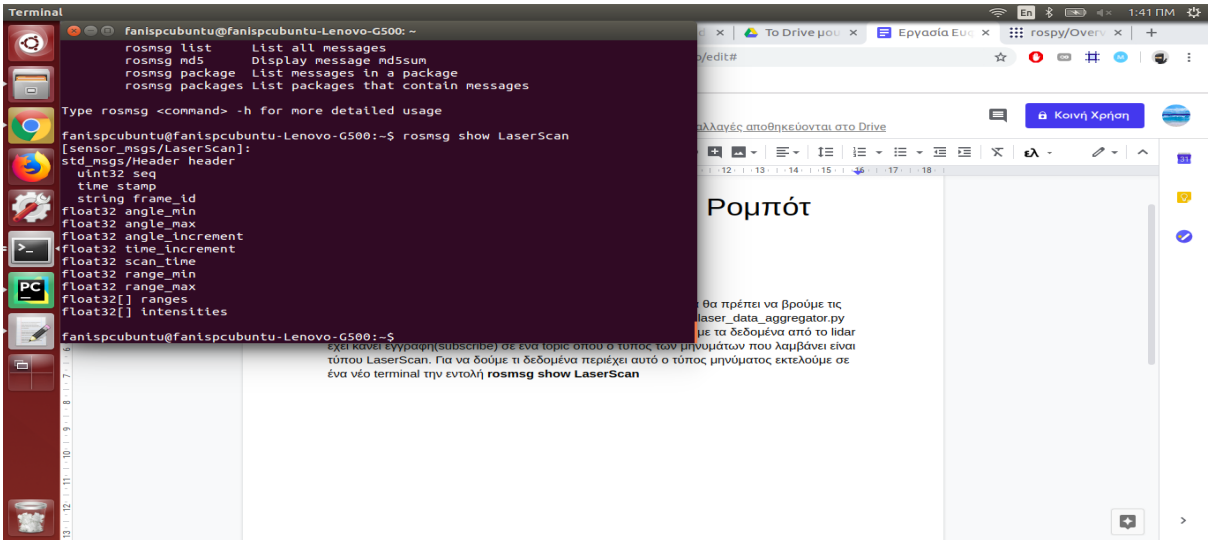
Εργασία Ευφυή Συστήματα Ρομπότ

Φοιτητής : Μπεκιάρης Θεοφάνης ΑΕΜ: 8200

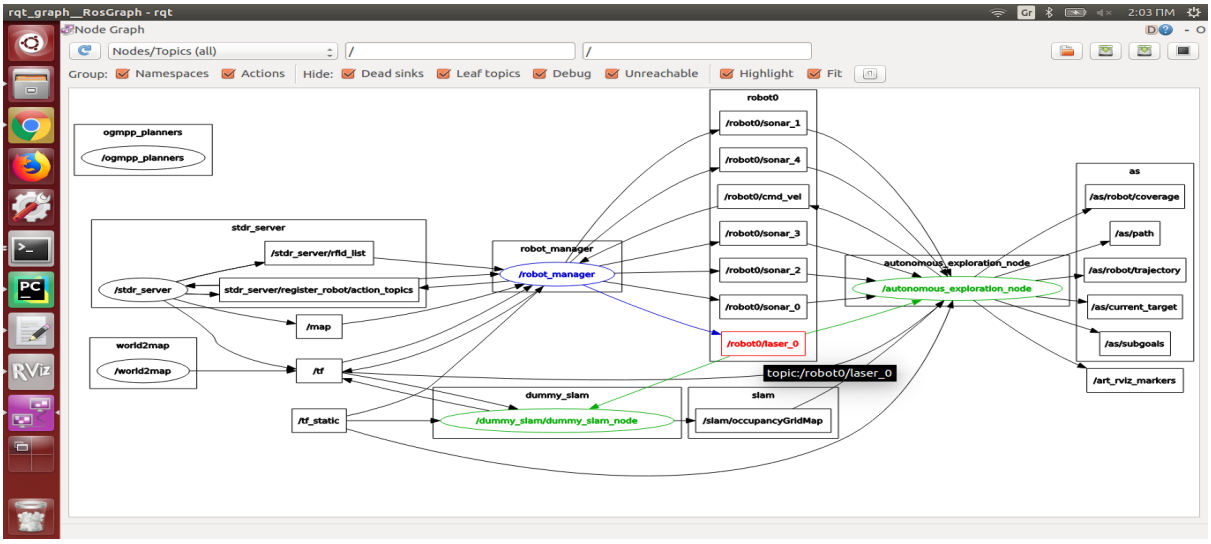
Ακαδημαϊκό Έτος 2018-2019

Δεδομένα της μεταβλητής laser_scan

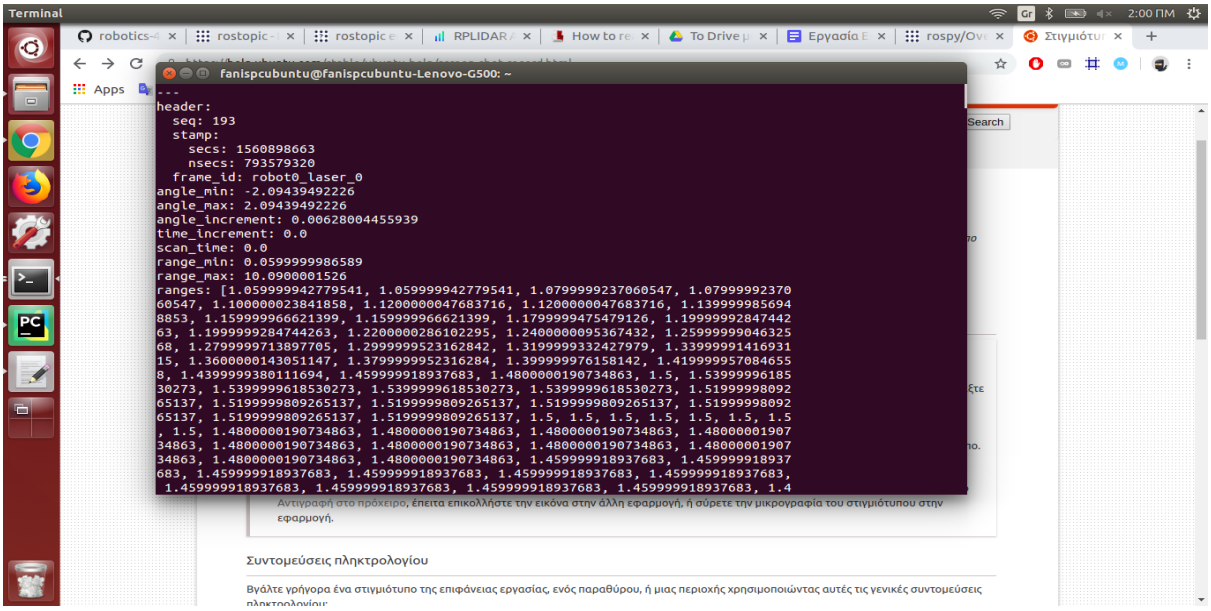
Αρχικά για να μπορέσουμε να απαντήσουμε στο πρώτο ερώτημα θα πρέπει να βρούμε τις δεδομένα περιέχει η μεταβλητή **laser_scan**. Ανοίγουμε το αρχείο laser_data_aggregator.py και παρατηρούμε ότι ο κόμβος(node) που λαμβάνει τα δεδομένα από το lidar έχει κάνει εγγραφή(subscribe) σε ένα topic όπου ο τύπος των μηνυμάτων που λαμβάνει είναι τύπου LaserScan. Για να δούμε τι δεδομένα περιέχει αυτό ο τύπος μηνύματος εκτελούμε σε ένα νέο terminal την εντολή **rosmsg show LaserScan** και παίρνουμε τις παρακάτω πληροφορίες για τα δεδομένα του μηνύματος. Στην συνέχεια στο αρχείο laser_data_aggregator.py γίνεται καταχώρηση του πίνακα **ranges** στην μεταβλητή laser_scan, μέσω της εντολής **self.laser_scan = list(data.ranges)**. Επομένως συμπεραίνουμε ότι η μεταβλητή **laser_scan** περιέχει τις τιμές των αποστάσεων που μετράει το lidar από τα εμπόδια οι οποίες περνάνε μέσω του πίνακα **ragnes** στην μεταβλητή **laser_scan**.



Στην συνέχεια θέλουμε να βρούμε το εύρος των γωνιών μέτρησης του lidar. Η πληροφορία αυτή βρίσκεται μέσα στον παραπάνω τύπο μηνύματος, δηλαδή στο μήνυμα τύπου LaseScan και περιέχεται στις μεταβλητές **angle_min** και **angle_max** που φαίνονται στην παραπάνω εικόνα. Για να δούμε τις τιμές αυτών των μεταβλητών μπορούμε να χρησιμοποιήσουμε την εντολή **rostopic echo <όνομα του topic>** που αντιστοιχεί στο topic τύπου LaserScan και στο οποίο γίνονται publish οι παραπάνω τιμές από το Lidar. Για να μπορέσουμε να εντοπίσουμε ποιό είναι το topic που περιέχει τα δεδομένα για το lidar και μας ενδιαφέρει μπορούμε να χρησιμοποιήσουμε το εργαλείο **rqt_graph** για να απεικονίσουμε ολα τα nodes και τις συνδέσεις που υπάρχουν μέσω των topics. Επομένως μέσω της εντολής **rosrun rqt_graph rqt_graph** παίρνουμε το εξής διάγραμμα.



Βλέπουμε ότι το topic που ψάχνουμε με τα δεδομένα του lidar είναι το **/robot0/laser_0** το οποίο φαίνεται με κόκκινο και στο οποίο publisher είναι ο robot_manager node. Άρα χρησιμοποιώντας την εντολή **rostopic echo /robot0/laser_0** παίρνουμε το εξής αποτέλεσμα της παρακάτω εικόνας.



Όπως βλέπουμε οι τιμές **angle_min** και **angle_max** είναι αντίστοιχα -2.094 και 2.094 σε μονάδες rad, οι οποίες αντιστοιχούν σε -120 και 120 μοίρες γωνία. Τέλος η γωνία μεταξύ διαδοχικών μετρήσεων είναι angle_increment = 0.00628 rad ή 0.36 μοίρες. Τέλος ο αριθμός των δειγμάτων που περιέχει ο πίνακας ranges μπορεί να βρεθεί γράφοντας την εντολή **print len(scan)** στο αρχείο speed_assignment.py και δείνει αποτέλεσμα 667. Αν πολλαπλασιάσουμε τον αριθμό των δειγμάτων με την διακριτική ικανότητα angle_increment έχουμε 667*0.36 = 240 που αντιστοιχεί στο συνολικό εύρος κίνησης του Lidar,δηλαδή όπως αναφέρθηκε από -120 έως 120 μοίρες.
Με τα παραπάνω δεδομένα στην συνέχεια θα υπολογίσουμε τις ζητούμενες ταχύτητες για την αποφυγή εμποδίων(obstacle avoidance).

Ερώτημα 1 (Laser-based obstacle avoidance)

Επιλογή συμπεριφοράς

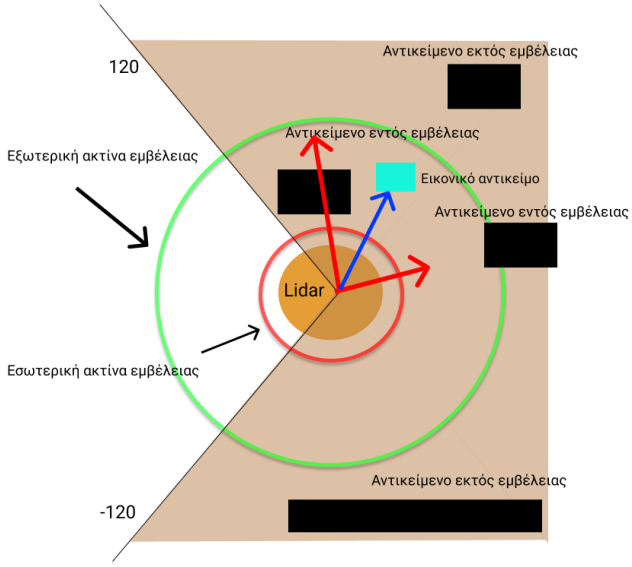
Η συμπεριφορά του ρομποτικού οχήματος σχετικά με την αποφυγή των εμποδίων θα είναι **παθητική**. Το όχημα θα “αισθάνεται” τα αντικείμενα στο περιβάλλον χώρο μέσω του lidar, θα σχεδιάζει την κίνηση του για την αποφυγή των εμποδίων και θα μεταφέρει την πληροφορία στο σύστημα ελέγχου της κίνησης του προσαρμόζοντας κατάλληλα την γραμμική και γωνιακή του ταχύτητα .

Επίσης για την κωδικοποίηση της συμπεριφοράς θα επιλέξουμε τον **συνεχή τρόπο** κωδικοποίησης καθώς **θα περιγράψουμε την συμπεριφορά με μαθηματικές συναρτήσεις**. Ο εναλλακτικός τρόπος είναι ο διακριτός με τον οποίο θα μπορούσαμε να κωδικοποιήσουμε την συμπεριφορά μέσω κανόνων if-then-else (πχ. αν ισχύει η συνθήκη Α τότε στρίψε δεξιά,στρίψε αριστερά κλπ).

Κώδικας για obstacle avoidance

Αρχικά ή κύρια λογική του κώδικα είναι να βρούμε ένα διάνυσμα που αντιστοιχεί στο εμπόδιο που θέλουμε να αποφύγουμε με μέτρο το αντίστροφο της απόστασης του εμποδίου και φορά την κατεύθυνση του εμποδίου. Λόγω του ότι έχουμε πολλά εμπόδια στο περιβάλλον για να υπολογίσουμε ένα μέσο διάνυσμα αντιμετωπίζουμε τις ακτίνες του Lidar που αποθηκεύονται στην μεταβλητή scan σαν διανύσματα με μέτρο το αντίστροφο της τιμής τους και κατεύθυνση την γωνία τους όπως καθορίζεται από το lidar. Άρα όσο πιο κοντά βρίσκεται ένα εμπόδιο το διάνυσμα θα έχει μεγαλύτερη τιμή. Επομένως μια ακτίνα από το Lidar στην θέση των -40 μοιρών που μετράει αντικείμενο σε απόσταση 3 μονάδων αντιστοιχεί σε διάνυσμα με μέτρο ⅓ και γωνία -40 μοίρες. Η ιδέα είναι να αθροίσουμε όλα αυτά τα διανύσματα και να πάρουμε ένα διάνυσμα που αντιστοιχεί στη μέση θέση των αντικειμένων που μετράει το Lidar στο χώρο. Στην παρακάτω εικόνα φαίνεται η παραπάνω λογική. Ο πορτοκαλί κύκλος στην μέση αναπαριστά το Lidar και τα μαύρα τετράγωνα τα αντικείμενα στο χώρο. Τα κόκκινα βέλοι αντιστοιχούν στα διανύσματα των εμποδίων που βλέπει το lidar με μέτρο αντίστροφο από την απόσταση τον αντικειμένων από το όχημα,για αυτό το πιο κοντινό εμπόδιο έχει μεγαλύτερο διάνυσμα. Μετά το διανυσματικό άθροισμα παίρνουμε το μπλε διάνυσμα που αντιστοιχεί σε ένα εικονικό εμπόδιο το οποίο τελικά αντιλαμβάνεται ο αλγόριθμος που

χρησιμοποιεί το Lidar. Στην εικόνα βλέπουμε μόνο δύο διανύσματα για χάρη απλούστευσης, στην πραγματικότητα τα διανύσματα είναι όσες και οι ακτίνες του Lidar δηλαδή 667.



Στο σχήμα επιπλέον βλέπουμε έναν πράσινο κύκλο και ένα κόκκινο, οι οποίοι αντιστοιχούν στο εύρος μέτρησης των αντικειμένων, όσες ακτίνες έχουν μέτρηση από αντικείμενα εκτός του πράσινου κύκλου αντιστοιχούν σε μηδενικά διανύσματα και άρα αγνοούνται κατά το άθροισμα των διανυσμάτων ενώ όταν τα αντικείμενα πλησιάζουν τον κόκκινο κύκλο τα διανύσματα των ακτίνων τους παίρνουν την μέγιστη τιμή. Μεταβάλλοντας τις τιμές των ακτίνων μπορούμε να καθορίσουμε πόσο κοντά στα εμπόδια θα μπορεί να πλησιάζει το όχημα μας.

Αφού υπολογίσουμε το διάνυσμα που αντιστοιχεί στο “μέσο εμπόδιο” υπολογίζουμε το μέτρο και την φορά του

[r, phi] = cmath.polar(obstacle_vector)

και το χρησιμοποιούμε για να υπολογίσουμε τις εξείς σταθερές

const_a = np.sign(-phi) / (1 + abs(phi)) if phi != 0 else 1
const_b = r if abs(phi) < angle1 else 0
const_c = r if abs(phi) < angle2 else 0
const_d = 1 - (math.pi - abs(phi)) / math.pi

της οποίες στην συνέχεια χρησιμοποιούμε για να ρυθμίσουμε την γραμμική και γωνιακή ταχύτητα.
angular = (const_a + np.sign(const_a) * const_b * 5) / 6 if const_b != 0 else ang_acc * const_a * const_c
linear = const_c*(const_d - 1)

Οι παραπάνω σχέσεις προεκυψαν στην προσπάθεια να περιγράψουμε την συμπεριφορά του ρομπότ με μαθηματικές συναρτήσεις και είναι αποτέλεσμα πειραματισμού και δοκιμών έτσι ώστε να ακολουθούν την λογική της κινήσεις σε σχέση με την θέση των εμποδίων, επομένως για την κατανόηση τους θα πρέπει να γίνει καλύτερα η ανάγνωση του κώδικα. Για

παράδειγμα όταν το εμπόδιο βρίσκεται μπροστά στο όχημα και άρα το διάνυσμα έχει phi --> 0 τότε η σταθερά const_d --> 0 άρα linear = const_c*(0-1) = - const_c όπου const_c είναι ίσο με το μέτρο του διανύσματος του εμποδίου αν βρίσκεται εντός κάποιου εύρους γωνιών διαφορετικά 0. Έτσι βλέπουμε ότι όταν το εμπόδιο είναι μπροστά μας η ταχύτητα παίρνει αρνητική τιμή με μέτρο που μεγαλώνει όσο πλησιάσουμε το εμπόδιο.

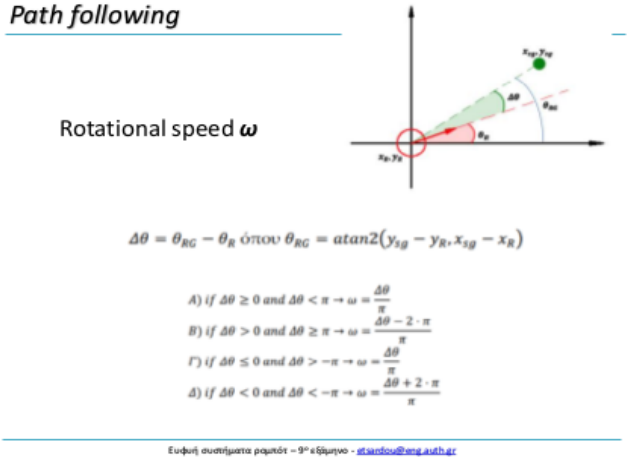
Ερώτημα 2 (Path visualization)

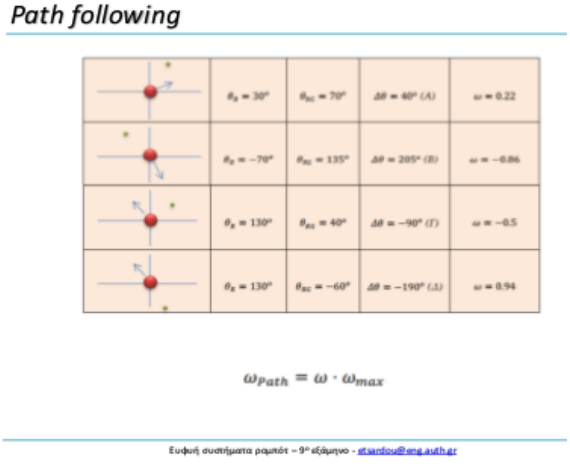
Παρατηρώντας το αρχείο **navigation.py** και robot_perception.py καταλαβαίνουμε οτι η μεταβλητή **self.robot_perception.origin** αντιστοιχεί στην μεταφορά μεταξύ του (0,0) του συστήματος συντεταγμένων του ρομπότ με το (0,0) του συστήματος συντεταγμένων του χάρτη. Η μεταβλητή **self.robot_perception.resolution** έχει αποθηκευμένη την ανάλυστη του occupancy grid map. Επομένως για να κάνουμε εμφανή το μονοπάτι θα πρέπει να πολλαπλασιάσουμε το resolution με τις θέσης των σημείων των subgoals και να προσθέσουμε στο αποτέλεσμα το origin για να γίνει η μεταφορά από το ένα σύστημα στο άλλο. Δηλαδή πρέπει να κάνουμε:

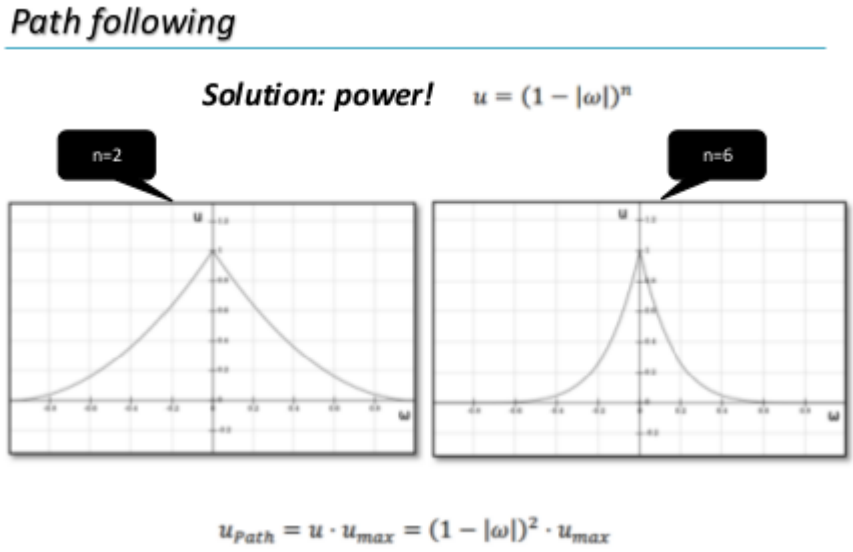
ps.pose.position.x = self.robot_perception.resolution * p[0] + self.robot_perception.origin["x"]
ps.pose.position.y = self.robot_perception.resolution * p[1] + self.robot_perception.origin["y"]

Ερώτημα 3 (Path following)

Για τον υπολογισμό των ταχυτήτων έχουμε ακουθήσει τις εξισώσεις που δίνονται από τις σημειώσεις του μαθήματος στην διαφάνεια 9.exploration_target_selection.pdf και δίνονται από τις παρακάτω εξισώσεις.







Οι τελικές ταχύτητες υπολογίζονται από τις σχέσεις

u = 0.3*(1 - abs(omega))15**
omega = 0.3*omega*3

όπου οι δυνάμεις έχουν επιλεγεί μετά από πειραματισμό με στόχο να μειώσουμε το χρόνο μετάβασης του οχήματος πρός τον στόχο και να εξαλείψουμε προβλήματα όπως είναι το overshoot όπου το όχημα δεν προλαβαίνει να στρίψει και προσπερνάει το στόχο ή κάνει συνεχές κυκλικές κινήσεις γύρω από τον στόχο χωρίς να καταφέρνει να τον προσεγγίσει.

Ερώτημα 4 (Path following & obstacle avoidance)

Οι ταχύτητες για το path following και το obstacle avoidance **συνδυάζονται με motor schema τρόπο** με το παρακάτω τύπο.

l_temp = l_goal + 0.4*l_laser9**
a_temp = a_goal + 0.4*a_laser15**

Οι εκθέτες έχουν επιλεγεί μετά από πειραματισμό και έχουν σκοπό να ομαλοποιήσουν την κίνηση του οχήματος ώστε το όχημα να μπορεί να αποφεύγει τα εμπόδια αλλά να μην στρίβει πολύ απότομα όταν πλησιάζει κάποιο εμπόδιο. Επίσης οι συντελεστές 0.4 έχουν επιλεγεί με σκοπό να δίνουν μεγαλύτερη βαρύτητα στην αποφυγή των εμποδίων πιο αναλυτικά οι μεταβλητή l_goal παίρνει τιμές [0, 0.3] και οι l_laser τιμές [-1, 0] άρα όταν και οι δύο ταχύτητες είναι στα ακρότατα τους δηλαδή στο 1 και στο -1 αντίστοιχα θα υπερτερησει η τιμή για την αποφυγή του εμποδίου και θα έχουμε τελικά l_temp = 0.3 + 0.4*(-1)**9 = -0.1 . Ομοίως για τις γωνιακές έχουμε για την a_goal τιμές στο διάστημα [-0.3, 0.3] και για την a_laser τιμές στο διάστημα [-1, 1] και άρα στα μέγιστα τους θα υπερτερησει η a_laser. Τέλος μετά τον υπολογισμό των ταχυτήτων ελέγχουμε να μην ξεπεράσουν το όριο των 0.3 m/s .

Ερώτημα 5 (Smarter subgoal checking)

Με τον προυπάρχον κώδικα κάθε φορά γίνεται έλεγχος για τον αν το ρομπότ έχει φτάσει στον αμέσως επόμενο στόχο αλλά μπορεί ήδη να έχει περάσει από κάποιον από τους επόμενους ή τον τελικό στόχο. Για να βελτιώσουμε αυτή την υλοποίηση αυτό που κάνουμε είναι να ελέγχουμε όλους τους επόμενους στόχους και αν το όχημα έχει φτάσει σε κάποιον από τους επόμενους συνεχίζουμε από αυτόν.

Ερώτημα 6 (Smart target selection)

Για το πρόβλημα της επιλογής του επόμενου στόχου θα χρησιμοποιήσουμε την τεχνική του cost based target selection. Θα ακολουθήσουμε τους τύπους από την διάλεξη 9 των διαφανειών 67-80. Συγκεκριμένα έχουμε

Κόστος απόστασης

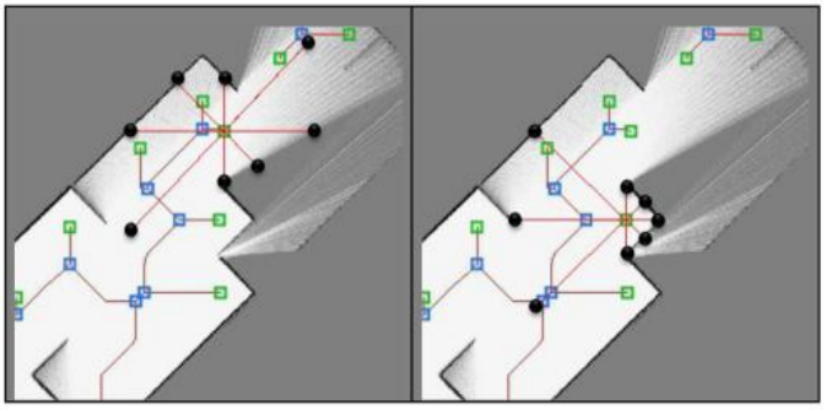
Υπολογίζεται ως το άθροισμα των αποστάσεων των υποστόχων.

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

Τοπολογικό κόστος

Υπολογίζεται ως το άθροισμα των αποστάσεων από τα γύρω εμπόδια, για τον υπολογισμό του κόστους θα χρησιμοποιήσουμε τις τιμές του brushfire, άρα έχουμε

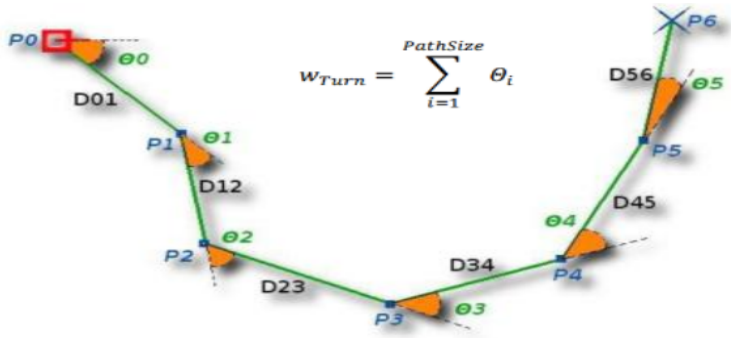
w_{topo} = brushfire(στόχος)



$$w_{Topo} = \frac{1}{8} \cdot \sum_{j=1}^{8} D_j$$

Κόστος στροφής

Υπολογίζεται ως το άθροισμα των γωνιών των υποστόχων.



Κόστος κάλυψης

Τέλος θα υπολογίσουμε το κόστος κάλυψης από τον τύπο.

$$w_{Cove} = 1 - \frac{\sum_{i=1}^{PathSize} Coverage[x_{P_i}, y_{P_i}]}{PathSize \cdot 255}$$

Κανονικοποίηση

Στην συνέχεια κανονικοποιούμε τα κόστη στο διάστημα [0,1] με τις παρακάτω σχέσεις

$$w^k_{Dist_n} = 1 - \frac{w^k_{Dist} - Min(w_{Dist})}{Max(w_{Dist}) - Min(w_{Dist})}$$

$$w^k_{Topo_n} = 1 - \frac{w^k_{Topo} - Min(w_{Topo})}{Max(w_{Topo}) - Min(w_{Topo})}$$

$$w^k_{Turn_n} = 1 - \frac{w^k_{Turn} - Min(w_{Turn})}{Max(w_{Turn}) - Min(w_{Turn})}$$

$$w^k_{Cove_n} = 1 - \frac{w^k_{Cove} - Min(w_{Cove})}{Max(w_{Cove}) - Min(w_{Cove})}$$

Συνδυασμός κόστων με συντελεστές βαρύτητας

Τέλος συνδυάζουμε τα κόστη με την εξής προτεραιότητα
1] topological 2] distance 3]coverage 4]turn με χρήση του παρακάτω τύπου.

$$w_{node} = 2^3 w_{topological} + 2^2 w_{distance} + 2^1 w_{coverage} + 2^0 w_{turn}$$

Τέλος επιλέγουμε το κόμβο με το μεγαλύτερο κόστος ως επόμενο στόχο.

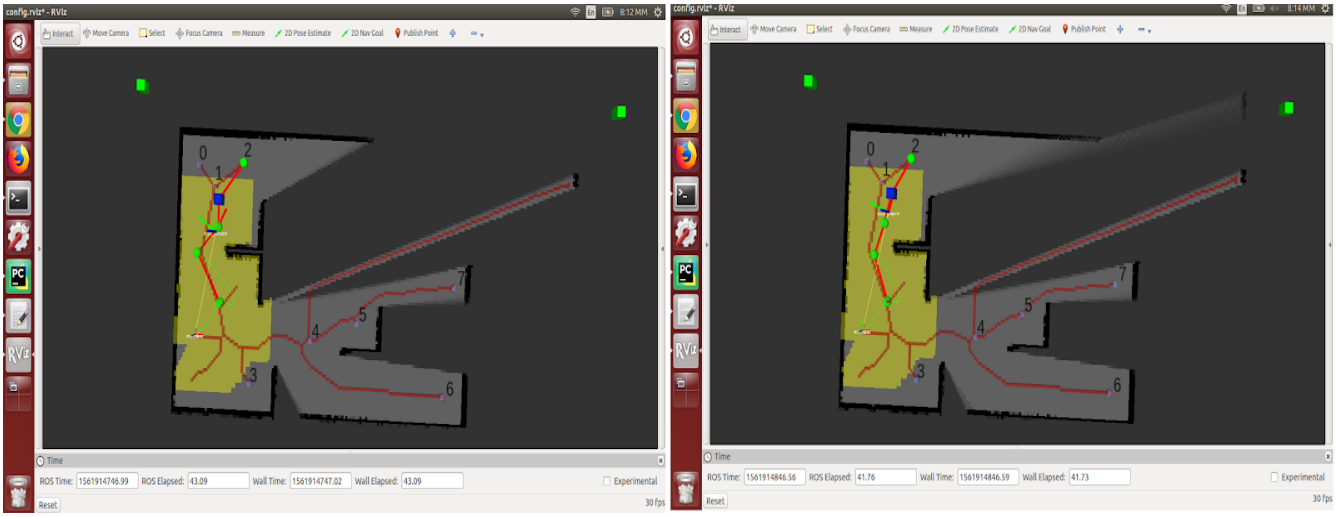
Για την εφαρμογή του cost based selection θέτουμε την μεταβλητή

target_selector: 'costBasedTargetSelection' στο αρχείο autonomous_expl.yaml διαφορετικά παραμένει ως έχει **target_selector: 'random'** για τυχαία επιλογή.

Extra Challenge 1 (Path optimization / alteration)

Για αυτό το ερώτημα για να βελτιστοποιήσουμε το μονοπάτι το οποίο δημιουργείται θα εκτελέσουμε path smoothing για να το κάνουμε πιο ομαλό και μικρότερο σε μήκος. Για την διαδικασία του smoothing έχει χρησιμοποιηθεί η τεχνική που παρουσιάζεται στη παρακάτω ιστοσελίδα.
https://medium.com/@jaems33/understanding-robot-motion-path-smoothing-5970c8363bc4

Παρακάτω φαίνεται το ίδιο μονοπάτι πρίν και μετά το smoothing. Φαίνεται ξεκάθαρα ότι στην δεύτερη εικόνα μετά το smoothing το μονοπάτι έχει γίνει πιο ομαλό και πιο μικρό σε μήκος.

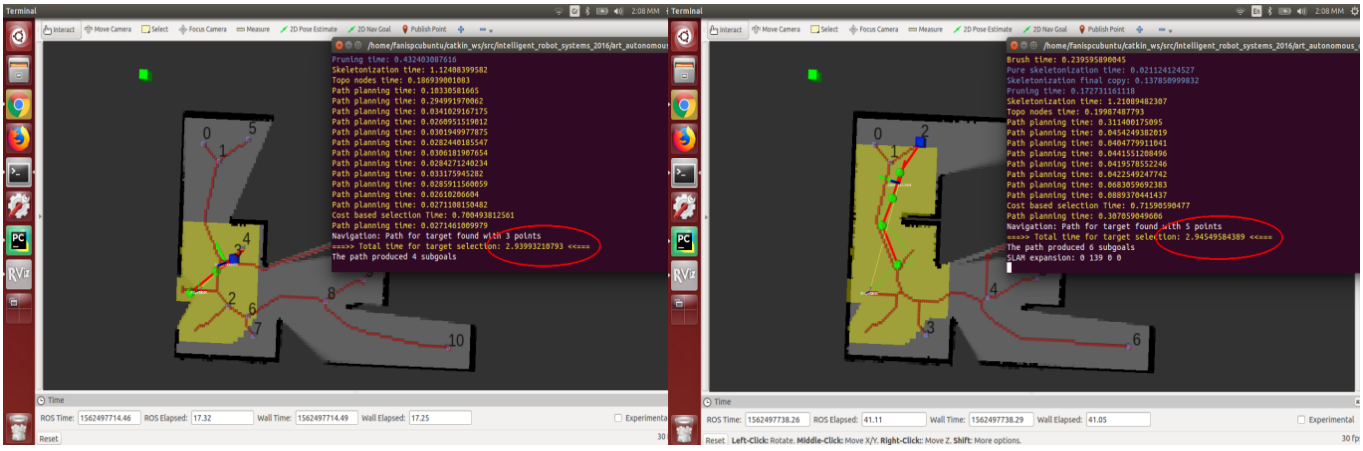


Extra Challenge 2 (Algorithmic optimization)

Για να βελτιώσουμε τον χρόνο εκτέλεσης των αλγορίθμων για το target selection θα τροποποιήσουμε τον κώδικα python των αρχείο utilities.py και topology.py που περιέχουν κρίσιμες συναρτήσεις για την διαδικασία του target selection. Πιο αναλυτικά στο αρχείο utilities.py θα τροποποιήσουμε τις συναρτήσεις brushfireFromObstacles, thinning, prune και blurUnoccupiedOgm και στο αρχείο topology.py τις συναρτήσεις skeletonizationCffi, skeletonization, και topologicalNodes. Παρατηρούμε ότι σε αυτές τις συναρτήσεις υπάρχουν εμφολευμένες for loop με πράξεις πινάκων. Ο συγκεκριμένος τρόπος διαχείρισης τέτοιων πράξεων δεν είναι αποδοτικός, αντ αυτού για τέτοιου είδους πράξεις μπορούμε να χρησιμοποιήσουμε τις συναρτήσεις του πακέτου numpy που είναι σχεδιασμένο για πράξει με πίνακες.
Για να χρονομετρήσουμε την διαδικασία με και χωρίς τις τροποποιήσεις στις παραπάνω συναρτήσεις έχουμε εισάγει στο αρχείο autonomous_expl.yaml την μεταβλητή **numpy_implementation: True,** η οποία όταν είναι True ο κώδικα τρέχει με τις αλλαγές που έχουμε κάνει διαφορετικά για τιμή False παραμένει όπως ήταν αρχικά.

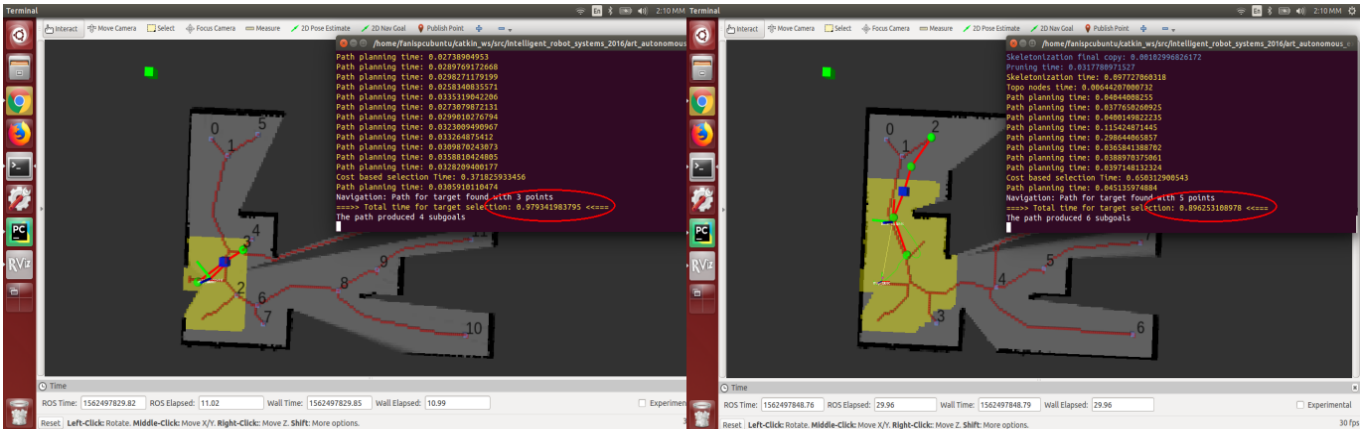
Χρόνοι εκτέλεσης κώδικα

Χωρίς βελτίωση του κώδικα (numpy_implementation: False) έχουμε:



Χρόνοι: 2.93 sec και 2.94 sec

Με βελτίωση του κώδικα (numpy_implementation: True) έχουμε:



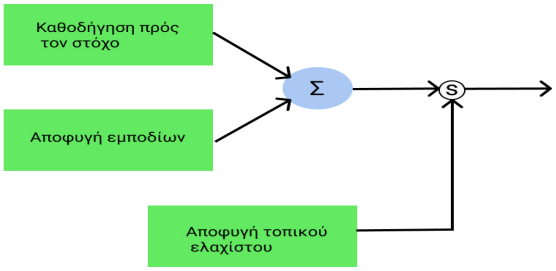
Χρόνοι: 0.97 και 0.89 sec

Παρατηρούμε ότι για την κατασκευή των ίδιων μονοπατιών ο συνολικός χρόνος του target selection χωρίς αλλαγες στον κώδικα είναι 2.93 και 2.94 και με εφαρμογή του numpy πακέτου οι αντίστοιχοι χρόνοι μειώνονται στο 0.97 και 0.89. Πετυχαίνουμε επομένως 3 με 4 φορές μεγαλύτερη ταχύτητα εκτέλεσης του κώδικα.

Extra Challenge 3 (Surprize me / DeadLock - Local Minimum Points)

Κατά την εξερεύνηση του χώρου από το όχημα, παρατηρούμε ότι πολλές φορές το μονοπάτι που σχεδιάζεται περνάει μέσα ή πολύ κοντά από εμπόδια. Το αποτέλεσμα είναι το όχημα να πλησιάζει πολύ κοντά στα εμπόδια και να κολλάει καθώς οι ταχύτητες του στόχου το κατευθύνουν πρός το εμπόδιο ενώ η ταχύτητες για την αποφυγή του εμποδίου το σπρώχνουν μακριά. Έτσι έχουμε τα λεγόμενα τοπικά ελάχιστα ή deadlock καταστάσεις στις οποίες το όχημα κολλάει και ακινητοποιείται σε ένα σημείο. Ένας τρόπος για την επίλυση του παραπάνω προβλήματος είναι να θέσουμε ένα χρονικό όριο το οποίο αν ξεπεραστεί τότε το όχημα να εγκαταλείπει το τρέχον στόχο και να επιλέγει άλλο στόχο με σκοπό να ξεφύγει το τοπικό ελάχιστο. Με αυτό τον τρόπο όμως χάνεται πολύτιμος χρόνος για τον υπολογισμό του νέου στόχου που τελικά μπορεί να μην είναι ικανός να βγάλει το όχημα από το τοπικό

ελάχιστο και έτσι η διαδικασία να επαναλαμβάνεται πολλές φορές μέχρι το όχημα τελικά να βγει από το τοπικό ελάχιστο. Επίσης τι γίνεται στην περίπτωση που το όχημα είναι περικυκλωμένο με εμπόδια και δεν μπορεί να κατασκευάσει μονοπάτι για να ξεφύγει από το τοπικό ελάχιστο ή στη περίπτωση που δεν θα μπορούσαμε να εγκαταλείψουμε το τρέχον στόχο και θα έπρεπε να αποφύγουμε το τοπικό ελάχιστο και να κατευθυνθούμε πρός το στόχο. Για αυτό το challenge έχει κατασκευαστεί η συνάρτηση **local_minimum_unstack** στο αρχείο **speed_assignment.py** η οποίο επιλύει το παραπάνω πρόβλημα. Η επιπλέον συμπεριφορά που προκύπτει για την αποφυγή το τοπικού ελαχίστου συνδυάζεται με subsumption τρόπο όπως παρουσιάζεται στο παρακάτω σχήμα.



Όπως παρουσιάζεται στο σχήμα όταν πέφτουμε σε τοπικό ελάχιστο η συμπεριφορά για την αποφυγεί του τοπικού ελαχίστου κυριαρχεί.
Από το simulation παρατηρήθηκαν οι εξείς περιπτώσεις τοπικού ελαχίστου τις οποίες επιλύει ο κώδικας.

A. Το όχημα κολλάει μπροστά η κοντά σε εμπόδιο
B. Το όχημα εκτελεί ταλάντωση μεταξύ δύο θέσεων
C. Το όχημα κινείται με πολύ μικρές ταχύτητες < 0.01

Αν το όχημα πέσει σε τοπικό ελάχιστο, οι ταχύτητες του υπολογίζονται όπως στην περίπτωση της απλής περιπλάνησης προσθέτοντας στην γωνιακή ταχύτητα έναν όρο ο οποίος δίνει γωνιακή ταχύτητα αντίθετη από την φορά του στόχου ώστε να ξεφύγουμε από το τοπικό ελάχιστο. Τέλος η συμπεριφορά διαρκεί για κάποιο διάστημα που καθορίζεται από την μεταβλητή self.localMinimumCount η οποία μετράει μέχρι μια ανώτατη τιμή και στην συνέχεια τερματίζει την συμπεριφορά. Η συμπεριφορά τέλος τερματίζεται και όταν βγούμε από το τοπικό ελάχιστο.

l_temp = (0.3 + 0.7 * l_laser)
a_temp = (-np.sign(a_goal)*(0.3-abs(a_goal))/(0.3+abs(a_goal)))15 + 0.4 * a_laser**

Για να δώσουμε στο ρομπότ το χρόνο να ξεφύγει από το τοπικό ελάχιστο θα μεταβάλλουμε τις παρακάτω μεταβλητές που διατηρούν το χρονικό όριο για το time reset που βρίσκονται στο αρχείο navigation.py.

self.count_limit = 200 # 20 sec
self.counter_to_next_sub = self.count_limit

Παρατηρούμε ότι 10 μονάδες αντιστοιχούν σε 20 sec άρα για να δώσουμε ένα χρονικό όριο του ενάμιση λεπτού θα μεταβάλουμε την **self.count_limit** από 200 σε 900.