CS 460 Assignment 1 Report

Theodor Novitsky (tn347), Jad Ziab (jgz22)

Component 1: Validating Rotations

check_SOn(matrix: np.array, epsilon: float=0.01) → *bool*

Takes in an array which represent a matrix to check. Aswell as an option to customize an epsilon. To do so, we first check if the inputted matrix is a square matrix, then we check if the transposed matrix times the original matrix is equal to the identity matrix within the certain epsilon. Finally, if the determinant of the original matrix is 1. Then we can conclude the inputted matrix is indeed SO(n).

We have also implemented test variables you could play with in order to test our functions reliability. (theta, rotation_matrix_2D, rotation_matrix_3D_aroundX)

check_quaternion(vector: np.array, epsilon: float=0.01) → *bool*

Takes in a vector (vertical column np.array) and optional epsilon. To check if given vector is a quaternion, first we check its shape, in the assignment details you mention we want to check if the quaternion belongs to s3. So, the shape has to be very specific. We check that the shape is (4,1), then take the sum of the squares of each component of the vector. And if the result is close enough to 1, then we can conclude the inputted vector is a quaternion.

Also implemented test variables (quaternion_vector, quaternion_vector2).

check_SEn(matrix: np.array, epsilon: float=0.01) → *bool*

Takes in a matrix to check and optional epsilon. This function checks whether inputted matrix belongs to either SE(2) or SE(3) as described in the write up. To do, we first check if matrix is square. Then we find n specific to our inputted matrix. Then we find R which is the first nxn box in the top left. Then we find t which is the rightmost column, with n vertical values. Then we find bot which represents the entire bottom row of our inputted matrix. In order for the matrix to be SE(n), R must be SO(n), so we utilize our check_Son function to do this. Then we check if the bottom row values are all 0 except for the right most value. If both are true then we can conclude our matrix is indeed Se(2) or Se(3).

Additionally in component 1, we utilize helper functions "create_se2_matrix", and "create_se3_matrix". From utils.py file. In this file I store some random helper functions which we pull from during various parts of the project. I use these functions to test our check_Sen function. All these functions do is take theta and position values and creates a SE2 matrix or SE3 matrix based on these values.

Component 2: Uniform Random Rotations

random_rotation_matrix(naive: bool) → *np.array*

Takes in a Boolean which defines either naïve or uniform sampling for generating random rotations. If True, then we generate random rotations naively. The way I implemented this is by generating random yaw, pitch, roll angles using np.random.uniform. This part was a bit confusing to us because in the write up it was suggested to use random euler angles. And by doing so, my spheres were meant to cluster on the poles. However, regardless of which distribution I used for the naïve method. The rotations seemed to spread uniformly regardless of the inputted Boolean. Regardless after I had generated the angles I create a rotation matrix for all the dimensions (R_x, R_y, R_z) using helper functions I had defined in utils.py. Then I create a single rotation matrix R from R_z, R_y, R_x by multiply them in that order. Then I would return R.

If the Boolean inputted was false then I would use the given algorithm

for "correctly" (non-naively) generated uniform rotations given below from "fast_random_rotation_matrices.pdf". And return M.

random_rotation(x_1, x_2, x_2, M)

x_1, x_2, x_3 : **real**; *Three random variables.*

M : **matrix3**; *The resulting matrix.*

begin

$\theta \leftarrow 2\pi x_1$; *Pick a rotation about the pole.*

$\phi \leftarrow 2\pi x_2$; *Pick a direction to deflect the pole.*

$z \leftarrow x_3$; *Pick the amount of pole deflection.*

Construct a vector for performing the reflection.

$$V \leftarrow \begin{bmatrix} \cos\phi\sqrt{z} \\ \sin\phi\sqrt{z} \\ \sqrt{1-z} \end{bmatrix}$$

Construct the rotation matrix by combining two simple rotations: first rotate about the Z axis, then rotate the Z axis to a random orientation.

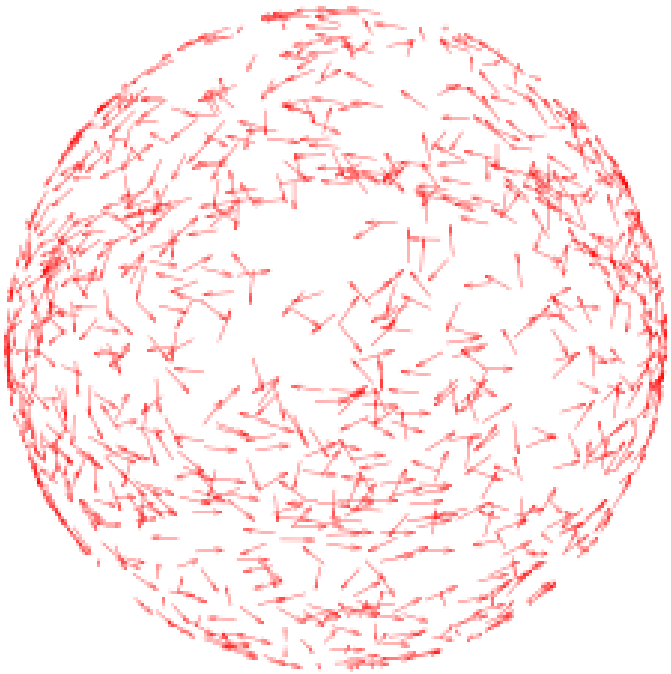
$$M \leftarrow (2VV^{\mathrm{T}} - I) \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

end

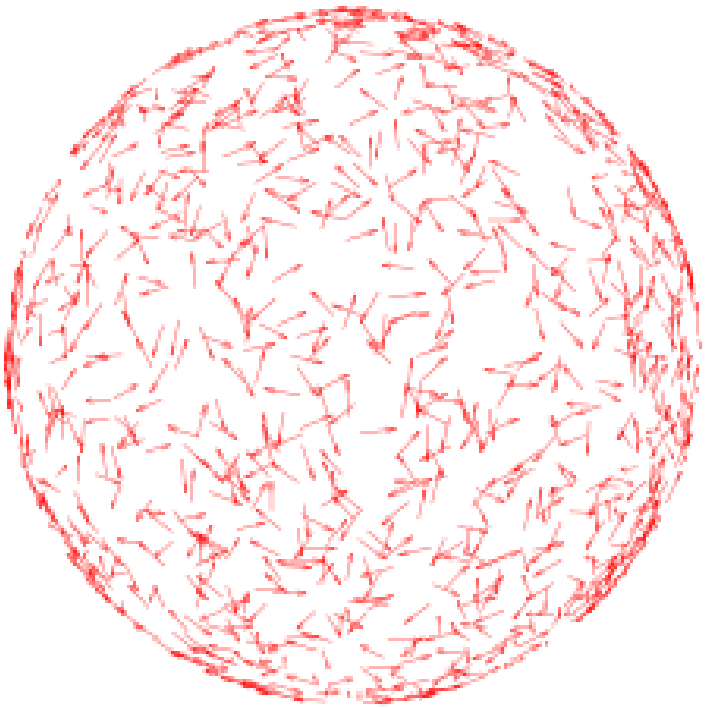
For visualizing the random rotations, I created a helper function called "generate_rotation_visualization" which took in a number of samples, boolean for

sampling method, and pngNum (useless just for saving pngs). I followed the Algorithm 1 given in the write up for rotation visualization for creating the generate_rotation_visualization function. MatPlotLib was used throughout the entire project for creating the visuals. It was difficult to work on the virtual machine with MatPlotLib and I was unsure if you would be able to see my plots through your machine by just using plt.show(). Therefor, I made it so that the plot saves as a png in the current folder your in. The plots generated for this component are respectively called "naiveSampling.png" and "uniformSampling.png". They can be seen below aswell.

NAÏVE SAMPLING 1000 ROTATIONS



UNIFORM SAMPLING 1000 ROTATIONS

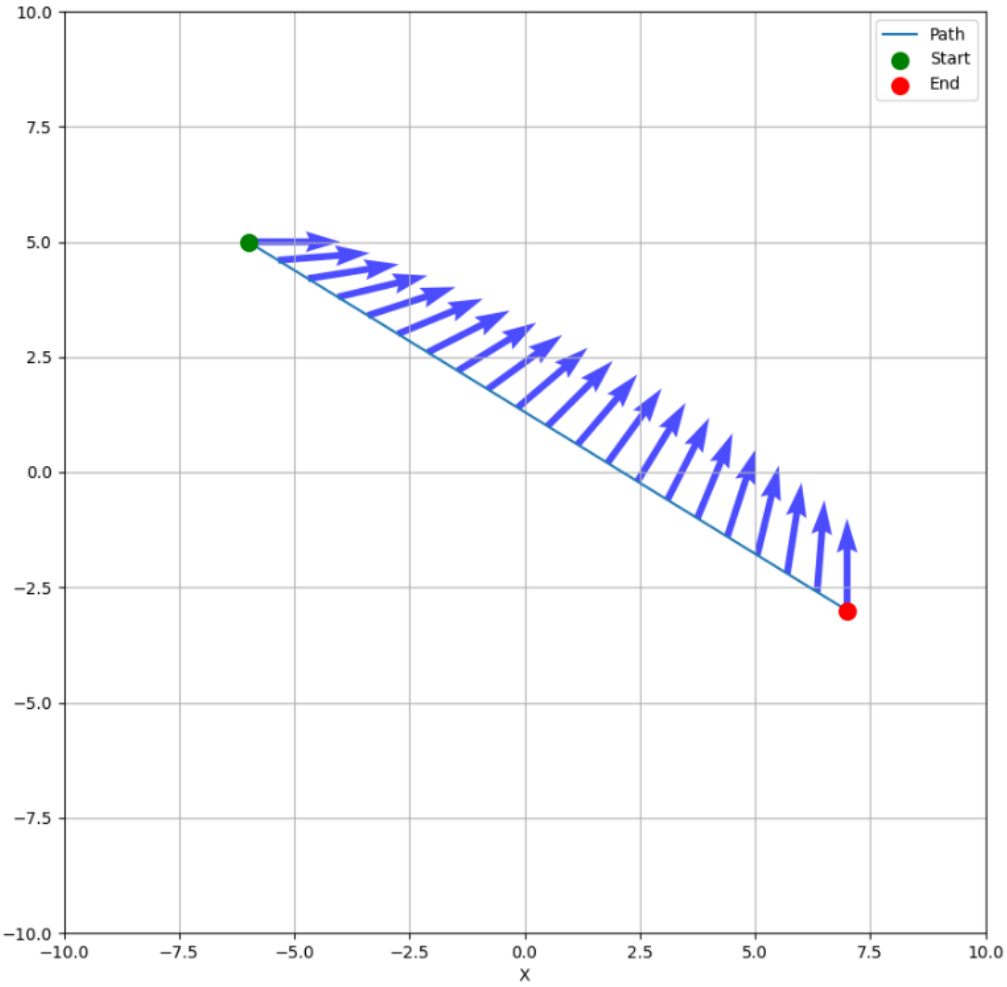


Component 3: Rigid Body in Motion

interpolate_rigid_body(start_pose: np.array, goal_pose: np.array) → *np.array*

Takes in a start pose and a goal pose. Poses are defined a np.array where the first 2 values are x,y positions, and the 3rd value is the theta position. Our function works by calculating the increments we should have for each component of the rigid body based on the number of steps in your path you want to take. This can be manually changed in our function and is represented by the variable "n". Then we run a simple for loop and appended each new position to our path.

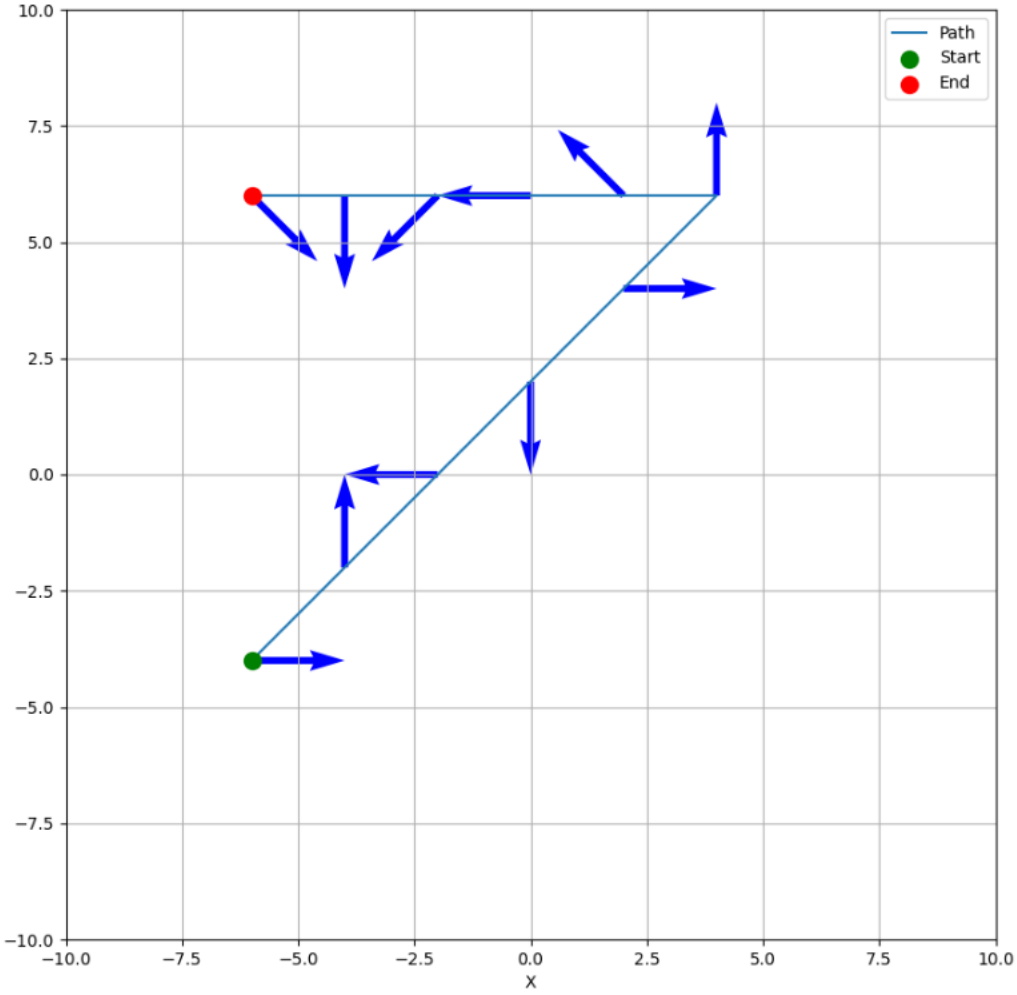
For the testing/plotting of this component we set the start pose at (-6,5) with a rotation of 0 (facing rightward). Our goal pose is (7,-3) with a rotation of pi/2(facing upward). We use MatPlotLib to visualize this process. The output can be observed below and in 'interpolate_rigid_body_Test.png' file.



forward_propagate_rigid_body(start_pose: np.array, plan: list) $\rightarrow$ *np.array*

This function takes in a start pose and a plan. A plan is defined as a List of Tuples where each tuple contains a velocity for each component x, y, theta. Aswell as a duration(number of steps). This way we can have our rigid body, move dynamically. Our function works by iterating through each tuple in the plan and then iterating through each step in the tuple, and appending the new position in the path. Because we append a new position for every step, the new position can be calculated simply with new_pos = old_pos + velocity. (Velocity represents change in component per step so this is okay).

For our testing, we defined out plan in 2 phases.
-Start Pose = (6, -4, 0). No rotation (facing rightward)
-Phase 1: Velocities for (x,y,theta) = (2, 2, pi/2) . 5 Steps
-Phase 2: Velocities for (x,y,theta) = (-2, 0, pi/4) . 5 Steps
The output can be observed below and in 'forward_propagate_rigid_body_Test.png'



visualize_path(path)

This function takes in a path and visualizes it in an animation. To do this, we start with an init() function to initialize the first frame of the animation. This is done by simply inputting the starting position. Next we have an update() function that updates the robots position (x, y), orientation (theta), and directional arrows (theta). We use the function np.degrees(theta_vals[frame]) to update the robot's rotation. The directional arrows are updated based on the rotation/orientation, which is then recalculated with dx and dy (we use dx = np.cos(theta_vals[frame]) * 0.5 and dy = np.sin(theta_vals[frame]) *

0.5)). This loops for each frame of the animation. We then use the FuncAnimation function (imported from MatPlotLib) to visualize the animation. We save the animation into a gif using ani.save('rigid_body_path_test.gif', writer='imagemagick').

In our test, we use the same path and start pose from our forward_propagate_rigid_body test. You can view the gif by opening the "rigid_body_path_test.gif" file. To test out different paths, create a new path and input it into this function. It'll update the gif file each time you run the program. Also if you wanted the visualization to be more fluid, then in your path you would need to include more steps(Make sure to adjust the velocity per step in ratio with the steps to mimic the exact motion of the body).

Component 4: Movement of a Arm

interpolate_arm(start_angles: np.array, goal_angles: np.array) → *np.array*

This function takes start_angles which is a np.array of 2 values, which represent the angle of the joints. The first angle for joint 1 and 2nd angle for joint 2.

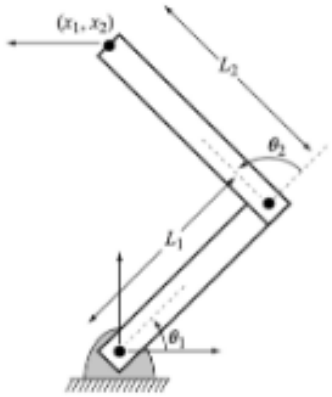
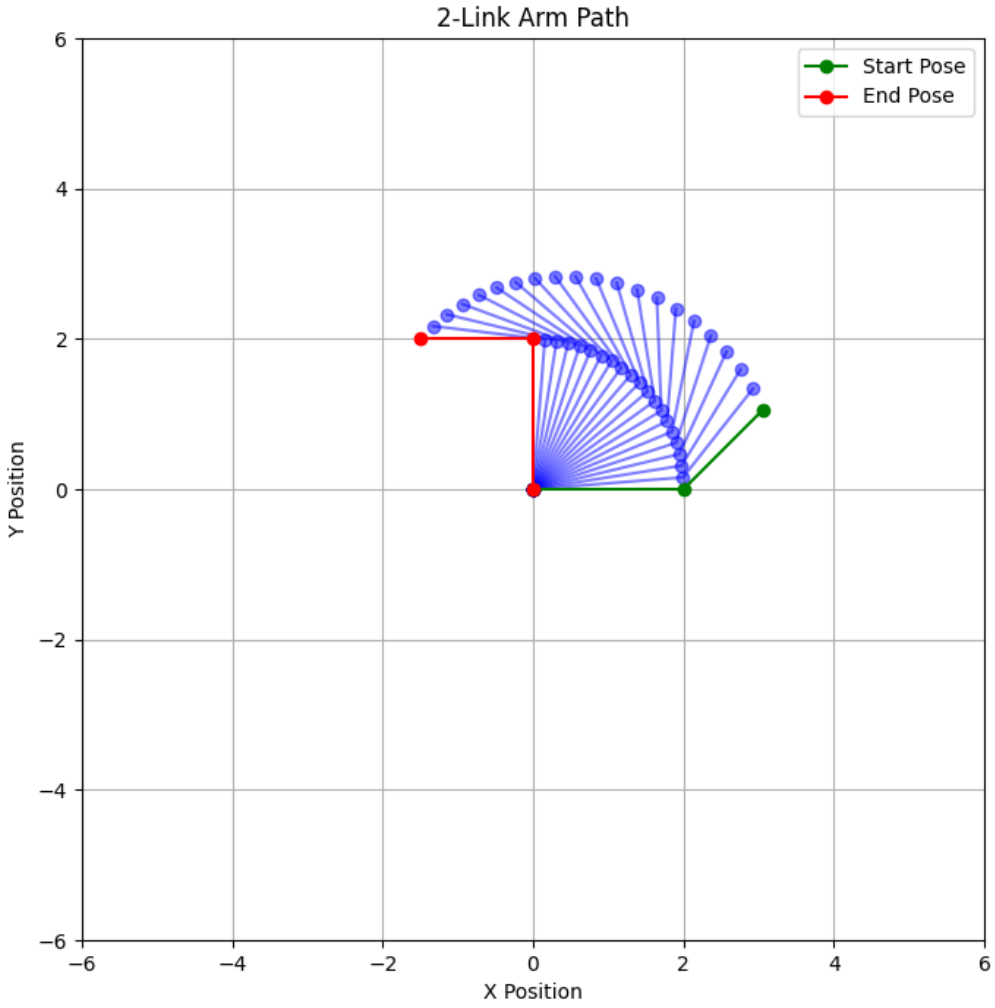


Figure 1: Robot Diagram: 2-link planar manipulator

We made this function work very similarly to interpolate_rigid_body(). It finds the right increments for both angles given a number of steps (n) which can be changed. Then appends the new pose after incrementing the angles. Then returns the full path.

For the plotting of this function. We defined a test case where the start angles are (0 ,pi/4) and goal angles (pi/2, pi/2). Its important to note that theta2, is using link1 as its reference. Not the base. Theta1 is using the ground as its reference. We decided to do it this way because this is how the photo depicted it. Additionally we created a plot_arm_path function for the visualization, this can be found in the utils.py file. The file it saves in is called "interpolate_arm_Test.png", can also be seen below.



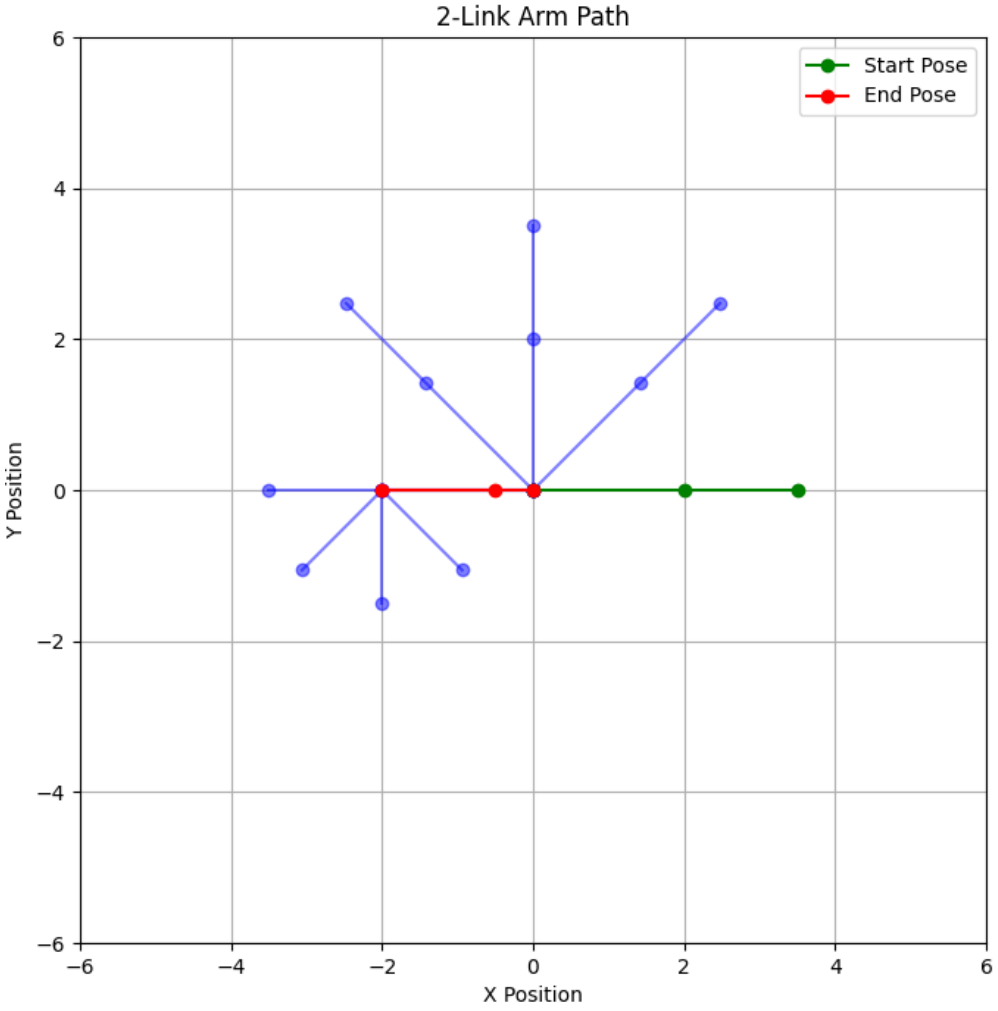
forward_propagate_arm(start_pose: np.array, plan: list) $\rightarrow$ *np.array*

This function once again works similarly to forward_propagate_rigid_body(). But now our plan is a list of tuples where each tuple has a pair of angular velocities (theta1,theta2) and a duration(steps). The function iterates through each tuple and then iterates through each step and calculates the

new angular positions per step and appends those positions to the path.

For our test case, we set the starting pose at (0,0) and defined the plan in 2 phases. We plot using the same function we used to plot for interpolate_arm (plot_arm_path()) which is defined in utils.py.

Phase 1: Angular velocities for (theta1, theta2) = (pi/4 , 0). 4 Steps
Phase 2: Angular velocities for (theta1, theta2) = (0 , pi/4). 4 Steps
Output can be observed in "forward_propagate_arm_Test.png" or below.



visualize_arm_path(path)

The function takes a path as the input and the function creates the visualization of the robot arm movement in the form of an animation. Similar to the visualize_path function, the visualize_arm_path function utilizes the FuncAnimation function (imported from MatPlotLib). The init() function sets both links to empty lists and returns link1 and link2. This makes it so nothing will display for the first frame. The update() function checks the angles of each joint on every frame. It also uses a function forward_kinematics(theta1, theta2), which we implement in utils.py. This function returns the coordinates (x, y) of the two joints given the angles of the joints. The update function then updates link1 and link2 to show the first and second segments respectively. The update function is called for every frame of the animation. The animation is saved when we run ani.save('movementOfaArm_test.gif', writer='imagemagick'). For our test case we used the same start_pose and plan as we did in forward_propagate_arm().

Summary of Files

- component_1.py : python file for Part 1 (Validating Rotations)
- component_2.py : python file for Part 2 (Uniform Random Rotations)
  - naiveSampling.png : png file for naive sampling
  - uniformSampling.png : png file for uniform sampling
- component_3.py : python file for Part 3 (Rigid body in Motion)
  - interpolate_rigid_body_Test.png : png file for testing interpolate_rigid_body function
  - forward_propagate_rigid_body_Test.png : png file for testing forward_propagate_rigid_body function
  - rigid_body_path_test.gif : gif file for testing visualize_path function
- component_4.py : python file for Part 4 (Movement of a Arm)
  - interpolate_arm_Test.png : png file for testing interpolate_arm function

  - forward_propagate_arm_Test.png : png file for testing forward_propagate_arm function
  - movementOfaArm_test.gif : gif file for testing visualize_arm_path function

- utils.py : python file with various helper functions
- report.pdf : pdf report
- report.tex : latex doc we used to create pdf report