Hao Le
A15547504
ECE 172A Winter 2022 HW1

Academic Integrity Policy: Integrity of scholarship is essential for an academic community. The University expects that both faculty and students will honor this principle and in so doing protect the validity of University intellectual work. For students, this means that all academic work will be done by the individual to whom it is assigned, without unauthorized aid of any kind. By including this in my report, I agree to abide by the Academic Integrity Policy.

1. Programming with Matrices

i.

```
import numpy as np

A = np.array([[66, -10, 16, 14, 65], [-91, 28, 97, -42, 4], [13, 50, -96, 92, -85], [21, -96, 49, 34, 93], [-53, 96, 80, 96, -68]])
B = np.array([[1, 1, 0, 1, 0], [1, 0, 1, 1, 1], [0, 0, 0, 1, 0], [1, 1, 0, 0, 1], [0, 0, 1, 1, 0]])

print(A)
print(B)
```

```
[[ 66 -10  16  14  65]
 [-91  28  97 -42   4]
 [ 13  50 -96  92 -85]
 [ 21 -96  49  34  93]
 [-53  96  80  96 -68]]
[[1 1 0 1 0]
 [1 0 1 1 1]
 [0 0 0 1 0]
 [1 1 0 0 1]
 [0 0 1 1 0]]
```

ii.

```
print(np.argwhere(A < -70) + 1)
```

```
[[2 1]
 [3 3]
 [3 5]
 [4 2]]
```

By adding 1, the result is more readable since it is 1-indexed.

The (row, column) where the element is less than 70 are: (2, 1), (3, 3), (3, 5), (4,2).

iii.

```
C = np.multiply(A, B)
print(C)
```

```
[[ 66 -10   0  14   0]
 [-91   0  97 -42   4]
 [  0   0   0  92   0]
 [ 21 -96   0   0  93]
 [  0   0  80  96   0]]
```

iv.

```
innerProduct = np.dot(C[:, 2], C[4, :])
print(innerProduct)
```

```
0
```

The result is 0.

v.

```
maxValueFourthColumnC = np.amax(C[:, 3])
print(maxValueFourthColumnC)
print(np.argwhere(C == maxValueFourthColumnC) + 1)
```

```
96
[[5 4]]
```

The max value in C's fourth column is 96. This is on the 5th row and 4rd column.

vi.

```
firstRowC = C[0, :]

DRow0 = np.multiply(firstRowC, firstRowC)
DRow1 = np.multiply(firstRowC, C[1, :])
DRow2 = np.multiply(firstRowC, C[2, :])
DRow3 = np.multiply(firstRowC, C[3, :])
DRow4 = np.multiply(firstRowC, C[4, :])

D = np.vstack((DRow0, DRow1, DRow2, DRow3, DRow4))
print(D)
```

```
[[ 4356   100     0   196     0]
 [-6006     0     0  -588     0]
 [    0     0     0  1288     0]
 [ 1386   960     0     0     0]
 [    0     0     0  1344     0]]
```

vii.

```
firstRowD = D[0, :]

ERow0 = np.multiply(firstRowD, firstRowD)
ERow1 = np.multiply(firstRowD, D[1, :])
ERow2 = np.multiply(firstRowD, D[2, :])
ERow3 = np.multiply(firstRowD, D[3, :])
ERow4 = np.multiply(firstRowD, D[4, :])

E = np.vstack((ERow0, ERow1, ERow2, ERow3, ERow4))
print(E)
```

```
[[ 18974736     10000         0     38416         0]
 [-26162136         0         0   -115248         0]
 [        0         0         0    252448         0]
 [  6037416     96000         0         0         0]
 [        0         0         0    263424         0]]
```

2. Robot Traversal

i.

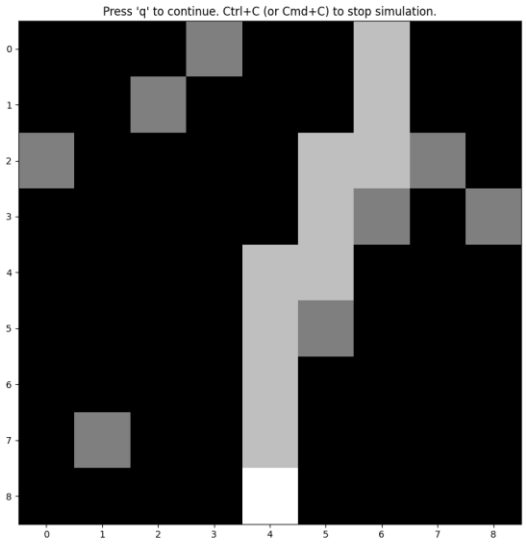
loc keeps track of the current and all the past locations of the robot. This is a list of 2D coordinates, with the last added coordinate being the current position.

Adding [1, 0] will make the robot move 1 step south only; this is because on the image display, the first coordinate represent the row index.

Adding an object to [3, 6] causes the robot to be stuck in an infinite loop due to the break condition of the while loop. Since there is an object in the way of the robot as it moves south, it can never add the location [3, 6] to loc and thus advance past the obstacle.

The robot is not intelligent because it does not act upon the obstacle to adapt and find a new solution to reach its south side goal. It would need to somehow acquire the new information regarding the obstacle and change its state so as the simulation carries on, it is not the same result (staying in one spot) infinitely.

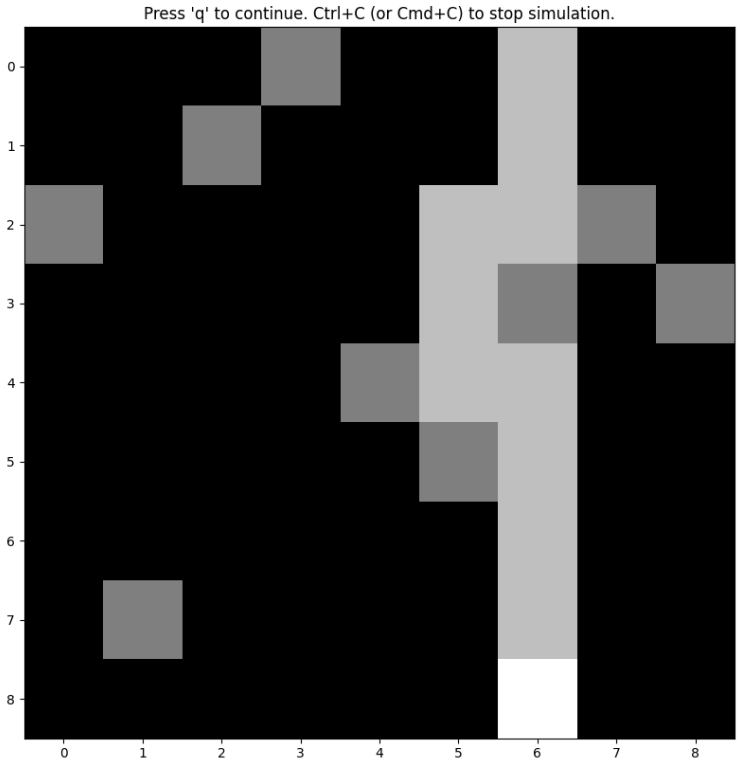
ii.



By adding the detectObject() function, the robot has the ability to sense the presence of an obstacle in front of it and intelligently use this information to adapt and come up with a new solution to reach the goal. In this case, it detects an obstacle in the southern direction, and as a result adds -1 to its column coordinate which is equivalent to moving west 1 step, effectively getting out of the obstacle's way and proceeds to the south side goal.

iii.

```
# If there is an object to the South, move a different direction
# START
if detectObject(loc, obj, 'S') and not detectObject(loc, obj, 'W'):
    nextStep = loc[-1] + np.array([0, -1])
if detectObject(loc, obj, 'S') and not detectObject(loc, obj, 'E'):
    nextStep = loc[-1] + np.array([0, 1])
# STOP
```



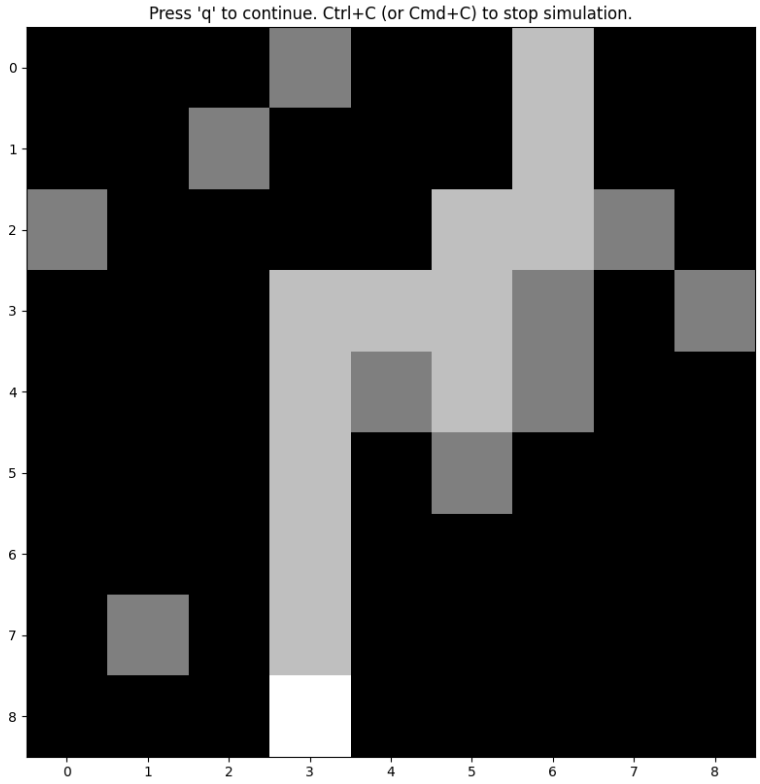
iv.

```
# If there is an object to the South, move a different direction
# START


if detectObject(loc, obj, 'S'):
    if not detectObject(loc, obj, 'W'):
        nextStep = loc[-1] + np.array([0, -1])
    elif not detectObject(loc, obj, 'E'):
        nextStep = loc[-1] + np.array([0, 1])
    elif not detectObject(loc, obj, 'N'):
        nextStep = loc[-1] + np.array([-1, 0])

if haveIBeenHereBefore(loc, nextStep):
    if not detectObject(loc, obj, 'W'):
        nextStep = loc[-1] + np.array([0, -1])
    elif not detectObject(loc, obj, 'N'):
        nextStep = loc[-1] + np.array([-1, 0])


# STOP
```



3. Bug Algorithms

3.1

i.

Since the goal line intersects all obstacles, the bug will have to traverse the entire perimeter of each obstacles, plus travel to the closest point to the goal via the shortest path along the perimeter – the worst path would be half the perimeter.

In addition to travelling each obstacles perimeter, plus half of that more, the worst case would be where the thickness of the obstacles does not contribute to the net distance between the bug and the goal i.e. the obstacles are super thin, and has the flat face directed towards the goal. This means the bug would still have to travel the original distance as the crow flies.

The upper bound is 200 + (600 + 0.5 * 600) + 4 * (12 + 12 * 0.5) = 200 + 900 + 72 = 1172m

In the best case scenario, since the goal line will always have to intersect the obstacles, this means each obstacle's perimeter would still need to be traversed. However, to get to the closest point to the goal, the bug virtually need not travel any distance i.e. the intersection is an infinitesimally small amount. Thus, aside from the perimeters, the bug would only need to travel the original as-the-crow-flies distance between the start and goal, from obstacle to obstacle.

The lower bound is 200 + 600 + 4*12 = 848m

ii.

Assuming that the goal line only intersects each obstacle twice:

The worst case would be where the bug would have to traverse the entire length of each obstacles perimeter as a result from moving left – additionally, the intersection must be a very small section of the obstacle, since the next point on the perimeter that intersects the goal line is virtually in the same spot as the original point where the bug started to traverse.

Moreover, the bug would still need to travel the original distance between the start and goal, since the obstacles do not contribute to the net distance gained.

The upper bound is 200 + 600 + 4 * 12 = 812m

The best case would be where the goal line intersects the left-most sides of the obstacles. In effect, the bug will stop at each obstacle, traverse leftwards which is just going along the leftward side, then jump to the next obstacle, gaining a net distance that equals to that side of the previous obstacle. The obstacles all contribute to the bug's net distance gain. This means the least distance travelled must be of how the crow flies.

The lower bound is 200m

3.2

i.

All bugs will find a solution

ii.

Though Bugs 0 and 2 have comparable distances, though Bug 2 may travel the least distance since it keeps close to the initial goal line, only strayed away by half the perimeter of Geisel. However, Bug 0 has to travel half of Geisel and half of Canyon, plus the distance in-between.

Bug 1 will behave much like Bug 0, though it has to go through roughly one and a half of Geisel's perimeter before landing on the point closest to the end, which is where Bug 2 ends up, but without the full search.

3.3

i.

All bugs will find a solution

ii.

Bug 2 will find the shortest path, since it only has to traverse the entire perimeter of the rectangular obstacle next to Geisel, as well as roughly half of Geisel's perimeter. The rest is following the initial goal line.

Bug 1 will be longer because it has to traverse the entire perimeters of Geisel and the rectangle, plus half of Geisel to get to the point closest to the goal, roughly where Bug 2 ends up anyways.

Bug 0 has to travel part of Geisel, the rectangle adjacent, and also Canyon, though this distance can be comparable to Bug 1's.

3.4

i.

All bugs will find a solution

ii.

Bug 0 will find the shortest path in this case, since it gets an advantage by having the most convex path taken. It's net distance gain is not hindered by the concave part created by obstacles pink and blue.

Bugs 1 have to traverse all of Geisel, and the adjacent rectangular obstacles, plus half of Geisel – this is adding distance that does not greatly contribute to the net distance gain. Plus, the grey obstacle near the end will be encountered by Bug 1 since it is in the way of the goal line – this will also have to be traverse approximately 1.5 times.

Bug 2 has a similar situation to Bug 1, though it does not have to traverse the entire perimeters of the mentioned obstacles.

iii.

Bug 0

Bug 1

Bug 2

4. BONGO Blocks

1. Since it is a binary classifying task, there are two options, thus a random guess would be correct 50% of the time. For 100 trials, 50 of them are expected to be correct.

2. Puzzle of type 1 is picking which BONGO block contains the shape, regardless of size and orientation, in the test image; the other BONGO block will be empty. Type 2 puzzle has shapes in both BONGO blocks. However, one will contain the shape, with the exact size and orientation, that is depicted in the test image. Type 3 puzzle is like type 2, but now the BONGO block that contains the shape present in the test image is different in size and orientation. Type 4 is like type 3, but instead of the shapes having at most four sides - a square - the shapes can now have up to the six sides of a hexagon.

3.

```python
def findNumberOfVertex(shape_img):
    contour, _ = cv2.findContours(shape_img.astype(np.uint8)*255, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
    approximatedContour = cv2.approxPolyDP(contour[0], 0.01 * cv2.arcLength(contour[0], True), True)
    return len(approximatedContour)
```

```python
def classifier1(A, B, test_img):

    if True in A:
        return 0
    else:
        return 1

def classifier2(A, B, test_img):

    if np.array_equal(A, test_img):
        return 0
    else:
        return 1

def classifier3(A, B, test_img):

    testVertexNum = findNumberOfVertex(test_img)
    AVertexNum = findNumberOfVertex(A)

    if testVertexNum == AVertexNum:
        return 0
    else:
        return 1

def classifier4(A, B, test_img):
    testVertexNum = findNumberOfVertex(test_img)
    AVertexNum = findNumberOfVertex(A)

    if testVertexNum == AVertexNum:
        return 0
    else:
        return 1
```

The 1st classifier checks if any of the blocks have a shape in them i.e. if any of the pixels are true. The one with true present has a shape present, therefore matches with the test image.

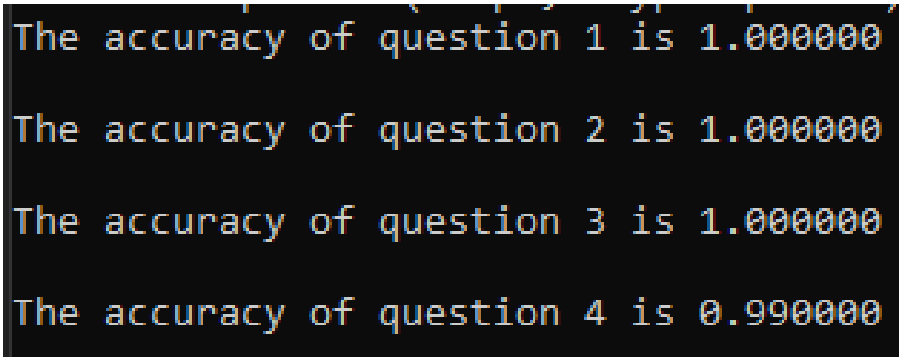
The 2nd classifier checks which block matches exactly in terms of pixel value to the test image. This is due to the absence of scaling or rotation done to the shapes in the BONGO blocks.

The 3rd and 4th classifier uses OpenCV's findContours and approxPolyDP functions to first approximate the contour points of the shape present, then reduce down the number of segments to a certain degree where sides are represented by just one segment. The output contour then has exactly the same number of 2D vectors as the number of sides of the shape it draws out. The vertex count can then be extracted from the number of 2D vectors using len(). The vertex count, and consequently the type of shape, can be extracted from the test image; this, invariant to scale and orientation, will match with the BONGO block with the same vertex count.

4.

This system is somewhat intelligent because it takes in visual information of the test and BONGO blocks to make correct matching decisions instead of being preprogrammed to take random guesses. It can adapt to variances of rotation and scale to the same type of shape by using computer vision techniques. However, the system may not be robust to noise and may perform just as well as a preprogrammed system in harder, more adversarial puzzles. My classifier for types 3 and 4 puzzles will generalize well to types 1 and 2, since they are simpler versions of the former types – the presence of contours or not will yield the answer for type 1, and matching vertex count can apply to type 2.

Accuracy reported:



5. Survey (Spot)

Spot is a commercial zoomorphic robot manufactured by Boston Dynamics designed to be used in a wide range of applications - an article by RumbleRum titled “Boston Dynamics Dog Robot ‘Spot’ Has Been Employed By Norwegian Oil Company To Detect Gas Leaks” reports an application in which Spot assumes the role of an autonomous industrial robot whose job is to look for potentially life-threatening gas leaks within a complex environment.

Spot as a base product allows basic gaiting and full vision of surroundings. The basic package comes with five monochromatic cameras [2] that overall produce 360 degrees of vision that allow motion controllers running on the onboard computer to not only react to unexpected obstacles but also plan motion predictively - this is done via simultaneous localization and mapping (SLAM) algorithms. Although the details on the types of actuators used is proprietary, by observing that Spot can perform dog-like, natural gaits, jump and absorb the impact upon landing on the ground, and react to external forces that push it to the side, there are certainly embedded force sensors that feed into reactive controllers.

A variety of sensors can be added to Spot by attaching modules as payloads [3], allowing expandability of applications. For example, Spot can be a social robot in the sense that it can replace the role of a therapy dog - in this case, a full color camera or heart rate monitor module can be utilized to better effectively guage a human’s mental state and inform internal algorithms to plan Spot’s nurturing behavior. Conversely, Spot can be a teleoperated robot for hazardous situations. An external robot arm and grasping end effector can be attached, enabling a human to remotely control Spot to walk into a dangerous environment and pick up objects. In the case of Norwegian Oil Company, an atmospheric sensor was attached to the payload interface of Spot, allowing the detection of anomalies in the air indicative of a gas leak [1]. Additionally, since Spot was programmed to autonomously patrol the entire worksite, this saves on human labor.

The advent of autonomous patrolling agents such as Spot gives high-risk industries a cost-effective and thorough method to increase safety. It is no surprise that many industrial tragedies occur because of human error and incompetence, not to mention errors in badly-maintained system sensors and controllers. Spot provides an external and independent way to check the integrity of a system.

Spot’s monochromatic vision system may pose an issue due to the loss of color data. In an instance where a certain environment may have similar-looking textures, color data is key to distinguishing features such as the foreground and the background, information that is required for predictive and reactive motion planning. Moreover, since Spot’s main form of travel is gaiting, fixed cameras on the body are subject to shaking and distortion. Perhaps this issue can be remedied via stabilization algorithms that fuse inertial sensor readings.

Technical issues that result from the complexity of Spot can hinder its productivity. With 12 actuators, any one of them can break from unexpected backdriving. Another issue can be onboard power distribution. The constant forces required to hold Spot up and computation of onboard CPU/GPUs translate to high power consumption. Plus, Spot needs to be compact in size which constrains the size of the battery.

Although Spot physically does not present a threat to a human, the modules attached to its payload can. For example, if Spot was tasked to cut up fruits, monochromatic cameras may detect a human hand as a fruit. In that case, some reactive behavior such as scream detection via microphone must be implemented to act as an emergency stop. In general, the objective of Spot must be cautiously designed to not permit greedy and irrational behavior. Moreover, the payload interface must implement a verification protocol that guards against third-party modules that enable dangerous actions - the modules must be electronically verified by Boston Dynamics.

Works Cited

[1] Hanif, Maheen, *Boston Dynamics Dog Robot 'Spot' Has Been Employed By Norwegian Oil Company To Detect Gas Leaks*, RumbleRum, 2020, https://www.rumblerum.com/boston-dynamics-robot-dog-spot-norwegian-oil-company-gas-leaks/

[2] Boston Dynamics, *About Spot*, accessed Jan 2022, https://dev.bostondynamics.com/docs/concepts/about_spot

[3] Boston Dynamics, *Spot*, accessed Jan 2022, https://www.bostondynamics.com/products/spot