ROS 2 - Tiago Robot + Moveit2

Alessio Caporali

alessio.caporali@unibo.it

DEI - LAR
University of Bologna
Viale del Risorgimento 2
40136 Bologna

Moveit2

In order to use MoveIt 2 with the Tiago robot, we need to launch the MoveIt 2 interface. Moveit is enabled by default in the Tiago simulation package.

Therefore, we just need to launch the simulation with the usual command:

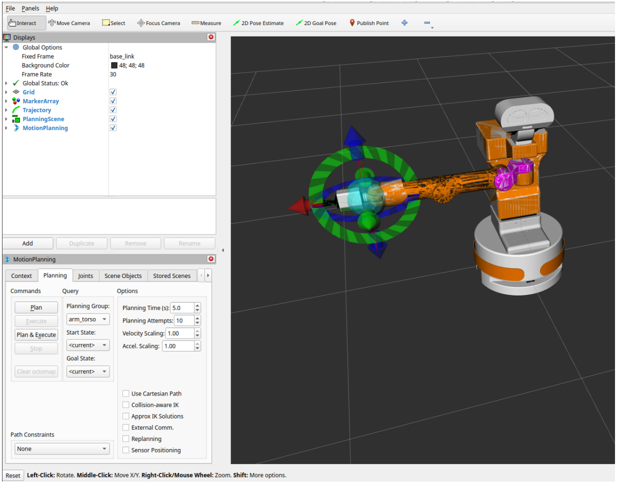
```
ros2 launch tiago_gazebo tiago_gazebo.launch.py
```

To launch the Moveit interface in Rviz, we need to run:

```
ros2 launch tiago_moveit_config moveit_rviz.launch.py
```

This will open the MoveIt 2 interface in Rviz, where we can plan and execute motion for the robot.

Moveit2



The MoveIt 2 interface in Rviz provides a graphical interface to plan and execute motion for the robot. However, we want to want to control the robot programmatically...

Moveit2

Tiago arm chain (from tf tree in Rviz2):

- torso_fixed_link
  - torso_lift_link
    - arm_1_link
      - arm_2_link
        - arm_3_link
          - arm_4_link
            - arm_5_link
              - arm_6_link
                - arm_7_link
                  - arm_tool_link
                    - wrist_ft_link
                      - wrist_ft_tool_link
                        - gripper_link
                          - gripper_right_finger_link
                          - gripper_grasping_frame
                          - gripper_left_finger_link
                        - gripper_tool_link

pymoveit2

Basic Python interface for MoveIt 2 built on top of ROS 2 actions and services (Python bindings for MoveIt 2 still missing in ROS Humble).

Clone the repository into the tiago_ws/src folder:

```
git clone https://github.com/AndrejOrsula/pymoveit2.git
```

Compile the workspace:

```
colcon build --symlink-install
```

Before utilising this package, remember to source the ROS 2 workspace.

```
source /tiago_ws/install/setup.bash
```

This enables importing of pymoveit2 module in external Python scripts (like in our case).

pymoveit2

We are going to use pymoveit2 to plan and execute different types of motion:

- **Pose Goal**: we are going to move the robot to a specific pose in the workspace (position and orientation).
- **Joint Goal**: we are going to move the robot to a specific joint configuration (joint angles).
- **Gripper Goal**: we are going to open and close the gripper plus move the fingers to a specific position.

In order to exploit pymoveit2, we need the Tiago robot to be up and running, as well as the MoveIt 2 interface (see previous slides).

Pose Goal

```python
from threading import Thread
import rclpy
from rclpy.callback_groups import ReentrantCallbackGroup
from rclpy.node import Node
from pymoveit2 import MoveIt2

# Tiago Parameters
JOINT_NAMES = [
    "torso_lift_joint",
    "arm_1_joint",
    "arm_2_joint",
    "arm_3_joint",
    "arm_4_joint",
    "arm_5_joint",
    "arm_6_joint",
    "arm_7_joint",
    "arm_tool_joint",
]
BASE_LINK_NAME = "base_link"
END_EFFECTOR_NAME = "arm_tool_link"
GROUP_NAME = "arm_torso"
```

Pose Goal

```
# Create callback group that allows parallel execution of callbacks
callback_group = ReentrantCallbackGroup()

# Create MoveIt 2 interface
moveit2 = MoveIt2(
    node=node,
    joint_names=JOINT_NAMES,
    base_link_name=BASE_LINK_NAME,
    end_effector_name=END_EFFECTOR_NAME,
    group_name=GROUP_NAME,
    callback_group=callback_group,
)

moveit2.planner_id = "RRTConnectkConfigDefault"
```

Pose Goal

```
# Spin the node in background thread(s) and wait a bit for init
executor = rclpy.executors.MultiThreadedExecutor(2)
executor.add_node(node)
executor_thread = Thread(target=executor.spin, daemon=True, args=())
executor_thread.start()
node.create_rate(1.0).sleep()


# Move to pose
position = [0.5, 0.0, 0.5]
quat_xyzw = [0.0, 0.0, 0.0, 1.0]
moveit2.move_to_pose(position=position, quat_xyzw=quat_xyzw)
moveit2.wait_until_executed()
```

Joint Goal

For the joint goal, we need to specify the joint angles of the robot. We can do this by using the `move_to_configuration` method of pymoveit2.

```
# Move to pose
joint_positions = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
moveit2.move_to_configuration(joint_positions)
moveit2.wait_until_executed()
```

Gripper Goal

Moving the gripper is quite similar. The tiago, in addition to the **arm_torso** group, has also a **gripper** group that allows to control the gripper.

```
from threading import Thread
import rclpy
from rclpy.callback_groups import ReentrantCallbackGroup
from rclpy.node import Node
from pymoveit2 import GripperInterface

# Tiago Parameters
JOINT_NAMES = [
    "gripper_left_finger_joint",
    "gripper_right_finger_joint",
]

OPEN_GRIPPER_JOINT_POSITIONS = [0.04, 0.04]
CLOSED_GRIPPER_JOINT_POSITIONS = [0.0, 0.0]
GRIPPER_GROUP_NAME = "gripper"
GRIPPER_COMMAND_ACTION_NAME = "gripper_controller/joint_trajectory"
```

Gripper Goal

```python
# Create gripper interface
gripper_interface = GripperInterface(
    node=node,
    gripper_joint_names=JOINT_NAMES,
    open_gripper_joint_positions=OPEN_GRIPPER_JOINT_POSITIONS,
    closed_gripper_joint_positions=CLOSED_GRIPPER_JOINT_POSITIONS,
    gripper_group_name=GRIPPER_GROUP_NAME,
    callback_group=callback_group,
    gripper_command_action_name=GRIPPER_COMMAND_ACTION_NAME,
)

# Perform gripper action
gripper_interface.open()
gripper_interface.wait_until_executed()

gripper_interface.close()
gripper_interface.wait_until_executed()

# Move to a specific position
gripper_interface.move_to_position(0.02)
gripper_interface.wait_until_executed()
```

Tip! PyKDL

We have seen that pymoveit2 requires the pose of the robot to be specified in terms of **position and orientation**. However, we know that it is in general more convenient to work directly with **homogeneous transformations**. This would simplify frame transformations, rotation representations, etc.

A standard library for managing transformations is KDL. There is a Python binding for KDL called PyKDL. We can use PyKDL to convert between different representations of the pose.

Full documentation of PyKDL can be found at: `https://docs.ros.org/en/diamondback/api/kdl/html/python/geometric_primitives.html`

Tip! PyKDL

PyKDL usage is quite straightforward. There are 3 main classes that we are going to use:

- **Vector**: represents a 3D vector (e.g. the position in our pose).
- **Rotation**: represents a 3D rotation (e.g. the orientation in our pose).
- **Frame**: represents a 3D frame (position and orientation).

Example of creating a Vector from a specified position:

```
from PyKDL import Vector
position = Vector(0.5, 0.0, 0.5) # x, y, z
```

Tip! PyKDL

Rotation are a bit more complex. We can create a rotation from a rotation matrix, from Euler angles, from a quaternion, etc.

Example of creating a Rotation from a rotation matrix or quaternion:

```
from PyKDL import Rotation

# Rotation matrix (9 individual elements)
rotation = Rotation(1, 0, 0, 0, 1, 0, 0, 0, 1)

# Quaternion (4 elements)
rotation = Rotation.Quaternion(0, 0, 0, 1)
```

Tip! PyKDL

Frames are created by combining a position and a rotation:

```
from PyKDL import Frame
frame = Frame(rotation, position)
```

The Frame object of PyKDL is quite powerful. It allows to perform operations like:

- **Inverse**: get the inverse of the frame (Frame.Inverse())
- **Multiply**: multiply two frames (Frame_1 $\times$ Frame_2)
- **GetRotation**: get the rotation part of the frame (Frame.M)
- **GetTranslation**: get the translation part of the frame (Frame.p)

But most importantly, it allows to easily **concatenate transformations** by simply multiplying frames together.

Tf Broadcaster

With a TF broadcaster, we can publish a transformation between two frames. This is useful when we want to define a new frame in the workspace, or when we want to publish the pose of an object.

The transformation is defined by the position and orientation of the child frame with respect to the parent frame.

We can check the broadcasted transformation in Rviz by adding a TF display.

```
from geometry_msgs.msg import TransformStamped
from tf2_ros import TransformBroadcaster

class FramePublisher(Node):

  def __init__(self):
      super().__init__('example_tf_broadcaster')
      self.tf_broadcaster = TransformBroadcaster(self)

      self.parent_name = "base_footprint"
      self.child_name = "my_new_frame"
```

Tf Broadcaster

```
def publish_frame(self):
    t = TransformStamped()
    t.header.stamp = self.get_clock().now().to_msg()
    t.header.frame_id = self.parent_name
    t.child_frame_id = self.child_name

    # position
    t.transform.translation.x = 1.0
    t.transform.translation.y = 0.0
    t.transform.translation.z = 1.0

    # orientation
    rot = kdl.Rotation()
    rot.DoRotX(np.pi/4)
    quat = rot.GetQuaternion()
    t.transform.rotation.x = quat[0]
    t.transform.rotation.y = quat[1]
    t.transform.rotation.z = quat[2]
    t.transform.rotation.w = quat[3]

    # Send the transformation
    self.tf_broadcaster.sendTransform(t)
```