Robotics Foundations (H) - Lab 2

ILOs

In this week's lab, you will understand what's behind the Baxter robot (http://www.rethinkrobotics.com/baxter/) (Figure 1) simulation and how its kinematic transformations are maintained. The Baxter robot is an industrial robot built by Rethink Robotics. It has two arms and was intended for use in manufacturing production lines. The Baxter robot can learn how to do a task through someone moving the robot's arms in the desired motion. It will memorise the action and be able to repeat it. This 'muscle memory' ability allows Baxter to be programmed by any regular person and makes it fast and straightforward to use. Hence, you will:

- Understand how ROS handles coordinate transformation and how this translates into a robot
- Understand the Unified Robot Description Format (URDF) and XML macros (aka XACRO)
- Be able to move Baxter

Figure 1: The Baxter robot at the School of Computing Science

Coordinate Transforms in ROS

As in the lecture, coordinate frame transformations are fundamental to robotics. For articulated robot arms, you can deduce the forward and inverse kinematics to compute gripper poses as a function of joint angles or vice-versa. This is possible by multiplying sequential transforms describing the current state of a robot. Sensor data, e.g. from cameras, can be merged into the robot's coordinate frame, and this data must be interpreted in terms of alternative frames (e.g. world frame or robot frame) to allow a robot to perceive and act within an environment.

ROS provides a powerful package for handling, maintaining and updating these transformations. This package is tf (http://wiki.ros.org/tf) and offers command-line tools for inspecting transformations of a robot. This package also provides APIs in ROS's supported programming languages to manipulate and plan movements with robots.

To start with, let's start Gazebo (ROS simulation from the previous lab) and bring up a robot (Baxter) by typing in a terminal:

```
cd ~/ros_ws
./baxter.sh
roslaunch baxter_gazebo baxter_world.launch
```

`baxter.sh` is similar to `source devel/setup.bash` but sets up extra `bash` arguments for simulation and when using the real robot. Baxter has several topics published, including the topic for transformations "tf". Press Ctrl+Enter to run the following command:

In [ ]:

```
%%bash
rostopic info tf
```

This topic uses messages of type tf2_msgs/TFMessage (http://docs.ros.org/kinetic/api/tf2_msgs/html/msg/TFMessage.html) which consists of a variable length array of a stamped transform (http://docs.ros.org/kinetic/api/geometry_msgs/html/msg/TransformStamped.html). ROS does not employ the homogenous 4-by-4 matrix representation covered in the lecture but uses a 3-D vector (equivalent to the forth column of a 4-by-4 transform) and a quaternion (http://www.euclideanspace.com/maths/algebra/realNormedAlgebra/quaternions/) (an alternative representation of orientation). This data structure is more efficient to carry pose information than the homogenous matrix representation, e.g. homogenous transformations need 12 floating point numbers (16 numbers but the last row is always a row vector with the 4th entry equal to 1) as opposed to 7 floating point numbers when using quaternions. Also, transform messages have timestamps, and they explicitly name the child frame and the parent frame (as text strings).

In ROS, orientations are commonly expressed as unit quaternions. The quaternion representation is an alternative to rotation matrices, have powerful mathematical properties, and avoid several problems while using homogeneous transformations and Euler angles. **However, you do not need to worry about quaternions as it is outside the scope of this course. TF provides functions to transform quaternions into Euler angles and even homogenous transformations!** At this point, it is sufficient to know that there is a correspondence between quaternions and rotation matrices (and ROS functions to perform such conversions will be introduced later) and that there are corresponding mathematical operations for coordinate transformations with quaternions.

There can be (and typically are) many publishers to the tf topic. Each publisher expresses a transform relationship, describing a named child frame with respect to a named parent frame - more about this in the following sections. You can now examine the output of the `tf` topic with

In [ ]:

```
%%bash
rostopic echo /tf -n 1
```

The above command shows all available transformations within Baxter. As mentioned above, each message has a child and parent frame which defines the transformation between the given link names. Each link name is a reference frame within the robot. Baxter's frames are defined with respect to a "world reference frame", e.g. the 0, 0, 0 coordinate - the origin, and is always defined in the `world` frame in ROS and is the parent for all transformations. To verify this, run in a terminal:

```
rosrun tf tf_echo /world /base
```

As you can observe, the transformation between these links is defined at the origin. Another topic of interest is `joint_states`. This topic displays the actual position of Baxter's joints, and this information feeds to TF to compose all transformations between reference frames. `joint_states` lives in a ROS node that maintains a record of the joint states of a robot by either updating values from the motor's sensor readings or in software when there is an open loop control system (i.e. no feedback from motors). To inspect the contents of `joint_states`, press Shift+Enter the following cell:

In [ ]:

```
%%bash
rostopic echo /robot/joint_states -n 1
```

In Baxter, the `joint_states` topic is published in `/robot/joint_states`; the rationale is that topics should be declared within a namespace. In this case, `joint_states` is part of the `robot` namespace (if you run `rostopic list`, you will find out that topics about Baxter are declared using the `robot` namespace). It is recommended to use namespaces in ROS to follow good software engineering practices, and it is convenient when the complexity of the ROS system increases.

The above message list the names of Baxter's joints and then the state of each joint in the same order as the list of names, e.g. the left finger joint of the left gripper, `l_gripper_l_finger_joint`, is `0.0208` radians. These values are needed to compute transforms for link poses for all links defined in Baxter. For instance, the transform of Baxter's `head` frame with respect to the `base` frame cannot be computed without knowing the value of the `head_pan` rotation. To find out the transformation from `base` to `head`, run in a terminal:

```
rosrun tf tf_echo /base /head
```

As stated above, transforms are published to the topic `tf`, where each such message contains a detailed description of how a child frame is spatially related to its parent frame. A parent can have multiple children, but a child must have a unique parent, thus generating a tree of geometric relationships with respect to this parent frame. In ROS, you can create a `tf_listener` which is typically started as an independent thread. This thread subscribes to the `tf` topic and assembles a kinematic tree from individual parent-child transform messages. Since the `tf_listener` incorporates all transformations, it can address specific queries, such as *where is Baxter's right-hand finger frame relative to Baxter's head?"* The reply of this query is a transform message that can be used to reconcile different frames. As long as a complete tree is published connecting frames, the transform listener can be used to transform all sensor data into a common reference frame, thus allowing to display sensory data from multiple sources in a common view (and you can view in rviz as last weeks lab).

To demonstrate the above, let's create a `tf_listener` ROS node, so open your prefered text editor, create a ROS package (if you forgot how to create ROS packages, check out Lab 1), create a new ROS node named `tf_listener.py`, and type the following:

```
#!/usr/bin/env python
import rospy
import math

# Importing TF to facilitate the task of receiving transformations
import tf2_ros

if __name__ == '__main__':
    rospy.init_node('baxter_tf_listener')

    # This creates a transform listener object; once created, it starts r
eceiving
    # transformations using the /tf topic and buffers them up for up to 1
0 seconds.
    tfBuffer = tf2_ros.Buffer()
    listener = tf2_ros.TransformListener(tfBuffer)

    # This is where the magic happens, a query is passed to the listener
 for the
    # /base to /head transform by means of the lookupTransform fn. The ar
guments are
    # from "this frame" to "this frame" at "this specific time"
    # (if you pass "rospy.Time(0), the fn will give you the latest availa
ble transform
    rate = rospy.Rate(1.0)
    while not rospy.is_shutdown():
        try:
            transformation = tfBuffer.lookup_transform('base', 'head', ro
spy.Time())
        except (tf2_ros.LookupException, tf2_ros.ConnectivityException, t
f2_ros.ExtrapolationException):
            rate.sleep()
            continue

        rospy.loginfo("Translation: \n" + str(transformation.transform.tr
anslation))
        rospy.loginfo("Quaternion: \n" + str(transformation.transform.rot
ation))

        rate.sleep()
```

Don't forget to make the node executable and `catkin_make`!

Each `transformation` will have the same structure as TransformStamped (http://docs.ros.org/kinetic/api/geometry_msgs/html/msg/TransformStamped.html). The above listener will print the values of the translation and rotation every second until you press Ctrl+C.

Transforms3d (http://matthew-brett.github.io/transforms3d/) is a very useful Python module that you can use to convert ROS transformations (translation and quaternions) into homogenous matrices, rotation matrices, Roll-Pitch-Yaw angles, etc. So to install it, run in a terminal the following:

```
sudo pip install transforms3d sympy
```

and import the `quaternions` module as `from transforms3d import quaternions` and `numpy` in the above ROS node. Now, add the following code above `rate.sleep()`:

```
q = transformation.transform.rotation
quat = (q.x, q.y, q.z, q.w)
rot_mat = quaternions.quat2mat(quat)
rospy.loginfo("Rotation matrix \n" + str(rot_mat))

H = np.eye(4)
trans = transformation.transform.translation
H[:3,:3] = rot_mat
H[:3,3] = [trans.x, trans.y, trans.z]
rospy.loginfo("Homogenous matrix: \n" + str(H))

new_quat = quaternions.mat2quat(rot_mat)
rospy.loginfo("New quaternion: " + str(new_quat))
```

The above code demonstrates how to manipulate TF messages into homogeneous coordinates. Feel free to inspect available functions in the Transforms3d (http://matthew-brett.github.io/transforms3d/), they will become useful when you want to pick up an object using Baxter's hand, and you need to correct the hand orientation to grasp the object successfully!

Exercise

From `tf_listener.py`, write a node that finds transformations from:

- `base` to `left_hand`
- `left_hand` to `right_hand`
- `head_camera` to `left_gripper_base`

Your node should return a 4-by4 homogenous matrix. Alternatively, this node can accept both link names as command-line arguments and return the corresponding homogenous matrix.

Note: The code you develop here will be useful when you work in subsequent labs :)

URDF - Unified Robot Description Format

Until now, you have been using the Gazebo simulator as it is. Indeed, ROS allows you to use 3D models of a robot or its parts to simulate them or to help the developers to visualise Baxter while operating - these models usually come from CAD drawings used when designing a robot. Gazebo and RViz employ the **Unified Robot Description Format** (http://wiki.ros.org/urdf) (URDF). URDF is an XML format that describes a robot, its parts, its joints, dimensions, and so on. So every time you see a 3D robot in ROS, a URDF file is associated with it. For instance, Baxter's URDF model is located at:

In [ ]:

```
%%bash
cd ~/ros_ws/src/baxter/baxter_common/baxter_description/urdf/
cat baxter.urdf
```

URDF files have two key fields that describe the geometry of a robot: `links` and `joints`. As in the previous section, the parent of all links is `base`, and this name must be unique to URDF files:

```
<link name="base">
  </link>
  <link name="torso">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://baxter_description/meshes/torso/base_li
nk.DAE"/>
      </geometry>
      <material name="darkgray">
        <color rgba=".2 .2 .2 1"/>
      </material>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://baxter_description/meshes/torso/base_li
nk_collision.DAE"/>
      </geometry>
    </collision>
    <inertial>
      <origin rpy="0 0 0" xyz="0.000000 0.000000 0.000000"/>
      <mass value="35.336455"/>
      <inertia ixx="1.849155" ixy="-0.000354" ixz="-0.154188" iyy="1.6626
71" iyz="0.003292" izz="0.802239"/>
    </inertial>
  </link>
```

To define what you see in the simulator, it is used the `visual` field in the preceding code. Inside the code, you can define the geometry (cylinder, box, sphere, or mesh), the material (colour or texture), and the origin. In this case, the 3D mesh is DAE file - DAE files are based on the XML COLLADA format, which is short for Collaborative Design Activity. The code for the joint comes after where it is defined the name, which must be unique as well. Also, it is defined the type of joint (fixed, revolute, continuous, floating, or planar), and the parent and the child links. For Baxter, the `torso` is a child of `base`, it is set to the origin and is fixed as shown below:

```
<joint name="torso_t0" type="fixed">
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <parent link="base"/>
    <child link="torso"/>
  </joint>
```

URDF files can become quite large, especially for complex robots such as Baxter. Baxter's URDF file has 1851 lines of code, where it is defined the robot, simulation parameters, grippers, hardware limits, etc. URDF files are used by ROS to construct the kinematic tree of a robot and consequently used by TF to build up and update the geometric transformations of the robot while operating.

In RF, you only need to understand the above XML elements to know which link refers to what - in some cases, you just get the URDF file of a robot. You can also use RViz to find out the latter. So, run Baxter's simulation if you don't have it running and type in a terminal:

```
rosrun rviz rviz
```

Then, select the Add button under the Displays sidebar. In the new window that pops up, scroll to `Robot Model`, and then hit 'OK' to add a visual representation of the robot's current pose. You now need to set the `Fixed Frame` (under `Global Options` to `base`; otherwise you will get a white lump in the middle. It might take a while to come up Baxter. In `RobotModel`, expand the tree, and you should see the default parameters; the important parameter here is *Robot Description* which should point to the `robot_description`. `robot_description` is the name of the ROS parameter where the URDF is stored on the parameter server - you can query the parameter server with:

```
rosparam get /robot_description
```

and you'll get displayed the contents of `baxter.urdf`. If everything is fine, you will see the following window with Baxter on it.

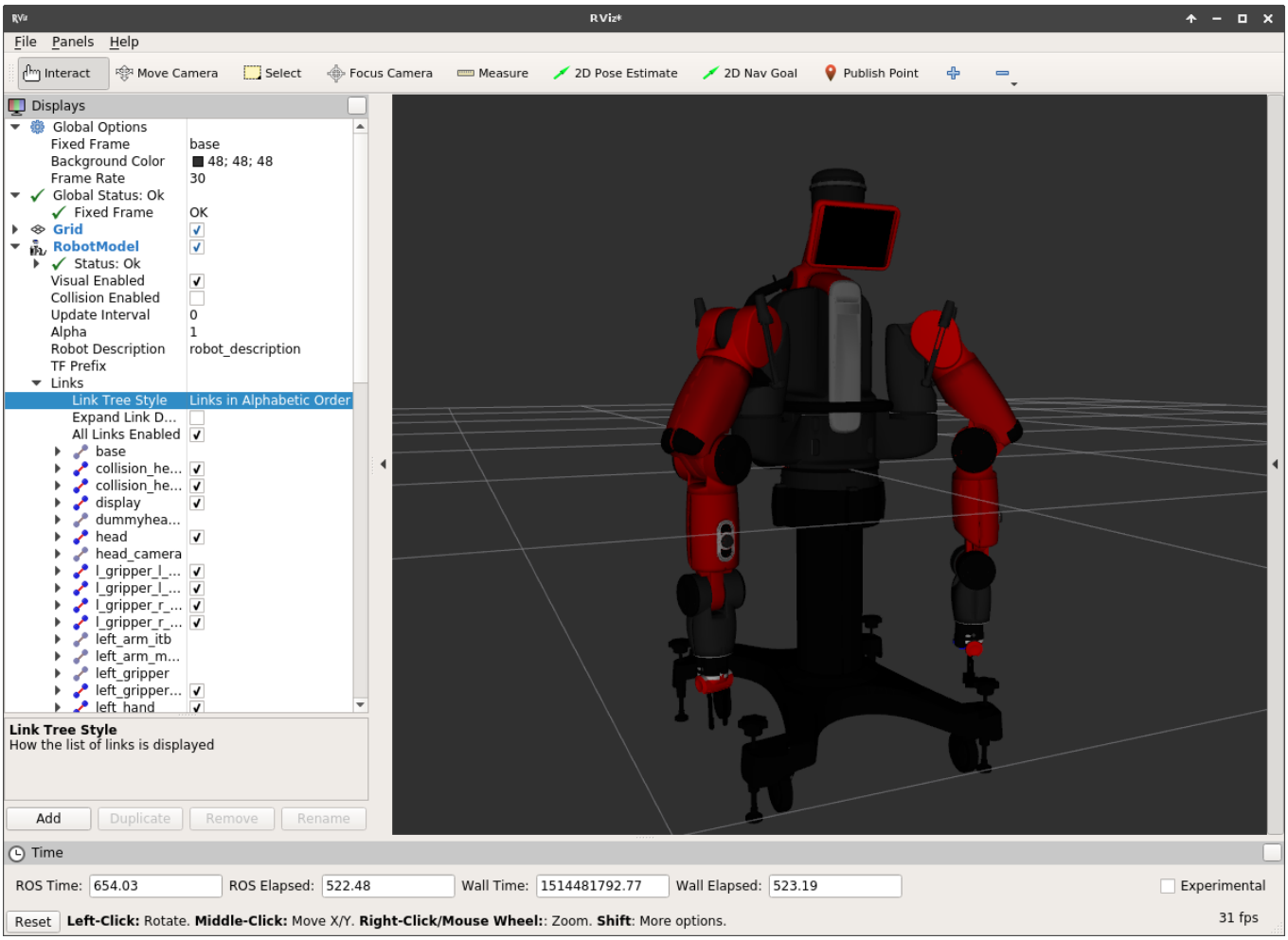

Figure 2: RViz

As stated above, you can examine Baxter's links and joints names and states in RViz. To do this, select Add and scroll to TF and hit 0K to add a TF visualisation of links. You can deactivate `RobotModel` to have a better view of the links and reference frames, as in the figure below. You can change the `Marker Scale` parameter to make the names more readable. You can also deactivate links, coordinate frames visualisation, etc.; feel free to tweak parameters.

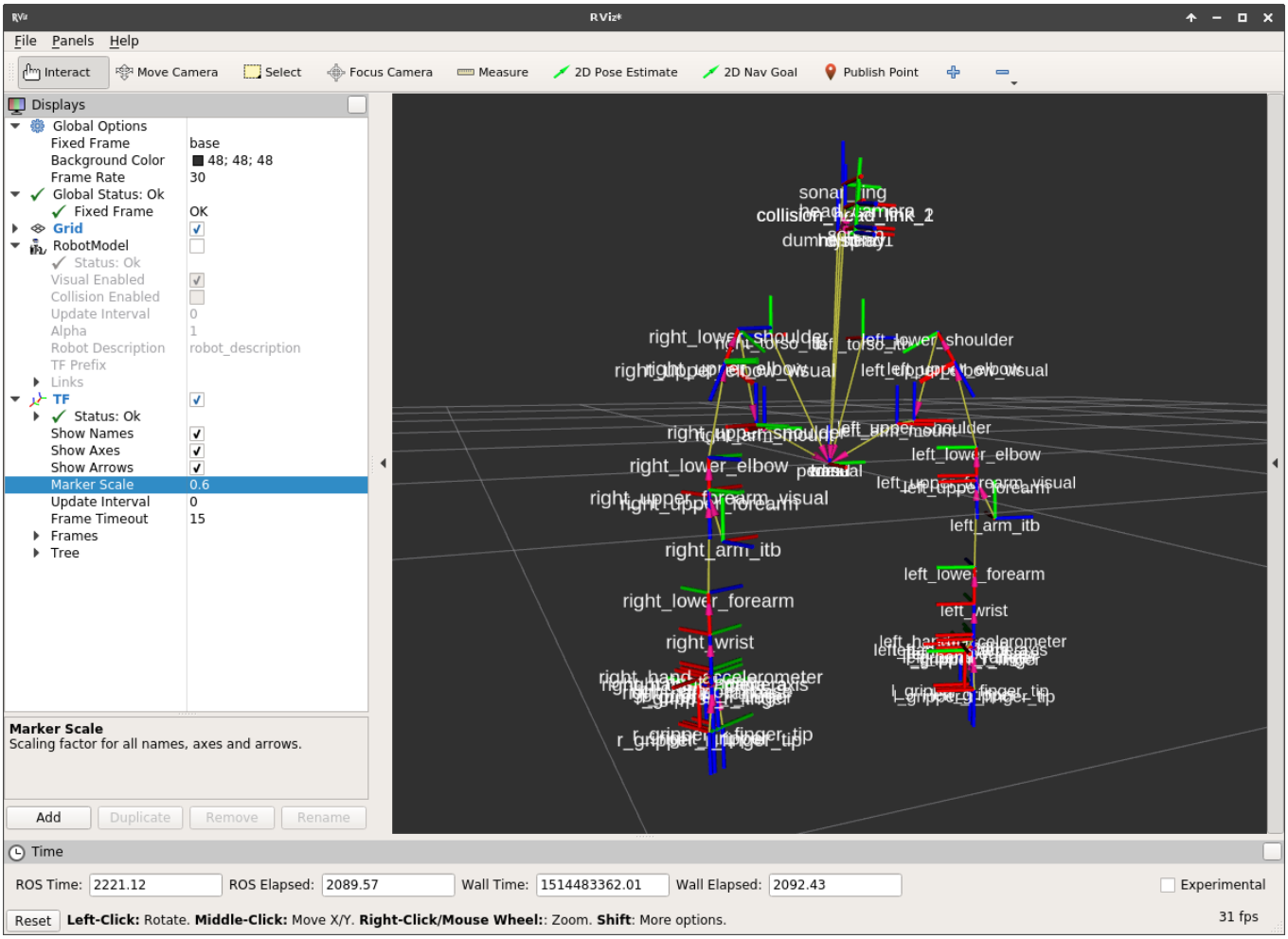

Figure 3: Displaying TF plugin in RViz

XACRO - XML Macros

Baxter's URDF file has 1851 lines of code and imagine adding cameras, sensors and other geometries in it. The file will start increasing, and the maintenance of the code will become more complicated. To minimise the complexity, ROS has the XACRO package which is XML macros and helps in minimising the overall size. XACRO also makes it easier to read and maintain and allows robotic developers to create modules and reutilise them to create repeated structures.

To use XACRO, it is important to specify a namespace so that the file is parsed properly. For instance, the namespace for Baxter in the XACRO file is `baxter`, as shown below:

In [ ]:

```
%%bash
cd ~/ros_ws/src/baxter/baxter_common/baxter_description/urdf
cat baxter.urdf.xacro | grep "robot name"
```

One of the most convenient features in XACRO is the ability to declare constant values. As a result, it can be avoided putting the same value in a lot of lines. Without the use of xacro, it would be almost impossible to maintain the changes if you had to change some values. For instance, the constant values defined in Baxter's XACRO file are:

In [ ]:

```
%%bash
cd ~/ros_ws/src/baxter/baxter_common/baxter_description/urdf
cat baxter.urdf.xacro | grep "default"
```

You can build up arbitrarily complex expressions in the `${}` construct using the four basic operations (+, -, *, /), the unary minus, and the parenthesis. Exponentiation and modulus are, however, not supported. To evaluate constants, use the `${}` construct to evaluate them.

Macros are the most useful component of the xacro package and can allow you to seperate each robot's component into a well-structure collection of xacro files. In Baxter's main XACRO file, you can see `xacro:include`. This XML tag will indicate the parser to include and read the given file, e.g.:

```
<!-- Baxter Pedestal -->
<xacro:include filename="$(find baxter_description)/urdf/baxter_base/baxt
er_base.urdf.xacro">
    <xacro:arg name="gazebo" value="${gazebo}"/>
</xacro:include>

<!-- Left End Effector -->
<xacro:include filename="$(find baxter_description)/urdf/left_end_effecto
r.urdf.xacro" />
```

The above shows how to include a XACRO file and how to evaluate constants. Now, let's take a look at `left_end_effector.urdf.xacro`:

In [ ]:

```
%%bash
cd ~/ros_ws/src/baxter/baxter_common/baxter_description/urdf
cat left_end_effector.urdf.xacro
```

The above file includes another xacro file with the macro for Baxter's electric gripper (parametrised with 9 constant values) and then "calls" the macro with the corresponding values for each parameter. To invoke the macro, you just need to add the XML tag name right after `xacro`! The definition of this macro is shown below:

In [ ]:

```
%%bash
cd ~/ros_ws/src/baxter/baxter_common/rethink_ee_description/urdf/electric_grippe
r/
cat rethink_electric_gripper.xacro
```

Note: If you inspect different Baxter's xacro files, you will find out that macros are scarce and there are several hard-coded variables. Robotic companies usually rely on this to circumvent Open Source licence. Indeed, releasing XACRO/URDF files give away the mechanical design of a robot!

To use the `.xacro` file with RViz and Gazebo, you need to convert it to `.urdf`. To do this, you need to execute the following command inside the folder where the main Baxter's xacro file is located; so in a terminal type:

```
cd ~/ros_ws/src/baxter/baxter_common/baxter_description/urdf
xacro --inorder baxter.urdf.xacro > baxter_lab.urdf
```

But you do not need to worry about running this command. `xacro` is invoked everytime you start Baxter's simulation! Later in these labs, you will use XACRO to add a camera sensor, objects and other robotic elements within the simulation and RViz.

Excercise: Moving Baxter

Note: This excercise is the basis for the following labs and your team project! So make sure you complete it.

At this point, you have the 3D model of Baxter and can see it on RViz, but *how do you move Baxter using a node and make it do something?* For this, you will create a simple node to move the robot using Baxter's inverse kinematic solver - you will apply your forward and inverse kinematic, and geometric transformation knowledge to move the left arm of Baxter to a specific point.

In the ROS package, you created above, create a new Python ROS node named `move_baxter.py` and make it executable. This node will command Baxter to move to a neutral (i.e. *home*) position and then move its left arm to a given 3D point and orientation. To start with, you need to import the required libraries:

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

# ROS Python API
import rospy

# Baxter SDK that provides high-level functions to control Baxter
import baxter_interface
from baxter_interface import CHECK_VERSION

#The Header message comprises a timestamp, message sequence id and frame
 id where the message originated from (see http://docs.ros.org/api/std_ms
gs/html/msg/Header.html)
from std_msgs.msg import Header

# Pose is the defacto message to store a 3D point and a quaternion, head
 over ROS API to find out the structure!
# PoseStamped consists of a Header and a Pose messages.
from geometry_msgs.msg import (
    PoseStamped,
    Pose,
    Point,
    Quaternion,
)

# Baxter ROS messages and services that allows communicating with the rob
ot. The Inverse Kinematic function runs inside the robot, and it is close
d-source. To access it, the developers have made available the SolvePosit
ionIK service; hence we need to import it, and also, its "request" messag
e (see: https://github.com/RethinkRobotics/baxter_common/blob/master/baxt
er_core_msgs/srv/SolvePositionIK.srv)
from baxter_core_msgs.srv import (
    SolvePositionIK,
    SolvePositionIKRequest,
)
```

Let's now declare the `Pose` of the left arm as a global variable in Python (you can parse command line arguments).

```
arm = "left"
pose_point = [0.644, 0.0, 0.066]
pose_orientation = [-0.381, 0.923, -0.015, 0.052]
```

The main function will thus initialise the node, subscribe to Baxter's IK service, move to the home position, and finally, move the arm to the above point. To do this, add the following code to your node:

```
def ik_baxter():
    rospy.init_node("rsdk_ik_service_client")

    # Connect to Baxter's service (see: http://sdk.rethinkrobotics.com/wi
ki/API_Reference#Inverse_Kinematics_Solver_Service)
    srv_name = "ExternalTools/" + arm + "/PositionKinematicsNode/IKServic
e"
    rospy.wait_for_service(srv_name)
    srv = rospy.ServiceProxy(srv_name, SolvePositionIK)

    # Setup Baxter SDK
    rs = baxter_interface.RobotEnable(CHECK_VERSION)
    rospy.loginfo("Enabling Baxter")
    # You need to enable Baxter to use it!
    rs.enable()

    # Move to home position
    move_neutral_position()

    # Indicate the SDK which arm we are going to use
    armcmd = baxter_interface.Limb(arm)

    # Generate the Pose message to be passed to the IK service
    pose_arm = generate_posemsg(pose_point, pose_orientation)

    # Call the Inverse Kinematic service
    limb_joints = ik_service(srv, pose_arm)

    # Move arm
    if limb_joints is not None:
        armcmd.move_to_joint_positions(limb_joints)

    # After moving to the desired location, tell the SDK that you stopped
 using the arm, this will disable the arm at the same time
    armcmd.exit_control_mode()

    return 0


if __name__ == '__main__':
    import sys
    sys.exit(ik_baxter())
```

You should be aware that `move_to_joint_positions` deactivates the *collision avoidance* controller inside Baxter; this results in Baxter not "feel" any obstacle while moving the arm(s) - In the next lab, we will go over *motion planners* that doesn't deactivate this function in Baxter. Now, you still need to define three

more functions; `move_neutral_position`, `generate_posemsg` and `ik_service`.

`move_neutral_position` uses a pre-defined function within `baxter_interface` as follows:

```
def move_neutral_position():
    _left_arm = baxter_interface.Limb("left")
    _right_arm = baxter_interface.Limb("right")
    _left_arm.move_to_neutral()
    _right_arm.move_to_neutral()
    _left_arm.exit_control_mode()
    _right_arm.exit_control_mode()
```

Moving Baxter to a neutral position is always a good idea as you want to start any robotic task from a known pose. You need to move both arms to avoid potential collisions between each other.

`generate_posemsg` consists of populating the message you want to send to the IK service. For this, you need to populate the `Header`, `Point` and `Quaternion` messages, wrap `Point` and `Quaternion` into a `Pose` message, and finally `Header` and `Pose` into a `PoseStamped` message.

```
def generate_posemsg(point, orientation):
    # Building up a PoseStamped message request
    hdr = Header(stamp=rospy.Time.now(), frame_id='base')
    pt = Point(x=point[0], y=point[1], z=point[2])
    qt = Quaternion(x=orientation[0], y=orientation[1], z=orientation[2],
 w=orientation[3])
    ps = Pose(position = pt, orientation = qt)
    pose_arm = PoseStamped(header=hdr, pose=ps)

    return pose_arm
```

Finally, the IK solution will allow you to set the end-effector to the desired pose while taking care of the correct angles for the arm. The IK service takes the `PoseStamped` message and the service declaration (see the `ik_baxter()` function). This function will return a potential solution to command each joint in the left arm.

```
def ik_service(srv, pose_arm):
    request = SolvePositionIKRequest()
    request.pose_stamp.append(pose_arm)
    try:
        resp = srv(request)
    except (rospy.ServiceException, rospy.ROSException), e:
        rospy.logerr("Service call failed: %s" % (e,))
        return None

    # Format solution into Limb API-compatible dictionary
    limb_joints = dict(zip(resp.joints[0].name, resp.joints[0].position))
    print limb_joints
    return limb_joints
```

You ROS node is now ready, so start the simulator (if you don't have it running) and run your ROS node. For this particular pose, you should see in the terminal the angles for each joint in radians as follows:

```
{'left_w0': 0.3050684684989051, 'left_w1': 0.8610601980794869, 'left_w2':
 -1.4742358938106646, 'left_e0': -0.33102594774694544, 'left_e1': 1.70336
57530084771, 'left_s0': -1.0051802515115758, 'left_s1': -1.03583779133785
56}
```

In the simulator, you can see in action your ROS node, and you should get Baxter as shown in Figure 4. To further your understanding, explore different 3D points and orientations and answer the following;

- *is there a pose that doesn't return any IK solution? Why?*
- *which is the reference frame for the IK service?*
- *which is the end-effector reference frame being commanded?* (Tip: run `rviz` and add the `tf` plugin to find out this)

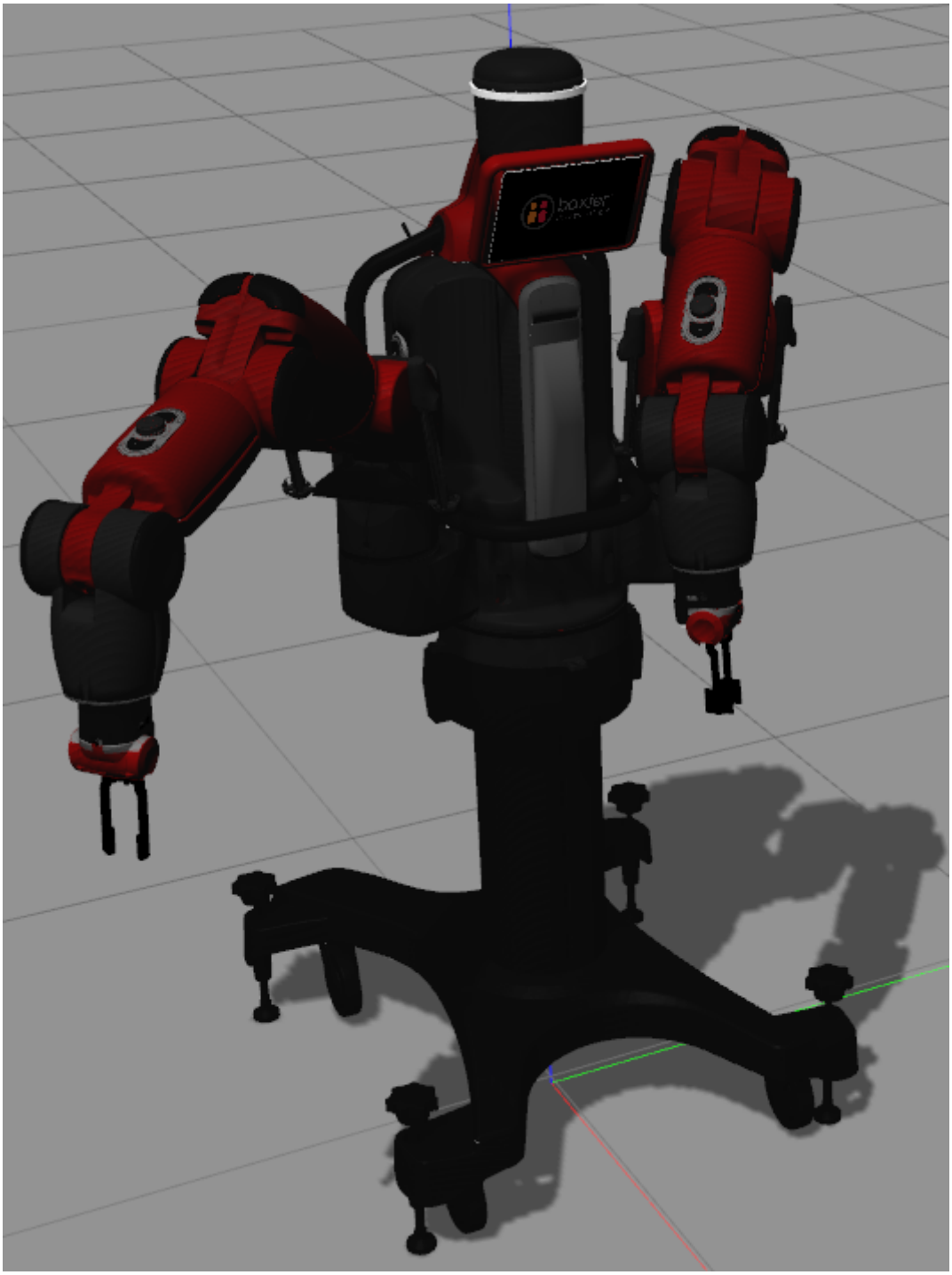

Figure 4: Result after running the ROS control node for Baxter

Let's now rotate Baxter's gripper along its rotation axis (e.g. `z` axis) while maintaining the above pose. To find out where is the `z` axis, run `rviz`! To achieve this, you need to import `tf2_ros`, `numpy` and `transforms3d` (its `quaternions` and `eulerangles` libraries, so type in your ROS node script the following (i.e. make sure to rename it!):

```
import tf2_ros
from transforms3d import quaternions
from transforms3d.derivations import eulerangles
import numpy as np
```

You also need to query the current transformation from base to end-effector (*for this, you should have answered the 3rd questions above*), and rotate a given pose accordingly. To implement this, type the following two functions:

```
    def query_transformation(from_frame, to_frame):
        tfBuffer = tf2_ros.Buffer()
        listener = tf2_ros.TransformListener(tfBuffer)

        trans = None
        quat = None
        try:
            transformation = tfBuffer.lookup_transform(from_frame, to_frame,
    rospy.Time(), rospy.Duration(1.0))
            trans = transformation.transform.translation
            quat = transformation.transform.rotation
        except (tf2_ros.LookupException, tf2_ros.ConnectivityException, tf2_r
    os.ExtrapolationException):
            rospy.logerr("Transformation is not available")

        return trans, quat

    def rotate_pose(point, quat, theta, axis="x"):

        # set quaternion to transforms3d format, i.e. w,x,y,z
        quat_t3d = [quat[3], quat[0], quat[1], quat[2]]
        rot_mat = quaternions.quat2mat(quat_t3d)

        H1 = np.eye(4)
        H1[:3,:3] = rot_mat
        H1[:3,3] = point

        H2 = np.eye(4)
        if axis == "x":
            rot_axis = np.array(eulerangles.x_rotation(theta))
        elif axis == "y":
            rot_axis = np.array(eulerangles.y_rotation(theta))
        elif axis == "z":
            rot_axis = np.array(eulerangles.z_rotation(theta))

        print rot_axis
        H2[:3,:3] = rot_axis

        H3 = np.dot(H1, H2)
        point_out = H3[:3,3]
        quat_t3dout = quaternions.mat2quat(H3[:3,:3])
        # Turn to ROS format, i.e. x, y, z, w
        quat_out = [quat_t3dout[1], quat_t3dout[2], quat_t3dout[3], quat_t3do
    ut[0]]
        rospy.loginfo("Rotated pose: ")
        print point_out, quat_out

        return point_out, quat_out
```

The above functions implement what we did in `tf_listener.py` in previous sections. That is, `query_transformation` returns the current position of the `to_frame` with respect to `from_frame` and `rotate_pose` rotates a pose given by a 3D `point` and a `quaternion`. You task now is to integrate the

above functions into `ik_baxter()` and rotate the gripper `90` degrees (you can use `numpy` to convert degrees into radians with `deg2rad` (https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.deg2rad.html).

- *What happens when you rotate the gripper along the x and y axes?*
- *Can you make Baxter wave with both arms?*

> Tip: you can use the parametric equations for the sine and cosine (https://www.mathopenref.com/coordparamcircle.html) to iterate over a set range of values; you will also need to modify `rotate_pose` to include the translation component while transforming your pose. You should put the latter in a loop!

Appendix - Getting Baxter ROS workspace

If you want to use Baxter's SDK, simulation and related tools in your computer, just copy/paste the following commands in a terminal console in your Ubuntu box. You should make sure you are using Ubuntu 16.04 LTS.

First, you need to install ROS related packages required for simulation. It is assumed you have installed ROS kinetic (see: http://wiki.ros.org/kinetic/Installation (http://wiki.ros.org/kinetic/Installation)).

- Adding Gazebo official Deb repository

```
sudo echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-sta
ble `lsb_release -cs` main" > /etc/apt/sources.list.d/gazebo-stabl
e.list'
```

- Setting up APT keys

```
wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-
key add -
```

- Update and install

```
sudo apt-get update & sudo apt-get -q -y install gazebo7 ros-kinet
ic-rviz ros-kinetic-qt-build ros-kinetic-gazebo-ros-control ros-ki
netic-gazebo-ros-pkgs ros-kinetic-ros-control ros-kinetic-control-
toolbox ros-kinetic-realtime-tools ros-kinetic-ros-controllers ros
-kinetic-xacro python-wstool ros-kinetic-robot-state-publisher ros
-kinetic-tf-conversions ros-kinetic-kdl-parser
```

ROS workspace and cloning Baxter's SDK

`wstool` (https://github.com/vcstools/wstool) is handy when you want to pull code using different version control systems, e.g. git, mercury, svn, etc. Even though Baxter's official repositories use only git, it is good practice to use `wstool` to manage your ROS workspace. In ROS, you can get a "rosinstall" file which contains all the instructions for wstool to pull code from different repos. For RF(H), there's a public repository for Baxter's SDK; you can always try the official repositories, and you can get it by replacing the `wstool merge` URL with https://raw.githubusercontent.com/RethinkRobotics/baxter_simulator/kinetic-devel/baxter_simulator.rosinstall (https://raw.githubusercontent.com/RethinkRobotics/baxter_simulator/kinetic-devel/baxter_simulator.rosinstall). The official Baxter's repos are continuously updated and these labs might not work.

- Getting and pulling Baxter's SDK

```
rosdep update && mkdir -p ~/ros_ws/src && cd ~/ros_ws/src && wstoo
l init . && wstool merge https://raw.githubusercontent.com/gerac8
3/rf_public/master/baxter_materials/baxter_rf.rosinstall && wstool
 update
```

- Running `catkin_make` and install workspace

```
cd ~/ros_ws && source /opt/ros/kinetic/setup.bash && catkin_make &
& catkin_make install && wget https://raw.githubusercontent.com/ge
rac83/rf_public/master/baxter_materials/baxter.sh && chmod u+x bax
ter.sh
```