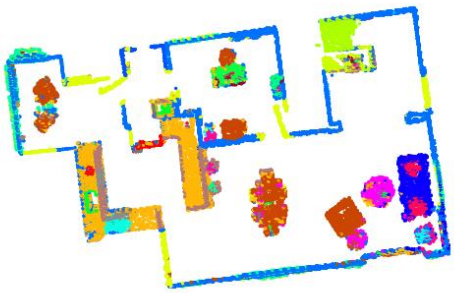
# Perception and Decision Making in Intelligent Systems

## Homework 3: A Robot Navigation Framework

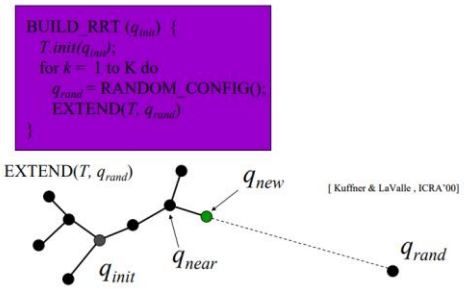
1. Code

- Create Map

  I use the source files from TA. After remove the roof and ceiling based on the y value, I scatter the colored points into a 640x480 image.



- RRT




I follow the algorithm to design my code.

- **q_rand**: random select one point in image based on the random threshold (default is 0.3). if the random number under than threshold, q_rand will be set as target location. Otherwise, it will be any point in the map image.

```
def get_random(theta):
    # random select a point as target
    if random.random()<theta:
        p_rand=p_target
    else:
        p_rand=[int(random.random()*480), int(random.random()*640)]
    return p_rand
```

- **q_near**: using the nearest neighbor with n being 1 to find the closest point to q_rand in the exist searching tree.

```
def get_nearest(tree,p_rand):
    # find a existed node which is the nearest one close to the target
    tree_nodes=np.array(tree.nodes())
    n = NearestNeighbors(n_neighbors=1,metric="euclidean")
    n.fit(tree)
    dis, ind = n.kneighbors([p_rand], return_distance=True)
    p_near=tree_nodes[ind.ravel()]
    return p_near.ravel(),dis.ravel()
```

- **q_new**: step along the direction from q_near to q_rand and create a new node. The offset 1 is to make sure that the distance between q_near and q_new is not over step length.

```
def get_new_node(p_near,p_rand,dis,step_size=step_size):
    # get p_new
    d=(p_rand-p_near)/dis*step_size

    p_new=p_near+d # one step along the direction
    p_new=p_new.ravel()

    p_new=[int(p_new[0]),int(p_new[1])]
    # this offset to make sure the distance<=step_size

    if d[0]<0:
        p_new[0]+=1
    if d[1]<0:
        p_new[1]+=1

    return p_new
```

After got the q_new, we need to check whether that point is achievable. If the new point is out of image or being colored, it is not a valid new point.

```
def check_achievable(p_near,p_new,np_map,n_sample=25):
    # check if it is achievable
    if p_new[0]>=w or p_new[0]<0 or p_new[1]>=h or p_new[1]<0:
        # out of map
        return False

    d=p_new-p_near
    d_step=d.ravel()/n_sample

    for i in range(n_sample):
        p=p_near+d_step*i
        p=[int(p[0]),int(p[1])]

        if (np_map[p[1]][p[0]]!=[1,1,1,1]).any():
            # is wall or object
            return False
    return True
```

Also, we need to check the mission completeness. If the target is close to current tree node, I directly add a new edge between searching tree and target, and then finish the job.

```
def check_mission(tree,p_target,thres_d=10):
    n = NearestNeighbors(n_neighbors=1,metric="euclidean")
    n.fit(tree)
    dis, ind = n.kneighbors([p_target], return_distance=True)
    if dis.ravel()<thres_d:
        p=np.array(tree.nodes())[ind.ravel()]
        return True,p.ravel()

    return False,None
```

After got the complete searching tree, I get the final result path by using the networks function “all_simple_paths”.

```
# start searching
search_G=RRT(p_start,p_target,np_map)
answer=nx.all_simple_paths(search_G,tuple(p_start),tuple(p_target))
```

To avoid algorithm collapse and decrease computational cost, I make the searching tree reset if it has been over 1500 iteration and does not find the target.

```
if count>=1500:
    # reset
    count=0
    G = nx.Graph()
    G.add_node(tuple(p_start))
```

- Transform from pixel to real position

  We need to convert the points in path from pixel coordinate to xyz-coordinate.

```
def transform(point):
    # (x,z) u,v
    # kitchen (-0.6427505, -1.5003817)  209,302
    # kitchen2 (1.7072494, 0.69961834)  272,235
    # wall corner (-3.0927505, 0.59961843) 275, 378
    # sofa corner (2.2072494, 9.099619)  595, 219
    # book corner (3.8072493, 5.749618)  435 171

    u, v = point

    scale_x=(2.2072494-1.7072494)/(219-235)
    scale_z=(0.69961834+1.5003817)/(272-209)

    x=(v-290)*scale_x
    z=(u-255)*scale_z

    return np.asarray([x,z])
```

- Navigation

  First need to rotate the agent to face the direction to next position. I first get the current forward direction and calculate the target direction from agent position and next point from the path.

```
def get_forward_direction():
    agent_state = agent.get_state()
    sensor_state = agent_state.sensor_states['color_sensor']
    rw, rx, ry, rz=sensor_state.rotation.w, sensor_state.rotation.x, sensor_state.rotation.y, sensor_state.rotation.z
    forward_dir=np.array([
        -2 * (rx * rz + rw * ry),
        #2*(ry*rz-rw*rx),
        2 * (rx * rx + ry * ry)-1,


        ])
    return forward_dir


def get_position():
    agent_state = agent.get_state()
    sensor_state = agent_state.sensor_states['color_sensor']
    return sensor_state.position[0],sensor_state.position[2]
```

By these two vectors, I can get the signed degree between them. If the angle is smaller than 0, the agent should turn left. Otherwise, the agent will turn right.

```python
def unit_vector(vector):
    """ Returns the unit vector of the vector.  """
    return vector / np.linalg.norm(vector)

def angle_between(v1, v2):
    """ Returns the angle in radians between vectors 'v1' and 'v2'::
        https://blog.csdn.net/hankerbit/article/details/84066629
    """
    x1,y1 = unit_vector(v1)
    x2,y2 = unit_vector(v2)


    dot=x1*x1+y1*y2
    det=x1*y2-x2*y1

    rad= np.arctan2(det,dot)
    angle=rad*  180.0 / np.pi

    return angle
```

After rounding the angle, I convert them into discrete command.

```python
# caculate angle between 2 vector
angle=angle_between(current_dir,next_dir)
if angle<0:
    action="turn_left"
else:
    action="turn_right"
round_angle=round(angle)
print(angle,round_angle,action)

for t in range(abs(round_angle)):
    navigateAndSee(action,videowriter)
    cv2.waitKey(1)
```

And also transform the distance from current position to next location into discrete command.

```python
for t in range(round(dis/0.01)):
    navigateAndSee("move_forward",videowriter)
    cv2.waitKey(1)
```

- Recording

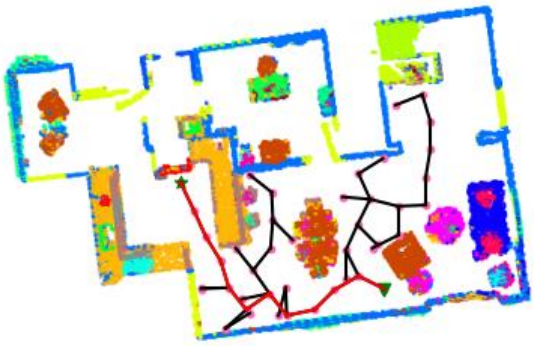
  I use Opencv VideoWriter to create the video.

2. Result

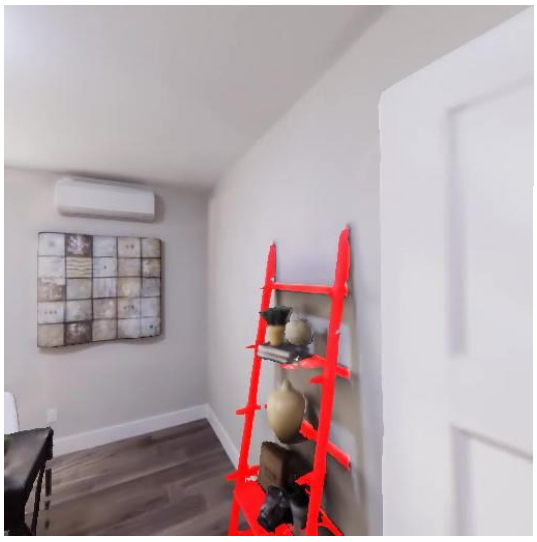
   Green tringle is the start point and the green star is the target location.

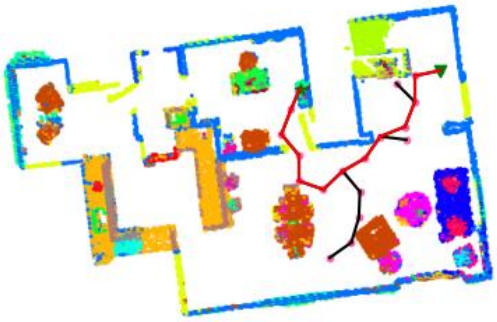
   Black line is searching path and red line means the optimal trajectory.

- Refrigerator

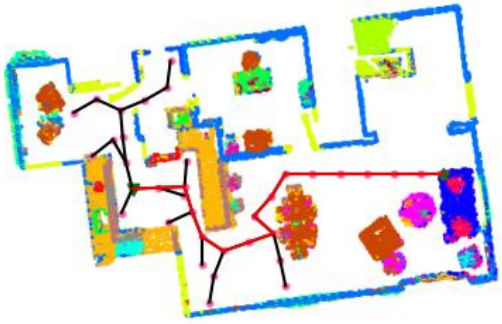
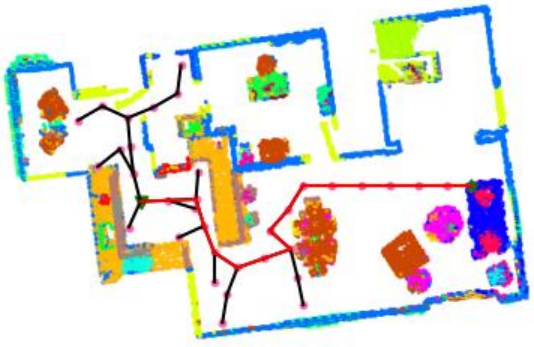


- Rack

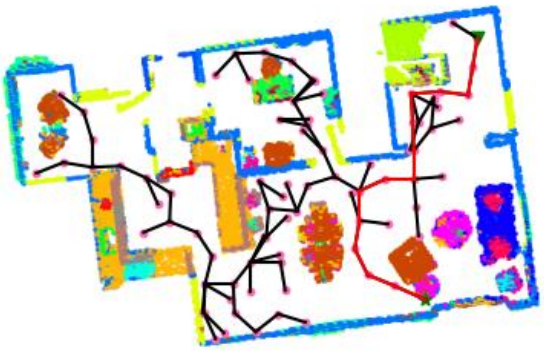





- Cushion

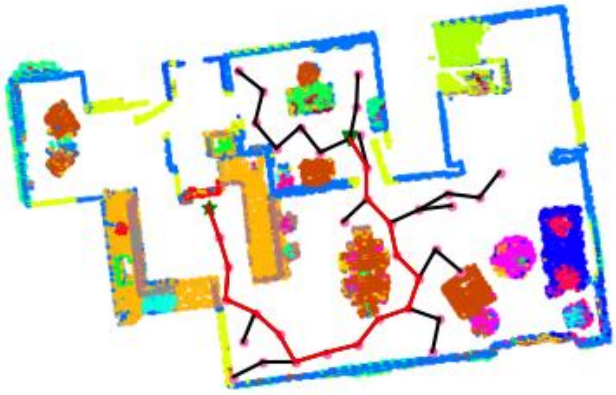

- Lamp



- Cooktop

3. Discussion

This RRT algorithm relies on chance. Sometime can achieve target easily and quickly and sometime being trapped. Here are the examples to searching refrigerator.



Another thing to note is that: because there are some bias between ideal position and current agent position caused by discrete command, the target forward direction should from current position rather than from the past point record in path.