CS460: Assignment 1

Arya Shetty —— Damon Lin

September 25, 2024

1 Validating Rotations

Important note, to run on the iLab computers just use the python environment:
$ source /common/system/venv/python310/bin/activate
The rest of the libraries were just numpy and matplotlib.
Within the program, the epsilon parameter was implemented as tolerance to check that the answer was within a given numerical precision. For example, in the first function, `check_SOn(m)`, if the determinant of the matrix minus one is greater than the epsilon numerical precision, then the matrix does not belong in the Special Orthogonal group.

1.1 `check_SOn(m)`

This function checks whether the input matrix `m` satisfies the conditions for Special Orthogonal Groups. The conditions to check for include: the input matrix is a square, the matrix is orthogonal, and the determinant of the matrix is equal to 1. In this context, epsilon checks whether the determinant is within its numerical precision.

1.2 `check_quaternion(v)`

This function checks whether the input vector `v` satisfies the quaternion conditions. The two conditions for a vector to be in Quaternions are: the vector is a unit vector and the vector is of length four. Therefore, the function first checks that the vector is of length four and then verifies whether the vector is a unit vector. The implementation to check the latter utilizes the `nn.allclose()` function comparing the normalized vector `v` with a unit vector while ensuring that the range is within the epsilon numerical precision.

1.3 `check_SEn(m)`

The function `check_SEn(m)` verifies that the input matrix `m` satisfies the conditions to belong in the Special Euclidean Groups. Thus, such a matrix must satisfy the following conditions:

1. The matrix is 2D.
2. The rotation matrix within the matrix is either for 2D or 3D rotation.
3. The rotation matrix must belong to the Special Orthogonal group.
4. The bottom row must have $[0, 0, \ldots, 1]$.

1.4 `correct_SOn(m)`

The function first checks if the matrix is orthogonal by testing $M \cdot M^T$. If it's not orthogonal, it corrects the matrix so it is closer to an orthogonal one. Then, it checks if the determinant is close to 1. If not, it corrects the matrix so that the determinant becomes 1. This approach ensures that the corrected matrix is orthogonal and has a determinant of 1, ensuring membership in the Special Orthogonal Groups.

1.5 correct_quaternion(v)

The function first checks that the input is a valid 4-dimensional vector. Then, it verifies whether the vector is a unit quaternion (norm of 1). If the vector is not a unit quaternion, the function normalizes it to make it one. The goal of the function is to ensure that the vector `v` represents a valid unit quaternion, correcting it if necessary while keeping it as close as possible to the original vector.

1.6 correct_SEn(m)

Similar to the implementation of `ccheck_SEn(m)`, this correction function checks to see whether the rotation matrix within matrix `m` is part of the Special Orthogonal group. If the rotation matrix is part of that group, the function checks the bottom row of the matrix and fixes it if there are any problems. On the other hand, if the rotation matrix is not part of the group, the function applies the `correct_SOn(m)` function onto the rotation matrix. After, the program goes through the same steps as before, correcting the bottom row of the matrix if there are issues.

2 Uniform Random Rotations

2.1 random_rotation_matrix(naive)

This function generates a random rotation matrix, in two different ways. The naive approach picks three random Euler angles (using np.random.uniform) and creates the three rotation matrices for each axis based on these angles. Finally, it multiplies all the matrices to get the final rotation matrix. Since they started as rotation matrices it already meets the SO(n) conditions. The non-naive approach involves an algorithm from the paper. Here the rotation matrix won't always meet the conditions. However, the method allows for a more random choice of rotation matrices as shown by the images below. The main reason is that the Euler angles have a bias in sampling random angles, while the other approach uses a random vector to create the reflection matrix and cover more of the space. The images below are made with 1000 randomly generated rotation matrices.

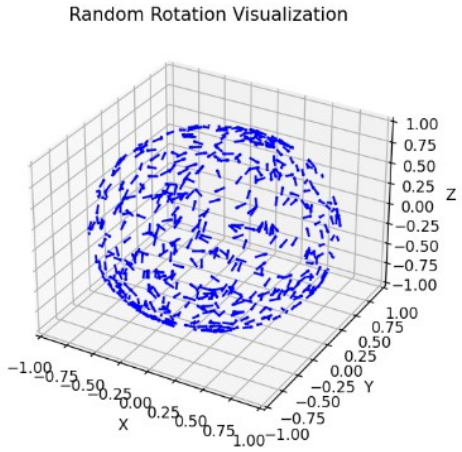

Figure 1: Random Rotation Visualization naive

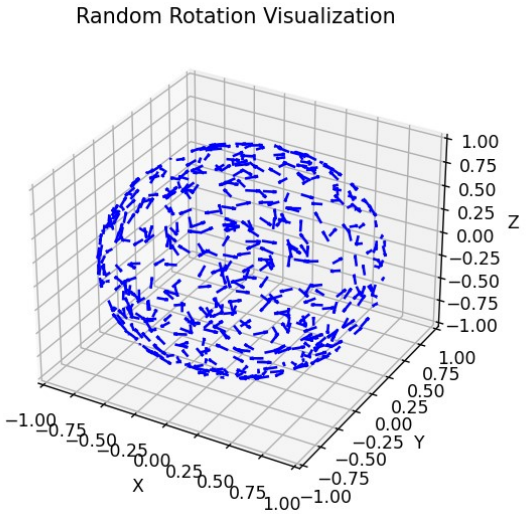

Figure 2: Random Rotation Visualization non-naive

2.2 `random_quaternion(naive)`

This function builds off the previous function `random_rotation_matrix(naive)`. As such, for the naive implementation, after the Euler angles are generated and the rotation matrices are calculated, the rotation matrices are converted to a quaternion. For the non-naive part, the same train of thought is applied once more wherein a new quaternion is produced until it satisfies `check_quaternion(v)`. In the same fashion, the images come from 500 randomly generated quaternions.

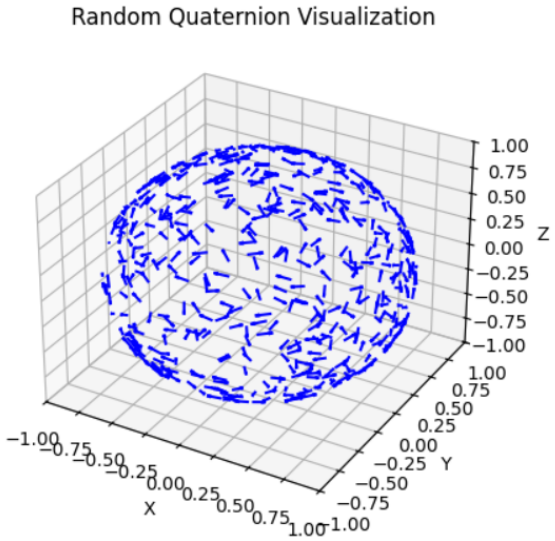

Figure 3: Random Quaternion Visualization naive

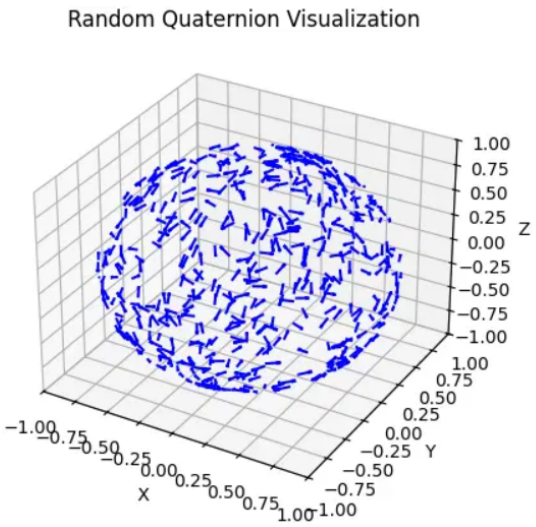

Figure 4: Random Quaternion Visualization non-naive

3 Rigid body in motion

3.1 `interpolate_rigid_body(start_pose, goal_pose)`

The idea for this function is to have the robot traverse from its start pose labeled in tuple form of (x, y, theta) to some goal. In this case, I predefined the amount of steps I want to take towards the goal, adjusting this will yield more or less poses depending on how smooth of a translation you want. The x and y translation is easily done using linspace to divide the distance equally. The rotation is the hard part since there could be singularities depending on whether the goal theta was greater than pi since a simple algorithm to divide the distance wouldn't work as it can take a longer path to reach the final angle (ex. going from 0 to 200 degrees, the faster way would be to go in the negative direction). So to account for this we used a SLERP design, spherical linear interpolation. The first step was getting the rotation matrices for the start and goal angles. Then we get the relative rotation matrix by multiplying the goal rotation matrix with the inverse of the start rotation matrix, which gives how much we should rotate without need of start orientation. Then you can get the angle from the matrix (use arctan) and iterate through it to get the poses for the path. To get the angle for the path you get the rotation matrix for the iterated angle and multiply it with the start pose, to get one of the iterations of the interpolated matrix. Then extract the angle like last time to use the interpolated theta for the pose. The following images are for start_pose = (0, 0, 0) and end_pose = (4, 5, np.pi/2). The box's bottom left point is in line with the pose.

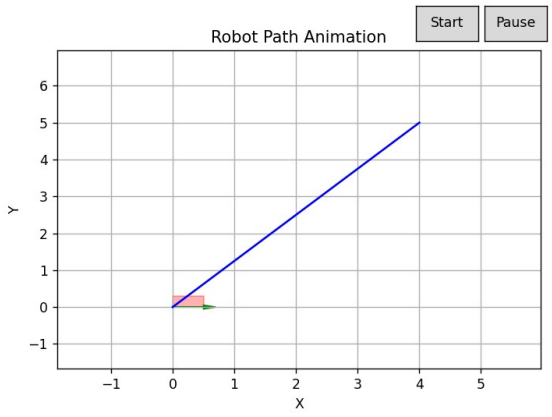

Figure 5: Start of robot path animation

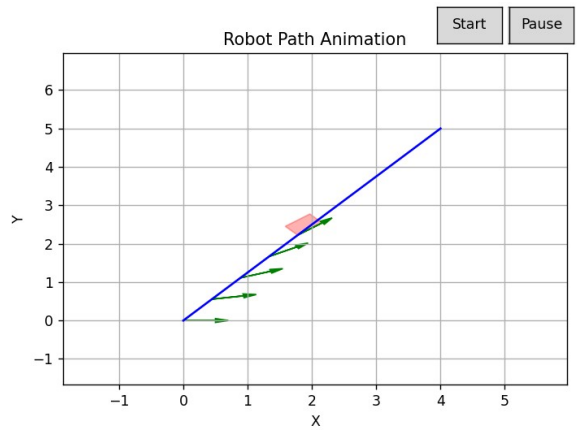

Figure 6: Middle of robot path animation

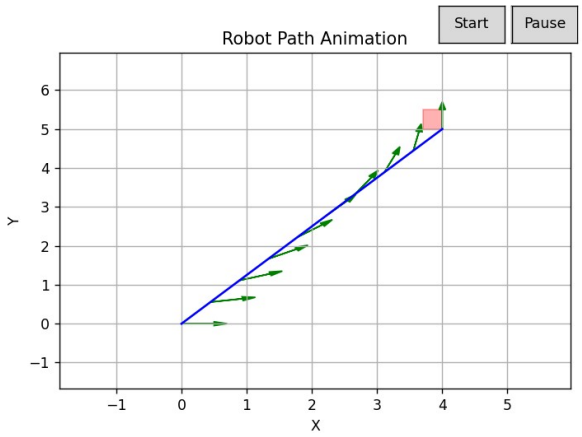

Figure 7: End of robot path animation

3.2 `forward_propagate_rigid_body(start_pose, plan)`

For this function we need to create a path based on the plan the robot is given, which is a sequence of tuples (velocity, duration). First we make sure the start pose is in bounds. Then we iterate through the plan, getting the Vx, Vy, Vtheta, and duration. Then multiply the velocities and duration to get the distance and angle traveled. Next, we applied the rotation matrix to get into the world frame and get the x and y the box ends up at. Finally the angle we just add to the box's theta currently. The images below are for plan = [(1, 1, np.radians(45), 1), (1, 0, np.radians(45), 2), (1, 0, 0, 2), (1, 0, np.radians(45), 1), (1, 0, 0, 2)]

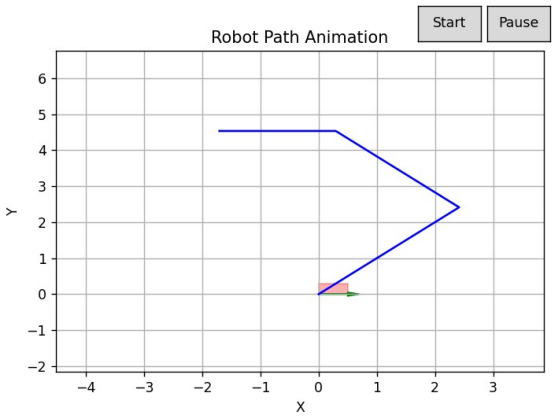

Figure 8: start of forward propagate anim

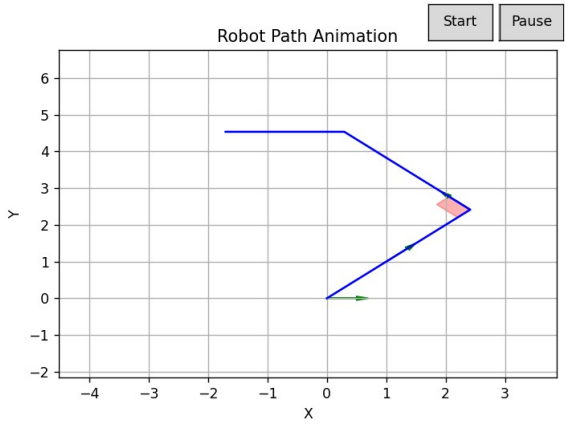

Figure 9: middle of forward propagate anim

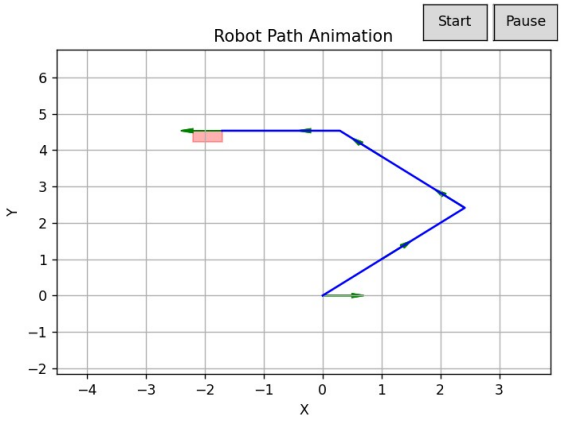

Figure 10: end of forward propagate anim

3.3 `visualize_path(path)`

This function was used to animate and view the paths generated from each of the functions. It uses Matplotlib's library to help. We used the patches import to create the box for the robot based on the dimensions given and updated the position of each frame with the next pose. We also rotated it based on the theta and added start and pause buttons to look at each step. Then for each pose in the path

the box's center is adjusted based on the x and y, and robot dimensions, and it is rotated to the theta. (Gifs in folder)

4 Movement of an Arm

4.1 `interpolate_arm(start, goal)`

This function is similar to the rigid body's movement except with two arm links. Here we apply the technique of unwrapping the angle to avoid singularities we can get when traversing to the goal angle. This time we need to apply it twice for each arm. Once we get the next angles for each arm, we want to get poses for the path of each arm. In this case, we will be using four tuples to animate the robot, the joint of the first arm, the link of the first arm, the joint of the second arm, and the link of the second arm. We get this by using forward kinematics, which calculates the orientation with our angles. So we find the first joint by using the angle of the arm, and that's done with simple geometry as we line the base with the origin. The second arm is similar but we add the first arm's coordinate and theta since it is relative to the arm. Then the function returns four tuples. The first is the center point of the first arm which is L1 which is just the length divided by 2. The second tuple is the midpoint of the second arm or L2. Then the final two tuples are the end effectors of each arm. Finally, we return the path of these poses at the end and go to the next iteration. The images below are for start = (np.radians(0), np.radians(45)) and goal = (np.radians(135), 0).

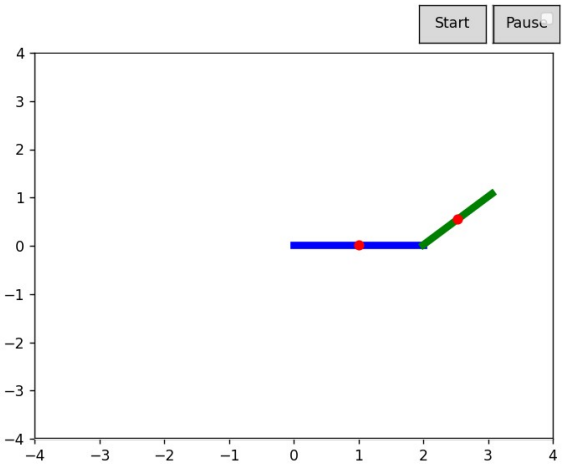

Figure 11: start of arm anim

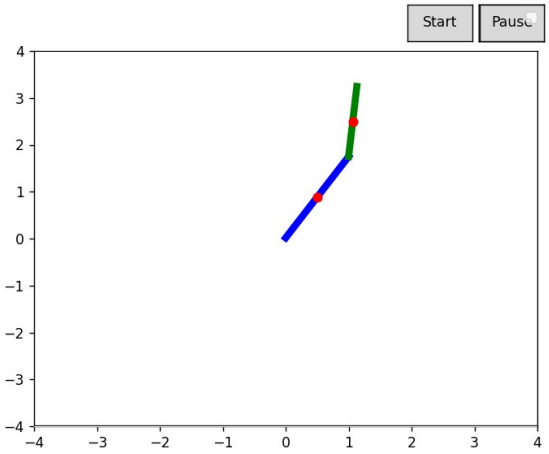

Figure 12: middle of arm anim

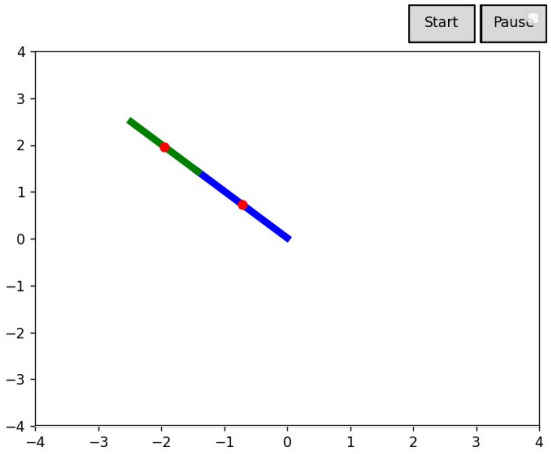

Figure 13: end of arm anim

4.2 `forward_propagate_arm(start_pose, plan)`

This function also returns a path but takes in a plan of tuples (velocity, duration) for the robot arms to move. Here We multiply the velocity for one arm to move by the duration to get the amount the theta should move based on the start pose. We then use the forward kinematics again to calculate the poses. The images below are for plan = [((0, np.radians(30)), 2), ((np.radians(45), 0), 1), ((np.radians(60), 0), 1)].

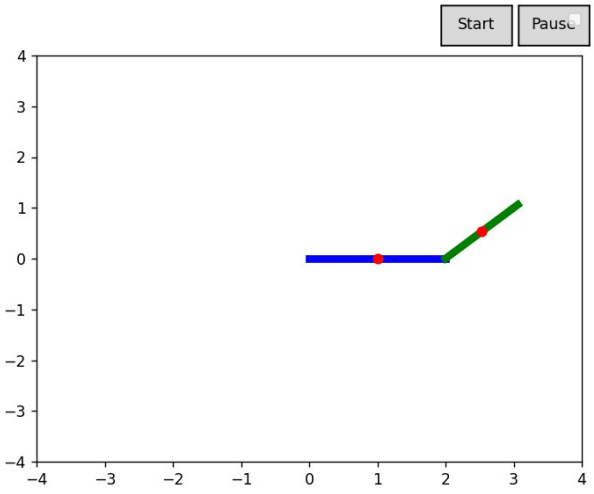

Figure 14: start of propagate arm

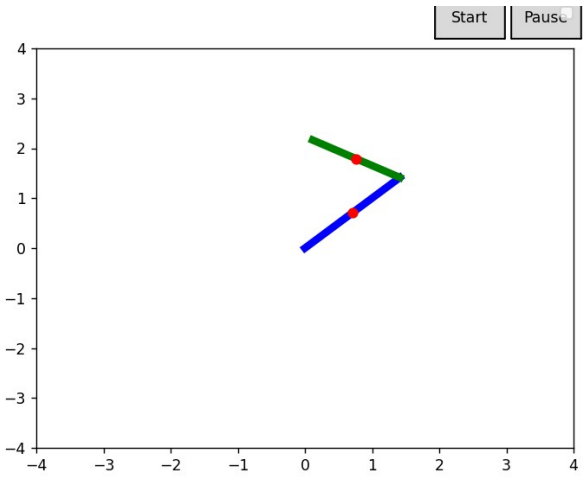

Figure 15: middle of propagate arm

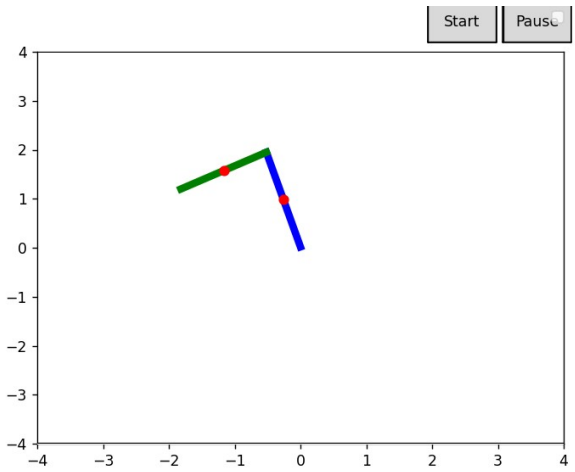

Figure 16: end of propagate arm

4.3 `visualize_arm_path(path)`

This function was similar to the last problem's animation to view the path of the two-link arm robot this time. It uses Matplotlib's library to help. For the plotting, we need the joint position and center of the link for each part of the arm. In the function, it is c_x, c_y for the center of the link and x, y for the joint point. We update based on the pose that is sent from the path and then add it to the initial set for plotting. After we use some built in libraries from Matplotlib for animating, and start and pause functionality. (Gifs in folder)