Jacobian and Inverse Kinematics

Question 1

How many inverse kinematic solutions exist for a 2D 4R-planar manipulator, if an achievable pose of the end-effector x_e is given? Give a full explanation to support your answer. [report - 5 pts]

When solving inverse kinematic problems, the solution involves trigonomretric equations (which give more than one solution), and polynomial equations (which introduce more solutions. See below.

$$cos(\theta) = \frac{1}{\sqrt(2)} \rightarrow \theta = \pm\frac{\pi}{4}$$

$$x = 1 \rightarrow x^2 = 1 \rightarrow = \pm 1$$

To determine the inverse kinematic solution with known pose x_e, hence a known ${}^0\mathbf{T}_e$.

$$\mathbf{x_e} = \begin{bmatrix}\mathbf{p_e}\\ \phi_\mathbf{e}\end{bmatrix} = \begin{bmatrix}p_{ex}\\ p_{ey}\\ p_{ez}\\ r_{ex}\\ r_{ey}\\ r_{ez}\end{bmatrix} \rightarrow {}^\mathbf{0}\mathbf{T_e} = \begin{bmatrix} r_1, & r_2, & r_3, & p_{e_x}\\ r_4, & r_5, & r_6, & p_{e_y}\\ r_7, & r_8, & r_9, & p_{e_z}\\ 0, & 0, & 0, & 1\end{bmatrix} =$$

$$= \begin{bmatrix} cos(\theta_{123)} & -sin(\theta_{123}) & 0 & l_1cos(\theta_1) + l_2cos(\theta_{12} + l_3cos(\theta_{123}\\ sin(\theta_{123}) & cos(\theta_{123}) & 0 & l_1sin(\theta_1) + l_2sin(\theta_{12} + l_3sin(\theta_{123}\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1) \end{bmatrix}$$

Thus solve:

$$cos(\theta_{123}) = r_1$$

$$sin(\theta_{123}) = r_3$$

$$l_1cos(\theta_1) + l_2cos(\theta_{123}) = p_{e_x}$$

$$l_1sin(theta_1) + l_2sin(\theta_{12}) + l_3sin(\theta_{123}) = p_{e_y}$$

Solve for θ_2:

$$\theta_2 = \pm arccos\left(\frac{(p_{e_x} - l_3r_1)^2 + (p_{e_y} - l_3r_3)^2 - l_1^2 - l_2^2}{2l_1l_2}\right)$$

Where the argument< 1
Solve for θ_1:

$$\theta_1 = atan2(p_{e_y} - l_3r_3, p_{e_x} - l_3r_q) - atan2(l_2sin(\theta_2, l_1 + l_2cos(\theta_2))$$

Solve for θ_3:

$$\theta_3 = atan2(r_3, r_1) - \theta_1 - \theta_2$$

This gives 6 solutions as the pose of the end effector is fixed, thus this can be treated as a 3R problem with fixed *position*, aswell as a $4R$ manipulator with fixed *pose*, therefore the 3 joints give 6 solutions.

For a 2D 4R- planar manipulator, there are 6 inverse kinematic solutions, given end pose x_e, as the pose of the end effector is fixed, and the first joint is fixed, therefore 3 joints can be in different orientations. Therefore there are 6 inverse kinematic solutions. This is assuming that x_e is not at the end of the workspace, as if x_e is at the edge of the workspace, there would only be one solution, all joints fully extended.

Question 2

Suppose that the robot is moving in a free space (i.e. there are no obstacles) and that more than one inverse kinematic solution exist for a desired pose of the end-effector xe, what criteria should you consider when choosing an optimal solution? [report - 5pts]

Criterion considered when choosing an optimal solution include the solvability and the robot workspace. If a desired position is outside the workspace then the solution cannot be considered, as there does not exist a set of real number solutions that can satisfy the solvability. Furthermore, the manufacturer joint limits must be considered, so to not direct the robot outside of its joint capabilities.

Question 3

When is the output of the function $atan2(y, x)$ different from $atan(\frac{x}{y})$? [report - 5pts]

The outputs of the functions $atan2(y, x)$ and $atan(\frac{x}{y})$ differ when $y < 0$. $atan2(y, x)$ is the four quadrants representation of $atan$ and can also handle when $x = 0$, notifying us which quadrant the co-ordinate (x, y) is in.

Question 4

Complete the following tasks by filling in the "cw2q4/youbotKineStudent.py" python code template. A simple code breakdown in the report is required for all sub-questions, except sub-question b which is report only. In the cw2q4 folder you can find three files.

- *youbotKineStudent.py: This is the coding template for the questions below.*
- *youbotKineBase.py: This class includes common methods you may need to call in order to solve the questions below. You should not edit this file.*

- *youbotKineKDL.py: This class provides implementations to the questions below in KDL in order to check your own solutions. You should not edit this file.*

a

Write a script to compute the Jacobian matrix for the YouBot manipulator. [report - 2 pts, code - 10 pts]

The Jacobian matrix $\mathbf{J}$ can be calculated as follows, as the joints are revolute:

$$\mathbf{J}_{p_i} = \mathbf{z}^0_{i-1} \times (\mathbf{p}^0_e - \mathbf{o}^0_{i-1})$$

$$\mathbf{J}_{O_i} = \mathbf{z}^0_{i-1}$$

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_{P_1} & \dots & \mathbf{J}_{P_i} & \dots & \mathbf{J}_{P_n} \\ \mathbf{J}_{O_1} & \dots & \mathbf{J}_{O_i} & \dots & \mathbf{J}_{O_n} \end{bmatrix}$$

Where:

$$\omega = \mathbf{z}^0_i - 1$$

$$\nu = \omega \times \mathbf{r}$$

$$\mathbf{r} = \mathbf{p^0_e} - \mathbf{o^0_{i-1}}$$

$\mathbf{z}^0_{i-1}$ is the third column of the rotation matrix ${}^0\mathbf{R}_{i-1}$.
$\mathbf{p}^0_e$ is gievn by the first three elements of the fourth column of ${}^0\mathbf{T}$.
$\mathbf{o}^0_{i-1}$ is given by the first three elements of the fourth column of ${}^0\mathbf{T}_{i-1}$. A python function to calculate this is as follows.

```
def get_jacobian(self, joint):
        """Given the joint values of the robot, compute the Jacobian matrix. Coursework 2 Question 4a.
        Reference - Lecture 5 slide 24.

        Args:
            joint (list): the state of the robot joints. In a youbot those are revolute

        Returns:
            Jacobian (numpy.ndarray): NumPy matrix of size 6x5 which is the Jacobian matrix.
        """
        assert isinstance(joint, list)
        assert len(joint) == 5

        # Your code starts here ----------------------------

        # For your solution to match the KDL Jacobian,
        # z0 needs to be set [0, 0, -1] instead of [0, 0, 1],
```

```
        # since that is how its defined in the URDF.
        # Both are correct.
        #SEE LAB 6 EXAMPLE
    ################################################################
        jacobian = np.empty(6,5)
        T = []
        #Forward Kinematic to calculate transformation matrices
        for i in range(5):
            T.append(self.forward_kinematics(joint, i))
            Tr = T[i]
            z_prev = Tr[0:3,2] # Third column of the rotation matrix R\
            o_prev = Tr[0:3,3] # First three elements of the fourth column
     of 0Te

            jacobian[0:3, i ] = np.cross(z_prev, (p - o_prev)) #J_P_i
            jacobian[3:6, i] = z_prev
        #jacobian = [-l1*sin(theta1)- l2sin(theta12)- l3sin(theta123), ]
        # Your code ends here ------------------------------

        assert jacobian.shape == (6, 5)
        return jacobian
```

Listing 1: 4a Code

b

Derive the closed-form inverse kinematics solutions for the YouBot manipulator. You can represent any non-zero length parameters as variables.[report 8 pts]

To derive the closed form inverse kinematic solution for the YouBot manipulator, we first need to calculate the ${}^0T_e$ using the DH parameters given, where e is the final joint.

$$
{}^0\mathbf{T}_e(\theta_1, \theta_2, \theta_3, ..., \theta_5) = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p^0_{e_x} \\ r_{21} & r_{22} & r_{23} & p^0_{e_y} \\ r_{31} & r_{32} & r_{33} & p^0_{e_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

Where:

(in the interest of space, the entire equations below are not shown)

$$r_{11} = sin(\theta_1)*sin(\theta_5)+cos(\theta_1)*cos(\theta_2)*cos(\theta_3)*cos(\theta_4)*cos(\theta_5)-cos(\theta_1)*cos(\theta_2)*cos(\theta_5)*sin(\theta_3)*sin(\theta_4)-c$$

$$r_{12} = cos(\theta_1)*cos(\theta_2)*cos(\theta_3)*cos(\theta_4)*sin(\theta_5)-cos(\theta_5)*sin(\theta_1)-cos(\theta_1)*cos(\theta_2)*sin(\theta_3)*sin(\theta_4)*sin(\theta_5)-c$$

$$r_{13} = \frac{sin(\theta_2 - \theta_1 + \theta_3 + \theta_4) + sin(\theta_1 + \theta_2 + \theta_3 + \theta_4)}{2}$$

$r_{21} = cos(\theta_2){*}cos(\theta_3){*}cos(\theta_4){*}cos(\theta_5){*}sin(\theta_1){-}cos(\theta_1){*}sin(\theta_5){-}cos(\theta_2){*}cos(\theta_5){*}sin(\theta_1){*}sin(\theta_3){*}sin(_4){-}co$

$r_{22} = cos(\theta_1){*}cos(\theta_5){+}cos(\theta_2){*}cos(\theta_3){*}cos(\theta_4){*}sin(\theta_1){*}sin(\theta_5){-}cos(\theta_2){*}sin(\theta_1){*}sin(\theta_3){*}sin(\theta_4){*}sin(\theta_5){-}c$

$$r_{23} = \frac{cos(\theta_2 - \theta_1 + \theta_3 + \theta_4) - cos(\theta_1 + \theta_2 + \theta_3 + \theta_4)}{2}$$

$$r_{31} = \frac{-sin(\theta_2 + \theta_3 + \theta_4 - \theta_5) - sin(\theta_2 + \theta_3 + \theta_4 + \theta_5)}{2}$$

$$r_{32} = \frac{cos(\theta_2 + \theta_3 + \theta_4 + \theta_5) - cos(\theta_2 + \theta_3 + \theta_4 - \theta_5)}{2}$$

$$r_{33} = cos(\theta_2 + \theta_3 + \theta_4)$$

$p_{e_x} = a2{*}cos(\theta_1){*}sin(\theta_2){-}a1{*}cos(\theta_1){+}a5{*}sin(\theta_1){*}sin(\theta_5){+}a3{*}cos(\theta_1){*}cos(\theta_2){*}sin(\theta_3){+}a3{*}cos(\theta_1){*}cos(\theta_3)$

$p_{e_y} = a2{*}sin(\theta_1){*}sin(th2){-}a5{*}cos(th1){*}sin(th5){-}a1{*}sin(th1){+}a3{*}cos(th2){*}sin(th1){*}sin(th3){+}a3{*}cos(\theta$

$p_{e_z} = d1{-}(a5{*}sin(\theta_2{+}\theta_3{+}\theta_4{+}\theta_5))/2{+}a3{*}cos(\theta_2{+}\theta_3){+}a2{*}cos(\theta_2){-}(a5{*}sin(\theta_2{+}\theta_3{+}\theta_4{-}\theta_5))/2{-}d5{*}cos(\theta_2{+}\theta$

Computed via MATLAB symbolic function. From $p^0_{e_x}$, $p^0_{e_y}$, we can rearrange the equations such that

$$\frac{p^0_{e_x} - a5 * sin(\theta_1)sin(\theta_5)}{cos(\theta_1)} = a2{*}sin(\theta_2){-}a1{+}a3{*}cos(\theta_2)sin(\theta_3){+}a3{*}cos(\theta_3){*}sin(\theta_2){-}a4{*}cos(\theta_2){*}cos(\theta_3){*}co$$

and

$$\frac{p^0_{e_y} + a5cos(\theta_1)sin(\theta_5)}{sin(\theta_1)} = a2{*}sin(\theta_2){-}a1{+}a3{*}cos(\theta_2){*}sin(\theta_3){+}a3{*}cos(\theta_3){*}sin(\theta_2){-}a4{*}cos(\theta_2){*}cos(\theta_3){*}cos$$

thus,

$$\frac{p^0_{e_x} - a5sin(\theta_1)sin(\theta_5)}{cos(\theta_1)} - \frac{p^0_{e_y} + a5cos(\theta_1)sin(\theta_5)}{sin(\theta_1)} = 0$$

$$p^0_{e_x}sin(\theta_1) - a5 * sin^2(\theta_1)sin(\theta_5) - p^0_{e_y}cos(\theta_1) - a5cos^2(\theta_1)sin(\theta_5)$$

$$a5 = \frac{-p^0_{e_y}cos(\theta_1) + p^0_{e_x}(sin(\theta_1)}{sin(\theta_5)}$$

Finally, now we have $\theta_{1,2,3}$:

$$^0\mathbf{R}_3(\theta_1, \theta_2, \theta_3)^3\mathbf{R}_5(\theta_4, \theta_5) =^0 \mathbf{R}_5$$

$$^3\mathbf{R}_5(\theta_4, \theta_5) = (^0\mathbf{R}_3(\theta_1, \theta_2, \theta_3))^{-1}\mathbf{R}$$

DH Parameters for the YouBot, from the code, being:

i	a	α	d	θ
1	-0.033	$\frac{\pi}{2}$	0.145	π
2	0.155	0.0	0.0	$\frac{\pi}{2}$
3	0.135	0.0	0.0	0.0
4	0.002	$\frac{\pi}{2}$	0.0	$-\frac{\pi}{2}$
5	0.00	π	-0.185	π

DH Parameters substituted into the above equations, the algebraic inverse kinematic solution is found.

c

Write a script to detect singularity in any input pose [report - 1 pts, code - 4 pts]

The function below calculates the determinant of the Jacobian matrix, if

$$det(\mathbf{J} = 0) \rightarrow n = 6$$

$$det(\mathbf{J^T J}) = 0 \rightarrow n \neq 6$$

, then the solution is singular. If $\mathbf{J}$ is rank deficient, the determinant will be zero and thus the solution will be singular.

```
def check_singularity(self, joint):
        """Check for singularity condition given robot joints. Coursework 2 Question 4c.
        Reference Lecture 5 slide 30.

        Args:
            joint (list): the state of the robot joints. In a youbot those are revolute

        Returns:
            singularity (bool): True if in singularity and False if not in singularity.

        """
        assert isinstance(joint, list)
        assert len(joint) == 5

        # Your code starts here ----------------------------
        # At singular configurations det(J) = 0
        if (self.jacobian(joint).shape[0] ==6 and self.jacobian(joint).shape[1] ==6):
            if (np.linalg.det(self.jacobian(joint))==0):
                singularity = True
            else:
```

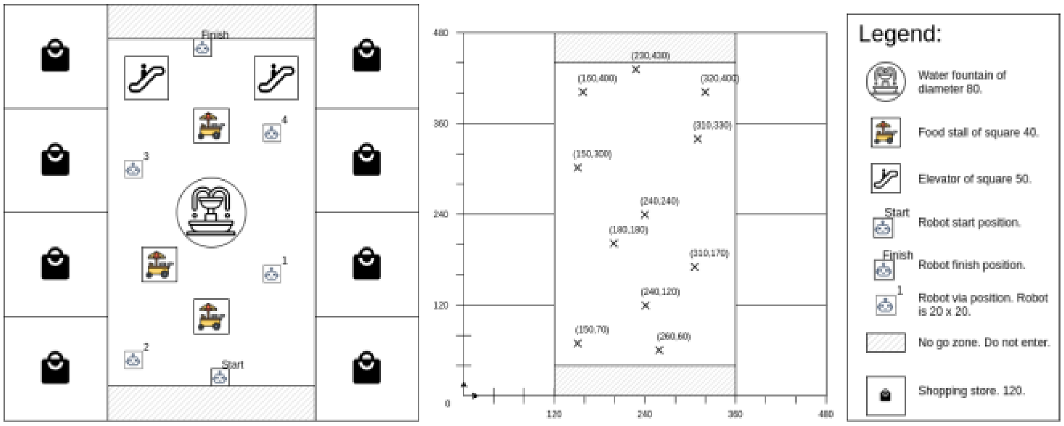
```
                singularity = False
        elif(self.jacobian(joint).shape[0] == 6 and self.jacobian(joint).
    shape[1] != 6):
            if (np.linalg.det((self.jacobian(joint)).T @ self.jacobian(
    joint) ) ==0):
                singularity = True
            else:
                singularity = False
        # Your code ends here ------------------------------
        assert isinstance(singularity, bool)
        return singularity
```

Listing 2: 4c Code

Path and Trajectory Planning

Question 5

Assume the following scenario: A shopping mall is testing autonomous cleaning robots to clean floors of a section of the mall. They have given you the following floor map defining no go zones, cleaning via points and obstacles. Cleaning will occur at night, so no dynamic obstacles will be present. Consider only passing through via points not total floor coverage. They also allow external cameras and tracking, so perfect odometry can be assumed. Present a robotic solution by choosing a drive system to complete the task, addressing path planning and trajectory planning. You can assume the position and orientation of your chosen robot is given as $q = (x, y, \theta)$ *where* x *and* y *describe the position and* θ *describes the orientation of the robot. You have a perfect controller that can control you wheel velocities to attain the given position and orientation. When choosing the drive system, consider that the vacuum is located at the back of the robot so during turns and rotations, the vacuum may miss water pickup. Your solution should address the following specifically. (Recommended answer is up to 50 words per sub-question and the word count for the entire question should not exceed 300 words.)*



(a) Map (b) Coordinates (c) Legend

Figure 1: Map, coordinates and legend of shopping mall cleaning area.

a

Present your drive system configuration and explain your reasoning. Is it holonomic or non-holonomic and why? What is the configuration space? [report - 5 pts]

The system used should be a Differential drive system, as each wheel speed can be controlled independently, although the system is non-holonomic as the non-holomonic constraints mean the robot cannot drive directly to it's destination if the destination is not in line with it's current orientation. The robot must either rotate before moving or rotate as it moves, as per the Figure below [1]. Configuration space is the entire arena that the robot can drive in, excluding obstacles. The drive system should be a DC stepper motor as this does not need an inverter, also the position of the motor can be known with ease, it is highly repeatable, has good speed control, and has full torque at low speeds. Although the stall torque for a stepper motor is very high, if it slips a tooth, then the position is not known.

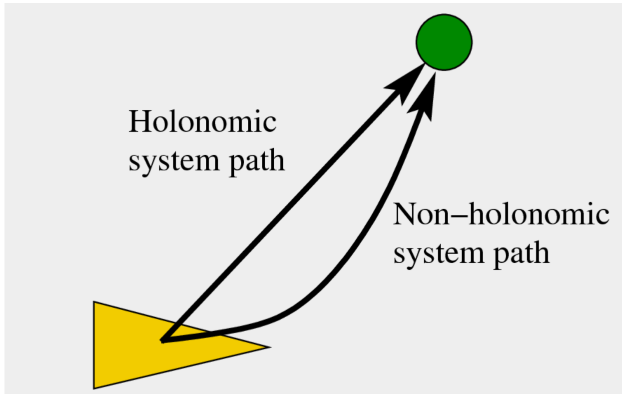


Figure 2: Holomonic vs Non-holomonic robot path [1].

b

Describe how you would find a path that goes from the starting point to the final point while passing through all the via points in order. How would planning change if the ordering did not matter? [report - 5 pts]

To pass through all the via points in order when traversing a path from the starting point to the final point, artificial potential fields can be used in the place of obstacles and checkpoints. Repulsive fields placed at obstacles and attractive fields placed at checkpoints. Thus the total potential field is calculated as follows:

$$U(\mathbf{q}) = U_{att}(\mathbf{q}) + U_{rep}(\mathbf{q})$$

Where: $U_{att}(\mathbf{q})$ is the field that attracts the robot to $\mathbf{q}_f$ and $U_{rep}(\mathbf{q})$ is the field that repels the robot. The global minimum of U can be found via gradient descent. To traverse the checkpoints in order, only the initial checkpoint attractive field is active, then when this is reached (within some tolerance), this field is deactivated and the subsequent field activated, and so on; this process will repeat until all checkpoints are completed. If the ordering did not matter, for time efficiency, the attractive field active would be the field closest to the robot at any given time. This is done in the Dijkstra search algorithm, in which the next node is found by calculating the least cost of travel between different configurations.

c

Describe a time scaling function you would use when constructing a trajectory. Consider initial and final accelerations as well as at the via points.[report - 5 pts]

The polynomial time scaling trajectory function has the form:

$$s(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$$

While $s(0) = \dot{s}(0) = \ddot{s}(T) = 0$
and terminal constraints $s(T) = 1$, and $\dot{s}(T) = 0$
Solving for $a_0, a_1, a_2, a_3, a_4, a_5$ with 6 terminal position, velocity and acceleration constraints, we find the constants $a_1, a_2, ..., a_5$
This yields a smoother motion with a higher maximum velocity than a cubic scaling function.

d

The robot is designed with a water vacuum at the back of the robot, meaning during turns and rotations, the vacuum may miss water pick-up. What ways can you design your path or trajectory to prevent left-over water? [report - 5 pts]

The trajectory can be designed so that the robot does not turn whenever it is at a checkpoint, where there is water, so that left over water is prevented. The path can be a decoupled rotation and translation, rather than a screw path, so the robot travels in straight lines. A screw path is when the axis of rotation and the line along which the translation of the body occurs is the same, and both rotation and translation are performed simultaneously. While decoupled translation and rotation have different axis of rotation and translation. Screw path being:

$$\mathbf{T}(t) = \mathbf{T}_s e^{log(\mathbf{T}_s^{-1}\mathbf{T}_f)t}$$

$$t \in [0, 1]$$

and Decoupled rotation and translation being:

$$\mathbf{p}(t) = \mathbf{p}_s + t(\mathbf{p}_f - \mathbf{p}_s)$$

$$\mathbf{R}(t) = \mathbf{R}_s e^{log(\mathbf{R}_s^{-1}\mathbf{R}_f)t}$$

$$t \in [0, 1]$$

In addition to this, to ensure water is not missed, the robot can be instructed to pass through a tolerance around the checkpoints.

e

Describe a path planning approach for full floor coverage while avoiding obstacles. [report - 5 pts]
A path planning approach for full coverage whilst avoiding obstacles using the probablistic roadmap algorithm is as follows:

- Divide the configuration space into sections, such that each section represents a checkpoint.

- Create a potential field in which there are repulsive fields in the place of obstacles, and attractive fields in the place of checkpoints.
- The potential field approach only returns a single path from q_s to q_f. If multiple paths are required, we have to apply potential field several times with different parameters tuning for each required path.
- Create a configuration space roadmap that is a network of curves in which each node represents a robot configuration.
- After constructing the roadmap visiting all possible sections (full floor coverage), the configuration space should be sampled.
- Pairs of nodes are then connected to their k^th nearest neighbour using 2-norm for example:
$$||\mathbf{q} - \mathbf{q}'||$$
- Enhancing allows for disjointed nodes to be joined up by adding more nodes or connecting small components to larger.
- Finally, the path is smoothed to prevent jerky motion.
- Dijkstra algorithm can be used to calculate the minimum cost moving between nodes.

Question 6

Complete the following question by filling in the 'cw2q6/cw2q6 node.py' python code templates to perform path planning via the shortest path. A simple code breakdown in the report is required for all subquestions, except subquestion e which is code only. You are given target joint positions in the bagfile 'data.bag'. Your task is the Youbot end-effector to reach each target Cartesian check-point associated with each target joint position, via shortest path in Cartesian space. To solve the question you need to:

a

Implement the "load targets()" method that loads the target joint positions from the bagfile and calculates the target end-effector position. [report - 2 pts, code - 3 pts]

This function loads in the targets from the bagfile (checkpoint data and target joint positions), computes cartesian checkpoints from these joint positions. This is achieved by looping through bag file to read the messages, and determining the position of the joints. The cartesian co-ordinates are then calculated using the forward_kinematics function, both target_cart_tf and target_joint_positions are outputted.

```python
def load_targets(self):
        """This function loads the checkpoint data from the 'data.bag' file. In the bag file, you will find messages
        relating to the target joint positions. You need to use forward kinematics to get the goal end-effector position.
        Returns:
            target_cart_tf (4x4x5 np.ndarray): The target 4x4 homogenous transformations of the checkpoints found in the
            bag file. There are a total of 5 transforms (4 checkpoints + 1 initial starting cartesian position).
            target_joint_positions (5x5 np.ndarray): The target joint values for the 4 checkpoints + 1 initial starting
            position.
        """
        # Defining ros package path
        rospack = rospkg.RosPack()
        path = rospack.get_path('cw2q6')

        # Initialize arrays for checkpoint transformations and joint positions
        target_joint_positions = np.zeros((5, 5))
        # Create a 4x4 transformation matrix, then stack 6 of these matrices together for each checkpoint
        target_cart_tf = np.repeat(np.identity(4), 5, axis=1).reshape((4, 4, 5))

        # Load path for selected question
        bag = rosbag.Bag(path + '/bags/data.bag')
        # Get the current starting position of the robot
        target_joint_positions[:, 0] = self.kdl_youbot.kdl_jnt_array_to_list(self.kdl_youbot.current_joint_position)
        # Initialize the first checkpoint as the current end effector position
        target_cart_tf[:, :, 0] = self.kdl_youbot.forward_kinematics(target_joint_positions[:, 0])

        # Your code starts here ------------------------------
        #if len(sys.argv) != 2:
        #    sys.stderr.write('[ERROR] This script only takes input bag file as argument.n')
        #else:
        #    inputFileName = sys.argv[1]
        #    print "[OK] Found bag: %s" % inputFileName

        topicList = []
        i = 1
        for topic, msgs, t in bag.read_messages(['joint_data']):
            target_joint_positions[:,i] = msgs.position
            target_cart_tf[:,:,i] = self.kdl_youbot.forward_kinematics(target_joint_positions[:,i], 5)
```

```
            i+=1
            my_pt = JointTrajectoryPoint()
            if topicList.count(topic) == 0:
                topicList.append(topic)
        #print '{0} topics found:'.format(len(topicList))


        #print(target_cart_tf)

        # Your code ends here ------------------------------

        # Close the bag
        bag.close()

        assert isinstance(target_cart_tf, np.ndarray)
        assert target_cart_tf.shape == (4, 4, 5)
        assert isinstance(target_joint_positions, np.ndarray)
        assert target_joint_positions.shape == (5, 5)

        return target_cart_tf, target_joint_positions
```

Listing 3: 6a Code load_targets function

b

Implement the "get shortest path()" method that takes the checkpoint transformations and computes the order of checkpoints that results in the shortest overall path. [report - 3 pts, code - 5 pts]

This function determines the order of checkpoints to have minimum Cartesian distance travelled by the end effector. This problem was modelled as a travelling salesman problem, in which the minimum distance travelled between points is calculated to give the shortest trajectory. First of all, the checkpoints are extracted from the checkpoints transformation matrix, the first 3 rows from the final column. The cost matrix is then calculated, giving a $n \times n$ matrix of euclidean distances between every point in the 3D Cartesian space, where n is the number of points. The shortest distance in the first column is then determined to determine the distance from the defined point 0 to the next closest point. The index from the next closest point is then utilised to determine the shortest distance in the column of said index. This is repeated until there are 5 points to give the shortest distance. The minimum distance travelled is also calculated in parallel. The outputs of this function are:

- index_shortest_dist (np.array): An array of size 5 indicating the order of checkpoint
- min_dist: (float): The associated distance to the sorted order giving the total estimate for travel distance.
- sorted_order: an array of the checkpoints sorted into order of shortest path.

```python
def get_shortest_path(self, checkpoints_tf):
        """This function takes the checkpoint transformations and computes the order of checkpoints that results
        in the shortest overall path.
        Args:
            checkpoints_tf (np.ndarray): The target checkpoint 4x4 transformations.
        Returns:
            sorted_order (np.array): An array of size 5 indicating the order of checkpoint
            min_dist:  (float): The associated distance to the sorted order giving the total estimate for travel
            distance.
        """

        # Your code starts here ------------------------------
        #Calculate the distance between all points, then choose the shortest distances in the cost matrix
        #print(checkpoints_tf.shape)
        checkpoints = []
        perm = permutations(checkpoints_tf)
        for i in range(checkpoints_tf.shape[2]):

            #checkpoints[i] = checkpoints_tf[0:3, 3, i]
            #print(checkpoints_tf[0:3,3])
            checkpoints.append(checkpoints_tf[0:3, 3, i])
        # get checkpoint coordinates from checkpoint transformation matrix, rows 1-3 of last column

        # Calculate cost matrix, distance between all n points, giving n x n matrix
        checkpoints= np.array(checkpoints)
        cost_matrix = np.zeros((checkpoints.shape[0], checkpoints.shape[0]))
        for i in range(checkpoints.shape[0]):
            for j in range(checkpoints.shape[0]):
                cost_matrix[i,j] = np.sqrt((checkpoints[i][0] - checkpoints[j][0])**2 + (checkpoints[i][1] - checkpoints[j][1])**2 + (checkpoints[i][2] - checkpoints[j][2])**2)
                #Make diagonals infinite so that distance between one point and itself isnt chosen
                cost_matrix[i,i] = np.inf
                # distance between each cartesian point
        # Find shortest path using Greedy algorithm
        index_shortest_dist = []
        shortest_dist = cost_matrix[:,i].min() # get minimum in each column ( shortest distance from first point) and next etc
        index = np.argmin(cost_matrix[:,1])
        index_shortest_dist.append(index)
        i = 0
```

```
        min_dist = 0
        while (i<6):
        #for i in range(1,5):
            shortest_dist = cost_matrix[:,index].min() # get minimum in each column ( shortest distance from first point) and next etc
            index = np.argmin(cost_matrix[:,index])
            index_shortest_dist.append(index) # add the index of the shortest distance
            cost_matrix[index,:] = np.inf #remove previous row from next loop by making distance infinite
            min_dist += shortest_dist # Add each shortest dist to get total min dist
            i+=1

        #Sort checkpoints into order dictated by index_shortest_dist
        sorted_order = []
        for i in range(5):
            sorted_order.append(checkpoints[index_shortest_dist[i]])
        # this will Append  and sort checkpoints in order of shortest path


        # Your code ends here ------------------------------

        #assert isinstance(sorted_order, np.ndarray)
        #assert sorted_order.shape == (5,)
        assert isinstance(min_dist, float)
        #return sorted_order
        return sorted_order, min_dist, index_shortest_dist
```

Listing 4: 6b Code get_shortest_path function

c

Implement the "decoupled rot and trans()" and "intermediate tfs()" methods that take the target checkpoint transforms and the desired order based on the shortest path sorting, and create intermediate transformations by decoupling rotation and translation. [report - 2 pts, code - 5 pts]

The Intermediate_tfs function computes the transforms of the intermediate points given the checkpoint order, target checkpoint transforms and number of intermediate points between checkpoints. This function calls decoupled_rot_and_trans which computes the transform between to checkpoints following a straight line path. The rotation matrices and translation matrices are extracted form the transformations 'checkpoint_a_tf' and 'checkpoint_b_tf', as the rotation matrix is the first 3 columns and rows, the translation matrix is the final column, first three rows.
Using the equations as follows:

$$\mathbf{p}(t) = \mathbf{p}_s + t(\mathbf{p}_f - \mathbf{p}_s)$$

$$\mathbf{R}(t) = \mathbf{R}_s e^{log(\mathbf{R}_s^{-1}\mathbf{R}_f)t}$$

Where $\mathbf{p}_s$ is the starting translation matrix and $\mathbf{p}_f$ is the final translation. $\mathbf{R}_s$ is the starting rotation matrix, and $\mathbf{R}_f$ is the final rotation matrix. Each corresponding to checkpoint_a_tf and checkpoint_b_tf.
The matrices are then combined back into a single transformation tf matrix.

```
def decoupled_rot_and_trans(self, checkpoint_a_tf, checkpoint_b_tf, num_points):
        """This function takes two checkpoint transforms and computes the intermediate transformations
        that follow a straight line path by decoupling rotation and translation.
        Args:
            checkpoint_a_tf (np.ndarray): 4x4 transformation describing pose of checkpoint a.
            checkpoint_b_tf (np.ndarray): 4x4 transformation describing pose of checkpoint b.
            num_points (int): Number of intermediate points between checkpoint a and checkpoint b.
        Returns:
            tfs: 4x4x(num_points) homogeneous transformations matrices describing the full desired
            poses of the end-effector position from checkpoint a to checkpoint b following a linear path.
        """

        # Your code starts here ------------------------------
        # tfs = combined rot and trans
        t = 1.0 / (num_points + 1)
        a_rot = np.empty([2,2])
        b_rot = np.empty([2,2])
        a_trans = []
        b_trans = []
        #print('hello')
        #print(checkpoint_a_tf)
        for i in range(2):
            for j in range(2):
                a_rot[i,j] = checkpoint_a_tf[i,j]#[0:2,0:2]
                b_rot[i,j] = checkpoint_b_tf[i,j]#[0:2,0:2]

        for i in range(3):
            b_trans.append(checkpoint_b_tf[i, -1])
            a_trans.append(checkpoint_a_tf[i, -1])
        c = [b_trans - a_trans for b_trans, a_trans in zip(b_trans, a_trans)]
        c = np.array(c)
        c = c.reshape((3,1))
        trans = a_trans + t * c#(b_trans - a_trans)
        #trans = a_trans + t * (b_trans - a_trans)

```

```
        rot = np.matmul(a_rot,expm((logm(np.matmul(np.linalg.inv(a_rot) ,
    b_rot))) * t))
        tfs = np.empty([4,4])
        for i in range(2):
            for j in range(2):
                tfs[i,j] = rot[i,j]
        #tfs[0:3,0:3] = rot
        for i in range(3):
            tfs[i, 3] = trans[i] # for loop?? # should be 3x1
        tfs[3,:] = [0,0,0,1]
        tfs = np.array(tfs)
        #Combine back into one matrix
        # Your code ends here ------------------------------

        return tfs
```

Listing 5: 6c Code decoupled_rot_and_trans function

The intermediate transformations between target checkpoints are calculated by:

- Using sorted checkpoint indices and target checkpoints to get a sorted list of checkpoints, in the shortest path possible, thanks to the shortest_path function.
- The full checkpoints matrix is initialised with the first target checkpoint.
- The decoupled rotation and translation matrix is called, with the input being the i^th and $i + 1^th$ target checkpoint to give the intermediate checkpoints to ensure a straight path between points.
- The target checkpoints are also appended between the intermediate points to give a full checkpoints transformation matrix 'full_checkpoint_tfs'.

```
def intermediate_tfs(self, sorted_checkpoint_idx, target_checkpoint_tfs,
    num_points):
        """This function takes the target checkpoint transforms and the
    desired order based on the shortest path sorting,
        and calls the decoupled_rot_and_trans() function.
        Args:
            sorted_checkpoint_idx (list): List describing order of
    checkpoints to follow.
            target_checkpoint_tfs (np.ndarray): the state of the robot
    joints. In a youbot those are revolute
            num_points (int): Number of intermediate points between
    checkpoints.
        Returns:
            full_checkpoint_tfs: 4x4x(4xnum_points+5) homogeneous
    transformations matrices describing the full desired
            poses of the end-effector position.
        """
```

```
        # Your code starts here ------------------------------
        #TODO
        full_checkpoint_tfs= np.repeat(np.identity(4), (4* num_points) +5,
     axis=1).reshape((4, 4, (4* num_points) +5))

        full_checkpoint_tfs[:,:,0] = target_checkpoint_tfs[:,:,0]

        #full_checkpoint_tfs= [target_checkpoint_tfs[0]]
        sorted_checkpoint_tfs = []
        #print(target_checkpoint_tfs)
        for i in range(5):
            sorted_checkpoint_tfs.append(target_checkpoint_tfs[:,:,
    sorted_checkpoint_idx[i]])
        #print(sorted_checkpoint_tfs[2].shape)
        for i in range(1,4):
            full_checkpoint_tfs[:,:,i] = (self.decoupled_rot_and_trans(
    sorted_checkpoint_tfs[i], sorted_checkpoint_tfs[i+1], num_points))
            full_checkpoint_tfs[:,:,i+1] = (target_checkpoint_tfs[:,:,i
    +1])# append target after initial point, between intermediate points.

        #print(len(full_checkpoint_tfs))

        # Your code ends here ------------------------------

        return full_checkpoint_tfs
```

Listing 6: 6c Code intermediate_tfs function

d

Implement the "ik position only()" and "full checkpoints to joints()" methods that take the full set of checkpoint transformations, including intermediate checkpoints, and compute the associated joint positions with position only inverse kinematics. [report - 2 pts, code - 8 pts]

The function to calculate the ik_position_only is shown below in listing 6, this function implements position only inverse kinematics to output the Inverse kinematic solution for a given pose, and the Cartesian error of the solution.

```
def ik_position_only(self, pose, q0, alpha = 0.1):
        """This function implements position only inverse kinematics.
        Args:
            pose (np.ndarray, 4x4): 4x4 transformations describing the
    pose of the end-effector position.
            q0 (np.ndarray, 5x1):A 5x1 array for the initial starting
    point of the algorithm.
        Returns:
            q (np.ndarray, 5x1): The IK solution for the given pose.
```

```
            error (float): The Cartesian error of the solution.
        """
        # Some useful notes:
        # We are only interested in position control - take only the position part of the pose as well as elements of the
        # Jacobian that will affect the position of the error.

        # Your code starts here ------------------------------
        Pd = pose[:3, 3].ravel()
        q = q0
        q = np.array(q)
        q0 = np.array(q0)
        J = self.kdl_youbot.get_jacobian(q0)[:3, :]
        J= np.array(J)
        # Take only first 3 rows as position only solution.
        P = np.array(self.kdl_youbot.forward_kinematics(q0))[:3, -1]
        e = Pd - P.ravel()
        e = np.array(e)

        q += alpha * np.matmul(J.T, e)
        error = np.linalg.norm(e)
        # Your code ends here ------------------------------

        return q, error
```

Listing 7: 6d Code

The function full_checkpoints_to_joints converts a set of checkpoint transformations into associated joint positions by iteratively calling the inverse kinematics function for each point. It iteratively calls the inverse kinematic function while the error is larger than a threshold 0.1. The error is reset to a large value so to allow for tuning of the subsequent q.

```
def full_checkpoints_to_joints(self, full_checkpoint_tfs, init_joint_position):
        """This function takes the full set of checkpoint transformations, including intermediate checkpoints,
        and computes the associated joint positions by calling the ik_position_only() function.
        Args:
            full_checkpoint_tfs (np.ndarray, 4x4xn): 4x4xn transformations describing all the desired poses of the end-effector
            to follow the desired path. (4x4x(4xnum_points+5))
            init_joint_position (np.ndarray):A 5x1 array for the initial joint position of the robot.
        Returns:
            q_checkpoints (np.ndarray, 5xn): For each pose, the solution of the position IK to get the joint position
            for that pose.
        """

        # Your code starts here ------------------------------
```

```
        q_checkpoints = []
        iter_count = 0
        q = init_joint_position
        for i in range(full_checkpoint_tfs.shape[2]):
            error = 10 # reset error to large for each point.
            while (error >= 0.1):
                [q, error] = self.ik_position_only(full_checkpoint_tfs[:,:,i], q, alpha = 0.1)
                iter_count += 1

                if (iter_count > 10000):
                    break
            q_checkpoints.append(q) # CHeck the indexing of full_checjpoint_tfs
        q_checkpoints = np.array(q_checkpoints)
            #Append position only inverse kinematic solution to the q_checkpoints, taking one sheet of 3d matrix full_checkpoint_tfs at once.
        # Your code ends here ------------------------------

        return q_checkpoints
```

Listing 8: 6d Code

e

Implement the "q6()" method, the main method of this question, where other methods are called in order to perform the path planning task. [code - 5 pts]

```
def q6(self):
        """ This is the main q6 function. Here, other methods are called to create the shortest path required for this
        question. Below, a general step-by-step is given as to how to solve the problem.
        Returns:
            traj (JointTrajectory): A list of JointTrajectory points giving the robot joint positions to achieve in a
            given time period.
        """
        # Steps to solving Q6.
        # 1. Load in targets from the bagfile (checkpoint data and target joint positions).
        # 2. Compute the shortest path achievable visiting each checkpoint Cartesian position.
        # 3. Determine intermediate checkpoints to achieve a linear path between each checkpoint and have a full list of
        #    checkpoints the robot must achieve. You can publish them to see if they look correct. Look at slides 39 in lecture 7
        # 4. Convert all the checkpoints into joint values using an inverse kinematics solver.
```

```python
        # 5. Create a JointTrajectory message.

        # Your code starts here ------------------------------
        # TODO
        #Create object (not necessary) youbot_traj_plan = YoubotTrajectoryPlanning()

        #Load targets from bagfile
        [target_cart_tf, target_joint_positions] = self.load_targets()

        #Sort targets to find shortest path
        [sorted_order, min_dist, index_shortest_dist] = self.get_shortest_path(target_cart_tf)

        #FInd intermediate points between checkpoints to ensure straight line path
        #num_points = 5, 5 intermediate points between checkpoints, for smooth straight movement
        full_checkpoint_tfs = self.intermediate_tfs(index_shortest_dist, target_cart_tf, 5)

        #This function gets a np.ndarray of transforms and publishes them in a color coded fashion to show how the
        #Cartesian path of the robot end-effector.
        self.publish_traj_tfs(full_checkpoint_tfs)

        #This function converts checkpoint transformations (including intermediates) into joint positions
        init_joint_position = np.array(target_joint_positions[:,0])
        q_checkpoints = self.full_checkpoints_to_joints(full_checkpoint_tfs, init_joint_position) #What is init_joint_position in this?

        traj = JointTrajectory()
        dt = 2
        t = 10
        for i in range(q_checkpoints.shape[1]):
            traj_point = JointTrajectoryPoint()
            traj_point.positions = q_checkpoints[:, i]
            t += dt
            traj_point.time_from_start.secs = t
            traj.points.append(traj_point)

        #This function converts joint positions to a kdl array
        #kdl_array = self.list_to_kdl_jnt_array(q_checkpoints) # is this traj??no

        # Your code ends here ------------------------------

        assert isinstance(traj, JointTrajectory)
```

```
        return traj
```

Listing 9: 6e Code

1 References

[1] Bower, Tim, "Steering the robot"
http://faculty.salina.k-state.edu/tim/robot_prog/MobileBot/Steering/steering.html
(accessed Nov. 28, 2021).