Team Members: Harshith Samayamantula (hs1018) and Albert Zou (arz41)

1 Validating Rotations

1.1 check_SOn(matrix: m, float: $\varepsilon = 0.01$) $\rightarrow$ bool:

Input: A matrix $m \in x$ and a numerical precision value $\varepsilon = 0.01$.

Output: A boolean value that is true if $m \in SO(n)$ and false otherwise.

Since the group $SO(n)$ consists of the orthogonal matrices with a determinant of 1 (representing proper rotations without scaling or reflections), we just need to ensure the following two properties:

$$m^\intercal m = I \tag{1}$$

$$\det m = 1 \tag{2}$$

Our code checks to ensure the orthogonality and the determinant is 1. The check returns true if every value falls within the provided ε

1.2 check_quaternion(vector: v, float: $\varepsilon = 0.01$) $\rightarrow$ bool:

Input: A vector $v \in x$ and a numerical precision value $\varepsilon = 0.01$.

Output: A boolean value that is true if $v \in S^3(n)$ within numerical precision and false otherwise.

Since the quaternion needs to represent a valid rotation, it has to be a unit quaternion. We need to check that it is a unit quaternion by ensuring the norm is 1.

$$\|q\| = \sqrt{w^2 + x^2 + y^2 + z^2} = 1 \tag{3}$$

Our code checks to ensure that the norm is 1 within the numerical tolerance provided.

1.3 check_SEn(matrix: m, float: $\varepsilon = 0.01$) $\rightarrow$ bool:

Input: A matrix $m \in x$ and a numerical precision value $\varepsilon = 0.01$.

Output: A boolean value that is true if $v \in SE(n)$ within numerical precision and false otherwise.

Since the group $SE(n)$ needs to represent a valid rotation, it has to have the following elements:

1. A valid rotation matrix: $R(\theta)$ that is a valid $SO(n)$
2. t that describes a valid translation
3. Last row of $SE(n)$ must be [0, 0, ... 1]

Our code checks to ensure that these conditions are met. A translation vector is arbitrary, and the rotation matrix can be checked by using the *check_SOn*() method, and ensuring the last row conditions are met within the numerical precision threshold ε

```
INPUT:
[[1 0 0]
 [0 1 0]
 [0 0 1]]
FUNCTION: correct_SOn(input)
OUTPUT:
[[1 0 0]
 [0 1 0]
 [0 0 1]]

INPUT:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
FUNCTION: correct_SOn(input)
OUTPUT:
[[-0.75271952  0.38914789  0.5310153 ]
 [ 0.38914789 -0.38759388  0.83566435]
 [ 0.5310153   0.83566435  0.1403134 ]]
```

Figure 1: correct_SOn()

1.4 correct_SOn(matrix: m, float: $\varepsilon = 0.01$) $\rightarrow$ matrix:

Input: A matrix $m \in x$ and a numerical precision value $\varepsilon = 0.01$.

Output: A matrix that is an augmented version of m such that it fits the conditions of a valid $SO(n)$.

Since the group $SO(n)$ consists of the orthogonal matrices with a determinant of 1 (representing proper rotations without scaling or reflections), we just need to ensure the following two properties:

$$m^\intercal m = I \tag{4}$$

$$\det m = 1 \tag{5}$$

Our code corrects for orthogonality by performing SVD on the input matrix M to get $U\Sigma V^T = M$, from which U and V are used to get an orthogonal approximation. Our code then corrects the determinant to equal 1 and returns the corrected matrix.

1.5 correct_quaternion(vector: v, float: $\varepsilon = 0.01$) $\rightarrow$ vector:

Input: A vector $v \in x$ and a numerical precision value $\varepsilon = 0.01$.

Output: A vector v that is a quaternion that represents a valid rotation.

Since the quaternion needs to represent a valid rotation, it has to be a unit quaternion. We need to check that it is a unit quaternion by ensuring the norm is 1.

$$\|q\| = \sqrt{w^2 + x^2 + y^2 + z^2} = 1 \tag{6}$$

Our code corrects a given quaternion to satisfy this condition by dividing the vector with it's norm, in effect turning it into a unit quaternion.

```
INPUT:
[1 0 0 0]
FUNCTION: correct_quaternion(input)
OUTPUT:
[1 0 0 0]

INPUT:
[2 2 0 0]
FUNCTION: correct_quaternion(input)
OUTPUT:
[0.70710678 0.70710678 0.         0.        ]

INPUT:
[10 10 10 10]
FUNCTION: correct_quaternion(input)
OUTPUT:
[0.5 0.5 0.5 0.5]
```

Figure 2: correct_quaternion()

1.6 correct_SEn(matrix: m, float: $\varepsilon = 0.01$) $\rightarrow$ matrix:

Input: A matrix $m \in x$ and a numerical precision value $\varepsilon = 0.01$.

Output: A matrix m that represents a valid $SE(n)$.

Since the group $SE(n)$ needs to represent a valid rotation, it has to have the following elements:

1. A valid rotation matrix: $R(\theta)$ that is a valid $SO(n)$
2. t that describes a valid translation
3. Last row of $SE(n)$ must be [0, 0, ... 1]

Our code utilizes the *correct_SOn*() method to correct the rotation matrix, keeps the translation t and ensures the last row is comprised of [0, 0, ... 1]

2 Uniform Random Rotations

2.1 random_rotation_matrix(bool: naive) $\rightarrow$ matrix:

Input: A boolean parameter *naive* to choose between the naive or efficient method to generate a random rotation matrix.

Output: A randomly generated element $R \in SO(3)$.

The method operates in two ways:

1. Naive Approach: When *naive* is set to `True`, it generates random Euler angles α, β, and γ from the uniform distribution:

$$R_z = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad R_y = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix}, \quad R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\gamma) & -\sin(\gamma) \\ 0 & \sin(\gamma) & \cos(\gamma) \end{bmatrix} \tag{7}$$

```
INPUT:
[[ 0 -1  0]
 [ 1  0  0]
 [ 0  0  1]]
FUNCTION: correct_SEn(input)
OUTPUT:
[[ 0 -1  0]
 [ 1  0  0]
 [ 0  0  1]]

INPUT:
[[3 2 4]
 [2 3 2]
 [0 1 1]]
FUNCTION: correct_SEn(input)
OUTPUT:
[[ 1.00000000e+00 -2.23711432e-17  4.00000000e+00]
 [ 1.21168839e-16  1.00000000e+00  2.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  1.00000000e+00]]
```

Figure 3: correct_SEn()

The final rotation matrix is computed as:

$$R = R_z R_y R_x \tag{8}$$

2. Efficient Approach: When *naive* is `False`, it randomly generates values x_1, x_2, z and computes the rotation angles $\theta = 2\pi x_1$ and $\phi = 2\pi x_2$. The vector V is calculated as:

$$V = \begin{bmatrix} \cos(\phi)\sqrt{z} \\ \sin(\phi)\sqrt{z} \\ \sqrt{1-z} \end{bmatrix} \tag{9}$$

The Householder matrix M is defined as:

$$M = I - 2VV^T \tag{10}$$

The resulting rotation matrix is obtained by:

$$R = MB \tag{11}$$

If the determinant of R is negative, the first column is negated to ensure it represents a proper rotation.

2.2 random_quaternion(bool: naive) → vector:

Input: A boolean parameter *naive* to choose between the naive or efficient method to generate a random quaternion.

Output: A randomly generated element $q \in S^3$

1. Naive Approach: When *naive* is set to `True`, it generates random Euler angles α, β, and γ from the uniform distribution. It then calculates the components of the quaternion (w, x, y, z) using the following formulas derived from the angles::

$$w = \cos\left(\frac{\alpha}{2}\right)\cos\left(\frac{\beta}{2}\right)\cos\left(\frac{\gamma}{2}\right) + \sin\left(\frac{\alpha}{2}\right)\sin\left(\frac{\beta}{2}\right)\sin\left(\frac{\gamma}{2}\right)$$
$$x = \sin\left(\frac{\alpha}{2}\right)\cos\left(\frac{\beta}{2}\right)\cos\left(\frac{\gamma}{2}\right) - \cos\left(\frac{\alpha}{2}\right)\sin\left(\frac{\beta}{2}\right)\sin\left(\frac{\gamma}{2}\right)$$
$$y = \cos\left(\frac{\alpha}{2}\right)\sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\gamma}{2}\right) + \sin\left(\frac{\alpha}{2}\right)\cos\left(\frac{\beta}{2}\right)\sin\left(\frac{\gamma}{2}\right)$$
$$z = \cos\left(\frac{\alpha}{2}\right)\cos\left(\frac{\beta}{2}\right)\sin\left(\frac{\gamma}{2}\right) - \sin\left(\frac{\alpha}{2}\right)\sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\gamma}{2}\right)$$

   The resulting quaternion is:

$$q = [w, x, y, z] \tag{12}$$

2. Efficient Approach: When *naive* is `False`, it generates random values s, θ_1, and θ_2 and computes:

$$\begin{aligned} s &= \text{random uniform value} \\ \sigma_1 &= \sqrt{1-s}, \quad \sigma_2 = \sqrt{s} \\ w &= \cos(\theta_2)\sigma_2 \\ x &= \sin(\theta_1)\sigma_1 \\ y &= \cos(\theta_1)\sigma_1 \\ z &= \sin(\theta_2)\sigma_2 \end{aligned}$$

   The resulting quaternion is:

$$q = [w, x, y, z] \tag{13}$$

The *visualize*() function allows us to visually see the randomly generated elements from the functions *random_rotation_matrix*() and *random_quaternion*(). The function operates as follows:

1. Quaternion Multiplication: The `quaternion_multiply` helper function computes the product of two quaternions:

$$q_{\text{result}} = q_1 \otimes q_0$$

2. Rotation Visualization: The `visualize_rotation` function applies the specified rotation to two vectors, using either quaternion or matrix representation depending on the input to `visualize(bool: quaternion)`. This uses 2 vectors $v_0 = [0, 0, 1]$ and $v_1 = [0, epsilon, 0] + v_0$. The rotation is then applied to the vectors.
   (a) Rotation Matrix: Simply performing matrix multiplication with the vectors and the rotation matrix allows us to find a resulting vector
   (b) Quaternion: In the case of a quaternion, the resulting vector v' can be found through $v' = qvq^{-1}$
3. Visualization Loop: The `visualize` function creates a 3D plot of a sphere and iterates 100 times to apply random rotations, either via quaternions or rotation matrices, resulting in a dynamic 3D representation of rotations.

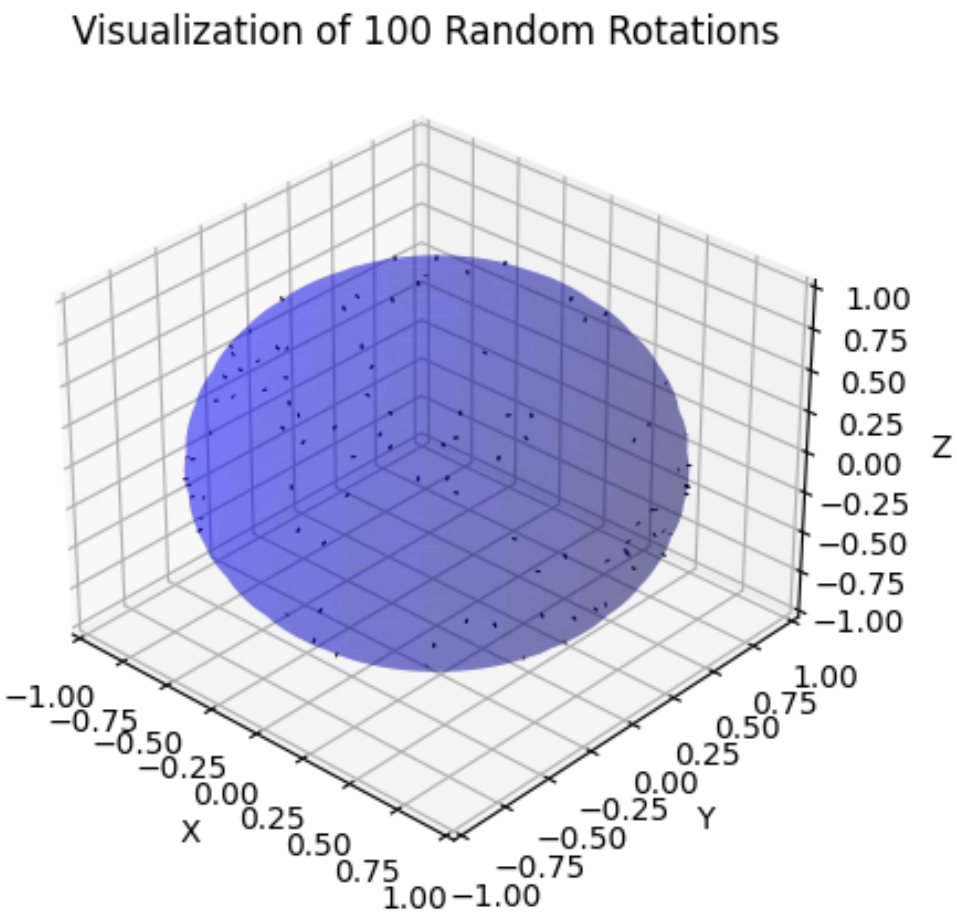

Figure 4: Visualizing Random Rotations using `random_rotation_matrix()`

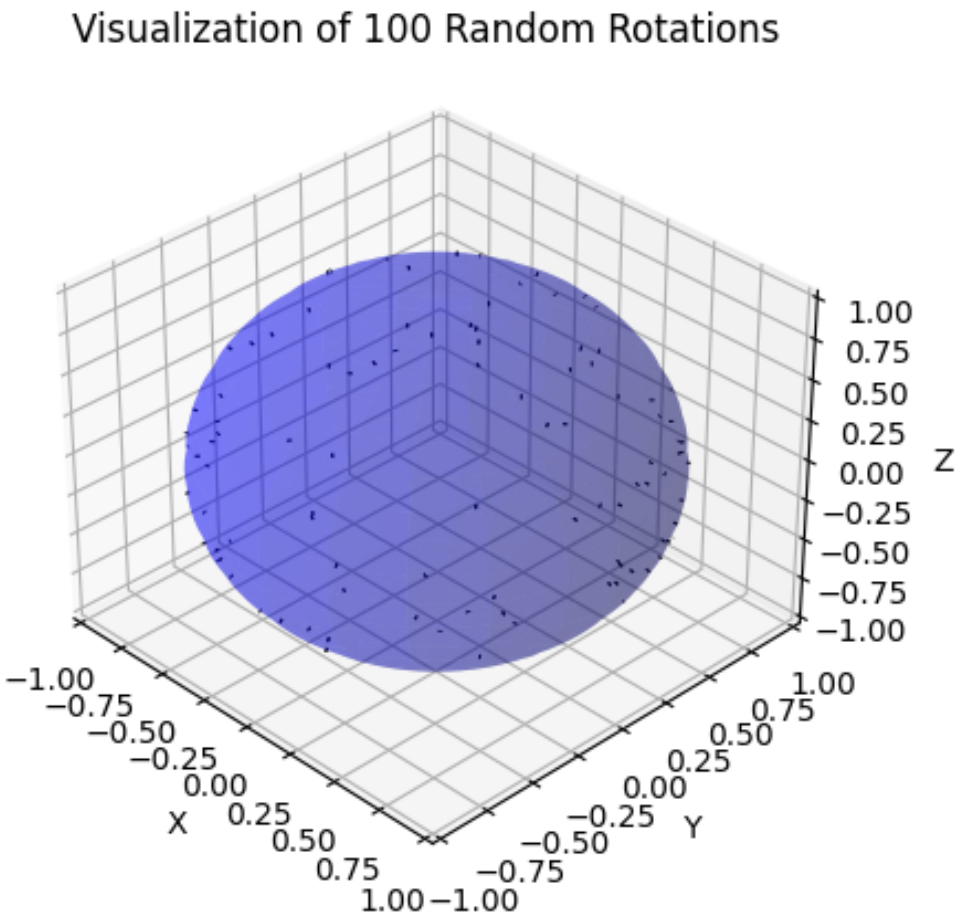

Figure 5: Visualizing Random Rotations using `random_quaternion()`

3 Rigid Body in Motion

3.1 interpolate_rigid_body(vector: start_pose, vector: goal_pose) → path:

Input: A start position $x_0 \in SE(2)$ and a goal position $x_G \in SE(2)$ (i.e. x, y, and θ).

Output: A path (sequence of poses) that start at x_0 and ends at x_G. For proof-of-concept, we chose for the path to linearly interpolate the poses over 10 steps, although more steps could be included for smoother motion.

The implemented interpolation is linear, meaning we compute intermediate poses by linearly blending the initial and final poses based on a parameter t ranging from 0 to 1. We chose this because it is straightforward and works for simple transitions where smooth motion is required. The robot moves gradually from the initial pose to the final pose in equal time steps.

Mathematically, this is done by looping through the steps, incrementing the x, y, and θ values and appending them to an array with the following formulas:

$$x = (1-t)x_0 + tx_1 \tag{14}$$

$$y = (1-t)y_0 + ty_1 \tag{15}$$

$$\theta = (1-t)\theta_0 + t\theta_1 \tag{16}$$

Where x_0, y_0, θ_0 represent the start pose and x_1, y_1, θ_1 represent the goal pose.

3.2 forward_propagate_rigid_body(vector: start_pose, Plan: plan) → path:

Input: A start pose $x_0 \in SE(2)$ and a Plan (a sequence of N tuples (velocity, duration)).

Output: The resulting path of $N+1$ states from applying the plan to the start pose.

In each step, the pose is updated based on the velocity applied over the given time. This method is ideal for simulating real-time control inputs. Each velocity vector changes the position and orientation of the robot incrementally. This allows for simulating more complex movements where the robot changes velocity or direction.

Mathematically, this is done by looping through the plan, incrementing the x, y, and θ values and appending them to an array with the following formulas:

$$x_{\text{next}} = x_{\text{current}} + v_x d \tag{17}$$

$$y_{\text{next}} = y_{\text{current}} + v_y d \tag{18}$$

$$\theta_{\text{next}} = \theta_{\text{current}} + v_\theta d \tag{19}$$

Where v_x, v_y, v_θ represent the velocities of each step of the plan and d represents duration of each step of the plan.

3.3 visualize_path(path):

Input: A path to visualize.

Output: A graph with the robot's path through the environment visualized as well as an animation of the robot's movement.

The robot is visualized as a rectangle, and its position and orientation are updated frame-by-frame in the animation. Matplotlib animations were chosen to enable smooth visualization of the motion. The robot's orientation is dynamically updated using the angle property of the rectangle, and the robot's path is plotted to track its progress.

Note: Gifs can be found in the zipped folder.

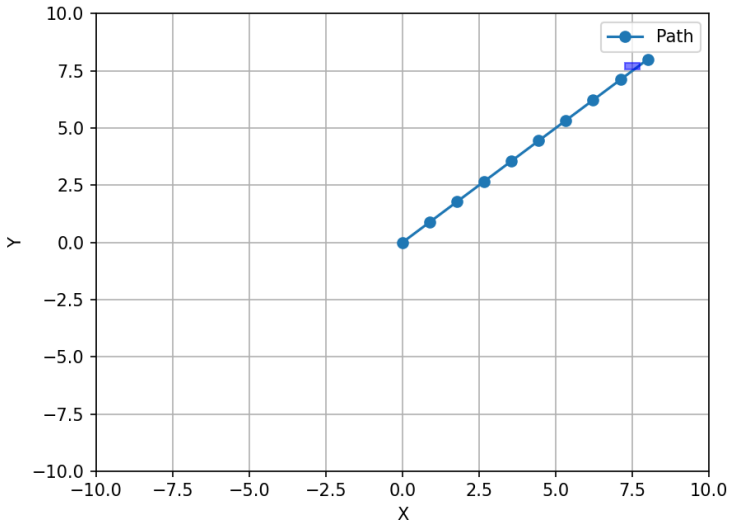

Figure 6: Visualizing Robot Motion With `interpolate_rigid_body()`

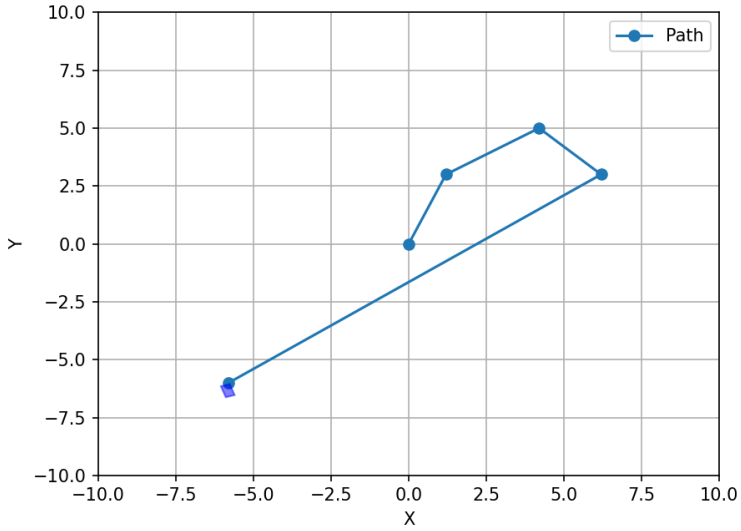

Figure 7: Visualizing Robot Motion With `forward_propagate_rigid_body()`

4 Movement of an Arm

The arm has 2 links: the first with a length of 2 units and rotates about joint J_0 (the origin) by an angle θ_0 and the second with a length of 1.5 units and rotates about joint $J1$ (the end of link 1) by an angle θ_1. The total rotation for link 2 is $\theta_0 + \theta_1$.

The frame associated with link 1 is located at its center. However, the calculation starts from the bottom of the link (i.e., joint J_0). Similarly, link 2's frame is at its center, but the position is computed from the end of link 1.

To compute the position of the arm's joints in space, the code applies the 2D rotation matrix corresponding to the joint angles to rotate each link appropriately and compute its new end-point.

4.1 interpolate_arm(vector: start, vector: goal) → path:

Input: A start configuration $q_0 = (\theta_0, \theta_1)$ and a goal configuration $q_G = (\theta_0, \theta_1)$.

Output: A path (sequence of joint angles) that start at q_0 and ends in q_G. For proof-of-concept, we chose for the path to linearly interpolate the poses over 10 steps, although more steps could be included for smoother motion.

Similar to interpolate_rigid_body(), this was calculated mathematically by looping through the steps, and incrementing θ_0 and θ_1 values and appending them to an array with the following formulas:

$$\theta_0 = (1 - t)\theta_{00} + t\theta_{01} \tag{20}$$

$$\theta_1 = (1 - t)\theta_{10} + t\theta_{11} \tag{21}$$

Where θ_{00}, θ_{01} are the starting angles for links 1 and 2 respectively and $\theta_{10}, \theta 11$ are the goal angles for links 1 and 2.The interpolation between joint angles is linear, providing a smooth transition from the start to the goal configuration.

4.2 forward_propagate_arm(vector: start_pose, Plan: plan) → path:

Input: A start pose $q_0 \in SE(2)$ and a Plan (a sequence of N tuples (velocity, duration)).

Output: The resulting path of $N + 1$ states from applying the plan to the start pose of the arm.

We generate paths by simulating the application of velocities to the joints over specific durations. Each time a velocity command is applied for a certain duration, the joint angles are updated.

Mathematically, this is done by looping through the plan, incrementing the θ_0, θ_1 values and appending them to an array with the following formulas:

$$\theta_{0,\text{next}} = \theta_{0,\text{current}} + v_0 d \tag{22}$$

$$\theta_{1,\text{next}} = \theta_{1,\text{current}} + v_1 d \tag{23}$$

Where v_0, v_1 are the two angular velocities of each step and d is the duration of each step.

4.3 visualize_arm_path(path):

Input: A path to visualize.

Output: A graph with the robot's arm's path through the environment as well as an animation of the robot arm's motion.

The robot is visualized two blue links and its path is represented using a red curve that updates dynamically as the robot arm moves.

Note: Gifs can be found in the zipped folder.

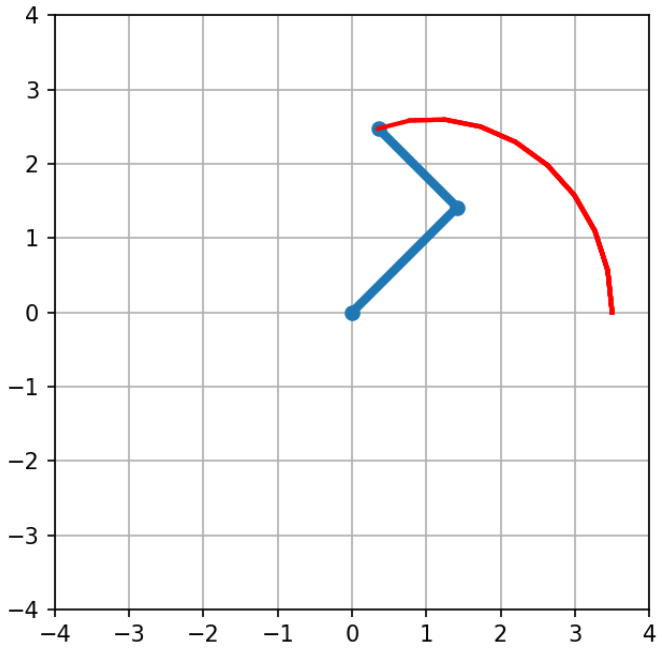

Figure 8: Visualizing Arm Motion With `interpolate_arm()`

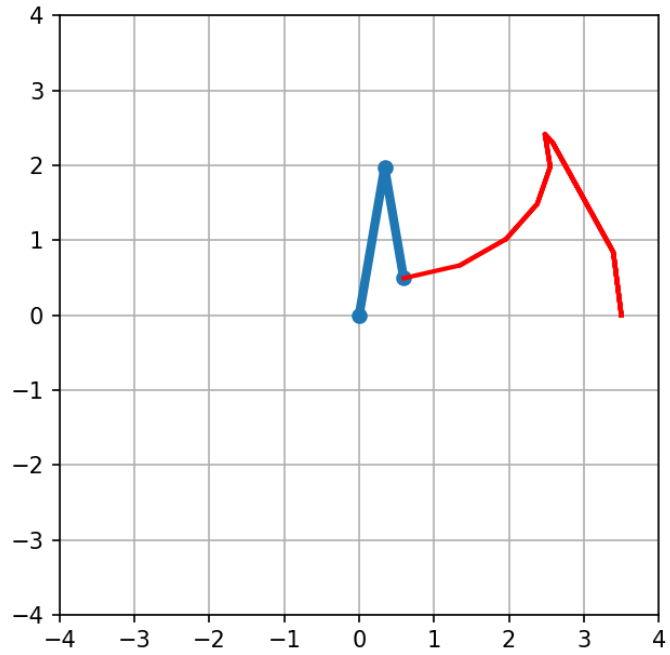

Figure 9: Visualizing Arm Motion With `forward_propagate_arm()`