Aerial Robotics Kharagpur Task Round Documentation

VISHAL BAGARIA 20EX20036

Abstract— **This document briefly summarizes my efforts to complete all the Tasks assigned to me by the ARK team as a part of the final round of selection to the software team. From learning Python from scratch to attempting to code an Artificial Neural Network, it has been one of the most enthralling journeys of my life. Over the course of the previous one and a half months, I extensively learnt Python and the applications of its various libraries in numerous fields, from decoding a Maze to coding a 3D - TicTacToe, I have explored an array of topics which have broadened my horizon extensively. I am grateful to be given such an opportunity.**

I have learnt about numerous things, in the next few lines, I will try to encapsulate them. Firstly, I learnt about Image Processing using various Python Libraries namely PIL, OpenCV. I have learnt how to read an image as an array of RGB (or BGR) pixels and convert them into NumPy arrays and perform various operations. This can be used when a drone feeds a GrayScale image to the control centre to remove the noise from the image and amplify the pixels in the image in order to improve the resolution of the image. Secondly, I also learnt about the working of Statistical Determination Algorithms like the Kalman Filter (Unscented, Extended) , Markov Localisation which are extensively used in Aviation to remove noise from the readings and thereby minimising the errors hence, predicting the correct location and trajectory of an aerial vehicle. Finally, I learnt to code AIs to play various games using MiniMax Algorithm and other sophisticated techniques like Reinforcement Learning using Q-Value Search etc, which have significantly developed my logical thinking and have given me a brief idea of the backend of Artificial Neural Networks and Machine Learning.

I. Problem Statement Task 2.1

We were provided with a code snippet and asked to use optimisation techniques on it to reduce the time complexity (running time) of the code. Since, I did not have prior knowledge of coding in C++, I attempted this task at last and spent considerably less time on it as compared to others, but still I have tried my best. We were also provided a markdown file with a detailed explanation of the problem statement.

II. Final Approach for Optimize me!

In the first part, we were provided with two functions 'FindmincostA' and 'FindmaxcostB' which calculated the minimum cost and maximum cost respectively of travelling from (i,j)th position in a square matrix (n*n) to the endpoint(n,n) .The functions provided were as below.

```
long long FindMinCostA(int i, int j, int n)
{
    //going out of bounds
    if (i >= n)
        return 0;
    //going out of bounds
    if (j >= n)
        return 0;
    //reaching the last cell
    if (i == n - 1 && j == n - 1)
        return costMatrixA[i][j];
    //going down or right
    return costMatrixA[i][j] + min
    (FindMinCostA(i + 1, j, n),
    FindMinCostA(i, j + 1, n));
}
```

```
long long FindMaxCostB(int i, int j, int n)
{
    //going out of bounds
    if (i >= n)
        return 0;
    //going out of bounds
    if (j >= n)
        return 0;
    //reaching the last cell
    if (i == n - 1 && j == n - 1)
        return costMatrixB[i][j];
    //going down or right
    return costMatrixB[i][j] +
    max(FindMaxCostB(i + 1, j, n),
    FindMaxCostB(i, j + 1, n));
}
```

Both if these matrices were evidently being called with the same parameters to compute the same quantities multiple times.The two recursive calls in the functions would significantly increase the time consumed in the process and calling them repeatedly did not make sense.Calculating FindMinCost(0,0) and FindMaxCost(0,0) would take an exponential amount of time .I have stored the value computed for all possible parameters with which the function can be called using bottom up dynamic programming. Here is my snippet.

```
void calculate() {
    int n = sizen;
    dpA[sizen - 1][sizen - 1] =
    costMatrixA[sizen - 1][sizen - 1];
    dpB[sizen - 1][sizen - 1] =
    costMatrixB[sizen - 1][sizen - 1];
    for (int i = n - 1; i >= 0; i--) {
        for (int j = n - 1; j >= 0; j--) {
            if (i == n - 1 && j == n - 1) {
                continue;
            } else if (i == n - 1) {
                dpA[i][j] = dpA[i][j + 1]
                + costMatrixA[i][j];
                dpB[i][j] = dpB[i][j + 1]
                + costMatrixB[i][j];
            } else if (j == n - 1) {
                dpA[i][j] = dpA[i + 1][j]
                v+ costMatrixA[i][j];
                dpB[i][j] = dpB[i + 1][j]
                + costMatrixB[i][j];
            } else {
                dpA[i][j] = min(dpA[i + 1][j],
                dpA[i][j + 1]) + costMatrixA[i][j];
```

```
                dpB[i][j] = max(dpB[i + 1][j],
                dpB[i][j + 1]) + costMatrixB[i][j];
            }
            dpB_transposed[j][i] = dpB[i][j];
        }
    }
}
```

From the product matrices, I have returned all the possible parameters and used the function Calculate to calculate the cost for traversing using the values stored, in this way we would not have to repeatedly recursively compute the values of the function and I have preferred iterations over recursion here,as we all know that time complexity of iterations is much lower than recursion and hence it can effectively work for larger dimensional square matrices as well. I have used bottom up dynamic programming in this snippet to calculate the parameters using stored values rather than recursively obtaining the parameters every time.

For the next part of the program, we had been provided a productmat which stored the value of repeated multiplications from the previously defined matrices Findmincost and Findmaxcost. I had optimised this part using the fact that arrays are always stored Row Major Wise, so when we access a element,the compiler - alongwith the element accessed, also brings a few other elements from the row into the cache, so if we could make use of this fact in our computation, it would significantly reduce the time complexity of the code.So I have transposed B and then performed the dot product operation of a particular row of A with all the rows of transposed B and calculated the matrix product by that method. Iteration along rows is preferred over iteration over columns as row elements are stored next to each other due to row major ordering, so there will be fewer cache misses. I also went through Strassen's Matrix multiplication Algorithm for this section of the program.

I did not attempt the last part of the program as I did not about parallel programming and I was not able to effectively spawn threads in CPP. I thought of creating multiple threads and assigning some rows to each of them so that the dot products of rows can be computed in parallel.

III. Problem Statement Task 2.2

We were asked to code a game environment using OpenCV, where we could detect our face and track it and simulate collisions with it and a ball according to an animation which was provided to us.The ball would follow all the rules of physics.The game would end if the ball touches the ground.We were allowed to assume gravity to be absent and all the motion in the environment could be assumed to be non-accelerated.

IV. Initial Attempts

I had to learn OpenCV from scratch, so after getting comfortable with python, I started with Numpy and OpenCV, then initially I thought of using pygame as a way to create the environment, but as we were instructed not to use other libraries other than OpenCV, I had to improvise and I initially came up with an idea where I would feed a blank video and create an animation frame by frame with particular value of x and y velocities,but feeding and simultaneously running two videos frame by frame seemed to be a problem.So I finally thought of doing so on a blank (NumPy zeros)image

V. Final Approach for Face Detection Game

I have done this task in two ways, in the first way I have done as the problem statement instructed me to do, and in the second one,I have actually copied the face in the animation instead of a circular object, both differ only minutely. I would like to clarify that this program is hard-coded, I was not comfortable with the idea of OOP in python at the time of attempting this problem.With that note, lets dive into the code

```
capture=cv2.VideoCapture('demo1.mp4')
t0=process_time()
i=int(input('Enter the starting x
coordinate of the ball: '))
j=int(input('Enter the starting y
coordinate of the ball : '))
velox=int(input ('Enter velocity in
x direction as an integer : '))
veloy=int(input('Enter velocity in
y direction as an integer : '))
```

I made a video capture object passing the location of a video with face movements.I wanted to give the user to dictate the environment so I the velocity and the starting coordinates of the ball are inputted by the user.Now we read the video frame by frame

```
face_d=cv2.CascadeClassifier('haar_face2.xml')
while True:
     ret,img=capture.read()
     height,width=img.shape[:2]
     gray=cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
     img_tomask=np.zeros(img.shape[:2],
     dtype='uint8')
     gray=cv2.flip(gray,1)#use only when using
          #webcam otherwise comment it out
     cv2.circle(img_tomask,(i,j),30,(235,206,135),
     thickness=cv2.FILLED)
```

We use "haar face2" classifier to detect faces in the image.We capture the video frame by frame and convert it to grayscale and if we use webcam to feed the video, flip it along a vertical axis and create an blank image of the same dimensions to simulate the dimensions.We also create the ball using which we would play the game using the cv2.circle function of OpenCV.

```
faces=face_d.detectMultiScale
(gray,1.1,6,minSize=(20,20),
flags=cv2.FONT_HERSHEY_SIMPLEX)
     for (x,y,w,h) in faces:
          #e=rescale(gray[y:y+h,x:x+w],.5)
          img_tomask[y:y+h,x:x+w]=gray[y:y+h,x:x+w]
          cv2.imshow('game_env',img_tomask)
```

We detect the faces and store their coordinates in faces. Now in the blank image which we previously created, we

paste the detected face at the exact location where it was detected and display the image to the user as the initial game environment.

```
for (x,y,w,h) in faces:
        #e=rescale(gray[y:y+h,x:x+w],.5)
        #img_tomask[y:y+h,x:x+w]=gray[y:y+h,x:x+w]
        center=(x+w)//2 , (y+h)//2
        cv2.circle(img_tomask,center,
        (h+w)//6,(100),cv2.FILLED)
        cv2.imshow('game_env',img_tomask)
```

As I mentioned before, I tried two versions of problem, in this part I created a circle on the environment image at the location where the face was detected with the center to the middle of the face

```
    if (i < 30) :
          i=30
          velox=-velox
        if (i+30)> width:
                  i= width-30
          velox=-velox
        if j < 30 :
          j=30
          veloy=-veloy
        if(j+30)>height :
          x1= float((process_time()-t0)/60)
          print("Game Over, Time Elapsed
          in minutes is ",x1)
          quit()
```

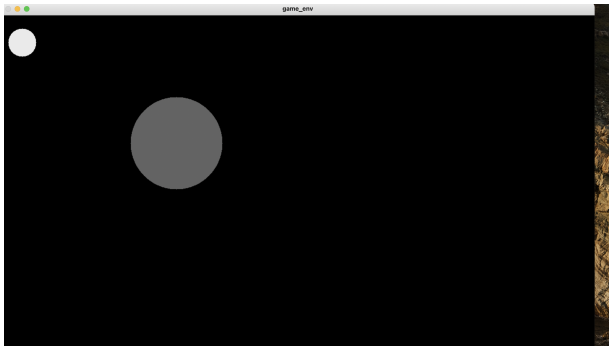
Now I have checked for boundary conditions,if the ball strikes any wall except the ground, it will reflect back following the usual laws of physics, but if it strikes the ground ,we exit a game and display the time for which the user lasted in the game.

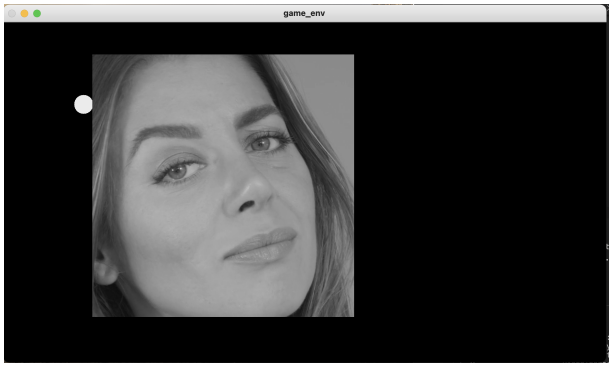
```
if distance(center,(i,j)) <= ((h+w)//6)+30:
          velox=-velox
          veloy=-veloy
        i+=velox
        j+=veloy
    if cv2.waitKey(1) & 0xFF == ord('d'):
        break
```

Now I check for collision between the ball and the face tracker. If distance between their centers is less than the sum of their radii, they have collided and I reverse the velocity of the ball, now I increment the coordinates of the ball using the velocity for the next iteration.In the other part too, I check for reflection similarly and I assume that will be self-explanatory.At last we check for the terminating condition which is if we enter d or the Waitkey time limit is crossed. Then we destroy all the windows where the video was displayed

VI. Results and Observation

The face detection model and both the versions of the game work efficiently but the detection is not sensitive,it is vulnerable to rapid facial movements which make it lag for a bit.It works fine against a plain background, otherwise it sometimes assume random patterns in the surroundings to be faces.





VII. Future Work

This task was really interesting and I could come up with so many versions of it in my mind. Like using our face to drive the ball through a maze and other cool stuff,I would like to try them in my leisure.I am also interested in facial recognition using dlibs which I hope to try my hands on in the near future.

VIII. Problem Statement Task 3.1

There are multiple steps in this problem, The only information we were provided was that we needed to decode the text hidden in the 'level1.png' provided to us. That would lead us further.I will discuss them in the final approach for my problem.

IX. Final Approach for Path planning puzzle

Firstly, I had to read the image 'Level1.png', then I converted it from BGR to RGB as it is the standard colour space. Upon inspection, I found out that all the pixels of the image 'level1.png' were identical. The top of the image stood out from the rest of the image, that hinted to me that only the top of the image could be encrypted, so I started extracting the pixels and converting them to characters through their ASCII characters. I was successful in decoding the first part of the puzzle.

```
img=cv2.cvtColor(cv2.imread('Level1.png'),
cv2.COLOR_BGR2RGB)
img1=cv2.cvtColor(cv2.imread('zucky_elon.png'),
cv2.COLOR_BGR2RGB)
a=(np.array(img)).tolist()
b=np.array(img)
t=0
for i in range (0,6):
   for j in range (0,177):
```

```
            a[i][j][0]=chr(a[i][j][0])
            print(a[i][j][0],end="")
for z in range (0 ,94):
          a[6][z][0]=chr(a[6][z][0])
          print(a[6][z][0],end="")
```

```
vishalbagaria@Vishals-MacBook-Pro vishalqq % /usr/local/bin/python3 /Users/vishalbagaria/Desktop/vishalqq/statement3/program3part1.py
Congrats on solving the first level of this task! You were able to figure out the ASCII text from the image, but wait!
 Your journey is not yet over. After the end of the text you will find a (200, 150, 3) coloured image. This image is a
 part of the bigger image called "zucky_elon.png". Find the top left coordinate (Image convention) from where this ima
ge was taken. The x coordinate represents the colour of a monochrome maze hidden in an image with coloured noise. Find
 the maze and solve the maze using any algothrim like dfs but better. Try comparing them and seeing how they perform l
ike A*, RRT, RRT* for example. Once the maze is solved you will see a word. This word is a password to a password prot
ected zip file which contains a png. Note that the password is case sensitive and all the aplhabets in the password wi
ll be capital letters This is your treasure. To open the treasure you need to convert the image in to an audio file in
 a simple way like you did for this ASCII text. Once converted, open the .mp3 file and enjoy your treasure, you deserv
ed it! A part of the image "zucky_elon.png" will begin immediatly after the colon. image=in2 :146
```

There were instructions to proceed further in the decoded text.We will go through that step by step. Firstly we were told that the same image 'level1.png' was infact part of another image. We were told the dimensions of the image and had to rearrange the first image to get a image, which was a part of another image.So first I designed a blank image of the mentioned dimensions and then filled it pixel by pixel.Luckily, I was successful in my first attempt and I found Zuck staring at me with a weird smile.

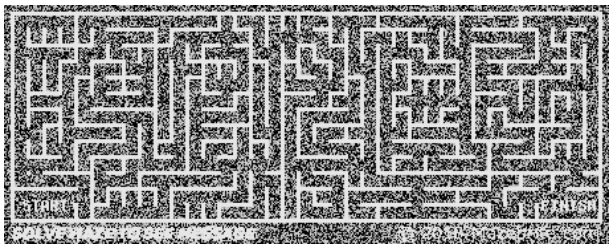
```
c=np.zeros((200,150,3),dtype='uint8')
for j in range (0,83):
       for k in range (0,3):
           c[0][j][k]=b[6][94+j][k]
for j in range (83,150):
        for k in range (0,3):
            c[0][j][k]=b[7][t][k]
        t+=1
for j in range (0,110):
        for k in range (0,3):
            c[1][j][k]=b[7][t][k]
        t+=1
t=110
q=1
for i in range (8,176):
    for j in range (0,177):
            if t==150:
                t=0
                q+=1
            for k in range (0,3):
                c[q][t][k]=b[i][j][k]
            t+=1
print(t)
h=Image.fromarray(c)
```

The next step was to superimpose this image on another image 'zucky elon.png',and to find the top left coordinate from where the image of zuck was taken from in 'zucky elon.png'.I tried manually superposing via two NumPy arrays but was repeatedly unsuccessful. At last I used the matchTemplate function to find superpose the image and find the top left coordinate which came out to be 230.

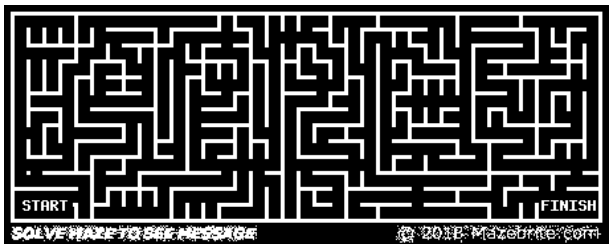
```
pq=cv2.cvtColor(cv2.imread('atbash12.png'),
cv2.COLOR_BGR2RGB)
h1=cv2.matchTemplate(img1,pq,
cv2.TM_SQDIFF_NORMED)
cv2.imshow('h1',h1)
min_val, max_val, min_loc,
max_loc = cv2.minMaxLoc(h1)
print(min_loc[0])
```

Now we were given a maze with coloured noise and were asked to first filter the noise out and find the maze and then solve it using various path planning algorithms. 230 was the value of the colourspace in which the maze was present in the picture, so we had to retrieve it. Instinctively splitting the image into b,g,r colour channels did the thing for me ,the blue channel clearly showed the maze and I was also able to retrieve the original image in the HSV color space using the inrange function in HSV color space.

```
lower=(230,0,0)
upper=(230,255,255)
mask=cv2.inRange(img,lower,upper)
o=cv2.bitwise_and(img,img,mask=mask)
b,g,r=cv2.split(img)
cv2.imshow('b',mask)
cv2.imshow('w',b)
cv2.imwrite('MAZE_D.png',mask)
```



I first took a printout of the image and solved it manually to find out the password to be 'APPLE'. I had read about RRT and RRT* as path planning algorithms to predct the trajectories of robots in a unknown environment. But in a maze with highly convoluted paths like this both RRT and RRT* did not prove to be useful at



all, cause randomness didn't go too well with restriction of walls of the maze, and it took me almost 0.4 million iterations to navigate to the path. Almost 8 in 10 moves had to be rejected because of the restrictions,We will discuss the code step by step now

```
import numpy as np
from cv2 import cv2 as cv2
import math
import random
im=cv2.imread('MAZE_D.png')
img=cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
img[140:155, 3:41] = 0
img[140:155, 400:441] = 0
start = [140, 41]
end = [140, 400]
```

This is the boiler plate code for the rrt algorithm, Ive stored the starting and ending points of the maze and blacked those portions out in the image.Grayscale is preferred as it needs only one value of color instead of BGR ,RGB,HSV etc.

```
def get_line(x1, y1, x2, y2):
    points = []
    issteep = abs(y2-y1) > abs(x2-x1)
    if issteep:
        x1, y1 = y1, x1
        x2, y2 = y2, x2
    rev = False
    if x1 > x2:
        x1, x2 = x2, x1
        y1, y2 = y2, y1
        rev = True
    deltax = x2 - x1
    deltay = abs(y2-y1)
    error = int(deltax / 2)
    y = y1
    ystep = None
    if y1 < y2:
        ystep = 1
    else:
        ystep = -1
    for x in range(x1, x2 + 1):
        if issteep:
            points.append((y, x))
        else:
            points.append((x, y))
        error -= deltay
        if error < 0:
            y += ystep
            error += deltax
    # Reverse the list if the
    #coordinates were reversed
    if rev:
        points.reverse()

    return points
```

The above method accepts two coordinates and returns the points lying between them ON the line formed by them.

```
def r_loc(img):
     x=random.randint(1,445)
     y=random.randint(1,159)
     if  img[y,x]!=0 :
         r_loc(img)
     return [y,x]
def dist_between_nodes(node1,node2):
#manhattan distance and the angle with the x axis
     y=abs(node1[0]-node2[0])
     x=abs(node1[1]-node2[1])
     if x!=0:
         ang=math.atan(y/x)
     else :
         ang=22/14
     return x+y,ang
```

The first method returns a point from our sample space after checking if it is black(permitted path) or not.The second method returns the manhattan distance and the angle made. by the line.

```
def nearest_node(node_list,node):
    maxloc=0
    minval=1000
    for i in range (len(node_list)-1,0,-1):
        if (dist_between_nodes(node_list[i],
        node)[0]) < minval:
             minval=(dist_between_nodes
             (node_list[i],
             node)[0])
             maxloc=i
    return node_list[maxloc]
```

This method finds the closest node from the list of permitted nodes which we have already traversed through(stored in nodelist).

```
def next_find(img,node1,node2,steo=3):
#node1 is the nearest node, node 2 is the
random node
    a=get_line(node1[1],node1[0],node2[1],node2[0])
    if len(a)<=steo:
         steo=len(a)-1
    if steo<2:
        return 0
    for i in range (0,steo):
        b=a[i]
        if img[b[1],b[0]]==255:
            return next_find(img,node1,node2,i-1)
    q=a[steo]
    return q[1],q[0]
```

This method accepts the nearest node and the random node picked and returns a point on that line at a distance of maximum three units, if the third unit is blocked the step size is decreased recursively and at one it returns zero implying no suitable point could be found

```
def planning(img,start,end,iter_len):
    nodelist=[start]
    nodetracker=[0]
    for i in range (0,iter_len):
         randomnode=r_loc(img)
         if img[randomnode[0],
         randomnode[1]]==255:
             continue
         nearnode=nearest_node(nodelist,
         randomnode)
         nextnode1=next_find(img,nearnode,
```

```
            randomnode)
            if(nextnode1==0):
                continue
            img[nextnode1[0],nextnode1[1]]=150
            nodelist.append(nextnode1)
            nodetracker.append
            (nodelist.index(nearnode))
            if nextnode1[0] in range (140,155) and
            nextnode1[1] in range (400,441):
                nodelist.append(end)
                nodetracker.append(nodelist.index
                (nearnode))
                return nodelist,nodetracker,img
    return None,None,img
```

This is the main function to implement the RRT algorithm,,nodelist contains all the nodes which we traverse through and nodetracker contains the index of the node which the latest node in the nodelist is nearest to.We find a randomnode, check for its validity and then find the node nearest to it and also find a node which we will insert(i.e, from the next find method) , then the node and the index of the nearest node is appended and for termination we check if the endpoint of the maze has been encountered and we return the required parameters.

```
nodelist,nodetracker,
img=planning(img,start,end,500000)
def pathp(nodelist,nodetracker,b,im):
     a=nodelist[b]
     if (a[0]==140 and a[1]==41):
         return im
     im[a[0],a[1]]=[0,255,255]
     q=nodetracker[b]
     b=q
     return pathp(nodelist,
     nodetracker,b,im)
imaged=pathp(nodelist,nodetracker,-1,im)
```

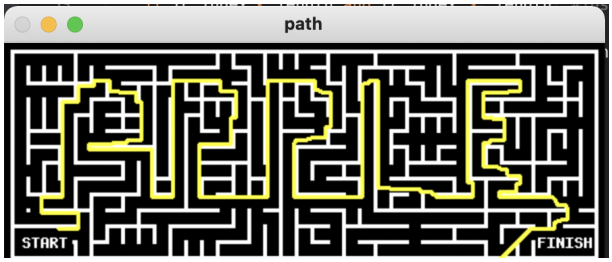
As I mentioned before, RRT and RRT* are not at all suggested for highly restricted paths such as mazes, I had to use .4 -.5 million iterations to reach the path.Now once we find the endpoint, we recursively backtrack and colour all the points till the start yellow, which would denote our path.

The following part is only required for RRT* and not RRT

The only difference between RRT AND RRT* is that after every addition of a node to the list, RRT* recursively updates the value of the nearest nodes of every element in the list.This reduces its space complexity but increases the time complexity as it is able to provide a more efficient path everytime,if the paths are thin and convoluted,RRT should be preferred as there wont be a significant difference between the RRT returned path and the optimal path.

```
def star(nodelist,nodetracker,c)
:#c initially would be the initial element
    a=nodelist
    b=a.pop(c)
    if b[0]==140 and b[1]==400:
        return nodelist,nodetracker
    c1=nearest_node(a,b)
    nodetracker[c]=nodelist.index(c1)
    return star(nodelist,nodetracker,c+1)
```

It recursively checks for the nearest node of every node that is in the list, after we have updated the list, and updates the list.



I have also tried Djikstra's and A* algorithm, which we will discuss now

Djikstra's and A* both are informed search algorithms in which the position of the starting node and ending node is known , They are way more efficient than uninformed search algorithms like Breadth and Depth First search.The only difference between Djikstra's and the A* algorithm is that ,A* makes use of the heuristic estimator which estimates the distance from the current node to the goal node.We will first go into the code for Djisktra's Algorithm

```
from cv2 import cv2 as cv2
import numpy as np
im=cv2.imread('MAZE_D.png')
img = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
img[140:155, 3:41] = 0
img[140:155, 400:441] = 150
start = [140, 41]
end = [140, 400]
```

This is the boiler plate code similar to the one for RRT algorithm

```
class node1:
    def __init__(self,abscissa,ordinate):
        self.x=abscissa
        self.y=ordinate
        self.d=float('inf') #distance from source
        self.parent_x=None
        self.parent_y=None
        self.computed=False
        self.ind__in_stack=None
```

I have defined a node class and initialised all its parameters through a parametrised constructor, the distance of any point from the source is initially assumed to infinite.

```
def transfer_up(stack, ind_):
     track_n=(ind_-1)//2
     if ind_ == 0:
         return stack
     if stack[ind_].d < stack[track_n].d:
         a=stack[ind_]
         stack[ind_]=stack[track_n]
         stack[track_n]=a
         stack[ind_].ind__in_stack=ind_
         stack[track_n].ind__in_stack=track_n
         stack = transfer_up(stack, track_n)
     return stack
```

The 'transfer up' function basically rearranges the stack on the basis of their distance from the source, in a way

similar to the binary search tree, it changes the branches of the tree based on its distance from the source

```
def transfer_down(stack, ind_):
    length=len(stack)
    lc_ind_=2*ind_+1
    rc_ind_=lc_ind_+1
    if lc_ind_ >= length:
        return stack
    if lc_ind_ < length and rc_ind_ >=
    length:
        if stack[ind_].d > stack[lc_ind_].d:
            stack[ind_], stack[lc_ind_]=
            stack[lc_ind_], stack[ind_]
            stack[ind_].ind__in_stack=ind_
            stack[lc_ind_].ind__in_stack=lc_ind_
            stack = transfer_down(stack, lc_ind_)
    else:
        small = lc_ind_
        if stack[lc_ind_].d > stack[rc_ind_].d:
            small = rc_ind_
        if stack[small].d < stack[ind_].d:
            stack[ind_],stack[small]=
            stack[small],stack[ind_]
            stack[ind_].ind__in_stack=ind_
            stack[small].ind__in_stack=small
            stack = transfer_down(stack, small)
    return stack
```

The above function analyses the left child of every node and rearranges it based on the distance of it from the start node, following the convention that the left lode is always less than the right node.

```
def get_livestack(mat,r,c):
    shape=mat.shape
    livestack=[]
    #ensure neighbors are within image boundaries
    if r > 0 and not mat[r-1][c].computed:
         livestack.append(mat[r-1][c])
    if r < shape[0] - 1 and
    not mat[r+1][c].computed:
            livestack.append(mat[r+1][c])
    if c > 0 and not mat[r][c-1].computed:
        livestack.append(mat[r][c-1])
    if c < shape[1] - 1 and
    not mat[r][c+1].computed:
            livestack.append(mat[r][c+1])
    return livestack
```

This method checks if the neighbors of the current node are within the boundaries and returns the neighbors of the current node in the order up,right,down,left

```
def next_node(img,src,dst):
    pq=[]
    source_x,source_y=src[0],src[1]
    dest_x,dest_y=dst[0],dst[1]
    row,imagecols=img.shape[0],img.shape[1]
    matrix = np.full((row, imagecols), None)
    #access by matrix[row][col]
    for r in range(row):
        for c in range(imagecols):
            matrix[r][c]=node1(c,r)
            matrix[r][c].ind__in_stack=len(pq)
            pq.append(matrix[r][c])
    matrix[source_y][source_x].d=0
    pq=transfer_up(pq, matrix[source_y]
    [source_x].ind__in_stack)
    while len(pq):
        u=pq[0]
        u.computed=True
        pq[0]=pq[-1]
        pq[0].ind__in_stack=0
        pq.pop()
        pq=transfer_down(pq,0)
        livestack = get_livestack(matrix,u.y,u.x)
        for v in livestack:
            dist=get_distance
            (img,(u.y,u.x),(v.y,v.x))
            if u.d + dist < v.d:
                v.d = u.d+dist
                v.parent_x=u.x
                v.parent_y=u.y
                idx=v.ind__in_stack
                pq=transfer_down(pq,idx)
                pq=transfer_up(pq,idx)
    path=[]
    iter_v=matrix[dest_x][dest_y]
    path.append((dest_x,dest_y))
    while(iter_v.y!=source_y or
    iter_v.x!=source_x):
        path.append((iter_v.x,iter_v.y))
        iter_v=matrix[iter_v.parent_y]
        [iter_v.parent_x]

    path.append((source_x,source_y))
    return path
```

This method is the most important method in Djikstra's Algorithm.We set the coordinates of source and destination and create a numpy matrix of rows and columns of the same dimensions as the maze image.Now we append each element from the image to the matrix and set the distance of source to be zero. Now we use the transfer up function to arrange the nodes we need in the order of their distance from the source.i Now run a while loop through the length of the matrix and change the first element of the matrix to be the last one and find its neighbors and find their euclidean distance (of color space) and check if its less than the previously computed/already stored value of distance from source in that loop and modify it accordingly,first calling the transfer down function and then calling the transfer up function, and then keep on appending the path from the matrix to the path list and return the path each time.
The A* algorithm is similar to it so I wont add A* here, We will now move to Depth First Search, which is one of the less efficient methods of path planning, as it is uninformed and has to search the entire space recursively, until it reaches the target.

```
def neighbors(img,node):
     up=node[0]-1,node[1]
     right=node[0],node[1]+1
     down=node[0]+1,node[1]
     left=node[0],node[1]-1
     if img[up]==0 and valid(up) :
         img[up]=100
         return img,up
     elif img[right]==0 and valid(right):
         img[right]=100
         return img,right
     elif img[down]==0 and valid(down):
         img[down]=100
         return img,down
     elif img[left]==0 and valid(left):
         img[left]=100
         return img,left
     return img,None
def depth(img,stack):
    a,next_node=neighbors(img,stack[-1])
    if next_node == None:
```

```
            stack.pop()
            return depth(a,stack)
        stack.append(next_node)
        b=stack[-1]
        if (b[0]==140 and b[1]==400):
            return stack
        return depth(a,stack)
```

I have a similar boiler plate code for this too, which I'll skip. DFS basically checks if the neighbors of the node are permitted or not and if they are permitted,it traverses to the permitted node(up,right,down,left) otherwise it backtracks to the previous node.
Since I was using a M1 Macbook for my tasks, I could not install ffmpeg, I even tried building it from the package but there was some access fault due to which I could not complete the last part of the task, still I have tried the code for it and it has been provided alongwith the other codes.

X. Problem Statement Task 4.1

This problem statement dealt with localization and trajectory prediction algorithms like Kalman Filter, EKF, UKF,Grid localisation and Monte-Carlo particle localisation. The task was to eliminate noise from the trajectory of a drone given through numerous ground stations assuming the first ground station to be situated at the origin. It was provided that the noise in the data was gaussian in nature.

XI. INITIAL ATTEMPTS

I first tried to familiarise myself with Gaussian noise and methods for its filtering and then I tried to learn about these localisation algorithms, I found a magnificent book - Probabilistic Robotics by Sebastian Thrun. It had developed the idea of Kalman Filter from scratch and had extended the idea of localisation to particle-filter based localization methods like Monte-Carlo etc. I spent almost 4 days completely going through those parts and developing the intuition for Kalman Filter.Then I tried various approaches, but I came across a few logical Roadblocks-which I would want to discuss during the demonstration. Firstly ,while incorporating velocity in the Kalman Filter, I came across numerous instances where velocity in a particular direction(x,y,z) as reported by a particular station was anti-parallel to the velocity reported by other stations in the same direction.I also tried to use few of the values provided to us per coordinate per station and generate a polynomial function and optimise the other readings based on that, but that seemed quite off track and could not be achieved in the limited time frame with the enormous pressure of academic mounting.

XII. FINAL APPROACH FOR THE LOCALISATION OF DRONE

I first analysed the coordinates given to us in a particular direction using matplotlib, it was evident that a good amount of noise was infused with the readings.To eliminate it, a polynomial trajectory would have been preferred, so i tried to infuse acceleration as the state control matrix and made velocity and position as the state matrices, I then assumed the time gap between the readings to be 1 seconds and then calculated the velocity by subtracting the previous coordinate from the latest coordinate from every six stations in a particular direction and averaged it, multiple measurements of the same quantity work to minimise the error in one particular quantity, initially in the first few readings, I found a few discrepancies but gradually as readings progressed the noise was consistently reducing as was evident from matplotlib. I made a method 'firstp' for performing this operation-It has been discussed below.

```
def firstp(k,q):
    a=np.zeros(3)
    for i in range (0,3):
        s=0.0
        for j in range (i,18,3):
                    s+=k[q+1][j]-k[q][j]
        a[i]=s/6.0
    return a
```

This method returns a 3-element array which has the velocities of each coordinates averaged from each of the six stations. Now I created the coefficient matrices A and B which would help in the time evolution of the state.Now I created the functions to calculate the error matrix and the Kalman Gain matrix, I've assumed the error to be 10 percent of the readings , as the last digit from the measurements can be considered uncertain, and the corresponding standard deviation value to be .01

```
a_coeff=np.zeros((6,6))
b_coeff=np.zeros((6,3))
for i in range (0,3):
    a_coeff[i][i]=1
    a_coeff[i+3][i+3]=1
    a_coeff[i][i+3]=1
    b_coeff[i][i]=.5
    b_coeff[i+3][i]=1
""" assuming the time gap/lag to be one second """
def error(matrix):
    a=np.zeros((6,6))
    for i in range (0,6):
        a[i][i]=(matrix[i][0]/10)**2
    return a
def kalman_gain(predicted,error):
    q=predicted + error
    a=np.zeros((6,6))
    for i in range (0,6):
        a[i][i]=predicted[i][i]/q[i][i]
    return a
```

I have created two numpy arrays to store the values of position and velocity as updated by the Kalman Filter.These 2 arrays will be recursively filled by the two optimised values. Now I have created method to arrange the states(position and velocity) in a 6x1 column matrix, in which the first three values correspond to position while the last three correspond to velocity.

```
predicted_values=np.zeros((9997,3))
predicted_values[0]=j[0,0:3]
predicted_velocity=np.zeros((9998,3))
predicted_velocity[0]=firstp(j,0)
def format(predicted,index,predicted_velocity):
```

```
    a=np.zeros((6,1))
    for i in range (0,3):
        a[i][0]=predicted[index][i]
        a[i+3][0]=predicted_velocity[index][i]
    return a
```

Now I have created a function to calculate the acceleration which is our control state matrix and will be used to update the state matrices. I have calculated the acceleration by calculating the velocity for traversing to the next position and the velocity with which the drone traversed to this position and I subtracted the latter from the first.
I have also created a function to update the last state to the latest state, where the update equation from the Kalman Filter Theory was used, the update equation basically involves the time evolution of the states (position and velocity) using acceleration as the state control matrix. The coefficient matrices acoeff and bcoeff are pre-multiplied to the state matrix and the control matrix respectively to get the time-evolved latter state,which we optimise using Kalman Gain. Kalman Gain is similar to the exploration vs exploitation trade-off, it basically allots weight to the measured readings and the predicted values and the fused readings are optimised on the basis of the weights alloted by the Kalman filter. We had been asked to assume the error covariance matrix to be 0 and the noise covariance matrix Q to be of an appropriate value so as to reduce the noise to minimum, I found that assuming Q to be a null matrix drastically reduced the noise while alloting any other value to Q (preferably a Diagonal matrix) led to a gradual reduction in noise, which is clearly demonstrated by the plots below for Data1csv First station,X coordinate.

```
def acceleration(array,index):
    a=np.zeros(3)
    b=np.zeros((3,1))
    a=firstp(array,index)-firstp(array,index-1)
    for i in range(0,3):
        b[i][0]=a[i]
    return b
def latest(a_coeff,b_coeff,array,index,
predicted_values,predicted_velocity):
     statec=acceleration(array,index+1)
     latest_state=np.matmul(b_coeff,statec)
     +np.matmul(a_coeff,format(predicted_values,
     index,predicted_velocity))
     return latest_state
```

Now the most important method is discussed - 'Sensor-fusion' which recursively updates the states according to the Kalman Gain matrix.

```
 def sensor_fusion
 (array,p_matrix,index,a_coeff,b_coeff,
 predicted_values,predicted_velocity):
     measurement=np.zeros((6,1))
     if index ==9997:
         return predicted_values
     for i in range (0,3):
         measurement[i][0]=array[index][i]
     measurement[3:6,0]=firstp(array,index)
     errorf=error(measurement)
     kalman_factor=kalman_gain(p_matrix,errorf)
     latest_state=latest
     (a_coeff,b_coeff,array,index-1,
     predicted_values,predicted_velocity)
     final=latest_state+np.matmul
     (kalman_factor,(measurement-latest_state))
     p_matrix=np.matmul
     ((identity(6)-kalman_factor),p_matrix)
     p_matrix=np.matmul
     (a_coeff,p_matrix)
     p_matrix=np.matmul
     (p_matrix,np.transpose(a_coeff))+q_mat
     for i in range (0,3):
         predicted_values
         [index][i]=final[i]
         predicted_velocity
         [index][i]=final[i+3]
     return sensor_fusion
     (array,p_matrix,index+1,a_coeff,b_coeff,
     predicted_values,predicted_velocity)
```

The terminating condition of this function is if we checked through all the data i.e. 9998 rows then I returned the matrix with predicted values through Kalman Filter. If the terminating condition was not met, I first made the measurement matrix, which contains the position in the three directions as the first three coordinates and the velocity in the three directions as the remaining coordinates, and then I also time evolved the last state which was predicted by the Kalman Filter to the present state and updated the predicted values and predicted velocity matrices, after calculating the Kalman optimised readings for the current state. And then updated the values of the covariance matrices and the state matrices.Then the function has been recursively called and we proceed to the next iteration incrementing the index by 1 and passing all the incremented values which were updated in the last iteration. The readings are almost concordant with the readings of the first ground station (assumed to be situated at the origin), the noise was effectively eliminated, particularly by averaging the velocity values reported from different ground stations as velocity and acceleration are independent of the choice of reference frame and also by using the Kalman Filter to fuse the measured and predicted values.
I have used matplotlib to plot the readings.

XIII. Results and Observation

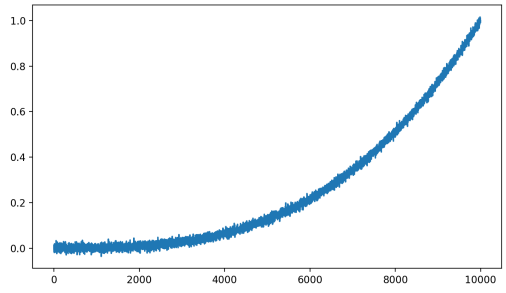


Fig. 1. Raw Data

As it is evident from the diagram that when the Q noise covariance matrix is given to be zero, the noise is

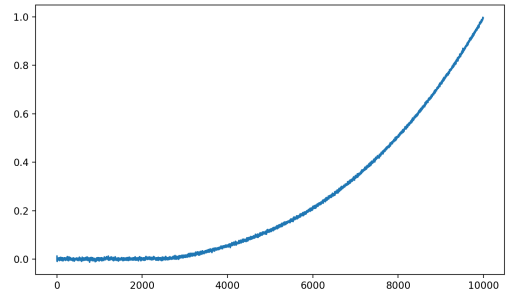


Fig. 2. Standard Dev (Noise) =0

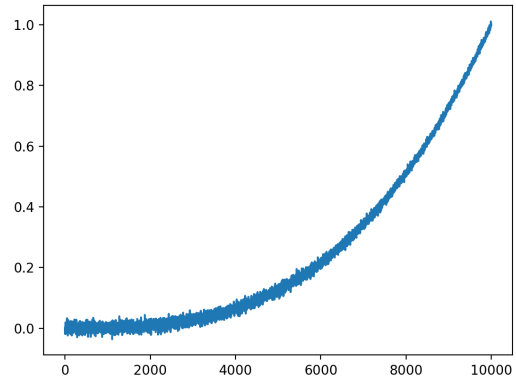


Fig. 3. Standard Dev (Noise) =0.01

considerably reduced and as we increment the Q-matrix , the noise does not reduce initially ,but gradually around the 5000 mark, the curve somewhat becomes similar to the curve with 0 standard deviation of noise.

XIV.

PART 2: DATA INFESTED WITH ERRONEOUS READINGS, DUE TO ISSUES WITH THE SERVER.

In this part, we were provided with the coordinates like the last part but the coordinates contained a lot of values which did not make sense, they were completely out of bound , this was also evident on plotting the coordinates as shown ,with the help of matplotlib, as shown below. The comb-like structure of the graph -i.e. the sharp peaks signified the erroneous readings and and we had to recover the base-line curve which ran from 0-1 and then remove noise from it using Kalman Filter. I used Bienaymé–Chebyshev inequality to remove all these sharp peaks from the data, the inequality states that almost 96 percent of the data lies within 2 standard deviations of the mean and 99.7 percent data lies within 3 standard deviations of the data. Optimally , most of the readings except the erroneus ones would lie withing 2 standard deviations of the data, I have used a function standarddev to find all the values which do not lie in the specified range. I t has been discussed below.

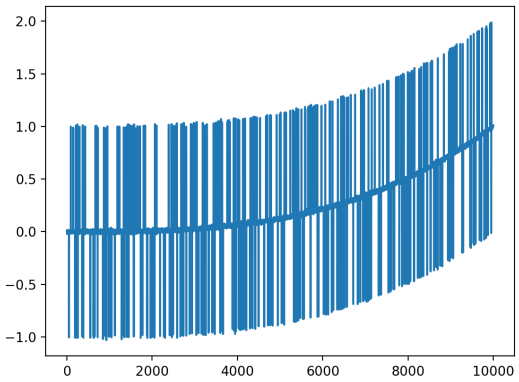


Fig. 4. Raw Data for Part 2, station 1 (x coordinate)

```
def standard_dev(array,index):
    for i in range (0,20):
        a=np.zeros(500)
        c=0
        for j in range (i*500,(i+1)*500-1):
            a[c]=array[j][index]
            c+=1
        q=np.std(a)
        w=np.mean(a)
        a1=[]
        for j in range (i*500,(i+1)*500-1):
            if array[j][index] < w-(2*q)
            or array[j][index]>w+(2*q):
                a1.append(j)
        while a1:
            array[a1[-1]][index]=0
            a1.pop()
    return array
```

Since the drone is moving, i.e. its coordinates are changing with respect to time, I had to partition all the readings into small intervals over which the readings didnt change much, I could not take a very small interval as it would significantly increase the time consumed by the code, so I took a space of 500 observations per iteration and stored them in a separate array, the variable index is the index of the column we are currently working on, and then I calculated the mean and standard deviation of the data. Now, I have checked if the element in the array lies within two standard deviations of the mean or not, if they lie outside the range, I've made them zero and after processing the entire 9999 readings, I've returned the optimised array.

```
for i in range (0,18):
  j=standard_dev(j,i)
```

Now I have removed the erroneous readings from all the 18 columns and now we can proceed in the same manner as we did for the first case but a few measurements here would be zero, which need not be taken into consideration, which otherwise would produce erroneous values. So I have made a few changes in the firstp method and sensor fusion method, which are shown below.

```
def firstp(k,q):
    a=np.zeros(3)
    for i in range (0,3):
        s=0.0
        x=0
        for j in range (i,18,3):
            if k[q+1][j] ==0 or k[q][j]==0:
                x+=1
                continue
            s+=k[q+1][j]-k[q][j]
        a[i]=s/(6.0-x)
    return a
```

I have checked if the values of the position (with the help of which the current velocity would be calculated) are 0. If they are zero, they would not be used for the calculation of velocity, as demonstrated in the above code .

```
def sensor_fusion(array,p_matrix,index,
a_coeff,b_coeff,predicted_values,
predicted_velocity):
     #array=standard_dev(array,0)
     measurement=np.zeros((6,1))
     latest_state=latest(a_coeff,
     b_coeff,array,index-1,predicted_values,
     predicted_velocity)
     if index ==9997:
         return predicted_values
     for i in range (0,3):
         if array[index][i]==0:
            measurement[i][0]=latest_state[i][0]
         else:
            measurement[i][0]=array[index][i]
     measurement[3:6,0]=firstp(array,index)
     errorf=error(measurement)
     kalman_factor=kalman_gain
     (p_matrix,errorf)
     final=latest_state+np.matmul
     (kalman_factor,
     (measurement-latest_state))
     p_matrix=np.matmul((identity(6)
     -kalman_factor),
     p_matrix)
     p_matrix=np.matmul(a_coeff,p_matrix)
     p_matrix=np.matmul(p_matrix,
     np.transpose(a_coeff))
     for i in range (0,3):
         predicted_values[index][i]=final[i]
         predicted_velocity[index][i]=final[i+3]
     return sensor_fusion(array,
     p_matrix,index+1,a_coeff,
     b_coeff,predicted_values,predicted_velocity)
```

I have checked if the any of the positional states is zero, if any one of the three comes out to be zero, we go with only the predicted values for the iteration and not the measured values as they were erroneous.

XV. Results and Observation

On first removing the erroneous values and replacing them by 0, we get a graph where all the erooneous values point down to zero and we can see from graph that the erroneous values are distributed almost at regular intervals There are a few corners and edges in the final graph, which should have been removed and smoothened, but I could not figure out a legitimate way to filter them out.

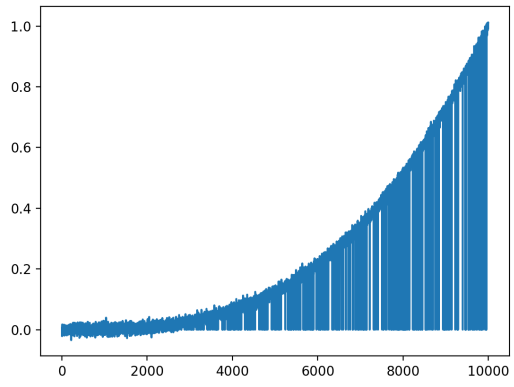


Fig. 5. After removing the erroneous values, station 1 (x coordinate)

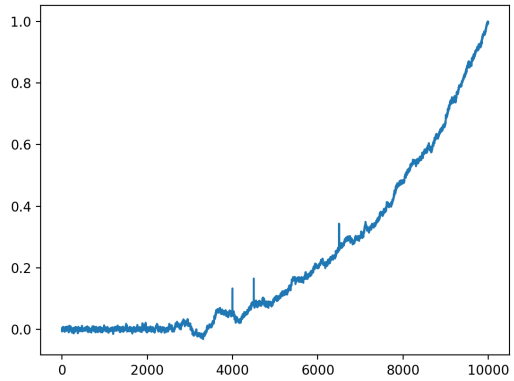


Fig. 6. Final optimised values, station 1 (x coordinate)

XVI. Future Work

I read extensively about the Extended and Unscented versions of the Kalman Filter which are non-linear recursive localisation algorithms, I also read about grid based localisation methods like Grid Localisation and Monte Carlo localisation , at first I thought of implementing Monte Carlo localisation to the problem statement, but the mathematics aspect of it was not completely clear to me,so I had to let that idea go, but I would love to improvise on this and design a Monte-Carlo localizer with the help of seniors.

XVII. Problem Statement Task 4.2

We were asked to code an agent to play a game of TicTacToe using the Minimax Algorithm in the first part of the Problem, in the Second part of the problem, we were asked to code an agent to play 3D-TicTacToe using the TicTacToe environment provided to us using the Gym Library. Since the 3-D TicTacToe would have 3^{27} states and that would tremendously increase the complexity of the program, raw implementation of the MiniMax algorithm is not suggestive for this case. We

can use alpha-beta pruning or alternatively we can use Reinforcement Learning Algorithms like Q-Value Learning or Genetic Algorithms. Minimax Algorithm is a backtracking, recursive algorithm for finding the optimal move for the player.

XVIII. INITIAL ATTEMPTS

After reading the problem statement, I first tried to figure out the maths behind a 3-D TicTacToe game, upon reading I found that a win is guaranteed for the first player if he plays optimally (i.e., chooses the best possible move for every turn he gets, unlike in 2-D TicTacToe where both players playing optimally leads to a tie. I, then tried to find out the total number of possible winning combinations in 3-D TicTacToe, which came out to be 49. Then I checked the Env Code Base provided in the attached GitHub Repository and looked up for the Minimax Algorithm on the Internet. I tried to solve the 2D part first and tried playing a few games online to figure out the way the AI was working. I came to a few conclusions which would optimise the function at least for the 2-D case. Then I tried to learn Reinforcement Learning methods, I learnt about the Q-Value search Algorithm. I looked at a few research papers on the application of various methods to code an AI to play 3D TicTacToe. The 2-D case was on the simpler side, with a limited sample space and a less number of winning combinations. Initially,I tried a brute force non-recursive approach to extensively search the sample space and deduce the best possible move. Then, I tried implementing recursion in order to reduce the code size a bit.

XIX. FINAL APPROACH FOR 2-D TICTACTOE

In the first part of the problem, I have made a separate class for the 2-D TicTacToe since the environment provided to us was suitable for 3-D TicTacToe. To work on the positions I have used key-value pairs as shown below, also I have assigned 'X' to the player and 'O' to computer, to reduce confusions. This cannot be changed

```
import random
human = 'X'
# Player is assigned x while computer is assigned 0
computer = 'O'
print("Positions are as follows,
enter an unoccupied digit to proceed and an
empty board has been printed.
\n 1, 2, 3 \n 4, 5, 6 \n 7, 8, 9 \n")
tictac = {1: ' ', 2: ' ', 3: ' ',
          4: ' ', 5: ' ', 6: ' ',
          7: ' ', 8: ' ', 9: ' '}
```

The most important part of this problem is certainly the MiniMax function, which is a static evaluator which returns the optimal move for the AI with the assumption that both the players are playing optimally. It ideally should accept three parameters - the environment , the action and the depth of the search but due to the small size of the sample space we are working with - I have carried about Full Depth Traversal to find out the best possible move for the AI, it must be noted that the depth can be reduced along with the optional introduction of Alpha-Beta Pruning algorithm to speed up the program , but that might make the AI vulnerable. Since MiniMax is a recursive algorithm, we first need to specify the terminating condition, that is, when the game ends,there are three cases for it :- AI wins, Human wins, Draw.

```
    if (team_vict(computer)):
        return 1
    elif (team_vict(human)):
        return -1
    elif (stalemate()):
        return 0
```

Now, the function checks if it has to maximise or minimise, if it has to maximise, we set a very small value in the maximum variable which will store the best move possible and then we recursively check for the best possible move by running minimax (now with the minimiser or maximiser=False) over all the empty spaces present, the iteration which returns the maximum value is the one which is used as the next move of the AI. If the current move was of the AI, the next move will be Human's which will be the minimising move, as human will try to minimise the rewards of AI as much as possible. So in the minimising condition, the code base will be almost similar to the maximising one except we will iteratively search for the minimum value during the recursion.

```
if (maximiser):
      extreme = -800
      for key in tictac.keys():
          if (empty_space(key)==True):
            tictac[key] = computer
            score=minimax(tictac,False)
            tictac[key] = ' '
            if (score > extreme):
              extreme = score
      return extreme
else:
       extreme = 800
       for key in tictac.keys():
           if (empty_space(key) == True):
               tictac[key] = human
               score = minimax(tictac,True)
               tictac[key] = ' '
               if (score < extreme):
                   extreme = score
       return extreme
```

Next, I have defined an array of functions which would help us simplify the logic of the program. I have defined emptyspace to check if the inputted space is empty or not, stalemate to check if the game ends in a draw- which occurs when all the spaces in the board are filled.

```
def empty_space(position):
    if tictac[position] == ' ':
        return True
    else:
        return False
def stalemate():
    for key in tictac.keys():
        if (tictac[key] == ' '):
            return False
    return True
```

Now we define a nextmove function which takes the team and the position as arguments, it returns a new board with the current action applied (team and position) if the position is empty and also checks for termination (Human win, AI win, Draw) and exits the program, and if the position is occupied we raise an error and prompt the user to enter in a new position as the input position is already filled.

```
def nextmove(letter, position):
    if empty_space(position):
        tictac[position] = letter
        board(tictac)
        if (stalemate()):
            print("The game ends in a tie")
            exit()
        if win_term():
            if letter == 'O':
                print("Computer wins!")
                exit()
            else:
                print("Human wins!")
                exit()
```

Next we define functions to check for the winning conditions in the game environment. A TicTacToe game can be won by occupying 3 consecutive positions along a row, a column or a diagonal. The following functions check for the above specified conditions.

```
def win_term():
    if (tictac[1] == tictac[2]
    and tictac[1] == tictac[3]
    and tictac[1] != ' '):
        return True
    elif (tictac[4] == tictac[5]
    and tictac[4] == tictac[6]
    and tictac[4] != ' '):
        return True
    elif (tictac[7] == tictac[8]
    and tictac[7] == tictac[9]
    and tictac[7] != ' '):
        return True
    elif (tictac[1] == tictac[4]
    and tictac[1] == tictac[7]
    and tictac[1] != ' '):
        return True
    elif (tictac[2] == tictac[5]
    and tictac[2] == tictac[8]
    and tictac[2] != ' '):
        return True
    elif (tictac[3] == tictac[6]
    and tictac[3] == tictac[9]
    and tictac[3] != ' '):
        return True
    elif (tictac[1] == tictac[5]
    and tictac[1] == tictac[9]
    and tictac[1] != ' '):
        return True
    elif (tictac[7] == tictac[5]
    and tictac[7] == tictac[3]
    and tictac[7] != ' '):
        return True
    else:
        return False
def team_vict(mark):
    if tictac[1] == tictac[2]
    and tictac[1] == tictac[3]
    and tictac[1] == mark:
        return True
    elif (tictac[4] == tictac[5]
    and tictac[4] == tictac[6]
    and tictac[4] == mark):
        return True
    elif (tictac[7] == tictac[8]
    and tictac[7] == tictac[9]
    and tictac[7] == mark):
        return True
    elif (tictac[1] == tictac[4]
    and tictac[1] == tictac[7]
    and tictac[1] == mark):
        return True
    elif (tictac[2] == tictac[5]
    and tictac[2] == tictac[8]
    and tictac[2] == mark):
        return True
    elif (tictac[3] == tictac[6]
    and tictac[3] == tictac[9]
    and tictac[3] == mark):
        return True
    elif (tictac[1] == tictac[5]
    and tictac[1] == tictac[9]
    and tictac[1] == mark):
        return True
    elif (tictac[7] == tictac[5]
    and tictac[7] == tictac[3]
    and tictac[7] == mark):
        return True
    else:
        return False
```

The next functions are to process the moves played by Human and Computer.For the Computer's move we call the method minimax and find the best move possible and play that move.I have also incorporated a toss feature where if the human wins the toss, he gets to move first otherwise AI moves first.

```
def human_():
 position=int(input("Enter the position for 'X':"))
    nextmove(human, position)
def computer_():
    extreme = -800
    bestMove = 0
    for key in tictac.keys():
        if (tictac[key] == ' '):
            tictac[key] = computer
            score = minimax(tictac,  False)
            tictac[key] = ' '
            if (score > extreme):
                bestMove = key
                extreme = score
    nextmove(computer, bestMove)

    o=int(input("Lets have a toss to see who gets
    to move first, select either 0 or 1:"))
p=random.randint(0,2)
if(o==p):
    print ("You have won the toss, you go first:")
     while not win_term():
         human_()
         computer_()
else:
    print("You have lost the toss,AI goes first:")
     while not win_term():
        computer_()
        human_()
```

I have also tried a Q-Value approach to the problem, in the next few lines I have attached the method to train the AI to play TicTacToe, I will attach the entire code in my GitHub Repository

```
def trainers(games, model, model_2):
  global x_train
```

```
xt = 0
ot = 0
dt = 0
states = []
q_values = []
statenext = []
qval2 = []
for game in games:
  rewcalc = get_outcome(game[len(game) - 1])
  if rewcalc == -1:
    ot += 1
  elif rewcalc == 1:
    xt += 1
  else:
    dt += 1
  # print('reward =', rewcalc)

  for i in range(0, len(game) - 1):
    if i % 2 == 0:
      for j in range(0, 9):
        if not game[i][j] == game[i + 1][j]:
          reward_vector = np.zeros(9)
          reward_vector[j] = rewcalc*
          (reward_dep**(math.floor
          ((len(game) - i) / 2) - 1))
          states.append(game[i].copy())
          q_values.append(reward_vector.copy())
    else:
      for j in range(0, 9):
        if not game[i][j] == game[i + 1][j]:
          reward_vector = np.zeros(9)
          reward_vector[j] = -1*rewcalc*
          (reward_dep**(math.floor
          ((len(game) - i) / 2) - 1))
          statenext.append(game[i].copy())
          qval2.append(reward_vector.copy())

if x_train:
  zip_ = list(zip(states, q_values))
  random.shuffle(zip_)
  states, q_values = zip(*zip_)
  new_states = []
  for state in states:
    new_states.append(ohencode(state))


  model.fit(np.asarray(new_states),
  np.asarray(q_values), epochs=4,
  batch_size=len(q_values), verbose=1)
  model.save('tic_tac_toe.h5')
  del model
  model = load_model('tic_tac_toe.h5')
  print(xt/20, ot/20, dt/20)
else:
  zip = list(zip(statenext, qval2))
  random.shuffle(zip_)
  statenext, qval2 = zip(*zip_)
  new_states = []
  for state in statenext:
    new_states.append(ohencode(state))
  model_2.fit(np.asarray(new_states),
  np.asarray(qval2)
  , epochs=4, batch_size=len(qval2), verbose=1)
  model_2.save('tic_tac_toe_2.h5')
  del model_2
  model_2 = load_model('tic_tac_toe_2.h5')
  print(xt/20, ot/20, dt/20)

x_train = not x_train
```

XX. FINAL APPROACH FOR 3-D TICTACTOE

We were already provided with the environment for 3-D TicTacToe and the HumanAgent class was also provided to us, we just had to code an AI agent and call it alternatively with HumanAgent class to enable the AI to play the game,I also initially tried the Reinforcement Learning approach but it seemed overwhelming so I switched to a basic approach in which I assigned a value to every n-sequence the AI had and it was less than the opponents except when the AI had 2 in a row already, it was to encourage the AI to stop the position rather than prospect an alternative move for itself. The weight assigned to a n sequence is always more than a n-1 sequence possible so that the AI always looks out for the better move. I stored all the positions of the board in a linear array while doing computations to reduce confusion.Then I added all the values to a 3x3x3 matrix and then transposed it and found out the empty position which had the highest weight and the AI occupied that place in the environment. Playing with the assigned weights showed that the difficulty of the game from human's perspective can be reduced by decreasing the difference between the weights and can be correspondingly increased by increasing the weights. In most cases, AI took not more than 4 turns to come out as a winner.

```
class AIAgent(object):
    def __init__(self, mark):
        self.mark = mark
    def act(self,ava_actions,reward):
        """
     convert reward to a linear array in the form
     of bcr
        """
        l=np.zeros(27,dtype='uint8')
        o=0
        reward=np.transpose(reward,[0,1,2])
        for i in range (0,3):
            for j in range (0,3):
                for k in range (0,3):
                    l[o]=reward[i][j][k]
                    o+=1
        reward=l
        rew=np.zeros(27,dtype='uint8')
```

The code for assigning weights is very long and repetitive so I have only attached the columnwise assignment

```
for i in range (0,27,3):#checking column wise

            if (reward[i]==reward[i+1]
            and reward[i+2] ==0 and
            reward[i]!=0) :
                if reward[i]=='o':
                    rew [i+2] += 98
                else :
                    rew [i+2] += 196
            elif (reward[i]==reward[i+2]
            and reward[i+1]==0
            and reward[i]!=0) :
                if reward[i]=='o':
                    rew [i+1]+=98
                else :
                    rew[i+1]+=196
            elif (reward[i+1]==reward[i+2]
            and reward[i] ==0
            and reward[i+2]!=0) :
                if reward[i+1]=='o':
                    rew [i]+=98
                else :
```

```
                rew[i]+=196
            elif (reward[i+1]==reward[i+2]
            and reward[i+1]==0
            and reward [i]!=0):
                if reward [i]=='o':
                    rew[i+1]+=50
                    rew[i+2]+=50
                else:
                    rew[i+1]+=49
                    rew[i+2]+=49
            elif (reward[i+1]==reward[i]
            and reward[i+1]==0
            and reward [i+2]!=0):
                if reward [i+2]=='o':
                    rew[i+1]+=50
                    rew[i]+=50
                else:
                    rew[i+1]+=49
                    rew[i]+=49
            elif (reward[i]==reward[i+2]
            and reward[i]==0
            and reward [i+1]!=0):
                if reward [i+1]=='o':
                    rew[i]+=50
                    rew[i+2]+=50
                else:
                    rew[i]+=49
                    rew[i+2]+=49
            else :
                for j in range (i,i+3):
                    rew[j]+=1
```

In the both the methods 'act', (HumanAgent and AIAgent), I've added an extra argument reward- which contains the current state of the world which would be required by the AI for static evaluation, then I have converted the linear array of weights in to 3D array and find the position with the most weight and make it AI's next move

```
matrix=np.zeros((3,3,3),dtype='uint8')
        for i in range (0,3):
            for j in range (0,3):
             for k in range (0,3):
                matrix[i][j][k]=rew[o]
                o+=1
        y1=np.transpose(matrix,(0,2,1))
        ind=np.unravel_index
        (np.argmax(y1,axis=None),y1.shape)
        l=("{}{}{}".format(ind[0], ind[1], ind[2]))
        return self.mark+l
```

XXI. Results and Observation

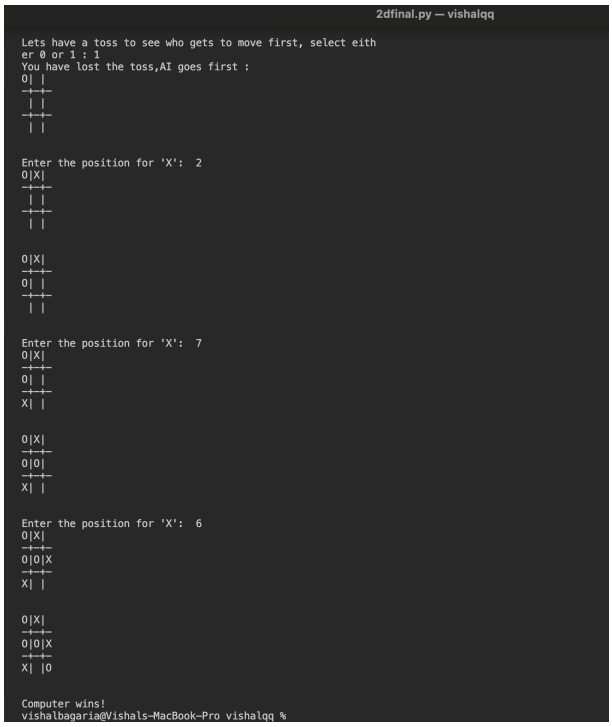
For the 2-D TicTacToe, under optimal playing conditions the MiniMax AI never loses, the worst outcome is draw. I have tabulated the moves of human below, provided AI plays first, the AI always takes the first place on the grid.

AI PLAYING FIRST OUTCOME DISTRIBUTION

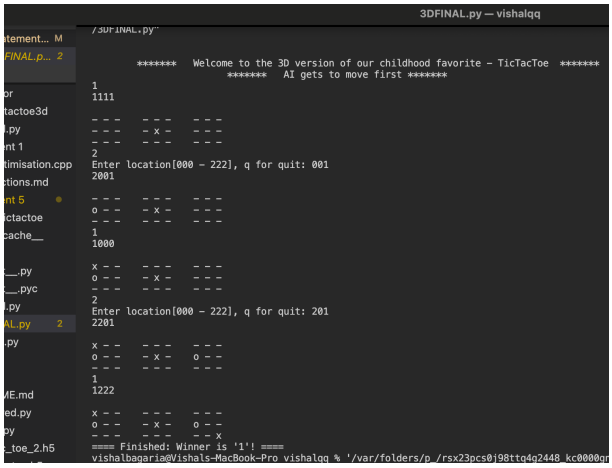
HUMAN'S MOVE	OUTCOME
2,3,4,6,7,8,9	AI WINS
5	DRAW

HUMAN PLAYING FIRST

HUMAN'S MOVE	OUTCOME
1,2,3,4,5,6,7,8,9	DRAW



For the 3-D round of TicTacToe, it can be mathematically derived that if the first player plays optimally, i.e. plays the best possible moves in all his turns, he always wins. We can change the difficulty of the game by altering the values of different moves in static evaluators.



XXII. Future Work

Unfortunately, I couldnt implement reinforcement learning algorithms, which are commonly used to solve such problems, I'm working on them to improve the accuracy of the q-value search algorithm, and then with the use of ANNs, deep q-value search algorithms can also be implemented which could easily play higher dimensional TicTacToe after a couple thousand attempts at training.

REFERENCES

[1] Ronald l. Rivest, et al:Game Tree Searching by Min / Max Approximation* 1988

[2] Adina Rubinoff,University of Rochester, 3D TicTacToe ,2010

[3] (https://www.youtube.com/c/deeplizard/featured)

CONCLUSION

The five tasks were all unique and asked for specific skills to be used while solving them. I had more fun doing the second half of the tasks namely the puzzle, Kalman Filter and the TicTacToe, the past month added a lot to my learning curve, and I am eager pursue new things in this field which I read about - in the past month, like integration of Q-value learning and Artificial Neural Networks to form a efficient Deep Learning Model and also try particle filter based localisation methods like Monte-Carlo and Grid localisation. I never had imagined that doing these tasks would be so fun and involving.