Assignment 2: Rotations and ROS TF2

Author: dilip kumar
Date: Feb 1, 2026

1.0 About this code

This is a ROS 2 package designed for a robotics assignment focused on 3D Coordinate Transformations.

Key Components

1. assignment2.py (The Core Task)
    - Contains empty function stubs that we need to implement.
    - Functions include: euler_rotation_matrix, quaternion_rotation_matrix, quaternion_multiply, rotate, inverse_rotation, and transform_pose.
    - These functions implement the mathematical logic for rotating and transforming points/poses in 3D space.
2. coordinates.py (The Consumer Node)
    - A ROS 2 node named minimal_subscriber.
    - Subscribes to dummy_objects (Pose of an object in the camera frame).
    - Listens to TF2 transforms (from single_rrbot_camera_link to world).
    - Uses our code (transform_pose from assignment2.py) to transform the object's pose into the world frame.
    - Publishes the result to dummy_objects_world.
3. dummy_object_detector.py (The Mock Data Source)
    - A ROS 2 node named dummy_object_detector.
    - Publishes a static Pose to dummy_objects every 0.5 seconds.
    - Simulates a camera detecting an object.
4. assignment2.launch.py (The Launcher)
    - Launches the entire system:
    - robot_state_publisher (loads single_rrbot.urdf).
    - dummy_object_detector (starts sending data).
    - dummy_joint_states (from dummy_sensors package).
    - coordinates (runs our transformation logic).

Workflow

1. Mock Data: dummy_object_detector publishes a pose.
2. Processing: coordinates receives the pose, looks up the robot's current transform, and calls our transform_pose function.

3. Output: The transformed pose is published to dummy_objects_world.

Next Step: We need to implement the math functions in assignment2.py to make the coordinates node work correctly.

Connect to ROS

```
docker rm -f en613_ros2_jazzy
docker run   -p 6080:80   -p 8888:8888   -v "$pwd/assignment2:/home/ubuntu/EN613"
--name en613_ros2_jazzy   tiryoh/ros2-desktop-vnc:jazzy
```

Then launch Ubuntu in browser http://127.0.0.1:6080/

2.0 Failure: colcon build --packages-select assignment2

This command throws the following error.

```
$ colcon build --packages-select assignment2
Starting >>> assignment2
/usr/lib/python3/dist-packages/setuptools/dist.py:744:
SetuptoolsDeprecationWarning: Invalid dash-separated options
!!


********************************************************************************
        Usage of dash-separated 'script-dir' will not be supported in future
        versions. Please use the underscore name 'script_dir' instead.

        This deprecation is overdue, please update your project and remove
deprecated
        calls to avoid build errors in the future.

        See https://setuptools.pypa.io/en/latest/userguide/declarative_config.html
for details.

********************************************************************************
```

```
!!
  opt = self.warn_dash_deprecation(opt, section)
/usr/lib/python3/dist-packages/setuptools/dist.py:744:
SetuptoolsDeprecationWarning: Invalid dash-separated options
!!


********************************************************************************
        Usage of dash-separated 'install-scripts' will not be supported in future
        versions. Please use the underscore name 'install_scripts' instead.

        This deprecation is overdue, please update your project and remove
deprecated
        calls to avoid build errors in the future.

        See https://setuptools.pypa.io/en/latest/userguide/declarative_config.html
for details.

********************************************************************************

!!
  opt = self.warn_dash_deprecation(opt, section)
--- stderr: assignment2
/usr/lib/python3/dist-packages/setuptools/dist.py:744:
SetuptoolsDeprecationWarning: Invalid dash-separated options
!!


********************************************************************************
        Usage of dash-separated 'script-dir' will not be supported in future
        versions. Please use the underscore name 'script_dir' instead.

        This deprecation is overdue, please update your project and remove
deprecated
        calls to avoid build errors in the future.

        See https://setuptools.pypa.io/en/latest/userguide/declarative_config.html
for details.

********************************************************************************

!!
  opt = self.warn_dash_deprecation(opt, section)
---
Finished <<< assignment2 [2.39s]

Summary: 1 package finished [2.60s]
```

```
  1 package had stderr output: assignment2
```

Fix setup.cfg

Old

```
[develop]
script-dir=$base/lib/assignment2
[install]
install-scripts=$base/lib/assignment2
```

New

```
[develop]
script_dir=$base/lib/assignment2
[install]
install_scripts=$base/lib/assignment2
```

After fix it ran successfully

```
$ colcon build --packages-select assignment2
Starting >>> assignment2
Finished <<< assignment2 [2.36s]

Summary: 1 package finished [2.57s]
```

After that run the following command to load the env variables etc.

```
$ source install/setup.bash
```

3.0 assignment2/assignment2.py

Euler_rotation_matrix

The euler_rotation_matrix method takes three separate angles—roll, pitch, and yaw—and combines them into a single mathematical tool called a "rotation matrix" that represents an object's full 3D orientation.

Here is the breakdown:

1. The Inputs:
    - Roll (alpha): How much to rotate around the X-axis (like a plane doing a barrel roll).
    - Pitch (beta): How much to rotate around the Y-axis (like a plane pointing its nose up or down).
    - Yaw (gamma): How much to rotate around the Z-axis (like a plane turning left or right).
2. The Process:
    - The method first calculates the rotation for each axis individually. It figures out what the rotation looks like if we only rolled, only pitched, or only yawed.
    - Then, it combines these three individual rotations into one final result by multiplying them together in a specific order (usually Z, then Y, then X).
3. The Output:
    - It returns a 3x3 matrix (a grid of 9 numbers). This matrix is a standard way in robotics to describe exactly how an object is oriented in space relative to a starting point. You can later use this matrix to mathematically "rotate" points or vectors to match this new orientation.

Following is implementation of this method.

```
def euler_rotation_matrix(alpha: float,beta: float,gamma:float) -> np.ndarray:
    """
    15 pts
    Creates a 3x3 rotation matrix in 3D space from euler angles

    Input
    :param alpha: The roll angle (radians)
    :param beta: The pitch angle (radians)
    :param gamma: The yaw angle (radians)

    Output
    :return: A 3x3 element matix containing the rotation matrix

    """

    # Precompute sines and cosines
    ca = np.cos(alpha)
    sa = np.sin(alpha)
    cb = np.cos(beta)
    sb = np.sin(beta)
    cg = np.cos(gamma)
    sg = np.sin(gamma)

    # Rotation around X-axis (Roll)
    Rx = np.array([
        [1, 0, 0],
        [0, ca, -sa],
        [0, sa, ca]
    ])

    # Rotation around Y-axis (Pitch)
    Ry = np.array([
        [cb, 0, sb],
        [0, 1, 0],
        [-sb, 0, cb]
    ])

    # Rotation around Z-axis (Yaw)
    Rz = np.array([
        [cg, -sg, 0],
        [sg, cg, 0],
        [0, 0, 1]
    ])

    # Combined Rotation R = Rz * Ry * Rx
    R = Rz @ Ry @ Rx

    return R
```

Quaternion_rotation_matrix

The quaternion_rotation_matrix method takes a Quaternion—a 4-number system used to describe rotation without running into "gimbal lock" issues—and converts it into a standard 3x3 Rotation Matrix.

Here is the breakdown:

1. The Input:

- Q: A list of 4 numbers (q_0, q_1, q_2, q_3). In this system, q_0 is the "scalar" (real) part, and the other three are the "vector" (imaginary) parts corresponding to the X, Y, and Z axes.

2. The Process:
    - The method uses a specific mathematical formula to translate these 4 quaternion numbers into the 9 numbers required for a rotation matrix.
    - It involves squaring the input numbers and multiplying them in various combinations to fill the grid.
3. The Output:
    - It returns a 3x3 matrix. This is the same kind of output as the euler_rotation_matrix method, but it comes from a more robust source (a quaternion) that is better for handling complex 3D movements in robotics.

Following is implementation of this method.

```
def quaternion_rotation_matrix(Q: np.ndarray) -> np.ndarray:
    """
    15 pts
    Creates a 3x3 rotation matrix in 3D space from a quaternion.

    Input
    :param q: A 4 element array containing the quaternion (q0,q1,q2,q3)

    Output
    :return: A 3x3 element matix containing the rotation matrix

    """

    # Extract the values from Q
    q0 = Q[0]
    q1 = Q[1]
    q2 = Q[2]
    q3 = Q[3]

    # First row of the rotation matrix
    r00 = 2 * (q0 * q0 + q1 * q1) - 1
    r01 = 2 * (q1 * q2 - q0 * q3)
    r02 = 2 * (q1 * q3 + q0 * q2)

    # Second row of the rotation matrix
    r10 = 2 * (q1 * q2 + q0 * q3)
    r11 = 2 * (q0 * q0 + q2 * q2) - 1
    r12 = 2 * (q2 * q3 - q0 * q1)

    # Third row of the rotation matrix
    r20 = 2 * (q1 * q3 - q0 * q2)
    r21 = 2 * (q2 * q3 + q0 * q1)
    r22 = 2 * (q0 * q0 + q3 * q3) - 1

    # 3x3 rotation matrix
    rot_matrix = np.array([[r00, r01, r02],
                           [r10, r11, r12],
                           [r20, r21, r22]])

    return rot_matrix
```

Quaternion_multiply

The quaternion_multiply method is essentially "rotation addition" for 3D space.

1. The Goal:
    - If we rotate an object by Rotation A (e.g., turn 90° right) and then by Rotation B (e.g., tilt 45° up), we end up in a specific final orientation.
    - This method calculates a single rotation (Rotation C) that represents that entire combined movement in one step.
2. The Process:
    - It takes two quaternions (which are just math representations of those two separate rotations) and mathematically "multiplies" them.
    - This merges them into a new, single quaternion.
3. Key Rule:
    - Order matters! Just like turning right then looking up is different from looking up then turning right, the order in which we multiply these quaternions changes the result.

Following is implementation of this method.

```
def quaternion_multiply(Q0: np.ndarray,Q1: np.ndarray) -> np.ndarray:
    """
    15 pts
    Multiplies two quaternions.

    Input
    :param Q0: A 4 element array containing the first quaternion (q01,q11,q21,q31)
    :param Q1: A 4 element array containing the second quaternion (q02,q12,q22,q32)

    Output
    :return: A 4 element array containing the final quaternion (q03,q13,q23,q33)

    """
    # Extract values from Q0
    w0 = Q0[0]
    x0 = Q0[1]
    y0 = Q0[2]
    z0 = Q0[3]

    # Extract values from Q1
    w1 = Q1[0]
    x1 = Q1[1]
    y1 = Q1[2]
    z1 = Q1[3]

    # Computer the product of the two quaternions, term by term
    w = w0 * w1 - x0 * x1 - y0 * y1 - z0 * z1
    x = w0 * x1 + x0 * w1 + y0 * z1 - z0 * y1
    y = w0 * y1 - x0 * z1 + y0 * w1 + z0 * x1
    z = w0 * z1 + x0 * y1 - y0 * x1 + z0 * w1

    # Create a 4 element array containing the final quaternion
    final_quaternion = np.array([w, x, y, z])
```

Quaternion_to_euler

The quaternion_to_euler method is the reverse of what we discussed earlier. It converts a Quaternion back into Euler Angles (Roll, Pitch, and Yaw).

Here is the breakdown:

1. The Goal:
    a. Quaternions are great for math and computers, but they are hard for humans to visualize.
    b. Euler angles (Roll, Pitch, Yaw) are much more intuitive (e.g., "tilt up 30 degrees").
    c. This method translates the computer-friendly quaternion back into human-friendly angles.
2. The Process:
    - It uses a set of standard trigonometry formulas (involving atan2, asin, etc.) to extract the three angles from the four quaternion components.
3. The Output:
    - It returns a list of 3 numbers: Roll, Pitch, and Yaw (in radians).

Following is implementation of this method.

```
def quaternion_to_euler(Q: np.ndarray) -> np.ndarray:
    """
    15 pts
    Takes a quaternion and returns the roll, pitch yaw array.

    Input
    :param Q0: A 4 element array containing the quaternion (q01,q11,q21,q31)

    Output
    :return: A 3 element array containing the roll,pitch, and yaw
(alpha,beta,gamma)

    """

    # Extract values from Q
    w = Q[0]
    x = Q[1]
    y = Q[2]
    z = Q[3]

    # Roll (x-axis rotation)
    sinr_cosp = 2 * (w * x + y * z)
    cosr_cosp = 1 - 2 * (x * x + y * y)
    roll = np.arctan2(sinr_cosp, cosr_cosp)

    # Pitch (y-axis rotation)
    sinp = 2 * (w * y - z * x)
    if abs(sinp) >= 1:
        pitch = np.sign(sinp) * np.pi / 2 # use 90 degrees if out of range
    else:
        pitch = np.arcsin(sinp)

    # Yaw (z-axis rotation)
    siny_cosp = 2 * (w * z + x * y)
    cosy_cosp = 1 - 2 * (y * y + z * z)
    yaw = np.arctan2(siny_cosp, cosy_cosp)

    return np.array([roll, pitch, yaw])
```

Rotate

The rotate method moves a specific point in 3D space to a new position based on rotation angles.

Here is the breakdown:

1. The Inputs:

    - p1: The original location of the point (X, Y, Z coordinates).
    - alpha, beta, gamma: The three angles (Roll, Pitch, Yaw) that define how much to rotate.
2. The Process:
    - First, it bundles those three angles into a single Rotation Matrix (using the logic from the first method we wrote).
    - Then, it takes that matrix and "multiplies" it by the point. In math terms, this applies the rotation to the point, effectively swinging it around the center to its new spot.
3. The Output:
    - It returns the new coordinates (X, Y, Z) of the point after the rotation.

Following is implementation of this method.

```
def rotate(p1: np.ndarray,alpha: float,beta: float,gamma: float) -> np.ndarray:
    """
    15 pts
    Rotates a point p1 in 3D space to a new coordinate system.

    Input
    :param p1: A 3 element array containing the original (x1,y2,z1) position]
    :param alpha: The roll angle (radians)
    :param beta: The pitch angle (radians)
    :param gamma: The yaw angle (radians)

    Output
    :return: A 3 element array containing the new rotated position (x2,y2,z2)

    """

    # Get the rotation matrix
    R = euler_rotation_matrix(alpha, beta, gamma)

    # Rotate the point
    p2 = R @ p1

    return p2
```

Inverse_rotation

The inverse_rotation method is the undo button for the rotate method.

Here is the breakdown:

1. The Goal:

    - If we took a point and rotated it to a new spot, this method figures out where it originally came from.
    - It reverses the rotation defined by the angles.
2. The Process:
    - It calculates the Rotation Matrix for the given angles (just like before).
    - Then, it finds the Transpose of that matrix. For rotation matrices, the "transpose" (flipping rows and columns) is mathematically the same as the "inverse" (the reverse operation).
    - Finally, it multiplies this "reversed" matrix by the current point to send it back to its starting position.
3. The Output:
    - It returns the original coordinates (X, Y, Z) of the point before it was rotated.

Following is implementation of this method.

```
def inverse_rotation(p2: np.ndarray,alpha: float,beta: float,gamma: float) ->
np.ndarray:
    """
    15 pts
    Inverse rotation from a point p2 in 3D space to the original coordinate system.

    Input
    :param p: A 3 element array containing the new rotated position (x2,y2,z2)
    :param alpha: The roll angle (radians)
    :param beta: The pitch angle (radians)
    :param gamma: The yaw angle (radians)

    Output
    :return: A 3 element array containing the original (x1,y1,z1) position]

    """

    # Get the rotation matrix
    R = euler_rotation_matrix(alpha, beta, gamma)

    # Inverse rotation (transpose of R)
    p1 = R.T @ p2

    return p1
```

Transform_pose

The transform_pose method is the final piece of the puzzle. It takes an object's position and orientation in one coordinate system and figures out where it is in a new coordinate system.

Here is the breakdown:

1. The Inputs:
    - P: The object's original position (X, Y, Z).
    - Q: The object's original orientation (as a Quaternion).
    - T: The "translation" (shift) between the old coordinate system and the new one.
    - R: The "rotation" between the old coordinate system and the new one (also as a Quaternion).
2. The Process:
    - Step 1: Rotate the Position: It first rotates the original position P using the new coordinate system's rotation R. This aligns the point with the new axes.
    - Step 2: Translate the Position: Then, it adds the translation T to this rotated point. This shifts it to the correct location in the new frame.
    - Step 3: Combine Orientations: It calculates the object's new orientation by multiplying the old orientation Q by the coordinate system's rotation R. This gives the final direction the object is facing relative to the new world.
3. The Output:
    - It returns two things:
        - The New Position (X, Y, Z).
        - The New Orientation (Quaternion).

```
def transform_pose(P: np.ndarray,Q: np.ndarray,T: np.ndarray,R: np.ndarray) ->
Tuple[np.ndarray, np.ndarray]:
    """
    10 pts
    Takes a position and orientation in the original frame along with a translation
and
    rotation.

    Then converts the original point into the new coordinate system.

    Hints:
    - Compute the quaternion that represents the new orientation by multiplying the
      old quaternion by the new quaternion (order matters!)
    - When transforming the point rotation is applied before translation

    Input
    :param P: A 3 element array containing the position (x0,y0,z0) in the original
```

```
frame
    :param Q: A 4 element array containing the quaternion (q0,q1,q2,q3)
    :param T: A 3 element array containing the vector between the origins in the
two coordinate systems (dx,dy,dz)
    :param R: A 4 element array containing the rotation in the form of a quaternion
(q0,q1,q2,q3)

    Output
    :return: New Pose, A 3 element array (x1,y2,z1) containing the position in the
new coordinate frame
    :return: New Quaternion, A 4 element array containing the orientation
(q0,q1,q2,q3) in the new coordinate frame

    """

    # Convert the rotation quaternion R to a rotation matrix
    Rot_matrix = quaternion_rotation_matrix(R)

    # Rotate the position P
    P_rotated = Rot_matrix @ P

    # Translate the position
    P_new = P_rotated + T

    # Compute the new orientation
    # Note: Applying the transform rotation R to the original orientation Q
    # Standard order for frame transformation is R * Q
    Q_new = quaternion_multiply(R, Q)

    return P_new, Q_new
```

4.0 Testing our code

How to test?

```
ros2 launch assignment2 assignment2.launch.py
```

Error on running the code

It throws following error.

```
ros2 launch assignment2  assignment2.launch.py
[INFO] [launch]: All log files can be found below
/home/ubuntu/.ros/log/2026-02-01-18-16-47-282862-bfc31c14de10-1804
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [robot_state_publisher-1]: process started with pid [1807]
[INFO] [dummy_object_detector-2]: process started with pid [1808]
[INFO] [dummy_joint_states-3]: process started with pid [1809]
[INFO] [coordinates-4]: process started with pid [1810]
[robot_state_publisher-1] [INFO] [1769969807.799306903] [robot_state_publisher]:
Robot initialized
[coordinates-4] Traceback (most recent call last):
[coordinates-4]   File
"/home/ubuntu/EN613/install/assignment2/lib/assignment2/coordinates", line 33, in
<module>
[coordinates-4]     sys.exit(load_entry_point('assignment2==0.0.0',
'console_scripts', 'coordinates')())
[coordinates-4]
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
[coordinates-4]   File
"/home/ubuntu/EN613/install/assignment2/lib/python3.12/site-packages/assignment2/co
ordinates.py", line 83, in main
[coordinates-4]     rclpy.spin(minimal_subscriber)
[coordinates-4]   File
"/opt/ros/jazzy/lib/python3.12/site-packages/rclpy/__init__.py", line 247, in spin
[coordinates-4]     executor.spin_once()
[coordinates-4]   File
"/opt/ros/jazzy/lib/python3.12/site-packages/rclpy/executors.py", line 845, in
spin_once
[coordinates-4]     self._spin_once_impl(timeout_sec)
[coordinates-4]   File
"/opt/ros/jazzy/lib/python3.12/site-packages/rclpy/executors.py", line 840, in
_spin_once_impl
[coordinates-4]     raise handler.exception()
[coordinates-4]   File "/opt/ros/jazzy/lib/python3.12/site-packages/rclpy/task.py",
line 269, in __call__
[coordinates-4]     self._handler.send(None)
[coordinates-4]   File
"/opt/ros/jazzy/lib/python3.12/site-packages/rclpy/executors.py", line 524, in
handler
[coordinates-4]     await call_coroutine()
[coordinates-4]   File
"/opt/ros/jazzy/lib/python3.12/site-packages/rclpy/executors.py", line 412, in
_execute
[coordinates-4]     await await_or_execute(sub.callback, *msg_tuple)
[coordinates-4]   File
"/opt/ros/jazzy/lib/python3.12/site-packages/rclpy/executors.py", line 111, in
```

```
await_or_execute
[coordinates-4]     return callback(*args)
[coordinates-4]            ^^^^^^^^^^^^^^^
[coordinates-4]   File
"/home/ubuntu/EN613/install/assignment2/lib/python3.12/site-packages/assignment2/co
ordinates.py", line 57, in listener_callback
[coordinates-4]     X1, Q1 = transform_pose(X0,Q0,T,R)
[coordinates-4]     ^^^^^^
[coordinates-4] TypeError: cannot unpack non-iterable NoneType object
[ERROR] [coordinates-4]: process has died [pid 1810, exit code 1, cmd
'/home/ubuntu/EN613/install/assignment2/lib/assignment2/coordinates --ros-args'].
```

Root cause of error

The error TypeError: cannot unpack non-iterable NoneType object confirms that the code running on our system is still the old version where transform_pose returns None.

I have verified that the file assignment2/class2_ws/src/assignment2/assignment2.py does contain the correct implementation.

The Fix

We need to rebuild the package so that the changes in the src folder are compiled/copied to the install folder.

Build the package (using the correct ROS 2 environment)

```
$ colcon build --packages-select assignment2
Starting >>> assignment2
Finished <<< assignment2 [2.29s]

Summary: 1 package finished [2.49s]
```

Source the new setup file

```
$ source install/setup.bash
```

Run the launch file again

```
$ ros2 launch assignment2 assignment2.launch.py

[INFO] [launch]: All log files can be found below
/home/ubuntu/.ros/log/2026-02-01-18-33-05-770930-bfc31c14de10-2279
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [robot_state_publisher-1]: process started with pid [2282]
[INFO] [dummy_object_detector-2]: process started with pid [2283]
[INFO] [dummy_joint_states-3]: process started with pid [2284]
[INFO] [coordinates-4]: process started with pid [2285]
[robot_state_publisher-1] [INFO] [1769970785.969198697] [robot_state_publisher]:
Robot initialized
```

Verify the solution

The above result is expected and it is good news!

- The previous error (crash) is gone, and all nodes (robot_state_publisher, dummy_object_detector, coordinates) have started successfully and are running.
- The coordinates node is designed to be silent when it is working correctly (it only prints logs if there is an error looking up the transform).

How to Verify it is Working

- To confirm that our code is actually calculating and publishing the transformed coordinates, open a new terminal (keep the launch file running) and run:

```
ros2 topic echo /dummy_objects_world
```

If our implementation is correct, we should see a stream of geometry_msgs/msg/Pose messages with the transformed X, Y, Z coordinates and orientation.

```
$ ros2 topic echo /dummy_objects_world
position:
  x: 1.423650859221517
  y: 0.45
  z: 0.9252147209455421
orientation:
  x: 0.5559205345140332
  y: 0.4635465538217541
  z: -0.11906180073655695
  w: 0.6796331431209602
---
position:
  x: 0.7364129786627411
  y: 0.45
  z: 4.348972212051 8834
orientation:
  x: 0.3694506911417045
  y: -0.37684189566025816
  z: 0.43212213540145894
  w: 0.7312775139874622
---
position:
  x: -0.262295251999345
  y: 0.45
  z: 4.540213672139864
orientation:
  x: 0.25783401305504094
  y: -0.5377969351660072
  z: 0.5067000836796192
  w: 0.6225360016212311
---
position:
  x: 1.491232862044016
  y: 0.45
  z: 0.9837611541250959
orientation:
  x: 0.558821512262986
  y: 0.4457636569141773
  z: -0.104603396317277
  w: 0.6914270597561246
---
position:
  x: -2.2141528532773886
  y: 0.4499999999999999
  z: 3.4894800821574847
orientation:
  x: -0.025824965158209923
  y: -0.7749771429000212
  z: 0.5679405112012852
  w: 0.27602000791794035
---
position:
  x: 1.8209168519782901
  y: 0.45
  z: 1.3902167441037547
orientation:
  x: 0.5681573260549798
  y: 0.33821039297100985
  z: -0.020508679602921742
  w: 0.7499268943536397
---
position:
  x: 0.6070835893687356
  y: 0.44999999999999996
  z: 4.400091204722071
orientation:
  x: 0.3552163418943145
  y: -0.40040159141616183
  z: 0.44389717676531965
  w: 0.7186481631368057
---
position:
  x: -0.13185396085152212
  y: 0.45
  z: 4.539982581353318
orientation:
  x: 0.2728207848787937
  y: -0.5190044459061586
  z: 0.49879071004492354
  w: 0.6382875542560275
---
position:
  x: 1.1391433223425624
  y: 0.45000000000000023
  z: 0.7362801282836426
orientation:
  x: 0.5410996603405323
  y: 0.5296492305423904
  z: -0.17445489579934603
  w: 0.629482572732839
```

What is going on here?
It's not publishing the robot arm's location, but rather the location of a "dummy object" (like a ball or a target) relative to the world.

Here is the flow of data:

1. The Source (dummy_object_detector):
    - This node simulates a camera attached to the robot.
    - It says: "I see an object at x=1.5, y=0.25, z=0.01 relative to the camera."
    - It publishes this to the /dummy_objects topic.

2. The Transformation (coordinates.py):
    - This node listens to /dummy_objects.
    - It asks TF2: "Where is the camera (single_rrbot_camera_link) right now relative to the world?"
    - It gets the robot's current position/rotation (which is moving because the arm is swinging).
    - It uses your code (transform_pose) to combine the "object relative to camera" + "camera relative to world".
3. The Result (/dummy_objects_world):
    - The output is the object's position in the world frame.
    - Since the object is "fixed" in the camera's view (hardcoded in dummy_object_detector), but the camera is moving (the arm is swinging), the calculated "world position" of the object will actually move around in the world as the arm moves.
    - That's why you see the red arrow (the object) moving in RViz. It's tracing where that object would be in the world if it were stuck in front of the moving camera.

5.0 Visualize using RViz2

Launch RViz2

Based on the file list, this project uses a custom robot called single_rrbot, not TurtleBot3. However, we can visualize it using RViz2, which is the standard visualization tool for ROS 2.

To visualize the robot and the coordinate transformations:

1. Keep your launch file running in the first terminal.
2. Open a new terminal and run:

```
rviz2
```

3. Configure RViz:
    - Set Fixed Frame (top left) to world.
    - Click Add (bottom left) -> RobotModel (to see the arm).
    - Click Add -> TF (to see the coordinate frames moving).
    - Click Add -> By Topic -> /dummy_objects_world -> Pose (to see your calculated result).
4. Single_rrbot.urdf suggest it's designed for the RRBot arm.

Topic as unvisualizable

The reason we see the topics as "unvisualizable" or disabled is likely because the message type is geometry_msgs/msg/Pose, which does not have a header (no Frame ID).

RViz2 needs to know which coordinate frame the pose belongs to (e.g., "world", "map", "base_link") to draw it in the 3D view. Without a header, it doesn't know where to put it.

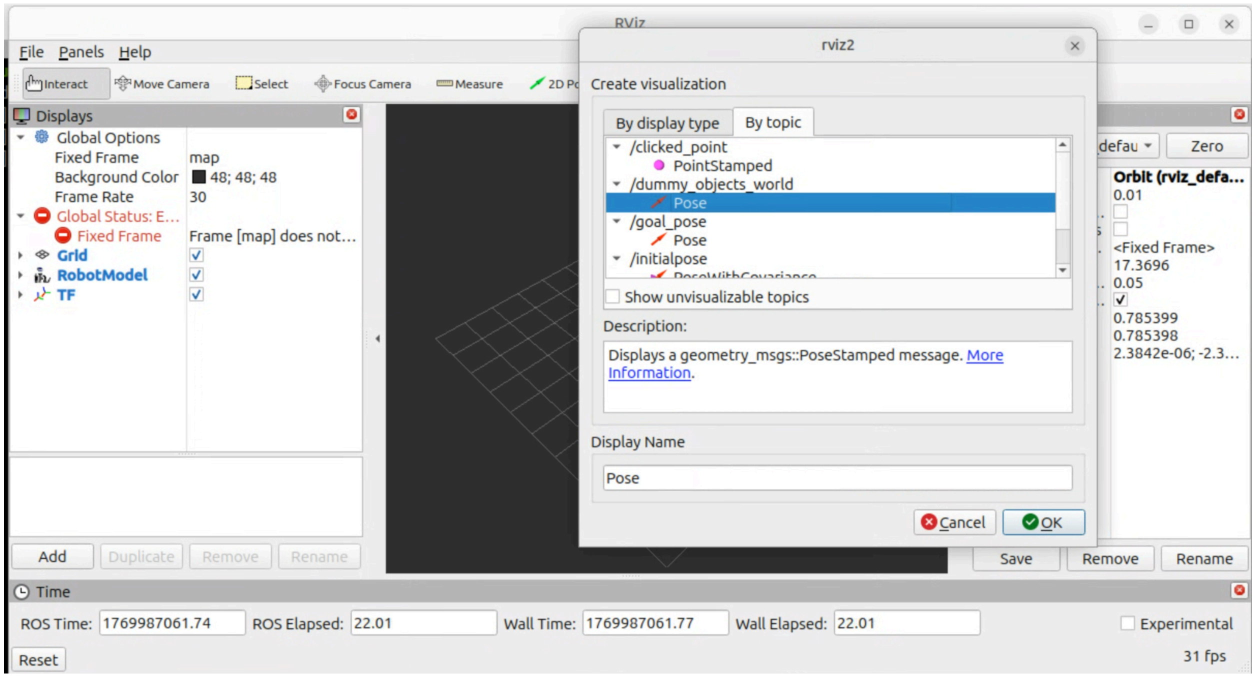
How to Fix It
We can modify coordinates.py to publish PoseStamped instead of Pose. This adds a header with the frame ID (world), making it natively visualizable in RViz2.

```
# coordinates.py
import math
import tf2_ros
import rclpy
from time import sleep
from rclpy.duration import Duration
from rclpy.node import Node
from std_msgs.msg import String
from geometry_msgs.msg import Pose, PoseStamped
import numpy as np
from .assignment2 import transform_pose

class MinimalSubscriber(Node):

      def __init__(self):
             super().__init__('minimal_subscriber')
             self.subscription = self.create_subscription(
                    Pose,
                    'dummy_objects',
                    self.listener_callback,
                    10)
             self.subscription  # prevent unused variable warning

             self.publisher_ = self.create_publisher(PoseStamped,
'dummy_objects_world', 10)

             self._tf_buffer = tf2_ros.Buffer()
             self.listener = tf2_ros.TransformListener(self._tf_buffer,self)

             self._to_frame = 'world'
             self._from_frame = 'single_rrbot_camera_link'
      def listener_callback(self, msg: Pose):

            try:
                  when = rclpy.time.Time()
                  trans = self._tf_buffer.lookup_transform(self._to_frame,
self._from_frame,

       when, timeout=Duration(seconds=5.0))
            except tf2_ros.LookupException:
                  self.get_logger().info('Transform isn\'t available,
waiting...')
                  sleep(1)
                  return

            X0 = np.array([msg.position.x,msg.position.y,msg.position.z])
            Q0 =
np.array([msg.orientation.w,msg.orientation.x,msg.orientation.y,msg.orientation.z])

            T = np.array([trans.transform.translation.x,
                          trans.transform.translation.y,
                          trans.transform.translation.z
                         ])
            R = np.array([
                          trans.transform.rotation.w,
                          trans.transform.rotation.x,
                          trans.transform.rotation.y,
                          trans.transform.rotation.z,
                         ])

            X1, Q1 = transform_pose(X0,Q0,T,R)

            self.publish_pose(X1,Q1)


      def publish_pose(self,X1: np.ndarray,Q1: np.ndarray):

            msg = PoseStamped()
            msg.header.stamp = self.get_clock().now().to_msg()
            msg.header.frame_id = self._to_frame
            msg.pose.position.x = X1[0]
            msg.pose.position.y = X1[1]
            msg.pose.position.z = X1[2]
            msg.pose.orientation.w = Q1[0]
            msg.pose.orientation.x = Q1[1]
            msg.pose.orientation.y = Q1[2]
            msg.pose.orientation.z = Q1[3]
            self.publisher_.publish(msg)




def main(args=None):
    rclpy.init(args=args)

    minimal_subscriber = MinimalSubscriber()

    rclpy.spin(minimal_subscriber)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

After this change, we need to rebuild and launch it again

```
colcon build --packages-select assignment2
source install/setup.bash
ros2 launch assignment2 assignment2.launch.py
```

Now we should see the dummy topic enabled as below.



Global Options
Fixed Frame | map
Background Color | 48; 48; 48
Frame Rate | 30
Global Status: E…
Fixed Frame | Frame [map] does not…
Grid ✓
RobotModel ✓
TF ✓

rviz2
Create visualization
By display type | By topic
/clicked_point
PointStamped
/dummy_objects_world
Pose
/goal_pose
Pose
/initialpose
PoseWithCovariance
Show unvisualizable topics
Description:
Displays a geometry_msgs::PoseStamped message. More Information.
Display Name
Pose
Cancel | OK

Orbit (rviz_defa…
0.01
<Fixed Frame>
17.3696
0.05
✓
0.785399
0.785398
2.3842e-06; -2.3…

ROS Time: 1769987061.74 ROS Elapsed: 22.01 Wall Time: 1769987061.77 Wall Elapsed: 22.01
31 fps

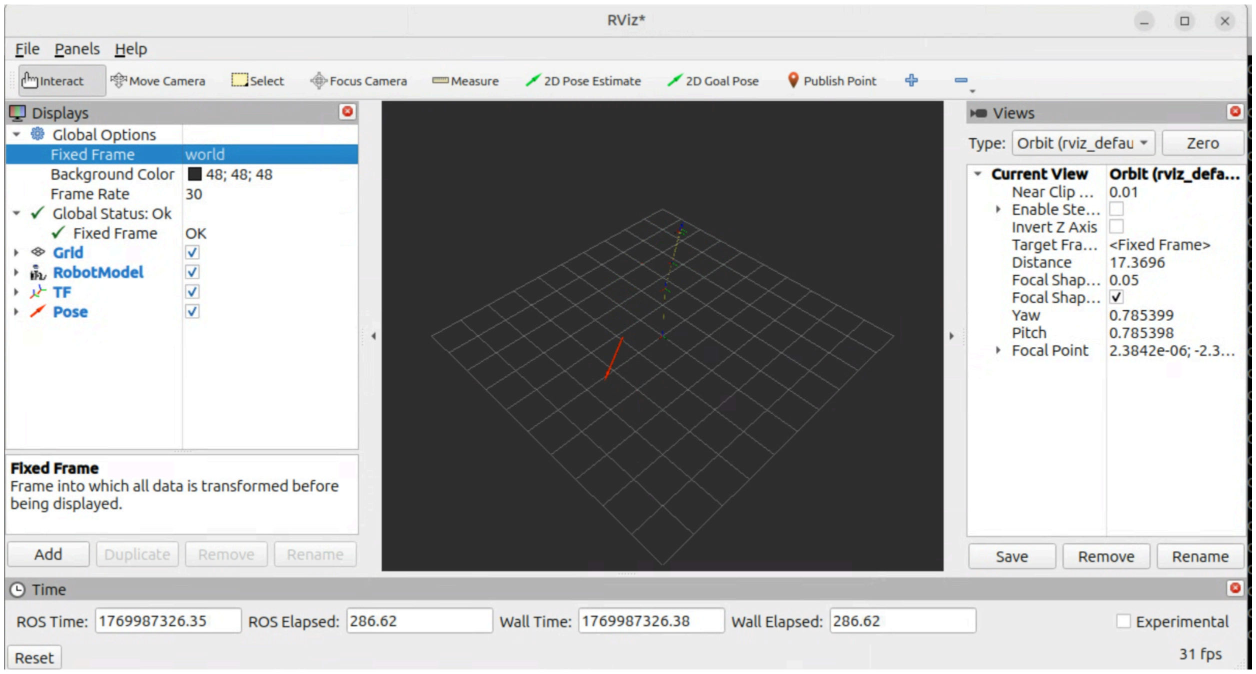
Fix the Fixed Frame error

The Problem: In the "Global Options" section (top left of RViz), the Fixed Frame is set to map. Since our project doesn't have a map frame (it only has world), RViz doesn't know where to draw anything, so it shows an error and hides the robot.

The Fix:

Look at the top-left panel in RViz.

- Click on the word map next to "Fixed Frame".
- Type world and press Enter.
    - (Or click the dropdown arrow if it appears and select world).
- Once you change this to world, the error should turn into a green checkmark, and the robot arm should appear!


RViz*
File Panels Help
Interact Move Camera Select Focus Camera Measure 2D Pose Estimate 2D Goal Pose Publish Point
Displays
Views
Add Duplicate Remove Rename
Save Remove Rename
Time
Reset
Experimental

Global Options	
Fixed Frame	world
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
TF	✓
Pose	✓

Fixed Frame
Frame into which all data is transformed before being displayed.

Type: Orbit (rviz_defau Zero

Current View	Orbit (rviz_defa…
Near Clip …	0.01
Enable Ste…	☐
Invert Z Axis	☐
Target Fra…	<Fixed Frame>
Distance	17.3696
Focal Shap…	0.05
Focal Shap…	✓
Yaw	0.785399
Pitch	0.785398
Focal Point	2.3842e-06; -2.3…

ROS Time: 1769987326.35 ROS Elapsed: 286.62 Wall Time: 1769987326.38 Wall Elapsed: 286.62

31 fps

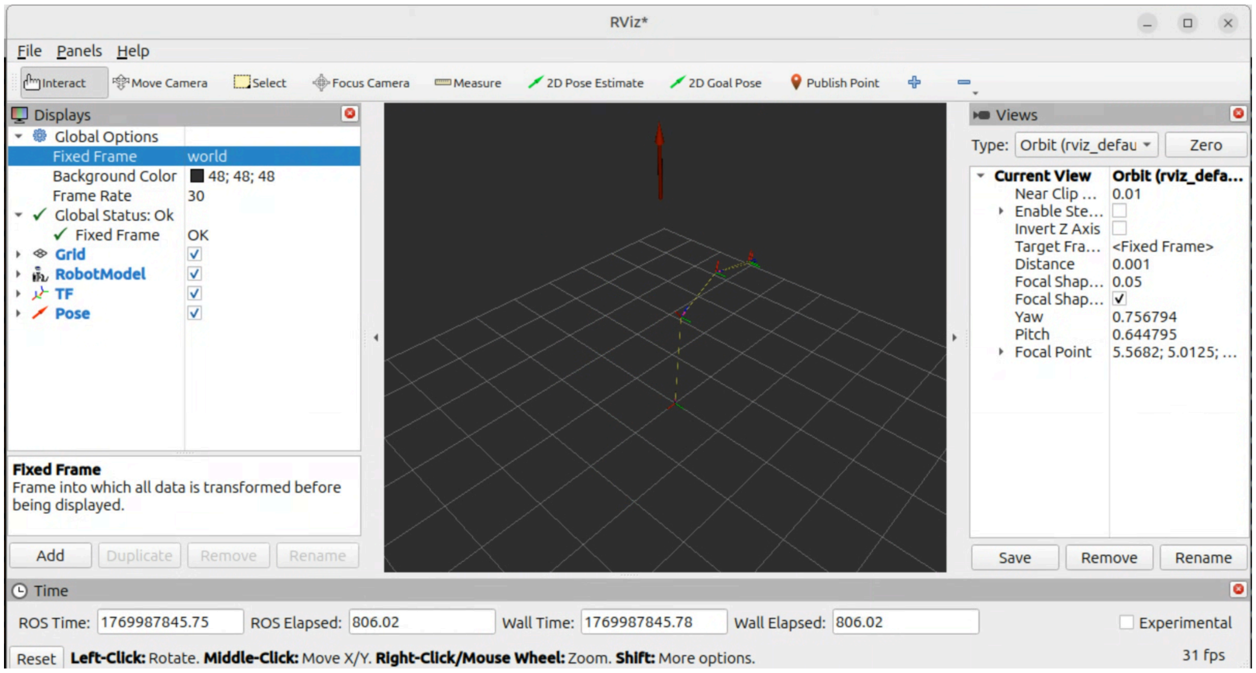
That's great progress! The robot is visible, which means the frames are working.

The "tiny" issue is just a camera zoom problem. The camera is currently very far away (Distance: ~17.3 meters).

How to Zoom In

1. **Mouse Wheel:** Scroll UP on your mouse wheel while hovering over the black 3D view. This is the easiest way to zoom in.
2. **Right-Click + Drag:** If you don't have a mouse wheel, hold the Right Mouse Button and drag DOWN to zoom in (or UP to zoom out).
3. **Focus Camera**: You can also click the "Focus Camera" button in the top toolbar, then click on the robot to center and zoom on it.

Once you zoom in, you should clearly see the robot arm and the red arrow (your transformed pose) moving relative to it!


RViz*
File Panels Help
Interact Move Camera Select Focus Camera Measure 2D Pose Estimate 2D Goal Pose Publish Point
Displays
Views
Add Duplicate Remove Rename
Save Remove Rename
Zero
Time
Reset

Global Options
Fixed Frame: world
Background Color: 48; 48; 48
Frame Rate: 30
Global Status: Ok
Fixed Frame: OK
Grid ✓
RobotModel ✓
TF ✓
Pose ✓

Fixed Frame
Frame into which all data is transformed before being displayed.

Type: Orbit (rviz_defau

Current View	**Orbit (rviz_defa...**
Near Clip ...	0.01
Enable Ste...	☐
Invert Z Axis	☐
Target Fra...	<Fixed Frame>
Distance	0.001
Focal Shap...	0.05
Focal Shap...	✓
Yaw	0.756794
Pitch	0.644795
Focal Point	5.5682; 5.0125; ...

ROS Time: 1769987845.75 ROS Elapsed: 806.02 Wall Time: 1769987845.78 Wall Elapsed: 806.02 ☐ Experimental

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:** Zoom. **Shift:** More options. 31 fps