどのように動くかを確認してください．

非常停止ボタン．押すと解除

マネージャの選択．１つしか選べない

MoveIt!用のlaunchファイル

第４部では，RViz上でロボットを直感的に動かす．
RVizでは，ロボットの動作計画などを行うことができる．

動作計画のためのlaunchファイルの起動

```
$roslaunch seed_r7_bringup moveit.launch
```

SEED-Noidが表示され，MoveIt!も読み込まれた状態でRVizが起動する．

MoveIt!の基礎(1)

MoveIt!でマニピュレータの軌道を計画するためには，以下の３つのツールを用いて計画を行う．

手先のボール

ドラッグしながら動かすことで，手先の位置を決めることができる．

矢印

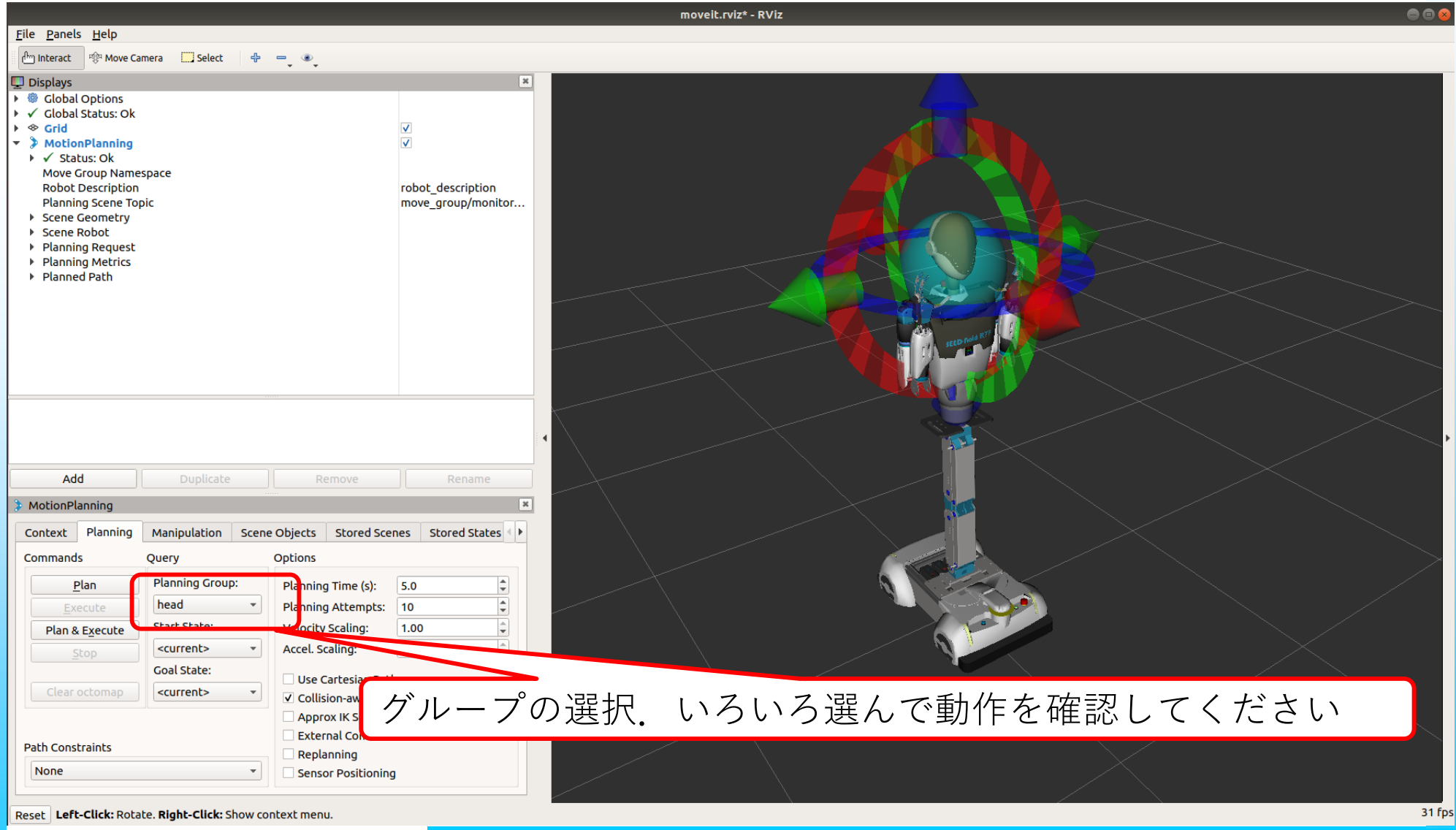
ドラッグしながら手先を並進移動できる(左の図の各軸移動の矢印を参照)．

円

手先をドラッグしながら手先を回転できる(左の図の各軸回転の円環を参照)．

MoveIt!の基礎(2)

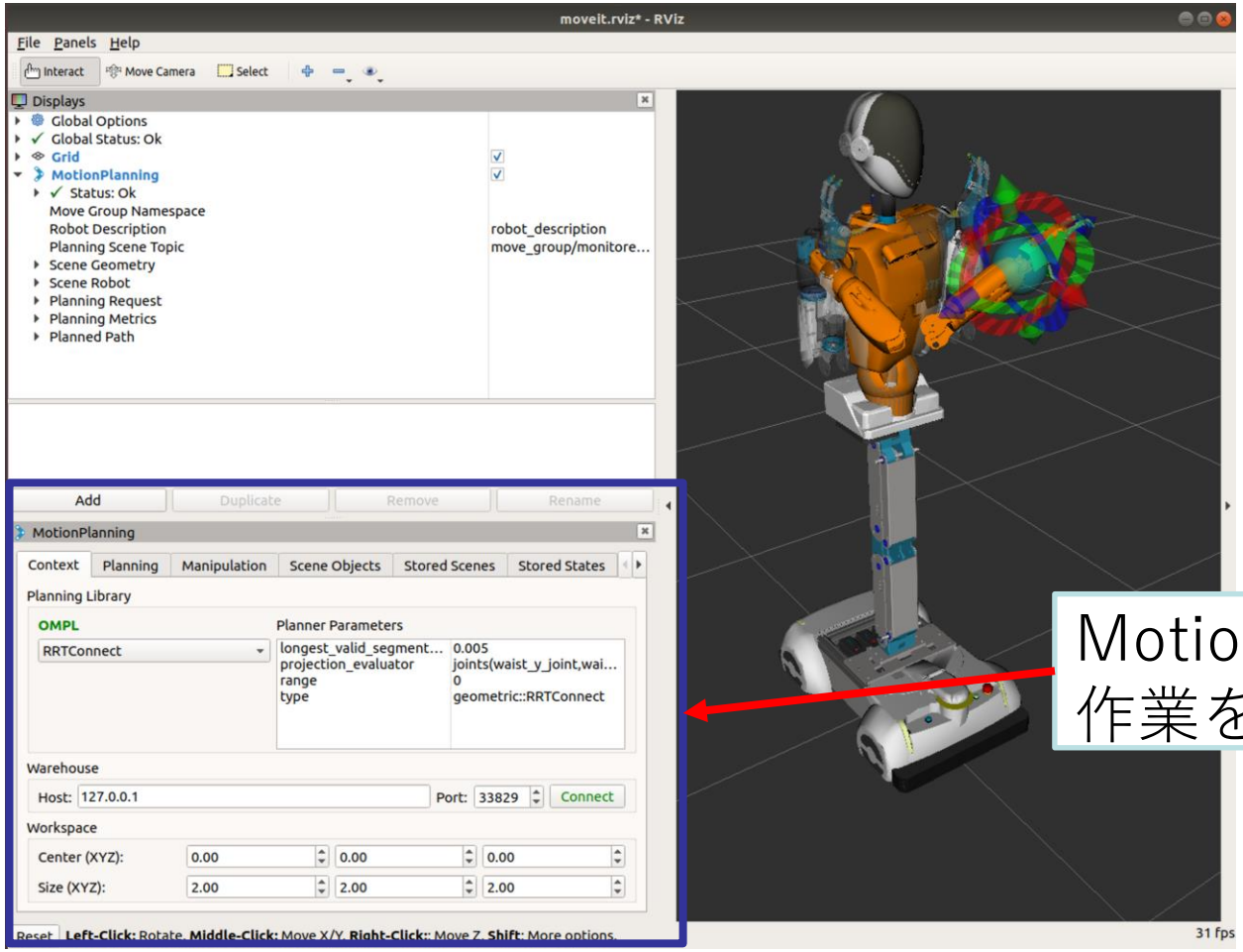
指定するグループとその対応を確認しながら，MoveIt!でのSEED-Noidの動きを確認しましょう．



MoveIt!の基礎

ボールや矢印による手先の操作は，単に動かしてみただけで，実際に軌道としては登録されていない．

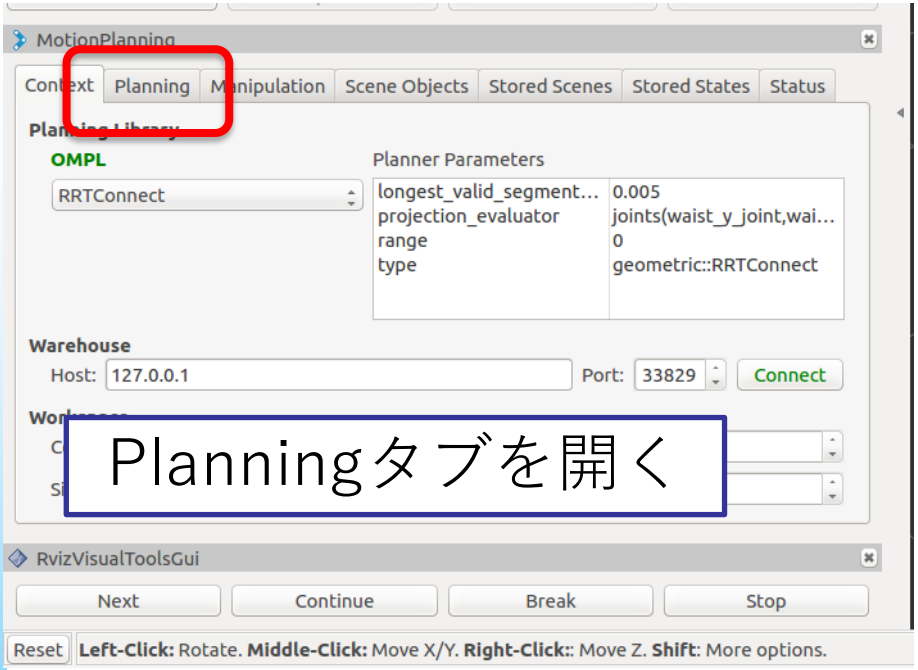
ここでは，軌道計画から実行までの方法を学ぶ．

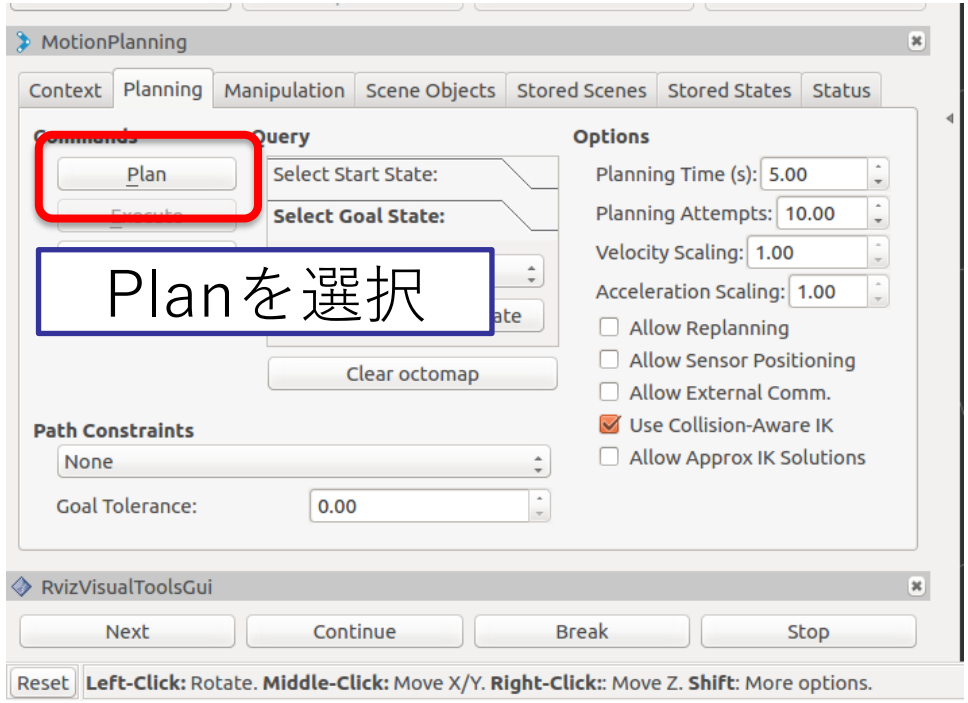


MotionPlanningウィンドウで作業をする．

MotionPlanningウィンドウでの作業

- 「Plan」では，軌道計画を行う．
- 「Execute」で計画した軌道を再生(実行)する．
  (実際にロボットが動作する．)

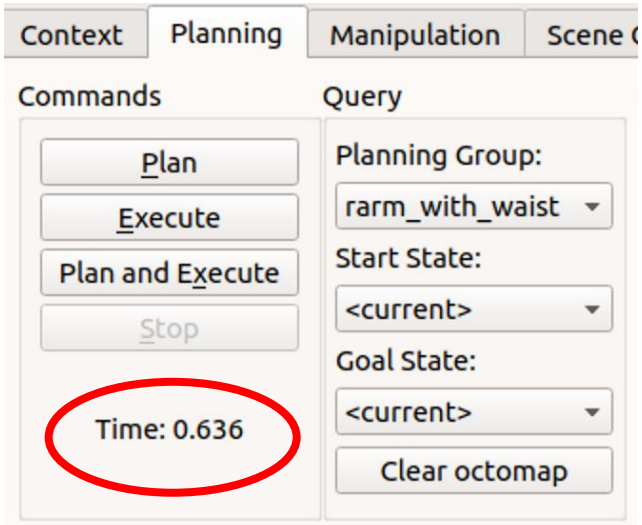




軌道を決めてから，「plan」を行うと，
現在位置から指定先まえの軌道計画が行われる．

MotionPlanningウィンドウでの作業

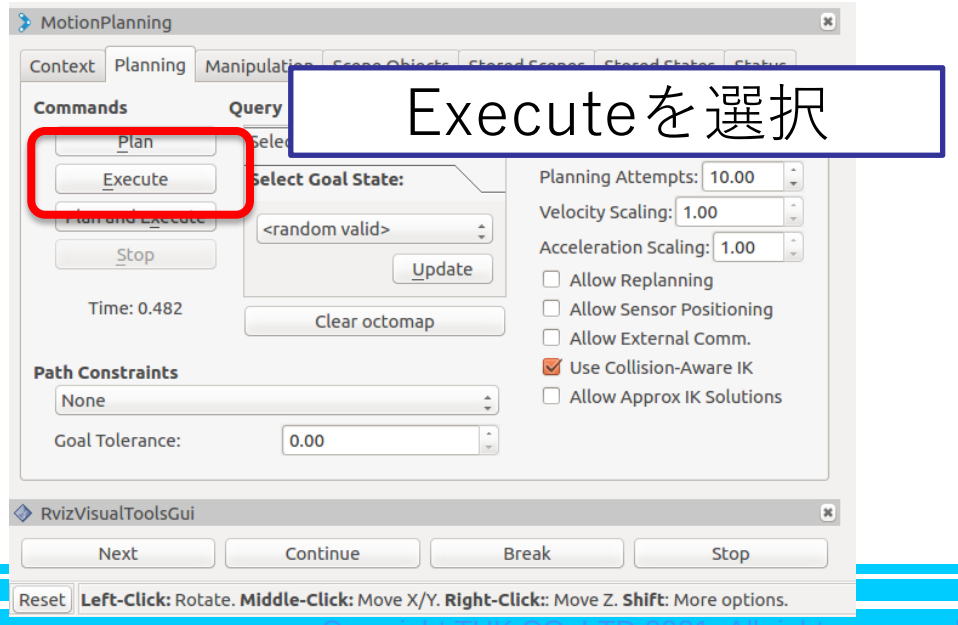
「Plan」が正常に行われると，動作時間が表示される．



動作時間が表示

正しく軌道が生成できない場合は，時間は表示されないので，目標位置や姿勢を変更する．

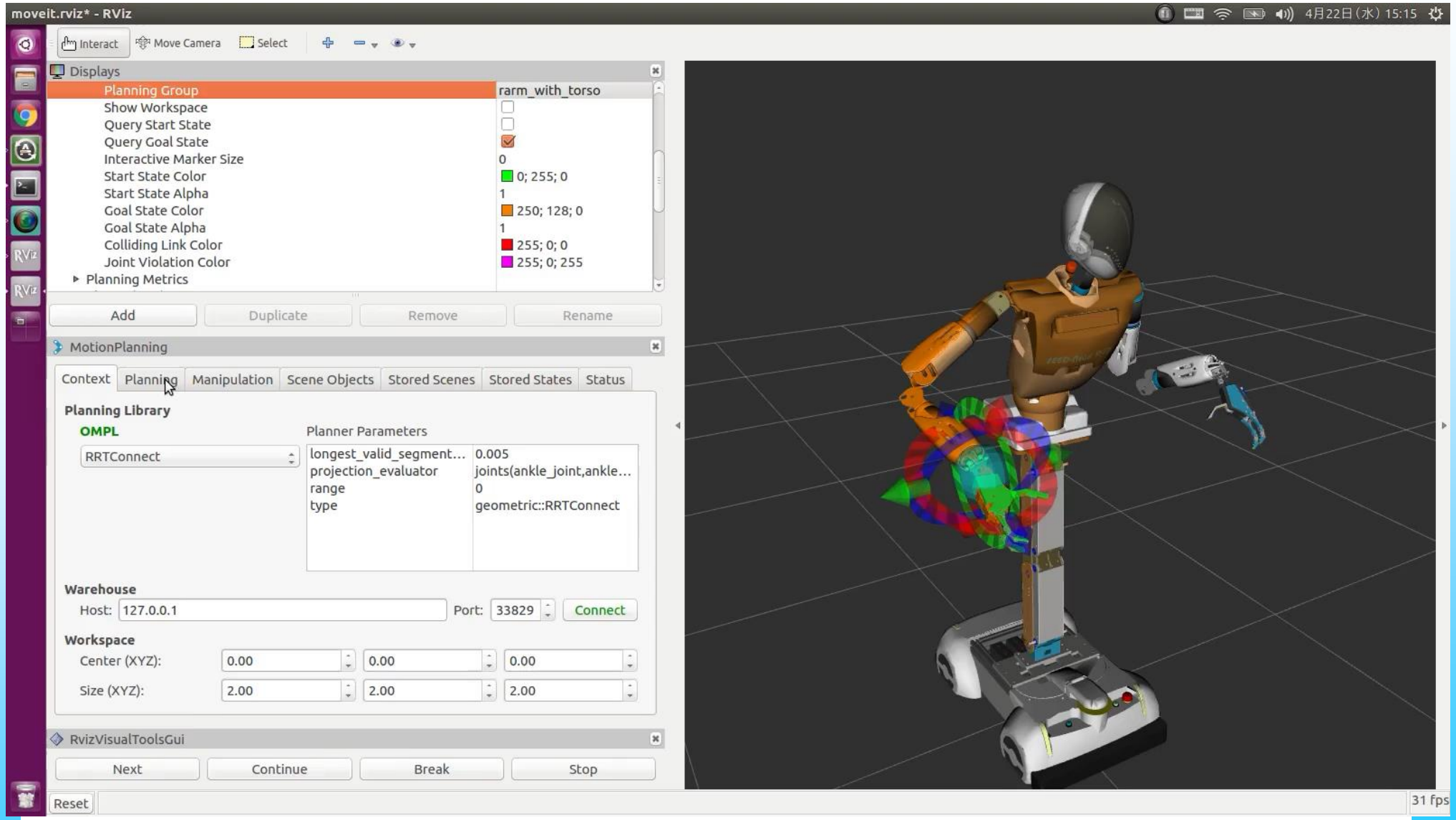
軌道計画が完了すると，アニメーションで動きが再生される．
（ただし，この段階では，実際の動作は行われていない．）



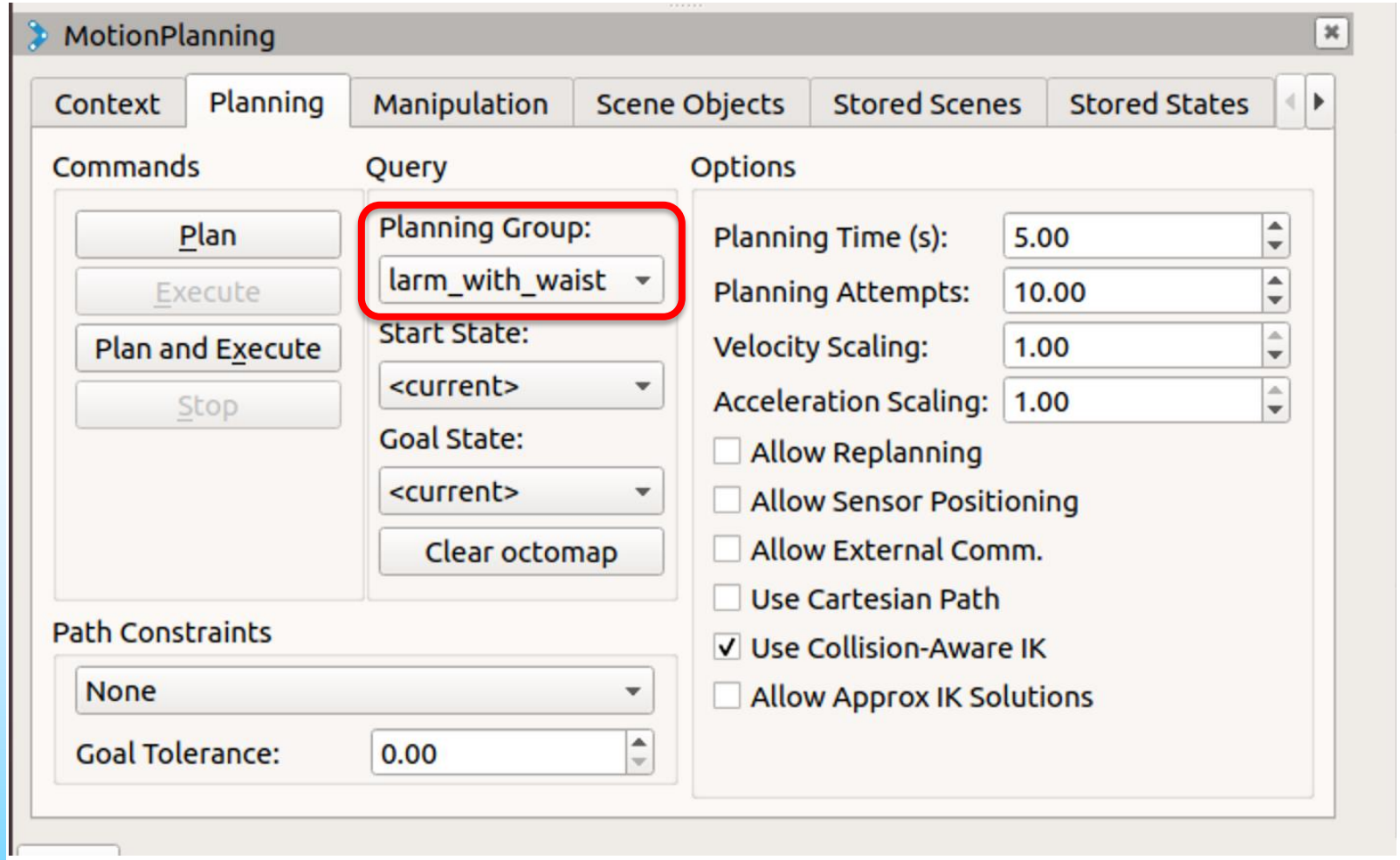
Executeを選択

「Execute」を選択すると，実際の動作が行われる．
（Rviz上でのロボットの姿勢にも反映される．）

MotionPlanningの例



MoveIt!におけるグループ設定



軌道計画するグループを変更して，グループ名と動きの対応や可動範囲などを確認する．

スクリプトを用いたモーション作成・再生

THK株式会社
産業機器統括本部 技術本部
事業開発統括部 永塚BU

Pythonノードでのモーション生成

MoveIt!のウィンドウから，簡単な動作計画はできるが，連続した計画，再生は難しい．

コンパイル型言語(C++やJAVA)などに比べて，Pythonなどのスクリプトは，手軽に実行・変更・検証ができるので，Pythonでモーション再生用のノードを作成する．

ベースとなるノードの作成

Pythonのノードは以下のディレクトリに作成する．

```
~/ros/melodic/src/seed_r7_ros_pkg/seed_r7_samples/scripts
```

以降，スクリプトの作成はPythonの作法に従いますので，基本的な部分については，適宜Pythonについて調べてください．

テスト用ノードの作成(1)

エディタで，下記のファイルを生成する．

```
$gedit moveit_tutorial1.py
```

以降，このファイルを書き換えながらいろいろなモーション生成を行う．

演習1

右に書かれているサンプルノードを作成し，動作検証せよ．

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

import rospy

##############################################
#Main Part
##############################################
def main():
  #Initialization of Node
  rospy.init_node("moveit_tutorial_node")

  print("MoveIt! tutorial node is created!!")

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

```

テスト用ノードの作成(2)

演習１のノードが作成できたら，動作検証を行う．

以下の様にノードに実行権限を与える．

```
$chmod +x moveit_tutorial1.py
```

以下の様にテスト用ノードを実行する．（ターミナルを2個起動）

ターミナル1

```
$roscore
```

ターミナル2

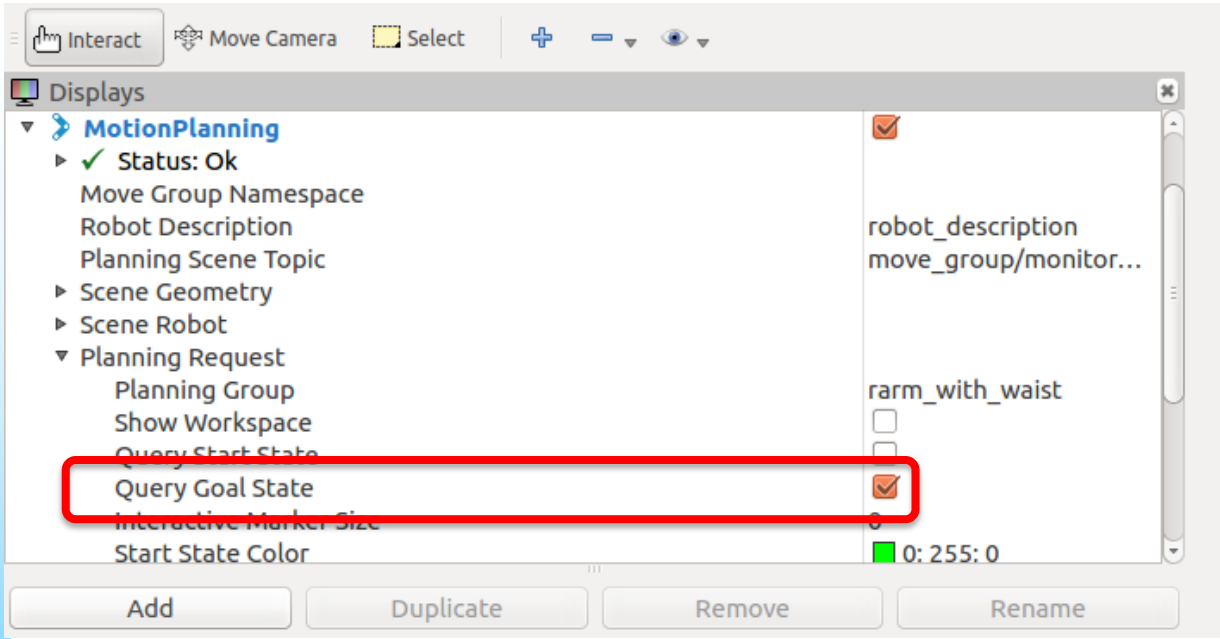
```
$rosrun seed_r7_samples moveit_tutorial1.py
```

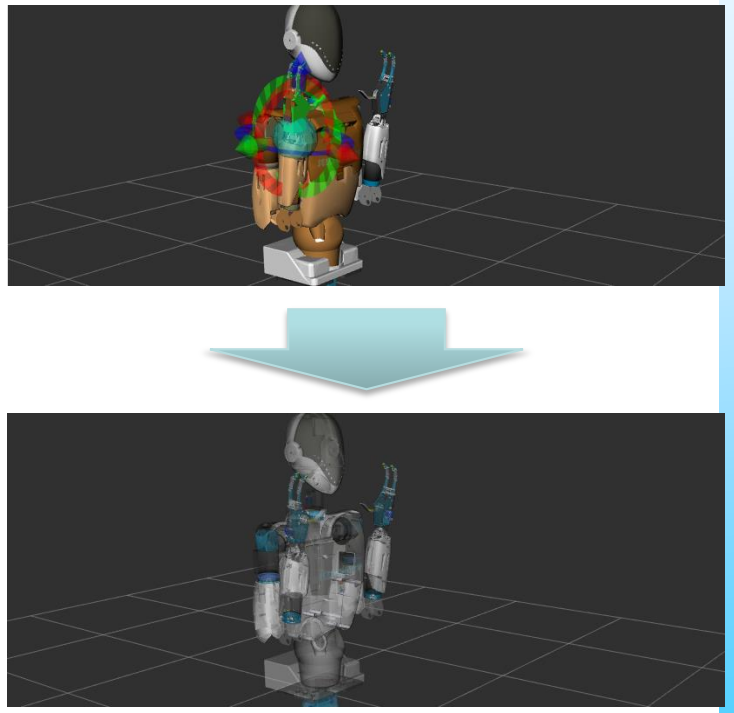
ターミナル２で，“MoveIt! tutorial node is created!!”と表示されればOK.

Pythonノード利用のための準備(1)

Pythonノードからロボットを動かすために，ロボットの動きが見やすいように，RVizの設定を変更する．

手先の操作用インタフェースの消去



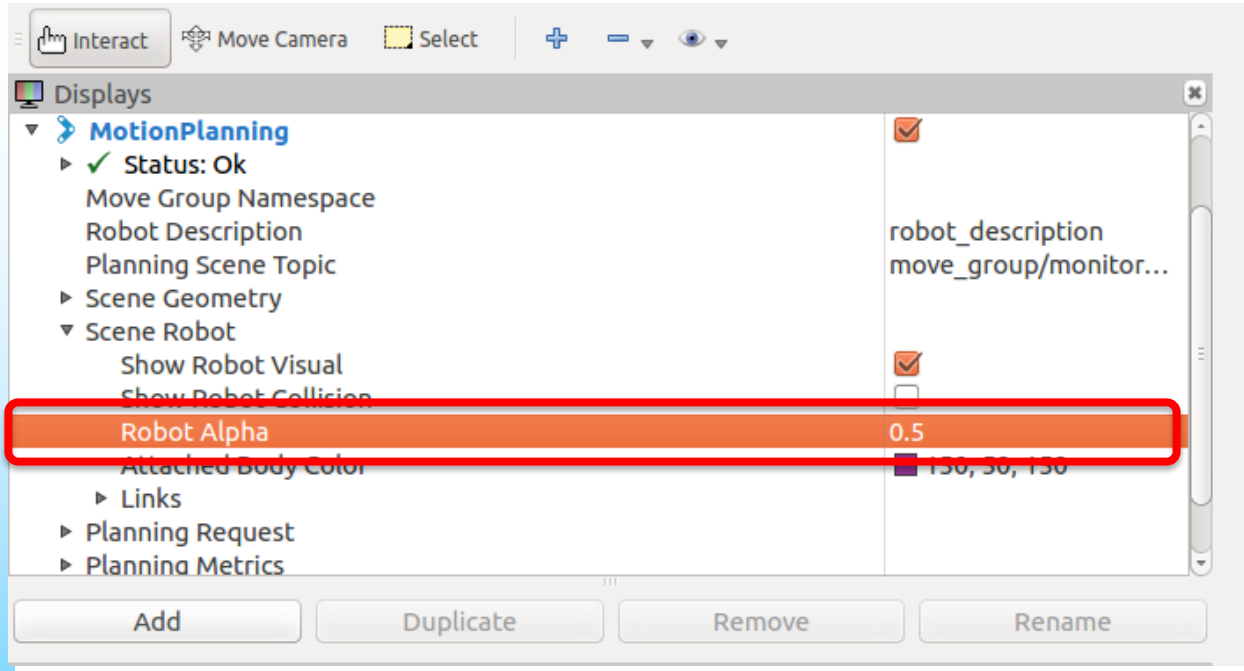


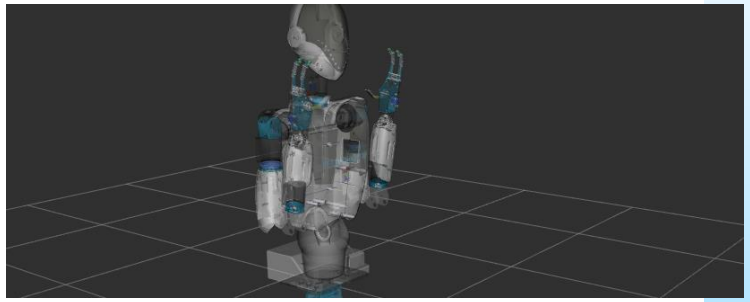
“Query Goal State”のチェックを外す．
(手先に表示されている球や矢印などが消える．)

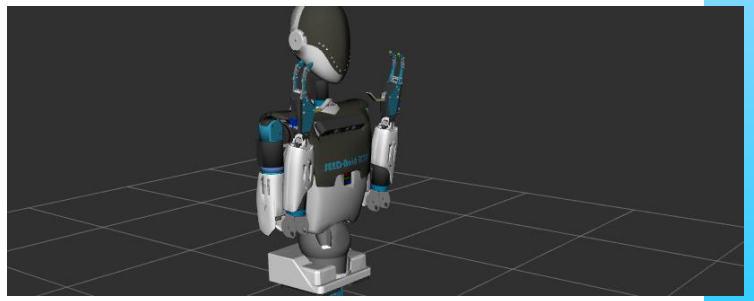
Pythonノード利用のための準備(2)

ロボットの透過率を上げる(好みに合わせて)

ロボットが透過していると少し見にくいので，透過率を上げて，透過しないようにする．





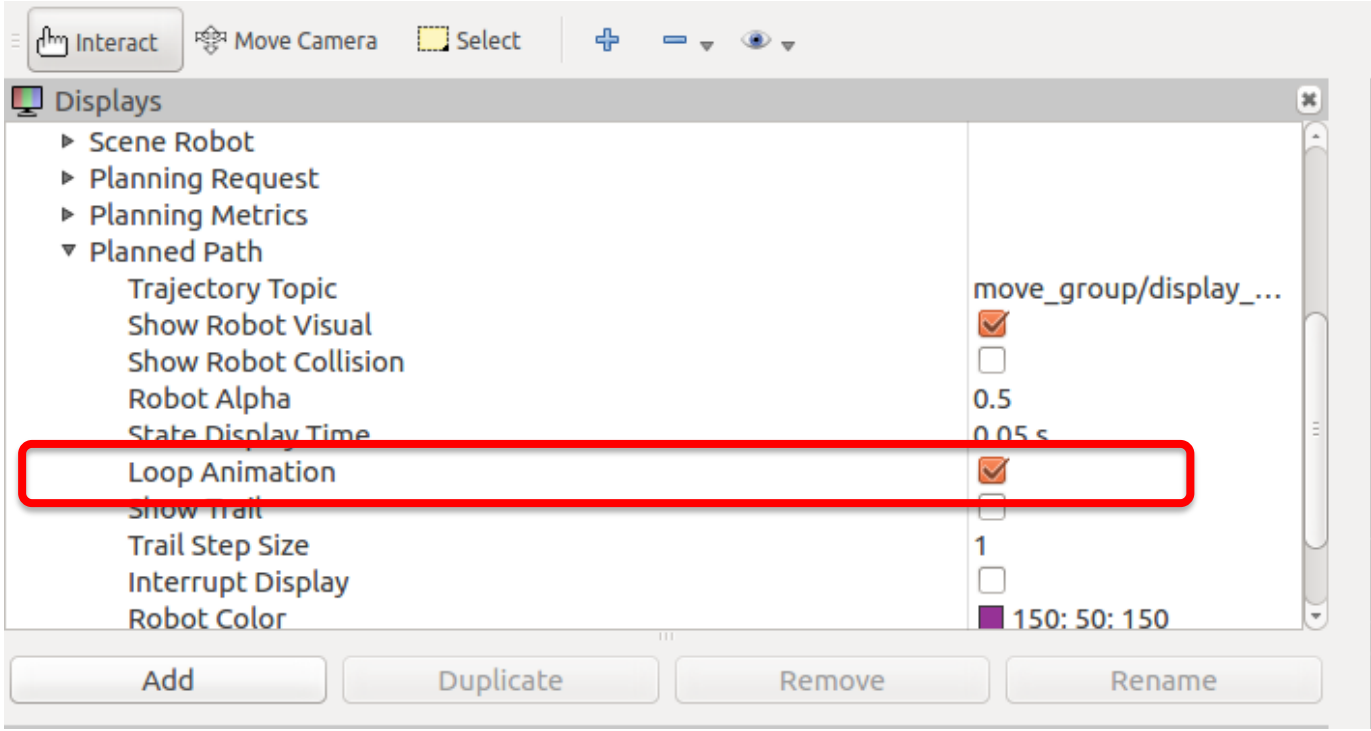


“Robot Alpha”を“1”に変更する．

Pythonノード利用のための準備(3)

軌道生成時のアニメーションのループの停止

軌道生成時のアニメーションをループしないように設定.



“Loop Animation”のチェックを外す

チェックを外すことで，アニメーションがループしないようになる.

腕の関節角度を表示するノード作成

演習2　ファイル名はmoveit_tutorial2.pyとする.

指定したグループ(左腕，右腕)の関節角度を表示するノードを作成.

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

import rospy
import math
import copy
import moveit_commander
import moveit_msgs.msg
from geometry_msgs.msg import Quaternion, Pose, PoseStamped, Vector3

##############################################
#Main Part
##############################################
def main():
  #Initialization of Node
  rospy.init_node("moveit_tutorial_node")

  #Configuration for MoveIt!
  robot = moveit_commander.RobotCommander()
  #Display of Group Name
  print "Robot Groups:", robot.get_group_names()
  print "Robot State:", robot.get_current_state()
  #Store the objects of each group name
  rarm = moveit_commander.MoveGroupCommander("rarm")
  larm = moveit_commander.MoveGroupCommander("larm")
  upper_body = moveit_commander.MoveGroupCommander("upper_body")
  rarm_with_waist = moveit_commander.MoveGroupCommander("rarm_with_waist")

  #Display of Joint Name
  print ""
  print "Right Arm Group Joint Names:", robot.get_joint_names("rarm")
  print ""
  print "Left Arm Group Joint Names:", robot.get_joint_names("larm")
  print ""
  print "Upper Body Group Joint Names:", robot.get_joint_names("upper_body")
  print ""
  print "Right Arm with Waist Group Joint Names:", robot.get_joint_names("rarm_with_waist")

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

演習１からの追記部分

ノードの解説(演習2)

作成したノードに関する説明(抜粋)を以下に示す.
(右腕の例だが，左腕についても同様)

```
#ロボットが持つグループ名一覧表示（Rvizで変更したグループ）
print "Robot groups:", robot.get_group_names()

#ロボットの現在の情報の表示
print "Robot state:", robot.get_current_state()

#右腕のグループを「rarm」として設定
rarm = moveit_commander.MoveGroupCommander("rarm")

#（右腕の）関節の名前および順番の一覧表示
print "RARM Joint names:", robot.get_joint_names("rarm")
```

ノードの起動と動作検証

作成したノードの動作検証を行う．なお，今回作成したノードは，関節角度を表示するだけのものなので，ロボットは動作しない．

ターミナル1

```
$roslaunch seed_r7_bringup moveit.launch
```

以降，ターミナル１は，上記のコマンドが実行されていると仮定．

ターミナル2

```
$rosrun seed_r7_samples moveit_tutorial2.py
```

ターミナル２において，現在の状態などが表示されていれば正常に動作している．このように指定したグループの情報を取得することができる．
表示された内容から，各腕の関節名と順番を確認する．

腕の関節角度を制御するノード作成

演習3

ファイル名はmoveit_tutorial3.pyとする.

右腕の指定関節を任意の角度に変更するノードを作成する.

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

import rospy
import math
import copy
import moveit_commander
import moveit_msgs.msg
from geometry_msgs.msg import Quaternion, Pose, PoseStamped, Vector3

##############################################
#Main Part
##############################################
def main():
  #Initialization of Node
  rospy.init_node("moveit_tutorial_node")

  #Configuration for MoveIt!
  robot = moveit_commander.RobotCommander()
  #Display of Group Name
  print "Robot Groups:", robot.get_group_names()
  print "Robot State:", robot.get_current_state()

  #Store the objects of each group name
  rarm = moveit_commander.MoveGroupCommander("rarm")
  larm = moveit_commander.MoveGroupCommander("larm")
  upper_body = moveit_commander.MoveGroupCommander("upper_body")
  rarm_with_waist = moveit_commander.MoveGroupCommander("rarm_with_waist")
```

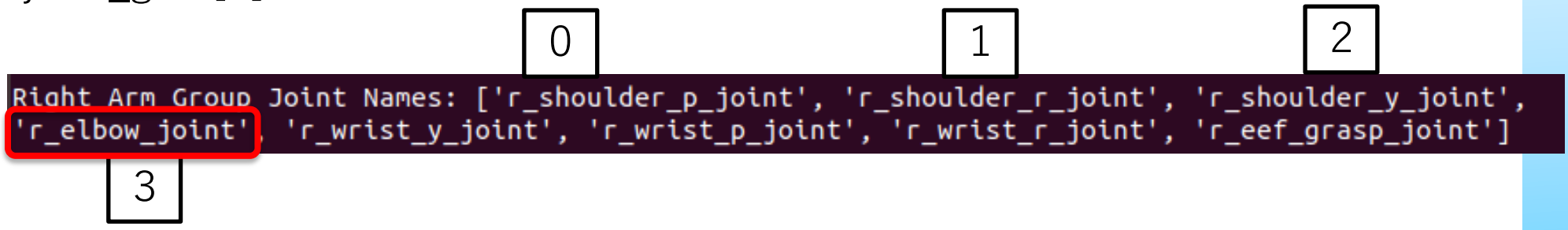
演習2からの変更部分

```python
  #Set Pose to Home Position
  joint_goal = upper_body.get_current_joint_values()
  for i in range(0, len(joint_goal)):
    joint_goal[i] = 0
  joint_goal[6] = -3
  joint_goal[16] = -3
  upper_body.go(joint_goal, wait=True)

  #Right Arm Joint Control
  #Set joint Goal
  joint_goal = rarm.get_current_joint_values()
  #Set elbow joint of Right Arm to 90[deg]
  joint_goal[3] = -1.57
  #Plan and Execute
  print "Right Arm:", rarm.go(joint_goal, wait = True)
  #Display of joint status for Right Arm
  print ""
  print "Right Arm Joint Status:", rarm.get_current_joint_values()

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

このサンプルでは，第４関節に注目

ノードの解説（演習3）

```
#　現在の関節角度を取得
joint_goal = rarm.get_current_joint_values()

#　肘の関節角度を-90[deg]に設定
#　SEED-Noidにおける４番目の関節（0から数えて)は肘
#　角度の単位はラジアンのため，ラジアンで角度指定.
joint_goal[3] = -1.57
```



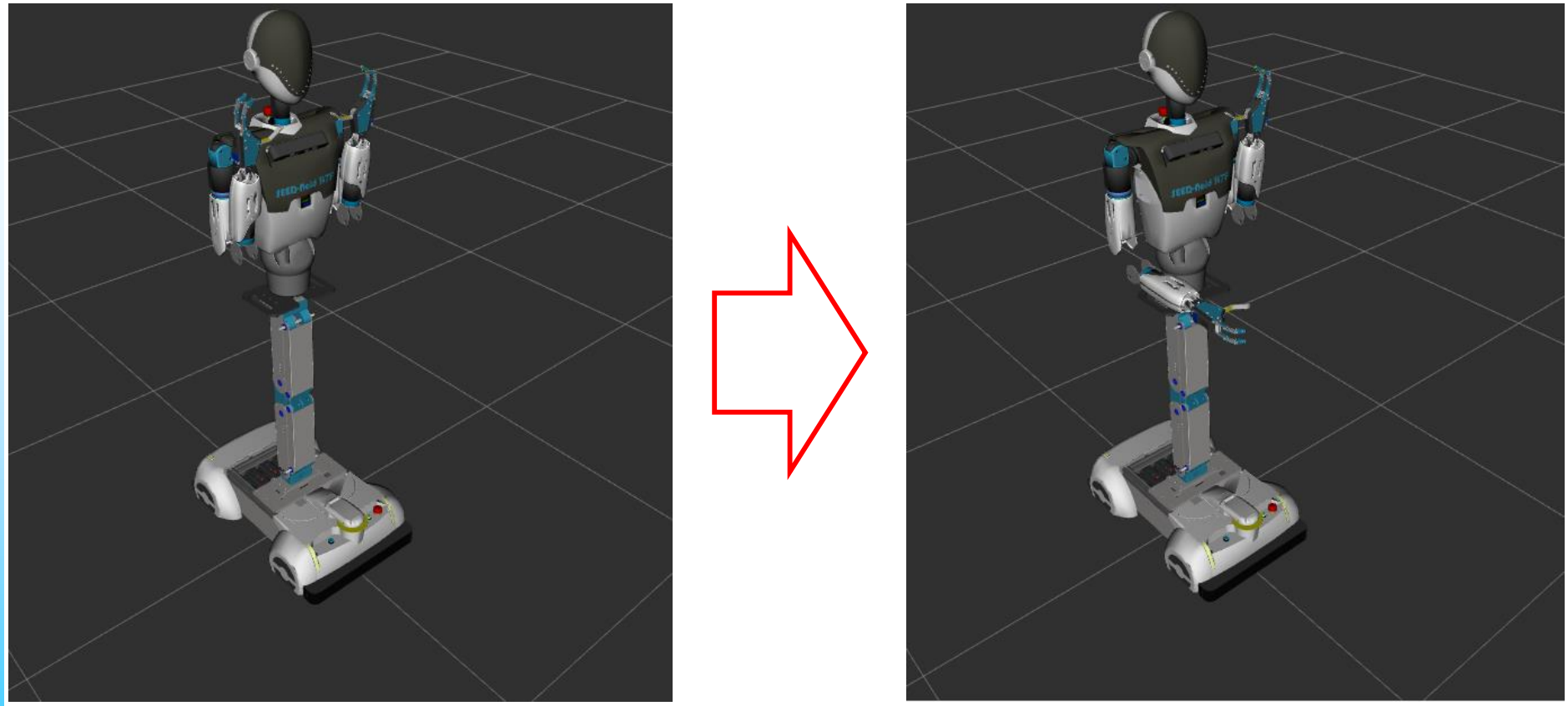

```
#　指定した関節角度で動かす
print "Right Arm:", rarm.go(joint_goal, wait = True)

#　現在の関節角度を取得
print "Right Arm Joint Status:", rarm.get_current_joint_values()
```

ノードの起動と動作検証

ターミナル2

```
$rosrun seed_r7_samples moveit_tutorial3.py
```

正しくプログラムが書けていれば，以下の様に動作する．



以降の演習について

以降の演習では，下記の部分を変更しないで利用します．

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

import rospy
import math
import copy
import moveit_commander
import moveit_msgs.msg
from geometry_msgs.msg import Quaternion, Pose, PoseStamped, Vector3
from tf.transformations import quaternion_from_euler
from seed_r7_ros_controller.srv import *

##################################################
#Global Function
##################################################
SLEEP_TIME = 1.5

def euler_to_quaternion(role, pitch, yaw):
  q = quaternion_from_euler(role, pitch, yaw)
  return Quaternion(q[0], q[1], q[2], q[3])

##################################################
#Main Part
##################################################
def main():
  #Initialization of Node
  rospy.init_node("moveit_tutorial_node")

  #Configuration for MoveIt!
  robot = moveit_commander.RobotCommander()
  scene = moveit_commander.PlanningSceneInterface()
  #Display of Group Name
  print "Robot Groups:", robot.get_group_names()
  print "Robot State:", robot.get_current_state()

  #Store the objects of each group name
  rarm = moveit_commander.MoveGroupCommander("rarm")
  larm = moveit_commander.MoveGroupCommander("larm")
  upper_body = moveit_commander.MoveGroupCommander("upper_body")
  rarm_with_waist = moveit_commander.MoveGroupCommander("rarm_with_waist")

  #Set Pose to Home Position
  joint_goal = upper_body.get_current_joint_values()
  for i in range(0, len(joint_goal)):
    joint_goal[i] = 0
  joint_goal[6] = -3
  joint_goal[16] = -3
  upper_body.go(joint_goal, wait=True)

  #ここから演習４以降のコードを追加してください。

  #ここまで


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

moveit_tutorial_template.py として提供．

以降はここにコードを追加

以降の演習について

これまでの演習同様に，ターミナルは２つ利用します．
１つ目のターミナルでMoveIt!系を起動し，もう１つのターミナルで，作成したノードを起動します．
<ファイル名>は各演習に合わせたファイル名を指定してください．

ターミナル1

```
$roslaunch seed_r7_bringup moveit.launch
```

ターミナル2

```
$rosrun seed_r7_samples <ファイル名>
```

複数の関節を同時に動かす

これまでの演習では，1つの関節のみ制御する形であったが，複数の軸を同時に動かすことを考える．

演習4	ファイル名はmoveit_tutorial4.pyとする．

左腕，右腕＋腰，上半身を同時に制御する．

```
#ここから演習４以降のコードを追加してください。
#左腕の指令値の生成
larm.set_end_effector_link("l_eef_pick_link")
joint_goal = larm.get_current_joint_values()
#左腕の肘を90度に設定
joint_goal[3] = -1.57
print "Left Arm:", larm.go(joint_goal, wait = True )

#右腕と腰を一緒に動かすための指令値を生成
joint_goal = rarm_with_waist.get_current_joint_values()
#腰を回転させる
joint_goal[0] = 1.0
#右の肘を動かす
joint_goal[6] = -1.0
print"Right Arm with Waist:", rarm_with_waist.go(joint_goal, wait = True)

```

```
#上半身の制御を行うための指令値を生成
joint_goal = upper_body.get_current_joint_values()
#腰を回転させる
joint_goal[0] = 0
#両腕の制御
joint_goal[6] = -2.3
joint_goal[16] = -2.3
print"Upper Body:", upper_body.go(joint_goal, wait = True)
#ここまで
```

(左側にタブがはいっているので，その点を注意してください．)

複数の関節を同時に動かす（補足）

この演習で指定している配列は，演習2で表示した情報に従っています．各ジョイントの配列は一番はじめを0とするので，左から順番に，０，１，２．．．として，動かしたい関節の番号を指定する．

関節の名前および順番

右腕

```
Right Arm Group Joint Names: ['r_shoulder_p_joint', 'r_shoulder_r_joint', 'r_shoulder_y_joint',
'r_elbow_joint', 'r_wrist_y_joint', 'r_wrist_p_joint', 'r_wrist_r_joint', 'r_eef_grasp_joint']
```

左腕

```
Left Arm Group Joint Names: ['l_shoulder_p_joint', 'l_shoulder_r_joint', 'l_shoulder_y_joint', '
l_elbow_joint', 'l_wrist_y_joint', 'l_wrist_p_joint', 'l_wrist_r_joint', 'l_eef_grasp_joint']
```

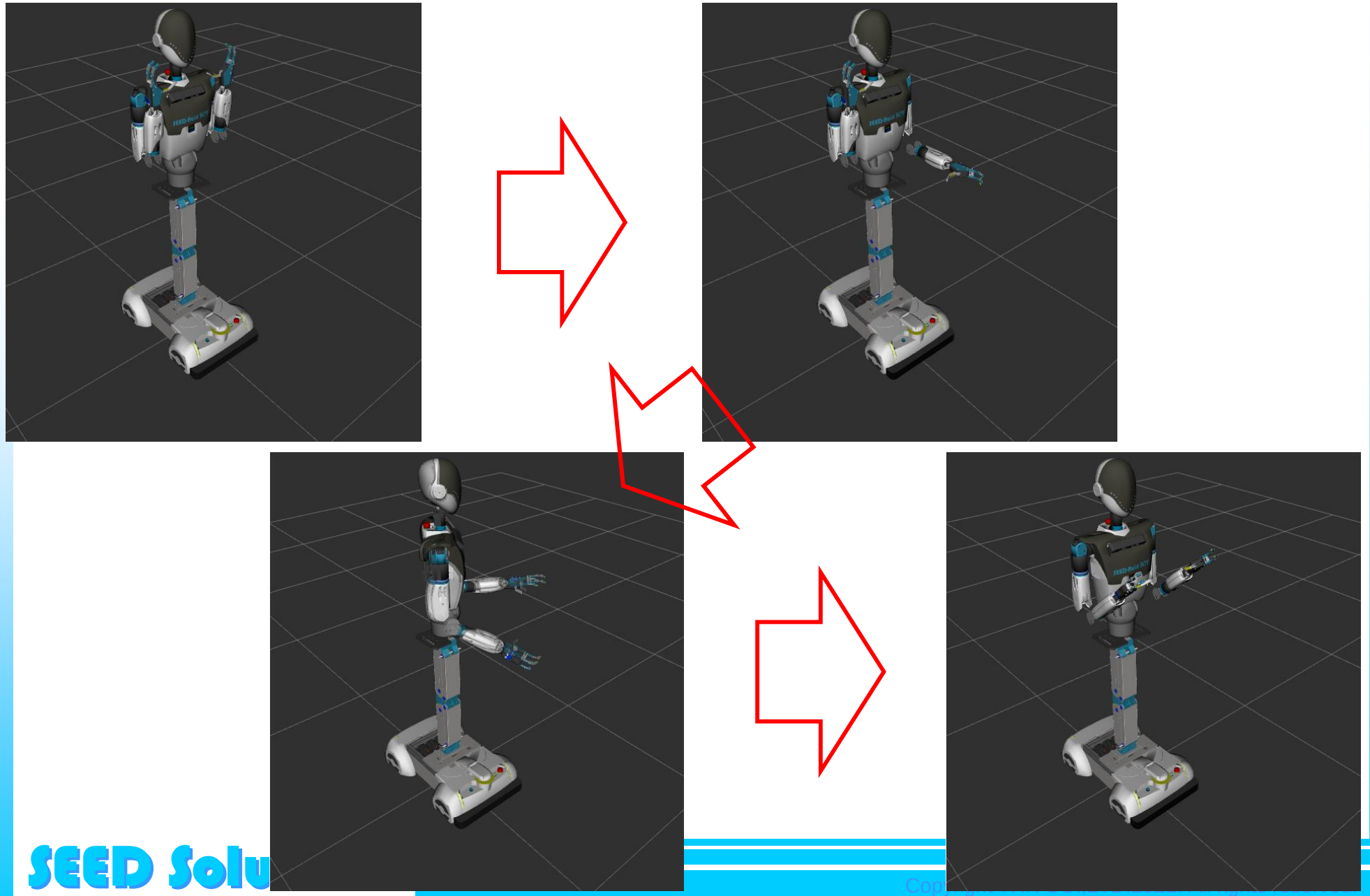
上半身

```
Upper Body Group Joint Names: ['waist_y_joint', 'waist_p_joint', 'waist_r_joint', 'l_shoulder_p_
joint', 'l_shoulder_r_joint', 'l_shoulder_y_joint', 'l_elbow_joint', 'l_wrist_y_joint', 'l_wrist
_p_joint', 'l_wrist_r_joint', 'l_eef_grasp_joint', 'neck_y_joint', 'neck_p_joint', 'neck_r_joint
', 'r_shoulder_p_joint', 'r_shoulder_r_joint', 'r_shoulder_y_joint', 'r_elbow_joint', 'r_wrist_y
_joint', 'r_wrist_p_joint', 'r_wrist_r_joint', 'r_eef_grasp_joint']
```

上半身は配列の数が多いですが，全てを１個の配列で管理しているだけで，名前を見ればどこの関節かわかるかと思います．

複数の関節を同時に動かす（動作例）

演習４のコードが正しく記述てきていると，以下のように動作します．



手先座標を指定して動かす

実際にマニピュレータを制御する場合は，角度指定よりは手先位置を指定して動作させることが多いため，ここでは手先の目標位置姿勢を指定して動作させてみる．

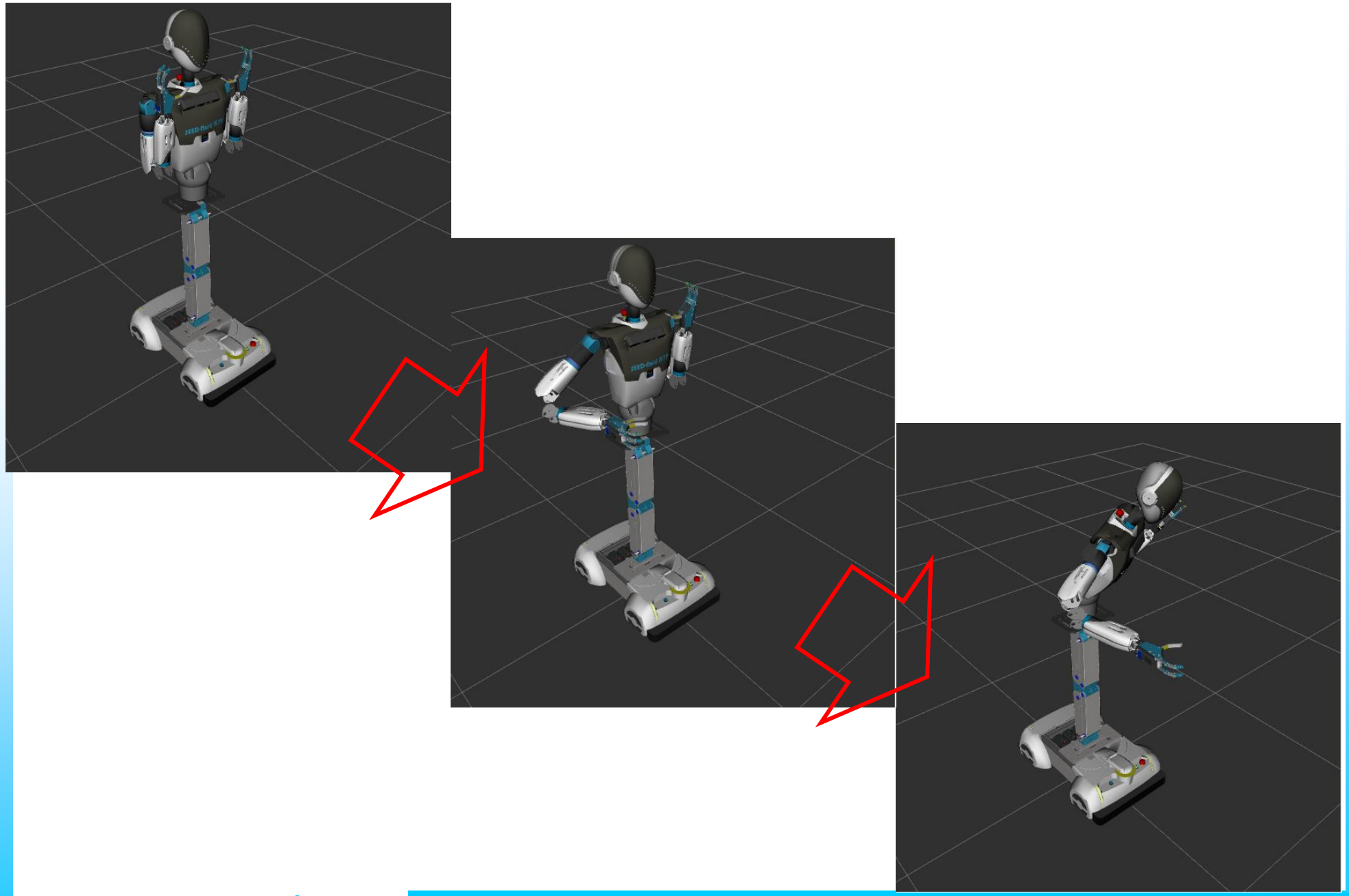
演習5　ファイル名はmoveit_tutorial5.pyとする．

```
50  #ここから演習４以降のコードを追加してください。
51  #右腕の目標位置・姿勢へ動かす
52  rarm.set_end_effector_link("r_eef_pick_link")
53  pose_goal = Pose()
54  pose_goal.orientation.w = 1.0
55  pose_goal.position.x = 0.4
56  pose_goal.position.y = -0.25
57  pose_goal.position.z = 1.2
58  rarm.set_pose_target(pose_goal)
59  rarm.go(wait=True)
60  print "Right Arm Pose:"
61  print "", rarm.get_current_pose().pose
62
63  #右腕と腰の全体での目標位置・姿勢へ動かす
64  rarm_with_waist.set_end_effector_link("r_eef_pick_link")
65  pose_goal = Pose()
66  pose_goal.orientation.w = 1.0
67  pose_goal.position.x = 0.8
68  pose_goal.position.y = -0.2
69  pose_goal.position.z = 1.2
70  rarm_with_waist.set_pose_target(pose_goal)
71  rarm_with_waist.go(wait = True)
72  print "Right Arm with Waist Pose:"
73  print "", rarm_with_waist.get_current_pose().pose
74  #ここまで
```

コードの解説

```
#　右腕のエンドエフェクタ情報を設定
rarm.set_end_effector_link("r_eef_pick_link")

#　目標位置・姿勢を格納する変数の準備
pose_goal = Pose()

#　目標姿勢をクォータニオン(x,y,z,w)で定義
#　(詳しくは調べること．)
pose_goal.orientation.w = 1.0

#　目標位置を定義
pose_goal.position.x = 0.4

#　目標位置姿勢をセット
rarm.set_pose_target(pose_goal)

#　セットした目標位置姿勢で動かす
print "RARM",rarm.go(wait=True)

#　現在の位置姿勢を表示
print "RARM Pose:",rarm.get_current_pose().pose
```

手先座標を指定して動かす(動作例)



手先座標を指定して動かす(直線軌道)

先ほどは，特に移動する際の軌道に制約を設けていなかったが，実際には，障害物を回避するなどの目的で，軌道は厳密に規定するべきである．そこで，ここでは一番初歩的な軌道として，直線軌道を用いて，手先の目標位置・姿勢まで持って行く方法を演習する．

演習6

ファイル名はmoveit_tutorial6.py

手先位置・姿勢を指定して，直線軌道でマニピュレータを動かす

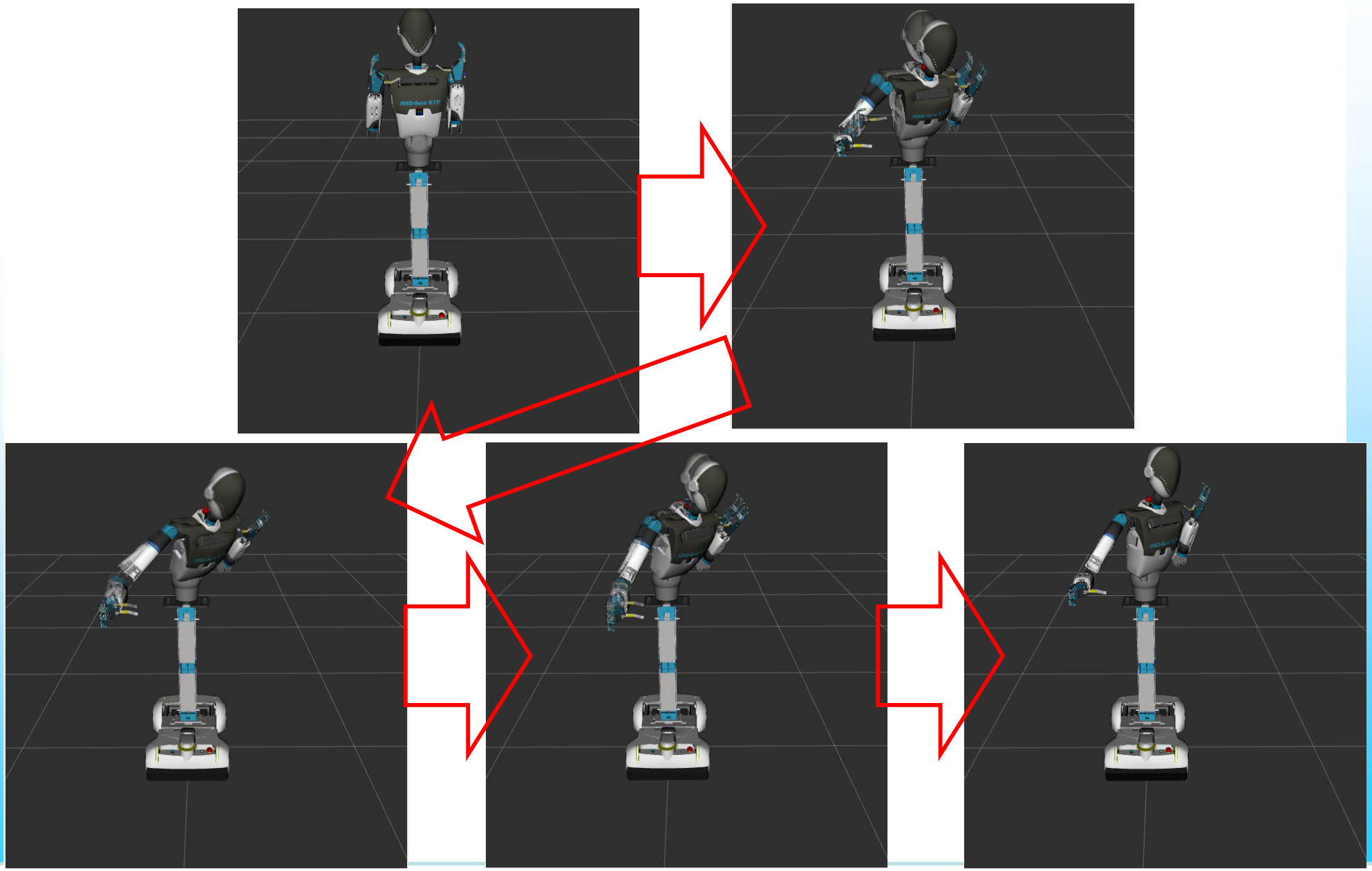
```
50  #ここから演習４以降のコードを追加してください。
51  #右腕と腰の全体での目標位置・姿勢へ動かす
52  rarm_with_waist.set_end_effector_link("r_eef_pick_link")
53  pose_goal = Pose()
54  pose_goal.orientation.w = 1.0
55  pose_goal.position.x = 0.8
56  pose_goal.position.y = -0.2
57  pose_goal.position.z = 1.2
58  rarm_with_waist.set_pose_target(pose_goal)
59  rarm_with_waist.go(wait = True)
60
61  #手先の直線軌道での制御
62  waypoints = []
63  wpose = rarm_with_waist.get_current_pose().pose
64  wpose.position.z +=-1*0.1
65  waypoints.append(copy.deepcopy(wpose))
66  wpose.position.y += 1*0.1
67  waypoints.append(copy.deepcopy(wpose))
68  wpose.position.z += 1*0.1
69  waypoints.append(copy.deepcopy(wpose))
70  wpose.position.y +=-1*0.1
71  waypoints.append(copy.deepcopy(wpose))
72
73  (plan, fraction) = rarm_with_waist.compute_cartesian_path(waypoints, 0.01, 0.0)
74  print "CARTESIAN", rarm_with_waist.execute(plan, wait=True)
75  #ここまで
```

手先座標を指定して動かす(直線軌道)

作成したコードの補足

```
#   経由点リストを作成
waypoints = []

#   現在の右腕の位置・姿勢を取得
wpose = rarm_with_waist.get_current_pose().pose

#   z軸に-0.1少ない目標位置姿勢を経由点に設定
#   （移動量の重みを変えられるように，0.1としている．）
wpose.position.z += -1 * 0.1
waypoints.append(copy.deepcopy(wpose))

#   直線軌道を計算.
#   ２番目の引数は0.01は、刻み幅を表す
(plan, fraction) = rarm_with_waist.compute_cartesian_path(waypoints,0.01,0.0)

#   実行
print "CARTESIAN",rarm_with_waist.execute(plan, wait=True)
```

手先座標を指定して動かす(直線軌道)



手先で正方形を描くように直線で軌道が生成されていることがわかる.

ピックアンドプレース

アームを持つロボットでは，ワーク(対象物)を把持し，なにかしらの操作を行い，別の場所に移動するという作業が行われる．ここでは，マニピュレータにおける基本的な作業であるピックアンドプレースについて行う．

今回利用しているパッケージでは，RViz上でグリッパの開閉動作が行えないため，ここでは，グリッパの開閉ができると仮定した動きを行います．

ピックアンドプレース

演習7　ファイル名はmoveit_tutorial7.pyとする．

把持姿勢に移動してから，箱をつかみ(と想定し，)，別の位置に搬送し，開放し元の姿勢に戻る．

```
#ここから演習４以降のコードを追加してください。
#右腕と腰の全体での目標位置・姿勢へ動かす
rarm_with_waist.set_end_effector_link("r_eef_pick_link")
pose_goal = Pose()
pose_goal.orientation = euler_to_quaternion(-1.57, 0, 0)
pose_goal.position.x = 0.8
pose_goal.position.y = -0.2
pose_goal.position.z = 1.2
rarm_with_waist.set_pose_target(pose_goal)
rarm_with_waist.go(wait = True)
rospy.sleep(SLEEP_TIME*3)

#RViz上に箱を表示
box_name = "box"
box_pose = PoseStamped()
box_pose.header.frame_id = "r_eef_pick_link"
box_pose.pose.orientation = euler_to_quaternion(0,0,0)
scene.add_box(box_name, box_pose, size=(0.03, 0.03, 0.03))
rospy.sleep(SLEEP_TIME)

#箱を手先と関連付ける
grasping_group = 'rarm_with_waist'
touch_links = robot.get_link_names(group=grasping_group)
eef_link = rarm_with_waist.get_end_effector_link()
scene.attach_box(eef_link, box_name, touch_links=touch_links)
rospy.sleep(SLEEP_TIME)
```

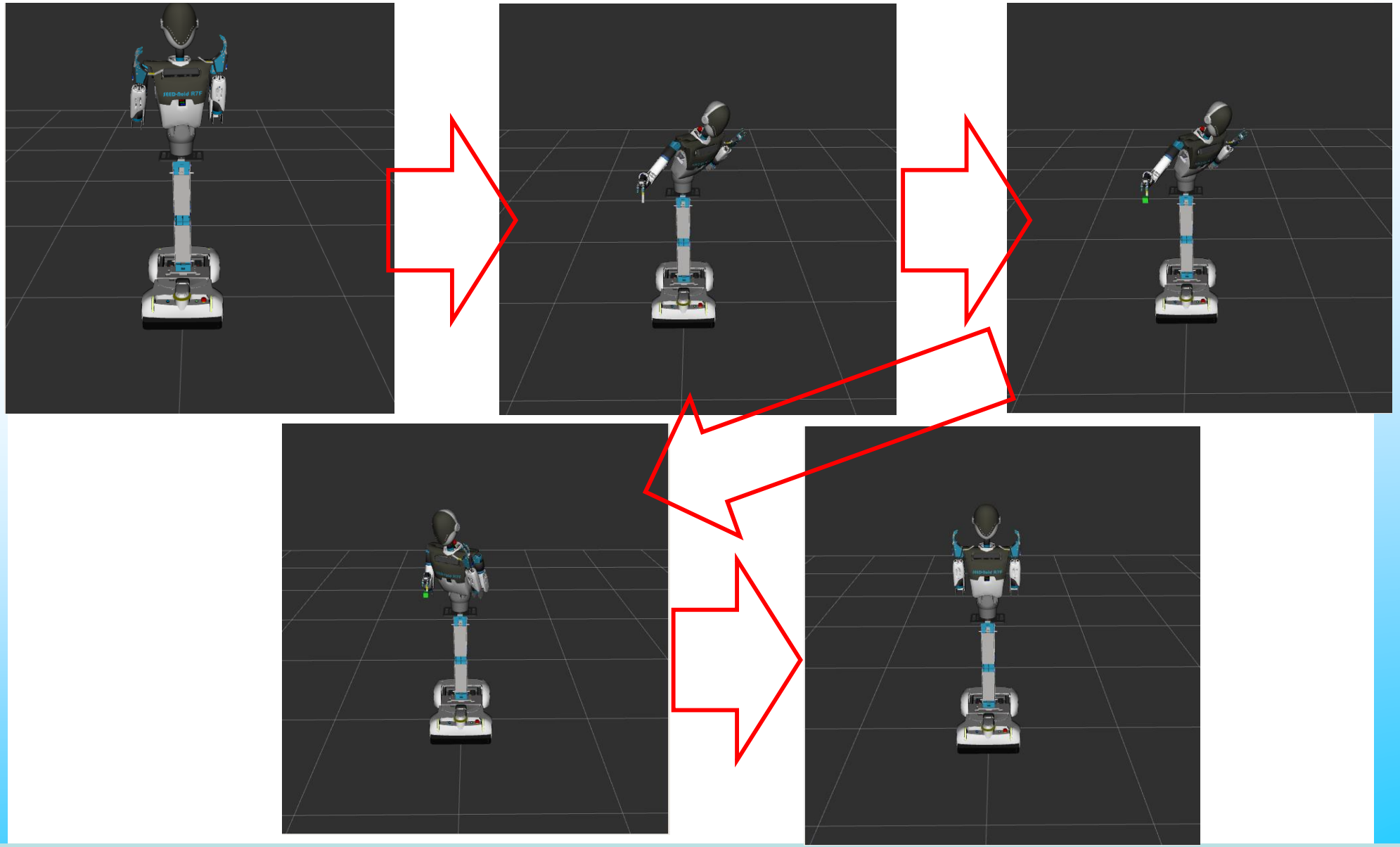
```
#右手と腰の全体を用いて、物体を移動する
pose_goal.orientation = euler_to_quaternion(-1.57, 0, 0)
pose_goal.position.x = 0.4
pose_goal.position.y = -0.2
pose_goal.position.z = 1.2
rarm_with_waist.set_pose_target(pose_goal)
rarm_with_waist.go(wait = True)

#箱を置く
scene.remove_attached_object(eef_link, name=box_name)
rospy.sleep(SLEEP_TIME)

#初期位置に戻る
joint_goal = upper_body.get_current_joint_values()
for i in range(0, len(joint_goal)):
  joint_goal[i] = 0
joint_goal[6] = -3
joint_goal[16] = -3
upper_body.go(joint_goal, wait=True)

#箱を消す
scene.remove_world_object(box_name)

#ここまで
```

ピックアンドプレース

作成したコードの補足

```
#　オイラー角からクオータニオンへと変換を行う
pose_goal.orientation = euler_to_quaternion(-1.57,0,0)

#　シーン（RVizで見えているところ）変更のための準備
scene = moveit_commander.PlanningSceneInterface()

#　ボックス（ピックアンドプレースの対象物）の設定
box_name = "box"
box_pose = PoseStamped()
box_pose.header.frame_id = "r_eef_pick_link"
box_pose.pose.orientation = euler_to_quaternion(0,0,0)

#　シーン上にボックスを追加
scene.add_box(box_name, box_pose, size=(0.03, 0.03, 0.03))

#　ここでは、シーン変更待機のためにスリープ
rospy.sleep(SLEEP_TIME)
```

ピックアンドプレース(動作例)



箱を把持し，移動させてからリリースしている様子がわかる．

第４部のまとめ

MoveIt!をもちいて，SEED-Noidのマニピュレーションのためのモーション生成手法について学んだ.

ここで学んだ事は，マニピュレータの軸の名称の違いなどはありますが，他のマニピュレータでも活用できる知識であるので，第４部の流れをよく理解してもらえると応用が効くと思います.

全体のまとめ

本講座を通じて，以下を学んだ.

- ROSの基礎
- SEED-Noidによる地図生成とナビゲーション
- SEED-Noidによるマニピュレーション

今回学んだ内容は，RVizやGazeboが中心でしたが，これらは実機と通信ができる状態になれば，そのままで実機を動かすことができます！

ぜひ，実際に実機に触れていただき，ROSの利便性を体感していただくとともに，SEED-Noidのロボットとしての良さを感じてほしいとも思います.