Χαρτογράφηση με Sonars
Nao v5 - ROS

Ομάδα : Νεκτάριος Σφυρής
Email : neksfiris@gmail.com

Εισαγωγή

Για να μπορεί ένα ρομποτικό σύστημα να κινείται ελεύθερα σε ένα χώρο, είναι απαραίτητη η κατανόηση του για την θέση στην οποία βρίσκεται και η δυνατότητα δημιουργίας χάρτη έχωντας γνώση αυτής του της θέσης. Το πρόβλημα που προσπαθήσαμε να επιλύσουμε είναι μέρος του μεγαλύτερου προβλήματος της χαρτογράφησης.

Η εργασία αυτή έγινε στα πλαίσια του μαθήματος Ενσωματωμένα Συστήματα και σκοπός της είναι η χαρτογράφηση ενός στατικού περιβάλλοντος με την χρήση sonars και του ρομποτικού μοντέλου Nao. Για την αποκόμιση των δεδομένων από τους αισθητήρες και την επικοινωνία με το ρομπότ έγινε χρήση του ROS (Robot Operating System). Η γλώσσα υλοποίησης είναι η C++.

Μοντελοποίηση

Η μοντελοποίηση της συμπεριφοράς του Nao, όσον αφορά την κινηματική, έγινε με την χρήση του γραφικού εργαλείου Choregraphe και του simulator webots. Για τον έλεγχο λειτουργίας των sonars χρησιμοποιήθηκε το rviz, το οποίο λειτουργεί παράλληλα με το ros.

Γενικά το Nao έχει δύο sonars στο επιστήθιο μέρος του. Κάθε sonar αποτελείται από ένα ζεύγος αποστολέα υπερηχητικών κυμάτων και αποδέκτη των ανακλασμένων. Υπάρχουν τέσσερις κύριες περιπτώσεις γεγονότων σχετικά με το αποτέλεσμα, είτε να βρει κάτι στα δεξιά, είτε κάτι στα αριστερά, είτε να μην βρει κάτι στα δεξιά και είτε να μην βρει κάτι στα αριστερά. Η συχνότητα με την οποία εκπέμπουν είναι 40 Hz, έχουν 0.25m - 2.55m εύρος ανίχνευσης και εκτείνονται σε κώνο 60 μοιρών. Σε αποστάσεις <0.3 και >2.5 βέβαια, τα σήματα που λαμβάνουμε είναι πιθανότατα θόρυβος. Ακόμα, γνωρίζουμε ότι τα sonar έχουν πολύ θόρυβο στα αποτελέσματά τους και αυτό θα είναι κάτι που

περιμένουμε να δούμε και στα δικά μας αποτελέσματα. Τον χάρτη μας, θα τον φτιάξουμε παίρνοντας τις τιμές των sonars και επεξεργάζοντάς τες στον κώδικα. Για να δούμε τι διαβάζουν τα sonar online από το Nao αρκεί να εκτυπώσουμε στο terminal το ανάλογο topic, ενώ έχουμε τρέξει το naoqi_driver:

$rostopic echo naoqi_driver/sonar/left

Συνολικά το Nao έχει 25 βαθμούς ελευθερίας, και σε όλους εμπλέκονται τα motors για να κινηθεί. Ανάλογα με το πόσο αλλάζουν οι γωνίες των μοτέρ, αλλάζουν και οι θέσεις των άκρων στον χώρο. Στην δική μας περίπτωση θα ελένξουμε τις τιμές των αρθρώσεων μέσα από τον κώδικά μας σε C++, δίνοντας εντολή για το goal σε pose που θέλουμε να έχει.

Το Choregraphe αποτελεί ένα λογισμικό προσομοίωσης αλλά και ανάπτυξης κώδικα ειδικά για Nao και γενικά για ρομποτικά συστήματα της Aldebaran (πλέον Softbank). Για την προσομοίωση φτιάξαμε ένα μικρό πρότζεκτ στο οποίο δείχνουμε μερικές από τις συμπεριφορές που χρησιμοποιήσαμε ή θα μπορούσαμε να χρησιμοποιήσουμε, σε κίνηση και ομιλία. Εκτός του Choregraphe, προσομοιωτές που υπάρχουν και υποστηρίζουν Nao είναι και το webots αλλά και το gazebo. Στο webots υλοποιήσαμε ακόμα μία απλή συμπεριφορά κίνησης που παρουσιάσαμε.

Το rviz είναι ένα visualization tool του ROS το οποίο μπορεί να υποστηρίξει Nao. Αν και δεν είναι προσομοιωτής μπορεί να μας δώσει στοιχεία για τον τρόπο λειτουργίας των sonars αλλά και το εύρος τους και την πραγματική τους συμπεριφορά στο περιβάλλον. Για να το χρησιμοποιήσουμε πρέπει να έχουμε κάνει launch τον naoqi_driver και να έχουμε δημιουργήσει ένα bridge με το Nao. Ταυτόχρονα πρέπει να έχουμε κατεβάσει και να έχουμε κάνει install τα nao_meshes για να μπορεί να το απεικονίσει. Τέλος :

$rosrun rviz rviz

Και ανοίγει το λογισμικό. Κάνοντας τις απαραίτητες ρυθμίσεις το φτιάξαμε να δείχνει το μοντέλο του ρομπότ και τα sonar του.

Η μοντελοποίηση τελικά μας βοήθησε να αποφασίσουμε τον τρόπο κίνησης του Nao για να κάνει την χαρτογράφηση, το τι να περιμένουμε από τα δεδομένα που θα μας δώσουν τα sonar, αλλά

και το πως να κάνουμε το σύστημά μας πιο αξιόπιστο με βάση τις πληροφορίες που έχουμε για αυτό.

Σχεδίαση & Υλοποίηση

Το ROS (Robot – Operating – System) αποτελεί μία πηγή ανοιχτού κώδικα (υποστηριζόμενη απο linux διανομές και MacOS) η οποία προσφέρει δυνατότητες για χαμηλού επιπέδου έλεγχο των συσκευών, μετάδοσης μηνυμάτων μεταξύ διαδικασιών και διαχείρησης πακέτων σε ρομποτικά συστήματα και όχι μόνο.

Για να μπορέσει το ROS σαν λειτουργικό σύστημα να επιτύχει την σωστή λειτουργικότητα και επικοινωνία μεταξύ των μερών ενός συστήματος, χρειάζεται, κατά κύριο λόγο, κάποιον που να ελέγχει την επικοινωνία, και κάποιους που να μεταφέρουν και να επεξεργάζονται τις πληροφορίες.

Βασικές έννοιές του λοιπόν είναι ο ROS Master, ο Parameter Server, τα nodes, τα topics, τα services και τα messages. Τα nodes αποτελούν τα “κουτιά” που υλοποιούν τους υπολογισμούς, και επικοινωνούν μεταξύ τους για την επίλυση μεγαλύτερων προβλημάτων μέσω topics. Τα topics τώρα, είναι οι δίαυλοι μέσω των οποίων τα nodes ανταλλάσουν messages. Σε κάθε topic, τα nodes μπορούν να είναι είτε subscriber, είτε publisher, είτε και τα δύο. Όταν λειτουργούν ως subscriber δέχονται πληροφορίες (messages) για το συγκεκριμένο topic και όταν λειτουργούν ως publisher, δημοσιεύουν πληροφορίες (messages) για το topic. Τα services απαρτίζονται από ένα request και ένα response. Τέλος, ο ROS Master είναι αυτός που ελέγχει την διακίνηση των πληροφοριών καθώς επιτρέπει στα nodes να εντοπίσουν το ένα το άλλο και τελικά να επικοινωνήσουν, και ο Parameter Server, ο οποίος τρέχει μέσα στον ROS Master, είναι αυτός που δίνει την δυνατότητα στα nodes να παίρνουν και να αποθηκεύουν πληροφορίες για τις παραμέτρους των τμημάτων του συστήματος κατά την διάρκεια που αυτό είναι σε λειτουργία.

Στην δικιά μας περίπτωση πρόκειται να χρησιμοποιήσουμε το ROS στον υπολογιστή μας, το οποίο θα λειτουργεί σαν “bridge” μεταξύ του λειτουργικού συστήματος που τρέχει στο Nao (NAOqi OS – linux based) και του έξω κόσμου, στον οποίο είμαστε εμείς. Επίσης το Nao θα κάνει publish όλα τα topics των επιμέρους αισθητήρων και μηχανισμών του, στα οποία εμείς θα μπορούμε να έχουμε πρόσβαση μέσω εντολών του ROS.

Έτσι αυτό που κάναμε πρώτα ήταν η εγκατάστασή του (για την δική μας περίπτωση που δουλεύουμε σε Ubuntu 16.04 η σωστή επιλογή, με βάση την έκδοσή μας, είναι το ROS Kinetic). Για να γίνει αυτό απλά εκτελέσαμε, με δικαιώματα root, την εντολή :

$apt-get install ros-kinetic-desktop-full

και στη συνέχεια κάναμε source την διεύθυνση του αρχείου εγκατάστασης στο .bashrc για να ξέρει από που να διαβάσει.

Αφού το εγκαταστήσαμε, χρειαζόμαστε τα packages που υλοποιούν τον driver μεταξύ του NAOqiOS και ROS (που παρέχει το πραγματικό bridge), και το module που παρέχει την κύρια λειτουργικότητα για Nao σε ROS (implemented in C++). Το πρώτο είναι το naoqi_driver και το δεύτερο το nao_robot.

Για να μπορέσουμε να συνεχίσουμε είναι απαραίτητη η δημιουργία ενός project, το οποίο κάνει απαραίτητη τη χρήση ενός workspace, και αυτό καθώς πρέπει ο κώδικας που το περιλαμβάνει πρέπει να είναι οργανωμένος ώστε να μπορεί να δημιουργήσει τα εκτελέσιμα και τα links μεταξύ των αρχείων. Έτσι στο δικό μας project χρησιμοποιήσαμε το catkin workspace μέσα στο οποίο και αρχίσαμε την υλοποίηση του project μας σε μορφή package.

Για να δημιουργήσουμε το workspace πρέπει πρώτα να δημιουργήσουμε έναν φάκελο, ο οποίος πια θα αναφέρεται σε αυτό, και να κάνουμε build:

```
$mkdir -p ~/catkin_ws/src
$cd catkin_ws
$catkin_make
```

Αφού έχουμε πλέον workspace προχωρούμε με το installation των packages. Για κάθε package κάναμε το installation απο source, δηλαδή το παίρναμε από την πηγή που υπήρχε στο github. Αφού κάναμε αντιγραφή των δεδομένων στο source file του catkin_ws (το workspace που δημιουργήσαμε) τότε προχωρούσαμε σε install, και στην συνέχεια για να γίνει το build των αρχείων κάναμε source στο setup.sh και τελικά catkin_make.

For example:

$git clone https://github.com/ros-naoqi/nao_robot.git
$rosdep install -i -y --from-paths ./nao_robot
$source /opt/ros/kinetic/setup.sh
$cd ../
$catkin_make

Έγινε προσπάθεια εγκατάστασης και του nao_bringup, το οποίο προτείνεται για tutorials αλλά λόγο αρκετών errors δεν χρησιμοποιήθηκε, κάτι που όμως δεν άλλαξε το τελικό αποτέλεσμα επειδή οι υλοποιήσεις του ήταν υποκατηγορία αυτών του naoqi_driver, το οποίο και λειτουργεί μια χαρά και μπορούμε να το χρησιμοποιήσουμε για τα ίδια πράγματα και παραπάνω.

Τώρα πια μπορορούμε να συνδεθούμε στο Nao. Η διασικασία είναι να ενεργοποιήσουμε το Nao, να ανοίξουμε ένα terminal και να κάνουμε :

$roslaunch naoqi_driver naoqi_driver.launch nao_ip:=*** roscore:=*** network_interface:=***

Τα αστεράκια είναι set names δικτύου που πρέπει να γνωρίζουμε για εμάς και το ρομπότ, και τα δικά μας μπορούμε να τα ανακτήσουμε κάνοντας :

$ifconfig

Για το ρομπότ αρκεί να πατήσουμε μία φορά το κουμπί στο στήθος του και θα μας πει την IP του.

Έτσι ο υπολογιστής μας αρχίζει και λαμβάνει πακέτα πληροφοριών για το Nao. Μερικοί headers από τις βασικές εντολές ROS που χρειάζεται να τρέξουμε είναι :

Ότι έχει να κάνει με τα topics που λαμβάνουμε από το Nao
-rostopic

Τρέχει τα nodes locally ή και από απόσταση με ssh
-roslaunch

Δείχνει πληροφορίες για τα nodes
-rosnode

Αν και αυτές οι εντολές είναι σε επίπεδο terminal καταλήξαμε ότι ο κώδικας που θα εκτελεί τις ανάλογες ενέργειες για την επίλυση του προβλήματός μας θα είναι σε C++, καθώς κατά κύριο λόγο το ROS υποσηρίζει καλύτερα C++ και Python.

Έχοντας πια κάνει install το bridge μεταξύ Naoqi και ROS μπορούμε να αρχίσουμε και την υλοποίηση του δικού μας project. Αφού δημιουργήσαμε το workspace πηγαίνουμε στον υποφάκελο src του catkin_ws και δημιουργούμε τον φάκελο που θα αποτελεί το package του project μας. Το όνομα που δόθηκε ήταν sonar_package.

Σε σχέση με το ROS το κάθε package μπορεί να περιέχει αρχεία που αναφέρονται σε messages, services, nodes, scripts, launch file, όπως και τα αρχεία CmakeLists.txt και package.xml τα οποία είναι πολύ σημαντικά για την διαδικασία του build. Όλα αυτά πρέπει να γίνουν και να επεξεργάζονται κατά την διάρκεια εξέλιξης του project από εμάς.

Τα messages είναι απλά text files τα οποία περιγράφουν τα πεδία ενός ROS message. Αποτελούνται από το field type το οποίο ακολουθείται από το field name. Σε κάθε γραμμή υπάρχει ένας τέτοιος συνδυασμός. Στην δική μας περίπτωση στα πλαίσια παραδείγματος για να μάθουμε την χρήση τους, δημιουργήσαμε το Num.msg στο οποίο γράψαμε:

-int64 num

Τα messages μπορούν να είναι πολλών types ή και ROS specific (όπως geometry_msgs/PoseWithCovariance). Ότι δημιουργούμε το χρησιμοποιούμε μετά στον κώδικά μας. Για να δούμε το type και το package στο οποίο ανήκει ένα message κάνουμε:

-rosmsg show Num

Τα services αποτελούνται από δύο μέρη, το request και το response. Τα δύο αυτά μέρη διαχωρίζονται στο file από ‘---’. Στα πλαίσια παραδείγματος δημιουργήσαμε το AddTwoInts.srv στο οποίο γράψαμε:

```
int64 A
int64 B
---
int64 Sum
```

Τα services όπως και τα messages μπορούν να είναι πολλών types. Για να δούμε από τι αποτελείται ένα service κάνουμε:

$rossrv show AddTwoInts

Συγκεντρωτικά όσα messages δημιουργούμε τα αποθηκεύουμε στον φάκελο msg του package μας, και τα services στον φάκελο srv. Για να ενταχθούν και να μπορούν να χρησιμοποιηθούν στον κώδικά μας πρέπει να μπορούν να ενωθούν με κάποιον τρόπο με αυτόν. Την δουλειά αυτή την κάνει το CmakeLists.txt και το package.xml, τα οποία επιτρέπουν στα αρχεία να γίνουν build.

Ενημερώνοντας άρα το package.xml προσθέτουμε:

-<build_depend>message_generation</build_depend>
-<exec_depend>message_runtime</exec_depend>

Για το αρχείο CmakeLists.txt προσθέτουμε στο find_package το message_generations και το std_msgs. Για να εντάξουμε τα messages στο build προσθέτουμε στο catkin_package το message_generations και κάνουμε uncomment το πεδίο add_message_files για να προσθέσουμε το Num.msg. Κάνουμε επίσης uncomment το generate_messages για να προσθέσουμε το std_msgs. Για να εντάξουμε τα services στο build κάνουμε uncomment το πεδίο add_service_files και προσθέτουμε το AddTwoInts.srv.

Αφού εκτελέσαμε τα απαραίτητα βήματα για να κάνουμε build το package και να δημιουργηθούν τα εκτελέσιμα, κάνουμε:

$cd catkin_ws
$catkin_make install

Το κάθε node τώρα παίρνει την μορφή κώδικα (C++ για εμάς). Μέσα σε αυτόν δείχνουμε τον τύπο του, δηλαδή subscriber ή publisher και υλοποιούμε την λειτουργία του. Στο δικό μας δημιουργήσαμε το node με όνομα listener.cpp. Υποχρέωσή του είναι να εγγράφεται στο topic που αναφέρεται για κάθε sonar και έτσι να παίρνει τα δεδομένα που αυτό στέλνει. Το δημιουργήσαμε με την παρακάτω γραμμή κώδικα:

-ros::Subscriber sub = n.subscribe("/naoqi_driver_node/sonar/left", 1000, chatterCallback)

chatterCallback είναι η συνάρτηση που καλείται κάθε φορά που το topic στο οποίο εγγραφήκαμε στέλνει δεδομένα. Εμείς την βάλαμε προσωρινά να τα εκτυπώνει ώστε να διαπιστώσουμε ότι όντως λειτουργεί:

-ROS_INFO("Sonar Seq: [%d]", msg→header.seq)
-ROS_INFO("Sonar Range: [%f]\n", msg->range)

Με την ROS_INFO γίνεται η εκτύπωση των δεδομένων που δέχεται η συνάρτηση, και σε αυτή πρώτα εκτυπώνουμε την σειρά των δεδομένων και μετά την απόσταση που διαβάζει το sonar από το περιβάλλον του.

Με τον τρόπο με τον οποίο κινούμαστε, στο ίδιο node θα λαμβάνουμε τα δεδομένα των sonars (subscriber) αλλά και θα στέλνουμε στο ρομπότ τα δεδομένα για τα joints ώστε να κινείται στο χώρο (publisher). Η χαρτογράφηση θα γίνει πάνω σε ένα grid map στο οποίο ανάλογα με τα δεδομένα που δεχόμαστε θα εκτυπώνουμε σε αυτό τις θέσεις που το ρομπότ νομίζει ότι υπάρχει εμπόδιο. Για να μπορούμε να χρησιμοποιήσουμε opencv στον κώδικά μας για την δημιουργία του χάρτη, σημαντικό είναι να συμπεριλάβουμε την βιβλιοθήκη:

-#include <cv_bridge/cv_bridge.h>

η οποία δίνει την δυνατότητα του bridge μεταξύ του κώδικά μας και της opencv. Στη συνέχεια την αναφέρουμε στο CmakeLists.txt και στο package.xml.

Για να μπορέσει το node που δημιουργήσαμε να φτιάξει εκτελέσιμα πρέπει, στο CmakeLists.txt, να προσθέσουμε:

-add_executable(listener src/listener.cpp)
-target_link_libraries(listener ${catkin_LIBRARIES})
-add_dependencies(listener
-sonar_package_generate_messages_cpp)

Έτσι αφού έγινε και αυτό ξανακάνουμε build το package μας με τον ίδιο τρόπο που αναφέραμε και πριν.

Για να τρέξουμε το node μόνο του πρέπει πωτα να έχουμε κάνει source το workspace μας:

$source catkin_ws/devel/setup.bash

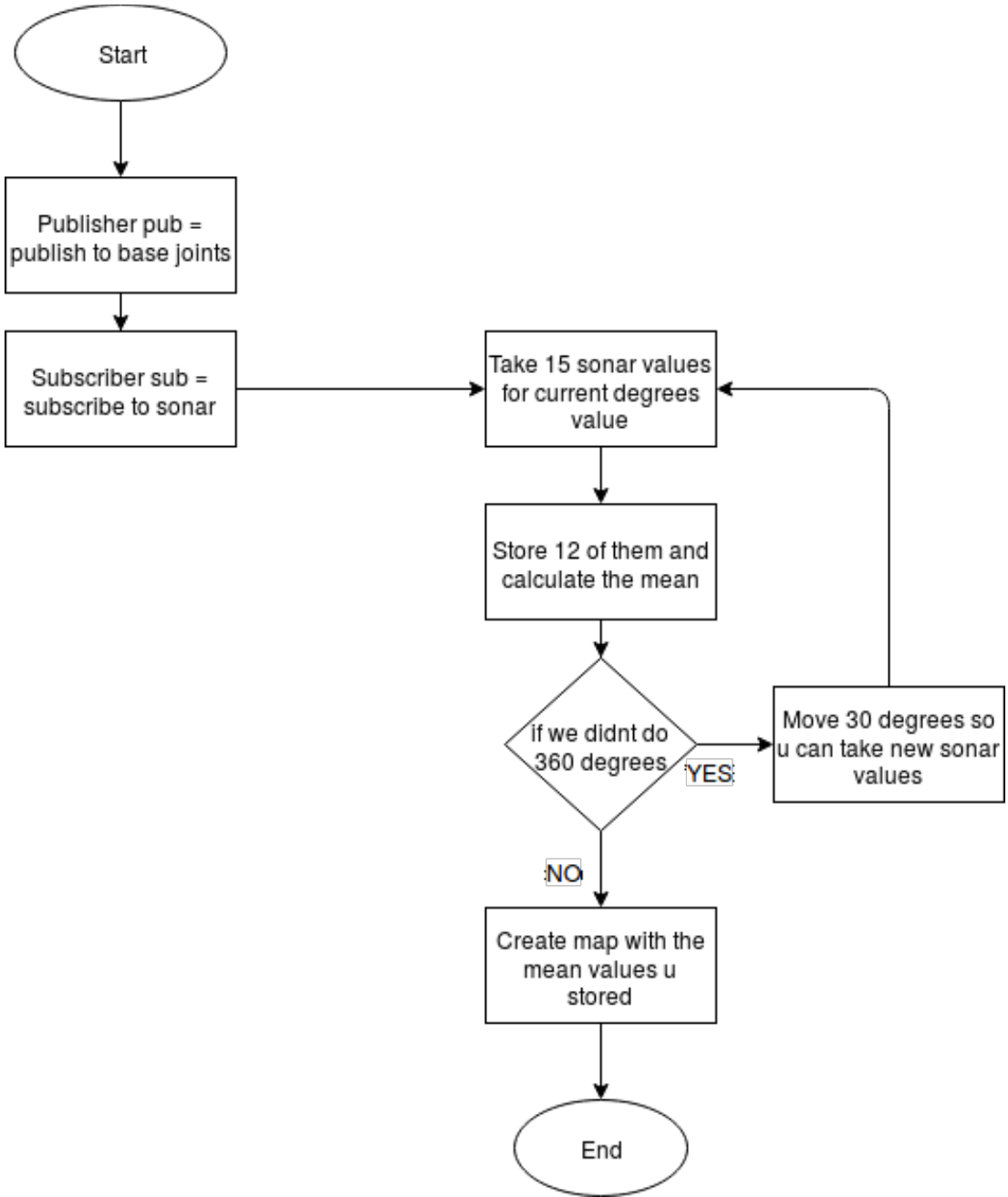
και στη συνέχεια:

$rosrun sonar_package listener

Για να μπορέσει να στρίψει το nao γύρω από τον εαυτό του μέσω του command line εκτελούμε:

$rostopic pub -1 /cmd_vel geometry_msgs/Twist '{linear: {x: 0.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.3}}'

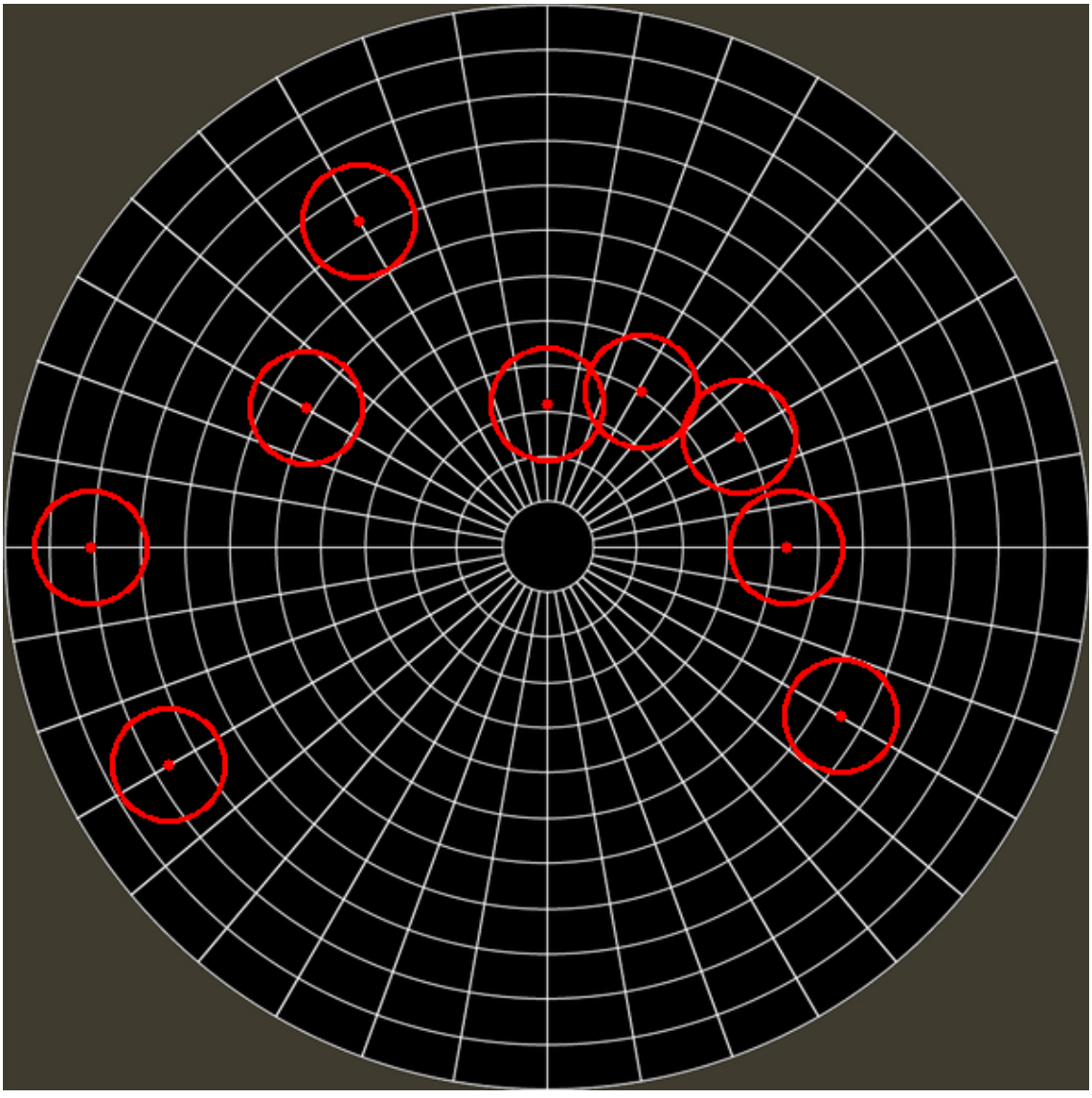
όπου κάνουμε publish την κίνηση που θέλουμε να ακολουθήσει. Η διαδικασία που υλοποιείται στο node μας είναι:



Λίγο πιο αναλυτικά, η λογική που θα ακολουθήσει το Nao είναι ότι παίρνει κάθε φορά 15 τιμές από ένα από τα δύο sonar, κάνει επεξεργασία των τιμών κρατώντας τις 12 από τις 15 και βρίσκοντας την μέση τιμή τους και στη συνέχεια λέει στο Nao να κινηθεί κατά 30 μοίρες γύρω από τον εαυτό του και να συνεχίσει την ίδια διαδικασία. Αυτό γίνεται έως ότου ολοκληρώσει 360 μοίρες. Αφού ολοκληρωθεί η σειρά της αποκόμισης των δεδομένων έρχεται η σειρά δημιουργίας του χάρτη και εκεί ολοκληρώνεται και η συμπεριφορά του ρομπότ.

Επιβεβαίωση λειτουργίας

Με την ολοκλήρωση της λειτουργίας του node δημιουργείται τελικά ο χάρτης μας σε μορφή grid map, στον οποίο φαίνονται ανα 30 μοίρες οι αποστάσεις των αντικειμένων που υπήρχαν στον χώρο γύρω του (αν υπήρχαν ή αν μπορούσε με βάση την απόσταση να τα εντοπίσει) :



Το κύριο σημείο εντοπισμού αναγράφεται με την κόκκινη τελεία και γύρω από αυτή αναφέρουμε την απόκλιση μέσα στην οποία μπορεί να υπάρχει το αντικείμενο παίρνοντας υπόψην τον θόρυβο που υπάρχει.

Συμπεράσματα

Μετά από αρκετά πειράματα καταλήξαμε ότι η καλύτερη ταχύτητα για να κινείται το ρομπότ γύρω από τον εαυτό του και να παίρνουμε όσο το δυνατόν πιο “καθαρές” σε θόρυβο κινήσεις ήταν 10 μοίρες ανά δευτερόλεπτο. Γενικά αντιμετωπίσαμε δυσκολίες στην απόφαση του πως να ανιχνεύουμε την πραγματική θέση ενός αντικειμένου, καθώς αν η επιφάνειά του δεν στέκοταν κάθετα προς το Nao, οι τιμές που έπαιρναν τα sonar κυμαίνονταν γύρω από την πραγματική του τιμή. Έτσι το να παίρνουμε την μέση τιμή από αυτές που εξετάζαμε μας βοήθησε να προσεγγίσουμε κατά πολύ την πραγματική θέση του εμποδίου.

Πάντως τα sonar σαν μέθοδος χαρτογράφησης δεν φαίνονται αρκετά αξιόπιστα από μόνα τους λόγο του μεγάλου θορύβου στα δεδομένα που παίρνουν, που δημιουργεί την ανάγκη για περισσότερη και καλύτερη πληροφορία, όπως αυτή που μας παρέχει η κάμερα. Για προβλήματα στο μέγεθος μιας πραγματικής χαρτογράφησης, η χρήση της κάμερα γίνεται απαραίτητη.

Για το δικό μας πρόβλημα ωστόσο η λύση που δώσαμε μας παρείχε πολύ καλά αποτελέσματα και μπορεί να χρησιμοποιηθεί και σε μεγαλύτερα προβλήματα.

Βιβλιογραφικές Αναφορές

Η βιβλιογραφία που χρησιμοποιήθηκε αναφέρεται παρακάτω σε μορφή links στο διαδίκτυο. Σε αυτήν υπάρχουν αρκετές σελίδες για documentation σε ROS και tutorials όπως και μία προηγούμενη διπλωματική εργασία που είχε γίνει στο Πολυτεχνείο Κρήτης.

http://wiki.ros.org
http://wiki.ros.org/ROS/Tutorials
http://wiki.ros.org/nao
http://ros-naoqi.github.io/naoqi_driver/
http://artemis.library.tuc.gr/DT2013-0024/DT2013-0024.pdf