Intelligent Robot Systems

Athanasios Manolis (8856), Evangelos Zikopis (8808)

July 20, 2020

Abstract

Report regarding the assignment for the course **Intelligent Robot Systems**. The general goal is to develop specific modules of an autonomous simulated vehicle that performs full exploration and coverage of an unknown a priori environment.

1 Laser-based obstacle avoidance

```
leng = len(scan)    # Number of scans
angle_min = self.laser_aggregation.angle_min    # Minimum angle scanned (rad)
angle_increment = self.laser_aggregation.angle_increment  # Angle between
                                        different scans (rad)

# Calculate the effect every single scan has on the path that the robot is
# going to follow. The closer it is, the bigger the effect. We use sin and
# cos so that we can get positive or negative values, in order for the robot
# to move front/back and right/left
for i in range(0, leng):
    linear -= math.cos(angle_min + i*angle_increment) / (scan[i]*scan[i])
    angular -= math.sin(angle_min + i*angle_increment) / (0.5*scan[i]*scan[i])

# Get the average value of all scans' effect
linear = 0.3 + linear / leng  # In this case add it to something constant
angular = angular / leng

# Both speeds must have values in the range [-0.3, 0.3]
linear = min(max(linear, -0.3), 0.3)
angular = min(max(angular, -0.3), 0.3)
```

For each of the laser scans, we calculate a value which, essentially, is their effect on the robot's velocity. In order to find the optimal angular velocity, we calculate a positive or negative value, so that the robot goes left or right respectively.

The expression "angle_min+i*angle_increment" is used in order to get the angle transformation of each scan in the interval [-2.094, 2.094], while the trigonometric functions are used to get the appropriate sign. We also divide by $scan[i]^2$ so that we apply the effect of the distance from the obstacle (smaller if it is far or bigger if it is close). The coefficient 0.5 in variable "angular" was selected after tests aiming at smooth changes in the rotational speed of the robot.

Finally, we add the value we found for the linear velocity to 0.3, which we want to be the maximum value. We noticed that if we used a higher value (eg 0.5 or 0.6) then the robot would explore a larger part of the map, but it seemed that in reality it did not have enough time to turn where it should, so it would come very close to obstacles and therefore the speed would of become zero, resulting on an on-site turn. The choice of the value 0.3 seemed to offer the best balance between angular and linear velocity.

2 Path visualization

```
# Multiply subgoal's coordinates with resolution and add robot.origin in order
# to translate between the (0,0) of the robot pose and (0,0) of the map
ps.pose.position.x = p[0] * self.robot_perception.resolution + self.
                                        robot_perception.origin['x']
ps.pose.position.y = p[1] * self.robot_perception.resolution + self.
                                        robot_perception.origin['y']
```

This task is about visualizing the path's subgoals on the map. In order to implement it, firstly we multiply each subgoal's coordinates with the resolution of the map. Secondly, in order to make a translation between the coordinates of the robot pose and the global coordinates of the map we add the robot's origin to the previous multiplication result. Those two steps are performed for both 'x' and 'y' coordinates since we have a two-dimensional map. Finally, subgoal's coordinates are ready to be visualized on the map with a marker.

3 Path following

```
# Angular and linear velocities are calculated upon the math
# types presented at 9.exploration_target_selection.pdf

dtheta = math.degrees(math.atan2(st_y - ry, st_x - rx)) - math.degrees(theta)

if dtheta >= 0 and dtheta < 180:
    angular = dtheta / 180
elif dtheta > 0 and dtheta >= 180:
    angular = (dtheta - 2 * 180) / 180
elif dtheta <= 0 and dtheta > -180:
    angular =  dtheta / 180
elif dtheta < 0 and dtheta < -180:
    angular = (dtheta + 2 * 180) / 180

# Should be angular*0.3 but then the robot moves very slow and
# cannot reach the targets within the time set from the timer
if angular > 0.3:
    angular = 0.3
elif angular < -0.3:
    angular = -0.3

# **20 is used to avoid overshoot problem
linear = ( (1 - abs(angular))**20 ) * 0.3
```

This task is about producing the correct velocities for the robot to follow the produced path. The thinking behind our approach is depicted in the next figure and it was based upon the course's notes.

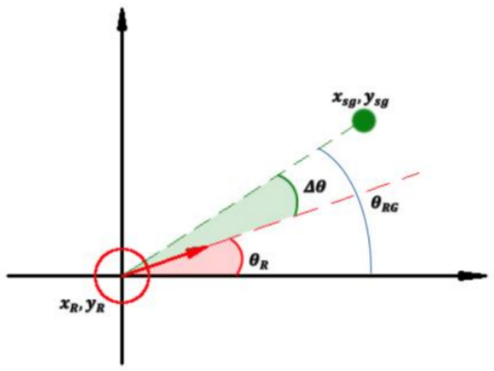

Figure 1: Angle difference

The first step of our method calculates the angle difference between the robot's orientation and the desired sub-target's position. This angle difference is stored in the variable *dtheta*. The second step concerns the calculation of the robot's angular (or rotational) speed and it is implemented via an if statement. In this if statement the angular speed is given a value based upon the angle difference between the robot and the sub-target. After the angular speed calculation, the linear speed is given from the formula

$$u = ((1 - |\omega|)^{20}) \cdot 0.3$$

The power factor is used to prevent the robot from "overshooting" the desired target. Finally, both linear and angular speeds are kept in [-0.3, 0.3]

4 Path following & obstacle avoidance

```
# We use a motor schema in order to combine velocities from
# obstacle avoidance and path following procedure
if l_laser == 0:
    c_l = 0
    c_a = 1
else:
    c_l = 0.2
    c_a= 0.2

self.linear_velocity = l_goal + c_l*l_laser
self.angular_velocity = a_goal + c_a*a_laser
```

We use a motor schema in order to combine velocities from obstacle avoidance and path following procedure. The values of the coefficients c_l and c_a were chosen after experiments. Although, if the linear velocity is 0 (meaning that there are many obstacles around), then the coefficient for the linear velocity becomes 0 and the coefficient for the angular velocity becomes 1, meaning that the robot will move neither forward nor backwards.

5 Smarter subgoal checking

```
# If the robot is in the starting point it immediately sets the next subtarget
if self.next_subtarget == 0:
    self.next_subtarget += 1
    self.counter_to_next_sub = self.count_limit
else:
    # Check if there is a later subtarget, closer than the next one
    # If one is found, make it the next subtarget and update the time
    for i in range( self.next_subtarget + 1, len(self.subtargets) - 1 ):
        # Find the distance between the robot pose and the later subtarget
        dist_from_later = math.hypot(\
            rx - self.subtargets[i][0], \
            ry - self.subtargets[i][1])
        if dist_from_later < 15:
            self.next_subtarget = i
            self.counter_to_next_sub = self.count_limit + 100
            dist = dist_from_later
            break

    # If distance from subtarget is less than 5, go to the next one
    if dist < 5:
        self.next_subtarget += 1
        self.counter_to_next_sub = self.count_limit
        # Check if the final subtarget has been approached
        if self.next_subtarget == len(self.subtargets):
            self.target_exists = False
```

The main idea of this solution is for the robot to decide to head to a later sub-target, without having to first reach the one that it is trying to reach at the moment. This will happen in 2 cases:

- If a later sub-target is closer than the next one, then we make it the next sub-target, and we update the time provided to reach it
- If the distance from the next sub-target is less than 5, then we immediately go to the later one, without having to reach it first

6 Smart target selection

```
# Get the robot pose in pixels
[rx, ry] = [\
    robot_pose['x_px'] - \
            origin['x'] / resolution,\
    robot_pose['y_px'] - \
            origin['y'] / resolution\
            ]
g_robot_pose = [rx,ry]
# If nodes > 25 the calculation time-cost is very big
# In order to avoid time-reset we perform sampling on
# the nodes list and take a half-sized sample
for i in range(0,len(nodes)):
    nodes[i].append(i)
if (len(nodes) > 25):
    nodes = random.sample(nodes, int(len(nodes)/2))

# Initialize weigths
w_dist = [0]*len(nodes)
w_rot = [robot_pose['th']]*len(nodes)
w_cov = [0]*len(nodes)
w_topo = [0]*len(nodes)
# Calculate weights for every node in the topological graph
for i in range(0,len(nodes)):
    # If path planning fails then give extreme values to weights
    path = self.path_planning.createPath(g_robot_pose, nodes[i], resolution)
    if not path:
        w_dist[i] = sys.maxsize
        w_rot[i] = sys.maxsize
        w_cov[i] = sys.maxsize
        w_topo[i] = sys.maxsize
    else:
        for j in range(1,len(path)):
            # Distance cost
            w_dist[i] += math.hypot(path[j][0] - path[j-1][0], path[j][1] -
                                                       path[j-1][1])
            # Rotational cost
            w_rot[i] += abs(math.atan2(path[j][0] - path[j-1][0], path[j][1] -
                                                        path[j-1][1]))
            # Coverage cost
            w_cov[i] += coverage[int(path[j][0])][int(path[j][1])] / (len(path
                                                       ))
        # Add the coverage cost of 0-th node of the path
        w_cov[i] += coverage[int(path[0][0])][int(path[0][1])] / (len(path))
        # Topological cost
        # This metric depicts wether the target node
        # is placed in open space or near an obstacle
        # We want the metric to be reliable so we also check node's neighbour
                                                   cells
        w_topo[i] += brush[nodes[i][0]][nodes[i][1]]
        w_topo[i] += brush[nodes[i][0]-1][nodes[i][1]]
        w_topo[i] += brush[nodes[i][0]+1][nodes[i][1]]
        w_topo[i] += brush[nodes[i][0]][nodes[i][-1]]
        w_topo[i] += brush[nodes[i][0]][nodes[i][+1]]
        w_topo[i] += brush[nodes[i][0]-1][nodes[i][1]-1]
        w_topo[i] += brush[nodes[i][0]+1][nodes[i][1]+1]
        w_topo[i] = w_topo[i]/7

# Normalize weights between 0-1
for i in range(0,len(nodes)):
    w_dist[i] = 1 - (w_dist[i]-min(w_dist))/(max(w_dist)-min(w_dist))
    w_rot[i] = 1 - (w_rot[i]-min(w_rot))/(max(w_rot)-min(w_rot))
    w_cov[i] = 1 - (w_cov[i]-min(w_cov))/(max(w_cov)-min(w_cov))
    w_topo[i] = 1 - (w_topo[i]-min(w_topo))/(max(w_topo)-min(w_topo))

# Set cost values
# We set each cost's priority based on experimental results
# from "Cost-Based Target Selection Techniques Towards Full Space
# Exploration and Coverage for USAR Applications
# in a Priori Unknown Environments" publication
C1 = w_topo
C2 = w_dist
C3 = [1]*len(nodes)
for i in range(0,len(nodes)):
    C3[i] -= w_cov[i]
C4 = w_rot
# Priority Weight
C_PW = [0]*len(nodes)
# Smoothing Factor
C_SF = [0]*len(nodes)
# Target's Final Priority
C_FP = [0]*len(nodes)
for i in range(0,len(nodes)):
    C_PW[i] = 2**3*(1-C1[i])/.5 + 2**2*(1-C2[i])/.5 + 2**1*(1-C3[i])/.5 + 2**0
                                            *(1-C4[i])/.5
    C_SF[i] = (2**3*(1-C1[i]) + 2**2*(1-C2[i]) + 2**1*(1-C3[i]) + 2**0*(1-C4[i
                                            ]))/(2**4 - 1)
    C_FP[i] = C_PW[i]*C_SF[i]

# Select the node with the smallest C_FP value
val, idx = min((val, idx) for (idx, val) in enumerate(C_FP))
target = nodes[idx]
```

This task is about finding a smart target selection solution. Our approach aims at covering the full map as fast (and smart) as possible in a Priory Unknown Environments. Our approach is based on the results and novelties presented in [1]. The selected method, referred as "minimum cost topological node method" examines and scores every node from the topological graph. If the nodes are more than 25 we take a sample of them, in order to reduce the time wasted in path calculations (more than 1-2 seconds for every node). We construct the path from the robot to every node using A* algorithm. Each nodes' score is then calculated based on the next weights:

- Distance weight (w_dist): The total distance weight is the sum of distances between every sub-path's neighbor node

- Rotation weight (w_rot): The total rotation weight is the sum of the angle difference between the neighbor nodes of the sub-path
- Coverage weight (w_cov): Total coverage weight is the sum of coverage value of every node in the sub-path divided by the length of the path
- Topological weight (w_topo): The topological weight is representing the presence of obstacles or boundaries near the target. In order to calculate the weight, we examine the brushfire field value of the target and its neighbors. The final value is the mean value of the above

After the calculation of the weight values for every node, the values are normalized between 0 and 1. Next, we assign the weights to the cost variables C1-C4. The weights are assigned with a chosen priority. We chose to assign w_topo to C1, since we want our robot to move close to the boundaries of the map. In this way the robot will be moving near the perimeter of the map and the centered free area will be probably covered. This is why we don't select targets located in free areas. Next, w_dist is assigned to C2 since we prefer the robot to catch targets close to its position. Like this we will avoid traversing revisited areas and target selection will be executed more often. At last, w_cov is assigned to C3 and w_rot is assigned to C4. Next, a priority weight (C_PW) and a smoothing factor (C_SF) are calculated based on the values of C1-C4 for every node. The final priority for every node is calculated as C_FP = C_PW*C_SF. The node with the lowest final priority is selected as the next target.

In figures 2 and 3 shown below, we can notice that the smart target approach is much better than the random one. The total time elapsed in order to get the same map coverage result may be similar in both methods. However ,it is clear that in the smart selection method the total distance traversed is way smaller.

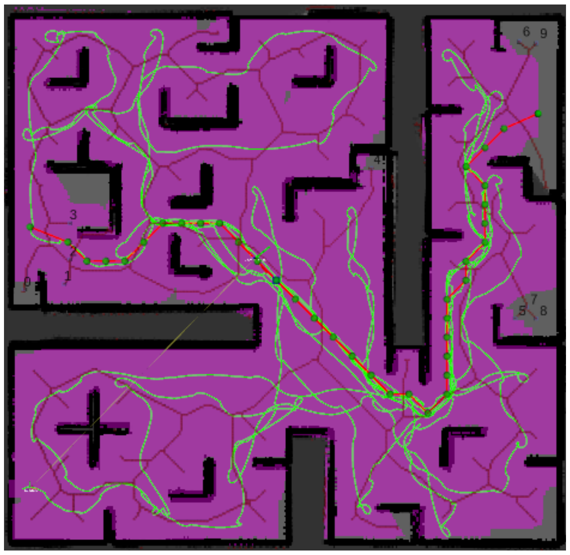
Figure 2: Random target selection

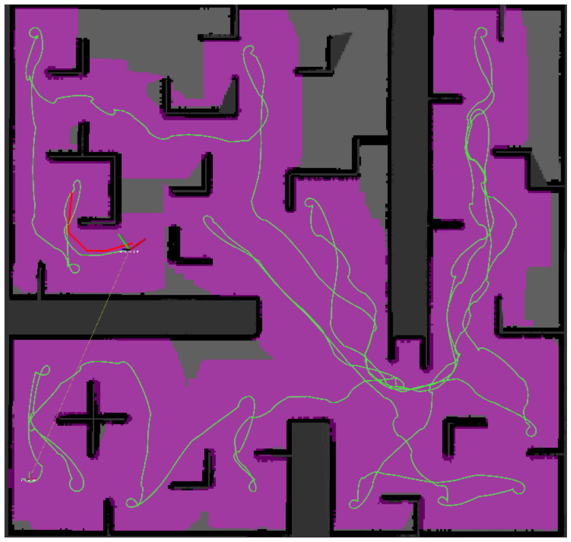
Figure 3: Smart target selection

7 Path optimization / alteration

```
weight_data = 0.5   # How much weight to update the data (a)
weight_smooth = 0.1 # How much weight to smooth the coordinates (b)
min_change = 0.0001 # Minimum change per iteration to keep iterating
new_path = np.copy(np.array(self.path))
path_length = len(self.path[0])
change = min_change

while change >= min_change:
    change = 0.0
    for i in range(1, len(new_path)-1):
        for j in range(path_length):
            # Initialize x, y
            x_i = self.path[i][j]
            y_i = new_path[i][j]
            y_prev = new_path[i-1][j]
            y_next = new_path[i+1][j]

            y_i_saved = y_i

            # Minimize the distance between coordinates of the original
            # path (y) and the smoothed path (x). Also minimize the
            # difference between the coordinates of the smoothed path
            # at time step i, and neighboring time steps. In order to do
            # all the minimizations, we use Gradient Ascent.
            y_i += weight_data * (x_i-y_i) + weight_smooth * (y_next + y_prev
                                               - (2*y_i))
            new_path[i][j] = y_i

            change += abs(y_i - y_i_saved)

self.path = new_path
```

In this task, we follow the method indicated at lecture 9 (also linked to the recommended Udacity course on AI in Robotics). Y represents the new, smoothed path, while X represents original path. In each iteration we minimize:

- the distance between the coordinates of the original path and the smoothed path
- the difference between the coordinates of the smoothed path at time step i, and neighboring time steps

In order to minimize all these values simultaneously, we use Gradient Ascent. We adjust the importance of the smoothing of y__i with respect to the un-smoothed trajectory by the weight 'a', and the importance of the smoothing of y__i with respect to neighboring smoothed coordinates, by the weight 'b'.

The iterations will keep going, only while the change to be done has a value of at least 0.0001.

References

[1] Tsardoulias, E.G., Iliakopoulou, A., Kargakos, A. et al. Cost-Based Target Selection Techniques Towards Full Space Exploration and Coverage for USAR Applications in a Priori Unknown Environments. J Intell Robot Syst 87, 313–340 (2017)