Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης
(Α.Π.Θ.)

Μάθημα ΗΛ0701: Ευφυή Συστήματα Ρομπότ

Αναφορά Εργασίας Εξαμήνου στο μάθημα Ευφυή Συστήματα Ρομπότ

Αντωνιάδης Δημήτριος
akdimitri@auth.gr
8462

23 Ιανουαρίου 2020

Περιεχόμενα

1 Εισαγωγή

Το παρόν έγγραφο αποτελεί την αναφορά της εργασίας εξαμήνου που πραγματοποιήθηκε στο πλαίσιο του μαθήματος **Ευφυή Συστήματα Ρομπότ**.

2 Υλοποίηση Εργασίας.

2.1 Challenge 1: Laser-based obstacle avoidance.

Για το ερώτημα 1 ζητείται να συμπληρωθεί κώδικας έτσι ώστε το σύστημα να υπολογίζει τη γραμμική (linear) και τη γωνιακή (rational) ταχύτητα (velocity) χρησιμοποιώντας τις LIDAR τιμές. Στόχος είναι το ρομπότ να περιπλανάται (wander) χωρίς να συγκρούεται (collide) σε εμπόδια.

Από το μάθημα είναι γνωστό ότι:

$$u_{obs} = -\sum_{i=1}^{LaserRays} \frac{cos(\theta_i)}{s_i^2} \tag{1}$$

και

$$\omega_{obs} = -\sum_{i=1}^{LaserRays} \frac{sin(\theta_i)}{s_i^2} \tag{2}$$

όπου θ_i η γωνία του Laser και s_i η απόσταση.

Επομένως στη μέθοδο produceSpeedsLaser έγιναν οι παρακάτω προσθήκες διασφαλίζοντας ότι η μέγιστη γραμμική ταχύτητα θα είναι $0.3m/s$ και η μέγιστη γωνιακή $0.3rad/s$:

```
# Challenge 1     edit: Dimitrios Antoniadis      date: 22/01/2020

# Calculate Number of Measurements
N_measurements = len(scan)

# Initialize angles Vectors
angles = [0 for i in range(N_measurements)]

# Calculate Angles
for x in range(N_measurements):
  angles[x] = math.radians(-135 + x*270/(N_measurements - 1))

# Calculate Linear Speeds based on presentation 9
for x in range(N_measurements):
  linear = linear -(math.cos(angles[x]) / scan[x]**2)

# Calculate Angular Speeds based on presentation 9
for x in range(N_measurements):
  angular = angular - (math.sin(angles[x]) / scan[x]**2)
```

Στη συνέχεια, στη μέθοδο produceSpeeds έγιναν οι παρακάτω προσθήκες:

```
# Challenge 1     edit: Dimitrios Antoniadis      date: 23/01/2020

l_laser = l_laser + 100      # add sth constant to move
a_laser = a_laser/300

# The robot must have a maximum absolute linear speed of 0.3 m/s
# and maximum absolute rotational speed 0.3 rad/sec.
if abs(l_laser) > 0.3:  # Correction if absolute linear is greater than 0.3 m/s
    if l_laser > 0:
        self.linear_velocity = 0.3
    else:
        self.linear_velocity = -0.3
else:
    self.linear_velocity = l_laser

if abs(a_laser) > 0.3:   # Correction if absolute angular is greater than 0.3 rad/sec
    if a_laser > 0:
        self.angular_velocity = 0.3
    else:
        self.angular_velocity = -0.3
else:
  self.angular_velocity = a_laser
```

2.2 Challenge 2: Path visualization.

Ο σκοπός του ερωτήματος αυτού ήταν να γίνει εμφανές το path στο RViz tool. Συνεπώς, έγιναν οι εξής προσθήκες selectTarget:

```
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']
```

Ουσιαστικά, πραγματοποιείται πολλαπλασιασμός του σημείου του path με το resolution και στη συνέχεια προστίθεται το origin.

2.3 Challenge 3: Path following.

Ο σκοπός του ερωτήματος αυτού είναι να παραχθούν οι κατάλληλες ταχύτητες ώστε το ρομπότ να ακολουθήσει το path. Από το μάθημα γνωρίζω ότι αν ξέρω την κατεύθυνση του ρομπότ και τη γωνία του αντικειμένου μπορώ να βρω τη σχετική μεταξύ τους γωνία και κατ΄ επέκταση τη ζητούμενη γωνιακή ταχύτητα ή:

$$\omega = \begin{cases} \frac{\Delta\theta+2\pi}{\pi}, \text{ αν } \Delta\theta < -\pi \\ \frac{\Delta\theta}{\pi}, \text{ αν } -\pi \leq \Delta\theta \leq \pi \\ \frac{\Delta\theta-2\pi}{\pi}, \text{ αν } \Delta\theta > \pi \end{cases} \tag{3}$$

Τώρα, η τιμή της γραμμικής ταχύτητας υπολογίζεται ως εξής:

$$linear_u = u_{max}(1 - |\omega|)^n \iff u = 0.3(1 - |\omega|)^n, \tag{4}$$

όπου μετά από προσομοιώσεις επιλέχθηκε $n = 6$.

Ακόμη, επειδή η γωνιακή ταχύτητα δεν ήταν αρκετά μεγάλη και δεν έστριβε γρήγορα το ρομπότ, επιλέχθηκε να τροποποιείται ως εξής: $angular_u = 0.3 * sgn(\omega)\omega^6$

Από τα παραπάνω γίνεται εμφανές ότι οι ταχύτητες ήταν ορισμένες στο διάστημα $[-0.3, 0.3]$.

Συμπληρώθηκε, λοιπόν, η μέθοδος velocitiesToNextSubtarget με:

```python
    # Challenge 3       edit: Dimitrios Antoniadis date 22/01/2020

if self.subtargets and self.next_subtarget <= len(self.subtargets) - 1:
    st_x = self.subtargets[self.next_subtarget][0]
    st_y = self.subtargets[self.next_subtarget][1]

    # We know goals position (x', y') and robot's position (x,y)
    # It is tan(Theta_RG) = (y'-y)/(x'-x) = lamda
    # Theta_RG = atan(lamda)
    # Theta_RG is the angle between vector RobotGoal and Ox axis
    Theta_RG = math.atan2(st_y - ry, st_x - rx)     # atan2 return the result in rads

    # We can calculate the angle between the direction of robot's movement
    # and the RG vector. The angle will be the difference of RGangle - theta(direction angle of robot's movement)
    D_Theta = Theta_RG - theta

    # Based on presentation 9, D_Theta
    # has to be readjusted in [-pi, pi]
    if (D_Theta < math.pi) and (D_Theta > -math.pi):
        omega = D_Theta / math.pi
    elif D_Theta >= math.pi:
        omega = (D_Theta - 2 * math.pi) / math.pi
    elif D_Theta <= -math.pi:
        omega = (D_Theta + 2 * math.pi) / math.pi

    # Linear Speed Calculation
    # presentation 9: u = umax(1- |omega|)^n
    # larger n -> lower speed
    # max speed = 0.3
    u = ((1 - abs(omega))**(6.00))
    linear = 0.3 * u

    # Robot is steering slowly
    # therefore we had to make steer faster
    # max speed = 0.3
    omega = (math.copysign(abs(omega)**(1.0/6), omega))
    angular = 0.3 * omega
```

2.4 Challenge 4: Path following & obstacle avoidance.

Στο ερώτημα αυτό σκοπός είναι να ακολουθεί το ρομπότ μία διαδρομή αποφεύγοντας τα εμπόδια. Οι εξισώσεις που επιλέχθηκαν είναι:

$$u = u_{path} + c_u * u_{obs}$$

$$\omega = \omega_{path} + c_\omega * \omega_{obs}$$

προσδιορίζουμε, λοιπόν, τις τελικές ταχύτητες ως εξής:

$$u = l_{goal} + l_{laser} * c1$$
$$\omega = a_{goal} + a_{laser} * c2$$

$$u = min(0.3, max(-0.3, u))$$
$$\omega = min(0.3, max(-0.3, \omega))$$

όπου $c_1 = 10^{-5}$ και $c2 = 10^{-5}$. Οι παράμετροι αυτοί προέκυψαν ύστερα από διάφορα πειράματα.

Οι προσθήκες που έγιναν στη μέθοδο produceSpeeds είναι οι εξής:

```
# Challenge 4 edit: Dimitrios Antoniadis date: 22/01/2020

# Based on presentation 9, we know
# that in a motor schema u = upath + c*u_obs
# and omega = omega_path + c*omega_obs
c1 = 10**(-5)
c2 = 10**(-5)
self.linear_velocity  = l_goal + l_laser * c1
if self.linear_velocity == 0: # just in case it stops
    self.linear_velocity = -0.05

self.angular_velocity = a_goal + a_laser * c2

# Make sure speeds are on the range [-3,3]
self.linear_velocity = min(0.3, max(-0.3, self.linear_velocity))
self.angular_velocity = min(0.3, max(-0.3, self.angular_velocity))
```

2.5 Challenge 5: Smarter subgoal checking.

Για το ερώτημα αυτό επιλέχθηκε να ελέγχεται η απόσταση από όλα τα επόμενα σημεία με μία πιο χαλαρή συνθήκη. Αν το ρομπότ βρίσκεται σχετικά σε μία κοντινή απόσταση από έναν στόχο, θεωρεί ότι τον έχει επιτύχει και συνεχίζει. Αν ακόμη παρατηρήσει ότι βρίσκεται κοντύτερα από κάποιο επόμενο στόχο (πάντα για μικρή απόσταση) προχωράει στον επόμενο στόχο. Ο μόνος στόχος για τον οποίο η συνθήκη απόσταση είναι αυστηρή είναι ο τελευταίος στόχος.

Έτσι η μέθοδος checkTarget τροποποιήθηκε ως εξής:

```
# Challenge 5 edit: Dimitrios Antoniadis

# Instead of checking the distance of the next target,
# check the distance of the remaining targets, therefore you may reach to a next subject
# bypassing current.

for i in range(self.next_subtarget, len(self.subtargets)):
    # Find the distance between the robot pose and the next subtarget
    dist = math.hypot( rx - self.subtargets[i][0], ry - self.subtargets[i][1])

    # check distance with the i_th target
    if i != (len(self.subtargets)-1):
        if dist < 10:        #if distance found to be small from a terget set as next the target i + 1
            self.next_subtarget = i + 1
            self.counter_to_next_sub = self.count_limit
    else:
        if dist < 5:         #if distance found to be small from a terget set as next the target i + 1
            self.next_subtarget = i + 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
```

2.6 Challenge 6: Smart target selection.

Το ερώτημα αυτό αφορά την έξυπνη επιλογή ενός στόχου. Η μεθοδολογία που ακολουθήθηκε είναι αυτή που περιγράφεται στην παρουσίαση 9. Αρχικά, υπολογίζεται το path για κάθε node. Στη συνέχεια, προκειμένου να γίνει η ¨έξυπνη¨ επιλογή πρέπει να εισάγουμε κάποια μετρική-κόστος, έτσι υπολογίζεται το κόστος της απόστασης:

$$w_{dist} = \sum_{i=1}^{PathSize-1} D_{i,i+1}$$

Επιπλέον, υπολογίζεται το τυπολογικό κόστος με τη χρήση της παρακάτω εξίσωσης:

$$w_{topo} = brush(node)$$

όπου $brush$ η τιμή του brushfire στο σημείο του στόχου.

Ένα ακόμη κόστος που πρέπει να υπολογιστεί είναι το κόστος της περιστροφής και αυτό γιατί δεν επιθυμούμε το ρομπότ να κάνει πολλές στροφές αλλά να ακολουθεί μία πιο ομαλή-ευθεία πορεία.

$$w_{turn} = \sum_{i}^{PathSize} \theta i$$

Τελευταίο κόστος το οποίο πρέπει να υπολογιστεί είναι το κόστος κάλυψης το οποίο θα ισούται με:

$$w_{cove} = 1 - \frac{\sum_{i=1}^{PathSize} Coverage[x_{p_i}, y_{p_i}]}{PathSize * 255}$$

Αφού υπολογιστούν όλα τα παραπάνω κόστη, ακολουθεί κανονικοποίηση:

$$w_{dist_n}^k = 1 - \frac{w_{dist}^k - min(w_{dist})}{max(w_{dist}) - min(w_{dist})}$$

$$w_{turn_n}^k = 1 - \frac{w_{turn}^k - min(w_{turn})}{max(w_{turn}) - min(w_{turn})}$$

$$w_{cove_n}^k = 1 - \frac{w_{cove}^k - min(w_{cove})}{max(w_{cove}) - min(w_{cove})}$$

$$w_{topo_n}^k = 1 - \frac{w_{topo}^k - min(w_{topo})}{max(w_{topo}) - min(w_{topo})}$$

Τελικά, χρησιμοποιείται η ακόλουθη έκφραση για το τελικό κόστος:

$$W = 2^3 * w_{topo} + 2^2 * w_{dist} + 2 * w_{cove} + w_{turn}$$

Προκειμένου να γίνουν όλα τα παραπάνω προστέθηκε στο αρχείο autonomous_expl.yaml η παρακάτω γραμμή κώδικα:

```
target_selector: 'smart'        # add smart target selector Challenge 6
```

Στη συνέχεια στο αρχείο target_selection.py δημιουργήθηκε η μέθοδος selectSmartTarget η οποία υλοποιεί όλη τη διαδικασία που περιγράφηκε παραπάνω.

```
    # Challenge 6. select Smart Target Function
    # this function follows the methodology presented
    # on lecture 9.
    def selectSmartTarget(self, coverage, brush, robot_pose, resolution, origin, nodes):
        tinit = time.time()

        # Get the robot pose in pixels
        [rx, ry] = [int(round(robot_pose['x_px'] - origin['x'] / resolution)), int(round(robot_pose['y_px'] - origin['y'] / resolution))]

        # Initialize weights matrix
        weights = []

        # Do procedure described in presentation 9
        # for each node.
        for i, node in enumerate(nodes):

            # Calculate the path
            path = np.flipud(self.path_planning.createPath([rx, ry], node, resolution))

            # Check if it found a path
            if path.shape[0] > 2:
                # Vectors of the path
                vectors = path[1:, :] - path[:-1, :]

                # Calculate paths weighted distance
                vectorsMean = vectors.mean(axis=0)
                vectorsVar = vectors.var(axis=0)
                dists = np.sqrt(np.einsum('ij,ij->i', vectors, vectors))
                weightCoeff = 1 / (1 - np.exp(-np.sum((vectors - vectorsMean)**2 / (2 * vectorsVar), axis=1)) + 1e-4)
                weightDists = np.sum(weightCoeff + dists)

                # Topological weight
                weightTopo = brush[node[0], node[1]]

                # Cosine of the angles
                c = np.sum(vectors[1:, :] * vectors[:-1, :], axis=1) / np.linalg.norm(vectors[1:, :], axis=1) / np.linalg.norm(vectors[:-1, :], axis=1)

                # Sum of all angles
                weightTurn = np.sum(abs(np.arccos(np.clip(c, -1, 1))))

                # Calculate the coverage weight
                pathIndex = np.rint(path).astype(int)
                weightCove = 1 - np.sum(coverage[pathIndex[:, 0], pathIndex[:, 1]]) / (path.shape[0] * 255)

                weights.append([i, weightDists, weightTopo, weightTurn, weightCove])


        if len(weights) > 0:
            weight = np.array(weights)

            # Normalize the weights at [0,1]
            weight[:,1:] = 1 - ((weight[:,1:] - np.min(weight[:,1:], axis=0)) / (np.max(weight[:,1:], axis=0) - np.min(weight[:,1:], axis=0)))

            # Calculatete the final weights
            finalWeights = 8 * weight[:, 2] + 4 * weight[:, 1] + 2 * weight[:, 4] + weight[:, 3]

            # Find the best path
            index = int(weight[max(xrange(len(finalWeights)), key=finalWeights.__getitem__)][0])

            target = nodes[index]

            Print.art_print("Smart target selection time: " + str(time.time() - tinit), Print.ORANGE)

            return target
        else:
            Print.art_print("Smart target selection failed!!! Time: " + str(time.time() - tinit), Print.ORANGE)

            return None
```

Τώρα, προκειμένου να κληθεί η παραπάνω συνάρτηση έπρεπε να προστεθεί και ένα μικρό κομμάτι στη συνάρτηση selectTarget όπως φαίνεται παρακάτω:

```
    # Challenge 6. Smart point selection demands autonomous_expl.yaml->target_selector: 'smart'
    # Smart point selection
    if self.method == 'smart' and force_random == False:
        nextTarget = self.selectSmartTarget(coverage, brush, robot_pose, resolution, origin, nodes)

        # Check if selectSmartTarget found a target
        if nextTarget is not None:
```

```
            # Check if the next target is the same as the previous
            dist = math.hypot( nextTarget[0] - self.previous_target[0], nextTarget[1] - self.previous_target[1])
            if dist > 5:
                target = nextTarget
            else:
                target = self.selectRandomTarget(ogm, coverage, brush, ogm_limits)
        else:
            # No target found. Choose a random
            target = self.selectRandomTarget(ogm, coverage, brush, ogm_limits)


    self.previous_target = target
    return target
```

2.7 Extra Challenge 1: Path optimization / alteration.

Προκειμένου να γίνει η διαδρομή πιο ομαλή επιλέχθηκε να εφαρμοσθεί η τεχνική Minimization via Gradient descent όπως αυτή περιγράφεται στην παρουσίαση 8. Δηλαδή, αν X το αρχικό μονοπάτι, x_i ένα σημείο του πρώτου μονοπατιού και Y μία δεύτερη καμπύλη, y_i ένα σημείο του δεύτερου μονοπατιού αντίστοιχα, τότε προκειμένου να ελαχιστοποιήσω τις:

$$f = (x_i - y_i)^2$$
$$g = (y_i - y_{i+1})^2$$

χρησιμοποιώ τον Gradient Descent:

$$GD : y_i = y_i + a * (x_i - y_i) + b * (y_{i+1} - 2y_i + y_{i-1}), \text{ με } 1 \leq i \leq N - 1$$

έως ότου το παρακάτω άθροισμα να συγκλίνει:

$$\sum_{i=1}^{N-1} a * (x_i - y_i) + b * (y_{i+1} - 2y_i + y_{i-1}) < 10^{-3}$$

Η παραπάνω διαδικασία πραγματοποιήθηκε με την προσθήκη μέρος κώδικα στη συνάρτηση selectTarget:

```
##################################################
# Extra Challenge 1
# Smooth path
if len(self.path) > 3:
    x = np.array(self.path)
    y = np.copy(x)
    a = 0.5
    b = 0.1

    epsilon = np.sum(np.abs(a * (x[1:-1, :] - y[1:-1, :]) + b * (y[2:, :] - 2*y[1:-1, :] + y[:-2, :])))

    while epsilon > 1e-3:
        y[1:-1, :] += a * (x[1:-1, :] - y[1:-1, :]) + b * (y[2:, :] - 2*y[1:-1, :] + y[:-2, :])

        epsilon = np.sum(np.abs(a * (x[1:-1, :] - y[1:-1, :]) + b * (y[2:, :] - 2*y[1:-1, :] + y[:-2, :])))

    # Copy the smoother path
    self.path = y.tolist()
###################################################
```