**智慧感知與決策**

311512064鄧書桓

1. **Code:**

**1.2D semantic map construction**

這邊首先經過多次測試，取得天花板與地板在y上面的數值，並藉此將超過此範圍的點雲濾除。接著將這些點雲的x, z利用matplotlib中的scatter打印在平面上，以此獲得2D平面的語意地圖。詳細的code在下方顯示:

pcd\_data\_path = "./semantic\_3d\_pointcloud/"

# 初始化點雲

pcd = o3d.geometry.PointCloud()

# 將位置與rgb寫入

temp\_points = np.load(pcd\_data\_path + "point.npy")

temp\_colors = np.load(pcd\_data\_path + "color01.npy")

temp\_points = temp\_points \* 10000. / 255

pcd\_points = np.empty((0, 3), float)

pcd\_colors = np.empty((0, 3), float)

# min = -1.775325843388430  max = 2.365105159246803

flour = -1.35  # -1.35

ceiling = -0.1  # -0.1

for i in range(temp\_colors.shape[0]):

    if (temp\_points[i][1] > flour and temp\_points[i][1] < ceiling):

        pcd\_points = np.append(pcd\_points, [temp\_points[i]], axis=0)

        pcd\_colors = np.append(pcd\_colors, [temp\_colors[i]], axis=0)

pcd.points = o3d.utility.Vector3dVector(pcd\_points)

pcd.colors = o3d.utility.Vector3dVector(pcd\_colors)

o3d.io.write\_point\_cloud("./after\_filtering.pcd", pcd)

plt.scatter(pcd\_points[:, 2], pcd\_points[:, 0], s=5, c=pcd\_colors,  alpha=1)

plt.xlim(-6, 11)

plt.ylim(-4, 7)

plt.axis("off")

plt.savefig("map.png", bbox\_inches='tight', pad\_inches=0)

plt.show()

**2.RRT Algorithm**

這邊改善為Basic RRT，與Single RRT的差異為此為一次在原點與起點各自生點，最終連在一起以快速生路徑，這樣的RRT可以擁有較快的路徑生成速度，由於Basic RRT的初始架構與Single RRT類似，這邊就僅對Basic RRT做解釋。詳細過程如下:

img = cv2.imread(args.imagePath, 0)  # load grayscale maze image

    # 將灰階圖轉黑白圖

    thresh = 230

    img = cv2.threshold(img, thresh, 255, cv2.THRESH\_BINARY)[1]

    kernel = np.ones((5, 5), np.uint8)

    img = cv2.erode(img, kernel, iterations=2)

首先先將2D 語意地圖改成黑白圖，以便進行dilation。Dilation的概念為先使用Erosion將黑色的部分變細，順便去雜訊，再使用delicaiont將黑色部分膨脹，這邊膨脹的比縮小的更多，因此會使照片中黑色的部分變得更大，以避免robot在模擬環境中發生撞到桌椅的問題。

class Nodes:

    """Class to store the RRT graph"""

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

        self.parent\_x = []

        self.parent\_y = []

利用Node這個class分別存入當下node的位置與其parent list(這邊設計為與前幾個nearest poit相連，並與起點或終點相連)。這樣做能夠高效的儲存不同node的資訊，且可以快速使用他們各自的parant list，來達成相連。

def rnd\_point(h, l):

    new\_y = random.randint(0, h)

    new\_x = random.randint(0, l)

    return (new\_x, new\_y)

隨機生成的點是使用random.randint生成，這邊的h, l分別為2D語意地圖的長寬。

def collision(x1, y1, x2, y2):

    color = []

    # 在兩點間直線取100間隔去判斷是否有碰到障礙物

    if (x1 != x2):

        x = list(np.arange(x1, x2, (x2-x1)/100))

        y = list(((y2-y1)/(x2-x1))\*(x-x1) + y1)

        # print("collision", x, y)

        for i in range(len(x)):

            # 因為code中的x, y方向與img相反, 所以會相反

            color.append(img[int(y[i]), int(x[i])])

        if (0 in color):  # 若有一個全黑

            return True  # collision

        else:

            return False  # no-collision

    else:

        x = x1

        y = list(np.arange(y1, y2, (y2-y1)/100))

        # print("collision", x, y)

        for i in range(len(y)):

            # 因為code中的x, y方向與img相反, 所以會相反

            color.append(img[int(y[i]), int(x[i])])

        if (0 in color):  # 若有一個全黑

            return True  # collision

        else:

            return False  # no-collision

判斷兩點之間是否有障礙物是使用collision這個函數來實現的，首先先將兩點間的x距離分割成100份，並以此根據線性與斜率去推算出該點y的座標，最後利用這些共100個點的顏色去判斷是否為黑色，只要有一點為黑色，就算有障礙物，並會回傳True 或 False。

# check the  collision with obstacle and trim

# stepSize 這邊才會用到

def check\_collision(x1, y1, x2, y2, end\_points):

    \_, theta = dist\_and\_angle(x2, y2, x1, y1)

    x = x2 + stepSize\*np.cos(theta)

    y = y2 + stepSize\*np.sin(theta)

    hy, hx = img.shape

    directCon = False

    nodeCon = False

#判斷是否會超出地圖

    if y < 0 or y > hy or x < 0 or x > hx:

        {}

    else:

        # check connection between two nodes

        if collision(x, y, x2, y2):

            nodeCon = False

        else:

            nodeCon = True

            # 是否該node(與給定點多一個step size的點）與終點相連

            if collision(x, y,  end\_points[0], end\_points[1]):

                directCon = False

            else:

                directCon = True

    return (x, y, directCon, nodeCon)

Check\_collision 函式的功能為利用隨機生成的點與其和已生成的tree中的最近點連成的方向，以一個stepsize(本功課設計為20)生成一個暫態點，並根據先前使用到的各個function來此暫態點與最近點是否相連且無障礙物，若符合則改變變數並繼續判斷此暫態點是否能直接和終點相連，若符合則完成Basic\_RRT，反之則繼續生成隨機點做一直重複下去。

def RRT(img, img2, start, end, stepSize):

    h, l = img.shape  # dim of the loaded image

    # insert the starting point in the node class

    # node\_list = [0] # 裡面存有所有node的位置資訊

    # 起點長的

    temp\_node\_list = [0]

    end\_node\_list = [0]

    temp\_node\_list[0] = Nodes(start[0], start[1])

    temp\_node\_list[0].parent\_x.append(start[0])

    temp\_node\_list[0].parent\_y.append(start[1])

    # 終點長的

    end\_node\_list[0] = Nodes(end[0], end[1])

    end\_node\_list[0].parent\_x.append(end[0])

    end\_node\_list[0].parent\_y.append(end[1])

    # display start and end , 都為藍色圈圈

    cv2.circle(img2, (start[0], start[1]), 5,

               (0, 0, 255), thickness=3, lineType=8)

    cv2.circle(img2, (end[0], end[1]), 5, (0, 0, 255), thickness=3, lineType=8)

    end\_points = [end[0], end[1]]

    i = 1

    num\_nodes = 1

    pathFound = False

while pathFound == False:

        nx, ny = rnd\_point(h, l)

        nearest\_ind = nearest\_node(nx, ny, temp\_node\_list)

        nearest\_x = temp\_node\_list[nearest\_ind].x

        nearest\_y = temp\_node\_list[nearest\_ind].y

        outerside\_nearestid = nearest\_node(nx, ny, end\_node\_list)

        end\_points = [end\_node\_list[outerside\_nearestid].x,

                      end\_node\_list[outerside\_nearestid].y]

        # check direct connection

        # 會往前走一個step tx, ty

        tx, ty, directCon, nodeCon = check\_collision(

            nx, ny, nearest\_x, nearest\_y, end\_points)

        # print("Check collision:", tx, ty, directCon, nodeCon)

        # 都true表示可以直接與終點相連

        if directCon and nodeCon:

            # print("Node can connect directly with end")

            k = len(temp\_node\_list)

            temp\_node\_list.append(k)  # 用於初始化node\_list[i] ,沒其他功能

            temp\_node\_list[k] = Nodes(tx, ty)

            # list.copy() 用於list內容的複製，而非記憶體複製

            # 此處將最近點的路徑都複製進新走一步的parent list 中

            temp\_node\_list[k].parent\_x = temp\_node\_list[nearest\_ind].parent\_x.copy()

            temp\_node\_list[k].parent\_y = temp\_node\_list[nearest\_ind].parent\_y.copy()

            temp\_node\_list[k].parent\_x.append(tx)

            temp\_node\_list[k].parent\_y.append(ty)

            for j in range(len(end\_node\_list[outerside\_nearestid].parent\_x)):

                temp\_node\_list[k].parent\_x.append(

                    end\_node\_list[outerside\_nearestid].parent\_x[-(j+1)])

                temp\_node\_list[k].parent\_y.append(

                    end\_node\_list[outerside\_nearestid].parent\_y[-(j+1)])

            cv2.circle(img2, (int(tx), int(ty)), 2,

                       (0, 0, 255), thickness=3, lineType=8)

            # 將最近點與剛走得step以綠色相連

            cv2.line(img2, (int(tx), int(ty)), (int(temp\_node\_list[nearest\_ind].x), int(

                temp\_node\_list[nearest\_ind].y)), (0, 255, 0), thickness=1, lineType=8)

            # 最佳相連的路徑變藍色

            for j in range(len(temp\_node\_list[k].parent\_x)-1):

                cv2.line(img2, (int(temp\_node\_list[k].parent\_x[j]), int(temp\_node\_list[k].parent\_y[j])), (int(

                    temp\_node\_list[k].parent\_x[j+1]), int(temp\_node\_list[k].parent\_y[j+1])), (255, 0, 0), thickness=2, lineType=8)

            cv2.waitKey(1)

            cv2.imwrite("media/"+str(i)+".jpg", img2)

            cv2.imwrite("out.jpg", img2)

            cv2.imshow("out.jpg", img2)

            cv2.waitKey()

            break

        # 表示新的step不會撞牆, 但還沒到終點

        elif nodeCon:

            temp\_node\_list.append(i)

            j = len(temp\_node\_list)-1

            temp\_node\_list[j] = Nodes(tx, ty)

            temp\_node\_list[j].parent\_x = temp\_node\_list[nearest\_ind].parent\_x.copy()

            temp\_node\_list[j].parent\_y = temp\_node\_list[nearest\_ind].parent\_y.copy()

            temp\_node\_list[j].parent\_x.append(tx)

            temp\_node\_list[j].parent\_y.append(ty)

            print("{}th iteration".format(i))

            i = i+1

            num\_nodes = num\_nodes + 1

            # display

            cv2.circle(img2, (int(tx), int(ty)), 2,

                       (0, 0, 255), thickness=3, lineType=8)

            # 將最近點與剛走得step以綠色相連

            cv2.line(img2, (int(tx), int(ty)), (int(temp\_node\_list[nearest\_ind].x), int(

                temp\_node\_list[nearest\_ind].y)), (0, 255, 0), thickness=1, lineType=8)

            cv2.imwrite("media/"+str(i)+".jpg", img2)

            cv2.imshow("sdc", img2)

            cv2.waitKey(1)

            temp\_list = temp\_node\_list.copy()

            temp\_node\_list = end\_node\_list.copy()

            end\_node\_list = temp\_list.copy()

            continue

        else:

            # print("No direct con. and no node con. :( Generating new rnd numbers")

            continue

這邊為主要Basic\_RRT在做疊代的地方，大致想法如下，先將起始點與終點各自生成list，裡面的element為上述的Node(class)，用來記錄不同的node與其parent tree。接著，根據一次只生長一邊tree的想法來記錄各點資訊，其中，end point會隨著生成的點而改變，舉例來說，起點的tree新生成的node會變成終點tree的end point，以此來判斷兩個樹是否能夠直接相連。最後，這邊使用類似swap的想法來節省coding的空間與時間，每次生為點後，都會將兩個樹swap，以此來達成高效Basic\_RRT的想法。

temp\_x = temp\_node\_list[k].parent\_x

    temp\_y = temp\_node\_list[k].parent\_y

    temp\_x = np.float64(temp\_x)\*(17/l) - 6

    temp\_y = 7 - (np.float64(temp\_y)\*(11/h))

    final\_nodes = np.empty((0, 2), float)

    for j in range(temp\_x.shape[0]):

        final\_nodes = np.append(final\_nodes, [[temp\_y[j], temp\_x[j]]], axis=0)

    np.save("path\_nodes", final\_nodes)

最後將生成路徑上各個node的資訊轉成模擬環境的座標，先前存2D語意地圖時已經將長寬scale成6\*7，因此這邊僅要做些許調整就能轉換成功，實際轉換如上述code。

3. **Convert route to discrete actions**

這邊的概念為判斷robot於模擬環境中當下的點與下個node點的距離與角度差，並先將角度差降至0(利用當下的點、過去的點與下一個node位置來形成兩個向量，並判斷他們外積的正負來判斷該左轉還是右轉)，之後再直走以到達下一個node，值得注意的是，這邊都是使用”步數”來實現的，也就是每一步轉多少或是往前走多遠來計算的，另外，當下的點是利用robot在模擬環境的位置來做後續判斷的，因此，儘管因為其他因素(如撞到桌子)而沒抵達下一個node，robot也能成功抵達目標點。實際的程式如下:

while not arrive:

    keystroke = cv2.waitKey(0)

    if keystroke == ord(STEP\_PLUS\_ONE):

        current\_face = new\_node - current\_node

        theta, distance = point\_dif(current\_face, last\_face)

        # 先轉角度

        if agl\_step != int(theta / agl\_per\_step) and (not Angle\_temp\_arrive):

            print("total ratate step :", int(theta / agl\_per\_step))

            if np.cross(last\_face, current\_face) > 0:

                action = "turn\_right"

            else:

                action = "turn\_left"

            agl\_step += 1

            print("angle\_approuching")

            print("i th turn : ", agl\_step)

        elif not Angle\_temp\_arrive:

            Angle\_temp\_arrive = not Angle\_temp\_arrive

            agl\_step = 0

            print("angle\_finish")

        # 在直走

        if Angle\_temp\_arrive and (dis\_step != int(distance / dis\_per\_step)):

            action = "move\_forward"

            dis\_step += 1

            print("total walk step :", int(distance / dis\_per\_step))

            print("i th move foward : ", dis\_step)

        elif Angle\_temp\_arrive and (not Distance\_temp\_arrive):

            print("straight\_finish")

            Distance\_temp\_arrive = not Distance\_temp\_arrive

        if Distance\_temp\_arrive & Angle\_temp\_arrive:

            print("{}th node is find".format(node\_num))

            dis\_step = 0

            node\_num += 1

            print(node\_num)

            print(len(path\_nodes))

            Angle\_temp\_arrive = not Angle\_temp\_arrive

            Distance\_temp\_arrive = not Distance\_temp\_arrive

            last\_node = current\_node.copy()

            current\_node = [posit[0], posit[2]]

            new\_node = path\_nodes[node\_num+1]

            # last\_node = path\_nodes[node\_num-1]

            last\_face = [current\_node[i] - last\_node[i] for i in range(2)]

            print("ideal point : ", path\_nodes[node\_num])

            print("current  point : ", current\_node)

            continue

        else:

            action\_img = navigateAndSee(action)

            videowriter.write(action\_img)

    elif keystroke == ord(FINISH):

        print("action: FINISH")

        break

    else:

        print("INVALID KEY")

        continue

    if node\_num == len(path\_nodes)-2:  # 這邊很奇怪, 本來是2

        arrive = not arrive

        videowriter.release()

        print("finish  ")

        break

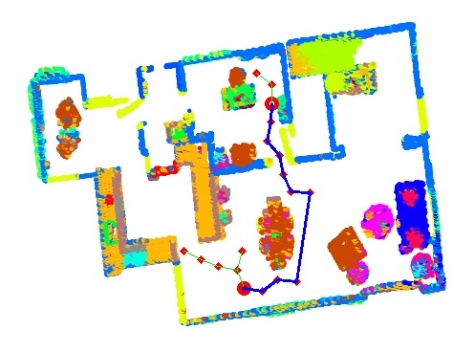
print("find the target point!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!")

1. **Results and Discussion**
2. **Show and discuss the results from the RRT algorithm with different start points and targets.**

隨機起點與cooktop 隨機起點與cusion 一張含有 文字 的圖片

自動產生的描述

隨機起點與lamp 隨機起點與rack

隨機起點與refrigerator



由上述幾張圖可知，Basic RRT成功找到目標點並將此與起點相連，圖中綠色的虛線為各個node生成的parent tree，綠色線條為找到的最佳路徑，紅色小點為node，紅色大點為起點與目標點。Basic RRT在實作時遇到的問題為一開始設計時沒想到使用swap來節省coding的空間與時間，所幸在朋友的提醒下才想到，這樣不但省時又省空間，還很好debug。還有個地方值得注意，由於這邊是使用swap的方式來不斷更改end point，因此會有機率導致實際的起點與終點相反，導致路途是反的，所幸有即時發現，並沒有一直錯下去。

1. **Discuss about the robot navigation results**

由於有放影片，這邊就不放圖片了。這邊遇到的問題主要是在實現想法時遇到的，本身並沒有什麼較難的數學推導，而我遇到卡最久的地方為設定是否達成角度正確的部分，由於變數使用錯誤導致一直失敗，最終也是找了很久才成功解決。

1. **Bonus**

我有使用Basic RRT來做優化，實際如何設計已在上方做說明，這邊僅給出比較的差異。

Single RRT Basic RRT



由上方兩個圖可知，rrt僅起點開始生長tree，引此需要較多的點也較費時，而basic rrt由於是起點與終點接生長tree，因此花的點較少且省時。

**iii. Reference**本篇的RRT是根據<https://github.com/nimRobotics/RRT>改寫的。