Computational Robotics Assignment 1

Noor Hasan
nnh29@scarletmail.rutgers.edu
210006853

Dylan Turner
dpt50@scarletmail.rutgers.edu
207004235

1: Validating Rotations

Computers perform math operations with finite precision, meaning we cannot expect entirely perfect results. When checking that a rotation or quaternion is valid, we use helper functions such as np.isclose() and np.allclose(). A parameter of these functions is atol, the tolerance parameter. By passing epsilon for this value, we can check our rotation matrices and quaternions with epsilon numerical precision.

check_SOn():
A matrix that is an element of SO(n) is orthogonal and has a determinant of one. To check that a matrix A is orthogonal, you can simply check that $AA^T = I$. Then we can check the determinant by using numpy.linalg.det().

check_quaternion():
For this function, we simply check that the input vector has four components and a magnitude of 1. These are the requirements for a valid quaternion.

check_SEn():
A matrix that is an element of SE(n) includes an n x n rotation matrix paired with any translation vector in R$\hat{3}$. The bottom row is all zeros except for the last entry, expressing the matrix in homogeneous coordinates. To validate that the input matrix is part of SE(n), we use check_SO(n) on the subset representing the rotation. Secondly, we ensure the bottom row is of the form [0,0,...,1]. There is no need to check the translation, as all possible vectors are valid translations.

1: - Extra Credit -

correct_SOn():
To correct a square matrix that is not an element of SO(n), you can use Singular Value Decomposition. By decomposing the input A into U, S, and V, we can easily find the closest valid rotation matrix: $A' = UV^T$. At this point, the determinant could be -1 or 1. If it is -1, we invert the sign of one of the columns of U, and we have our answer.

correct_quaternion():
To find the closest quaternion to a vector with four components we can simply divide each component by the vectors magnitude. This will give you a valid unit quaternion.

correct_SEn():
First, we will clarify that we are assuming a 3x3 input matrix is SE(2), and a 4x4 input is SE(3). Knowing this, it is simple enough to correct these matrices. We extract the rotation part of the matrix and correct it with correct_SO(n). Then we ensure the bottom row is of the form [0,0,...,1].

2: Uniform Random Rotations

random_rotation_matrix():
For the naive solution, we generated three random numbers between 0 and 2pi. Using the uniformly random numbers, we created rotation matrices about each axis and then combined them into one random rotation. This is a naive solution, as Euler angles do not uniformly cover the surface of the sphere in rotation space.

Therefore we also implemented the non-naive solution, as detailed in Algorithm 1 in the write-up. A 3D visualization of these random rotations can be seen in **Figure 1**.

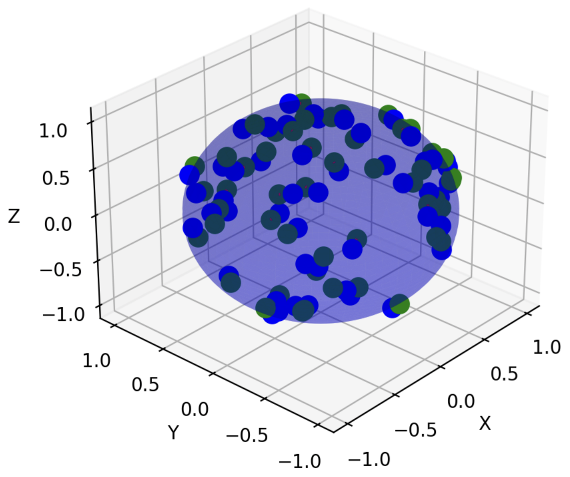

Figure 1: 3D Visualization of 50 Random Rotations on Unit Sphere

2: - Extra Credit -

random_quaternion():
Our naive solution to generating a random quaternion is to generate four random numbers to assign to each component, and then divide the vector by its magnitude.

The non-naive way to generate a random quaternion is detailed in Algorithm 2 in the write-up.

3: Rigid body in motion

interpolate_rigid_body(): For this function, we use np.linspace() to create a linear spacing of points between the start and end (x,y) points, as well as for theta. For theta, we make sure to rotate in the direction of the shortest path to avoid spinning around excessively. At each state, we convert the position and rotation into a pose and append it to path for the output.

forward_propagate_rigid_body():
For this function, we loop over each tuple in the input path. At the start of the loop, we multiply the velocity of each component by the duration it's applied to get the total change for each step. Then we can rotate and translate accordingly, and append the new pose to the path output.

visualize_path():
For the visualization, we saved the graph after each step from the path was added, and compiled them as a .gif using the pillow library. In the graph, the red arrow represents the orientation of the robot, the green lines represent the path across each step, and the blue points represent the origin of the robot. **Figure 2 and 3** show the first and final frame of the .gif for both movement functions.

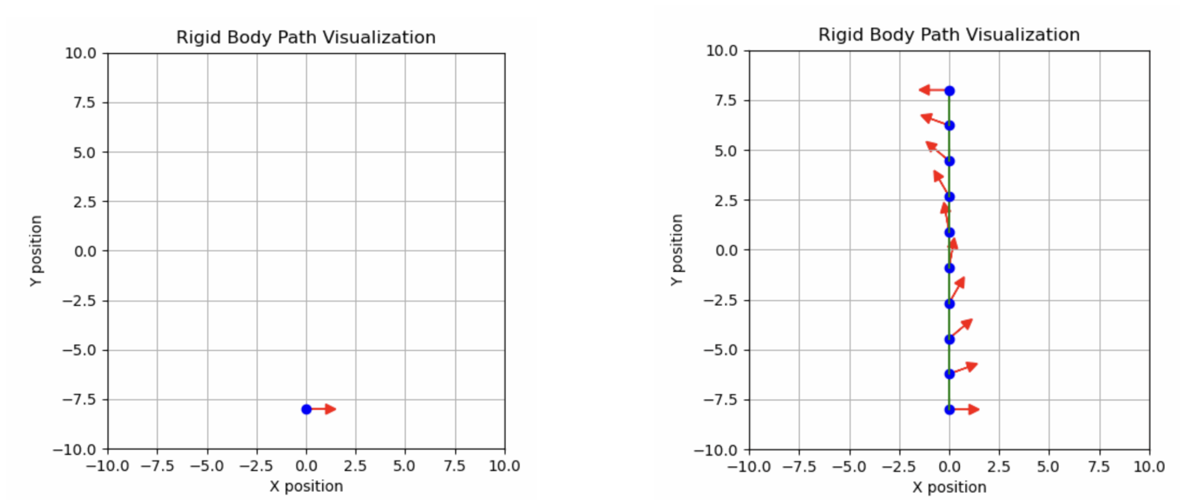

Figure 2: First and Final Pose Interpolation

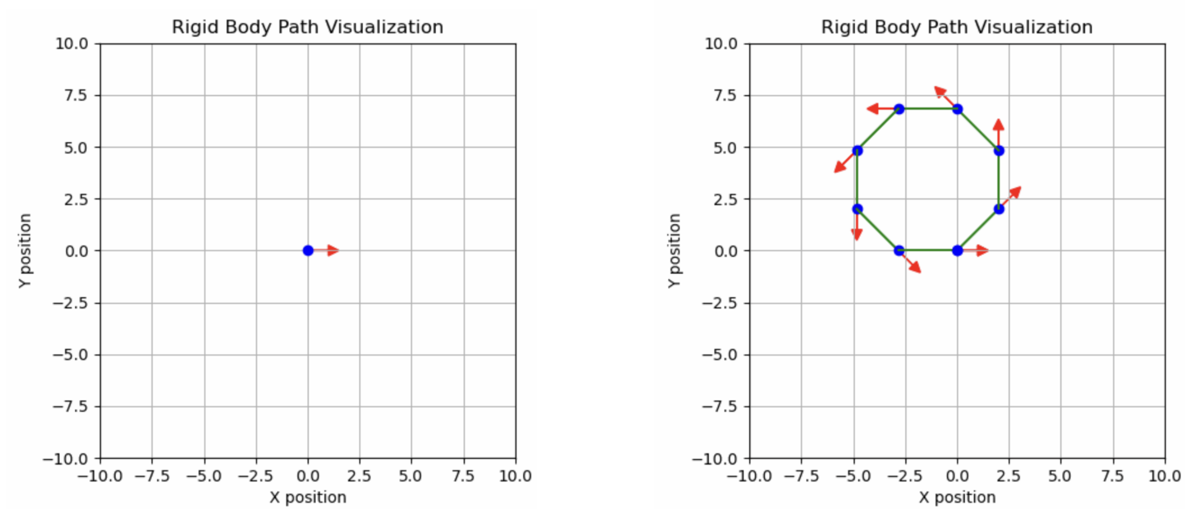

Figure 3: First and Final Pose Forward Propagation

4: Movement of an Arm

interpolate_arm():
Unlike the robot in part 3, the transformations to the 2 link arms must be made sequentially. It is not quite as simple as using np.linspace() in this case. Instead, we have a loop that repeats for the amount of steps between the start and end pose. At each iteration, we calculate the new angle for θ_1, create the transformation matrix, and apply it to link 1. Then we repeat the same for arm 2, relative to the end of link 1.

forward_propagate_arm():
Our forward propagate function is similar to interpolate, except that the rotation at each step is based on the plan rather than a fixed interpolation. Additionally, we save the rotation of each arm between each iteration so we can add the next rotation to it. Similar to before, the transformations are done sequentially, and the output path is a list of tuples containing the transform for both links.

visualize_arm_path):
The general process for this function is the same as the visualization function for the robot in part 3. The difference however is we clear the graph at each iteration to show a fluid transition of the arms. **Figure 4 and 5** show the first and final frame of the .gif for both movement functions.

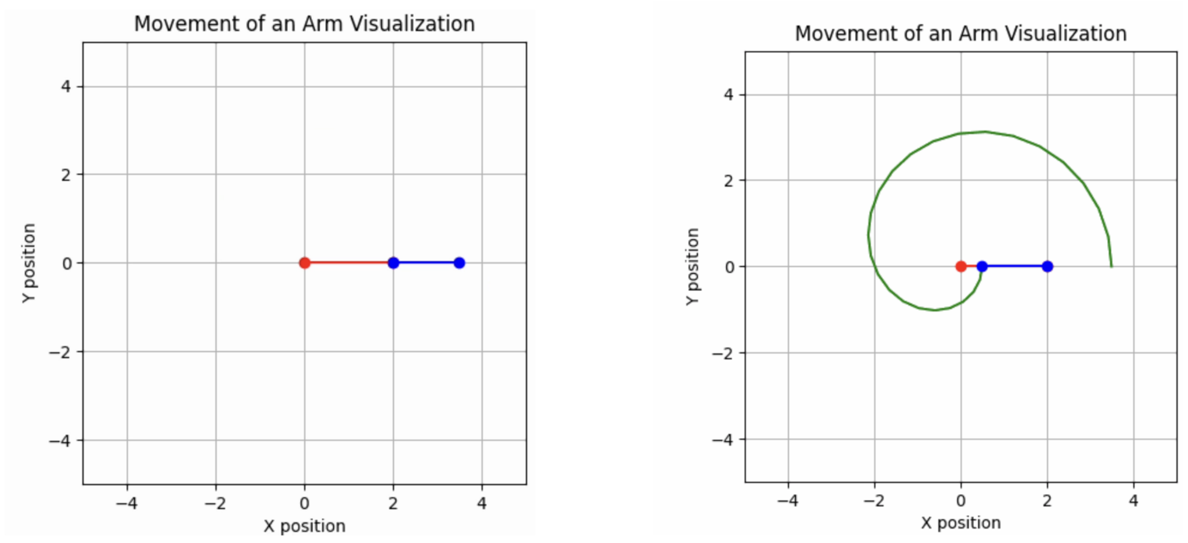

Figure 4: First and Final Pose Interpolation

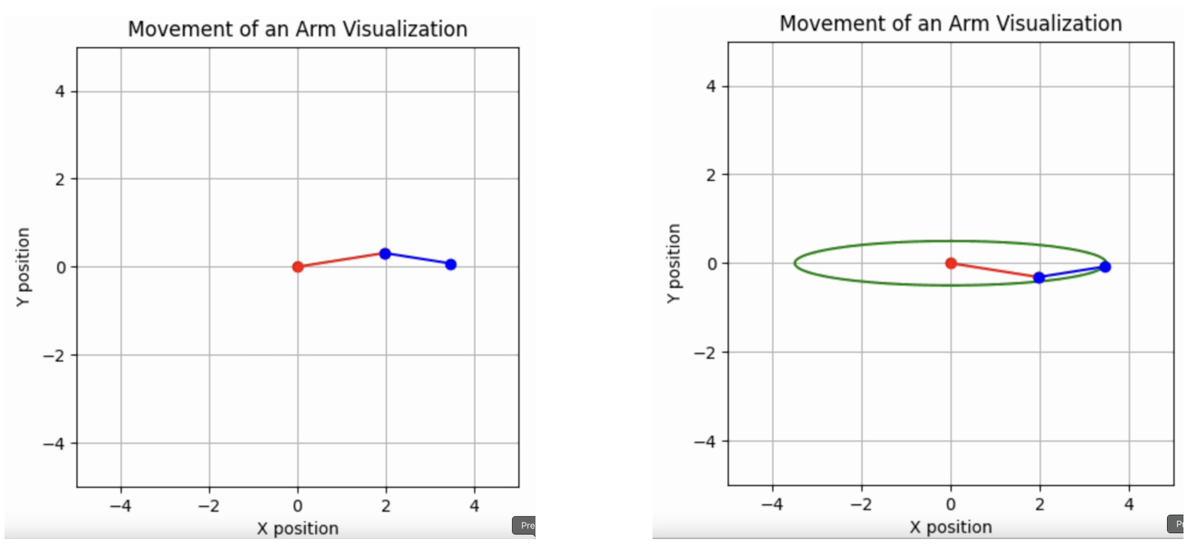

Figure 5: First and Final Pose Forward Propagation