MEAM520 Lab 3
Divyanshu Rajendra Sahu
Vanshil Atul Shah

6.1 Methods

The description of the helper functions is as follows.:

1) Displacement and Axis: The input for this function is the current and the target position. Using this we calculate the displacement and the axis of rotation to convert from the current to the target frame with respect to the world.

   Given
   $current = T_c^0$
   $target = T_t^0$

   We calculate the displacement by simply taking the difference of the position vectors of the current and the target position. We extracted the position vector from the transformation matrix.
   $Displacement = o_t^0 - o_c^0$

   To calculate the axis of rotation, we first calculate the relative rotation of the target frame with respect to the current end effector frame.
   $R_t^c = (R_c^0)^{-1} R_t^0$

   Then we take the skew symmetric part of the rotation, and stack the elements to yield the axis of rotation which, currently, is described in the current end effector frame. Now, to get the rotation axis w.r.t the world frame, we multiply the calculated axis vector with the rotation matrix of the current frame which converts this axis coordinates w.r.t world frame.The axis of rotation is the axis along which the current frame will rotate by an angle of θ to convert from the current frame to the target frame.

   $S = 0.5(R_t^c - (R_t^c)^T)$
   $axis^{current} = [a1, a2, a3]$
   $axis^0 = R^0_{current} axis^{current}$

2) Distance angle function: In this function, the input is G and H transformation matrix each of dimension 4*4. We calculate the distance between these two frames and the angle between them.

Given the matrices G and H, we calculate distance between the current pose and the desired target as the L2 norm of the displacement vector. We extract the current and the target pose from as the last column of the transformation matrix excluding the last element ($G[:-1,3], H[:-1,3]$)

$$distance = ||displacement||_2$$

We use the hint given to us in the Lab3.pdf to calculate the angle between current and target frame which is calculated using the formula given below.

$$|\theta| = arccos((tr(R) - 1)/2)$$

First we calculate the value of k which is given as follows:

$$k = (tr(R) - 1)/2$$

We then constrain the value of k between 1 and -1 so that the arccos function of python does not throw an error.

3) Is.valid.solution: In this function we have to calculate the success of our solution given the candidate solution and the target.

   In this we check three conditions for the final solution to be a valid solution.

   1) The final solution of joint angles should lie between the upper limit and lower limit
      $\text{lower joint limits} < q\_final < \text{upper joint limits}$

   2) The distance between the final end effector pose and target pose should be less than the linear tolerance provided.
      $\text{distance} < \text{linear distance tolerance}$

   3) The angle between the final end effector pose and target should be less than the angular tolerance provided.
      $\text{angle} < \text{angular tolerance}$

   If all of the above conditions are true then we return success = True else we return False.

4) End_effector_task:
   This function implements the primary optimization task i.e the end effector should reach the desired target. Here we equate the desired end effector linear velocity as displacement between current and target frame and desired angular velocity as axis of rotation of current frame to reach target frame with magnitude of sin theta. For a given displacement and axis of rotation from current to target frame and the current configuration of the robot, we find joint velocities using IK_velocity function which we implemented in Lab2.

5) joint_centering_task.
   The input to the function are the joint angles and a predetermined rate. This function implements the secondary task i.e the joint angles should be centered. Here this is achieved by calculating a velocity vector which is proportional to the offset of joint angles with the joint center. Means if there is more offset then the velocity would be more and that too in a direction which counters this offset. The rate determines how strongly one wants to center the joint.

   $v = \text{rate}^*\text{-offset}$

6) Inverse Optimisation loop: This is the main function where everything we have implemented is used to make the robot go from a current to a target position.

   We perform a gradient descent type optimization to make our current pose go near the target frame. For minimising the error between the current and the target pose, we take the gradient step as the velocity equal to our displacement and our angular velocity equal to our axis. The displacement and axis vector help us in going toward the target frame. So if we take velocity and angular velocity equal to displacement and axis and make our end effector follow that velocity profile the end effector would reach the target position.

   Here, as we close in on the goal, the velocity and angular velocity becomes zero and we converge in our solution analogous to a gradient of a function when it is near local minimum.

   We add our solution that we got from the end effector task with the joint centering task projected on the nullspace of Jacobian. So this projection on the null space doesn't have any effect on the end effector velocity as it's in the null space. The primary task tries to make the robot go from its current to its target position, whereas the secondary task tries to center the robot. Hence, together the two tasks make the robot go from the current to the target position as well as respect the joint of the robot.

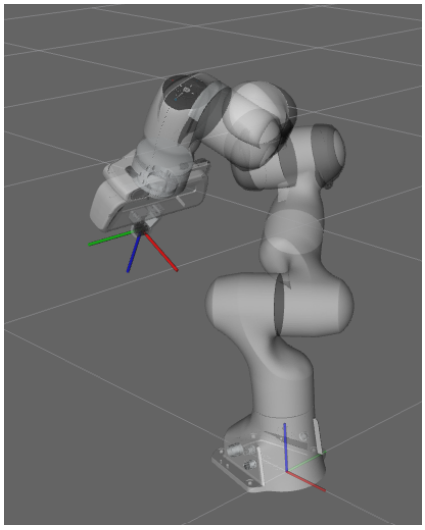
6.2 Evaluation

A brute force way of checking the correctness of our solution was by passing the final solution of our optimiser to the forward kinematics class which we had implemented in Lab1 and comparing the output to the target position that was specified.

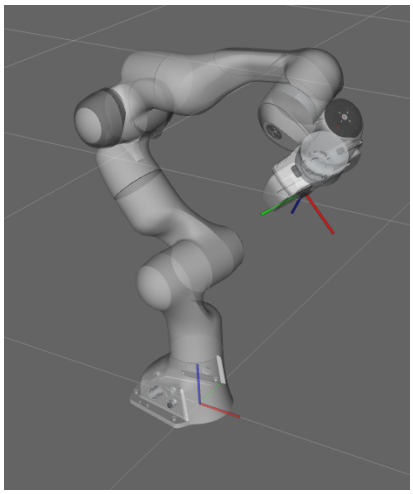
We also added various breakpoints in our code to check the distance,angle,axis and omega. The print statements already given in the code also aided in debugging. The distance and angle showed a decreasing trend when the solver converged to the right solution which gave a physical intuition on how the solver was behaving.

The IK.is_valid_solution() function implementation also helped us to realise in which condition our solver was wrong and thus sped the debugging process. Other than this, we also added several new configurations and checked the behavior of our optimiser and the solutions it created.

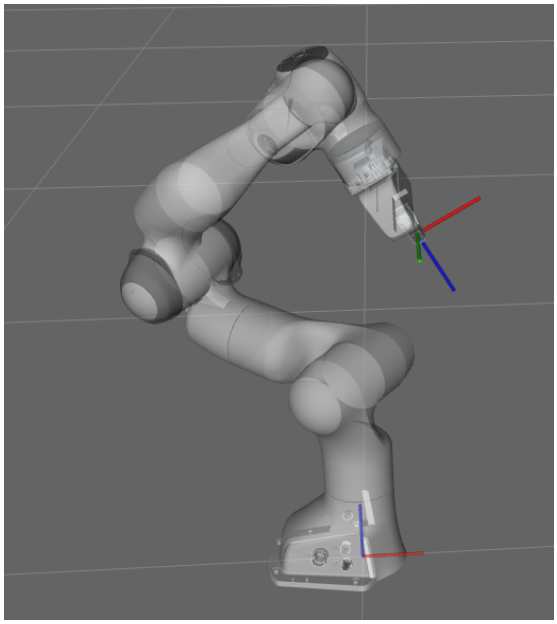
New poses:



End effector target coordinates and orientation in consideration [-0.1, -0.2, 0.5], [0,5/6*pi,pi]



End effector Target coordinates and orientation in consideration [0.2, 0.1, 0.5], [pi/6,5/6*pi,7/6*pi]



End effector Target coordinates and orientation in consideration [.1, -0.1, .6], [0,7/6*pi,pi]

Performance Statistics:

```
targets = [
    transform( np.array([-.1, -.2, .5]), np.array([0,5/6*pi,pi])         ),
    transform( np.array([.2, .1, .5]),  np.array([pi/6,5/6*pi,7/6*pi]) ),
    transform( np.array([.1, -0.1, .6]),    np.array([0,7/6*pi,pi])            ),
    transform( np.array([.7, 0, .5]),    np.array([0,pi,pi])            ),
    transform( np.array([.1, .6, 0.5]),  np.array([0,pi,pi])            ),
    transform( np.array([.2, .5, 0.5]),  np.array([0,pi,pi-pi/2])       ),
    transform( np.array([.2, -.5, 0.5]), np.array([0,pi-pi/2,pi])       ),
    transform( np.array([.2, -.5, 0.5]), np.array([pi/4,pi-pi/2,pi])    ),
    transform( np.array([.4, 0, 0.2]),   np.array([0,pi-pi/2,pi])       ),
    transform( np.array([.4, 0.1, 0.2]),   np.array([pi/2,pi-pi/2,pi])    ),
    transform( np.array([.4, 0.2, 0.2]),     np.array([pi/2,pi-pi/2,pi])    ),
]
```

We collected data for time elapsed, number of iterations and the rate of success for the above 10 new configurations and calculated the mean median and the maximum values for time elapsed and the number of iterations performed. Moreover, we also calculated the rate of success for these 10 configurations. The results are as follows:

Mean time elapsed: 0.299 seconds
Median Time elapsed: 0.255 seconds
Maximum time elapsed: 0.412 seconds

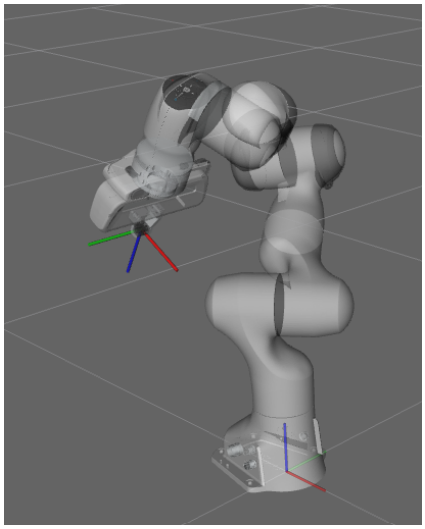
Mean iterations: 37.28
Median iterations: 43
Maximum iteration: 49

Rate of Success: 70 percent

6.3 Analysis

Uniqueness of solution:

To test the effect of the different seeds on the solution, we tested the same target position with two different seeds and found the solution.



The end effector coordinates and angles in consideration are [-0.1, -0.2, 0.5] and [0,5/6*pi,pi] respectively. When we start the solver with the neutral position as the seed, our solver returns success and we are able to reach the target position successfully. Our final joint configuration values are as follows.

$$q[-0.90196353 -1.20063106 -0.23601956 -2.50140986 -0.736391431.09303507 -0.44824068]$$

When we start the solver with a custom seed of [0,0,0,-pi/4,0,pi/2,pi/4], we fail to find the solution and our robot is not able to reach the target position. The final joint configuration returned by our solver is as follows:

$$q[-2.26726992 -4.495965492.73423083 -27.366524250.3086548612.71289603 -4.17330216]$$

Hence, we can say that the success of a solution does depend on the initial configuration or seed of the robot. This is a major drawback of numerical IK methods. The gradient descent would converge to the nearest local minimum that it finds. Hence different initial seeds would result in different solutions.

Algorithmic Completeness:

Yes, sometimes the target end effector poses would result in failure of our algorithm. This does not mean that we have coded our algorithm wrong. The reason for the failure of our optimizer could have multiple interpretations.

One reason might be an incorrect or bad guess of the initial seed that we picked which made the algorithm get stuck in a local minima which, although is a minima, is not the solution to our problem.

Another reason may be that the gradient descent needed more iterations to converge to our solution but as we have limited the number of maximum iterations, the optimizer was not able to converge and hence it did not reach the appropriate solution.

Warm start:

For a warm start, starting from a random seed, we can first solve the optimizer for one given target point and we can record the solution we get. Now, on the next target which is nearby, we run our optimiser with the previous solution as the current seed. This would help in faster convergence of numerical Ik_solver and aid in real time operations. The idea is that If our initial seed is better, we will converge faster to our solution.