Lab 2

Total points: 100

Part 1: URDF

1. From the xacro file, can you determine the lengths of each link of the robot arm? What are the lengths? Given these, as well as the visualization of our robot arm in yourdfpy, what do you think the maximum range of the robot arm is (approximately)? Note that units are in meters.

You can determine the length of each link of the robot arm. The xarco file has lengths for each arm stored in a <xarco:property> component near the top of the file. The shoulder is **0.089m**, the upper arm is **0.42m**, the forearm is **0.39m**, and the wrists are **0.1m**, **0.09m**, and **0.08m** respectively. The arm should have a range of roughly **1m**, give or take a little bit depending on how the joints need to be angled to reach a given point.

Part 2: Inverse Kinematics

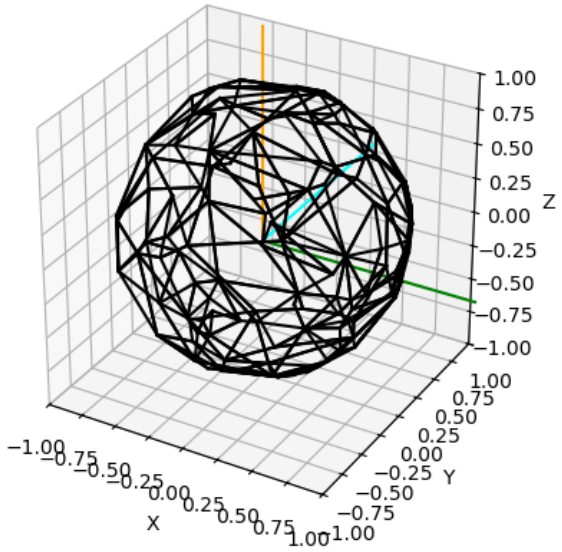
2. Can you get your robot end effector to the maximum range you estimated in Q1, along any of the x/y/z axes? Why or why not?

The end effector cannot reach one meter of range, however it does get very close. I was counting the shoulder along with the wrists in my range calculation, when I should have left those out since they primarily contribute rotation rather than distance.

2. Write code to find and visualize the 3D volume of points that are reachable by the robot arm. Hints: the forward kinematics function gives you the actual position of the end effector for a given target position. The target position is visualized as a red dot in the original plotting code. You want to randomly sample from points on a large sphere around the robot, and attempt to get the robot to

reach those points, such that the arm is fully "stretched". Record the resulting points, and visualize. You may want the "Convex Hull" method in the "scipy" library, as well as the method plot_trisurf from the matplotlib library. AI is allowed for producing the visualization code, but not for sampling from the sphere. Sampling points on the surface of a sphere is a non-trivial problem.

```
import ikpy.chain
import numpy as np
import ikpy.utils.plot as plot_utils
import matplotlib.pyplot as plt
from scipy.spatial import ConvexHull

my_chain  = ikpy.chain.Chain.from_urdf_file("./ur5/ur5_gripper.urdf")
num_samples = 128
r = 2
u = np.random.uniform(low=-r, high=r, size=(num_samples))
theta = np.random.uniform(low=0, high=2*np.pi, size=(num_samples))
end_points = np.empty((num_samples, 3))

for i in range(num_samples):
    k = u[i]
    j = theta[i]
    x = np.sqrt(np.square(r)-np.square(k)) * np.cos(j)
    y = np.sqrt(np.square(r)-np.square(k)) * np.sin(j)
    z = k
    target = [x, y, z]
    end_pos = my_chain.forward_kinematics(my_chain.inverse_kinematics(target))
    end_points[i] = end_pos[:3,3] # Gets pos. of end effector

hull = ConvexHull(end_points)
fig, ax = plot_utils.init_3d_figure()
ax.set_box_aspect((1, 1, 1))
for simplex in hull.simplices: # https://docs.scipy.org/doc/scipy/reference/generated/scipy.
    plt.plot(end_points[simplex, 0], end_points[simplex, 1], end_points[simplex, 2], 'k-')
plt.show()
```



Part 3: Collision Checking

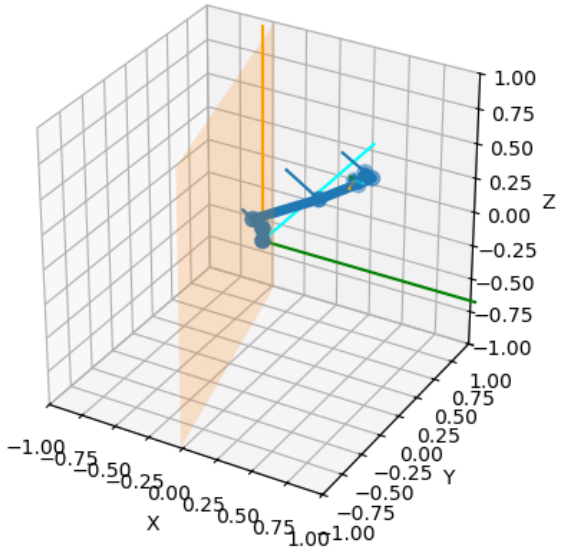
4. Define and visualize a plane in the space with the robot arm; the plane should not intersect the robot arm when all the joints are set to zero position. Write code to determine if any part of the robot arm is intersecting the plane for an arbitrary pose of the robot arm, and include at least three test cases for your method.

```
from os.path import join
import ikpy.chain
import numpy as np
import ikpy.utils.plot as plot_utils
import matplotlib.pyplot as plt

def collision_detection(plane_vector, plane_point, target_position):
    my_chain  = ikpy.chain.Chain.from_urdf_file("./ur5/ur5_gripper.urdf")

    fig, ax = plot_utils.init_3d_figure()
    ax.set_box_aspect((1, 1, 1))

    real_frame = my_chain.forward_kinematics(my_chain.inverse_kinematics(target_position), f
```

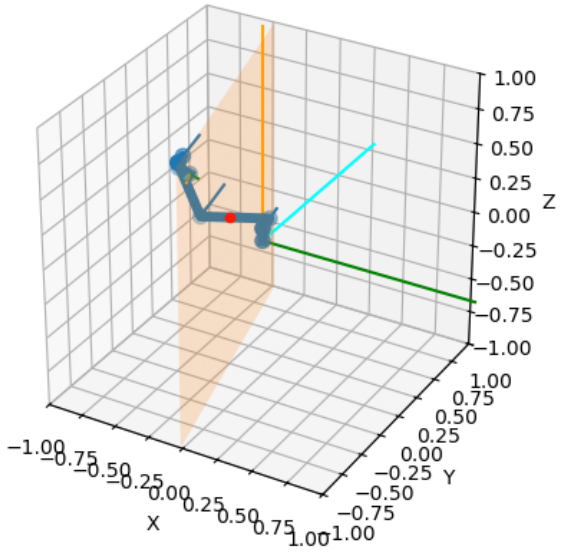
```
    joint_positions = [x[:3, 3] for x in real_frame]

    my_chain.plot(my_chain.inverse_kinematics(target_position), ax)

    axes = [0, 1, 2]

    plane_vector = plane_vector / np.linalg.norm(plane_vector) # Get unit vector

    main_axis = np.argmax(plane_vector)
    axes = list(filter(lambda x: x != main_axis, axes))

    a = np.linspace(-1, 1, 10)
    b = np.linspace(-1, 1, 10)
    a, b = np.meshgrid(a, b)
    c = (a * (-plane_vector[axes[0]] * plane_vector[main_axis]) + b * (-plane_vector[axes[1]

    meshes = [None] * 3
    meshes[axes[0]] = a
    meshes[axes[1]] = b
    meshes[main_axis] = c

    ax.plot_surface(meshes[0], meshes[1], meshes[2], alpha=0.20, shade=False)

    for i in range(len(joint_positions) - 1):
        distance = np.abs(np.subtract(joint_positions[i], joint_positions[i+1]))
        magnitude = np.linalg.norm(distance)
        unit = distance / magnitude

        intersect = np.dot(unit, plane_vector)
        if intersect != 0:
            d = (np.dot((plane_point - joint_positions[i]), plane_vector)) / intersect
            collision = joint_positions[i] + unit * d
            collision_dist_a = np.linalg.norm(np.abs(np.subtract(joint_positions[i], collisi
            collision_dist_b = np.linalg.norm(np.abs(np.subtract(joint_positions[i + 1], col
            if collision_dist_a <= magnitude and collision_dist_b <= magnitude:
                print("Collision on joints", i, i+1)
                ax.scatter(collision[0], collision[1], collision[2], color='red')
                break
    plt.show()
```

Below are the test cases in code along with their output image. The planes are defined by a vector along with a point. If a collision occurs it is represented in the visualization by a red dot.

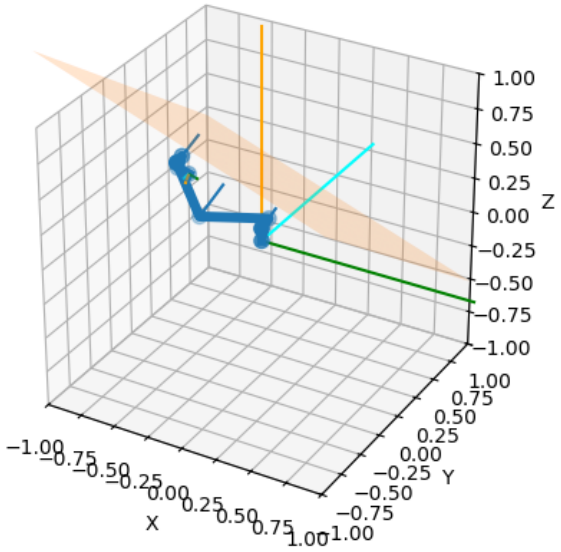
```
collision_detection([1,0.25,0],[-0.25,0,0],[0.25,0.75,0.15])
```




```
collision_detection([1,0.25,0],[-0.25,0,0],[-0.5,0,0.25])
```




```
collision_detection([0.5,0.5,0.75],[0,0,0.5],[-0.5,0,0.25])
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Z
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Y
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
X