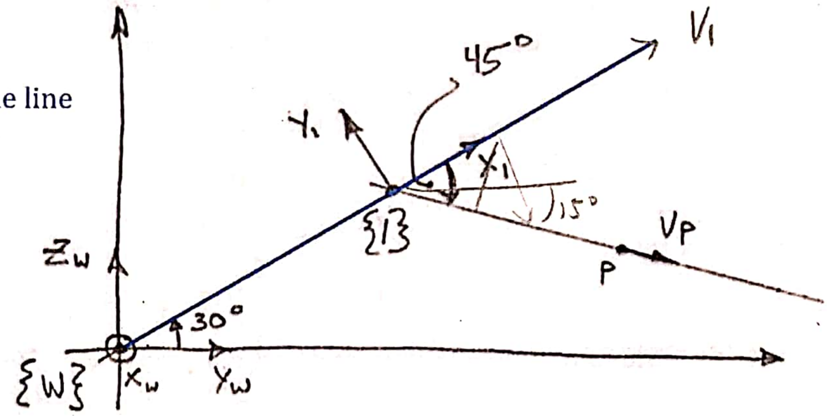
Problem Set 1

1.1 Velocities and Frames

Here are some facts:

- {1} is a moving frame along the blue line
- $^1(^WV_1) = [\cancel{0, 0.867, 0.5}]^T$ $[1,0,0]$
- $|V_p| = 2.$

$$^1(^WV_1) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$



1.1.1 What is $^W(^1V_P)$?

$$^W_1R = Rot(\hat{y}, 90^\circ)\, Rot(\hat{z}, 120^\circ) = \begin{bmatrix} 0 & 0 & 1 \\ \frac{\sqrt{3}}{2} & \frac{-1}{2} & 0 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 0 \end{bmatrix}$$

$$^W(^1V_p) = {}^W_1R(^1V_p)$$

$$= \begin{bmatrix} 0 & 0 & 1 \\ \frac{\sqrt{3}}{2} & \frac{-1}{2} & 0 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 0 \end{bmatrix} \begin{bmatrix} 2 \times \cos 45^\circ \\ -2 \times \sin 45^\circ \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{\sqrt{6}-\sqrt{2}}{2} \\ \frac{\sqrt{2}-\sqrt{6}}{2} \end{bmatrix}$$

1.1.2 What is $^W(^WV_P)$?

$$^W(^WV_p) = {}^W(^WV_1 + {}^1V_p) = {}^W(^WV_1) + {}^W(^1V_p) = {}^W_1R\,{}^1V_1 + {}^W_1R(^1V_p)$$

$$= \begin{bmatrix} 0 & 0 & 1 \\ \frac{\sqrt{3}}{2} & \frac{-1}{2} & 0 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{\sqrt{6}-\sqrt{2}}{2} \\ \frac{\sqrt{2}-\sqrt{6}}{2} \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{\sqrt{6}-\sqrt{2}+\sqrt{3}}{2} \\ \frac{1+\sqrt{2}-\sqrt{6}}{2} \end{bmatrix}$$

1.1.3 What is $^W(^1V_1)$?

$$^W(^1V_1) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

1.2 Numerical Velocities

Perform the following computations numerically, you may use any combination of hand calculations, Python, Scilab, Matlab or any other software you wish.

(1) The pose of the robot is: $\theta_1 = 90^\circ$, $\theta_2 = 25^\circ$, $\theta_3 = -68^\circ$

(2) The joint velocities are: $\dot{\theta}_1 = 10^\circ/\text{sec}$, $\dot{\theta}_2 = -15^\circ/\text{sec}$, $\dot{\theta}_3 = 3^\circ/\text{sec}$

(3) ${}^0\omega_0 = {}^0v_0 = [0,0,0]^T$

(4) This is the DH table for the robot:

N	α_{N-1}	a_{N-1}	d_N	Θ_N	
1	0 α_0	7 a_0	0 d_1	$\Theta_1 = 90^\circ$	θ_1
2	$-\pi/2$ α_1	0 a_1	1 d_2	$\Theta_2 = 25^\circ$	θ_2
3	π α_2	2 a_2	2 d_3	$\Theta_3 = -68^\circ$	θ_3

Note that when using numpy with our column vector convention (3x1 np.matrix), you have to write your own cross product function. numpy's cross product function, np.cross(a,b), assumes a,b are lists.

1.2.1 Find ${}^3\omega_3$.

$${}^3W_3 = {}^3_2R\,{}^2W_2 + \dot{\theta}_3$$

$${}^2W_2 = {}^2_1R\,{}^1W_1 + \dot{\theta}_2$$

$${}^1W_1 = {}^1_0R\,{}^0W_0 + \dot{\theta}_1$$

角度 弧度

$${}^1W_1 = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 10 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 10 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0.1745 \end{bmatrix}$$

角度 弧度

$${}^2W_2 = \begin{bmatrix} 0.9063 & 0 & -0.4226 \\ -0.4226 & 0 & -0.9063 \\ 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \\ 10 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -15 \end{bmatrix} = \begin{bmatrix} -4.226 \\ -9.063 \\ -15 \end{bmatrix} = \begin{bmatrix} -0.0738 \\ -0.1582 \\ -0.2618 \end{bmatrix}$$

$${}^3W_3 = \begin{bmatrix} 0.3746 & 0.9272 & 0 \\ 0.9272 & -0.3746 & 0 \\ 0 & 0 & -1 \end{bmatrix}\begin{bmatrix} -4.226 \\ -9.063 \\ -15 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 3 \end{bmatrix} = \begin{bmatrix} -9.9862 \\ -0.5232 \\ 18 \end{bmatrix} = \begin{bmatrix} -0.1743 \\ -0.0091 \\ 0.3142 \end{bmatrix}$$

1.2.2 Find ${}^3V_3$.

$${}^1V_1 = {}^1_0R({}^0V_0 + {}^0W_0 \times {}^0P_1) =$$

$${}^2V_2 = {}^2_1R({}^1V_1 + {}^1W_1 \times {}^1P_2)$$

$${}^3V_3 = {}^3_2R({}^2V_2 + {}^2W_2 \times {}^2P_3)$$

$${}^1V_1 = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \times \begin{bmatrix} 7 \\ 0 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

degree → radian

$${}^2V_2 = \begin{bmatrix} 0.9063 & 0 & -0.4226 \\ -0.4226 & 0 & -0.9063 \\ 0 & 1 & 0 \end{bmatrix}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0.1745 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} -0.1581 \\ 0.0737 \\ 0 \end{bmatrix} \quad \begin{bmatrix} -0.0738 \\ -0.1582 \\ -0.2618 \end{bmatrix}$$

$${}^3V_3 = \begin{bmatrix} 0.3746 & 0.9272 & 0 \\ 0.9272 & -0.3746 & 0 \\ 0 & 0 & -1 \end{bmatrix}\left(\begin{bmatrix} -0.1581 \\ 0.0737 \\ 0 \end{bmatrix} + \begin{bmatrix} -0.0738 \\ -0.1582 \\ -0.2618 \end{bmatrix} \times \begin{bmatrix} 2 \\ 0 \\ -2 \end{bmatrix}\right) = \begin{bmatrix} -0.4946 \\ 0.3704 \\ -0.3164 \end{bmatrix}$$

1.2.3 Find $\boxed{{}^0\omega_3}$.

$${}^0W_3 = {}^0_3R\,{}^3_3W = {}^0_1R\,{}^1_2R\,{}^2_3R\,{}^3_3W = \begin{bmatrix} 0.3142 \\ 0.1745 \end{bmatrix}$$

1.2.4 Find ${}^0V_3$.

$${}^0V_3 = {}^0_3R\,{}^3V_3 = {}^0_1R\,{}^1_2R\,{}^2_3R\,{}^3V_3 = \begin{bmatrix} -0.3164 \\ 0.3958 \\ 0.4945 \end{bmatrix}$$

$$ {}^0_1T = \begin{bmatrix} C_1 & -S_1 & 0 & L_1 \\ S_1 & C_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^1_2T = \begin{bmatrix} C_2 & -S_2 & 0 & 0 \\ 0 & 0 & 1 & L_2 \\ -S_2 & -C_2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^2_3T = \begin{bmatrix} C_3 & -S_3 & 0 & L_3 \\ -S_3 & -C_3 & 0 & 0 \\ 0 & 0 & -1 & -L_4 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

1.3 Symbolic Velocities

The following question must be done by hand in symbolic form. Apply the sum-of-angles and similar triangular identities where possible to simplify your results.

(1) Assume ${}^0V_0 = {}^0\omega_0 = 0$

(2) This is the DH table for the robot.

N	α_{N-1}	a_{N-1}	d_N	Θ_N
1	0	L1	0	Θ_1
2	$-\pi/2$	0	L2	Θ_2
3	π	L3	L4	Θ_3

1.3.1 Find ${}^3\omega_3$.

$$ {}^1W_1 = {}^1_0R\,{}^0W_0 + \dot\theta_1 = \begin{bmatrix} C_1 & S_1 & 0 \\ -S_1 & C_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \dot\theta_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \dot\theta_1 \end{bmatrix} $$

$$ {}^2W_2 = {}^2_1R\,{}^1W_1 + \dot\theta_2 = \begin{bmatrix} C_2 & 0 & -S_2 \\ -S_2 & 0 & -C_2 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \dot\theta_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \dot\theta_2 \end{bmatrix} = \begin{bmatrix} -S_2\dot\theta_1 \\ -C_2\dot\theta_1 \\ \dot\theta_2 \end{bmatrix} $$

$$ {}^3W_3 = {}^3_2R\,{}^2W_2 + \dot\theta_3 = \begin{bmatrix} C_3 & -S_3 & 0 \\ -S_3 & -C_3 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} -S_2\dot\theta_1 \\ -C_2\dot\theta_1 \\ \dot\theta_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \dot\theta_3 \end{bmatrix} = \begin{bmatrix} -C_3S_2\dot\theta_1 + S_3C_2\dot\theta_1 \\ S_3S_2\dot\theta_1 + C_3C_2\dot\theta_1 \\ \dot\theta_3 - \dot\theta_2 \end{bmatrix} = \begin{bmatrix} \dot\theta_1 \sin(\theta_3-\theta_2) \\ \dot\theta_1 \cos(\theta_3-\theta_2) \\ \dot\theta_3 - \dot\theta_2 \end{bmatrix} $$

1.3.2 Find ${}^3V_3$.

$$ {}^1V_1 = {}^1_0R\,{}^0V_1 = {}^1_0R({}^0V_0 + {}^0W_0 \times {}^0P_1) = \begin{bmatrix} C_1 & S_1 & 0 \\ -S_1 & C_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \times \begin{bmatrix} L_1 \\ 0 \\ 0 \end{bmatrix} \right) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} $$

$$ {}^2V_2 = {}^2_1R\,{}^1V_2 = {}^2_1R({}^1V_1 + {}^1W_1 \times {}^1P_2) = \begin{bmatrix} C_2 & 0 & -S_2 \\ -S_2 & 0 & -C_2 \\ 0 & 1 & 0 \end{bmatrix} \left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \dot\theta_1 \end{bmatrix} \times \begin{bmatrix} 0 \\ L_2 \\ 0 \end{bmatrix} \right) = \begin{bmatrix} -L_2C_2\dot\theta_1 \\ L_2S_2\dot\theta_1 \\ 0 \end{bmatrix} $$

$$ {}^3V_3 = {}^3_2R\,{}^2V_3 = {}^3_2R({}^2V_2 + {}^2W_2 \times {}^2P_3) = \begin{bmatrix} C_3 & -S_3 & 0 \\ -S_3 & -C_3 & 0 \\ 0 & 0 & -1 \end{bmatrix} \left(\begin{bmatrix} -L_2C_2\dot\theta_1 \\ L_2S_2\dot\theta_1 \\ 0 \end{bmatrix} + \begin{bmatrix} -S_2\dot\theta_1 \\ -C_2\dot\theta_1 \\ \dot\theta_2 \end{bmatrix} \times \begin{bmatrix} L_3 \\ 0 \\ -L_4 \end{bmatrix} \right) = \begin{bmatrix} C_3 & -S_3 & 0 \\ -S_3 & -C_3 & 0 \\ 0 & 0 & -1 \end{bmatrix} \left(\begin{bmatrix} -L_2C_2\dot\theta_1 \\ -L_2S_2\dot\theta_1 \\ 0 \end{bmatrix} + \begin{bmatrix} L_4C_2\dot\theta_1 \\ -L_4S_2\dot\theta_1 \\ -L_3C_2\dot\theta_1 \end{bmatrix} + \begin{bmatrix} 0 \\ L_3\dot\theta_2 \\ 0 \end{bmatrix} \right) $$

1.3.3 Derive the Jacobian matrix from the previous result

$$ \begin{bmatrix} {}^3V_3 \\ {}^3W_3 \end{bmatrix} = \begin{bmatrix} (L_4-L_2)\cos(\theta_2-\theta_3) & -L_3\sin\theta_3 & 0 \\ (L_4-L_2)\sin(\theta_2-\theta_3) & -L_3\cos\theta_3 & 0 \\ -L_3\cos\theta_2 & 0 & 0 \end{bmatrix} \begin{bmatrix} \dot\theta_1 \\ \dot\theta_2 \\ \dot\theta_3 \end{bmatrix} = \begin{bmatrix} C_3(L_4C_2\dot\theta_1 - L_2C_2\dot\theta_1) + S_3(L_2S_2\dot\theta_1 + L_4S_2\dot\theta_1 - L_3\dot\theta_2) \\ S_3(L_2C_2\dot\theta_1 - L_4C_2\dot\theta_1) + C_3(L_2S_2\dot\theta_1 + L_4S_2\dot\theta_1 - L_3\dot\theta_2) \\ -L_3C_2\dot\theta_1 \end{bmatrix} $$

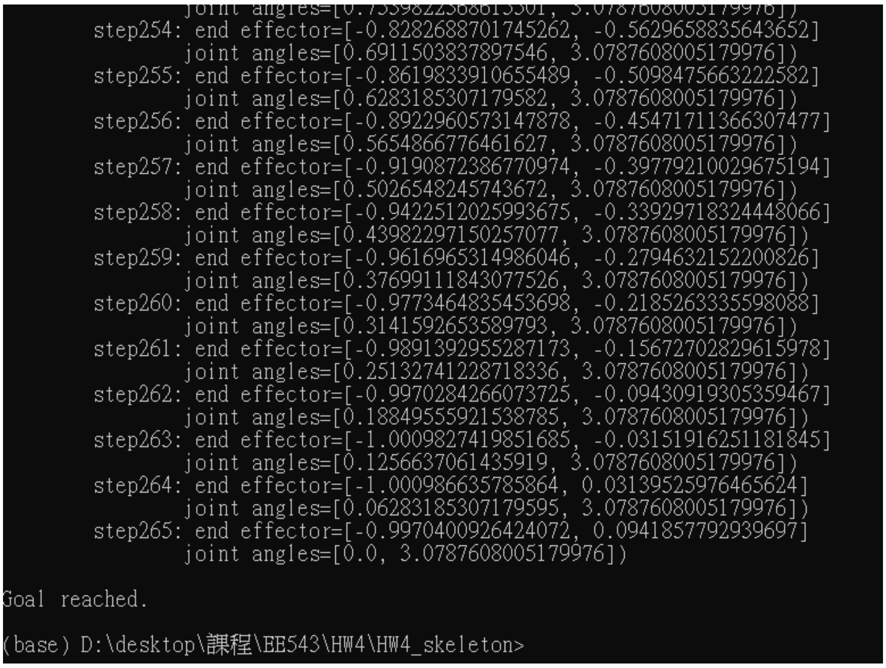
$$ \begin{bmatrix} (L_4-L_2)\cos(\theta_2-\theta_3) & -L_3\sin\theta_3 & 0 \\ (L_4-L_2)\sin(\theta_2-\theta_3) & -L_3\cos\theta_3 & 0 \\ -L_3\cos\theta_2 & 0 & 0 \end{bmatrix} = {}^3J $$

1.3.4 **Cross Checking: By substituting: L1 = 7, L2 = 1, L3 = 2, L4 = 2.**
Do answers in 1.2.1 and 1.2.2 agree with 1.3.1 and 1.3.2?

After I turn the unit of $\dot\theta_1, \dot\theta_2, \dot\theta_3$ from degree/sec to radian/sec, the answer agree with 1.3.1 and 1.3.2

EE543_HW4_c

a.



```
        step254: end effector=[-0.8282688701745262, -0.5629658835643652]
                joint angles=[0.6911503837897546, 3.0787608005179976])
        step255: end effector=[-0.8619833910655489, -0.5098475663222582]
                joint angles=[0.6283185307179582, 3.0787608005179976])
        step256: end effector=[-0.8922960573147878, -0.4547171136630747 7]
                joint angles=[0.5654866776461627, 3.0787608005179976])
        step257: end effector=[-0.9190872386770974, -0.39779210029675194]
                joint angles=[0.5026548245743672, 3.0787608005179976])
        step258: end effector=[-0.9422512025993675, -0.33929718324448066]
                joint angles=[0.43982297150257077, 3.0787608005179976])
        step259: end effector=[-0.9616965314986046, -0.2794632152200826]
                joint angles=[0.37699111843077526, 3.0787608005179976])
        step260: end effector=[-0.9773464835453698, -0.2185263335598088]
                joint angles=[0.3141592653589793, 3.0787608005179976])
        step261: end effector=[-0.9891392955287173, -0.15672702829615978]
                joint angles=[0.25132741228718336, 3.0787608005179976])
        step262: end effector=[-0.9970284266073725, -0.09430919305359467]
                joint angles=[0.18849555921538785, 3.0787608005179976])
        step263: end effector=[-1.0009827419851685, -0.03151916251181845]
                joint angles=[0.1256637061435919, 3.0787608005179976])
        step264: end effector=[-1.000986635785864, 0.03139525976465624]
                joint angles=[0.06283185307179595, 3.0787608005179976])
        step265: end effector=[-0.9970400926424072, 0.09418577929396 97]
                joint angles=[0.0, 3.0787608005179976])

Goal reached.

(base) D:\desktop\課程\EE543\HW4\HW4_skeleton>
```

Fig.1 Screenshot of terminal after running code successfully

b.

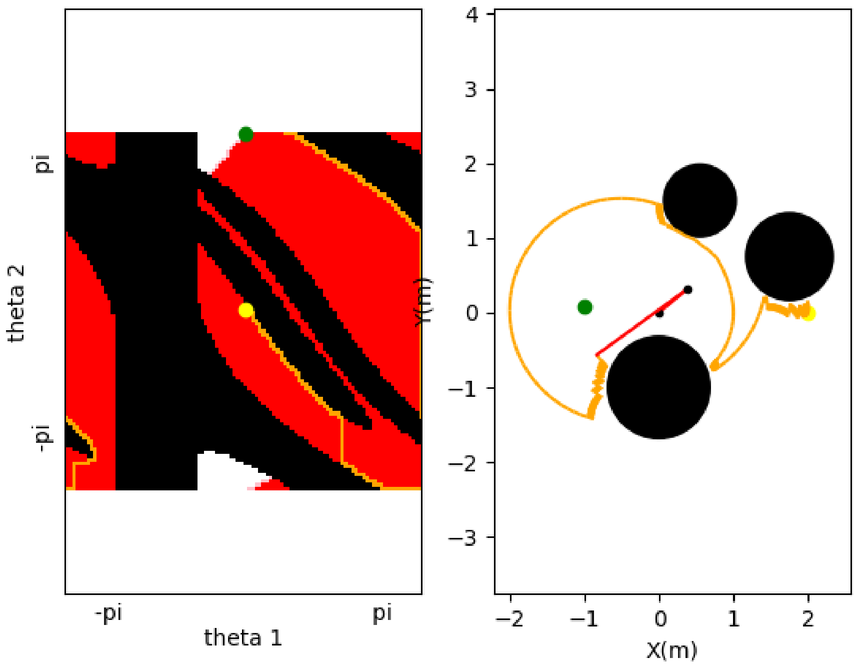


Fig.2 Screenshot of graphic window after running code successfully

c.

(i).

Because in cartesian space, I can guarantee every step I take (every decision I made) won't have any collision with any other obstacle, once I already build the cartesian space map with identifiers of collision or collision free space. On the contrast, in cartesian space, if I want to get a trajectory from start point to goal point, every step I made, I need to use inverse kinematic to check my every joint angles are ok (with no collision) in joint space and which is especially computationally expensive in a limited space.

(ii).

If we apply A-star search in joint space, then it will surely give us a global optimal solution under our criteria in the joint space. However, it is not global optimal in cartesian space because the mapping between joint space and cartesian space is nonlinear. Thus, we cannot get the global optimal solution in cartesian space from the A-star search in joint space.

(iii).

There is no speed constrain in the robot motion because in joint space, it can only move 1 degree in each direction. If the length of arms are constant, then because in $v = r\triangle\Theta/\triangle t$, we didn't set up any limitation on $\triangle t$. If $\triangle t$ is small then we will get high velocity with no limitation, thus it means we don't have any speed constrain.

(iv).

In the calc_heuristic_map, at first, it calculates the L1 distance between point and goal node (every point), after that, it starts from the initial point and look around four points and himself to get the minimal value (minimum cost) to replace the original value with that. After it went through all points in the graph, it returns the overall heuristic_map.

v.

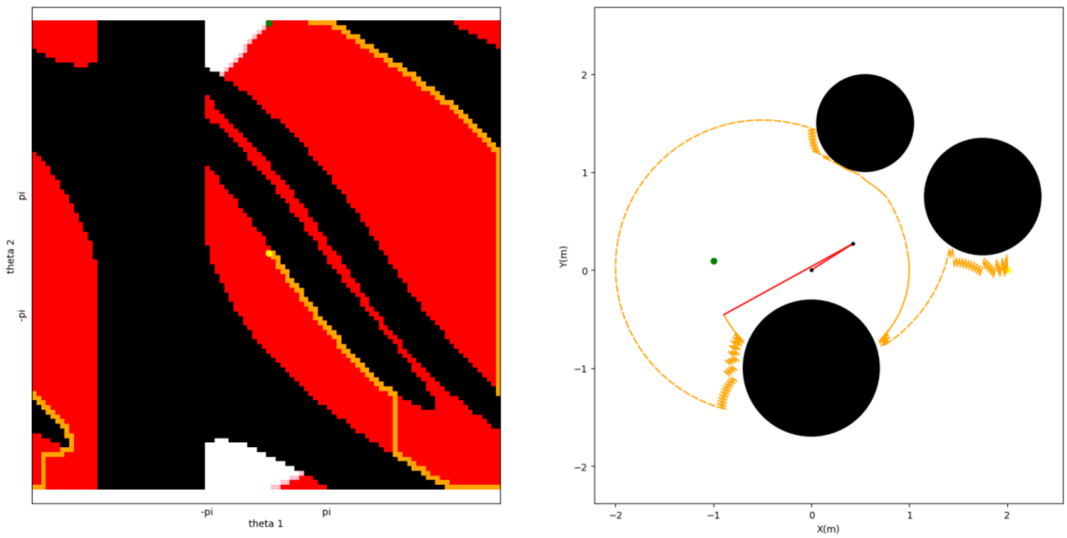


Fig.3 Goal point(50,99) with 0.1*L1 distance to build up the heuristic map

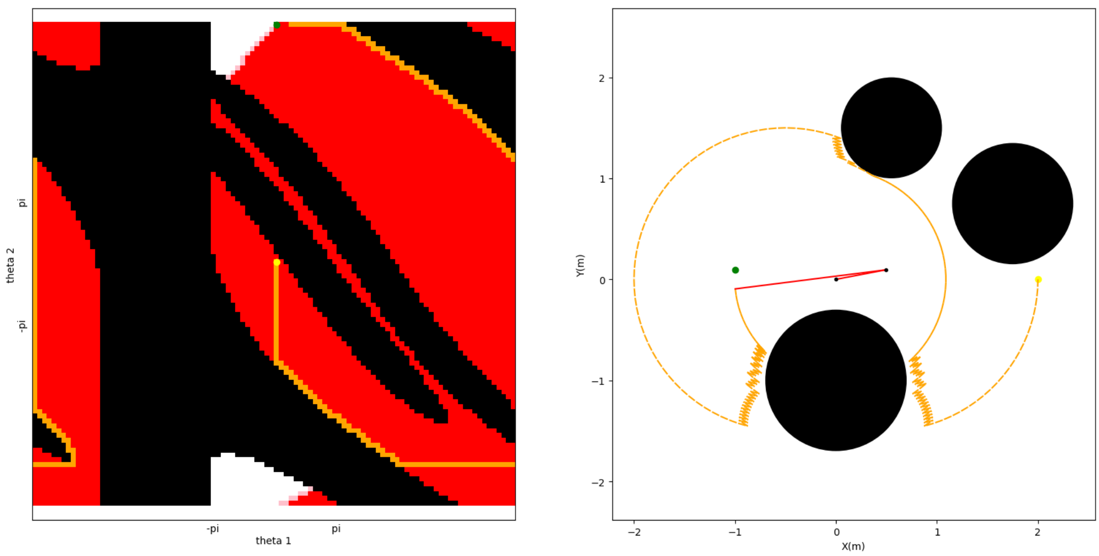


Fig.4 Goal point(50,99) with 0.1*L2 distance to build up the heuristic map

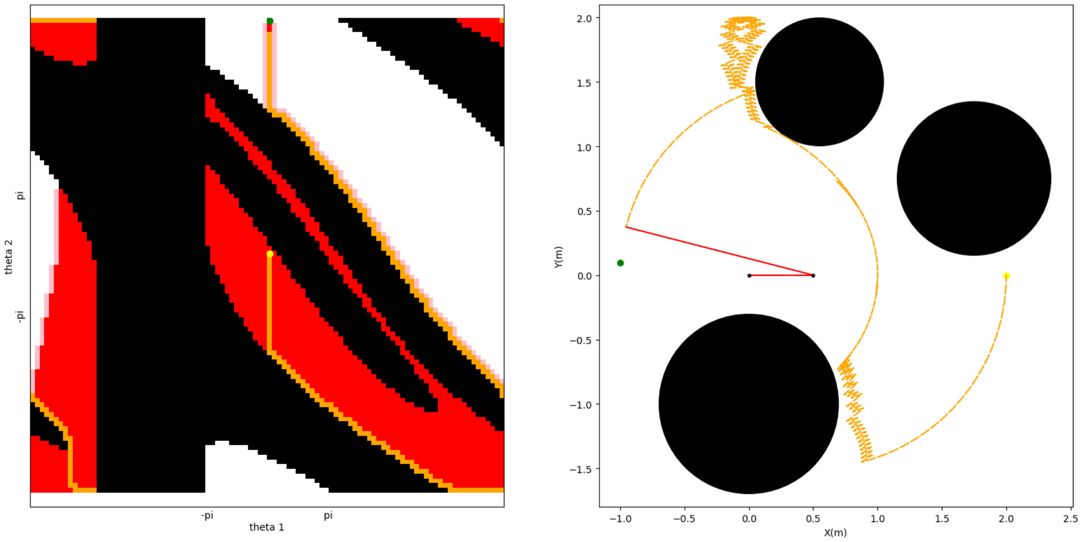


Fig.5 Goal point(50,99) with 10*L1 distance to build up the heuristic map

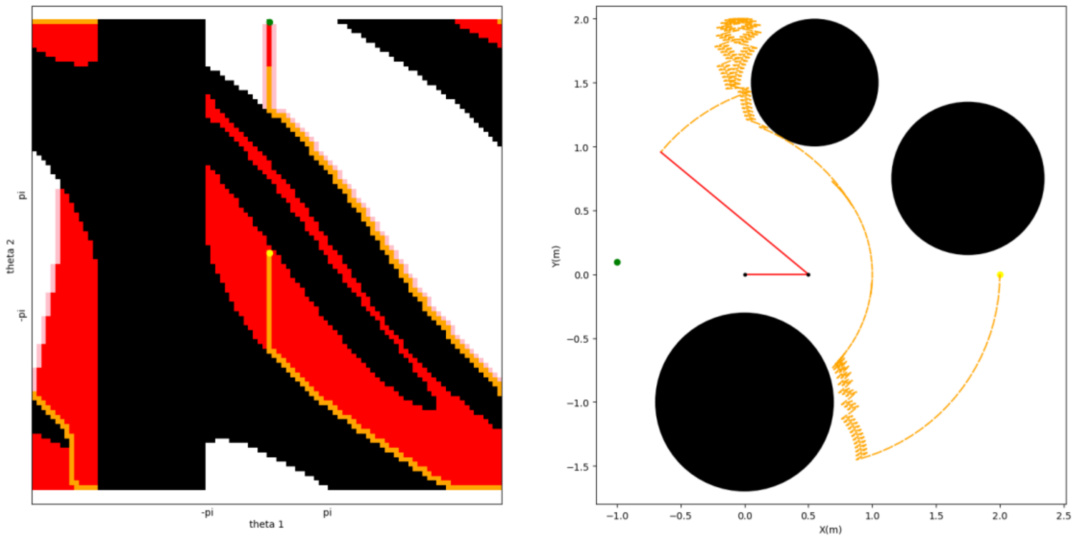


Fig.6 Goal point(50,99) with 10*L2 distance to build up the heuristic map

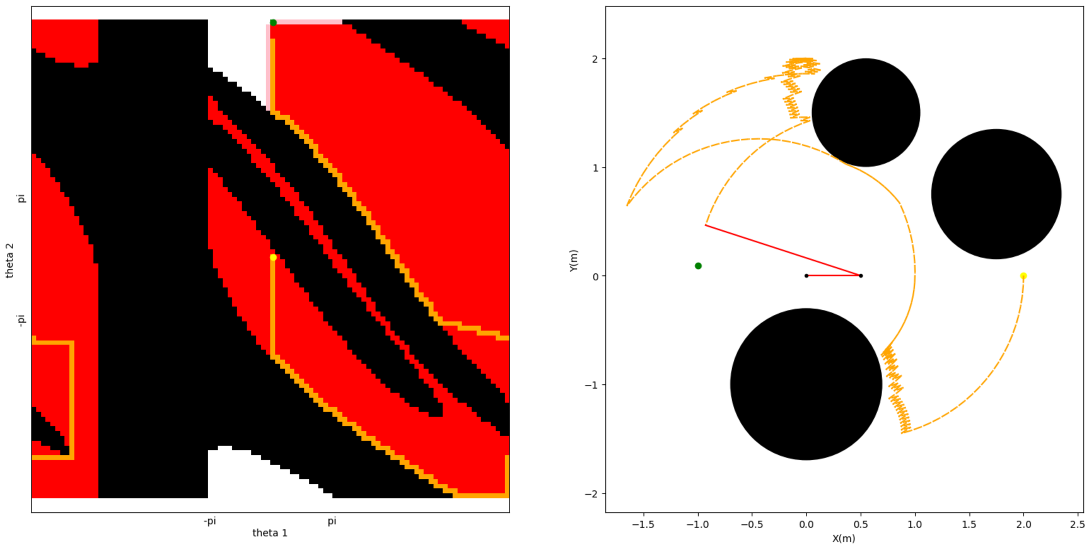


Fig.7 Goal point(50,99) with 1*L2 distance to build up the heuristic map

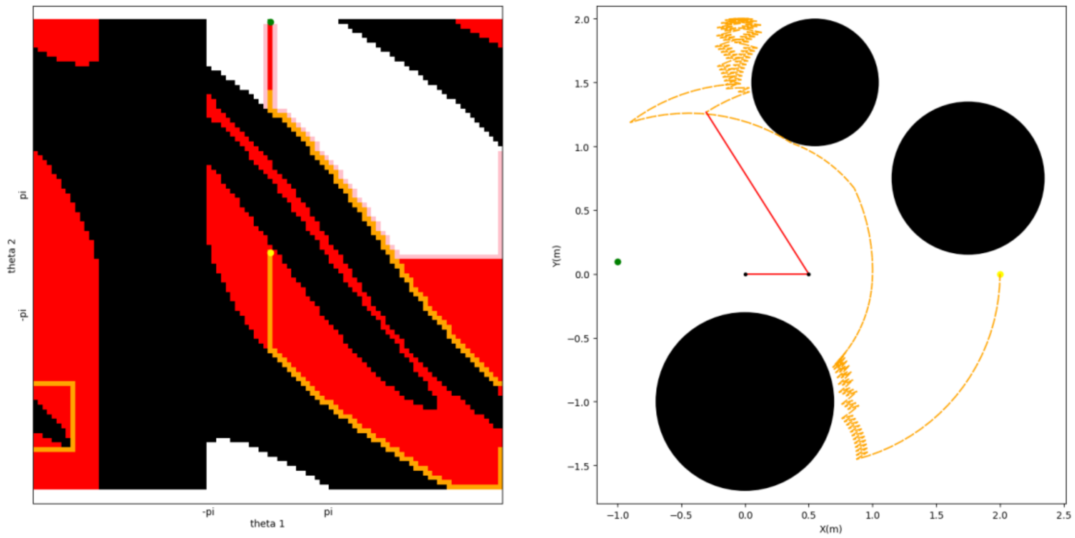


Fig.8 Goal point(50,99) with 1*L1 distance to build up the heuristic map

(v).

In heuristic map, I only changed the distance strategy, and the weighting. I found that if the cost of distance between current node and goal is weighted larger, then it will search for fewer area. And if the weighting of the heuristic cost is 0.1 which is much smaller than g(x), then it will go over much more area. In my intuition, if the h(x) is higher than g(x), which means it is more careful about every next step to make sure every next step is closer to the goal. While the h(x) is much smaller than g(x), then it will more focus on what it have done, and if every next step doesn’t matter about the total cost too much, then he can just take a look more area and pick the one he wants comfortably.

d.

Code of arm_obj_avoid.py

```
from NLinkArm import *

def main():
    # step 0: simulation parameters
    M = 100
    link_length = [0.5, 1.5]
    initial_link_angle = [0, 0]
    obstacles = [[1.75, 0.75, 0.6], [0.55, 1.5, 0.5], [0, -1, 0.7]]
    goal_found = False

    # step 1: robot and environement setup
    arm = NLinkArm(link_length, initial_link_angle)
    # arm.n_links = len(link_length)
    # arm.link_lengths = np.array(link_lengths) [0.5,1.5]
    # arm.joint_angles = np.array(joint_angles) [0,0]
    # arm.points = [[0, 0], [0.5, 0.0], [2.0, 0.0]]

    grid = get_occupancy_grid(arm, obstacles, M)
    print("\nstep 1: robot and environment ready.")

    while not goal_found:
        # step 2: set start and random goal
        start = angle_to_grid_index(initial_link_angle,M)
        #goal = (randint(0,M),randint(0,M))
        goal = (50,99)
        start_js = grid_index_to_angle(start, M)
        goal_js = grid_index_to_angle(goal, M)
        goal_pos = forward_kinematics(link_length, goal_js)
        start_pos = forward_kinematics(link_length, start_js)
        if not (grid[start[0]][start[1]] == 0):
            print("Start pose is in collision with obstacles. Close system.")
            break

        elif (grid[goal[0]][goal[1]] == 0):
            print("\nstep 2: \n\tstart_js    = {}, goal_js    = {}".format(start_js,
```

```
goal_js))
                print("\tstart_pos = {}, goal_pos = {}".format(start_pos, goal_pos))
                goal_found = True

    if(goal_found):
        # step 3: motion planning
        #print('arm.points=',arm.points)
        route = astar_search(grid, start, goal, M)
        print("\nstep 3: motion planning completed.")

        # step 4: visualize result
        print("\nstep 4: start visualization.")
        if len(route) >= 0:
            animate(grid, arm, route, obstacles, M, start_pos, goal_pos, start,
goal)

        print("\nGoal reached.")

def forward_kinematics(link_length, joint_angles):
    posx =
link_length[0]*np.cos(joint_angles[0])+link_length[1]*np.cos(np.sum(joint_angles))
    posy =
link_length[0]*np.sin(joint_angles[0])+link_length[1]*np.sin(np.sum(joint_angles))
    return (posx,posy)

def inverse_kinematics(posx, posy, link_length):
    goal_th = atan2(posy,posx)
    A = link_length[0]
    B = sqrt(posx*posx+posy*posy)
    C = link_length[1]

    C_th = np.arccos(float((A*A + B*B - C*C)/(2.0*A*B)))
    B_th = np.arccos(float((A*A + C*C - B*B)/(2.0*A*C)))
    theta1_sol1 = simplify_angle(goal_th + C_th)
    theta2_sol1 = simplify_angle(-pi + B_th)
    theta1_sol2 = simplify_angle(goal_th - C_th)
    theta2_sol2 = simplify_angle(pi - B_th)
```

```
    return [[theta1_sol1, theta2_sol1],[theta1_sol2, theta2_sol2]]

def detect_collision(line_seg, circle):
    # TODO: Return True if the line segment is intersecting with the circle
    #           Otherwise return false.
    #       (1) line_seg[0][0], line_seg[0][1] is one point (x,y) of the line segment
    #       (2) line_seg[1][0], line_seg[1][1] is another point (x,y) of the line segment
    #       (3) circle[0],circle[1],circle[2]: the x,y of the circle center and the circle
radius
    #       Hint: the closest point on the line segment should be greater than radius
to be collision free.
    #           Useful functions: np.linalg.norm(), np.dot()
    x_vec=line_seg[1][0]-line_seg[0][0]
    y_vec=line_seg[1][1]-line_seg[0][1]
    vec_len=np.sqrt(x_vec**2+y_vec**2)
    unit_vec = [x_vec/vec_len,y_vec/vec_len]
    point_to_circle = [circle[0]-line_seg[0][0],circle[1]-line_seg[0][1]]
    projection=np.dot(point_to_circle,unit_vec)
    #projection_len=np.sqrt(projection[0]**2+projection[1]**2)
    #next_point_in_line =
[line_seg[0][0]+projection[0],line_seg[0][1]+projection[1]]
    if projection < 0:
        closest_point = [line_seg[0][0],line_seg[0][1]]
    elif projection > vec_len:
        closest_point = [line_seg[1][0],line_seg[1][1]]
    else :
        projection_vec = [projection*i    for i in unit_vec]
        closest_point = [sum(x) for x in zip(line_seg[0], projection_vec)]
        #closest_point = [line_seg[0][0]+projection_vec[0],line_seg[]
    dist_circle_line = np.sqrt((circle[0]-closest_point[0])**2+(circle[1]-
closest_point[1])**2)
    if dist_circle_line > circle[2]:
        return False
    else:
        return True


def get_occupancy_grid(arm, obstacles, M):
```

```
    #obstacles = [[1.75, 0.75, 0.6], [0.55, 1.5, 0.5], [0, -1, 0.7]]
    grid = [[0 for _ in range(M)] for _ in range(M)]
    theta_list = [2 * i * pi / M for i in range(-M // 2, M // 2 + 1)]
    #print(grid)
    #print(theta_list)
    for i in range(M):
        for j in range(M):

            # TODO: traverse through all the grid vertices, i.e. (theta1, theta2)
combinations
            #       Find the occupacy status of each robot pose.
            #       Useful functions/variables: arm.update_joints(), arm.points,
theta_list[index]

            # points = # somthing...
            goal_joint_angles = np.array([theta_list[i], theta_list[j]])
            arm.update_joints(goal_joint_angles)
            points = arm.points
            collision_detected = False
            for k in range(len(points) - 1):
                for obstacle in obstacles:
                    # TODO: define line_seg and detect collisions
                    #       Useful functions/variables: detect_collision(),
points[index]
                    line_seg = [points[k],points[k+1]]
                    collision_detected = detect_collision(line_seg, obstacle)
                    # line_seg = [something,something]
                    # collision_detected = ?

                    if collision_detected:
                        break
                if collision_detected:
                    break
            grid[i][j] = int(collision_detected)
    return np.array(grid)


#grid =
```

```
def astar_search(grid, start_node, goal_node, M):
	colors = ['white', 'black', 'red', 'pink', 'yellow', 'green', 'orange']
	levels = [0, 1, 2, 3, 4, 5, 6, 7]
	cmap, norm = from_levels_and_colors(levels, colors)

	grid[start_node] = 4
	grid[goal_node] = 5

	print("show grid")
	plt.imshow(grid, interpolation='nearest')
	plt.show()
	print(grid)

	parent_map = [[() for _ in range(M)] for _ in range(M)]

	heuristic_map = calc_heuristic_map(M, goal_node)

	evaluation_map = np.full((M, M), np.inf)
	distance_map = np.full((M, M), np.inf)
	evaluation_map[start_node] = heuristic_map[start_node]
	distance_map[start_node] = 0
	while True:
		grid[start_node] = 4
		grid[goal_node] = 5

		current_node = np.unravel_index(np.argmin(evaluation_map, axis=None),
evaluation_map.shape)
		min_distance = np.min(evaluation_map)

		if (current_node == goal_node) or np.isinf(min_distance):
			break

		grid[current_node] = 2
		evaluation_map[current_node] = np.inf

		i, j = current_node[0], current_node[1]
		#print('arm.points=',NLinkArm.points)
		neighbors = find_neighbors(i, j, M)
```

```
            print('neighbors=',neighbors)
            for neighbor in neighbors:
                if grid[neighbor] == 0 or grid[neighbor] == 5 or grid[neighbor] == 3:
                    evaluation_map[neighbor] =
min(evaluation_map[neighbor],heuristic_map[neighbor]+distance_map[current_nod
e]+1)
                    if
evaluation_map[current_node]>heuristic_map[neighbor]+distance_map[current_no
de]+1:
                        parent_map[neighbor[0]][neighbor[1]]=current_node
                    distance_map[neighbor]=distance_map[current_node]+1
                    grid[neighbor]=3
                    # TODO: Update the score in the following maps
                    #       (1) evaluation_map[neighbor]
                    #       (2) distance_map[neighbor]:    update distance using
distance_map[current_node]
                    #       (3) parent_map[neighbor]:       set to current node
                    #       (4) grid[neighbor]:                  set value to 3
                    #       Update criteria: new evaluation_map[enighbor] value is
decreased
        if np.isinf(evaluation_map[goal_node]):
            route = []
            print("No route found.")
        else:
            route = [goal_node]
            while parent_map[route[0][0]][route[0][1]] != ():
                # TODO: find the optimal route based on your exploration result
                #           Useful functions:
                #     (1) route.insert(index, element): to add new node to route
                #     (2) parent_map[index1][index2]: find the parent of the node at
grid coordinates (index1,index2)
                #     (3) route[0][0], route[0][1] to access the grid coordinates of a
node
                # route.insert(something...)
                index1,index2=parent_map[route[0][0]][route[0][1]]
                route.insert(0,(index1,index2))
            print("The route found covers %d grid cells." % len(route))
        return route
```

```
def find_neighbors(i, j, M):
        neighbors = []
        ileft = i-1
        iright = i+1
        jlow = j-1
        jhigh = j+1
        if (i == 0):
                ileft = M-1
        if (i == M-1):
                iright=0
        if (j==0):
                jlow = M-1
        if (j==M-1):
                jhigh=0
        neighbors.append((ileft,j))
        neighbors.append((iright,j))
        neighbors.append((i,jhigh))
        neighbors.append((i,jlow))
        # TODO: add the four neighbor nodes to the neighbor list
        #        grid index: i goes from 0 to M-1 (M possible values)
        #        grid index: j goes from 0 to M-1 (M possible values)

        # at i=0, theta1 = -pi
        # at j=0, theta2 = -pi
        # at i=M-1, theta1 = pi
        # at j=M-1, theta2 = pi

        # So be aware of the rounding effect in finding the neighbors at border nodes.
        # Useful function: neighbors.append((index1, index2)), where index1, index2 are
the grid indices of it's neighbors

        #neighbors.append((index1,index2))
        return neighbors


def calc_heuristic_map(M, goal_node):
```

```
    X, Y = np.meshgrid([i for i in range(M)], [i for i in range(M)])
    #heuristic_map = 10*np.sqrt((X-goal_node[1])**2+(Y-goal_node[0])**2)
    heuristic_map = 0.1*(np.abs(X - goal_node[1]) + np.abs(Y - goal_node[0]))
    print("heuristic_map=",heuristic_map)
    for i in range(heuristic_map.shape[0]):
        for j in range(heuristic_map.shape[1]):
            heuristic_map[i, j] = min(heuristic_map[i, j],
            i + 1 + heuristic_map[M - 1, j],
            M - i + heuristic_map[0, j],
            j + 1 + heuristic_map[i, M - 1],
            M - j + heuristic_map[i, 0]
            )
    #print('heuristic_map=',heuristic_map)
    return heuristic_map

def simplify_angle(angle):
    if angle > pi:
        angle -= 2*pi
    elif angle < -pi:
        angle += 2*pi
    return angle
if __name__ == '__main__':
    main()
```