ROS 2 Tiago Robot + Vision

Alessio Caporali

alessio.caporali@unibo.it

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

Tiago Robot

Tiago is a mobile manipulator robot developed by PAL Robotics.

- 1 arm with 7 DoF
- 2-finger gripper
- RGBD head camera (with 2DoF pan-tilt)
- Mobile base

Tiago in ROS2

To start the simulation of the Tiago robot in ROS 2, first we need to reach the tiago workspace and build the packages.

```
cd /home/student/tiago_ws
```

We build and source the workspace (if neccessary)

```
colcon build --symlink-install
. install/setup.bash
```

We can start the simulation with the following command

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

Controlling the Joints of the Tiago Robot

One of the simplest ways to control the joints of the Tiago robot is to use the **rqt_joint_trajectory_controller**.

In case the package is not installed, you can install it with the following command:

```
sudo apt install ros-humble-rqt-joint-trajectory-controller
```

To start the controller, use the following command:

```
ros2 run rqt_joint_trajectory_controller
rqt_joint_trajectory_controller
```

Tiago with Moveit!

To start Moveit!, use the following command:

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

Part 1
Camera Subscriber and Publisher

Tiago RGBD Head Camera

RGB + Depth camera (similar to Kinect v1 or Asus Xtion Pro).
The sensor uses a **structured light** pattern to estimate depth.

Tiago RGBD Head Camera - ROS Topics

The relevant topics of the Tiago RGBD head camera are:

```
    # -> sensor_msgs/CameraInfo
    /head_front_camera/rgb/camera_info
    /head_front_camera/depth_registered/camera_info

    # -> sensor_msgs/Image
    /head_front_camera/rgb/image_raw
    /head_front_camera/depth_registered/image_raw

    # -> sensor_msgs/PointCloud2
    /head_front_camera/depth_registered/points
```

How to visualize the camera images?

There are two simple ways to visualize an image topic in ROS 2:

Method 1
Use rqt to bring up an image viewer

```
ros2 run rqt_image_view rqt_image_view
```

Method 2
Use rviz2 with the image plugin. If you can not see the image, pay attention to select a correct "*fixed frame*" in the "*Global Options*", i.e. a frame for which there is a valid connection in the TF tree.

sensor_msgs/Image Message

Detailed description of the sensor_msgs/Image message:

```
# This message contains an uncompressed image
# (0, 0) is at top-left corner of image

Header header # timestamp (acquisition time of image)
             # frame_id (optical frame of camera)
             # origin of frame (optical center of camera)
             # +x should point to the right in the image
             # +y should point down in the image
             # +z should point into to plane of the image

uint32 height # image height (number of rows)
uint32 width # image width (number of columns)
string encoding # Encoding of pixels
uint8 is_bigendian # is this data bigendian?
uint32 step # Full row length in bytes
uint8[] data # actual matrix data, size is (step * rows)
```

Image Subscriber

Task: Create a ROS 2 node that subscribes to the RGB image topic of the Tiago head camera and displays the image in a window.

```
import cv2
from sensor_msgs.msg import Image
from cv_bridge import CvBridge

# inside init
self.bridge = CvBridge()
self.camera_sub = self.create_subscription(
  Image, "camera_topic", self.callback_camera_image, 1)

# callback + image conversion (from ROS to OpenCV)
self.img = self.bridge.imgmsg_to_cv2(
  msg, desired_encoding="bgr8")

# display image
cv2.imshow("Camera Image", self.img)
cv2.waitKey(1)
```

Image Publisher

Task: Create a ROS 2 node that publishes a new image (e.g. the original image flipped vertically) on a new topic.

```
import cv2
from sensor_msgs.msg import Image
from cv_bridge import CvBridge

# inside init
self.bridge = CvBridge()
self.camera_pub = self.create_publisher(
  Image, "camera_topic, 1)

# publisher + image conversion (from OpenCV to ROS)
img_flipped = cv2.flip(img_np, 0) # just for example
img_msg = self.bridge.cv2_to_imgmsg(
  img_flipped, encoding="bgr8")
self.camera_pub.publish(img_msg)
```

Part 2
3D Point Projection

3D Point Projection

How to project a 3D (world) point into the image plane of a camera? For this we need the **camera intrinsic** parameters and the **camera extrinsic** parameters. As an example of point to project, we will use the origin of the *gripper frame*.

Key steps:

- Use the TF listener to get the *gripper frame* with respect to the *camera frame*.
- Use the camera intrinsic parameters to project the *gripper frame* origin into the image plane.

Frames to use in the next slide:

```
frame_1 = "head_front_camera_rgb_optical_frame"
frame_2 = "gripper_grasping_frame"
```

3D Point Projection - TF Listener

```
from tf2_ros.buffer import Buffer
from tf2_ros.transform_listener import TransformListener

def __init__(self):
  self.tf_buffer = Buffer()
  self.tf_listener = TransformListener(self.tf_buffer, self)

  # Call on_timer function every second
  self.timer = self.create_timer(0.1, self.on_timer)

def on_timer(self):
  try:
      self.t = self.tf_buffer.lookup_transform(
        frame_1, frame_2, rclpy.time.Time())
  except TransformException as ex:
      self.get_logger().info('Could not transform!')
      return
```

3D Point Projection - Homogeneous Transformation Matrix

The output of the TF listener is a geometry_msgs/TransformStamped message.

We need to extract the rotation and translation parts of the message, in order to build the homogeneous transformation matrix. We are going to use Scipy (in case of missing package, try to install it with *sudo apt install python3-scipy*).

3D Point Projection - Camera Pose

```
from scipy.spatial.transform import Rotation

# t: TransformStamped message
def homogeneous_matrix_from_transform(self, t):
  position = np.array([ t.transform.translation.x,
                        t.transform.translation.y,
                        t.transform.translation.z])
  quat = [t.transform.rotation.x,
          t.transform.rotation.y,
          t.transform.rotation.z,
          t.transform.rotation.w]

  T_matrix = np.eye(4)
  T_matrix[:3, 3] = position
  T_matrix[:3, :3] = Rotation.from_quat(quat).as_matrix()
  return T_matrix
```

3D Point Projection - Camera Intrinsics Message

The camera intrinsic parameters are provided by the camera_info topic.

sensor_msgs/CameraInfo

```
std_msgs/Header header
uint32 height
uint32 width
string distortion_model
float64[] D # For "plumb_bob": (k1, k2, t1, t2, k3).
float64[9] K # 3x3 row-major matrix
float64[9] R
float64[12] P
uint32 binning_x
uint32 binning_y
sensor_msgs/RegionOfInterest roi
```

3D Point Projection - Retrieve Camera Intrinsics

We can use the *wait_for_message* function to retrieve the camera intrinsic parameters (alternatively use a subscriber)

```
from sensor_msgs.msg import CameraInfo
from rclpy.wait_for_message import wait_for_message

self.camera_info_topic =
       "/head_front_camera/rgb/camera_info"

ret, msg = wait_for_message(
       CameraInfo, self, self.camera_info_topic)

self.cam_K = np.array(msg.k).reshape(3,3)
self.cam_D = np.array(list(msg.d))
```

3D Point Projection - Put it all together

```python
import cv2
import numpy as np

def project(points3d, camera_pose, cam_K, cam_D):
    R, _ = cv2.Rodrigues(camera_pose[:3, :3])
    t = camera_pose[:3, 3]
    points3d = points3d.reshape(-1, 3).astype(np.float32)
    points2d, _ = cv2.projectPoints(
      points3d, R, t, cam_K, cam_D)
    return points2d.squeeze()
```

- points3d is the list of 3D points in the reference frame *frame_2*. In our case we will use just the origin of the gripper frame, i.e. [0, 0, 0].
- camera_pose is the homogeneous transformation matrix expressing *frame_2* with respect to *frame_1*.
- cam_K and cam_D are the camera intrinsic and distortion parameters.

3D Point Projection - Put it all together

To visualize the result, we can draw the projected point on the image and publish it.

```
import cv2
from sensor_msgs.msg import Image
from cv_bridge import CvBridge

# draw projected point on image
cv2.circle(img, (int(point2d[0]), int(point2d[1])),
5, (0, 255, 0), -1)

# convert image to ROS message
img_msg = self.bridge.cv2_to_imgmsg(
  img, encoding="bgr8")

# publish image
self.camera_pub.publish(img_msg)
```

Part 3
From 2D to 3D

2D Point → 3D - Setup

Now let's try to do the opposite: given a 2D point in the image plane, we want to project it into the 3D world.

Input point publisher (in the old node)
First step, we need to get the input 2D point from the image topic.
We can use the previous node to publish the point in a new topic called *target_point*.
For this purpose, we can use the message *geometry_msgs/Point*.

New node with depth image subscriber
We are going to use the depth image to get the Z coordinate of the 2D point.

```
topic = "/head_front_camera/depth_registered/image_raw"

# inside callback
self.depth_img = self.bridge.imgmsg_to_cv2(
    msg, desired_encoding=msg.encoding)
```

2D Point → 3D - 3D Point Calculation

To calculate the 3D point, we need to use the camera intrinsic parameters and the depth image. We can use the *wait_for_message* function to retrieve the camera intrinsic parameters (like before)

Notice that OpenCV uses the column/row convention for the image coordinates, the opposite of the row/column convention used in the ROS message and for numpy arrays. Therefore, when accessing the depth image, we need to swap the indices.

```
    depth_value = self.depth_img[
          int(self.point[1]), int(self.point[0])]

    Z = depth_value
    X = Z * (self.point[0] - self.cam_K[0,2]) / self.cam_K[0,0]
    Y = Z * (self.point[1] - self.cam_K[1,2]) / self.cam_K[1,1]
```

We can compare the calculated 3D point with the original 3D point (the previous *T_matrix*). They should be close to each other.