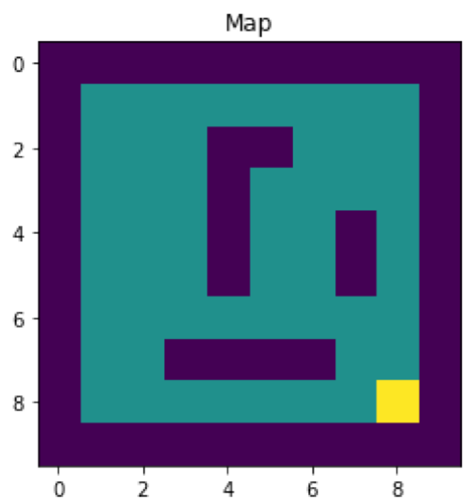
ESE 650 - Learning in Robotics
Homework 3

Akash Sundar

April 2023

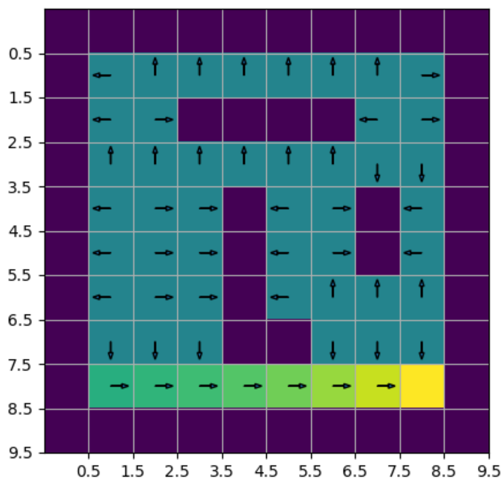
1 Problem 1

a) Obstacle Map:

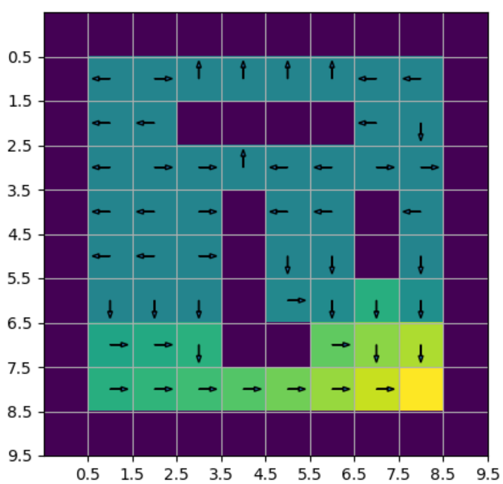


b) $J^{\pi^{(0)}}$ (x) is plotted as follows.
Cost of going into obstacles and staying in obstacles is 10. Reward at goal is 10 (or cost is -10).
Control was initialised with all right

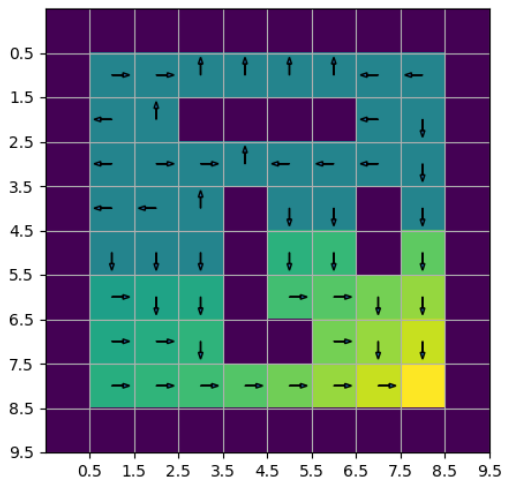
c) Policy Iteration Output:


0.5
1.5
2.5
3.5
4.5
5.5
6.5
7.5
8.5
9.5
0.5 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5

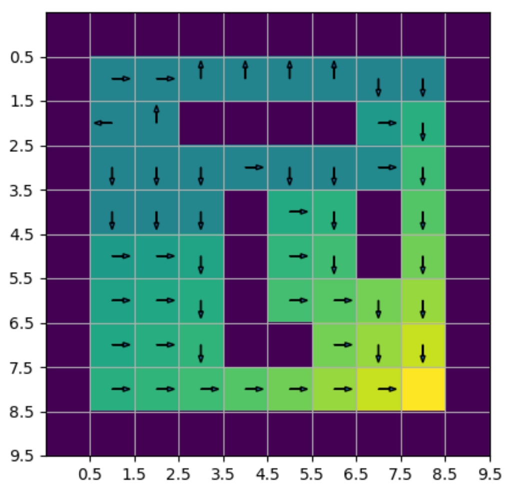
Iter 1:


0.5
1.5
2.5
3.5
4.5
5.5
6.5
7.5
8.5
9.5
0.5 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5

Iter 2:


0.5
1.5
2.5
3.5
4.5
5.5
6.5
7.5
8.5
9.5
0.5 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5

Iter 3:


0.5
1.5
2.5
3.5
4.5
5.5
6.5
7.5
8.5
9.5
0.5 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5

Iter 4:

2 Problem 2

c) Dynamics Step

```
def dynamics_step(s, t):

    noise_vector = np.random.multivariate_normal([0,0,0], s.Q, s.n).T
    control = s.get_control(t)

    for i in range(s.n):
        s.p[:,i] = smart_plus_2d(smart_plus_2d(s.p[:,i],control), noise_vector[:,i])
```

Obtained noise vector from a multivariate normal distribution. Obtained control from the predefined function.
For each particle, the particle was updated using the provided smart plus 2d functions.

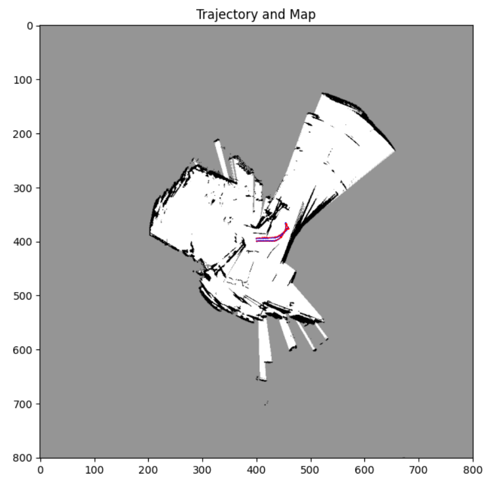
d) Observation Step

```
def observation_step(s, t):
    t_lidar = s.find_joint_t_idx_from_lidar(s.lidar[t]['t'])
    head_angle = s.joint['head_angles'][0, t_lidar]
    neck_angle = s.joint['head_angles'][1, t_lidar]
    angles = s.lidar_angles
    d = s.lidar[t]['scan']

    lin_space = lambda x,y, dim, dist: np.linspace(x[dim], y[dim], dist, endpoint=False, dtype=int)
    xy = lambda x,y,dist: (lin_space(x,y,0,dist), lin_space(x,y,1,dist))

    if t == 0:
        grid = s.map.grid_cell_from_xy(*s.p.T[0, :2])
        lidar_scanned_pts = s.rays2world(s.p.T[0], d, head_angle, neck_angle, angles)
        occ = s.map.grid_cell_from_xy(lidar_scanned_pts[0], lidar_scanned_pts[1])
        s.map.cells[occ[0], occ[1]] = 1

        for i in occ.T:
            dist = int(np.linalg.norm(i - grid.T))
            x, y = xy(grid, i, dist)
        free_cells = np.unique(np.hstack((grid, np.vstack((x.T, y.T)))), return_index=False, axis=1)

        s.map.cells = np.zeros_like(s.map.cells)
        s.map.cells[occ[0], occ[1]] = 1
        s.map.cells[free_cells[0], free_cells[1]] = 0
```

```python
else:
    log_obs = np.zeros(s.p.shape[1])
    for i in range(0, s.p.shape[1]):
        lidar_scanned_pts = s.rays2world(s.p.T[i], d, head_angle, neck_angle, angles)
        occ = s.map.grid_cell_from_xy(lidar_scanned_pts[0], lidar_scanned_pts[1])
        log_obs[i] = np.sum(s.map.cells[occ[0], occ[1]])

    s.w = s.update_weights(s.w, log_obs)
    p = s.p.T[np.argmax(s.w)]

    grid = s.map.grid_cell_from_xy(p[0], p[1])

    lidar_scanned_pts = s.rays2world(p, d, head_angle, neck_angle, s.lidar_angles)

    for i in occ.T:
        dist = int(np.linalg.norm(i - grid.T))
        x, y = xy(grid, i, dist)
    free_cells = np.unique(np.hstack((grid, np.vstack((x.T, y.T)))), return_index=False, axis=1)

    occ = s.map.grid_cell_from_xy(lidar_scanned_pts[0], lidar_scanned_pts[1])

    np.add.at(s.map.log_odds, (occ[0], occ[1]), s.lidar_log_odds_occ)
    np.add.at(s.map.log_odds, (free_cells[0], free_cells[1]), s.lidar_log_odds_free)

    s.map.log_odds = np.clip(s.map.log_odds, -s.map.log_odds_max, s.map.log_odds_max)

    s.map.cells = np.zeros_like(s.map.cells)
    s.map.cells[s.map.log_odds >= s.map.log_odds_thresh] = 1
    s.map.cells[s.map.log_odds <= s.lidar_log_odds_free] = 0

    s.resample_particles()

    return s.lidar[t]['xyth'], s.p[:, np.argmax(s.w)]
```

The world coordinates were found using the rays2world function defined before. The obstacle map was checked to provide more insight on current scan. Particles were resampled using np.searchsorted to identify n largest cumulative sums.

f) I followed the instructions given in the problem pdf carefully. Following is an example of a map generated by running the code



3 Problem 3

a) Given $M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(\dot{q}) = Bu$ and the individual definitions for M, C, G and B, we can rewrite it in the form:

$$\begin{bmatrix} \dot{q} \\ \ddot{q} \end{bmatrix} = \begin{bmatrix} \dot{q} \\ f(q,\dot{q}) \end{bmatrix} \approx \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \frac{\partial f(q,\dot{q})}{\partial q_1}\big|_{x=x_f} & \frac{\partial f(q,\dot{q})}{\partial q_2}\big|_{x=x_f} & \frac{\partial f(q,\dot{q})}{\partial \dot{q_1}}\big|_{x=x_f} & \frac{\partial f(q,\dot{q})}{\partial \dot{q_2}}\big|_{x=x_f} \end{bmatrix} \begin{bmatrix} q_1 - \pi \\ q_2 - 0 \\ \dot{q}_1 - 0 \\ \dot{q}_2 - 0 \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ f(x_f) \end{bmatrix}$$

$$\frac{\partial f(q,\dot{q})}{\partial q_1}\big|_{x=x_f} = \frac{1}{-I_1 I_2 - I_2 l_2^2 m_2 + \frac{1}{4} l_1^2 l_2^2 m_2^2} \begin{bmatrix} -I_2(\frac{1}{2} g l_1 m_1 + g l_1 m_2 + \frac{1}{2} g l_2 m_2) + \frac{1}{2} g l_2 m_2 (I_2 + \frac{1}{2} l_1 l_2 m_2) \\ \frac{1}{2} g l_2 m_2 (-I_1 - I_2 - l_1 l_2 m_2 - l_2^2 m_2) + (I_2 + \frac{1}{2} l_1 l_2 m_2)(\frac{1}{2} g l_1 m_1 + g l_1 m_2 + \frac{1}{2} g l_2 m_2) \end{bmatrix}$$

$$\frac{\partial f(q,\dot{q})}{\partial q_2}\big|_{x=x_f} = \frac{1}{-I_1 I_2 - I_2 l_2^2 m_2 + \frac{1}{4} l_1^2 l_2^2 m_2^2} \begin{bmatrix} \frac{1}{2} g l_2 m_2 (\frac{1}{2} l_1 l_2 m_2) \\ \frac{1}{2} g l_2 m_2 (-I_1 - \frac{1}{2} l_1 l_2 m_2 - l_2^2 m_2) \end{bmatrix}$$

$$f(x_f) = \frac{1}{-I_1 I_2 - I_2 l_2^2 m_2 + \frac{1}{4} l_1^2 l_2^2 m_2^2} \times \begin{bmatrix} \tau (I_2 + \frac{1}{2} l_1 l_2 m_2) \\ \tau (-I_1 - I_2 - l_1 l_2 m_2 - l_2^2 m_2) \end{bmatrix}$$

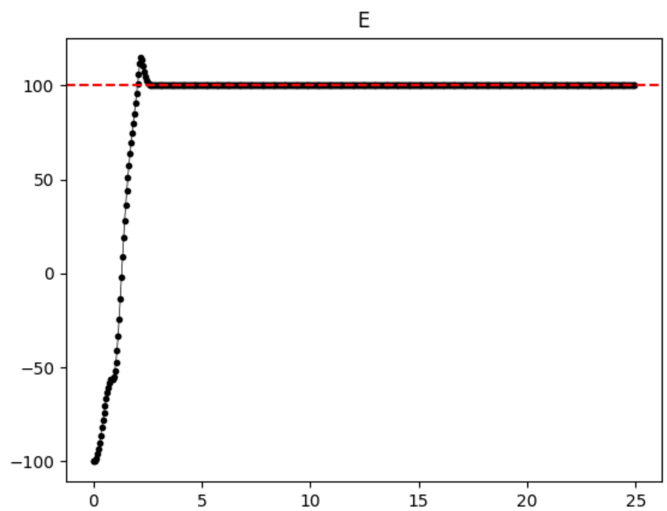
$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \frac{\partial f(q,\dot{q})}{\partial q_1}\big|_{x=x_f} & \frac{\partial f(q,\dot{q})}{\partial q_2}\big|_{x=x_f} & \frac{\partial f(q,\dot{q})}{\partial \dot{q_1}}\big|_{x=x_f} & \frac{\partial f(q,\dot{q})}{\partial \dot{q_2}}\big|_{x=x_f} \end{bmatrix}$$

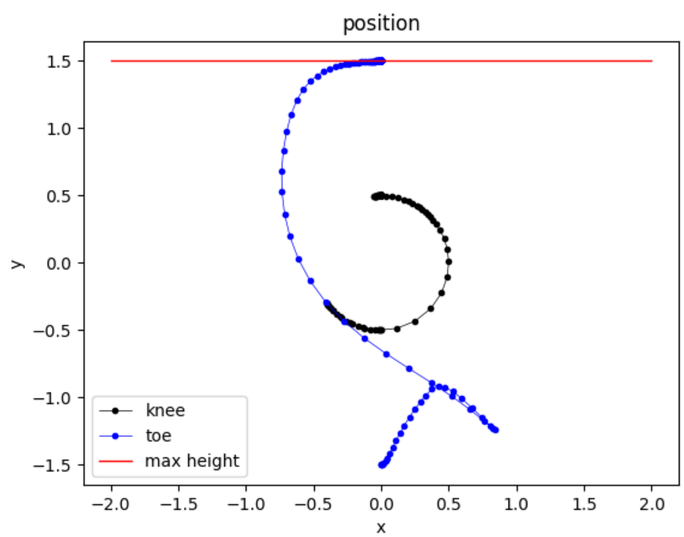
$$B = \frac{1}{-I_1 I_2 - I_2 l_2^2 m_2 + \frac{1}{4} l_1^2 l_2^2 m_2^2} \times \begin{bmatrix} I_2 + \frac{1}{2} l_1 l_2 m_2 \\ -I_1 - I_2 - l_1 l_2 m_2 - l_2^2 m_2 \end{bmatrix}$$

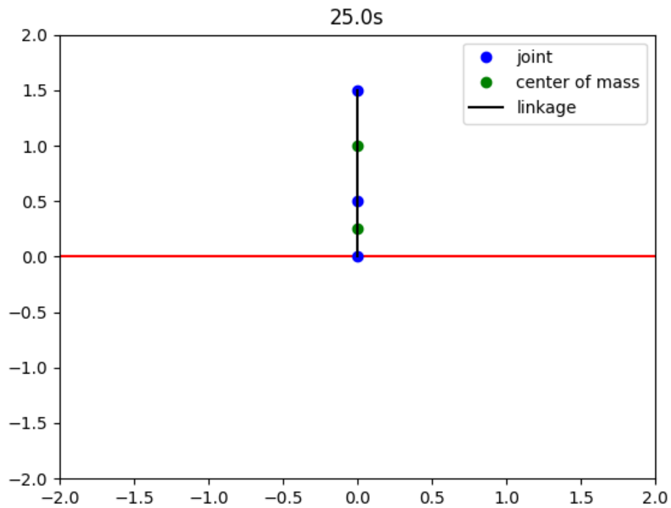
b) I implemented two conditions. I determined an arbitary case to determine the region around which the dynamics were linearized. If the acrobot was outside that region, I use the swing up input in order to try to move it to the desired final position. By manual trials, I determined 1e3 to be a suitable threshold above which the error calculated as (x - xf).T @ P @ (x - xf) must lie.
I used an initial K of [0,0,1] to provide it the initial perturbation required for the swing.

Derived the func3() from the equations provided in the pdf for the energy controller to be as such:

```
def func3():
    M11, M12, M21, M22 = M[0,0], M[0,1], M[1,0], M[1,1]
    ddq1 = (t[0] - M12*ddq_des)/ M11
    u = M21*ddq1 + M22*ddq_des - t[1] + eps
    return u
```


E
100
50
0
−50
−100
0
5
10
15
20
25


position
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
y
knee
toe
max height
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x


25.0s
joint
center of mass
linkage
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0