智慧感知與決策

311512064 鄧書桓

i. 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!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!")
```

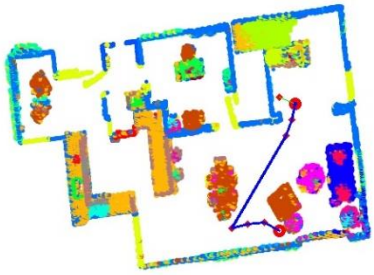
ii.Results and Discussion

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

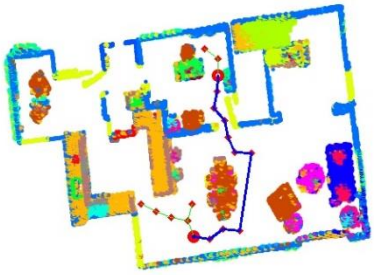
   隨機起點與 cooktop　　隨機起點與 cusion

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。

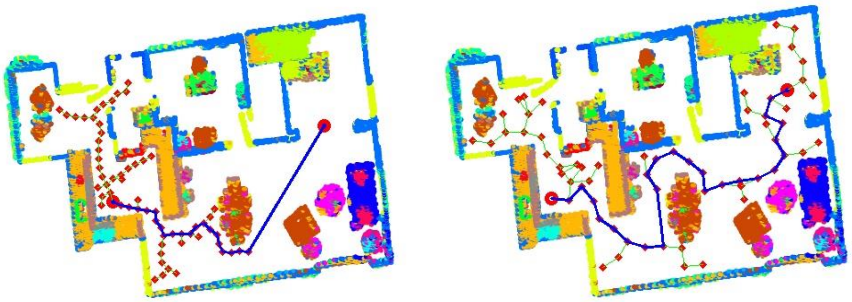
2. **Discuss about the robot navigation results**

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。

3. 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。