COMP0129 Transformations, Kinematics, Pick and Place Group Assignment 1

Chiang Hou Nam
15055142
`hou.chiang.20@ucl.ac.uk`

Renjie Zhou
17000505
`renjie.zhou.20@ucl.ac.uk`

Mohamed Ali Shire
17019507
`mohamed.shire.17@ucl.ac.uk`

10th Feb, 2021

Module Lead:	Dimitrios Kanoulas
Contact e-mail:	`d.kanoulas@ucl.ac.uk`
Department:	Department of Computer Science
Submission Date:	10th of February 2021

Contents

Question 1

In general, for a homogeneous transformation matrix ${}^{A}T_{B}$ we have:

$$ {}^{A}T_{B} = \begin{bmatrix} {}^{A}R_{B} & {}^{A}d_{B} \\ 0_{1\times 3} & 1 \end{bmatrix} \tag{1} $$

where ${}^{A}R_{B}$ is the rotation matrix and ${}^{A}d_{B}$ is the translation from frame A to frame B. In this question we investigate the transformation between the frames in the Figure 1

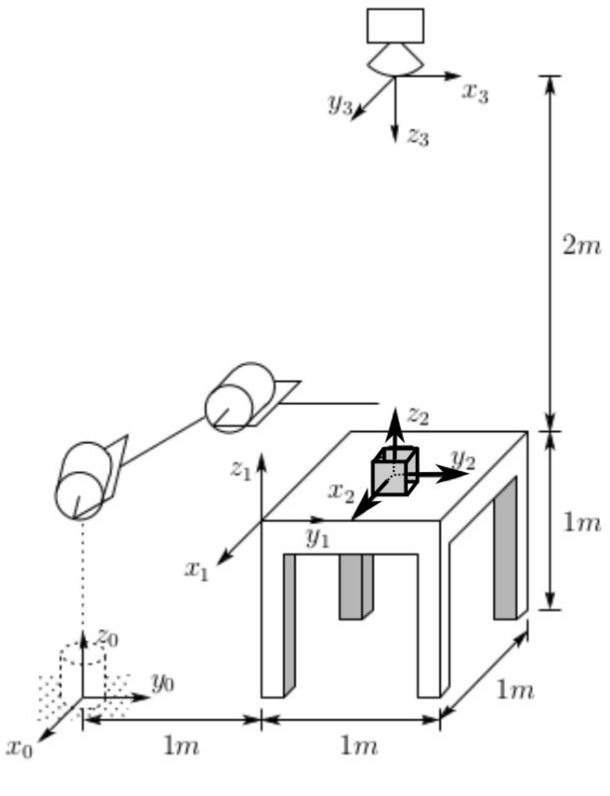


Figure 1: *Robot and Environment configuration 1*

1.1

From Figure 1, we can derive the homogeneous transformations ${}^{0}T_{1}, {}^{0}T_{2}, {}^{0}T_{3}$ as follows:

- From frame 0 to frame 1, we can see that there is only a pure translation of 1m in both positive y and z direction:

$$
\begin{aligned}
{}^0T_1 &= \begin{bmatrix} {}^0R_1 & {}^0d_1 \\ 0_{1\times 3} & 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned} \tag{2}
$$

- From frame 0 to frame 2, we can see that there is only a pure translation of -0.5m, 1.5m, 1.1m in x, y, z direction respectively:

$$
\begin{aligned}
{}^0T_2 &= \begin{bmatrix} {}^0R_2 & {}^0d_2 \\ 0_{1\times 3} & 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & -0.5 \\ 0 & 1 & 0 & 1.5 \\ 0 & 0 & 1 & 1.1 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned} \tag{3}
$$

- From frame 0 to frame 3, we can see that the frame is first rotated $\frac{\pi}{2}$ rad clockwise around z axis, then rotate π rad around y axis. After these two rotation, we have a translation of -0.5m, 1.5m, 3m in x, y, z direction respectively. Using Tait-Bryan Rotation:

$$
{}^0T_3 = \begin{bmatrix} {}^0R_3 & {}^0d_3 \\ 0_{1\times 3} & 1 \end{bmatrix} \tag{4}
$$

Then:

$$
\begin{aligned}
{}^0R_3 &= R_z\left(-\frac{\pi}{2}\right) R_y(\pi) \\
&= \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \\
&= \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}
\end{aligned} \tag{5}
$$

Now we substitute (5) back to (4), we have:

$$
{}^0T_3 = \begin{bmatrix} 0 & 1 & 0 & -0.5 \\ 1 & 0 & 0 & 1.5 \\ 0 & 0 & -1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}
$$

1.2

From frame 2 to frame 3, we have the frame is first rotated $\frac{\pi}{2}$ rad around z axis, then rotate π rad around y axis, followed by a translation of 1.9m in the positive z direction only. So we have the transformation matrix ${}^2T_3$:

$$\begin{aligned} {}^2T_3 &= \begin{bmatrix} {}^2R_3 & {}^2d_3 \\ 0_{1\times3} & 1 \end{bmatrix} \\ &= \begin{bmatrix} {}^0R_3 & {}^2d_3 \\ 0_{1\times3} & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1.9 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{7}$$

1.3

We can also get (7) by using (3) and (4):

$$\begin{aligned} {}^0T_3 &= {}^0T_2\, {}^2T_3 \\ {}^2T_3 &= {}^0T_2^{-1}{}^0T_3 \\ {}^2T_3 &= \begin{bmatrix} 1 & 0 & 0 & 0.5 \\ 0 & 1 & 0 & -1.5 \\ 0 & 0 & 1 & -1.1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & -0.5 \\ 1 & 0 & 0 & 1.5 \\ 0 & 0 & -1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1.9 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{8}$$

which is the same as (7).

1.4

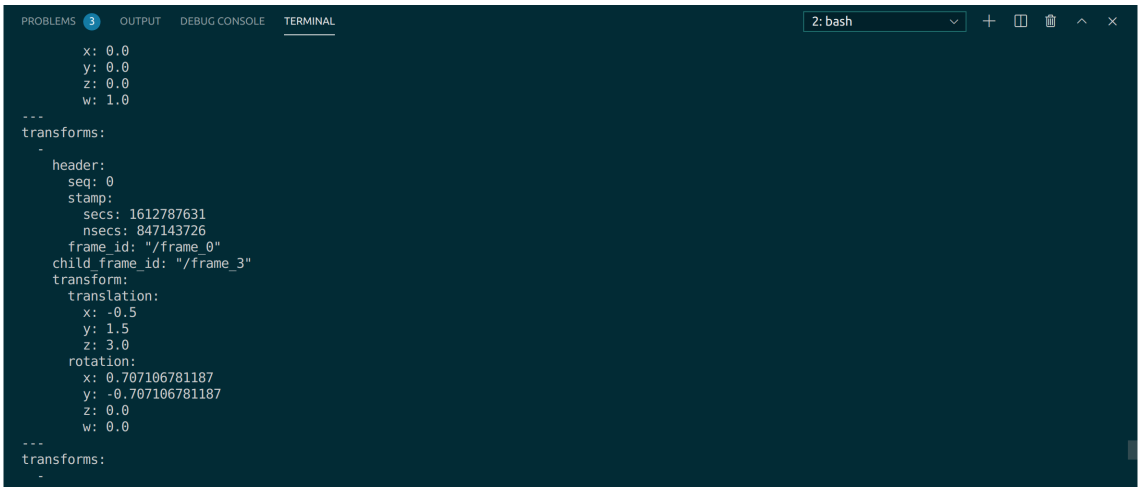
To generate all frame transformations ${}^0T_1, {}^0T_2, {}^0T_3$, frame0 to frame3 are first initialized in the function `CW1::initParams()`. After that, in the function `CW1::cw1Q1GenFrames()` we do two things for every transformation:

- Set the values of the rotation matrices and convert them to quaternions.
- Set the values of the coordinates of the frames.

Then we covert them to TFs using `tf::StampedTransform()`

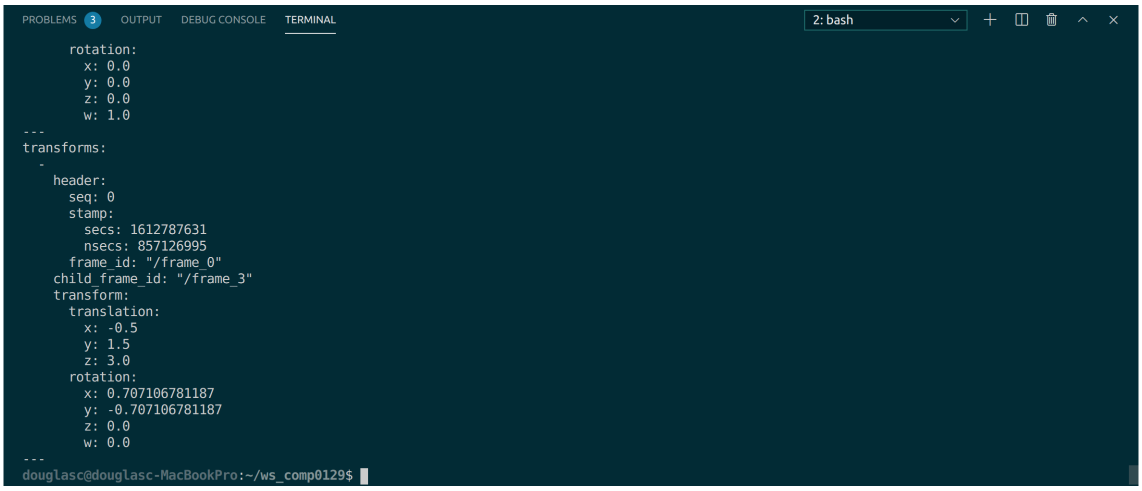
1.5

In this part, the frames in Q1.4 are published using the function `sendTransform()` in 100Hz frame rate using `ros::Rate loop_rate(100)`. The frame rate is then verified by using a new terminal and calling `rostopic echo /tf`. The result is shown in Figure 2 and 3.



```
        x: 0.0
        y: 0.0
        z: 0.0
        w: 1.0
---
transforms:
  -
    header:
      seq: 0
      stamp:
        secs: 1612787631
        nsecs: 847143726
      frame_id: "/frame_0"
    child_frame_id: "/frame_3"
    transform:
      translation:
        x: -0.5
        y: 1.5
        z: 3.0
      rotation:
        x: 0.707106781187
        y: -0.707106781187
        z: 0.0
        w: 0.0
---
transforms:
  -
```

Figure 2: *Verifying frame rate*

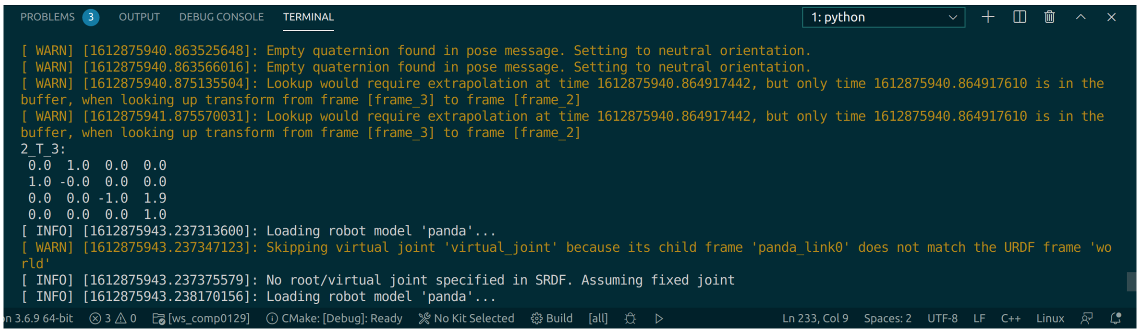


```
      rotation:
        x: 0.0
        y: 0.0
        z: 0.0
        w: 1.0
---
transforms:
  -
    header:
      seq: 0
      stamp:
        secs: 1612787631
        nsecs: 857126995
      frame_id: "/frame_0"
    child_frame_id: "/frame_3"
    transform:
      translation:
        x: -0.5
        y: 1.5
        z: 3.0
      rotation:
        x: 0.707106781187
        y: -0.707106781187
        z: 0.0
        w: 0.0
---
douglasc@douglasc-MacBookPro:~/ws_comp0129$
```

Figure 3: *Verifying frame rate*

We can see that the difference of the time stamp: $(1612787631847143726 - 1612787631857126995) \times 10^{-9} = 9.98 \times 10^{-3} secs \longrightarrow 100.2Hz$ which agrees with the frame rate (100Hz) requested.

After publishing the TFs, we can then proceed to get the transformation ${}^2T_3$ by using the function `listener_.lookupTransform(frame_2_, frame_3_, ros::Time(0), transf_23_);`. This give us the transformation straight away. Then the rotation matrix and the frame coordinate can be extracted by using the functions `getBasis()` and `getOrigin()` respectively. The values are then placed in the relevant entries in a 4×4 transformation matrix. The printed result is shown in Figure 4



```
[ WARN] [1612875940.863525648]: Empty quaternion found in pose message. Setting to neutral orientation.
[ WARN] [1612875940.863566016]: Empty quaternion found in pose message. Setting to neutral orientation.
[ WARN] [1612875940.875135504]: Lookup would require extrapolation at time 1612875940.864917442, but only time 1612875940.864917610 is in the
buffer, when looking up transform from frame [frame_3] to frame [frame_2]
[ WARN] [1612875941.875570031]: Lookup would require extrapolation at time 1612875940.864917442, but only time 1612875940.864917610 is in the
buffer, when looking up transform from frame [frame_3] to frame [frame_2]
2_T_3:
 0.0  1.0  0.0  0.0
 1.0 -0.0  0.0  0.0
 0.0  0.0 -1.0  1.9
 0.0  0.0  0.0  1.0
[ INFO] [1612875943.237313600]: Loading robot model 'panda'...
[ WARN] [1612875943.237347123]: Skipping virtual joint 'virtual_joint' because its child frame 'panda_link0' does not match the URDF frame 'wo
rld'
[ INFO] [1612875943.237375579]: No root/virtual joint specified in SRDF. Assuming fixed joint
[ INFO] [1612875943.238170156]: Loading robot model 'panda'...
```

Figure 4: *Transformation* ${}^2T_3$

Please note that since we use `std::cout << std::setprecision(1) << std::fixed;` to fix the numbers to have 1 decimal place only, any trivial numbers (e.g. at the magnitude of 10^{-16}) is outputed as 0.0.
As the frames are published, we can also visualize the frames in RViz as shown in Figure 5.

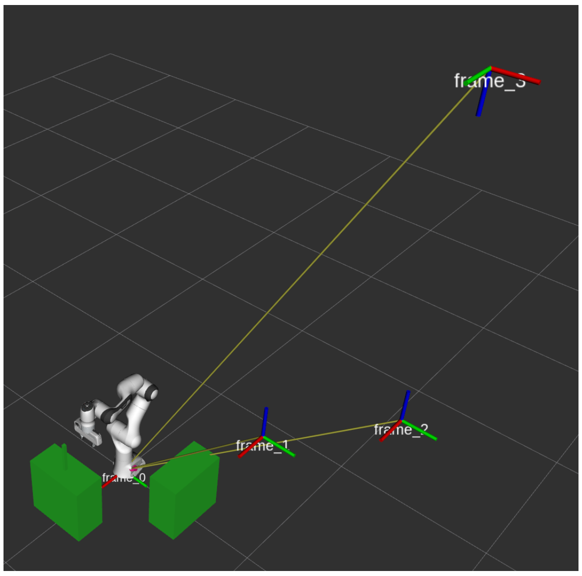


Figure 5: *Visualization of frames in RViz*

Question 2

2.1

For this question, a `JointState` message from `sensor_message` and add the home positions, 0.0, 0.0, 0.0, 0.0, 0.0, -1.5708, 0.0, 1.5708, 0.785398, to the `position` field of the message, where the first two fields are the positions for **panda_finger_joint1** and **panda_finger_joint2**. Then, the message is published using `joint_pub` which is the input to the function, so that the robots can move to the home position after pressing 'h'. The homed robot looks like following:

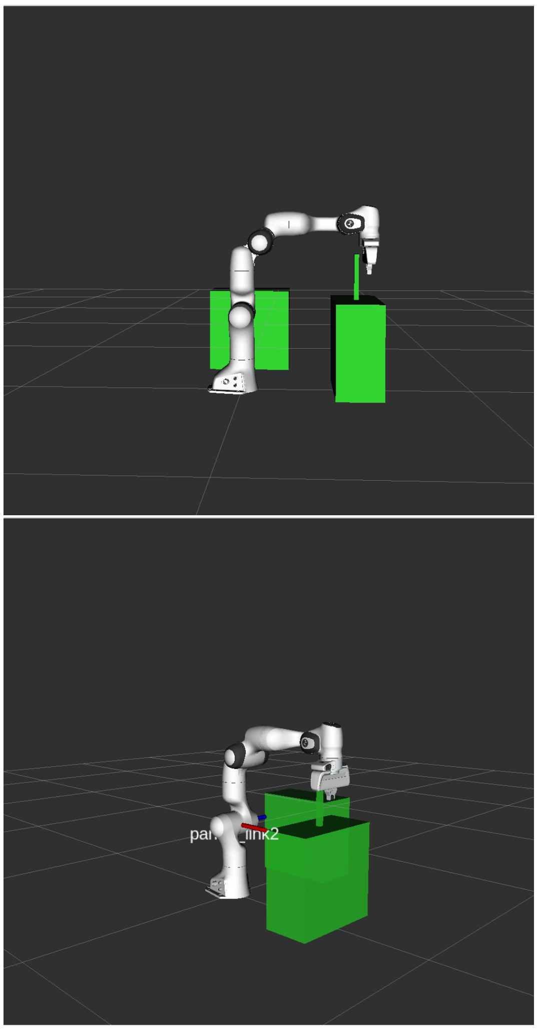


Figure 6: *Home position of the robot*

2.2

Printing by pressing 'p'

For this question, `tf listener` is used to look up for the transformation from frame `panda_hand` to frames `panda_link0` and `panda_link2` and store the transformation in `StampedTransform`. The transformation obtained from the `tf listener` contains transition vector and rotation in quaternion format between the two given frames. Afterwards, `tf::Matrix3x3` is created from the rotation in quaternion to obtain the rotation matrix, namely `rotM2Base` and `rotM2Elbow`. Subsequently, with the respective rotation matrices and transition vector, the transformation matrices are obtained and printed in the format given in question 1.5.

"Bowing" by pressing 'k'

For this part of the question, after the robot reaches the home position, a new position array with 1.5707 radians to `panda_joint2` is published to `joint_pub`, resulting in robot rotating counter-clockwise around the z-axis of `panda_link2`. The final position after rotating looks like this:

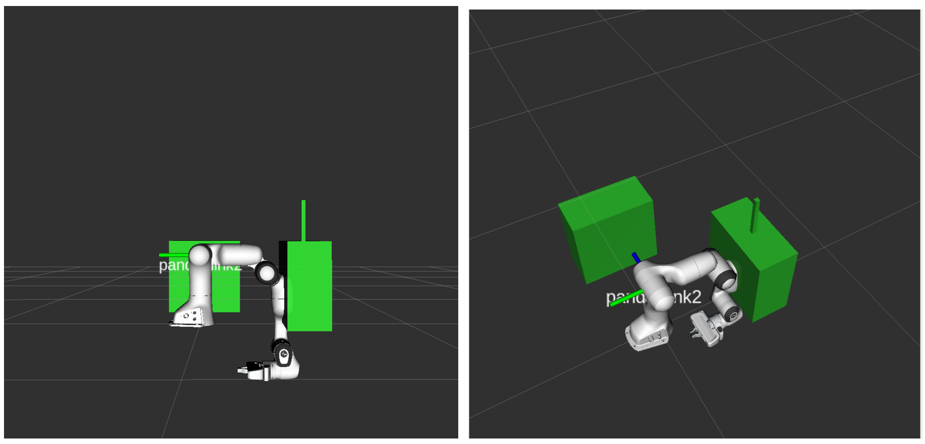


Figure 7: *Robot position after rotating 90 degrees around panda_link2*

Pose of panda_hand after rotation

The pose of `panda_hand` with respect to `panda_link0` after pressing 'k', i.e. rotation, is calculated using forward kinematics. Transformation between every two frames (i.e. frame i-1 and frame i) is obtained from the rotation matrix and transitional vector, where the rotation matrix is obtained from the quaternion rotation. Both rotation in the quaternion format and the transitional vector between every two adjacent frames can be obtained from the transform messages by using `tf listener`. By using the above-described methodology, the transformation matrices from frame i to frame i-1, notated as ${}^{i-1}T_i$, are as such:

$$
{}^{0}T_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.333 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{1}T_2 = \begin{bmatrix} 9.63268e{-05} & -1 & 1.66533e{-16} & 0 \\ 4.89636e{-12} & 5.55112e-16 & 1 & 0 \\ -1 & -9.63268e{-05} & 4.89631e{-12} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{2}T_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 4.89664e{-12} & -1 & -0.316 \\ 0 & 1 & 4.89664e{-12} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{3}T_4 = \begin{bmatrix} -3.67321e{-06} & 1 & 1.11022e{-16} & 0.0825 \\ -4.89642e{-12} & 1.11022e-16 & -1 & 0 \\ -1 & -3.67321e{-06} & 4.89642e{-12} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{4}T_5 = \begin{bmatrix} 1 & -0 & 0 & -0.0825 \\ 0 & 4.89664e{-12} & 1 & 0.384 \\ -0 & -1 & 4.89664e{-12} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{5}T_6 = \begin{bmatrix} -3.67321e{-06} & -1 & -1.11022e{-16} & 0 \\ 4.89642e{-12} & 1.11022e{-16} & -1 & 0 \\ 1 & -3.67321e{-06} & 4.89642e{-12} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{6}T_7 = \begin{bmatrix} 0.707107 & -0.707107 & 1.11022e{-16} & 0.088 \\ 3.46245e{-12} & 3.46234e{-12} & -1 & 0 \\ 0.707107 & 0.707107 & 4.89653e{-12} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{7}T_8 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.107 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{8}T_h = \begin{bmatrix} 0.707107 & 0.707107 & 0 & 0 \\ -0.707107 & 0.707107 & -0 & 0 \\ -0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

where ${}^{8}T_h$ is the transformation matrix from `panda_hand` frame to `panda_link8` frame. Hence, the final pose of `panda_hand` to `panda_link0` frame is calculated as such:

$$
\begin{aligned}
{}^{0}T_h &= {}^{0}T_1\, {}^{1}T_2\, {}^{2}T_3\, {}^{3}T_4\, {}^{4}T_5\, {}^{5}T_6\, {}^{6}T_7\, {}^{7}T_8\, {}^{8}T_h \\
&= \begin{bmatrix} 0 & 0 & -1 & 0.2915520 \\ 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & -0.2214722 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$

where all values of which the absolute values are smaller than 0.001 are taken as 0 as required by the question.

Meanwhile, after rotation the robot position by pressing 'k', pressing 'p' to print the final pose of `panda_hand` in `panda_link0` frame using `tf listener` lookup function directly outputs the following matrix to the screen:

$$
\begin{aligned}
&0\_T\_h = \\
&\begin{bmatrix} 0 & 0 & -1 & 0.29155 \\ 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & -0.221472 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$

which is the same as the theoretically calculated final pose.

2.3

For this question, `tf listener` is used again to look up for the transformation from frame `panda_hand` to frame `panda_link0`. This time, transition vector and rotation in quaternion format are extracted directly from the transformation and put into the `PoseStamped` for publishing into the `ee_pose_pub` publisher, since `PoseStamped` contains two fields: `Point` and `Quaternion` from `geometry_msgs`, namely position and orientation.

Question 3

3.1

The objective of this question was to publish a message that contains the geometric information of the object that is being manipulated in the simulations. This was accomplished using a variety of different methods.

First, It was important to distinguish the state of the simulation, whether the robot was in the pregrasping, grasping or ungrasping position. This was done by monitoring the joint state messages of the finger joints.

If the previous finger state is greater than the current finger state, the robot is about to grasp the object and is currently in its pregrasping position. While the finger state = 0, the object is being grasped. If the previous finger state is less than the current finger state, the object is being ungrasped.

While the robot is in pregrasping, the object would be in its initial position (x:0.5 y:0 z:0.5).

While the robot is grasping the object and in motion, we no longer know the position of the object with respect to the panda link0 frame. However we do have the position of the object with respect to the hand/panda link8 frames. The pose of the object in the hand/panda link8 frame while in motion is just an offset in the z direction. As the offset from the panda link8 frame to the palm of the hand was already defined, the pose of the object was tracked with respect to the panda link8 frame.

During grasping, the position of the object in the panda link8 frame is constant. As the robot is in motion, the pose and orientation of the panda link8 frame is continuously changing. The transform from panda link8 frame to panda link0 frame is obtained using a tf listener, this transform is then applied at every time-step to the position of the object in the panda link8 frame. This returns the real-time pose of the object in panda link0 frame. The orientation of the object during grasping is also the same as the orientation of the panda hand.

If the robot has been detected to be in the ungrasping position, this means that the object has reached its final destination (x:0.0 y:0.5 z:0.5).

3.2

To find if the robot is within grasping range of the robot, the euclidean distance between the end effector and object needs to be calculated. subscribers were written to listen to the end effector and object pose rostopics.

Once the pose of the end effector and object is obtained, the euclidean distance between the two is calculated. Then using a conditional statement to check if this distance is within the specified range (0.2m), a Boolean message is published to the topic ee_obj_close as true, otherwise it is published as false.

when pressing e, the state of ee_obj_close is printed alongside the distance between the end effector and the object.

3.3

3.3.1

For this question, object position with respect to panda_link8 is calculated using fixed. Then, object pose with respect to panda_link0 is calculated using forward kinematics by multiplying the transformation matrix ${}^{0}T_{8}$ and filled into the `PoseStamped` message for publishing into the topic `"object_pose"`.

3.3.2

In this question, we subscribed to both "`object_pose`" and "`ee_pose`" to obtain the object pose and hand pose in real time, and use the position vectors to calculate the Euclidean distance and set the boolean value `ee_obj_close` accordingly before publishing it.
We then printed the boolean value of `ee_obj_close` and the euclidean distance from the hand to the object accordingly in function `cw1Q3EeObjClosePrint`.

3.3.3

Problems that we did not manage to solve:
Because of the way we implemented the function to find and pose the object position in the previous parts, if we obtain the object positions from listening to "`object_pose`", when 'h' is **not pressed** to home the robot, everything is fine and object's pre-grasping positions can be correctly identified, i.e. (0.5, 0, 0.5) and 'r', 's' or 't' commands can be executed successfully; however, when 'h' is **pressed** to home the robot, the finger joints are set to 0 and the object pose is calculated using inverse kinematics, which will result in slight difference in the final object positions from the exact (0.5, 0, 0.5), resulting in inaccurate initialisation of the grasp in pre-grasping stage, potentially resulting in collision between the hand and the objects in the scene.

Hence, to avoid any potential collision due to inaccuracy in the calculated object positions, we hard coded the static positions of the object pre-grasping i.e. (0.5, 0, 0.5), instead of obtaining the object positions from "`object_pose`".

Besides, as we didn't have enough time to figure out the correct post-grasping retreat position when attempting to grasp from top, we implemented the 'g' command, i.e. grasp function, with only grasping from side position.

Apart from that, everything else is solved in the code.

3.3.4

For this question, we implemented the code to place the object on table2, and check for the difference between the object's current position in z-axis and the sum of table height and half the object height (i.e. $0.4 + 0.1 = 0.5$) is within a small tolerance (0.0001 in our code) before publishing a message indicating the touch of the object with table2.