Homework 3 Report

Name: 謝元碩
Student ID: 311512015

i. Code

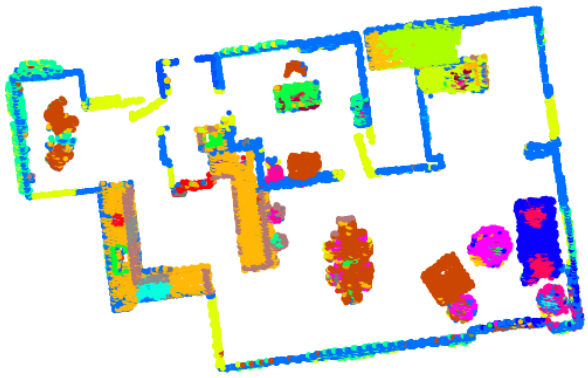
1. map.py: 將 3D 語義地圖轉成 2D 地圖

```
point = np.load('semantic_3d_pointcloud/point.npy')
color = np.load('semantic_3d_pointcloud/color01.npy')

# remove the ceiling and the floor
point_cut = point[np.where((point[:,1]>(-0.03)) & (point[:,1]<(-0.005)))]
color_cut = color[np.where((point[:,1]>(-0.03)) & (point[:,1]<(-0.005)))]
point_cut = point_cut*10000./255.

# plt.figure(figsize=(15,11))
plt.scatter(point_cut[:,2], point_cut[:,0], s=5, c=color_cut, alpha=1)
plt.axis('equal')
plt.ylim(-4,7)
plt.xlim(-5,10)
plt.axis('off')
plt.savefig("map.png", bbox_inches='tight',pad_inches = 0)
plt.show()
```

- 讀取 spec 中所給予的 point 和 color 後，首先使用 np.where 去除天花板和地板部分的點雲，並將點雲座標*10000/255 轉換成實際座標。
- 接著將點雲座標的 x、z 值用 Matplotlib 呈現，這裡要注意的是，橫軸及縱軸必須手動設定上下界，且兩軸的比例必須要一樣。最後存圖時，我將座標軸移除顯示，同時去除座標軸以外的空白部分。
- 前述動作的用意，主要是為了後面使用 rrt 在圖片中做完路徑規劃後，讓路徑能夠依照已知的比例轉換到 open3D 環境中。這裡所儲存的照片比例為 496pixels*369pixels，圖中座標橫軸為 x=15m、縱軸為 z=11m，儲存地圖如下。



2. 2D 地圖之 RRT Algorithm 實現

以下將對個人完成的 RRT 演算法做程式細節說明，以及 Bonus 部分：我如何改善 RRT 並給予優化。

(1) Single RRT

```
map = cv2.imread("map.png")
map_bw = cv2.imread("map.png",0)

#####image erosion to make obstacle more transparent#####
kernel = np.ones((5,5), np.uint8)
erosion = cv2.erode(map_bw, kernel, iterations = 2)
```

- 首先是讀取圖片，以及使用影像侵蝕技術建立一張 gray-level image。經影像侵蝕後的地圖，會減少原本白色的部分，非白色部分則有膨脹效果，這代表牆壁的厚度也將變厚，此目的是為了不要讓 RRT 在產生新的點時太容易穿越障礙物。後面的 RRT 計算都是使用這張圖進行，原始彩色地圖僅為顯示用途。

```
#####get object color rgb#####
object = input("I want to search: ")
wb = openpyxl.load_workbook('category.xlsx', data_only=True)
s1 = wb['Sheet1']
for i in range(2,103):
    if s1.cell(i,5).value == object:
        object_color = s1.cell(i,2).value
        break
wb.save('category.xlsx')
object_color = object_color.replace("(","").replace(")","").replace(",","")
object_color = np.array([int(temp)for temp in object_color.split() if temp.isdigit()]) # take the numbers
# print(object_color)
```

- 這裡輸入欲抵達的目標物件，再使用作業提供之物件色盤對照表找出目標物件的對應顏色。

```
#####get the mean position pixel of the object#####
object_point = []
for i in range(map.shape[0]):
    for j in range(map.shape[1]):
        if map[i,j,0] == object_color[2] and map[i,j,1] == object_color[1] and map[i,j,2] == object_color[0]:
            object_point.append([i,j])
# dst = map[np.where((map[:,:,0] == object_color[2]) & (map[:,:,1] == object_color[1]) & (map[:,:,2] == object_color[0]))]
object_mean = np.round(np.mean(np.array(object_point), axis=0)).astype(int)
# print("mean point: ", object_mean)
```

```
#####choose the end point(hard code)#####
if object == "refrigerator":
    end_point = [object_mean[1]+2, object_mean[0]+15]
    cv2.circle(map, end_point, 3, (0, 0, 0), -1)
elif object == "rack":
    end_point = [object_mean[1]-16, object_mean[0]+4]
    cv2.circle(map, end_point, 3, (0, 0, 0), -1)
elif object == "cushion":
    end_point = [object_mean[1]-27, object_mean[0]-26]
    cv2.circle(map, end_point, 3, (0, 0, 0), -1)
elif object == "lamp":
    end_point = [object_mean[1]-23, object_mean[0]-1]
    cv2.circle(map, end_point, 3, (0, 0, 0), -1)
elif object == "cooktop":
    end_point = [object_mean[1]+1, object_mean[0]-28]
    cv2.circle(map, end_point, 3, (0, 0, 0), -1)
```

- 接著去紀錄該顏色出現在地圖中的座標位置，並計算出這些座標之平均值(目標物件中心座標)。
- 將這些中心點座標平移至欲抵達的座標點，以作為未來 RRT 規劃的**導航終點**。

```
#####choose a start point#####
cv2.imshow("map",map)
cv2.setMouseCallback('map', click_event)
cv2.waitKey()
cv2.destroyAllWindows()
```

```
start_point = []
def click_event(event, x, y, flags, params):

    global start_point
    if event == cv2.EVENT_LBUTTONDBLCLK:
        start_point = [x, y]
        cv2.circle(map, start_point, 3, (0, 0, 255), -1)
        cv2.imshow('map', map)
```

- 使用滑鼠左鍵在地圖上雙擊起始點位置，點擊之 callback function 如右圖。

```
# initialize
step_size = 15
node_list = [0]

# insert the starting point in the node class
node_list[0] = Nodes(start_point[0],start_point[1])
node_list[0].parent_x.append(start_point[0])
node_list[0].parent_y.append(start_point[1])
cv2.circle(map, start_point, 3, (0, 0, 255), -1)
```

```
class Nodes:
    def __init__(self, x,y):
        self.x = x
        self.y = y
        self.parent_x = []
        self.parent_y = []
```

- 在 RRT 計算前，必須初始化節點 list 以及 step size。這裡使用資料結構方法：物件 list 去紀錄每個節點的 xy 值和 parent node 紀錄，第 0 個 node 即為起始點。

```python
def find_nearest_point(node_list, target):
    temp_dist = []
    for j in range(len(node_list)):
        dist = np.linalg.norm(np.array(target) - np.array([node_list[j].x, node_list[j].y]))
        temp_dist.append(dist)
    near_idx = temp_dist.index(min(temp_dist))
    nearest = [node_list[near_idx].x, node_list[near_idx].y]
    return near_idx, nearest
```

- RRT 演算法中，我將三項重複使用到的步驟設計成函式，首先是尋找最近點。
- 此函式會搜尋出 node list 中距離 target 最近的節點，並回傳節點座標及索引。

```python
def generate_point(target, nearest):
    vecter = np.array(target)-np.array(nearest)
    if target == nearest:
        new_point = nearest
    else:
        length = np.linalg.norm(vecter)
        vecter = (vecter / length) * min(step_size, length)
        new_point = [int(nearest[0]+vecter[0]), int(nearest[1]+vecter[1])]
    return new_point
```

- 第二項是產生新點。這裡會將 target 和最近點的線段切成數段，並使用 step size 或 length(單位向量長)，讓最近點往 target 走一小步，產生一顆新的點。

```python
def add_new_node(node_list, i, new_point, near_idx):
    node_list.append(i)
    node_list[i] = Nodes(new_point[0],new_point[1])
    node_list[i].parent_x = node_list[near_idx].parent_x.copy()
    node_list[i].parent_y = node_list[near_idx].parent_y.copy()
    node_list[i].parent_x.append(new_point[0])
    node_list[i].parent_y.append(new_point[1])
    i = i + 1
    cv2.circle(map, new_point, 3, (0, 255, 0), -1)
    cv2.line(map, nearest, new_point, (0, 255, 0), 1)
    return node_list, i
```

- 第三個函式為新增新節點。這裡會在 list 中新增新的 class Node，紀錄新節點的座標以及從起始點延伸過來的所有父節點。

```
##### RRT algorithm #####
i = 1
while True:
    Xrand = [random.randint(0,map.shape[1]), random.randint(68,map.shape[0])] # set a random point

    # find closest point
    near_idx, nearest = find_nearest_point(node_list, Xrand)

    # generate new point
    new_point = generate_point(Xrand, nearest)

    if erosion[new_point[1]-1, new_point[0]-1] == 255:
        # save nodes
        node_list, i = add_new_node(node_list, i, new_point, near_idx)


    end_range = 5
    if end_point[0]-end_range<new_point[0]<end_point[0]+end_range and end_point[1]-end_range<new_point[1]<end_point[1]+end_range:
        node_list, i = add_new_node(node_list, i, new_point, near_idx)
        i = i - 1
        final_i = i
        print("new point: ", new_point)
        print("Find path")
        for j in range(len(node_list[i].parent_x)-1):
            cv2.line(map, (int(node_list[i].parent_x[j]),int(node_list[i].parent_y[j])), (int(node_list[i].parent_x[j+1]),int(node_list[i].parent_y[j+1])), (0,0,255), 2)
        break
```

- 此部分為 RRT 演算法最主要的程式部份。在找到終點前，此迴圈會不斷重複執行，直到抵達目的地為止。
- 每次迭代中，第一步使用 random library 在圖片中產生一顆隨機點。
- 接著尋找此隨機點與當前 tree node list 的最近節點，並生成新點座標。
- 若此新座標為白色(不是障礙物)，則將此點新增至 tree 中。
- 每次迭代不斷重複執行前述步驟。一旦抵達終點附近(我設置為±5pixel 內)，即紀錄該節點 index，畫出導航路徑並退出迴圈，完成計算。

```
# save navigation point
navigation_point = []
for i in range(len(node_list[final_i].parent_x)):
    pixel2real_x = int(node_list[final_i].parent_x[i])*(15/map.shape[1])
    pixel2real_y = int(node_list[final_i].parent_y[i])*(11/map.shape[0])
    navigation_point.append([(-pixel2real_y)+7, pixel2real_x-5]) # translation hard code
# print(navigation_point[i])
```

- 由於圖片上座標系為 pixel，因此需經過座標轉換，將圖片座標轉移至 open3D 環境座標。
- 此迴圈即對每一個 node 做座標轉換。在前面 map.py 中已將實際長寬與圖片長寬比例固定，因此僅需透過比例上的轉換，再做些許平移，即可取得在 open3D 中的導航座標，完成 RRT 路徑規劃以及地圖與實際座標轉換。

(2) Bonus: Single smooth RRT

由於在撰寫完基本款 RRT 後，發現此路徑規劃方式有難收斂的狀況。若生成的節點都正好在終點附近(終點範圍邊界外)，則很難將附近的節點再做延伸以抵達終點範圍，只有隨機點距離終點非常近時才可能收斂成功。因此我將演算法做以下改善，如下圖：

```
if erosion[new_point[1]-1, new_point[0]-1] == 255:
    # save nodes
    node_list, i = add_new_node(node_list, i, new_point, near_idx)

    # find nearest point to end point
    near_idx, nearest = find_nearest_point(node_list, end_point)
    new_point = generate_point(end_point, nearest)
    while erosion[new_point[1]-1, new_point[0]-1] == 255:
        node_list, i = add_new_node(node_list, i, new_point, near_idx)
        new_point = generate_point(end_point, new_point)

        if new_point == end_point:
            node_list, i = add_new_node(node_list, i, new_point, near_idx)
            i = i - 1
            path_i = i
            print("new point: ", new_point)
            print("Find path")
            for j in range(len(node_list[i].parent_x)-1):
                cv2.line(map, (int(node_list[i].parent_x[j]),int(node_list[i].parent_y[j])), (int(node_list[i].parent_x[j+1]),int(node_list[i].parent_y[j+1])), (0,0,255), 2)
            find_path = 1
            break

cv2.imshow("map",map)
cv2.waitKey(1)

if find_path:
    break
```

- 紅色框框內為主要變動的部分。原本在每次迭代中，最多僅會往隨機點新增一個新節點，這樣的效率其實並不好，且抵達終點有部分運氣成分在，因此我在演算法後面新增了「對終點延伸新節點」的無窮迴圈。
- 每當透過隨機點新增新節點後，我都會再讓 tree node list 對終點做最近點延伸，一樣透過前面對隨機點延伸的方法，我將「對終點的最近點」向終點不斷延伸，直到新節點碰到障礙物為止。
- 此方法的好處是，這將會省下非常多隨機延伸新節點的計算時間，將原本導航的路徑產生平滑化的效果，只要是能夠直線靠近終點的區域，演算法都會以直線路徑取代彎曲路徑，不僅提高演算法效率，連規劃路徑都變漂亮許多。

(3) Bonus: Bi-RRT

另外，我也嘗試使用 Bi-RRT 的概念，去測試、比較與前兩者的路徑差異。此方法的核心概念，就是終點座標也會像起始點座標一樣生成隨機樹，當兩顆隨機樹節點間的距離小於一定閥值，即會完成連線，生成一導航路徑。演算法如下：

```
# insert the end point in the node class b
node_list_b[0] = Nodes(end_point[0],end_point[1])
node_list_b[0].parent_x.append(end_point[0])
node_list_b[0].parent_y.append(end_point[1])
cv2.circle(map, end_point, 3, (0, 0, 255), -1)
```

(一開始需新增第二個 node list 並初始化)

以隨機點對 Ta 尋找最近點並新增

```
##### RRT algorithm #####
i = 1
ib = 1
find_path = 0
while True:
    Xrand = [random.randint(0,map.shape[1]), random.randint(68,map.shape[0])] # set a random point

    # find closest point
    near_idx, nearest = find_nearest_point(node_list, Xrand)

    # generate new point
    new_point = generate_point(Xrand, nearest)

    if erosion[new_point[1]-1, new_point[0]-1] == 255:
        # save nodes
        node_list, i = add_new_node(node_list, i, new_point, near_idx)
```

Ta 新節點成為 target，尋找與 Tb 的最近點，若無障礙物則不斷延伸

```
        # find nearest point to target, and extend
        target = new_point
        near_idx, nearest = find_nearest_point(node_list_b, target)
        new_point = generate_point(target, nearest)
        while erosion[new_point[1]-1, new_point[0]-1] == 255:
            node_list_b, ib = add_new_node(node_list_b, ib, new_point, near_idx)
            new_point = generate_point(target, new_point)

            cv2.imshow("map",map)
            cv2.waitKey(1)
```

若 Tb 新節點與 Ta target 距離小於 step size，則完成規劃並記錄路徑

```
        dist = np.linalg.norm(np.array(target) - np.array(new_point))
        if dist < step_size:
            print("Match the tree")
            ib = ib - 1
            i = i - 1
            for j in range(len(node_list_b[ib].parent_x)-1):
                record_path.append([node_list_b[ib].parent_x[j], node_list_b[ib].parent_y[j]])
                cv2.line(map, (int(node_list_b[ib].parent_x[j]),int(node_list_b[ib].parent_y[j])), (int(node_list_b[ib].parent_x[j+1]),int(node_list_b[ib].parent_y[j+1])), (0,0,255), 2)
            for j in range(len(node_list[i].parent_x)-1):
                record_path.append([node_list[i].parent_x[len(node_list[i].parent_x)-1-j], node_list[i].parent_y[len(node_list[i].parent_x)-1-j]])
                cv2.line(map, (int(node_list[i].parent_x[j]),int(node_list[i].parent_y[j])), (int(node_list[i].parent_x[j+1]),int(node_list[i].parent_y[j+1])), (255,0,0), 2)
            cv2.imshow("map",map)
            cv2.waitKey()
            cv2.destroyAllWindows()
            for j in range(len(record_path)-1):
                cv2.line(map, record_path[j], record_path[j+1], (0,0,255), 2)
            if record_path[0] == end_point:
                record_path.reverse() # make the end of path list is the end point
            find_path = 1
            # print(record_path)
            break
```

Ta、Tb 互換

```
    node_list, node_list_b = node_list_b, node_list
    i, ib = ib, i

    cv2.imshow("map",map)
    cv2.waitKey(1)

    if find_path:
        break
```

- 演算法一開始，對第一個節點 Ta 做產生隨機點、找最近點、新增新節點等動作。
- 接著將 Ta 的新節點當作第二個 tree Tb 的 target，並尋找距離 target 最近的節點。這裡同 single smooth RRT，若無障礙物，則將最近點持續往 target 延伸生成新節點，直到遇到障礙物為止。
- 接著將 Ta、Tb 以及 index 值 i、ib 分別互換，讓每次迭代可以讓兩顆 tree 輪流進行前述步驟，直到其中一個 tree 的新節點與 target 距離小於 step size。

- 最後將兩個 node list 的 parent x,y 值連接並新增至 record_path，退出迴圈，此 record_path 即為 Bi-RRT Algorithm 所規劃出之導航路徑。
- 這個演算法同時對起點跟終點做 Single smooth RRT，兩邊的 tree 同時生長，讓路徑銜接起來的速度也有一定的提升效果，路徑平滑程度也比基本款 RRT 高出許多。

3. navigation.py: 規劃路徑在 open3D 環境中導航之實現

   在此部分，首先會說明我如何應用 2D 導航路徑，在 3D 環境中以動畫呈現導航。接著會說明如何將目標物件 highlight。

(1) 3D 環境導航

```
# save video initial
path = "video/" + object + ".mp4"
fourcc = cv2.VideoWriter_fourcc(*'mp4v')
videowriter = cv2.VideoWriter(path, fourcc, 2, (512, 512))

action = "move_forward"
x_now, y_now, theta_now, img = navigateAndSee(action)
videowriter.write(img)

nav_step = 1
nav_state = 0
```

- 由於必須記錄導航過程的畫面，我使用 opencv 的 videowriter，以照片記錄每一次的節點和轉向，並用影片儲存及呈現。
- 在進入導航迴圈前，需先執行一次 navigateAndSee，紀錄初始的鏡頭姿態，並初始化導航 step 和導航狀態(0 為轉彎、1 為直走)。

```python
while True:
    # rotate state
    if nav_state == 0:

        # make the angle range be -180~180 degrees, not 0~90 and 0~-90 degrees
        if theta_now[0] == 0 and theta_now[2] == 180:
            if 0<=theta_now[1]<=90:
                theta_now[1] = 180 - theta_now[1]
            elif -90<=theta_now[1]<=0:
                theta_now[1] = (-180) - theta_now[1]
        # print("euler: ", theta_now)

        # decide to turn left or right
        nav_vector = np.array(navigation_point[nav_step]) - np.array([x_now, y_now])
        theta_nav = math.atan2(nav_vector[1], nav_vector[0])/math.pi*180
        theta_face = theta_now[1]
        delta = theta_nav - theta_face + 90
        # print("delta: ", delta)

        if delta < (-180):
            delta = 360 + delta
        elif delta > 180:
            delta = -360 + delta

        if 0<=delta<180:
            print("turn right")
            action = "turn_right"
            x_now, y_now, theta_now, img = navigateAndSee(action)
        elif -180<=delta<0:
            print("turn left")
            action = "turn_left"
            x_now, y_now, theta_now, img = navigateAndSee(action)

        if -0.5<delta<0.5:
            print("turn complete")
            videowriter.write(img)
            nav_state = 1
```

- 導航迴圈主要分成兩個狀態：原地轉向及直線前進。在此部分演算法的主要概念，是讓 agent 每次在節點上時，先旋轉至面向下一個節點的角度，接著直走至下一個節點。經不斷重複轉向和直走後，即可抵達目的地並停止。
- 上圖為轉向的程式碼。角度部分，我皆使用尤拉角做計算，因此我在 navigateAndSee 中會對四元數做轉換的動作。
- 首先將尤拉角轉換成-180~180 度的表示方式(原本四元數轉尤拉角後是兩邊範圍皆為-90~90 度，不好拿來計算)，接著我使用下個節點和當下節點座標相減產生一向量，此向量可產生一個欲朝向的目標角度。透過此目標角度及當下角度的差，去決定 agent 要進行左轉或右轉。這裡有一步驟是針對 delta 角度值做修正，因為在某些角度相減後會產生超出±180度的 delta，因此我會將這些角度+360 或-360，讓 delta 回歸到±180度內，這樣才能做左右轉決策。
- 若角度差值小於一定閥值，代表已經面向正確朝向，state 轉換為 1。

```
# go straight state
elif nav_state == 1:
    print("go")
    action = "move_forward"
    x_now, y_now, theta_now, img = navigateAndSee(action)
    end_range = 0.1
    if navigation_point[nav_step][0]-end_range<x_now<navigation_point[nav_step][0]+end_range and navigation_point[nav_step][1]-end_range<y_now<navigation_point[nav_step][1]+end_range:
        videowriter.write(img)
        if nav_step == len(navigation_point)-1: # when arrived
            print("arrived")
            videowriter.release()
            cv2.waitKey()
            break
        else:
            print("get to one point")
            nav_state = 0
            nav_step = nav_step +1

cv2.waitKey(1)
```

- 在狀態二，會進行直走的動作。此狀態較單純，會不斷讓 agent 直走，直到當下座標進入下個節點的座標範圍內即停止，並且在當抵達的節點 index 為最後一項時，代表已經導航到終點了，所以會結束導航、儲存影片。

(2) Object highlight

```
# load colors
colors = loadmat('color101.mat')['colors']
colors = np.insert(colors, 0, values=np.array([[0,0,0]]), axis=0) #to make the color be correct
```

- 首先如 HW2 一樣讀取顏色 mat 檔
- 由於在實際上色時，發現顏色有錯誤的情況，後來發現每個物件都使用到色盤中下一個 index 的顏色，因此我在這個顏色 array 最前面新增一行任意 array，讓顏色與物件能夠正確配對並上色。

```
def transform_semantic(semantic_obs):
    semantic_img = Image.new("P", (semantic_obs.shape[1], semantic_obs.shape[0]))
    semantic_img.putpalette(colors.flatten())
    semantic_img.putdata(semantic_obs.flatten().astype(np.uint8))
    semantic_img = semantic_img.convert("RGB")
    semantic_img = cv2.cvtColor(np.asarray(semantic_img), cv2.COLOR_RGB2BGR)
    return semantic_img
```

- 此部分我做了些修改，讓程式中可以使用作業提供的色盤上色，產生語義地圖。

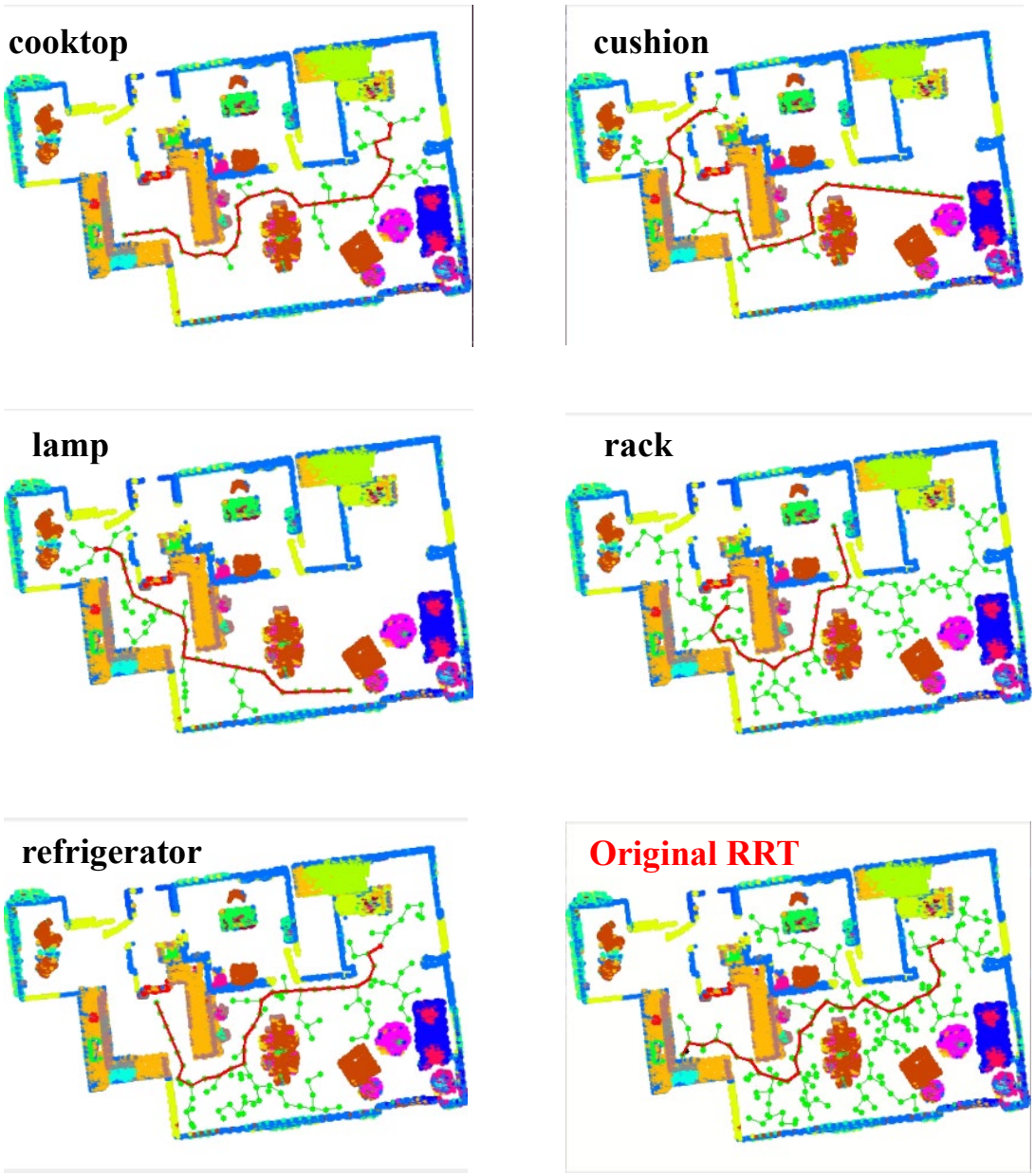
```
highlighted = object_highlight(transform_rgb_bgr(observations["color_sensor"]), transform_semantic(id_to_label[observations["semantic_sensor"]]))
```

```
# highlight the object
def object_highlight(rgb, sem):
    index = np.where((sem[:,:,2]==object_color[0]) & (sem[:,:,1]==object_color[1]) & (sem[:,:,0]==object_color[2]))
    if len(index[0]) != 0:
        rgb[index] = cv2.addWeighted(rgb[index], 0.6, sem[index], 0.4, 50)
    return rgb
```

- 前面產生的語義地圖，主要是為了使用地圖中目標物件的顏色，去加到 RGB 圖片中做上色。
- 這裡我用 np.where 尋找目標物件顏色的 index，並對 RGB 圖中對應的 index 部分做影像相加，RGB 和 semantic 相加比例為 0.6:0.4。
- 最後回傳 highlight 後的圖片至 navigateAndSee 中，導航動畫及 mp4 製作部分也是都使用 highlight 後的圖片。

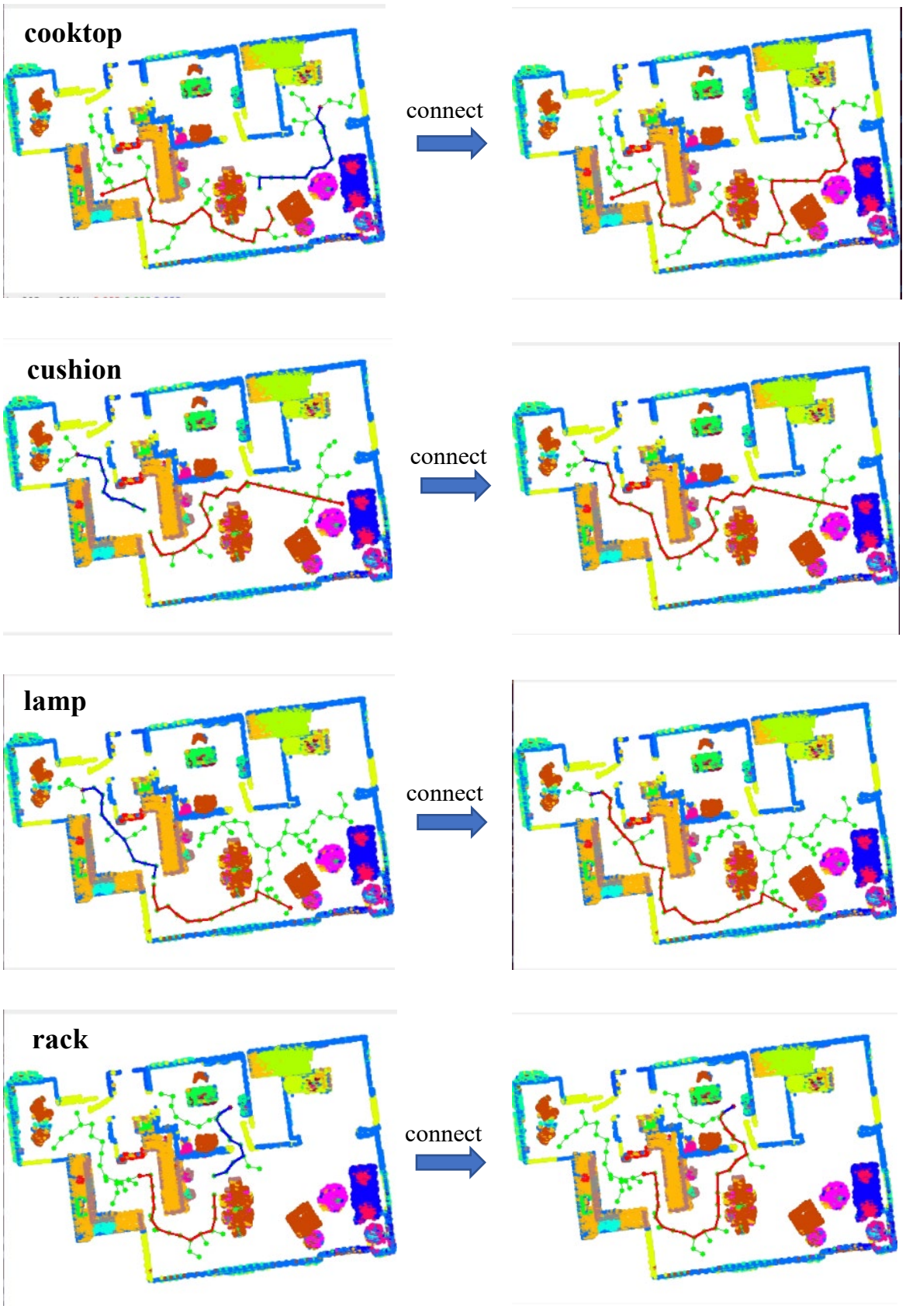
ii. Results and Discussion

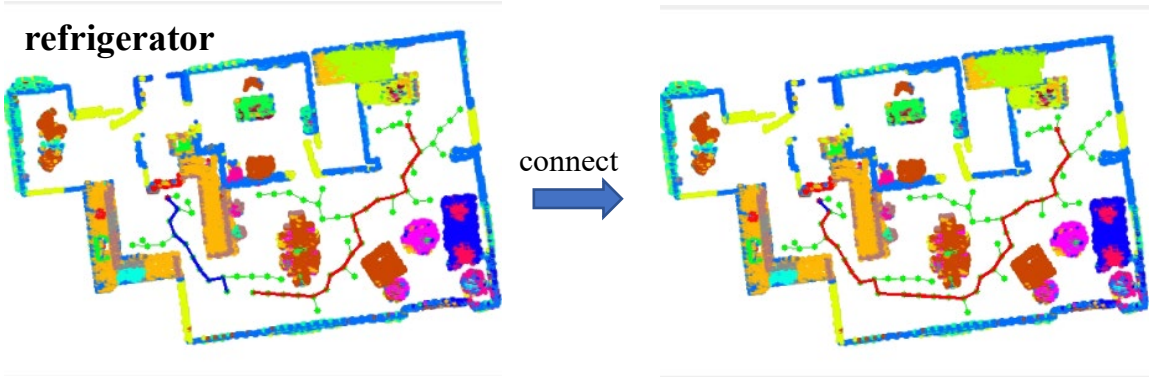
1. Single smooth RRT



- 從上圖可以明顯看到，在 smooth 前的 RRT 所規劃出的路徑是非常曲折的，生出的 tree 也特別密，代表此演算法經過迭代多次才找到終點。
- 經 smooth 後的 RRT 路徑規劃圖，五種目標物的導航路徑如上。可以看到在部分的路段，演算法以直線前進的動作延伸 tree，取代繼續產生隨機點這個方式，同時也讓迭代次數下降非常多，實測計算時間也快了非常多倍。因此我們在比較後發現，smooth RRT 相較基本款 RRT，效率及路線皆優化不少。

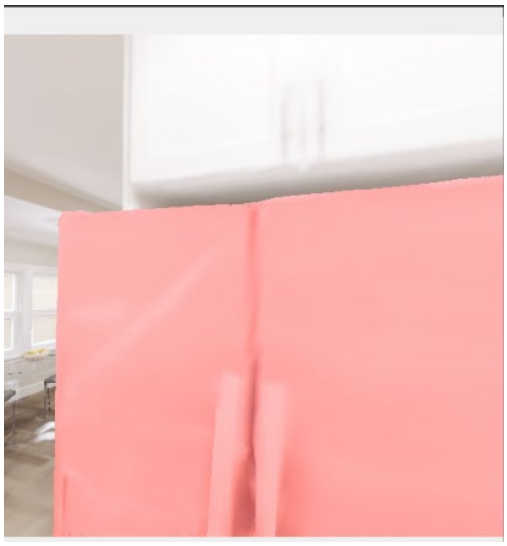
2. Bi-RRT


cooktop
connect
cushion
connect
lamp
connect
rack
connect



- 在 Bi-RRT 中，路徑一樣相較原始 RRT 平滑，不過在規劃上似乎有一點繞路的情況。
- 繞路的可能原因，也許是因為兩邊都使用隨機點生成新節點的方式去延伸隨機樹，雖然有使用到「tree 最近點往 target 方向不斷生成新節點」，但由於生成隨機點這個步驟無論在哪一個 tree 都是第一步，因此會有較多的隨機點，發生繞路的狀況才會變多，不過路徑仍是滿漂亮的，計算效率一樣相對原本 RRT 提升非常多。

3. Navigation results



- 上圖為部分導航過程的照片，包括 highlight 的部分。
- 導航過程中，agent 在每個節點都會先旋轉至下一個節點的方向，接著直走前往該節點，走一段時間後，即可抵達終點。
- 在這裡我曾遇到一個問題。由於沿著路徑移動需要有一定的準確度及較低的偏移程度，因此旋轉及直走的量值不能設太大。若角度旋轉量值太大，會發生在目標角度不斷左右轉的情形(卡住)；若直走量值太大，則可能正好跨過下個節點的位置，導致 agent 離開導航路線(失敗)。

iii. Reference

1. nimRobotics/RRT: https://github.com/nimRobotics/RRT/blob/master/rrt.py
2. Robotic Motion Planning: RRT's
3. 老師上課講義