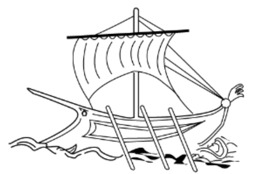
ΑΝΩΤΑΤΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΠΕΙΡΑΙΑ Τ.Τ.



Α.Ε.Ι. ΠΕΙΡΑΙΑ Τ.Τ.

# Χαρτογράφηση και εντοπισμός θέσης με την χρήση RGB-D κάμερας

**ΣΠΟΥΔΑΣΤΕΣ: ΔΗΜΗΤΡΙΟΣ ΠΑΝΤΕΛΗΣ, ΕΝΚΕΛΕΝΤΑ ΜΠΟΤΣΑΪ**

**ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: ΓΡΗΓΟΡΙΟΣ ΝΙΚΟΛΑΟΥ**

## ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

**ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ**

**ΤΜΗΜΑ ΑΥΤΟΜΑΤΙΣΜΟΥ**

**ΟΚΤΩΒΡΙΟΣ 2015 (14/10/2015)**

……………………………………….                    ……………………………………………….

Παντελής Δημήτρης                                  Ενκελέντα Μποτσάϊ

Πτυχιούχοι Μηχανικοί Αυτοματισμού ΑΕΙ Πειραιά Τ.Τ.

Copyright © Δημήτρης Παντελής και Ενκελέντα Μποτσάϊ

Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Ανώτατου Εκπαιδευτικού Ιδρύματος Πειραιά Τ.Τ. .

## ΔΗΛΩΣΗ ΣΥΓΓΡΑΦΕΑ ΠΤΥΧΙΑΚΗΣ ΕΡΓΑΣΙΑΣ

Ο / Η κάτωθι υπογεγραμμένος / η ......Δημήτριος Παντελής......,

του ......Νικολάου......, με αριθμό μητρώου ......39388...... φοιτητής / τρια του Τμήματος **Μηχανικών Αυτοματισμού Τ.Ε.** του Α.Ε.Ι. Πειραιά Τ.Τ. πριν αναλάβω την εκπόνηση της Πτυχιακής Εργασίας μου, δηλώνω ότι ενημερώθηκα για τα παρακάτω:

«Η Πτυχιακή Εργασία (Π.Ε.) αποτελεί προϊόν πνευματικής ιδιοκτησίας τόσο του συγγραφέα, όσο και του Ιδρύματος και θα πρέπει να έχει μοναδικό χαρακτήρα και πρωτότυπο περιεχόμενο.

Απαγορεύεται αυστηρά οποιοδήποτε κομμάτι κειμένου της να εμφανίζεται αυτούσιο ή μεταφρασμένο από κάποια άλλη δημοσιευμένη πηγή. Κάθε τέτοια πράξη αποτελεί προϊόν λογοκλοπής και εγείρει θέμα Ηθικής Τάξης για τα πνευματικά δικαιώματα του άλλου συγγραφέα. Αποκλειστικός υπεύθυνος είναι ο συγγραφέας της Π.Ε., ο οποίος φέρει και την ευθύνη των συνεπειών, ποινικών και άλλων, αυτής της πράξης.

Πέραν των όποιων ποινικών ευθυνών του συγγραφέα σε περίπτωση που το Ίδρυμα του έχει απονείμει Πτυχίο, αυτό ανακαλείται με απόφαση της Συνέλευσης του Τμήματος. Η Συνέλευση του Τμήματος με νέα απόφασης της, μετά από αίτηση του ενδιαφερόμενου, του αναθέτει εκ νέου την εκπόνηση της Π.Ε. με άλλο θέμα και διαφορετικό επιβλέποντα καθηγητή. Η εκπόνηση της εν λόγω Π.Ε. πρέπει να ολοκληρωθεί εντός τουλάχιστον ενός ημερολογιακού 6μήνου από την ημερομηνία ανάθεσης της. Κατά τα λοιπά εφαρμόζονται τα προβλεπόμενα στο άρθρο 18, παρ. 5 του ισχύοντος Εσωτερικού Κανονισμού.»

Ο Δηλών

Ημερομηνία

18/03/2016

## ΔΗΛΩΣΗ ΣΥΓΓΡΑΦΕΑ ΠΤΥΧΙΑΚΗΣ ΕΡΓΑΣΙΑΣ

Ο / Η κάτωθι υπογεγραμμένος / η ......Ενκελέντα Μποτσάι......,

του ......Λαβντιμ......, με αριθμό μητρώου ......39661...... φοιτητής / τρια του Τμήματος **Μηχανικών Αυτοματισμού Τ.Ε.** του Α.Ε.Ι. Πειραιά Τ.Τ. πριν αναλάβω την εκπόνηση της Πτυχιακής Εργασίας μου, δηλώνω ότι ενημερώθηκα για τα παρακάτω:

«Η Πτυχιακή Εργασία (Π.Ε.) αποτελεί προϊόν πνευματικής ιδιοκτησίας τόσο του συγγραφέα, όσο και του Ιδρύματος και θα πρέπει να έχει μοναδικό χαρακτήρα και πρωτότυπο περιεχόμενο.

Απαγορεύεται αυστηρά οποιοδήποτε κομμάτι κειμένου της να εμφανίζεται αυτούσιο ή μεταφρασμένο από κάποια άλλη δημοσιευμένη πηγή. Κάθε τέτοια πράξη αποτελεί προϊόν λογοκλοπής και εγείρει θέμα Ηθικής Τάξης για τα πνευματικά δικαιώματα του άλλου συγγραφέα. Αποκλειστικός υπεύθυνος είναι ο συγγραφέας της Π.Ε., ο οποίος φέρει και την ευθύνη των συνεπειών, ποινικών και άλλων, αυτής της πράξης.

Πέραν των όποιων ποινικών ευθυνών του συγγραφέα σε περίπτωση που το Ίδρυμα του έχει απονείμει Πτυχίο, αυτό ανακαλείται με απόφαση της Συνέλευσης του Τμήματος. Η Συνέλευση του Τμήματος με νέα απόφασης της, μετά από αίτηση του ενδιαφερόμενου, του αναθέτει εκ νέου την εκπόνηση της Π.Ε. με άλλο θέμα και διαφορετικό επιβλέποντα καθηγητή. Η εκπόνηση της εν λόγω Π.Ε. πρέπει να ολοκληρωθεί εντός τουλάχιστον ενός ημερολογιακού 6μήνου από την ημερομηνία ανάθεσης της. Κατά τα λοιπά εφαρμόζονται τα προβλεπόμενα στο άρθρο 18, παρ. 5 του ισχύοντος Εσωτερικού Κανονισμού.»

Ο Δηλών

Ημερομηνία

18/3/16

# Περιεχόμενα


Κεφάλαιο 1. - Εισαγωγή....11

1.1. Τι είναι το SLAM και γιατί είναι σημαντικό .... 11

1.2. Η ιστορία του SLAM .... 11

Κεφάλαιο 2. - Υλικό και λογισμικό....13

2.1. Robot Operating System (ROS) .... 13

2.1.1. Θεωρητικό μέρος .... 13

2.1.2. Σημαντικές τεχνικές λεπτομέρειες .... 15

2.2. PCL .... 15

2.3. OpenCV .... 16

2.3.1. Γενικές γνώσεις .... 16

2.3.2. Τρόπος χρήση του OpenCV στην εργασία .... 16

2.4. MATLAB .... 19

2.5. Kinect .... 19

2.6. Υπολογιστής πειραμάτων .... 22

Κεφάλαιο 3. Φίλτρα και μαθηματικά μοντέλα....**Error! Bookmark not defined.**

3.1. Bayes Filter – Bayes Θεώρημα .... 23

3.1.1. Bayes Filter .... 23

3.1.2. Bayes Θεώρημα .... 23

3.2. Kalman Filter .... 24

3.3. Εκτεταμένο Φίλτρο Kalman (Extended Kalman – EKF) .... 27

3.4. Particle Filter .... 27

3.5. Motion Model .... 28

3.5.1. Αρχικοποίηση του κινηματικού μοντέλου .... 29

3.5.2. Probabilistic Kinematics .... 29

3.5.3. Velocity Motion Model .... 30

3.5.4. Odometry Motion Model .... 32

3.6. Measurement Model .... 33

Κεφάλαιο 4. - Αλγόριθμοι....35

4.1. SLAM .... 35

4.1.1. Probabilistic form του SLAM .... 35

4.1.2. Θεμελιώδη βήματα του SLAM .... 36

4.1.3. Maps in SLAM .... 37

4.2. FastSLAM .... 39

4.2.1. Θεωρητικό υπόβαθρο .... 39

4.2.2. Ο αλγόριθμος ... 41
4.3. AKAZE ... 49
4.4. RANSAC ... 50
Κεφάλαιο 5. - Δοκιμή εφαρμογής και αξιολόγηση ... 53
5.1. Εύρεση σημαντικών σημείων (keypoints) στο Matlab με τη χρίση του Kinect και του αλγόριθμου Sift. ... 53
5.2. Εύρεση σημαντικών σημείων μέσω του Point Cloud Library (PCL) και Kinect. ... 56
5.3. Δοκιμή της MRPT βιβλιοθήκης για την υλοποίηση της εφαρμογής ... 60
Κεφάλαιο 6. - Τελική Υλοποίηση ... 62
6.1. Κτίσιμο ROS πακέτου ... 62
6.2. Front_end_node ... 67
6.2.1. Γενικές πληροφορίες ... 67
6.2.2. front_end κώδικας και αλγόριθμος ... 67
6.2.3. Ανάλυση του fe_MainTools.cpp ... 72
6.2.4. Ανάλυση του fe_ImageTools.cpp ... 81
6.3. SLAM node ... 87
6.3.1. Γενικές πληροφορίες ... 87
6.3.2. Ανάλυση του SLAM.py ... 87
6.3.3. Ανάλυση του SLAM_extretools.py ... 91
Κεφάλαιο 7. - Αποτελέσματα ... 98
7.1. Πως το front_end επεξεργάζεται τα Kinect δεδομένα ... 98
7.2. Κατασκευή του χάρτη ... 104
7.3. Κίνηση των particles. ... 107
Κεφάλαιο 8. - Μελλοντικές βελτιώσεις ... 111
8.1. Συνδυασμός feature-based map με grid-based map ... 111
8.2. Εναλλακτικό motion model ... 111
Παραρτήματα ... 112

// Περίληψη

Η ραγδαία και συνεχής ανάπτυξη των υπολογιστών έκανε πρόσφορο το έδαφος για την περαιτέρω ανάπτυξη της ρομποτικής. Μεγάλο ενδιαφέρον υπήρχε, εδώ και δεκαετίες, για κινούμενα ρομποτικά συστήματα, όπως αυτά έχουν παρουσιαστεί στην λογοτεχνία [0-1] και στον κινηματογράφο [0-2]. Αλλά για να είναι πραγματικά αυτόνομο ένα ρομπότ πρέπει να μπορεί να κινείται μόνο του χωρίς προηγουμένως να γνωρίζει για το χώρο και την διαδρομή. Εδώ είναι που εμφανίζεται το SLAM (Simultaneous Localization And Mapping).

Ενώ το SLAM σαν θέμα έρευνας υπάρχει εδώ και κάποιες δεκαετίες [1.2-3, 1.2-4], η χρήση του σε μη ελεγχόμενους χώρους ήταν πολύ περιορισμένη. Αυτό οφείλεται σε τρία, κυρίως, προβλήματα: στην έλλειψη μικρο-υπολογιστών μεγάλων επεξεργαστικών δυνατοτήτων, στην έλλειψη αποτελεσματικών αλγορίθμων και στην έλλειψη αισθητήρων μεγάλης ακρίβειας και προσιτού κόστους. Τα τελευταία χρόνια όμως, έχουν γίνει μεγάλα βήματα σε αυτούς τους τομείς και αυτό είχε ως αποτέλεσμα να έχει γίνει σημαντική έρευνα σε αυτόν τον τομέα.

Στην παρούσα πτυχιακή εργασία σχεδιάσαμε ένα SLAM σύστημα που κάνει χρήση του FastSLAM [1.4-1] αλγορίθμου σε 3D χώρο. Μέχρι σήμερα, η μοναδική δουλειά σχετική με το 3D FastSLAM είναι το [1.4-2]. Αντίθετα, όμως, με το [1.4-2] η δική μας περίπτωση περιλαμβάνει features-based SLAM.

Ο FastSLAM κάνει χρήση Particle Filter για τον εντοπισμό της θέσης και Extended Kalman Filter για την διατήρηση της θέσης των landmarks. Όλο το σύστημα μας έχει χτιστεί πειραματικά πάνω στο Robotics Operating System (ROS). Τα διάφορα μέρη του συστήματος έχουν γραφτεί σε C++ και Python. Επιπροσθέτως, η υλοποίηση του FastSLAM έχει γίνει με τη χρήση ενός αισθητήρα Kinect. Η εφαρμογή είναι σχεδιασμένη για χρήση σε εσωτερικό χώρο, χωρίς την ύπαρξη κινούμενων αντικειμένων μέσα σε αυτόν. Ο αλγόριθμος θεωρεί ότι κάνουμε χρήση differential drive αν και μπορεί εύκολα να προσαρμοστεί σε άλλο μοντέλο κίνησης.

Ως αποτελέσματα θα παρουσιάσουμε τις ακριβείς μετρήσεις που κάνει το Kinect και την διαδικασία επεξεργασίας τους στον αλγόριθμο μας. Επίσης θα αξιολογήσουμε πόσο καλά εντοπίζει την θέση του οχήματος σε διάφορες συνθήκες.

## Λέξεις Κλειδιά

SLAM, 3D SLAM, FastSLAM, Kinect, Robotics Operating System, Particle Filter, Extended Kalman Filter

// Abstract

The rapid and continues development of computer opened the door for further growth of robotics. There has been a huge interest, for many decades, for mobile robotic systems, similarly to those presented in literature [0-1] and movies [0-2]. But for a robot to be truly autonomous it needs to have the ability to move independent of any prior knowledge of its surrounding map and location. This is the problem SLAM (simultaneous localization and mapping) came to solve.

While SLAM has been in the scope of researchers for a few decades [1.2-3, 1.2-4], its use, in a non-controlled area, was limited. There were mainly 3 reasons that we can attribute this to: microcontrollers with low computational power, non-existence of effective algorithms and expensive sensors or sensors with low precision. In recent years however, there has been remarkable progress in these fields.

In this thesis we designed a SLAM system that uses the FastSLAM [1.4-1] algorithm in 3D space. Until now the only work concerning 3D space FastSLAM is [1.4-2], but in our system we are going to make it be feature-based. The FastSLAM uses a Particle Filter to detect its own position and an Extended Kalman Filter to keep track of the landmarks' position. Our whole system will be built on the Robotics Operating System (ROS). Our software components will we written in C++ and Python. Also, the FastSLAM implementation will be done using the Kinect Sensor. Our application is designed for in-door use, with no moving objects in the environment. The algorithm assumes that we are using the differential drive motion model although it can be easily adapted to any other.

As results we will present the exact measurements that the Kinect makes and how they are processed in the algorithm. Also, we will evaluate how well it can detect the true position of the robot under different conditions.

## Key Words

SLAM, 3D SLAM, FastSLAM, Kinect, Robotics Operating System, Particle Filter, Extended Kalman Filter

# Πίνακας Περιεχομένων

## Λίστα Εικόνων

Εικόνα 2.3.1 ........ 18
Εικόνα 2.3.2 ........ 19
Εικόνα 2.5.1 ........ 21
Εικόνα 2.5.2 ........ 21
Εικόνα 3.5.1 ........ 32
Εικόνα 3.5.2 ........ 33
Εικόνα 4.2.1 ........ 42
Εικόνα 4.2.2 ........ 45
Εικόνα 4.2.3 ........ 45
Εικόνα 4.2.4 ........ 47
Εικόνα 6.2.1 ........ 67
Εικόνα 7.1.1 ........ 98
Εικόνα 7.1.2 ........ 99
Εικόνα 7.1.3 ........ 99
Εικόνα 7.1.4 ........ 100
Εικόνα 7.1.5 ........ 101
Εικόνα 7.1.6 ........ 101
Εικόνα 7.1.7 ........ 102
Εικόνα 7.1.8 ........ 103
Εικόνα 7.1.9 ........ 103
Εικόνα 7.2.1 ........ 104
Εικόνα 7.2.2 ........ 105
Εικόνα 7.2.3 ........ 105
Εικόνα 7.2.4 ........ 106
Εικόνα 7.2.5 ........ 106
Εικόνα 7.2.6 ........ 107
Εικόνα 7.3.1 ........ 107
Εικόνα 7.3.2 ........ 108
Εικόνα 7.3.3 ........ 108
Εικόνα 7.3.4 ........ 109
Εικόνα 7.3.5 ........ 109
Εικόνα 7.3.6 ........ 110

## Λίστα Πινάκων

Πίνακας 3.5.1 ........ 29
Πίνακας 3.5.2 ........ 31
Πίνακας 3.5.3 ........ 32
Πίνακας 3.6.1 ........ 34
Πίνακας 4.2.1 ........ 44
Πίνακας 4.2.2 ........ 44
Πίνακας 4.2.3 ........ 48
Πίνακας 4.2.4 ........ 49
Πίνακας 6.3.1 ........ 87

# Κατάλογος Συμβόλων & Συντομογραφιών

| | |
|---|---|
| $s_t$ | Robot pose |
| $S_t$ | Particle |
| m | χάρτης |
| $m_t$ | ο πιο πρόσφατο χάρτης |
| $z_t$ | πραγματική μέτρηση |
| $z'_t$ | υπολογισμένη μέτρηση |
| $u_t$ | εντολή κίνησης |
| $c_t$ | data association της μέτρησης |
| $h( s_{t-1}, u_t)$ | measurement model |
| $H$ | γραμμικοποιημένο measurement model |
| $Q$ | διακύμανση μέτρησης |
| $w_t^{[m]}$ | importance weight |
| $[m]$ | δείκτης particle |
| n | δείκτης ενός landmark |
| M | πλήθος των particles |
| N | πλήθος landmarks |
| $\mu_{kinect}$ | [x,y,z] from locate frame |

# Ευχαριστίες

Θα θέλαμε να εκφράσουμε τις ειλικρινείς ευχαριστίες μας στον κ. Γρηγόριο Νικολάου για την άρτια συνεργασία που είχαμε καθ' όλη τη διάρκεια της εκπόνησης αυτής της διπλωματικής εργασίας, καθώς και στον καθηγητή κ. Χαράλαμπο Πατρικάκη για την πολύτιμη βοήθεια του κατά την διάρκεια δυσκολιών που αντιμετωπίσαμε.

Τέλος θα θέλαμε να ευχαριστήσουμε τις οικογένειες μας για την στήριξη που μας έχουν προσφέρει κατά την διάρκεια των σπουδών μας στο ΑΕΙ Πειραιά Τ.Τ. .

# Κεφάλαιο 1. Εισαγωγή

Στο κεφάλαιο αυτό θα μιλήσουμε σε θεωρητικό επίπεδο για το SLAM, καθώς και για το πώς ξεκίνησε και εξελίχθηκε έως σήμερα.

## 1.1. Τι είναι το SLAM και γιατί είναι σημαντικό

To Simultaneous Localization And Mapping (SLAM) είναι ένας όρος "ομπρέλα" που αναφέρεται σε αλγορίθμους που κάνουν χαρτογράφηση ενός άγνωστου χώρου, ενώ συγχρόνως εντοπίζουν την θέση του ρομπότ μέσα στον χάρτη που έχει δημιουργηθεί. Το SLAM θεωρείται η λύση σε ένα από τα πιο βασικά προβλήματα της ρομποτικής, που είναι η έλλειψη πρόσβασης στο χάρτη του περιβάλλοντος στον οποίο κινείται το ρομπότ και η ανεπαρκής γνώση της ακριβής του θέσης. Τα μόνα στοιχεία που έχουμε είναι η μέτρηση $z_t$ και η εντολή κίνησης $u_t$.

Οι αλγόριθμοι που συσχετίζονται με το SLAM υπάγονται συνήθως σε τρεις ευρύτερες ομάδες:

- Kalman Filter
- Particle Filter
- Graph-Based SLAM

Το SLAM είναι πολύ σημαντικό γιατί επιτρέπει σε ένα ρομπότ που κινείται μόνο του να "κατασκευάζει" τον χώρο γύρο του και να εντοπίζει την θέση του σε αυτόν. Του επιτρέπει, δηλαδή, να δρα αυτόνομα. Αυτή την διαδικασία της χαρτογράφησης και του εντοπισμού της θέσης εμείς οι άνθρωποι την κάνουμε συνέχεια στην καθημερινότητά μας. Για παράδειγμα, όταν οδηγούμε σε έναν άγνωστο δρόμο αφού στρίψουμε, προσπαθούμε να θυμόμαστε κάποιο ιδιαίτερο κτίριο κοντά στην γωνία έτσι ώστε όταν ξαναπεράσουμε από αυτόν τον δρόμο να γνωρίζουμε περίπου που είμαστε. Είναι συνεπώς αυτονόητο ότι μια τέτοια ιδιότητα μπορεί να είναι πολύ χρήσιμη για ένα ρομποτικό σύστημα. Το επόμενο βήμα, εφόσον έχουμε τον χάρτη και την θέση (δηλαδή έχουμε λύσει το πρόβλημα SLAM), είναι η πλοήγηση και ο σχεδιασμός διαδρομής.

## 1.2. Η ιστορία του SLAM

Δύο από τα πρώτα έργα που ξεκίνησαν την ιδέα του SLAM ήταν το [1.2-3] και [1.2-4]. Έπειτα, στην δεκαετία του '90, η ερευνητική ομάδα του Hugh F. Durrant-Whyte και του John J. Leonard [1.2-5] έκανε σημαντικές προόδους πάνω στο SLAM. Ο Durrant-Whyte και ο

Leonard ήταν αυτοί που πρώτα καθιέρωσαν τον όρο SLAM, αν και αρχικά το είχαν ονομάσει SMAL (που εγκαταλείφθηκε σαν ακρωνύμιο για λόγους ευηχίας). Αν και υπήρξε κάποιο αρχικό ενδιαφέρον για το SLAM ήταν, εντούτοις, περιορισμένο. Μόνο ύστερα από την νίκη του STANLEY [1.2-9] στο DARPA Grand Challenge δημιουργήθηκε αρκετό ενδιαφέρον ώστε να στραφούν τα βλέμματα του επιστημονικού και ερευνητικού κόσμου πάνω στο SLAM.

Την τελευταία δεκαετία είναι πολλά τα πανεπιστήμια και τα ιδρύματα που έχουν ασχοληθεί με την έρευνα πάνω στο SLAM. Ένα βασικό στοιχείο που έχει συμβάλει στην ανάπτυξη και βελτίωση των αλγορίθμων του, είναι η συνεχής και ραγδαία ανάπτυξη της επεξεργαστικής ισχύος των υπολογιστών.

Για την εφαρμογή του SLAM μπορεί να γίνει χρήση πολλών διαφορετικών αισθητηρίων. Ένα από τα πιο συνηθισμένα είναι το laser scanner, το οποίο έχει πολύ μεγάλη ακρίβεια αλλά υψηλή τιμή, και για αυτόν τον λόγο δεν είναι δυνατή η χρήση του σε ευρεία κλίμακα. Ένα άλλο αισθητήριο είναι η stereo camera, η οποία επιτρέπει στο ρομπότ να "συνειδητοποιεί" την απόσταση από ένα αντικείμενο χρησιμοποιώντας παρόμοια τεχνική με αυτήν των ανθρώπινων ματιών. Παρότι έχει γίνει προσπάθεια να λειτουργήσει SLAM με μια απλή RGB κάμερα (το λεγόμενο VSLAM), η φωτεινότητα του χώρου επηρεάζει σε απαγορευτικό βαθμό τα αποτελέσματα και συνεπώς η χρήση τόσο της stereo camera όσο και της RGB συνήθως αντενδείκνυνται.

Οι εφαρμογές του SLAM είναι πολλές. Μερικές από αυτές είναι τα self-driving cars [1.2-3, 1.2-4], τα τηλεκατευθυνόμενα εναέρια οχήματα [1.2-5, 1.2-6], τα αυτόνομα υποβρύχια ρομπότ [1.2-7, 1.2-8], τα planetary rovers, τα εγχώρια ρομπότ, η χαρτογράφηση ορυχείων[1.2-1, 1.2-2] και τα ρομπότ νανοιατρικών εφαρμογών. Άλλες εφαρμογές στις οποίες θα μπορούσε να βρει χρήση το SLAM είναι η χαρτογράφηση επιβλαβών για την υγεία χώρων, η εικονική πραγματικότητα και η αναζήτηση αγνοούμενων ατόμων.

# Κεφάλαιο 2. Υλικό και λογισμικό

Σε αυτό το κεφάλαιο θα αναλύσουμε το λογισμικό που χρησιμοποιήσαμε και τον τρόπο με τον οποίο αυτό συνεργάστηκε με το hardware μας. Θα αναφέρουμε, επίσης, και τυχόν λογισμικά που χρησιμοποιήθηκαν για τις προεργασιακές δοκιμές και ας μην είναι το τελικό αποτέλεσμα.

## 2.1. Robot Operating System (ROS)

### 2.1.1. Θεωρητικό μέρος

Το Robot Operating System (ROS) [2.2-1] είναι ο πυρήνας της εφαρμογής μας. Μέσω αυτού συνδέονται και επικοινωνούν όλα τα επιμέρους λογισμικά και hardware. Το ROS είναι μια

ευέλικτη πλατφόρμα (framework) για τον προγραμματισμό ρομπότ που προσφέρει λειτουργίες παρεμφερείς με αυτές ενός λειτουργικού συστήματος. Το ROS παρέχει πρότυπες υπηρεσίες ενός λειτουργικού συστήματος, όπως hardware abstraction, χαμηλού επιπέδου έλεγχο συσκευών, υλοποιημένες λειτουργίες που χρησιμοποιούνται συχνά, ανταλλαγή μηνυμάτων ανάμεσα σε διεργασίες και εργαλεία διαχείριση πακέτων. Η διεργασίες που τρέχουν πίσω από το ROS αναπαρίστανται με μια γραφική αρχιτεκτονική (graph architecture) στην οποία οι διεργασίες τρέχουν μέσω *nodes*. Τα *nodes* μπορούν να δέχονται, να στέλνουν και να κάνουν πολυπλεξία διαφόρων τύπων μηνυμάτων (ROS messages), ανεξαρτήτως προγραμματιστικής γλώσσας ή άλλων χαρακτηριστικών. Τα μηνύματα μπορεί να προέρχονται από αισθητήρες, εντολές, κατάσταση διεργασιών, σχεδιασμός και άλλα.

Το Robot Operating System Ecosystem μπορεί να χωριστεί σε τρεις βασικές κατηγορίες:

- Χρήση εργαλείων, που είναι ανεξάρτητα από την γλώσσα προγραμματισμού ή την ρομποτική πλατφόρμα, που επιτρέπουν την δημιουργία και διανομή ROS-based λογισμικού.
- ROS “client library implementations” όπως είναι το rospy, roscpp, rosjava και το roslisp, που επιτρέπει την ανάπτυξη διεργασιών, μέσα στο ίδιο ρομποτικό σύστημα, με διαφορετικές γλώσσες προγραμματισμού.
- Πακέτα (ROS packages) που προσφέρουν κώδικα για συγκεκριμένες εφαρμογές, τα οποία με την σειρά τους είναι βασισμένα σε βιβλιοθήκες του ROS.

Τα εργαλεία του ROS μαζί με τα client libraries (C++, Python, Java και LISP) που έχουν προαναφερθεί, βρίσκονται κάτω από την BSD άδεια, ανήκουν δηλαδή στον “ανοικτό κώδικα λογισμικού” και μπορούν να χρησιμοποιηθούν δωρεάν για εμπορικούς και ερευνητικούς σκοπούς. Η πλειοψηφία των υπόλοιπων πακέτων υπάγονται κάτω από άλλες εκδοχές του Open Source License. Τα τελευταία αυτά πακέτα προσφέρουν συνηθισμένες λειτουργίες όπως hardware drivers, ρομποτικά μοντέλα, τύπους δεδομένων (data types), σχεδιασμός διαδρομής (path planning), αντίληψη περιβάλλοντος (perception), simultaneous localization and mapping, εργαλεία εξομοίωσης (simulation tools) και άλλους αλγόριθμους.

Οι κύριες γλώσσες προγραμματισμού που χρησιμοποιούνται στο ROS, ανάλογα με το client library , μπορεί να είναι η C++ , η Python, η Java και η LISP. Αυτά τα client libraries στοχεύουν κυρίως στα Unix συστήματα. Ο κύριος λόγος για αυτό είναι επειδή το ROS είναι εξαρτημένο από πολλά open source λογισμικά. Το επισήμως «υποστηριζόμενο» λειτουργικό σύστημα είναι το Ubuntu Linux ενώ κάτω από τον τίτλο «πειραματικά» είναι τα Fedora Linux, Mac OS X και Microsoft Windows. Η C++, η Python και η LISP είναι για τα προαναφερθέντα λειτουργικά συστήματα, ενώ η Java χρησιμοποιείτε για ROS εφαρμογές σε Android περιβάλλον. Τέλος, υπάρχει και σε συμβατή με το JavaScript έκδοση που επιτρέπει την ενσωμάτωση λογισμικού στο ROS μέσω ενός συμβατού φυλλομετρητή (web browser), αν και βρίσκεται ακόμη σε πειραματικό στάδιο.

Καταλήγοντας, ο βασικός λόγος ύπαρξης του ROS είναι διότι η δημιουργία ενός πραγματικά ευέλικτου και γενικής χρήσης λογισμικού για ρομπότ είναι πολύ δύσκολη. Ενώ για έναν άνθρωπο μια σωματική κίνηση μπορεί να είναι μία απλή λειτουργία, για ένα ρομπότ η ίδια απλή λειτουργία μπορεί να χρειάζεται πολλές μικρές αλλαγές στις πολλές παραμέτρους που έχει. Το να χρειάζεται να ασχολείται κάποιος με κάθε διαφορετικό σενάριο που θα αντιμετωπίσει το ρομπότ είναι πολύ δύσκολο ακόμη και για μεγάλα εργαστήρια ή ιδρύματα.

Για τον λόγο αυτό, το ROS χτίστηκε με έναν τέτοιο τρόπο ώστε να μπορούν εύκολα να επαναχρησιμοποιούνται και να αναπτύσσονται κομμάτια ανεξάρτητα μεταξύ τους που λειτουργούν αρμονικά μαζί.

Η έκδοση του ROS που χρησιμοποιήσαμε εμείς είναι το “Indigo Igloo” που κυκλοφόρησε στις 22 Ιουλίου 2014.

### 2.1.2. Σημαντικές τεχνικές λεπτομέρειες

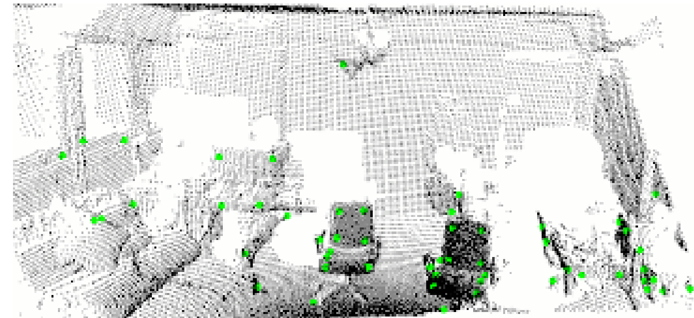
Μια από τις βασικές λειτουργίες του ROS είναι να φτιάχνει ένα δίκτυο από nodes, να εξασφαλίζει την σωστή λειτουργία τους και την σωστή επικοινωνία τους. Ένα node μπορούμε να πούμε ότι είναι ένα εκτελέσιμο κομμάτι κώδικα. Το κάθε node μπορεί να γραφτεί σε οποιαδήποτε από τις προαναφερθείσες γλώσσες προγραμματισμού που επικοινωνούν μεταξύ τους χρησιμοποιώντας ROS messages. Εμείς έχουμε δύο nodes: το front_end_node και το SLAM.py. Το πρώτο είναι γραμμένο σε C++, ενώ το δεύτερο σε python. Η δουλειά του front_end_node είναι να παίρνει τα δεδομένα που παράγει το Kinect (RGB frame και Depth frame), να βρίσκει μοναδικά σημεία μέσα στο RGB frame, να υπολογίσει τις αντίστοιχες 3-D συντεταγμένες του κάθε σημείου, χρησιμοποιώντας το Depth frame, σε σχέση με το τοπικό frame και να στέλνει στο SLAM.py έναν συνδυασμό καινούριων landmarks και παλαιών (σύμφωνα με το τι του έχει είδη στείλει).

Μια άλλη σημαντική λειτουργία που προσφέρει το ROS είναι τα topics. Τα topics επιτρέπουν την επικοινωνία ανάμεσα σε nodes. Ένα node μπορεί να κάνει “publish” σε ένα topic. Ύστερα, όποιο node είναι “subscribed” σε αυτό το topic, θα λάβει το μήνυμα. Αργότερα θα εξηγήσουμε πώς μεταφέρουμε μηνύματα στο δίκτυο του ROS.

## 2.2. PCL

Το Point Cloud Library (PCL) είναι μια βιβλιοθήκη “ανοιχτού κώδικα” για την επεξεργασία 2D και 3D εικόνας καθώς και για την επεξεργασία point cloud. Είναι στην ουσία μια βιβλιοθήκη που περιέχει πολυάριθμο πλήθος αλγορίθμων που σχετίζονται με τα φίλτρα, με feature extraction, με την αναγνώριση επιφανειών, με το registration εικόνας, με το model fitting καθώς και με το segmentation της εικόνας. Η γλώσσα προγραμματισμού που χρησιμοποιεί είναι η C++ .

Οι αλγόριθμοί του PCL χρησιμοποιούνται σε πολλές εφαρμογές. Μερικές από αυτές είναι η ένωση 3D point clouds, η δημιουργία αντίληψης για ρομποτικά συστήματα, το φιλτράρισμα



των δεδομένων για θόρυβο, ο εντοπισμός keypoints και των αντίστοιχων descriptors για την αναγνώριση αντικειμένων με βάση την γεωμετρία τους, την δημιουργία επιφανειών από point cloud καθώς και

την απεικόνιση (visualization) τους.

Το PCL είναι Cross-platform, δηλαδή μπορεί να γίνει compiled και deployed σε Linux, MacOS, Windows και Android/iOS. Επιπροσθέτως, το PCL είναι διαμοιρασμένο σε μικρότερα modules ώστε να γίνει χρήση μόνο των κομματιών που χρειάζεται ο χρήστης με αποτέλεσμα να καταλαμβάνει λιγότερο χώρο και να χρειάζεται λιγότερη επεξεργαστική ισχύ [2.2-1].

Το PCL υπάγεται στο BSD license [2.2-3]. Είναι ελεύθερο για κάθε εμπορική ή ερευνητική χρήση.

Η ανάπτυξη του PCL ξεκίνησε το Μάρτιο του 2010 από τον Willow Garage. Το project, που αρχικά βρισκόταν σε ένα sub-domain του Willow Garage, μετακινήθηκε στο [2.2-1 ] τον Μάρτιο του 2011 και η πρώτη σταθερή και επίσημη έκδοση του PCL (version 1.0) κυκλοφόρησε τον Μάιο του 2011. Το PCL αρχικά ξεκίνησε ως πακέτα για το ROS και στην συνέχεια εξελίχθηκε σε αυτόνομη βιβλιοθήκη.

## 2.3. OpenCV

### 2.3.1. Γενικές γνώσεις

Το OpenCV είναι μια βιβλιοθήκη που έχει μεγάλη ποικιλία συναρτήσεων που σκοπό έχουν την επίτευξη real time επεξεργασία εικόνας. Αρχικά ξεκίνησε από την Intel αλλά τώρα είναι κάτω από την tiseez. Η βιβλιοθήκη είναι cross-platform, δηλαδή μπορεί να γίνει compiled και deployed σε Linux, MacOS, Windows και Android/iOS. Επίσης είναι ελεύθερη για χρήση κάτω από ανοικτού κώδικα BSD license.

Η OpenCV (v.2) είναι γραμμένη σε C++ και η βασική γλώσσα διεπαφής (interface) είναι σε C++, ενώ η παλιότερη έκδοση του OpenCV ήταν σε C (για v.1). Πλέον υπάρχει πλήρης διεπαφή (interface) μέσω Python, Java και Matlab/Octave. Έχουν επίσης δημιουργηθεί wrappers για C#, Perl, Ch και Ruby. Παρόλα αυτά, όλες οι καινούριες λειτουργίες που προσφέρει γίνονται σε C++ αρχικά.

### 2.3.2. Τρόπος χρήση του OpenCV στην εργασία

Τώρα ας δούμε πως χρησιμοποιούμε εμείς το OpenCV για την εργασία μας.

Όπως έχει ήδη αναφερθεί, θα κατασκευάσουμε feature-based SLAM. Ύστερα, θα δούμε την σημασία που έχει για τον αλγόριθμο μας. Αυτό που μας ενδιαφέρει για την ώρα είναι να

μπορούμε να εντοπίσουμε μοναδικά *features* χρησιμοποιώντας το RGB stream βίντεο που μας προσφέρει το Kinect. Συγκεκριμένα, θα δούμε εδώ ποιους αλγορίθμους προσφέρει το OpenCV για αυτήν την δουλειά, ποιον επιλέξαμε και γιατί.

Αρχικά έγιναν δοκιμές χρησιμοποιώντας το SIFT. Τα αποτελέσματα ήταν αρκετά ικανοποιητικά όσον αφορά τον αριθμό των features που εντόπιζε αλλά ο χρόνος που χρειαζόταν για να εκτελέσει αυτήν την διαδικασία, ακόμη και με RGB frame χαμηλής ανάλυσης όπως αυτή που προσφέρει το Kinect, ήταν πολύ μεγάλος. Ο διάδοχος του SIFT είναι ο αλγόριθμος SURF. Μένει πιστό στις βασικές αρχές του SIFT, αλλά το πλεονέκτημα του SURF είναι ότι μπορεί να κάνει feature extraction σε αρκετά συντομότερο χρονικό διάστημα από το SIFT. Αυτό όμως το καταφέρνει θυσιάζοντας την ανεκτικότητα που έχει το SIFT σε αλλαγές φωτεινότητας, γωνίας και απόστασης.

Μετά από περαιτέρω έρευνα, βρέθηκε ο αλγόριθμος KAZE, αλλά και η εξέλιξη του, ο βελτιωμένος σε ταχύτητα AKAZE. Συγκρίνοντας KAZE με SIFT και, αντίστοιχα, AKAZE με SURF, μπορούμε να δούμε ότι και στις δύο περιπτώσεις η οικογένεια KAZE υπερισχύει όσον αφορά τα αποτελέσματα, ειδικά το AKAZE έναντι του SURF, παρόλο που σε θέματα ταχύτητας είναι παρόμοια. Μια ακόμη λεπτομέρεια που έχει ενδιαφέρον, αν και δεν μας επηρεάζει λόγο της ακαδημαϊκής φύσης της εργασίας, είναι ότι ο αλγόριθμός είναι open source. Για τους παραπάνω λόγους, αποφασίστηκε ότι θα γίνει χρήση του AKAZE αλγορίθμου.

Είναι σημαντικό να σημειωθεί ότι όταν λέμε ότι ο KAZE παράγει καλύτερα αποτελέσματα από τον SIFT αναφερόμαστε αποκλειστικά στην υλοποίηση που έχει γίνει αυτών των αλγορίθμων μέσα στο OpenCV, και όχι γενικά για τους αλγορίθμους αυτούς καθαυτούς.

Κάνοντας χρήση του AKAZE παίρνουμε key points που συνοδεύονται από ένα *descriptor*. Τα *key points* είναι οι συντεταγμένες του σημείου πάνω στην εικόνα. Το *descriptor* έχει τη λογική ενός ID, περιγράφοντας με "μοναδικό" τρόπο ένα σημείο (και τον περιβάλλοντα χώρο) και επιτρέπει να το εντοπίσουμε όταν το ξαναδούμε. Όταν πάρουμε ένα frame, κάνουμε χρήση του AKAZE για να βρούμε έναν αριθμό μοναδικών *key points* μαζί με τα αντίστοιχα *descriptors* τους. Στην συνέχεια αποθηκεύουμε αυτές τις δύο λίστες στην βάση δεδομένων μας – αυτό το ζευγάρι από λίστες ορίζεται ως ένα frame. Έτσι όταν πάρουμε ένα καινούριο frame, και βρούμε τα *key points* και *descriptors* του, τα συγκρίνουμε με κάθε frame αποθηκευμένο στην βάση δεδομένων. Ο λόγος που στην βάση δεδομένων αποθηκεύουμε τα

*key points* και *descriptors* ανά frame μαζί είναι ώστε να κάνουμε χρήση homography όταν γίνεται η σύγκριση. Αυτό μας δίνει πολύ πιο αξιόπιστα αποτελέσματα.

Έστω ότι έχουμε δύο frame (ένα από το database και ένα από το Kinect) και χρησιμοποιώντας το AKAZE έχουμε βρει τα *key points* και τα *descriptors* του καθενός, πως τα συγκρίνουμε μεταξύ τους να δούμε πόσα κοινά σημεία έχουν; Το OpenCV έχει υλοποιήσει δύο αλγορίθμους εύρεσης κοντινότερου σημείου: το FLANN και το Brute Force Matcher (BF). Το FLANN είναι πιο γρήγορο αλλά λιγότερο αξιόπιστο από το BF.

Στην Εικόνα 2.3.2 βλέπουμε το αποτέλεσμα της χρήσης του FLANN και στην Εικόνα 2.3.1 το αποτέλεσμα της χρήση του BF Matcher. Παρότι το BF ήταν πιο αργό, δεν ήταν τόσο πολύ ώστε η χρήση του να γίνει απαγορευτική. Το FLANN κερδίζει σε ταχύτητα όταν του δώσουμε δύο τεράστιες λίστες από *descriptors*, εμείς όμως δίνουμε ξεχωριστά την κάθε λίστα (δηλαδή το κάθε frame). Ο BF δέχεται τις λίστες με τα *key points* και *descriptors*, από τις οποίες συγκρίνει κάθε point της πρώτης λίστας με κάθε point στην δεύτερη λίστα, με σκοπό να βρει το πλέον παρόμοιο. Αυτό θα πει ότι κάθε point από την πρώτη λίστα θα βρει κάποιο από την δεύτερη. Στο αποτέλεσμα όμως, μαζί με τα ζευγάρια των points, μας δίνει και την απόσταση τους. Αυτή η απόσταση μας λέει πόσο κοντά είναι το ένα σημείο με το άλλο. Η χρήση αυτής της πληροφορίας μας επιτρέπει μετά να φιλτράρουμε τα αποτελέσματα μας απορρίπτοντας κάθε ζευγάρι από points που έχει πολύ μεγάλη απόσταση. Τέλος, χρησιμοποιώντας homography, φιλτράρουμε ακόμη μια φορά τα σημεία μας ώστε να είμαστε σίγουροι ότι αυτά τα σημεία όντως είναι τα ίδια ή τουλάχιστον πάρα πολύ κοντά.

Εικόνα 2.3.1 AKAZE with Brute Force Matcher

Εικόνα 2.3.2 AKAZE with FLANN

## 2.4. MATLAB

Το MATLAB είναι ένα περιβάλλον μαθηματικών υπολογισμών και θεωρείται προγραμματιστική γλώσσα τέταρτης γενιάς. Το MATLAB δουλεύει με πίνακες και επιτρέπει το plotting συναρτήσεων και δεδομένων, την εφαρμογή αλγορίθμων, την δημιουργία γραφικού περιβάλλοντος (User Interface) και την διεπαφή με άλλες γλώσσες προγραμματισμού όπως C, C++, Java, Fortran και Python [2.4-1].

Χρησιμοποιείται κατά κύριο λόγο για την επίλυση μαθηματικών προβλημάτων, ωστόσο είναι πολύ "ισχυρό" και μπορεί να χρησιμοποιηθεί και για προγραμματισμό καθώς περιέχει εντολές από την C++ όπως την while, την switch και την if. Διαθέτει επίσης μερικά toolboxes που επιτρέπουν και άλλου είδους υπολογισμούς πέρα των μαθηματικών προβλημάτων. Για παράδειγμα, υπάρχει το MuPAD που επιτρέπει "συμβολικό προγραμματισμό" ή το Simulink που επιτρέπει με γραφικό τρόπο εξομοιώσεις διαφόρων μοντέλων σχεδίασης για δυναμικά και ενσωματωμένα συστήματα.

## 2.5. Kinect

Το Kinect [2.5-1] είναι μια συσκευή που αποτελείται από μια σειρά αισθητήρων κίνησης,

φτιαγμένο από την Microsoft για το Xbox360, το Xbox One και για το PC . Το Kinect πρωτοκυκλοφόρησε το Νοέμβριο του 2010 και είχε σαν βασικό στόχο να ανταγωνιστεί αισθητήρες

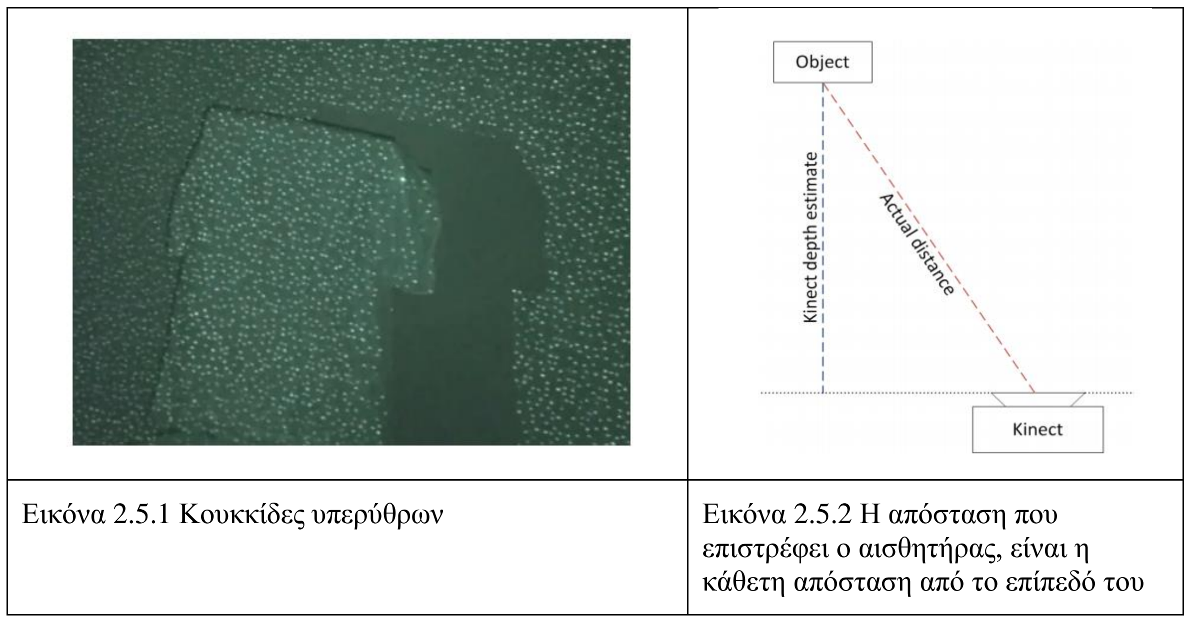
κίνησης για παιχνιδομηχανές της Sony και της Nintendo.

Το Kinect περιέχει μία RGB camera, μια IR κάμερα βάθους καθώς και τέσσερα μικρόφωνα παράλληλα συνδεδεμένα. Η RGB-κάμερα μεταφέρει την εικόνα με ταχύτητα από 9 FPS μέχρι 30 FPS και ανάλυση 8-bit VGA δηλαδή 640x480 pixels με την βοήθεια του Bayer color filter, ωστόσο η RGB-κάμερα έχει την δυνατότητα ανάλυσης μέχρι τα 1280x1024 με χαμηλότερη συχνότητα frame και άλλο format κάμερας όπως είναι το UYVY. Η IR κάμερα έχει ανάλυση 640x480 pixels ή 1280x1024 pixels με μικρότερη συχνότητα απόκτησης frame.

Ο αισθητήρας βάθους του Kinect έχει εύρος 0.7m – 4m θεωρητικά , ωστόσο στην πράξη το εύρος του είναι 1m – 3.5m. Επιπροσθέτως, το Kinect από μόνο του έχει γωνία πεδίου όρασης 57 μοίρες οριζόντια και 43 μοίρες κάθετα, και χάρη στον ενσωματωμένο κινητήρα του έχει την δυνατότητα να μετατοπιστεί 27 μοίρες πάνω και κάτω.

Συνδέεται με τον υπολογιστή σειριακά μέσω USB2.0 και χρειάζεται τροφοδοσία για να μπορέσει να λειτουργήσει.

Όπως έχει αναφερθεί παραπάνω, το Kinect έχει την δυνατότητα να δημιουργεί 3D χάρτες βάθους μιας σκηνής, σε πραγματικό χρόνο. Μια δομή από σημεία υπέρυθρου φωτός προβάλλεται στον χώρο και ένας αισθητήρας εικόνας CMOS δέχεται τις ανακλώμενες ακτίνες. Το PS1080 SoC – chip που περιέχει το σύστημα του Kinect, ελέγχει τη δομή αυτή και επεξεργάζεται τα δεδομένα από τον αισθητήρα CMOS παράγοντας δεδομένα βάθους σε πραγματικό χρόνο. Πιο συγκεκριμένα, εκπέμπεται ένα IR μοτίβο (Εικόνα 2.5.1) από τον πομπό υπερύθρων, το οποίο στη συνέχεια λαμβάνεται από μία CMOS κάμερα. Αυτή η κάμερα έχει ένα ζωνοπερατό φίλτρο που επιτρέπει τις IR ακτίνες να περάσουν. Ο επεξεργαστής του Kinect χρησιμοποιεί τις σχετικές θέσεις των κουκκίδων του IR – μοτίβου για να υπολογίσει το βάθος στο οποίο βρίσκεται το κάθε pixel της εικόνας και έτσι τελικά σχηματίζεται η 3D αναπαράσταση του χώρου. Εδώ πρέπει να σημειωθεί ότι οι τιμές βάθους που επιστρέφονται από το Kinect είναι η κάθετη απόσταση από το επίπεδο του αισθητήρα όπως φαίνεται και στην Εικόνα 2.5.2.




| Εικόνα 2.5.1 Κουκκίδες υπερύθρων | Εικόνα 2.5.2 Η απόσταση που επιστρέφει ο αισθητήρας, είναι η κάθετη απόσταση από το επίπεδό του |
|---|---|

Η μέγιστη ανάλυση της εικόνας βάθους που παράγει το PS1080 είναι 640x480, με συχνότητα 30 FPS. Στα 2 μέτρα απόστασης από τον αισθητήρα, έχει τη ακρίβεια 3 χιλιοστών σε ύψος και πλάτος και 1 εκατοστό σε βάθος. Η εμβέλεια ορθής λειτουργίας είναι από 1.0 μέχρι 3.5 μέτρα. Τα δεδομένα βάθους (depth data stream) παρέχουν frames στα οποία το κάθε pixel περιέχει την καρτεσιανή απόσταση (σε χιλιοστά) από την επιφάνεια της κάμερας μέχρι το κοντινότερο αντικείμενο στις συγκεκριμένες x και y συντεταμένες, στο οπτικό πεδίο του αισθητήρα. Κάθε pixel στο depth stream χρησιμοποιεί 13 bits για το βάθος. Η τιμή βάθους 0 υποδεικνύει ότι δεν υπάρχουν δεδομένα βάθους για το συγκεκριμένο σημείο, γιατί το αντικείμενο που βρίσκεται σε αυτή τη θέση είναι είτε πολύ κοντά, είτε πολύ μακριά από τον αισθητήρα.

Υπάρχουν οι εξής 3 drivers για το Kinect:

Το ένα είναι το “Kinect for Windows SDK” που είναι και το επίσημο της Microsoft. Αυτό όμως δεν τρέχει στο επιθυμητό λειτουργικό σύστημα, που είναι το Linux Ubuntu.

Επιπροσθέτως, υπήρχε και το OpenNI 1 της PrimeSence. Το OpenNI 1 ήταν ένα open source project με σκοπό να επιτρέπει την ανάπτυξη εφαρμογών για 3D κάμερες (π.χ. Kinect, Asus XTionPRO κτλ.). Το OpenNI 1 έτρεχε σε Linux, έτσι ήταν εφικτό να λειτουργήσει το Kinect σε Linux. Όμως αφού η Apple αγόρασε την PrimeSense, σταμάτησε να υποστηρίζεται το OpenNI 1. Παρόλο που δημιουργήθηκε το OpenNI 2, απαιτούσε και την εγκατάσταση του Kinect for Windows SDK για να λειτουργήσει με το Kinect, κάτι που δεν γίνεται να τρέξει πάνω σε Linux.

Τέλος υπάρχει το libfreenect. Μετά από το reverse engineering του Kinect από μια ομάδα hackers, δημιουργήθηκε το Open Kinect project με σκοπό την ανάπτυξη ενός open source driver για το Kinect. Το libfreenect 1 (που είναι για το Kinect 1.0) επέτρεπε να λειτουργεί κανονικά το Kinect σε Linux με εξαίρεση το motor control. Παρόλο που αυτός ο driver βγήκε μέσω reverse engineering, η Microsoft επισήμως έχει πει ότι όσο χρησιμοποιείται για ερευνητικό σκοπό και όχι εμπορικό, δεν υπάρχει πρόβλημα με την χρήση αυτού του driver, από νομικής πλευράς. Εφόσον υπάρχει ROS πακέτο που επιτρέπει εύκολα να πάρουμε τα δεδομένα του Kinect μέσω του libfreenect, επιλέχτηκε αυτός ο driver για την εργασία μας.

## 2.6. Υπολογιστής πειραμάτων

Ο υπολογιστής στον οποίο κάναμε όλα τα πειράματα είναι ο Acer ASPIRE 5738G. Τα χαρακτηριστικά του είναι:

- Intel Core 2 Duo processor (2.2 GHz, 800 Hz FSB) – Centrino
- 4 GB RAM
- ATI Mobility Radeon HD 4570 Up to 2304 MB HyperMemory
- 240 GM SSD

# Κεφάλαιο 3. Φίλτρα και μαθηματικά μοντέλα

Σε αυτό το κεφάλαιο θα περιγράψουμε τα φίλτρα που θα χρησιμοποιήσουμε και τα μαθηματικά μοντέλα μας που θα περιγράφουν την κίνηση του ρομπότ.

## 3.1. Bayes Filter – Bayes Θεώρημα

### 3.1.1. Bayes Filter

Το Bayes Filter [3.1.1-1] είναι ο πιο γενικός αλγόριθμος για τον υπολογισμό της πεποίθησης (Belief). Υπολογίζει την κατανομή πεποίθησης bel από τις μετρήσεις και τα δεδομένα ελέγχου του συστήματος.

Ο αλγόριθμος του Bayes Filter για κάθε μεταβλητή είναι:

$$bel(x_t) = \int p(\, x_t |\, u_t \,, x_{t-1}) * bel(x_{t-1}) ds_{t-1} \quad (3.1.1.1)$$

$$bel'\,(x_t) = n\, p\,(\, z_t | x_t\,) * bel(\, x_t) \quad (3.1.1.2)$$

Το φίλτρο Bayes λειτουργεί αναδρομικά. Χρησιμοποιώντας σαν είσοδο την πεποίθηση τη χρονική στιγμή t-1 καθώς και τα πιο πρόσφατα δεδομένα ελέγχου $u_t$ και των μετρήσεων $z_t$ και υπολογίζει την πεποίθηση $bel'(x_t)$ τη χρονική στιγμή t.

Ο αλγόριθμος χωρίζεται σε δύο βασικά βήματα:

- Πρώτο βήμα είναι η πρόβλεψη, κατά το οποίο προβλέπει πόσο είναι η τιμή της πεποίθησης $bel(x_t)$ βασισμένη στην προηγούμενη πεποίθηση $bel(x_{t-1})$ και στα δεδομένα ελέγχου $u_t$ .
- Δεύτερο βήμα είναι η διόρθωση, όπου η μεταγενέστερη πεποίθηση υπολογίζεται πολλαπλασιάζοντας την $bel(\, x_t)$ με την πιθανότητα να έχουμε παρατηρήσει την κάθε μέτρηση. Ο συντελεστής η κανονικοποιεί την τιμή ώστε η πιθανότητα να έχει μέγιστο τη μονάδα.

### 3.1.2. Bayes Θεώρημα

Στη θεωρία πιθανοτήτων και στη στατιστική, το θεώρημα Bayes [3.1.1-2] ή νόμος Bayes ή κανόνας Bayes, σχετίζει την τρέχουσα πιθανότητα με την αρχική πιθανότητα του να συμβεί ένα γεγονός. Δηλαδή περιγράφει την πιθανότητα ενός γεγονότος, με βάση τις συνθήκες που θα μπορούσαν να σχετίζονται με το συμβάν αυτό.

Το θεώρημα Μπέυζ ορίζεται μαθηματικά με την ακόλουθη εξίσωση :

$$P\ (A/B) = \frac{P(B|A)P(A)}{P(A)} \quad (3.1.2.1)$$

όπου A και B είναι γεγονότα.

- P(A) και P(B) είναι οι πιθανότητες των A και B που είναι ανεξάρτητες μεταξύ τους.
- P(A | B), η υπό συνθήκη πιθανότητα, είναι η πιθανότητα του A δεδομένου του B να είναι αληθής.
- P(B | A), είναι η πιθανότητα του B δεδομένου του A να είναι αληθής.

## 3.2. Kalman Filter

Το φίλτρο Kalman αναπτύχθηκε από τον Rudolf E. Kalman [3.2-1] το 1960 ως ένας 'αλγόριθμος' βέλτιστου υπολογισμού της κατάστασης ενός συστήματος, το οποίο υπόκειται σε θορύβους που καθιστούν τα συμπεράσματα που λαμβάνουμε αναξιόπιστα. Στην ουσία έχει σαν σκοπό τον «καθαρισμό» των μετρήσεων που γίνονται από ένα σύστημα και τη δημιουργία μίας νέας εκτίμησης της κατάστασης του συστήματος, απογυμνωμένη από διαταραχές. Με τον όρο αλγόριθμος, εννοούμε πως είναι ένα σύνολο μαθηματικών εξισώσεων που παρέχει αποτελεσματικά υπολογιστικά (επαναληπτικά) μέσα για την εκτίμηση της κατάστασης μιας διαδικασίας, κατά τρόπο που να ελαχιστοποιείται ο μέσος όρος των τετραγώνων των σφαλμάτων. Είναι επίσης βέλτιστος, από την άποψη πως ελαχιστοποιεί τη διακύμανση του σφάλματος μεταξύ κατάστασης και εκτίμησης.

Αρχικά οι ιδέες του Kalman, αναφορικά με τη λειτουργικότητα και την αποτελεσματικότητα του φίλτρου είχαν αντιμετωπιστεί με σκεπτικισμό. Μάλιστα, αν και ο ίδιος ήταν ηλεκτρολόγος, αναγκάστηκε να πρωτοπαρουσιάσει την εργασία του σε ένα περιοδικό για μηχανολόγους μηχανικούς. Τελικά όμως το φίλτρο εφαρμόστηκε στην πράξη με επιτυχία. Η ενέργεια αυτή αποδίδεται στον Stanley F. Schmidt, ο οποίος κατά την επίσκεψη του Kalman στο NASA Ames Research Center, είδε τη δυνατότητα εφαρμογής των ιδεών του Kalman στο πρόβλημα της εκτίμησης της τροχιάς του σκάφους του Apollo Program. Το φίλτρο Kalman λοιπόν έπαιξε το ρόλο του στην πρώτη ιστορική αποστολή ανθρώπων στο φεγγάρι. Άξια αναφοράς είναι και η συμβολή του Richard R. Bucy, ο οποίος εργάστηκε μαζί με τον Kalman στη θεωρητική επέκταση του φίλτρου και στον συνεχή χρόνο, γνωστό ως Kalman-Bucy Filter.

Ο όρος αναδρομικός αλγόριθμος ή φίλτρο, αναφέρεται στην ιδιότητα του να χρησιμοποιεί μέρος προηγούμενης πληροφορίας χωρίς να χρειάζεται όλη η προσφερόμενη πληροφορία να αποθηκευτεί για να επεξεργαστεί ξανά κάθε φορά που μια καινούρια μέτρηση παρουσιάζεται. Συχνά όμως σε ένα σύστημα, οι μεταβλητές οι οποίες περιγράφουν την κατάσταση του, συμβαίνει να μην μπορούν να μετρηθούν είτε άμεσα είτε αξιόπιστα. Κατά συνέπεια, είναι απαραίτητο να μπορούν να εξαχθούν συμπερασματικά οι τιμές αυτών των μεταβλητών από οποιαδήποτε σχετική διαθέσιμη πληροφορία για αυτές. Εκτός των άλλων, είναι γνωστό πως σε κάθε μέτρηση εισάγεται, ως ένα βαθμό, κάποιο σφάλμα. Αυτό το σφάλμα προκύπτει συνήθως ως συνάρτηση, θορύβου προερχόμενου από το ίδιο το σύστημα ελέγχου και φυσικά σφαλμάτων που εισάγουν οι ίδιες οι συσκευές μέτρησης. Όλα τα παραπάνω λοιπόν, τονίζουν την αναγκαιότητα ύπαρξης κάποιου φίλτρου, ικανού να εξαλείφει κάθε παράγοντα αποπροσανατολισμού ή παραμόρφωσης της εκτίμησης της κατάστασης του συστήματος. Το φίλτρο ή αλγόριθμος Kalman, καλύπτει την παραπάνω αναγκαιότητα, συνδυάζοντας κάθε νέα παρεχόμενη μέτρηση μαζί με οποιαδήποτε πληροφορία για την κατάσταση του συστήματος και των συσκευών μέτρησης. Έτσι παρέχει μια βέλτιστη εκτίμηση των μεταβλητών που ενδιαφέρουν το σύστημα, κατά τέτοιο τρόπο ώστε το εκάστοτε σφάλμα να ελαχιστοποιείται στατιστικά.

Στην ουσία είναι ένα υποσύνολο του φίλτρου Bayes όπου ισχύουν οι υποθέσεις Gaussian κατανομής και ότι η τρέχουσα κατάσταση εξαρτάται γραμμικά από την προηγούμενη. Σε περίπτωση όμως που κάποιο γραμμικό μοντέλο δεν επαρκεί λόγω μη-γραμμικής συμπεριφοράς του συστήματος, τότε η πιο ασφαλής προσέγγιση προστάζει να προσεγγιστεί όσο πιο γραμμικά γίνεται το παρατηρούμενο σύστημα, δημιουργώντας κάποιο μοντέλο σφάλματος για το εν θόρυβο σύστημα. Αυτό το πετυχαίνει το διακριτό φίλτρο Kalman.

Η ανάπτυξη του φίλτρου λοιπόν, αποτελεί ένα από τα σημαντικότερα κατορθώματα του προηγούμενου αιώνα, τουλάχιστον από επιστημονικής απόψεως. Δεδομένου πως οποιαδήποτε διεργασία ή μέτρηση διαφθείρεται από θόρυβο, δε θα είχαμε τη δυνατότητα να αναπτύξουμε πολλές εφαρμογές αν δεν βρίσκαμε τρόπο να προβούμε σε αποφάσεις, αποβάλλοντας την αβεβαιότητα που δίνει ο θόρυβος. Επιπροσθέτως υποστηρίζει εκτιμήσεις του παρελθόντος, του παρόντος, και ακόμη και τις μελλοντικές καταστάσεις, και μπορεί να το πράξει ακόμη όταν η ακριβής φύση του διαμορφωμένου συστήματος είναι άγνωστη. Χρησιμοποιείται σε εφαρμογές όπως η εκτίμηση θέσης ενός αυτοκινήτου (GPS), η εξομάλυνση (smoothing) μηνυμάτων (ήχου είτε εικόνας) που έχουν φθαρεί από θόρυβο, σε Radar, γενικά σε οποιαδήποτε τεχνολογική διαδικασία απαιτείται υψηλή ακρίβεια δεδομένων.

Επίσης βρίσκει εφαρμογή σε οικονομικά μοντέλα, όπως τα Autoregressive-movingaverage-models για τη μελέτη και την πρόβλεψη της συμπεριφοράς χρονοσειρών αλλά και στη μελέτη βιολογικών δομών, όπως μοντέλα πληθυσμών, μεταβολισμού και συγκεντρώσεις στοιχείων (π.χ. χλωροφύλλη) σε μεγάλους υδάτινους όγκους (ποτάμια, λίμνες).

Το μαθηματικό μοντέλο του Kalman filter σε πραγματικό χρόνο k προκύπτει από την κατάσταση την χρονική στιγμή k-1 και είναι :

$$x_k = F_k x_{k-1} + B_k u_k + w_k \quad (3.1.2.1)$$

Όπου

- $x_k$ το NxN διάνυσμα κατάστασης
- $F_k$ ο NxN πίνακας μετάβασης από τη χρονική στιγμή $t_{k-1}$ στην $t_k$ βασισμένος στην προηγούμενη κατάσταση $x_{k-1}$
- $u_k$ το NxN διάνυσμα ελέγχου.
- $B_k$ ο NxN πίνακας του μοντέλου έλεγχου, το οποίο εφαρμόζεται στο διάνυσμα $u_k$
- $w_k$ ένα διάνυσμα Nx1, το οποίο είναι ο θόρυβος του συστήματος και θεωρούμε ότι είναι στατικό με διακύμανσης Q.

$$w_k \sim N(0, Q)$$

Για μια συγκεκριμένη στιγμή k η μέτρηση $z_k$ μιας πραγματικής κατάστασης $x_k$ βρίσκεται σύμφωνα με τον τύπο :

$$z_k = H_k x_k + v_k \quad (3.1.2.2)$$

Όπου

- $H_k$ είναι λxn πίνακας που ονομάζεται πίνακας μετρήσεων, αφού δίνει τις μετρήσεις των $z_k$ από τα $x_k$ , απουσία θορύβου.
- $v_k$ είναι λx1 πίνακας που περιέχει το σφάλμα μετρήσεων. Είναι δηλαδή ο θόρυβος παρατήρησης, όπου θεωρείται ότι η Gaussian του έχει μέση τιμή μηδέν με διακύμανσης $R_k$.

$$v_k \sim N(0, R_k)$$

Το λx1 διάνυσμα $z_k$ είναι ένας γραμμικός συνδυασμός των γνωστών καταστάσεων $x_k$ όπου εισέρχεται και ένα σφάλμα μετρήσεων $v_k$ που επίσης είναι μια λx1 πολυδιάστατη στατική στοχαστική διαδικασία με χαρακτηριστικά λευκού θορύβου.

## 3.3. Εκτεταμένο Φίλτρο Kalman (Extended Kalman – EKF)

Ο αλγόριθμος Kalman που παρουσιάστηκε στην προηγούμενη ενότητα αφορά την περίπτωση κατά την οποία οι εξισώσεις, οι οποίες περιγράφουν την διαδικασία μέτρησης και την δυναμική κατάσταση του στόχου είναι γραμμικές. Στη περίπτωση μη γραμμικών συστημάτων εφαρμόζεται μια επέκταση του Kalman φίλτρου, που καλείται Extended Kalman Filter (EKF) [3.2-1]. Η γενική τακτική του EKF είναι να διαμορφώνει όσο το δυνατόν πιο γραμμικά (linearize) την τελευταία *μέση τιμή* (mean) και της διακύμανσης (covariance). Σε πολλές πρακτικές εφαρμογές η συνάρτηση του συστήματος μπορεί να περιγράφεται από μία μη γραμμική στοχαστική διαφορική εξίσωση ή η εξίσωση μέτρησης να είναι και αυτή μη γραμμική. Στις εφαρμογές αυτές, το φίλτρο Kalman δεν μπορεί να δώσει βέλτιστη εκτίμηση αλλά η επέκτασή του (EKF) μέσω γραμμικοποίησης με τη χρήση Ιακωβιανών πινάκων ( Jacobian ), προσεγγίζει σε κάποιο βαθμό τη βελτιστότητα.

Σε σύγκριση με το διακριτό φίλτρο όμως το Εκτεταμένο Φίλτρο Kalman έχει το μειονέκτημα του ότι οι διάφορες μεταβλητές που περιγράφουν το σύστημα παύουν να ακολουθούν κανονική κατανομή τη στιγμή που διαφοροποιούνται μη- γραμμικά. Για το λόγο αυτό το EKF φίλτρο εκτιμά την κατάσταση του συστήματος προσεγγίζοντάς την γραμμικά με γνώμονα την ευνοϊκότερη δυνατή συνθήκη, κατά Bayes.

## 3.4. Particle Filter

Ήδη από την δεκαετία του '90 το Particle Filter (PF) ή αλλιώς Sequential Monte Carlo (SMC) έχει αρχίσει να χρησιμοποιείται ως μια δημοφιλής μαθηματική μέθοδος για την επίλυση προβλημάτων βέλτιστης εκτίμησης (optimal estimation problems).

Το Particle Filter [3.4-1] ανήκει στην κατηγορία των γενετικών αλγόριθμων. Δηλαδή αποτελεί μια μέθοδο αναζήτησης βέλτιστων λύσεων σε συστήματα που μπορούν να περιγραφούν ως μαθηματικό πρόβλημα.

Χρησιμοποιείται σε προβλήματα που είναι κυρίως μη γραμμικά και μη Gaussian. Αυτά τα προβλήματα περιέχουν πολλές παραμέτρους/διαστάσεις και δεν υπάρχει αναλυτική μέθοδος

που να μπορεί να βρει το βέλτιστο συνδυασμό τιμών για τις μεταβλητές ώστε το υπό εξέταση σύστημα να αντιδρά όσο το δυνατόν πιο κοντά στο επιθυμητό τρόπο. Μπορεί να βρει εφαρμογή στην επεξεργασία εικόνας, εντοπισμός θέσης, ευέλικτο περιβάλλον καθώς επίσης σε ένα πολυάριθμο πλήθος άλλων εφαρμογών.

Το PF χωρίζεται σε 4 βήματα:

1. Στην αρχή παράγουμε κάποια particles στο σύστημα, που το καθένα αναπαριστά έναν τυχαίο συνδυασμό των παραμέτρων του συστήματος, με ομοιόμορφη διακύμανση.
2. Στην συνέχεια παίρνουμε μια μέτρηση από το σύστημα
3. Έπειτα συγκρίνουμε τα particles μας με την μέτρηση που πήραμε, ώστε να δούμε ποια από τα particles είναι πιο κοντά στην πραγματική κατάσταση του συστήματος. Όσο πιο κοντά στην μέτρηση τόσο ανεβαίνει η αξία τους.
4. Τέλος, αναπαράγουμε μια καινούρια γενιά particles, όπως στο βήμα 1, αλλά παίρνουν την αξία που είχα τα προηγούμενα particles. Μετά την καινούρια μέτρηση, μεταβάλλεται (προς τα πάνω ή προς τα κάτω) η αξία του κάθε particle (όπως στο βήμα 3).
5. Επαναλαμβάνεται συνεχώς αυτή η διαδικασία μέχρι μερικά particles να έχουν ικανοποιητικά υψηλό βαθμό αξίας

## 3.5. Motion Model

Σε αυτό το κεφάλαιο θα συζητήσουμε για τα probabilistic motion models που υπάρχουν, ποιο επιλέξαμε εμείς και πως το υλοποιήσαμε. Η γενική μορφή του είναι:

$$p(s_t|s_{t-1}, u_t) \quad (3.1.2.1)$$

Ο όρος $p(s_t|s_{t-1}, u_t)$ προσδιορίζει μια μεταγενέστερη πιθανότητα (posterior probability) , όπου η εντολή $u_t$ μετατοπίζει το ρομπότ απο την θέση $s_{t-1}$ στην $s_t$.

Η κινηματική είναι η επιστήμη που μελετάει την κίνηση των σωμάτων, όταν έχει δοθεί σε αυτό το σώμα μια εντολή κίνησης. Μια σημαντική σημείωση είναι ότι η κινηματική δεν λαμβάνει υπόψιν τις δυνάμεις ή τις ροπές που την προκαλούν. Με την χρήση του probabilistic robotics, γενικεύονται οι κινηματικές εξισώσεις ώστε το αποτέλεσμα να επηρεάζεται από το σφάλμα. Το σφάλμα μπορεί να οφείλεται σε θόρυβο που είχε η εντολή

κίνησης ή φυσικές παρεμβολές όπως κινητήρες χωρίς ακρίβεια ή δάπεδα κίνησης που γλιστράνε.

### 3.5.1. Αρχικοποίηση του κινηματικού μοντέλου

Στην δικιά μας περίπτωση, η στάση (pose) του ρομπότ χαρακτηρίζεται από τρισδιάστατες καρτεσιανές διαστάσεις (x, y και z) και τρεις Euler γωνίες ( roll, pitch και yaw). Παρόλα αυτά, επιλέξαμε να υποθέσουμε ότι κάνουμε χρήση ενός Differential Drive. Αυτό θα πει ότι έχουμε δύο ρόδες που κινούν το ρομπότ μας. Άρα, είναι αδύνατον το όχημα, με εντολή, να μπορέσει να κινηθεί κατά τον z άξονα. Η κινηματική από μόνης της δεν μπορεί να το προβλέψει αυτό και εδώ είναι που η πιθανοθεωρητική ρομποτική (probabilistic robotics) βοηθάει. Το σφάλμα που εφαρμόζουμε στο motion model, θα μεταβάλλει την πιθανή θέση του ρομπότ κατά τον z άξονα και ως προς το roll και pitch. Έτσι εάν ανεβεί το ρομπότ έναν λόφο, θα μπορεί να το λάβει υπόψιν του. Το πώς θα εφαρμοστεί στο SLAM αλγόριθμο θα το δούμε σε επόμενο κεφάλαιο.

$$\begin{pmatrix} x \\ y \\ z \\ roll \\ pitch \\ yaw \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ \psi \\ \varphi \\ \theta \end{pmatrix}$$

Πίνακας 3.5.1 Για ευκολία στους τύπους που θα παρουσιάσουμε, κάνουμε τον παραπάνω συσχετισμό.

### 3.5.2. Probabilistic Kinematics

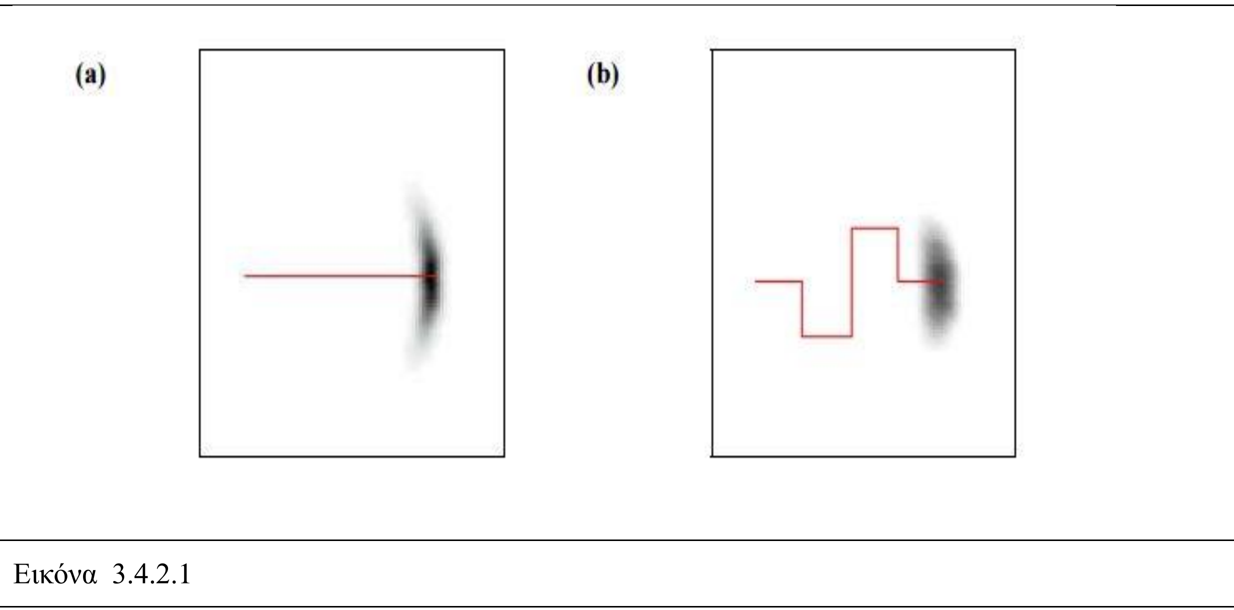
Το πιθανοθεωρητικό κινηματικό μοντέλο (Probabilistic Kinematics model) ή motion model είναι υπεύθυνο να αλλάζει το pose στο Probabilistic robotics. Ας ξαναδούμε τον τύπο μας:

$$p(s_t|s_{t-1}, u_t) \quad (3.5.2.1)$$

Όπου:

- $s_t$ → καινούρια θέση ρομπότ
- $s_{t-1}$ → παλιά θέση ρομπότ
- $u_t$ → εντολή κίνησης

Αυτό το μοντέλο περιγράφει την κατανομή που θα έχει το pose μετά την εφαρμογή του κινηματικού μοντέλου με την εντολή $u_t$.




Εικόνα 3.4.2.1

Στην εικόνα 3.4.2.1 μπορείτε να δείτε δύο παραδείγματα για ρομπότ που λειτουργούν σε δισδιάστατη επιφάνεια. Και στα δύο σενάρια έχουν την ίδια αρχική θέση $s_{t-1}$ αλλά λόγω των κινήσεων έχουμε άλλη εκτίμηση της κατανομής. Όσο πιο σκούρο γκρι είναι ο χώρος, τόσο πιο πιθανό να βρίσκεται το ρομπότ εκεί.

Υπάρχουν δύο είδη μοντέλων που μπορούμε να χρησιμοποιήσουμε.

- Odometry Motion Model
- Vector Motion Model

Η διαφορά είναι ότι στο vector model ξέρουμε την αρχική θέση μας $s_{t-1}$, δίνουμε μια εντολή κίνησης $u_t$ και υποθέτουμε πού θα καταλήξουμε $s_t$. Αντιθέτως, το Odometry model απαιτεί να έχει γνώση της κίνησης που κάναμε, πχ μέσω encoder, και μετά να υπολογίσουμε την πιθανή θέση στην οποία βρισκόμαστε.

### 3.5.3. Velocity Motion Model

Στο velocity model δίνουμε τις συντεταγμένες του ρομπότ και τι εντολή θέλουμε να εκτελέσει. Μετά απλά υπολογίζουμε στην οποία θα έπρεπε να βρίσκεται, παίρνοντας υπόψιν μας μερικά σφάλματα. Ένα σημαντικό πλεονέκτημα που έχει αυτό το μοντέλο έναντι του Odometry model είναι ότι επιτρέπει να σχεδιάζεις διαδρομές και να αποφύγεις κινούμενα

αντικείμενα. Αυτό γίνεται διότι στο odometer model πρώτα πραγματοποιείται η κίνηση, στην συνέχεια διαβάζεις τους encoder και τέλος υπολογίζει που πρέπει να βρίσκεσαι.

Οι εντολές που δίνουμε έχουν την εξής μορφή:

$$u_t = \begin{pmatrix} v_t \\ \omega_t \end{pmatrix} \quad (3.5.3.1)$$

Όπου:

- $v_t$ → ευθύγραμμη ταχύτητα
- $\omega_t$ → γωνιακή ταχύτητα

Το motion model είναι ως εξής:

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} -\frac{v}{\omega} sin(\theta) + \frac{v}{\omega} sin(\theta + \omega \Delta t) \\ \frac{v}{\omega} cos(\theta) - \frac{v}{\omega} cos(\theta + \omega \Delta t) \\ \omega \, \Delta t \end{bmatrix} \quad (3.5.3.2)$$

Στον πίνακα 3.5.2 μπορούμε να δούμε τον αλγόριθμο για την παραγωγή την νέας θέσης. Όπου βλέπουμε sample(), είναι η Gaussian συνάρτηση με κέντρο το μηδέν και τυπική απόκλιση την είσοδο της συνάρτησης. Τα «α» είναι ειδικές παράμετροι σφάλματος.

| $sampeling_pose(x_{t-1}, u_t)$:<br><br>$v' = v + sample(\alpha_1 \lvert v \rvert + \alpha_2 \lvert \omega \rvert)$<br>$\omega' = \omega + sample(\alpha_3 \lvert v \rvert + \alpha_4 \lvert \omega \rvert)$<br>$\xi' = sample(\alpha_7)$<br>$\beta' = sample(\alpha_8)$<br>$k' = sample(\alpha_9)$<br>$\gamma' = sample(\alpha_5 \lvert v \rvert + \alpha_6 \lvert \omega \rvert)$<br>$x' = x - \frac{v'}{\omega'} \sin(\theta) + \frac{v'}{\omega'} \sin(\theta + \omega' \Delta t)$<br>$y' = y + \frac{v'}{\omega'} \cos(\theta) - \frac{v'}{\omega'} \cos(\theta + \omega' \Delta t)$<br>$z' = z + \xi' \Delta t$<br>$\psi' = \psi + \beta' \Delta t$<br>$\varphi' = \varphi + \kappa' \Delta t$<br>$\theta' = \theta + \omega' \Delta t + \gamma' \Delta t$<br>$return\ s_t = (x', y', z', \psi', \varphi', \theta')^T$ |
|---|
| Πίνακας 3.5.2 |

**Σφάλμα – συνάρτηση sample()**

Τα Typical Distributions for Probabilistic Motion Models δίνεται από την ακόλουθη Gaussian εξίσωση:

$$\varepsilon_{\sigma^2}(x) = \frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-\frac{1x^2}{2\sigma^2}} \quad (3.5.3.3)$$

| Sample(b): $k = \frac{b}{6}\sum_{i=1}^{12} rand(-1,1)$ return $k$ |
|---|

Εικόνα 3.5.1 Normal distribution

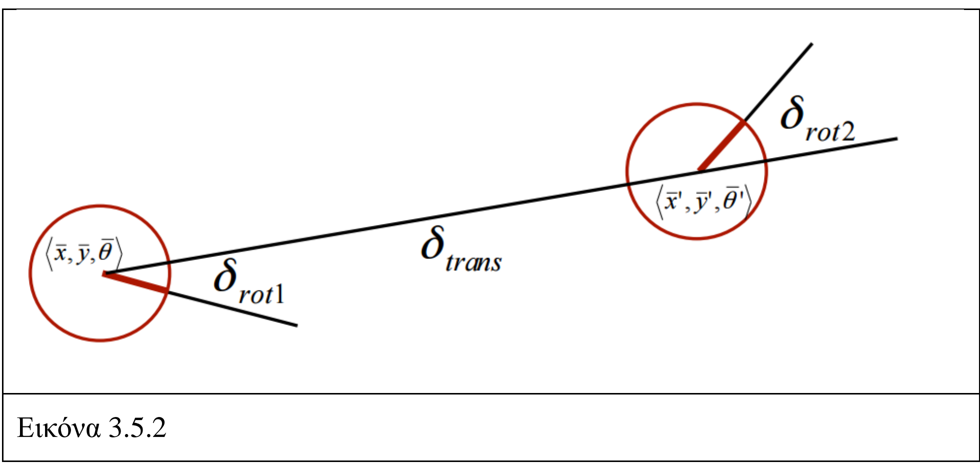
Αλλά επειδή αυτή η πράξη είναι πολύ ακριβή, κάνουμε μια προσέγγιση της χρησιμοποιώντας το θεώρημα κεντρικού ορίου.

| Sample(b):<br>$k = \frac{b}{6}\sum_{i=1}^{12} rand(-1,1)$<br>$return\ k$ |
|---|
| Πίνακας 3.5.3 |

### 3.5.4. Odometry Motion Model

Ας δούμε συνοπτικά και αυτό το μοντέλο. Για αυτό το μοντέλο χρειάζεται οι ρόδες να έχουν encoder. Αρχικά δίνουμε την εντολή κίνησης στο ρομπότ και διαβάζοντας τους encoder υπολογίζουμε την νέα θέση $s_t$. Στο μοντέλο αυτό δίνουμε την εντολή κίνησης $u_t$, την θέση $s_{t-1}$ και την θέση $s_t$. Έτσι μπορούμε να υπολογίσουμε την διαδρομή που έκανε το ρομπότ με το μοντέλο αυτό.

Η πληροφορία που λαμβάνουμε από το odometry model είναι ( $\delta_{rot1}, \delta_{rot2}, \delta_{trans}$ )




Εικόνα 3.5.2

Όπου προκύπτει ότι :

$$\delta_{trans} = \sqrt{(x' - x)^2 + (y' - y)^2} \quad (3.5.4.1)$$

$$\delta_{rot1} = atan2(y' - y, x' - x) - \theta \quad (3.5.4.2)$$

$$\delta_{rot2} = \theta' - \theta - \delta_{rot1} \quad (3.5.4.3)$$

## 3.6. Measurement Model

Μία από τις βασικότερες λειτουργίες του αλγόριθμου SLAM είναι να παράγει μοντέλο (generative model) από μετρήσεις που προέρχονται από αισθητήρες. Δηλαδή διαθέτει μαθηματικούς νόμους στατιστικής που προσδιορίζουν την διαδικασία με την οποία οι μετρήσεις παράγονται. Αυτό το μοντέλο αναφέρεται ως measurement model και έχει τον ακόλουθο τύπο:

$$p(z_t \mid s_t, \theta_{nt}, n_t) = g(\theta_{nt}, s_t) + \varepsilon_t \quad (3.5.4.1)$$

Το Measurement Prediction ($z_t$) εξαρτάται από την θέση του ρομπότ $s_t$, την τιμή του σημαντικού στοιχείου (landmark ) $n_t$ και από το feature $\theta_{nt}$ . Ισούται με την μη γραμμική συνάρτηση g που επηρεάζεται από την ύπαρξη του θορύβου. Ο θόρυβος την χρονική στιγμή t προσδιορίζεται από την τυχαία τιμή $\varepsilon_t$ , της οποία υποθέτουμε ότι η Gaussian έχει μέσο όρο μηδέν και τυπική απόκλιση ίση με $R_t$ . Συνήθως η Gaussian του θορύβου υποθέτουμε ότι προκύπτει κατά προσέγγιση, παρόλα αυτά τείνει να λειτουργεί καλά σε ένα ευρύ φάσμα του αισθητήρα.

Στην δική μας εφαρμογή το Measurement Model που χρησιμοποιούμε είναι το ακόλουθο :

| $$z'^{[m]} = h\left( x_t^{[m]}, m_{i,t-1}^{[k]} \right) = \begin{bmatrix} \sqrt{(x_l - x)^2 + (y_l - y)^2 + (z_l - z)^2} \\ atan2\big((y_l - y), (x_l - x)\big) - \theta_z \\ atan2\big((z_l - z), (x_l - x)\big) - \theta_z \end{bmatrix} = \begin{bmatrix} range \\ bearing\ 1 \\ bearing\ 2 \end{bmatrix}$$ |
|---|
| Πίνακας 3.6.1 |

Όπως φαίνεται από την παραπάνω εξίσωση εισάγουμε την θέση του ρομπότ ($s_t^{[m]}$) και το σημαντικό στοιχείο (landmark). Το measurement prediction μας αποτελείται από έναν 3x1 πίνακα, όπως φαίνεται στον Πίνακας 3.6.1 . Αυτό συμβαίνει διότι αυτά τα στοιχεία είναι αρκετά για να μπορούμε να χαρακτηρίσουμε ένα σημείο στον χώρο με μοναδικό τρόπο.

# Κεφάλαιο 4. - Αλγόριθμοι

Σε αυτό το κεφάλαιο θα αναλύσουμε τους αλγορίθμους τους οποίους θα χρησιμοποιήσουμε για την υλοποίηση της εφαρμογής μας.

## 4.1. SLAM

Στην ακόλουθη ενότητα θα ασχοληθούμε με το να εξηγήσουμε το SLAM σε πιο πρακτικό επίπεδο, συνδυάζοντας τα θεωρητικά που αναφέραμε στην εισαγωγή.

Όπως έχει ήδη αναφερθεί, ο σκοπός του SLAM είναι να χτίσει τον χάρτη του περιβάλλοντος και ταυτόχρονα να κρατάει μια εκτίμηση της θέσης του μέσα σε αυτόν τον χάρτη. Επίσης, προαναφέραμε ότι όλα αυτά πρέπει να γίνουν με βάση μόνο τις μέτρησεις $z_t$ και τις εντολές κίνησης $u_t$.

Ακολουθεί μια πιο μαθηματική ανάλυση:

Έστω ότι έχουμε ένα σύνολο από μετρήσεις $z_{1:t}$ και εντολές $u_{1:t}$ μετά από t βήματα στον διακριτό χρόνο. Ο στόχος του SLAM είναι να υπολογίσει την τοποθεσία $s_t$ και τον χάρτη του περιβάλλοντος του m. Το πρόβλημα αυτό μπορεί να χωριστεί σε δύο άλλα διαφορετικά προβλήματα της ρομποτικής:

1. Χαρτογράφηση χώρου με γνωστή τοποθεσία
2. Εντοπισμός θέσης με γνωστό χάρτη

Άμα το γράψουμε σε ένα γενικό probabilistic πρόβλημα:

(1) → $P(m_t|s_t, z_{1:t})$ (2) → $P(s_t|m_t, z_{1:t})$

### 4.1.1. Probabilistic form του SLAM

Από την πλευρά των πιθανοτήτων, έχουμε δύο βασικές μορφές SLAM. Το ένα είναι το online SLAM και το άλλο είναι το full SLAM:

- *On-line SLAM* → $p(s_t, m|z_{1:t}, u_{1:t})$ *( 4.1.1.1 )*
- *Full SLAM* → $p(s_{1:t}, m|z_{1:t}, u_{1:t})$ *( 4.1.1.2 )*

***On-line SLAM***

Το αποκαλούμε "on-line SLAM" γιατί μας ενδιαφέρει να υπολογίσουμε την θέση του ρομπότ μόνο την τωρινή χρονική στιγμή t. Ως αποτέλεσμα, συνήθως αγνοούνται οι προηγούμενες μετρήσεις και εντολές, εφόσον αυτές δεν επηρεάζουν την τωρινή θέση. Έτσι, ο παραπάνω τύπος μπορεί να γραφτεί $p(s_t, m|z_t, u_t)$ σε μερικές περιπτώσεις.

Η δική μας εφαρμογή θα είναι on-line και θα ασχοληθούμε με την απλοποιημένη μορφή του, δηλαδή αγνοώντας παλιότερες μετρήσεις και εντολές.

***Full SLAM***

Σε αυτήν την περίπτωση μας ενδιαφέρουν όλες οι θέσεις που είχε το ρομπότ από την στιγμή που ξεκίνησε.

### 4.1.2. Θεμελιώδη βήματα του SLAM

Το SLAM διαιρείται σε διάφορα βήματα ώστε να λυθεί το πρόβλημα. Τα πιο ευρέως αποδεκτά (άλλα όχι υποχρεωτικά μόνο αυτά) είναι τα εξής βήματα:

1. **Data acquisition:** Ποια αισθητήρια θα χρησιμοποιηθούν ώστε να πάρουμε τα επιθυμητά αποτελέσματα.

2. **Landmark extraction:** Ένας αριθμός από χαρακτηριστικά σημεία (*landmarks*) που μπορούμε να πάρουμε από τα δεδομένα που μας έδωσαν τα αισθητήρια μας. Ο σκοπός είναι να γίνονται εύκολα αναγνωρίσιμα και να είναι όσο μοναδικά γίνεται.

3. **Data association:** Να γίνει αντιστοίχιση των *landmarks* που εντοπίστηκαν με τα *landmarks* που είναι αποθηκευμένα στην βάση δεδομένων.

4. **Pose estimation:** Σύμφωνα με την τοπική αλλαγή θέσης των εντοπισμένων *landmarks* και των αποθηκευμένων *landmarks* να υπολογιστεί η τωρινή θέση του ρομπότ.

5. **Map adjustment:** Ο χάρτης είναι το σύνολο των *landmarks*. Ο σκοπός εδώ είναι να γίνει η ανανέωση με τις νέες υπολογισμένες θέσεις του ρομπότ και όλων των *landmarks*.

Ο κάθε αλγόριθμος SLAM προσπαθεί να λύσει κάθε ένα από αυτά τα βήματα με διαφορετικό τρόπο. Αυτά τα πέντε βήματα επαναλαμβάνονται συνέχεια με απώτερο σκοπό να χτιστεί ο χάρτης και να γνωρίσουμε την θέση του ρομπότ μέσα σε αυτόν.

Στην δικιά μας περίπτωση, θα βρίσκουμε μοναδικά σημεία μέσα στην RGB εικόνα που θα παίρνουμε, και αυτά θα τα χρησιμοποιούμε ως *landmarks*. Ο αλγόριθμος που θα χρησιμοποιήσουμε για τον εντοπισμό των σημείων είναι ο AKAZE [4.1-2], ενώ για το data association θα χρησιμοποιηθεί το BG matcher σε συνδυασμό με το homography των εικόνων. Ο AKAZE θα εξηγηθεί παρακάτω.

Μερικοί δημοφιλείς αλγόριθμοι SLAM είναι:

- EKF SLAM
- Unscented Kalman Filter (UKF) SLAM
- Sparse EIF SLAM
- FastSLAM 1 και FastSLAM 2
- Graph-Based SLAM

Οι δύο πιο διαδεδομένες τεχνικές είναι το EKF SLAM και το FastSLAM (1 & 2).

### 4.1.3. Maps in SLAM

Όταν μιλάμε για maps στο SLAM, μπορούμε να τα διαχωρίσουμε σε δύο βασικά είδη. Grid-based maps και featured-based maps.

***<u>Grid-based maps</u>***

Όταν μιλάμε για Grid-based maps σημαίνει ότι έχουμε χωρίσει τον χώρο σε πολλά κελιά (grid) και το κάθε κελί παίρνει μια τιμή από 0.0 έως 1.0 . Το ποσοστό που έχει το κελί καθορίζει εάν είναι occupied ή free το αντίστοιχο σημείο στον φυσικό χώρο. Όσο πιο κοντά στο 0.0, τόσο πιο πιθανό θεωρείται ότι υπάρχει κάτι εκεί (ένας τοίχος, μια πόρτα, μια καρέκλα κτλ.). Αντίστοιχα, όσο πιο κοντά στο 1.0, τόσο πιο σίγουρο είναι ότι δεν υπάρχει κάτι εκεί και μπορεί να μετακινηθεί μέσα σε εκείνο τον χώρο το ρομπότ. Είναι σημαντικό να σημειωθεί ότι όταν πάρουμε μια μέτρηση και βρούμε ότι ένα συγκεκριμένο κελί (με συντεταγμένες x,y) είναι occupied, πρέπει να υπολογίσουμε όλα τα υπόλοιπα κελιά που

μεσολαβούν ανάμεσα στο ρομπότ και σε εκείνο το σημείο. Αυτό γίνεται διότι πρέπει να δηλώσουμε σε εκείνα τα κελιά ότι είναι free και να κατεβάσουμε τον αριθμό τους.

Όταν δουλεύουμε στο 2-D απλά φτιάχνουμε έναν δισδιάστατο πίνακα που αναπαριστά τον χώρο στον οποίο κυκλοφορεί το ρομπότ και σύμφωνα με τις μετρήσεις που παίρνει, αυξάνουμε ή μειώνουμε τους αριθμούς των συγκεκριμένων κελιών.

Στην περίπτωση του 3-D, χρησιμοποιούμε μια επέκταση του grid map που λέγεται voxel map. Θα μπορούσαμε να κάνουμε χρήση ενός τρισδιάστατου πίνακα για την αναπαράσταση του voxel map αλλά αυτό είναι πολύ βαριά διαδικασία. Ειδικά για εμάς που θα κάνουμε χρήση του FastSLAM που θα κρατάει ταυτόχρονα, το λιγότερο, 100 χάρτες. Γι' αυτό χρησιμοποιείται το OctoMap [4.1-1] σε τέτοιες περιπτώσεις.

Για αυτήν την τεχνική τα συνηθισμένα αισθητήρια που χρησιμοποιούνται είναι laser scanner, sonar, ultrasonic.

***<u>Feature-based maps</u>***

Υπάρχει όμως και μια εναλλακτική στο Grid map, το feature-based map. Εδώ αντί να χτίζουμε ολόκληρο τον χώρο γύρω μας, βρίσκουμε σημεία στο χώρο που είναι μοναδικά αναγνωρίσιμα.

Για αυτήν την τεχνική τα συνηθισμένα αισθητήρια που χρησιμοποιούνται είναι RGB cameras, stereo cameras, 3D laser scanner και RGB-D cameras.

***<u>Grid-based vs Feature-based</u>***

Συγκρίνοντας τις δύο τεχνικές μπορούμε να βγάλουμε λίγα συμπεράσματα. Αρχικά, το grid-based θέλει μεγαλύτερη επεξεργαστική ισχύ γιατί πρέπει να υπολογίσει τις συντεταγμένες ολόκληρου του χώρου μπροστά του και μετά να τις αποθηκεύσει. Αντιθέτως, το feature-based βρίσκει το πολύ 100-150 σημεία κάθε φορά. Παρ' όλα αυτά το grid-based έχει ένα σημαντικό πλεονέκτημα, επιτρέπει στο ρομπότ να σχεδιάζει αυτόματα διαδρομές μέσα στον χάρτη του. Αυτό γίνεται γιατί το ρομπότ ξέρει στο grid που υπάρχει τοίχος και που δεν υπάρχει ενώ το feature-based έχει μόνο σημεία χωρίς orientation ή κάτι που να υποδεικνύει από ποια πλευρά του feature είναι ελεύθερο να περάσει το ρομπότ.

## 4.2. FastSLAM

### 4.2.1. Θεωρητικό υπόβαθρο

Εμείς επιλέξαμε να κάνουμε χρήση του FastSLAM [4.2-2]. Συγκεκριμένα χρησιμοποιούμε το FastSLAM 1. Η διαφορά ανάμεσα στα δύο είναι ότι το FastSLAM 1 είναι για feature-based maps ενώ το FastSLAM 2 είναι για grid-based maps. Από εδώ και πέρα θα αναφερόμαστε στον αλγόριθμο απλά ως FastSLAM.

Για το FastSLAM, κάθε *εντολή* (control) ή *μέτρηση* (observation) που καταγράφει το ρομπότ, constrains ένα μικρό αριθμό μεταβλητών κατάστασεις (state variables). Οι *εντολές* πιθανοθεωρητικά (probabilistically) constrains την τωρινή θέση του ρομπότ σε σχέση με την προηγούμενη θέση του ρομπότ. Οι *μετρήσεις* πιθανοθεωρητικά constrain την τοποθεσία των *landmarks* σε σχέση με την θέση του ρομπότ. Μόνο μετά από πολλά από αυτά τα πιθανοθεωρητικά constrain μπορούμε να πούμε ότι ο χάρτης είναι πλήρως συσχετισμένος (correlated) μεταξή του. Εδώ βλέπουμε κάτι που δεν αξιοποιεί καθόλου το EKF SLAM, διότι δεν κάνει καμιά υπόθεση για τις μεταβλητές κατάστασεις. Δηλαδή, ενώ το EKF SLAM δημιουργεί συσχέτιση (correlation) ανάμεσα σε όλες τις μεταβλητές κατάστασεις στο χώρο κατάστασης (state space), το FastSLAM θεωρεί ότι τα *landmark* είναι ανεξάρτητα το ένα από το άλλο και τα συσχετίζει μόνο με την διαδρομή του ρομπότ. Με άλλα λόγια, το EKF κατασκευάζει τον χάρτη σε σχέση με τις **θέσεις** του ρομπότ ενώ το FastSLAM κατασκευάζει τον χάρτη σε σχέση με την **διαδρομή** του ρομπότ. Αυτή είναι η πιο σημαντική διαφορά ανάμεσα στης δύο αυτές δημοφιλείς τεχνικές.

Ας δούμε μερικές ακόμη διαφορές ανάμεσα στο FastSLAM και το EKF SLAM. Σε αντίθεση με το EKF SLAM που κάνει χρήση του Extended Kalman Filter για το *localization* και το *mapping*, το FastSLAM κάνει χρήση του particle filter (πιο συγκεκριμένα το Rao-Blackwellized Particle Filter) για το *localization*, σε συνδυασμό με πολλά μικρά EKF για το κάθε *landmark*. Αυτό είναι σημαντικό διότι το EKF αντιπροσωπεύει την αβεβαιότητα με Gaussian καμπύλες ενώ το particle filter (που χρησιμοποιείται το FastSLAM) αντιπροσωπεύει την αβεβαιότητα με particles. Όσο πιο πυκνός είναι ο αριθμός των particles, τόσο υψηλότερη είναι η πιθανότητα να αντιπροσωπεύουν την πραγματική θέση του ρομπότ. Αντίστοιχα, όσο πιο αραιωμένα είναι τα particles, τόσο χαμηλότερη είναι η πιθανότητα. Αυτό επιτρέπει στο FastSLAM να μπορεί να αναπαριστά σύνθετες multi-modal κατανομές, ακόμη και αν είναι μη-γραμμικές (που συνήθως είναι).

Επίσης, ένα άλλο πλεονέκτημα είναι η περιπλοκότητα και η επεξεργαστική ισχύς που χρειάζεται. Στην περίπτωση του EKF SLAM όλες οι πληροφορίες (θέσει και landmarks) βρίσκονται μέσα σε έναν μεγάλο πίνακα. Στην δικιά μας περίπτωση έχουμε 6 μεταβλητές κατάστασεις για το ρομπότ και το κάθε landmark έχει 3 συντεταγμένες. Πιο αναλυτικά θα είχαμε:

- Τον πίνακα «μ» (πίνακας μέσης τιμής) με μέγεθος $\langle (6 + 3 * \text{αριθμός landmarks}) x\ 1 \rangle$
- και τον πίνακα «Σ» (πίνακας διακύμανσης) με μέγεθος $\langle (6 + 3 * \text{αριθμός landmarks})\ x\ (6 + 3 * \text{αριθμός landmarks}) \rangle$.

Σε κάθε επανάληψη χρειάζεται να ανανεώνουμε ολόκληρο τον πίνακα, ακόμη και αν ένα μικρό μέρος του επηρεάζεται. Αυτή είναι μια διαδικασία που κοστίζει πολύ. Σε αντίθεση με αυτό, το FastSLAM έχει για κάθε *landmark* ένα «μ» ⟨6x1⟩ και ένα «Σ» ⟨6x6⟩. Αυτό ισχύει διότι, όπως είπαμε λίγο πιο πάνω, το FastSLAM εκμεταλλεύεται το γεγονός ότι τα *landmarks* μπορούμε να τα θεωρήσουμε ανεξάρτητα μεταξύ τους. Στην περίπτωση που δουλεύουμε στον 2D χώρο, το FastSLAM έχει αποδειχτεί να είναι πιο αποδοτικό από το EKF SLAM, για μεγάλους χάρτες. Σε χώρο κατάστασης με πάρα πολλές μεταβλητές κατάστασης το FastSLAM γίνεται πολύ αναποτελεσματικό. Αυτό συμβαίνει διότι το κάθε *particle* αντιπροσωπεύει μια πιθανή κατάσταση του οχήματος και ένα πιθανό χάρτη, άρα κάθε *particle* έχει και έναν διαφορετικό συνδυασμό μεταβλητών κατάστασης. Έτσι, το σύνολο των μεταβλητών κατάστασης πολλαπλασιάζεται με τον αριθμό των *particles*. Το πόσο αποδοτικό είναι το FastSLAM για 3D χώρο είναι ακόμη υπό έρευνα. Ο στόχος της παρούσας εργασεία είναι να σχεδιάσει έναν τέτοιων αλγόριθμο.

Ένα τελευταίο πλεονέκτημα που έχει το FastSLAM και το οποίο θα εκμεταλλευτούμε, είναι ότι βελτιώνεται πολύ όταν με μια μέτρηση μπορεί να πάρει πολλά *landmarks*. Όπως μπορούμε να δούμε στο [4.2-2], πολλαπλοί εντοπισμοί *landmarks* σε με μια μέτρηση βελτιώνουν σημαντικά την απόδοση του αλγορίθμου.

Ας επιστρέψουμε σε κάτι σημαντικό που γράψαμε πιο πάνω, για το πώς το FastSLAM θεωρεί ότι όλα τα *landmarks* είναι ανεξάρτητα το ένα από το άλλο. Αυτό είναι πολύ σημαντικό διότι έτσι αν μας δοθεί η διαδρομή του ρομπότ, μπορούμε να υπολογίσουμε την θέση κάθε *landmark*. Άρα μπορούμε να πάρουμε το *( 4.21.4.2.1.1)* και το factored it σε έναν απλό πολλαπλασιασμό:

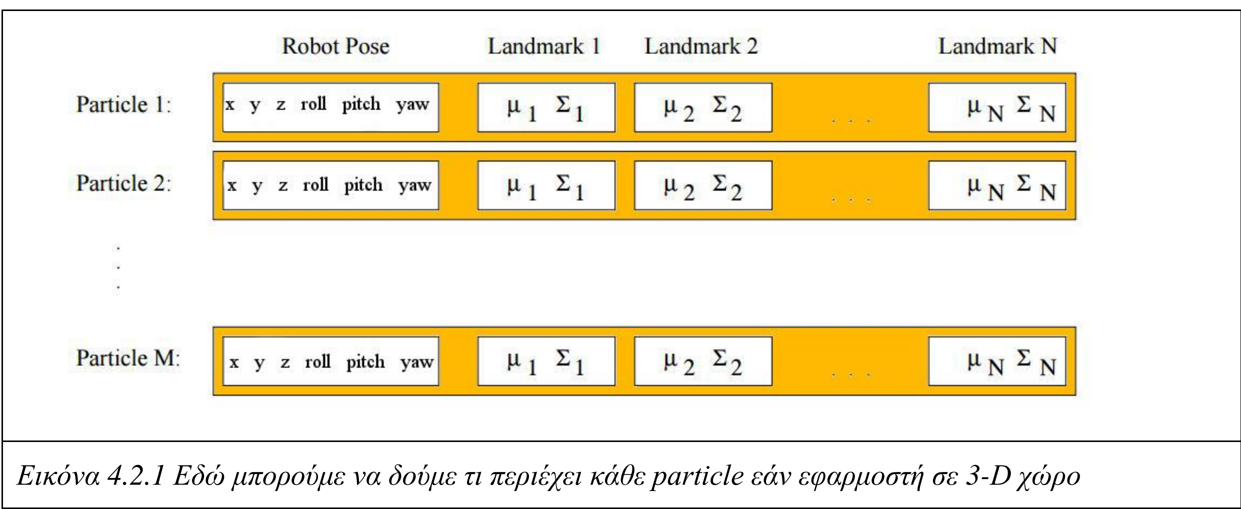
$$p(s_t, m|z_t, u_t, c_t) = p(s_t|z_t, u_t, c_t) \prod_{n=1}^{N} p(m_n|s_t, z_t, u_t, c_t) \qquad (4.2.1.2)$$

Αυτό το factorization, που σχεδιάστηκε αρχικά από τους Murphy και Russell [4.2-1], δηλώνει ότι το posterior μπορεί να βγει από τον πολλαπλασιασμό της διαδρομής του ρομπότ $p(s_t|z_t, u_t, c_t)$ και των N *landmarks* posterior $\prod_{n=1}^{N} p(m_n|s_t, z_t, u_t, c_t)$.

### 4.2.2. Ο αλγόριθμος

Αυτό που θέλουμε τώρα είναι να υπολογίσουμε τα δύο διαφορετικά μέρη του (4.2.2). Το posterior της διαδρομής του ρομπότ, $p(s_t|\mathrm{z_t}, \mathrm{u_t}, \mathrm{c_t})$, υπολογίζεται με την χρήση particles filter ενώ το posteriors των N *landmarks*, $p(m_n|s_t, z_t, u_t, c_t)$, υπολογίζονται με την χρήση του EKF. Κάθε EKF ασχολείται με μόνο ένα *landmark*, άρα έχουμε ένα χαμηλών διαστάσεων πίνακα με σταθερό μέγεθος. Όλα τα EKFs των landmarks είναι σε σχέση με την διαδρομή του ρομπότ. Κάθε *particle* μέσα στο Particle Filter να έχει το δικό του σύνολο από EKFs για τα *landmarks*. Άρα ο συνολικός αριθμός από EKFs είναι N*M. Μπορούμε στην Εικόνα 4.2.1 να δούμε γραφικά τα particles στην 3D περίπτωση.

Το κάθε *particle* έχει ένα set συντεταγμένων (για εμάς είναι x, y, z, roll, pitch και yaw) και ένα σύνολο από *landmarks* που αποτελούν τον χάρτη. Στην ουσία το κάθε *particle* θεωρεί ότι η δικές του συντεταγμένες του ρομπότ είναι οι σωστές και ότι ο χάρτης που έχει σχεδιάσει ανταποκρίνεται στην πραγματικότητα. Σύμφωνα με την προηγούμενη υπόθεση, υπολογίζει ποια μέτρηση θα πρέπει να επιστρέψει ο αισθητήρας. Αυτόν τον υπολογισμό τον ονομάζουμε *προβλεπόμενη μέτρηση*. Αφού πάρουμε μια μέτρηση, την συγκρίνουμε με την *προβλεπόμενη μέτρηση* που υπολογίζει κάθε *particle*, και αναλόγως του δίνουμε ένα importance weight που θα καθορίσει πόσο αξιόπιστο θεωρούμε ότι είναι το συγκεκριμένο particle. Μπορεί να πει κανείς ότι τα particles που επιβιώνουν υπάγονται κάτω από τον “νόμο” της «επιβίωσης του ισχυρότερου».




*Εικόνα 4.2.1 Εδώ μπορούμε να δούμε τι περιέχει κάθε particle εάν εφαρμοστή σε 3-D χώρο*

Μια πιο μαθηματική αναπαράσταση του κάθε *particle* είναι η εξής:

$$S_t^{[m]} = \langle s_t^{[m]}, \mu_{1,t}^{[m]}, \Sigma_{1,t}^{[m]}, \dots, \mu_{n,t}^{[m]}, \Sigma_{n,t}^{[m]} \rangle \qquad (4.2.2.1)$$

Εδώ το [m] είναι η αρίθμηση για κάθε *particle*. $S_t^{[1]}$ είναι το πρώτο *particle*, $S_t^{[2]}$ είναι το δεύτερο *particle* και $S_t^{[m]}$ είναι το m-th *particle*. Τα $\mu_{1,t}^{[m]}, \Sigma_{1,t}^{[m]}$ είναι η μέση τιμή και η διακύμανση της Gaussian που αντιπροσωπεύει το $1^o$ landmark του m-th *particle*.

Ας δούμε περιληπτικά την διαδικασία του FastSLAM αλγόριθμου. Αρχικά, μια καινούρια θέση ($s_t^{[m]}$) υπολογίζεται για το κάθε particle, χρησιμοποιώντας την εντολή κίνησης $u_t$ και την παλιά θέση $s_{t-1}^{[m]}$. Στην συνέχεια, τα EKFs των landmarks ενημερώνονται με την καινούρια μέτρηση που έγινε του landmark. Εφόσον όλα τα particles δεν πάνε στο ίδιο σημείο (σύμφωνα με το motion model) θα τους δοθεί ένα *importance weight* που δείχνει αυτή την διαφορά θέσης που έχουν. Τέλος, με την διαδικασία του *importance resampling*, χρησιμοποιώντας το *importance weight*, μια καινούρια γενιά από *particles* δημιουργείται. Το *importance resampling* είναι σημαντικό διότι επιτρέπει στα καινούρια *particles* ($S_t$) να αντιπροσωπεύουν το πραγματικό posterior. Αυτά τα 4 βασικά βήματα του FastSLAM είναι τα εξής:

1. Sample new particle pose
2. Ενημέρωση των landmarks
3. Ορισμός importance weight στο particle
4. Importance Resampling

Ας τα δούμε πιο αναλυτικά τώρα:

*Sample new particle pose*

Το πρώτο βήμα είναι να κάνουμε χρήση του Motion model. Όπως έχουμε ήδη αναφέρει, το motion model περιγράφει τον τρόπο κίνησης του ρομπότ. Έτσι, στην χρονική στιγμή t θα πάρουμε το pose του κάθε particle $S_{t-1}^{[m]}$ και την εντολή της επιθυμητής κίνησης $u_t$ ώστε να παράγουμε (generate) την καινούρια θέση $S_t^{[m]}$.

| $s_t^{[m]} \sim p(s_t \vert s_{t-1}^{[m]}, u_t)$ *(4.2.2.2)* |
|---|
| Η πράξει που γίνεται για την ολοκλήρωση αυτού του βήματος. |

*Ενημέρωση των landmarks*

Τα landmarks ενημερώνονται με την χρήση 3x3 EKF, όπως αναφέραμε στην εισαγωγή του κεφαλαίου 4. Αυτό το βήμα χωρίζεται σε δύο διαφορετικές περιπτώσεις. Σε περίπτωση ενημέρωσης παλιού landmark και στην περίπτωση προσθήκης νέου landmark.

Αλλά πρώτα για να γίνει κατανοητό αυτό το βήμα πρέπει να προσδιορίσουμε τι θεωρούμε measurement, measurement observation και measurement prediction. Ως measurement θεωρούμε τα εξής στοιχεία:

$$\begin{bmatrix} range \\ pitch \\ yaw \end{bmatrix} \qquad (4.2.2.3)$$

Αυτά είναι αρκετά να προσδιορίσουν ένα μοναδικό σημείο στο local frame του ρομπότ.

Το *measurement observation* είναι ο υπολογισμός των τριών στοιχείων του (4.2.5) βασισμένα μόνο στην μέτρηση που έκανε ο αισθητήρας μας, η RGB-D camera. Στην δική μας περίπτωση, το Kinect μέσω του driver libfreenect μαζί με το PCL, μας δίνει το X,Y και Z του κάθε σημείου που βλέπει το Kinect σε local frame. Έτσι, με τον παρακάτω τύπο υπολογίσουμε το $z_t$ (*measurement observation*):

$$z_t = \begin{bmatrix} \sqrt{x^2+y^2+z^2} \\ arcsin(\frac{y}{\sqrt{x^2+y^2}}) \\ arcsin(\frac{z}{\sqrt{x^2+z^2}}) \end{bmatrix} \qquad (4.2.2.4)$$

Το *measurement prediction* είναι ο υπολογισμός των τριών στοιχείων του (4.2.5) βασισμένου στο pose που μας έδωσε το motion model (δηλαδή $s_t^{[m]}$) και την τοποθεσία που είχαν τα *landmarks* (για το συγκεκριμένο particle) στην προηγούμενη θέση (δηλαδή $\mu_{j,t-1}^{[m]}$).

$$z_t'^{[m]} = h(s_t^{[m]}, \mu_{j,t-1}^{[m]}) \qquad (4.2.2.5)$$

*Ενημέρωση παλαιού landmark:*

Βλέποντας την διαφορά του measurement observation και του measurement prediction, μπορούμε να υπολογίσουμε πόσο σφάλμα έχει το συγκεκριμένο particle. Τα βήματα αυτού του βήματος είναι ως εξής:

$$z'^{[m]} = h(s_t^{[m]}, \mu_{j,t-1}^{[m]}) \qquad (4.2.2.6)$$

$$H = h'(x_t^{[m]}, \mu_{j,t-1}^{[m]}) \qquad (4.2.2.7)$$

$$Q = H\Sigma_{j,t-1}^{[m]}H^T + Q_t \qquad (4.2.2.8)$$

$$K = \Sigma_{j,t-1}^{[m]}H^T Q^{-1} \qquad (4.2.2.9)$$

$$\mu_{j,t}^{[m]} = \mu_{j,t-1}^{[m]} + K(z_t - z'^{[m]}) \qquad (4.2.2.10)$$

$$\Sigma_{j,t}^{[m]} = (I - KH)\Sigma_{j,t-1}^{[m]} \qquad (4.2.2.11)$$

*Πίνακας 4.2.1 Ο αλγόριθμος για την ενημέρωση των landmarks*

*Προσθήκη νέου landmark:*

Στην περίπτωση που εντοπίζουμε για πρώτη φορά ένα landmark οι πράξεις, ώστε να το προσθέσουμε στον χάρτη μας, είναι απλές. Ο πίνακας μέσης τιμής παίρνει όποια x, y και z μας δίνει το Kinect και ο πίνακας διακύμανσης το Jacobian του h()

$$\mu_{j,t}^{[m]} = \mu_{kinect} \qquad (4.2.2.12)$$

$$H = h'(\mu_{j,t}^{[m]}, s_t^{[m]}) \qquad (4.2.2.13)$$

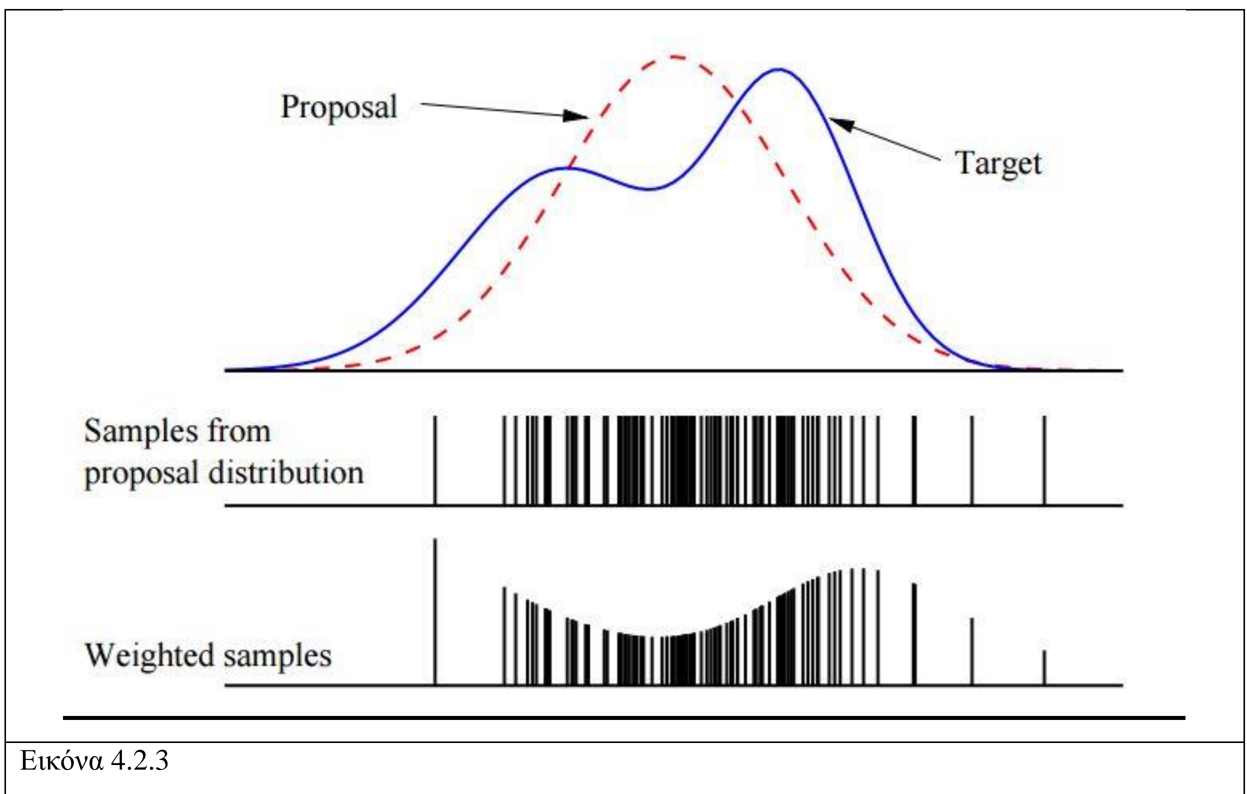
$$\Sigma_{j,t}^{[m]} = H^{-1}Q_t(H^{-1})^T \qquad (4.2.2.14)$$

*Πίνακας 4.2.2 Ο αλγόριθμος για την προσθήκη νέου landmark*

### *Ορισμός importance weight στο particle*

Σε γενικές γραμμές, το importance sampling είναι ένας αλγόριθμος για την δημιουργία δειγμάτων από μια συνάρτηση για την οποία δεν υπάρχει καμιά συγκεκριμένη διαδικασία δειγματοληψίας. Σε κάθε δείγμα που δημιουργήθηκε από την *προτεινόμενη κατανομή* (proposal distribution) του δίνεται ένα weight που ισούται με τον λόγο της *προτεινόμενης κατανομής* προς τη *στοχευμένη κατανομή* (target distribution), στο σημείο του δείγματος. Μια καινούρια γενιά από δείγματα χωρίς *weight* δημιουργούνται από τα *weighted* δείγματα, με κατανομή των βαρών των παλιών δειγμάτων.

Η *προτεινόμενη κατανομή* των δειγμάτων γίνεται σύμφωνα με $p(s_t|z_{t-1}, u_t, c_{t-1})$. Αυτό διαφέρει από το *προτεινόμενο* posterior που είναι $p(s_t|z_t, u_t, c_t)$. Αυτή η διαφορά διορθώνεται μέσω της μεθόδου “importance sampling”. Στην Εικόνα 4.2.2 μπορούμε να δούμε ένα τέτοιο παράδειγμα:




Εικόνα 4.2.3

Σε περιοχές που η *στοχευμένη κατανομή* έχει μεγαλύτερη τιμή από την *προτεινόμενη κατανομή* τα δείγματα παίρνουν μεγαλύτερο weight. Ως αποτέλεσμα, δείγματα σε αυτές τις περιοχές θα επιλέγονται πιο πολύ (όταν γίνεται το resampling), άρα και θα επιβιώσουν περισσότερο ως *particles*. Σε περιοχές που η *στοχευμένη κατανομή* είναι μικρότερη από την

*προτεινόμενη κατανομή* τα δείγματα θα παίρνουν μικρότερο *weight*. Αυτή η διαδικασία θα παράγει δείγματα με κατανομή ομοιόμορφη με αυτή της *στοχευμένης κατανομής*.

Τώρα ας δούμε συγκεκριμένα το FastSLAM. Το importance weight ($w_t^{[m]}$) του κάθε particle είναι ο λόγος του SLAM posterior ως προς την προτεινόμενη κατανομή (proposal distribution) που περιεγράφηκε προηγουμένως:

$$w_t^{[m]} = \frac{target\ distribution}{proposal\ distribution} = \frac{p(s_t^{[m]}|z_t,\ u_t,\ c_t)}{p(s_t^{[m]}|z_{t-1},u_t,c_{t-1})} \qquad (4.2.2.15)$$

Ο αριθμητής μπορεί να επεκταθεί χρησιμοποιώντας τον Bayes Rule.

$$w_t^{[m]} \xRightarrow{Bayes} \frac{\eta * p(z_t|s_t^{[m]},z_{t-1},u_t,c_t)\ p(s_t^{[m]}|z_{t-1},u_t,c_t)}{p(s_t^{[m]}|z_{t-1},u_t,c_{t-1})} \qquad (4.2.2.16)$$

Ο όρος $\eta$ που κανονικοποιεί το αποτέλεσμα του Bayes Rule, μπορεί να αγνοηθεί εφόσον θα κανονικοποιούσε το αποτέλεσμα μας πριν την διαδικασία της δειγματοληψίας (sampling).

$$w_t^{[m]} = \frac{p(z_t|s_t^{[m]},z_{t-1},u_t,c_t)\ p(s_t^{[m]}|z_{t-1},u_t,c_t)}{p(s_t^{[m]}|z_{t-1},u_t,c_{t-1})} \qquad (4.2.2.17)$$

Ο δεύτερος όρος του αριθμητή δεν είναι δεσμευμένος με το τελευταία $z_t$. Άρα μπορούμε να αφαιρέσουμε από εκεί το τελευταίο data association $c_t$.

$$w_t^{[m]} = \frac{p(z_t|s_t^{[m]},z_{t-1},u_t,c_t)\ p(s_t^{[m]}|z_{t-1},u_t,c_{t-1})}{p(s_t^{[m]}|z_{t-1},u_t,c_{t-1})} \qquad (4.2.2.18)$$

Τώρα το δεύτερο μέρος του αριθμητή και ο παρονομαστής απαλείφονται και έχουμε

$$w_t^{[m]} = p(z_t|s_t^{[m]}, z_{t-1}, u_t, c_t) \qquad (4.2.2.19)$$

Εφόσον η μέτρηση $z_t$ δεν επηρεάζεται από την εντολή κίνησης $u^t$, το αφαιρούμε και αυτό. Μας απομένει:

$$w_t^{[m]} = p(z_t|s_t^{[m]}, z_{t-1}, c_t) \qquad (4.2.2.20)$$

Όπως έχουμε αναφέρει ήδη, ο υπολογισμός των landmark γίνεται με EKF, άρα το observation likelihood υπολογίζεται σε closed form. Συνήθως αυτή η πιθανότητα ορίζεται με τον όρο innovation, δηλαδή την διαφορά ανάμεσα στο observation $z_t$ και την εκτίμηση $z'_t$. Η ακολουθία του innovation μέσα στο EKF είναι γκαουσιανά (Gaussianly) κατανεμημένη με μηδενική μέση τιμή και διακύμανση Q (από την εξίσωση 4.8). Η πιθανότητα του observation

$z_t$ είναι ίση με την πιθανότητα του innovation $z_t - z'_t$ που παράγεται από την γκαουσιανή, που γράφεται ως εξής:

$$w_t^{[m]} = \frac{1}{\sqrt{|(2\pi)^3 Q|}} exp\{-\frac{1}{2}(z_t - z'_t)^T Q^{-1}(z_t - z'_t)\} \qquad (4.2.2.21)$$

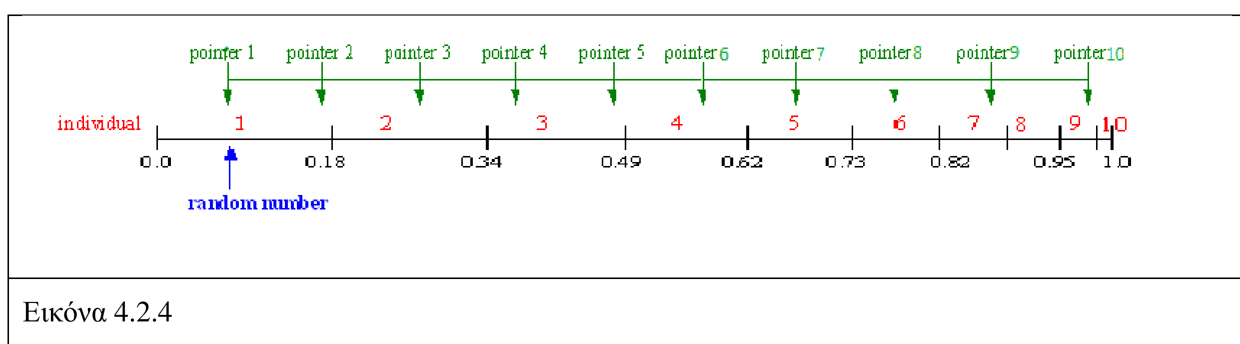
### *Importance Resampling*

Εφόσον σε κάθε *sampled particle* έχει οριστεί ένα *weight*, μια καινούρια γενιά από *particles* πρέπει να φτιαχτεί βασισμένη σε αυτά τα *weight*. Θα αναπαράγουμε τα *particles* σύμφωνα με το *weight* που έχουν, δηλαδή όσο πιο μεγάλο *weight* έχει ένα *particle*, τόσες περισσότερες φορές θα εμφανιστεί στην καινούρια γενιά. Αντίστοιχα, όσο μικρότερο *weight*, τόσο λιγότερα θα υπάρχουν στην καινούρια γενιά. Είναι σημαντικό να θυμίσουμε ότι ο αριθμός τον *particles* μένει ίδιος για όλες τις γενιές.

Δηλαδή

$$\text{size } (S^{t-1}) = \text{size } (S^t)$$

Έχοντας εξηγήσει τον λόγο που κάνουμε *resampling*, ας αναλύσουμε και τον αλγόριθμο *resampling* που επιλέξαμε. Ενώ υπάρχουν αρκετές επιλογές, εμείς κάνουμε χρήση του Stochastic Universal Sampling (SUS). Ας φανταστούμε ότι βάζουμε όλα τα *weight* σε μια σειρά, σύμφωνα με το μέγεθος τους (όπως φαίνεται στην εικόνα 4.2.3). Δηλαδή όσο πιο μεγάλο weight, τόσο περισσότερο χώρο πιάνει. Μετά βρίσκουμε έναν τυχαίο αριθμό ανάμεσα στο 0 και στο $M^{-1}$ (όπου M = πλήθος των particles) και ορίζουμε έναν δείκτη που δείχνει στην γραμμή από weights να έχει αυτήν την τιμή. Βλέπουμε μέσα σε ποιο *weight* είναι ο δείκτης και αναπαράγουμε το *particles* στο οποίο ανήκει. Έπειτα αυξάνουμε τον δείκτη μας κατά $M^{-1}$ και παίρνουμε το *particle* στο οποίο ανήκει αυτό. Συνεχίζουμε έτσι μέχρι το τέλος. Η εικόνα 4.2.3 δείχνει όλη τη διαδικασία.




Εικόνα 4.2.4

Το πλεονέκτημα που προσφέρει αυτή η τεχνική σε σχέση με άλλες, που απλά χρησιμοποιούν τυχαίους αριθμούς, είναι ότι είναι δίκαιο προς όλα. Δηλαδή εάν τύχει όλα τα *particles* να

έχουν το ίδιο ακριβώς *weight*, θα επιλέξει από μια φορά το κάθε *particle* (σε αντίθεση με το να επιλέγεις απλά τυχαίους αριθμούς).

Ας δούμε τον αλγόριθμο τώρα:

| $SUS_{function(X_t, W_t)}$:<br>    $X'_t = 0$<br>    $r = random(0, M^{-1})$<br>    $c = w_t^{[1]}$<br>    $i = 1$<br>    $for\ m = 1\ to\ M\ do$<br>        $U = r + (m-1)M^{-1}$<br>        $while\ U > c$<br>            $i = i + 1$<br>            $c = c + w_t^{[i]}$<br>        $end_while$<br>        $add\ x_t^i\ to\ X'_t$<br>    $end_for$<br>    $return(X'_t)$ |
|---|
| Πίνακας 4.2.3 |

### *Αλγόριθμος*

Τώρα που είδαμε τα επιμέρους κομμάτια, ας δούμε ολόκληρο τον αλγόριθμο:

| FastSLAM1_algoritm($z_t, c_t, u_t, S_{t-1}$)<br><br>$for\ m = 1\ to\ M\ do$:<br>    $Let\ \left\langle s_{t-1}^{[m]}, \left\langle \mu_{1,t-1}^{[m]}, \Sigma_{1,t-1}^{[m]}, id_{1,t-1} \right\rangle, \left\langle \mu_{2,t-1}^{[m]}, \Sigma_{2,t-1}^{[m]}, id_{2,t-1} \right\rangle, \dots \right\rangle\ be\ particle\ m\ in\ S_{t-1}$<br>    $s_t^{[m]} \sim p\left(s_t \middle\| s_{t-1}^{[m]}, u_t\right)$<br>    $j = c_t$<br>    if feature j never seen before:<br>        $\mu_{j,t}^{[m]} = \mu_{minect}$<br>        $H = h'\left(\mu_{j,t}^{[m]}, s_t^{[m]}\right)$<br>        $\Sigma_{j,t}^{[m]} = H^{-1} Q_t (H^{-1})^T$<br>        $w_t^{[m]} = default\ importance\ weight$<br>    else<br>        $z'^{[m]} = h\left(s_t^{[mm]}, \mu_{j,t-1}^{[m]}\right)$<br>        $H = h'\left(s_t^{[m]}, \mu_{j,t-1}^{[m]}\right)$<br>        $Q = H \Sigma_{j,t-1}^{[m]} H^T + Q_t$<br><br>        $K = \Sigma_{j,t-1}^{[m]} H^T Q^{-1}$<br>        $\mu_{j,t}^{[m]} = \mu_{j,t-1}^{[m]} + K\left(z_t - z'^{[m]}\right)$ |
|---|

| |
|---|
| $\Sigma_{j,t}^{[m]} = (I - KH)\Sigma_{j,t-1}^{[m]}$<br>$w_t^{[m]} = \frac{1}{\sqrt{\lvert(2\pi)^{3/2}Q\rvert}} exp\{-\frac{1}{2}(z_t - z'_t)^T Q^{-1}(z_t - z'_t)\}$<br>end_if<br>for all unobserved features j' do:<br>$\left\langle \mu_{j',t}^{[m]}, \Sigma_{j',t}^{[m]} \right\rangle = \left\langle \mu_{j',t-1}^{[m]}, \Sigma_{j',t-1}^{[m]} \right\rangle$<br>end_for<br>end_for<br>$S_t = resample(\left\langle s_t^{[m]}, \left\langle \mu_{1,t}^{[m]}, \Sigma_{1,t}^{[m]}, id_{1,t} \right\rangle, \left\langle \mu_{2,t}^{[m]}, \Sigma_{2,t}^{[m]}, id_{2,t} \right\rangle, \ldots, w_t^{[m]} \right\rangle)$<br>return $(S_t)$ |
| Πίνακας 4.2.4 |

## 4.3. AKAZE

Ο αλγόριθμος AKAZE είναι ένας αλγόριθμος για την ανίχνευση και την περιγραφή 2D feature σε μη γραμμικά scale space . Το KAZE [4.3-1], που στα ιαπωνικά σημαίνει άνεμος, ονομάστηκε έτσι προς τιμήν του Ιάπωνα Lijima, πατέρα της ανάλυσης scale space (scale space analysis ). Με την ονομασία αυτή οι δημιουργοί θέλουν να παρομοιάσουν τη μη γραμμική συμπεριφορά του ανέμου κατά την διαδικασία επεξεργασίας του με την αντίστοιχη μη γραμμική διαδικασία θόλωσης της εικόνας που προτείνει ο KAZE αλγόριθμος. Το AKAZE [4.3-2] προέρχεται από το Accelerated KAZE.

Αυτό που πρακτικά κάνει ο αλγόριθμός AKAZE είναι να ανιχνεύει και να περιγράφει features κάνοντας χρήση μη γραμμικού φίλτρου διάχυσης. Πιο συγκεκριμένα, στην αρχή εισάγουμε στον αλγόριθμο μια εικόνα. Μετά κάνουμε χρήση της AOS (Additive Operator Splitting) τεχνικής για να δημιουργήσουμε το μη γραμμικό scale space και την variable conductance diffusion. Στην συνέχεια ανιχνεύουμε τα 2D *features* ενδιαφέροντος που παρουσιάζουν το maxima της κανονικοποιημένης scale παραγώγου της Hessian ανταπόκρισης μέσο μη-γραμμικού scale space.

Τέλος υπολογίζεται ο προσανατολισμός των keypoints και αποκτούμε ένα descriptor που είναι αμετάβλητο σε κλίμακα και περιστροφή, παίρνοντας υπόψη την πρώτης τάξης παράγωγο της εικόνας.

Ο αλγόριθμός AKAZE αυτό που κάνει είναι να ανιχνεύει και να περιγράφει features κάνοντας χρήση μη γραμμικού φίλτρου διάχυσης. Η αλληλουχία των βημάτων που ακολουθείται στον αλγόριθμο είναι:

1. Εισάγουμε στον αλγόριθμο μια εικόνα
2. Κάνουμε χρήση της AOS Scheme για να δημιουργήσουμε το μη γραμμικό scale space και την variable conductance diffusion στην εικόνα. Η εξίσωση AOS Scheme [8] που χρησιμοποιείται για να το κάνουμε αυτό είναι

$$L^{i+1} = \left( I - (t_{i+1} - t_i) \cdot \sum_{l=1}^{m} A_l \left( L^i \right) \right)^{-1} L^i \quad (4.2.2.1)$$

3. Στην συνέχεια ανιχνεύουμε τα 2D features ενδιαφέροντος που exhibit a maxima of the scale-normalized determinant of the Hessian [9 ] ανταποκρίνεται σε ένα μη γραμμικό scale space. Η εξίσωση Hessian που χρησιμοποιούμε είναι :

$$L_{Hessian} = \sigma^2 \left( L_{xx} L_{yy} - L_{xy}^2 \right) \quad (4.2.2.2)$$

4. Τέλος βρίσκουμε και υπολογίζουμε (build) τον προσανατολισμό των keypoints. Ο στόχος του αλγόριθμου AKAZE είναι να μειώσει όσο το δυνατόν τον θόρυβο και συγχρόνως να έχουμε την μέγιστη δυνατή ακρίβεια εντοπισμού (localization) και μοναδικότητας (distinctiveness). Τα παραπάνω τα πετυχαίνει με το να θολώνει προσωρινά τα δεδομένα τις εικόνας ενώ ταυτόχρονα διατηρούνται οι μορφές των αντικειμένων.

## 4.4. RANSAC

RANdom SAmple Consensus (RANSAC) [4.4-7] είναι μια επαναλαμβανόμενη μέθοδος που προσεγγίζει παραμέτρους σε ένα μαθηματικό μοντέλο από ένα σύνολο δεδομένων. Δηλαδή έχει την ικανότητα να βρει τις παραμέτρους μιας συνάρτησης μέσω ενός συνόλου τιμών που του εισάγουμε. Ο αλγόριθμος RANSAC δημοσιεύτηκε το 1981 από τους Fischler και Bolles.

Ο αλγόριθμος RANSAC χαρακτηρίζεται σαν randomized estimator. Δηλαδή κάνει μια τυχαία εκτίμηση όσον αφορά ποιες είναι οι σωστές παράμετροι ενός μαθηματικού μοντέλου. Ο αλγόριθμος έχει εφαρμοστεί στην επίλυση μιας μεγάλης ποικιλίας προβλημάτων που συσχετίζονται με την εύρεση παραμέτρων για μοντέλα εκτίμησης. Όπως είναι για παράδειγμα ο υπολογισμός των παραμέτρων του μοντέλου της υπολογιστικής όρασης, δίνοντας του την δυνατότητα αντιστοίχισης, καταχώρησης ή και ανίχνευσης των γεωμετρικών σχεδίων.

Ο αλγόριθμος RANSAC αποτελείται ουσιαστικά από δύο βήματα που αποτελούν την επαναλαμβανόμενη διαδικασία :

- Στο πρώτο βήμα επιλέγεται ένα υποσύνολο, που περιέχει ελάχιστα στοιχεία των δεδομένων, τυχαία από το σύνολο δεδομένων εισόδου. Μέσα από το υποσύνολο αυτό υπολογίζεται μαθηματικό μοντέλο.
- Στο δεύτερο βήμα ο αλγόριθμος ελέγχει ποιο από τα υπόλοιπα στοιχεία του δείγματος είναι σταθερό σε σχέση με το μοντέλο, που προήλθε με την χρήση της παραμέτρου του πρώτου βήματος. Όσα στοιχεία δεν είναι σταθερά θεωρούνται ως ακραίες τιμές που προκλήθηκαν από θόρυβο και αγνοούνται.

Η διαδικασία αυτή συνεχίζεται N φορές, όπου N είναι τα στοιχεία του δείγματος.

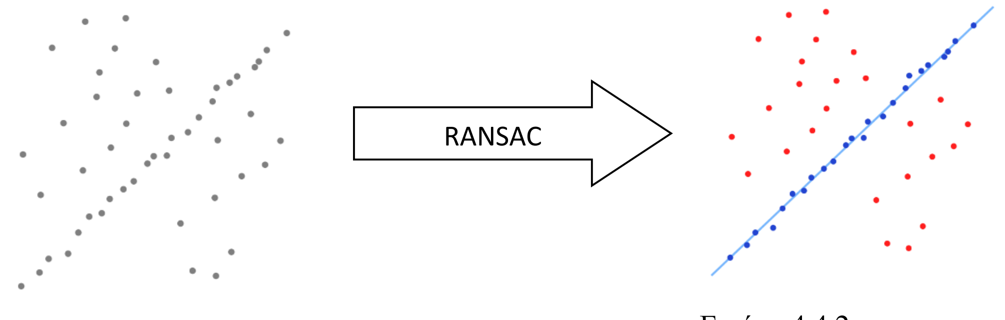
Με αυτόν τον τρόπο προσπαθούμε να βρούμε το μοντέλο που θα έχει τα περισσότερα σταθερά δείγματα ώστε να βρούμε την βέλτιστη λύση.

Ο αριθμός N που απαιτείται από τον αλγόριθμο RANSAC για τον υπολογισμό του βέλτιστου μοντέλου για την εύρεση των σωστών παραμέτρων μπορεί να υπολογιστεί από την εξίσωση:

$$N = \frac{log(1-p)}{log(\,1-(\,1-\varepsilon)^{s})} \qquad (4.2.2.1)$$

- Όπου ε είναι η πιθανότητα το δεδομένο να είναι λανθασμένο (outliner) άρα $(\,1-\varepsilon)$ είναι η πιθανότητα να είναι inliner το δεδομένο.
- Το s είναι ένα σύνολο τιμών και το $(\,1-\varepsilon)^{s}$ είναι η πιθανότητα του να είναι τα δεδομένα του συνόλου αυτού inliner.
- Το p είναι ο αριθμός των outliner που έχει το σύνολο των δεδομένων.

Για παράδειγμα στην Εικόνα 4.4.1 έχουμε ένα σύνολο δεδομένων. Αφού εκτελέσουμε τα παραπάνω βήματα βλέπουμε ότι το μαθηματικό μοντέλο που έχει τα περισσότερα σταθερά σημεία είναι μια ευθεία, όπως φαίνεται και στην Εικόνα 4.4.2. Με κόκκινο είναι οι τιμές που προκλήθηκαν από τον θόρυβο και με μπλε οι σωστές που θα σχηματίσουν την ευθεία .




Εικόνα 4.4.1

Εικόνα 4.4.2

# Κεφάλαιο 5. - Δοκιμή εφαρμογής και αξιολόγηση

Κατά την διάρκεια τις προσπάθειας μας να λύσουμε το SLAM πρόβλημα, πειραματιστήκαμε με τους αλγόριθμους που θα μπορούσαμε να χρησιμοποιήσουμε και πώς θα μπορούσαμε με τον κατάλληλο συνδυασμό τους να πετύχουμε το βέλτιστο αποτέλεσμα. Στο παρόν κεφάλαιο αναφερόμαστε στους αλγόριθμους με τους οποίους πειραματιστήκαμε, σε αυτούς που απορρίψαμε (και το γιατί) καθώς και ποιους θεωρήσαμε ότι είναι οι κατάλληλοι για την επίλυση του SLAM.

## 5.1. Εύρεση σημαντικών σημείων (keypoints) στο Matlab με τη χρίση του Kinect και του αλγόριθμου Sift.

Στην ενότητα αυτή χρησιμοποιούμε την συνάρτηση του Sift και του match ώστε να μπορέσουμε να βρούμε τα σημεία ενδιαφέροντος στο χώρο αφού πρώτα έχουμε λάβει δεδομένα από το περιβάλλον μέσα του Kinect. Στην συνέχεια, με την συνάρτηση του match, βρίσκουμε τα κοινά σημεία της προηγούμενης εικόνας με την τωρινή, με σκοπό να μπορέσουμε να κατατοπιστούμε στον χώρο.

Σύνδεση του Kinect με το MatLab . Κάνουμε χρήση δύο vedeoinput ώστε να μπορέσουμε να λάβουμε από τον αισθητήρα μας RGB εικόνα και RGB-D εικόνα.

```
vid = videoinput('kinect',1);
vidD = videoinput('kinect',2);
```

Φόρτωσε τις ιδιότητες πηγής της depth κάμερας.

```
srcDepth = getselectedsource(vidD);
```

Θέτουμε την αναλογία των frame ανά trigger να είναι 1.

```
vid.FramesPerTrigger = 1;
vidD.FramesPerTrigger = 1;
```

Θέτουμε το trigger να έχει την τιμή 29 , ώστε να κατορθώσουμε να κάνουμε λήψη 30 frames

από την κάμερα χρώματος (RGB).

```
vid.TriggerRepeat = 29;
vidD.TriggerRepeat = 29;
```

Δηλώνουμε αν η κάμερα μας είναι σε manual triggering, ώστε να μπορούμε να την μεταβάλλουμε σύμφωνα με τις αλλαγές που κάναμε παραπάνω στο trigger.

```
triggerconfig([vid vidD] ,'manual');
```

Ξεκινάμε την λήψη της εικόνας χρώματος και βάθους.

```
start([vid vidD]);
```

Στην συγκεκριμένη for προσπαθούμε να συλλέξουμε δεδομένα από την κάμερα βάθους καθώς και από τη κάμερα χρώματος. Τα δεδομένα που εκλαμβάνουμε από την εικόνα χρώματος είναι το χρώμα, ο χρόνος (σε sec) και το metadata. Αντίστοιχα για την εικόνα βάθους λαμβάνουμε x,y και z του κάθε σημείου στο χώρο , τον χρόνο και το metadata.

```
for i = 1:29
    preview(vid);
    preview(vidD);
    trigger([vid vid])
    [img1Color, ts1_color, metaData1_Color] = getdata(vid);
    [img1Depth, ts1_depth, metaData1_Depth] = getdata(vidD);

end
```

Αποθηκεύουμε τα δεδομένα σε μορφή εικόνας .pgm , ώστε να μπορέσουμε στην συνέχεια να εφαρμόσουμε πάνω στην εικόνα τον αλγόριθμο sift και να βρούμε τα σημεία ενδιαφέροντος.

```
imwrite(img1Color,'my1Gray.pgm')
```

Το σταματάμε 10 δευτερόλεπτα ώστε να μπορέσει το μηχάνημα να μετακινηθεί και να έχουμε λήψη νέας εικόνας.

```
pause(10);
```

Στην συγκεκριμένη for εφαρμόζουμε την ίδια τακτική με την προηγούμενη for ώστε να μπορέσουμε να κάνουμε λήψη μιας νέας εικόνας από τον αισθητήρα μας. Η αποθήκευση των δεδομένων γίνεται με τον ίδιο τρόπο που έγινε στην προηγούμενη for.

```
for i = 1:29

    preview(vid);
    trigger(vid );
    [img2Color, ts2_color, metaData2_Color] = getdata(vid);
    [img2Depth, ts2_depth, metaData2_Depth] = getdata(vidD);

end
```

Αποθηκεύουμε σε μορφή εικόνας .pgm τα νέα μας δεδομένα.

```
imwrite(img2Color,'my2Gray.pgm')
```

Σταματάμε το preview και στην συνέχεια σταματάμε τη συνδέση με το Kinect.

```
closepreview(vid);
closepreview(vidD);
stop(vid,vidD);
```

Εκτελούμε τον αλγόριθμο sift πάνω στην πρώτη εικόνα. Αυτό που παίρνουμε σαν αποτέλεσμα είναι η εικόνα που είχαμε, η τοποθεσία των σημαντικών σημείων καθώς και ένα μικρό σημαδάκι πάνω στο σημαντικό σημείο (το οποίο μας βοηθάει στην εύκολη αναγνώριση του σημαντικού σημείου όταν κάνουμε show).

```
[image1, descrips1, locs1] = sift('my1Gray.pgm');
```

Εμφανίζουμε τα σημαντικά σημεία πάνω στην πρώτη μας εικόνα.

```
showkeys(img1Color, locs1);
```

Εκτελούμε τον αλγόριθμο Sift πάνω στη δεύτερη εικόνα μας. Αυτό που παιρνούμε σαν αποτέλεσμα, όπως και πριν είναι η εικόνα που είχαμε, την τοποθεσία των σημαντικών σημείων καθώς και ένα μικρό σημαδάκι πάνω στο σημαντικό σημείο (το οποίο μας βοηθάει

στην εύκολη αναγνώριση του σημαντικού σημείου όταν κάνουμε show).

```
[image2, descrips2, locs2] = sift('my2Gray.pgm');
```

Εμφανίζουμε τα σημαντικά σημεία πάνω στην δεύτερη μας εικόνα.

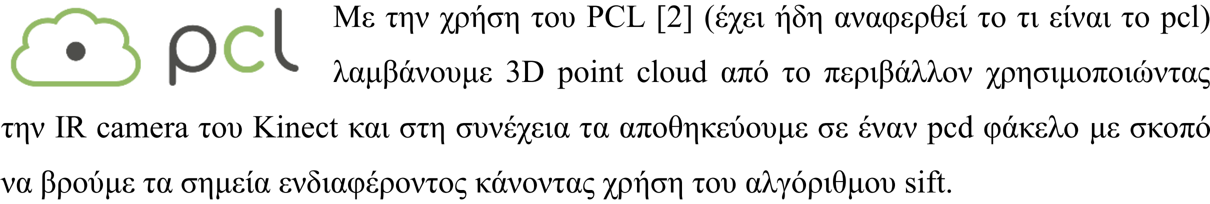
```
showkeys(img2Color, locs2);
```

Κάνοντας χρήση της συνάρτησης match μπορούμε να βρούμε τα κοίνα σημεία των δύο εικόνων.

```
match('my1Gray.pgm', 'my2Gray.pgm');
```

Παρόλο που τα δεδομένα που λαμβάνουμε από το περιβάλλον μπορούσαμε να τα επεξεργαστούμε γύρω στα 2sec, δυστυχώς η διαδικασία αυτή δεν θα μπορούσε να υλοποιηθεί για real-time SLAM και δεν την χρησιμοποιήσαμε στην εργασία.

## 5.2. Εύρεση σημαντικών σημείων μέσω του Point Cloud Library (PCL) και Kinect.




Με την χρήση του PCL [2] (έχει ήδη αναφερθεί το τι είναι το pcl) λαμβάνουμε 3D point cloud από το περιβάλλον χρησιμοποιώντας την IR camera του Kinect και στη συνέχεια τα αποθηκεύουμε σε έναν pcd φάκελο με σκοπό να βρούμε τα σημεία ενδιαφέροντος κάνοντας χρήση του αλγόριθμου sift.

Ο κώδικας μας χωρίζεται σε δύο σημεία. Στο πρώτο κομμάτι κάνουμε λήψη ενός 3D point cloud από το περιβάλλον και στην συνέχεια το αποθηκεύουμε σε ένα pcd αρχείο. Στο δεύτερο κομμάτι του κώδικα αυτό που κάνουμε είναι να διαβάζουμε το αρχείο αυτό και να εκτελούμε την συνάρτηση Sift.

*Πρώτο κομμάτι του κώδικα :*

| Δήλωση των βιβλιοθηκών (modules) που θα χρησιμοποιήσουμε |
|---|

```
#include <iostream>
#include <string>
#include <sstream>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/io/openni_grabber.h>
#include <pcl/visualization/cloud_viewer.h>
#include <pcl/features/integral_image_normal.h>
#include <pcl/io/io.h>
```

| Δηλώνουμε σε ποιον φάκελο στον υπολογιστή μας θα γίνει η αποθήκευση του pcd αρχείου. |
|---|

```
using namespace std;
const string OUT_DIR = "C:\\Kinects_Outputs\\";

class SimpleOpenNIViewer
{
public:
```

| Η λειτουργία της συγκεκριμένης συνάρτησης είναι να δώσει όνομα στο pcd αρχείο που θέλουμε να αποθηκεύσουμε καθώς επίσης και να δηλώσουμε σε τι μορφή θα είναι καταγεγραμμένα τα δεδομένα. Στην προκειμένη περίπτωση τα έχουμε ορίσει να είναι σε ASCII μορφή. Την καλούμε από την συνάρτηση run(). |
|---|

```
SimpleOpenNIViewer () : viewer ("PCL Viewer")
    {
                frames_saved = 0;
                save_one = false;
    }
       void cloud_cb_ (const
pcl::PointCloud<pcl::PointXYZRGBA>::ConstPtr &cloud)
    {
                if (!viewer.wasStopped()) {
                        viewer.showCloud (cloud);
                     if( save_one ) {
                                save_one = false;
                                std::stringstream out;
                                out << frames_saved;
                                std::string name = OUT_DIR + "cloud"
+ out.str() + ".pcd";
```

```
                                pcl::io::savePCDFileASCII( name,
*cloud );
                        }
                }     }
```

Κατά την διάρκεια τις εκτέλεσης της συνάρτηση run() κάνουμε την αποθήκευση του pcd αρχείου. Επίσης εμφανίζει ένα παράθυρο που δείχνει στον χρήστη σε real-time το τι αποθηκεύει.

```
void run ()
    {
                pcl::Grabber* interface = new pcl::OpenNIGrabber();
   boost::function<void (const
pcl::PointCloud<pcl::PointXYZRGBA>::ConstPtr&)> f = boost::bind
(&SimpleOpenNIViewer::cloud_cb_, this, _1);
      interface->registerCallback (f);
interface->start ();
char c;
               while (!viewer.wasStopped())
                {
c = getchar();
                          if( c == 's' ) {
                                cout << "Saving frame " <<
frames_saved << ".\n";
                                frames_saved++;
                                save_one = true;
                        }
                }
interface->stop ();
        }
pcl::visualization::CloudViewer viewer;
private:
                int frames_saved;
                bool save_one; };
```

Η λειτουργία της main μας είναι να δηλώνουμε μια μεταβλητή τύπου της κλάσης που έχουμε φτιάξει πριν και στην συνέχεια καλούμε την συνάρτηση run() της κλάσης.

```
int main ()
{
    SimpleOpenNIViewer v;
    v.run ();
    return 0;
}
```

*Δεύτερο κομμάτι του κώδικα*

Δήλωση των βιβλιοθηκών (modules) που θα χρησιμοποιήσουμε

```
#include <iostream>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/common/io.h>
#include <pcl/keypoints/sift_keypoint.h>
#include <pcl/features/normal_3d.h>
```

Εισάγουμε το pcd αρχείο που περιέχει το point cloud μας

```
int main(int, char** argv)
{
  std::cout << "Reading " <<
"C:\\Kinects_Outputs\\Binary\\cloud2.pcd" << std::endl;
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_xyz (new
pcl::PointCloud<pcl::PointXYZ>);
  if(pcl::io::loadPCDFile<pcl::PointXYZ>
("C:\\Kinects_Outputs\\Binary\\cloud2.pcd", *cloud_xyz) == -1) //
load the file
  {
    PCL_ERROR ("Couldn't read file");
    return -1;
  }
  std::cout << "points: " << cloud_xyz->points.size () <<std::endl;
```

Εκτελούμε την συνάρτηση του sift

```
  const float min_scale = 0.01f;
  const int n_octaves = 3;
  const int n_scales_per_octave = 4;
  const float min_contrast = 0.001f;
  pcl::NormalEstimation<pcl::PointXYZ, pcl::PointNormal> ne;
  pcl::PointCloud<pcl::PointNormal>::Ptr cloud_normals (new
pcl::PointCloud<pcl::PointNormal>);
  pcl::search::KdTree<pcl::PointXYZ>::Ptr tree_n(new
pcl::search::KdTree<pcl::PointXYZ>());
  ne.setInputCloud(cloud_xyz);
  ne.setSearchMethod(tree_n);
  ne.setRadiusSearch(0.2);
  ne.compute(*cloud_normals);
  for(size_t i = 0; i<cloud_normals->points.size(); ++i)
  {
    cloud_normals->points[i].x = cloud_xyz->points[i].x;
    cloud_normals->points[i].y = cloud_xyz->points[i].y;
    cloud_normals->points[i].z = cloud_xyz->points[i].z;
  }
```

Υπολογίζουμε τα σημεία ενδιαφέροντος 3D και συγχρόνως τα εμφανίζουμε.

```
  pcl::SIFTKeypoint<pcl::PointNormal, pcl::PointWithScale> sift;
  pcl::PointCloud<pcl::PointWithScale> result;
  pcl::search::KdTree<pcl::PointNormal>::Ptr tree(new
pcl::search::KdTree<pcl::PointNormal> ());
  sift.setSearchMethod(tree);
  sift.setScales(min_scale, n_octaves, n_scales_per_octave);
  sift.setMinimumContrast(min_contrast);
  sift.setInputCloud(cloud_normals);
  sift.compute(result);

  std::cout << "No of SIFT points in the result are " <<
result.points.size () << std::endl;


  return 0;

}
```

Ο αριθμός των δεδομένων που έχουμε λάβει σε 3D μέσω του Kinect από το περιβάλλον είναι πολύ μεγάλος. Έτσι η εκτέλεση της συνάρτησης Sift διαρκεί πολλή ώρα. Δοκιμάσαμε να αφαιρέσουμε την “άχρηστη” πληροφορία από τον pcd φάκελο μας ώστε να μειώσουμε τον χρόνο εκτέλεσης της συνάρτησης sift. Πέτυχε αλλά πάλι ο χρόνος δεν ήταν τόσο μικρός ώστε να μπορούμε να το χρησιμοποιήσουμε για να πραγματοποιήσουμε real-time SLAM.

## 5.3. Δοκιμή της MRPT βιβλιοθήκης για την υλοποίηση της εφαρμογής

Όταν είχαμε εντρυφήσει αρκετά σε θεωρία, ξεκινήσαμε να ψάχνουμε για τον τρόπο υλοποίησης της εφαρμογής. Μετά από αρκετή αναζήτηση ανακαλύψαμε δύο χρήσιμα εργαλεία που θα μας βοηθούσαν. Το πρώτο είναι το ROS, το οποίο έχουμε ήδη εξηγήσει. Το άλλο ήταν μία βιβλιοθήκη που ονομαζόταν Mobile Robot Programming Toolkit (MRPT). Το MRPT, όπως γράφει και στην ιστοσελίδα του [5.3-1], είναι μία βιβλιοθήκη γραμμένη σε C++ που έχει πολλά εργαλεία για κινούμενα ρομπότ. Αποφασίσαμε να κάνουμε χρήση του MRPT αρχικά, έναντι του ROS, διότι φαινόταν να έχει πολλά χρήσιμα εργαλεία για τον σκοπό μας. Το ROS, σε γενικές γραμμές, δεν έχει ακόμη πολύ μεγάλη ποικιλία πακέτων για κινούμενα ρομπότ. Καταφέραμε να συνδέσουμε το Kinect με το MRPT, να παίρνουμε 2-D και 3-D δεδομένα από αυτό, να κάνουμε πολύ βασική χαρτογράφηση του χώρου και να

απεικονίσουμε γραφικά τα δεδομένα του Kinect. Μετά από αρκετό καιρό βλέποντας σε βάθος την βιβλιοθήκη και τα προβλήματα της, αποφασίσαμε να την αφήσουμε και να δούμε το ROS. Οι δύο βασικοί λόγοι που το αφήσαμε ήταν:

- έλλειψη καλής βιβλιογραφίας – Η μεγάλη πλειοψηφία της βιβλιογραφίας ήταν ελλειπής και το forum που είχε φτιαχτεί για τη βιβλιοθήκη ήταν τελείως ανενεργό. Ο ίδιος ο δημιουργός απάντησε σε ένα ερώτημα μας μετά από περίπου 40 μέρες.
- ανικανότητα να γίνει σωστό configuration του Kinect – Το Kinect έχει την RBG camera του τοποθετημένη δίπλα στον IR δέκτη. Αυτό θα πει ότι υπάρχει μια απόσταση ανάμεσα στα pixel του ενός με τα pixel του άλλου. Ενώ έγιναν πολλές προσπάθειες, ποτέ δεν καταφέραμε να κάνουμε σωστό configure την κάμερα με τα εργαλεία που μας έδινε η βιβλιοθήκη.

Ενώ αυτοί ήταν οι βασικοί λόγοι, δεν ήταν οι μοναδικοί. Υπήρχαν επίσης αρκετά bugs, δυσκολία χρήσης του OpenCV με τον τρόπο που παρουσίαζε το MRPT τα δεδομένα του Kinect και τα περισσότερα εργαλεία (εάν όχι όλα) ήταν σχεδιασμένα για grid-based SLAM και όχι feature-based, όπως επιθυμούσαμε. Όπως δήλωσε ο ίδιος ο δημιουργός, η βιβλιοθήκη σχεδιάστηκε για συγκεκριμένο ερευνητικό σκοπό και για την ώρα δεν θα μπορεί να ασχοληθεί με την αναπτύξη της σε βαθμό που να θεωρείται πλήρως λειτουργική η βιβλιοθήκη, αν και μπορεί να γίνει κάτι τέτοιο στο μέλλον. Δυστυχώς, λόγω τεχνικής βλάβης που δημιουργήθηκε στον σκληρό δίσκο του υπολογιστή που χρησιμοποιήσαμε τότε, χάθηκε όλος ο κώδικας που είχαμε γράψει.

# Κεφάλαιο 6. - Τελική Υλοποίηση

Ήρθε η ώρα να περιγράψουμε την τελική υλοποίηση. Τα πάντα έγιναν σε ένα πακέτο ROS που φτιάξαμε. Αρχικά θα μιλήσουμε για τις γενικές ρυθμίσεις του πακέτου μας. Στην συνέχεια θα αναλύσουμε τα δύο κεντρικά μας nodes. Τέλος θα αναφέρουμε τα τύπου δεδομένων που μεταφέρουμε ανάμεσα στα topics και σε δύο βάσεις δεδομένων που κρατάμε.

## 6.1. Κτίσιμο ROS πακέτου

Ένα ROS πακέτο για να υπάρχει χρειάζεται δύο σημαντικά αρχεία. Το "CMakeLists.txt" και το "package.xml".

| ***CMakeLists.txt*** |
|---|
| Εδώ κάνουμε linking τα αρχεία με τις βιβλιοθήκες που χρησιμοποιούμε (όπως OpenCV και PCL), κάνουμε linking τα αρχεία που φτιάξαμε εμείς με header, δηλώνουμε ROS msg και srv, περιγράφουμε με ποιον τρόπο θα χρησιμοποιήσουμε packages και λειτουργίες του ROS και άλλα configuration. |
| code: |

```
cmake_minimum_required(VERSION 2.8.3)
project(otacon)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED
COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
  rgbd_launch
  freenect_camera
  freenect_launch
  cmake_modules
)


#############
## TinyXML ##
#############

find_package(TinyXML REQUIRED)
```

```
#########################
## Point Cloud Library ##
#########################

find_package(PCL 1.3 REQUIRED COMPONENTS common io)
include_directories(${PCL_INCLUDE_DIRS})
link_directories(${PCL_LIBRARY_DIRS})
add_definitions(${PCL_DEFINITIONS})
find_package(catkin REQUIRED COMPONENTS  pcl_ros pcl_conversions)

################################################
## Declare ROS messages, services and actions ##
################################################

## To declare and build messages, services or actions from within
this
## package

## Generate messages in the 'msg' folder
 add_message_files(
   FILES
   feature_coordinates.msg
   feature_coordinates_array.msg
   pose.msg
   movement.msg
 )

## Generate services in the 'srv' folder
 add_service_files(
   FILES
   motion_srv.srv
 )


## Generate added messages and services with any dependencies
listed here
 generate_messages(
   DEPENDENCIES
   std_msgs
 )


###################################
## catkin specific configuration ##
###################################
## The catkin_package macro generates cmake config files for your
package
## Declare things to be passed to dependent projects

catkin_package(
  INCLUDE_DIRS include
  CATKIN_DEPENDS roscpp rospy std_msgs message_runtime
rgbd_launch freenect_camera freenect_launch freenect_stack
  DEPENDS TinyXML
)
```

```
###########
## Build ##
###########

## Specify additional locations of header files
## Your package locations should be listed before other locations
# include_directories(include)
include_directories(

  include
  ${catkin_INCLUDE_DIRS}
  ${TinyXML_INCLUDE_DIRS}
)

## Add cmake target dependencies of the library
## as an example, code may need to be generated before libraries
## either from message generation or dynamic reconfigure
# add_dependencies(otacon ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})

## Declare a C++ executable
add_executable(front_end_node src/front_end.cpp)

## Add cmake target dependencies of the executable
## same as for the library above
# add_dependencies(otacon_node
${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})

## Specify libraries to link a library or executable target
against
add_library(fe_ImageTools src/fe_ImageTools.cpp)
add_library(fe_MainTools src/fe_MainTools.cpp)
add_library(tinyxml lib/tinyxml/tinyxml.cpp)

target_link_libraries(fe_MainTools fe_ImageTools
${catkin_LIBRARIES} )
target_link_libraries(front_end_node fe_MainTools fe_ImageTools
${catkin_LIBRARIES} )
target_link_libraries(fe_ImageTools ${TinyXML_LIBRARIES} )


#############
## OpenCV ##
#############

find_package(OpenCV REQUIRED)
include_directories(${OpenCV_INCLUDE_DIRS})
target_link_libraries(front_end_node ${OpenCV_LIBRARIES})
target_link_libraries(fe_ImageTools ${OpenCV_LIBRARIES})
target_link_libraries(fe_MainTools ${OpenCV_LIBRARIES})
```

***package.xml***

Εδώ έχει πιο τυπικές πληροφορίες όπως όνομα πακέτου, την άδεια του πακέτου, ονόματα των developers, τρόπος επικοινωνίας αλλά και σημαντικά πράγματα όπως ποια πακέτα του ROS θα χρησιμοποιήσουμε.

***code:***

```
<?xml version="1.0"?>
<package>
  <name>otacon</name>
  <version>0.0.0</version>
  <description>This is my robot named "otacon". It is desinged to
perform 3D FastSLAM using a RGB-D sensor.</description>

  <!-- One maintainer tag required, multiple allowed, one person
per tag -->
  <!-- Example:  -->
  <!-- <maintainer email="jane.doe@example.com">Jane
Doe</maintainer> -->
  <maintainer email="d.pantelis2@hotmail.com">D.
Pantelis</maintainer>
  <!-- One license tag required, multiple allowed, one license
per tag -->
  <!-- Commonly used license strings: -->
  <!--   BSD, MIT, Boost Software License, GPLv2, GPLv3,
LGPLv2.1, LGPLv3 -->
  <license>TODO</license>


  <!-- Url tags are optional, but mutiple are allowed, one per
tag -->
  <!-- Optional attribute type can be: website, bugtracker, or
repository -->
  <!-- Example: -->
  <!-- <url
type="website">http://wiki.ros.org/read_kinect_01</url> -->


  <!-- Author tags are optional, mutiple are allowed, one per tag
-->
  <!-- Authors do not have to be maintianers, but could be -->
  <!-- Example: -->
  <!-- <author email="jane.doe@example.com">Jane Doe</author> -->


  <!-- The *_depend tags are used to specify dependencies -->
  <!-- Dependencies can be catkin packages or system dependencies
-->
  <!-- Examples: -->
  <!-- Use build_depend for packages you need at compile time: --
>
  <build_depend>message_generation</build_depend>
  <!-- Use buildtool_depend for build tool packages: -->
  <!--   <buildtool_depend>catkin</buildtool_depend> -->
  <!-- Use run_depend for packages you need at runtime: -->
```

```
  <run_depend>message_runtime</run_depend>
  <!-- Use test_depend for packages you need only for testing: --
>
  <!--   <test_depend>gtest</test_depend> -->
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>rospy</run_depend>
  <run_depend>std_msgs</run_depend>

  <build_depend>rgbd_launch</build_depend>
  <run_depend>rgbd_launch</run_depend>
  <build_depend>freenect_camera</build_depend>
  <run_depend>freenect_camera</run_depend>
  <build_depend>freenect_launch</build_depend>
  <run_depend>freenect_launch</run_depend>
  <build_depend>freenect_stack</build_depend>
  <run_depend>freenect_stack</run_depend>

  <build_depend>pcl_ros</build_depend>
  <run_depend>pcl_ros</run_depend>
  <build_depend>pcl_conversions</build_depend>
  <run_depend>pcl_conversions</run_depend>

  <build_depend>cmake_modules</build_depend>
  <build_depend>tinyxml</build_depend>
  <run_depend>tinyxml</run_depend>


  <!-- The export tag contains other, unspecified, tags -->
  <export>
    <!-- Other tools can request additional information be placed
here -->

  </export>
</package>
```

Ας εξηγήσουμε την δομή του προγράμματος.

Έχουμε τα δύο κεντρικά μας nodes:

- front_end_node – γραμμένο σε C++
- SLAM – γραμμένο σε python

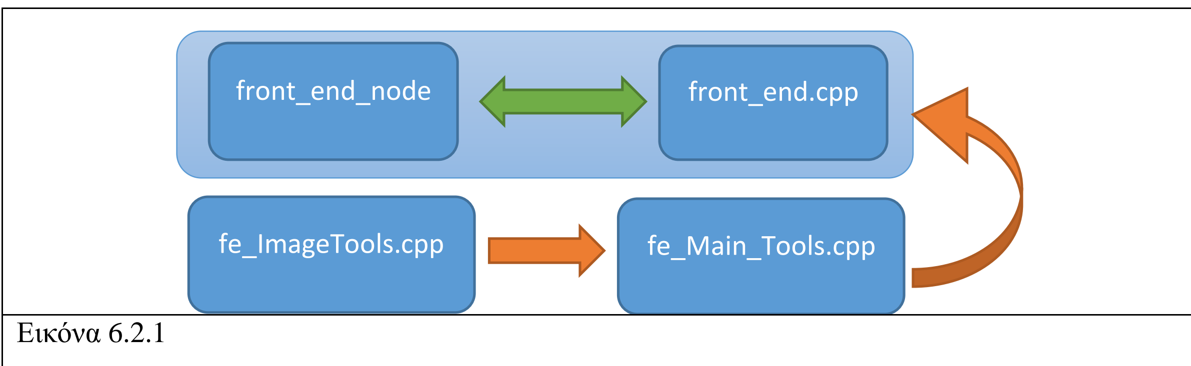
## 6.2. Front_end_node

### 6.2.1. Γενικές πληροφορίες

Δουλειά αυτού του node είναι:

- Να παίρνει μετρήσεις από το Kinect
- Να βρίσκει μοναδικά features με την βοήθεια του AKAZE αλγορίθμου
- Να αποθηκεύει αυτά τα σημεία στο database του
- Να κάνει data association την καινούρια εικόνα με τις αποθηκευμένες
- Να αντιστοιχίσει τα features με τις 3D συντεταγμένες που δίνει το Kinect
- Να στέλνει στο SLAM ένα συγκεκριμένο αριθμό από features

Το **front_end_node** κάνει χρήση του **front_end.cpp** αρχείου ως το βασικό του source code. Το **front_end.cpp** κάνει χρήση πολλών σχεδιασμένων από εμάς συναρτήσεων που βρίσκονται μέσα στο **fe_MainTools.cpp**. Αντίστοιχα, πολλές συναρτήσεις του **fe_MainTools.cpp** είναι από το **fe_ImageTools.cpp**




Εικόνα 6.2.1

Το παραπάνω διάγραμμα δίνει μια ιδέα για το πώς λειτουργεί το front end. Έχει το front_end_node να είναι στην ουσία το front_end.cpp. Η κατεύθυνση που έχουν τα **πορτοκαλί βελάκια** δηλώνουν σε ποιο αρχείο δίνουν τις υπηρεσίες τους.

### 6.2.2. front_end κώδικας και αλγόριθμος

| Ας δούμε τον αλγόριθμο που εκτελεί το front_end_node. |
|---|
| 1. Ελέγχει εάν έχει τίποτα στην database του. Εάν έχει, πηγαίνει απευθείας στην γραμμή #6 αλλιώς πηγαίνει από κάτω<br>2. Περιμένει για δύο RGB frames που είναι όσο πιο κοντά γίνεται |

3. Βρίσκει στο κάθε frame τα AKAZE features τους
4. Τα συγκρίνει μεταξύ τους και κρατάει μόνο όσα είναι κοινά
5. Τα αποθηκεύει στην database
6. Loop_start
7. Περιμένει για δύο RGB frames που είναι όσο πιο κοντά γίνεται
8. Βρίσκει στο κάθε frame τα AKAZE features τους
9. Τα συγκρίνει μεταξύ τους και κρατάει μόνο όσα είναι κοινά.
10. Συγκρίνει τα features με το database να δει αν υπάρχει ήδη αυτό το σύνολο από features
11. Παίρνει τα αποτελέσματα από την αναζήτηση και τα βάζει σε αύξουσα σειρά σύμφωνα με το πόσο κοντά είναι τα τωρινά features με τα αποθηκευμένα
12. Διαγράφει τυχόν features που είναι διπλά
13. Συγκρίνει τα τωρινά features με αυτά που έχει ήδη ενσωματώσει το SLAM μέσα στον χάρτη του. Αυτό γίνεται διότι από τον συνολικό αριθμό features που θα αποσταλούν, τα μισά θα είναι καινούρια και τα άλλα μισά παλιά
14. Βρίσκει τις 3D καρτεσιανές συντεταγμένες (σε σχέση με το local frame του ρομπότ) και μαζί με το id features και το old tag, τα στέλνει στο SLAM.
15. Loop_end

***code:***

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "geometry_msgs/Point.h"
#include "sensor_msgs/PointCloud2.h"
#include "otacon/feature_coordinates_array.h"

#include "pcl_ros/point_cloud.h"
#include <boost/make_shared.hpp>

#include <iostream>
#include <fstream>
#include <stdio.h>
#include <vector>

#include <opencv2/opencv.hpp>
#include <opencv2/features2d.hpp>
#include <opencv2/imgcodecs.hpp>
#include "opencv2/core.hpp"
#include "opencv2/calib3d.hpp"
#include "opencv2/imgcodecs.hpp"

#include "fe_ImageTools.h"
```

```
#include "fe_MainTools.h"

using namespace cv;
using namespace std;
ofstream myfile;

string topothesia =
"/home/jim/ROS/catkin_ws/src/otacon/xml/database_1.xml";

class Subscribe_And_Publish
{
private:
	ros::Publisher pub;
	ros::Subscriber sub;
	ros::NodeHandle n;
	Mat img1, img2;
	int pointID_last;
	int max_points_keep;
public:


	Subscribe_And_Publish()
	{
		FileStorage fs(topothesia, FileStorage::READ);
		fs["pointID_last"] >> pointID_last;
		fs.release();

		max_points_keep = 6;
		if(max_points_keep%2 != 0) { max_points_keep++; }

		//Topic you want to publish
		sub =
n.subscribe<pcl::PointCloud<pcl::PointXYZRGB> >
("/camera/depth_registered/points", 5,
&Subscribe_And_Publish::callback, this);

		//Topic you want to subscribe
		pub =
n.advertise<otacon::feature_coordinates_array>
("feature_coordinates", 500);


	}

	void callback(const
pcl::PointCloud<pcl::PointXYZRGB>::ConstPtr& cloud)
	{
		Mat imageFrame;
		unsigned char gray;
		pcl::PointCloud<pcl::PointXYZRGB> cloud_f1;
		bool error = false;

		if(cloud->isOrganized())
		{
			imageFrame = Mat(cloud->height, cloud->width,
CV_8UC1);
```

```
                cloud_f1.width = cloud->width;
                cloud_f1.height = cloud->height;
                cloud_f1.points.resize(cloud->width * cloud-
>height);

                for(int h=0; h<imageFrame.rows; h++)
                {
                     for(int w=0; w<imageFrame.cols; w++)
                     {
                          pcl::PointXYZRGB point = cloud-
>at(w,h);
                          cloud_f1.at(w,h) = point;
                          Eigen::Vector3i rgb =
point.getRGBVector3i();
                          gray = (rgb[0]+rgb[1]+rgb[2])/3;

                          imageFrame.at<unsigned char>(h,w)
= gray;
                     }
                }
           }

           if(pointID_last>0)
           {

                if( !img1.data )
                {
                     cout << "image 1 is done" << endl;
                     img1 = imageFrame.clone();

                }
                else
                {
                     cout <<"image 2 is done" << endl;
                     img2 = imageFrame.clone();

                     Mat desc_final;
                     vector<KeyPoint> kpts_final;
                     vector<float> dist_final;
                     vector<bool> old;
                     otacon::feature_coordinates_array
front_end_msg;

                     vector<KeyPoint> kpts1, kpts2;
                     Mat desc1, desc2;
                     find_AKAZE(img1,desc1,kpts1);
                     find_AKAZE(img2,desc2,kpts2);

                     compare_images(desc1, kpts1, desc2,
kpts2, desc_final, kpts_final, dist_final, error);
                     dist_final.clear();

                     if (!error)
                     {
                          make_classID_max(kpts_final);
```

```
                            search_in_database(desc_final,
kpts_final, dist_final, pointID_last, old, error);
                            if (!error)
                            {
                                  kpts_final =
ascending_kpts(kpts_final, dist_final);
                                  kpts_final =
remove_copies(kpts_final);
                                  old.clear();
                                  old = find_old(kpts_final);

                                  //kpts_final =
remove_extra_points(kpts_final, max_points_keep);
                                  get_3D(kpts_final, old,
front_end_msg, cloud_f1, max_points_keep);

                                  pub.publish(front_end_msg);
                            }
                      }

                      img1.release();
                      img2.release();
                      //waitKey(0);
                      //ros::shutdown();
                }
           }
           else
           {
                if( !img1.data )
                {
                      cout << "image 1 is done" << endl;
                      img1 = imageFrame.clone();
                }
                else
                {
                      cout <<"image 2 is done" << endl;
                      img2 = imageFrame.clone();

                      Mat desc_final;
                      vector<KeyPoint> kpts_final;
                      vector<float> dist_final;

                      vector<KeyPoint> kpts1, kpts2;
                      Mat desc1, desc2;
                      find_AKAZE(img1,desc1,kpts1);
                      find_AKAZE(img2,desc2,kpts2);
                      compare_images(desc1, kpts1, desc2,
kpts2, desc_final, kpts_final, dist_final, error);
                      if(!error)
                      {
                            for(int
j=0;j<kpts_final.size();j++)
                            {
                                  pointID_last++;
                                  kpts_final[j].class_id =
```

```
pointID_last;
                                 cout<<"Poins at save -->
"<<kpts_final[j].class_id<<endl;
                           }
                           add_to_database(desc_final,
kpts_final, pointID_last, topothesia);
                      }
                      img1.release();
                      img2.release();
                 }
           }

     }

};//End of class SubscribeAndPublish


int main(int argc, char **argv)
{
     ros::init(argc, argv, "front_end");

     Subscribe_And_Publish SAPObject;

     ros::spin();

     return 0;
}//----------------END_OF_MAIN-------------------------||
```

### 6.2.3. Ανάλυση του fe_MainTools.cpp

Τώρα θα εξετάσουμε ξεχώριστα τις function του fe_MainTools.cpp. Αυτό θα βοηθήσει και περαιτέρω στο ποιες λειτουργίες εκτελεί ο παραπάνω κώδικας.

Του δίνεται μια Gray-scale εικόνα (σε Mat μορφή) και επιστρέφει τα AKAZE keypoints μαζί με τα descriptors τους.

```
void find_AKAZE(Mat img, Mat &desc, vector<KeyPoint> &kpts)
{
     Ptr<AKAZE> akaze = AKAZE::create();
     akaze->detectAndCompute(img, noArray(), kpts, desc);
}
```

Η παράμετρος CV::KeyPoint έχει ένα πεδίο το οποίο είναι το ID. Δίνουμε σε κάθε point που χρησιμοποιούμε ένα τέτοιο ID ώστε να μπορούμε να κάνουμε data association. Με αυτό ξέρουμε ποια features έχει το SLAM. Σε κάθε καινούριο point που βρίσκουμε δίνεται ένας ακέραιος αριθμός για ID. Στο πρώτο point δίνουμε τον

αριθμό 1 και κάθε point που το ακολουθεί του δίνουμε τον επόμενο αριθμό.

Πριν κάνουμε την σύγκριση του καινούριου frame με το παλιό δίνουμε μια πάρα πολύ μεγάλη τιμή στο ID. Στην συνέχεια που θα γίνει το data association με την βάση δεδομένων, όποια σημεία είναι ίδια θα πάρουν το ID που υπάρχει στην βάση δεδομένων. Αντιθέτως, όσα σημεία δεν βρεθούν ίδια, θα έχουν αυτό το πολύ μεγάλο ID ώστε σε μετέπειτα φάση να τους δώσουμε ένα σωστό ID.

```
void make_classID_max(vector<KeyPoint> &kpts_final)
{
      for(int i=0;i<kpts_final.size();i++)
      {
            kpts_final[i].class_id = 32766;
      }
}
```

Εδώ του δίνονται δύο ζευγάρια από KeyPoints και descriptors ώστε με το Brute Force Matcher να τα συγκρίνουμε και να βρούμε τα κοινά σημεία. Αυτή η συνάρτηση χρησιμοποιείται στο front_end.cpp και στο fe_MainTools.cpp για διαφορετικούς λόγους.

Στο front_end.cpp:

Έχουμε κάνει μια διαδικασία ώστε να μειώσουμε το σφάλμα ανάμεσα σε δύο εικόνες που παίρνει το Kinect. Αυτό είναι να πάρουμε δύο frames από την RGB camera (με λίγα frames ανάμεσα τους). Μετά τις στέλνουμε στην παρακάτω function. Εφόσον είναι σχεδόν ολόιδια τα frames θα έπρεπε να βρεθούν τα ίδια points, όμως αυτό δεν γίνεται λόγω θορύβου στην κάμερα. Έτσι, καταφέρνουμε να φιλτράρουμε λίγο την είσοδο μας. Η επιστροφή αυτής της function προχωράει σαν είσοδος του Kinect (όσο αφορά την RGB) για τον υπόλοιπο αλγόριθμο.

Στο fe_MainTools.cpp :

Εδώ γίνεται χρήση μέσα σε μια άλλη function (που θα περιγράψουμε αργότερα) που ψάχνει να βρει εάν τα points που μόλις μας έστειλε το Kinect υπάρχουν ήδη στην βάσει δεδομένων ή όχι. Αυτή η function χρησιμοποιείται αφού έχουμε κάνει ήδη το data association και ξέρουμε με ποια εικόνα στην βάση δεδομένων είναι ίδια αυτή που μόλις πήραμε από το Kinect. Αυτή η συνάρτηση απλά μας βρίσκει τα κοινά σημεία, έτσι ώστε να ξέρουμε τα ID των points που μόλις είδαμε και το distance

που έδωσε το BF Matcher.

```
void compare_images(Mat desc1, vector<KeyPoint> kpts1, Mat
desc2, vector<KeyPoint> kpts2, Mat &desc_final,
vector<KeyPoint> &kpts_final, vector<float>& dist_final,
bool& error)
{
	//cout<<endl<<"*********************IN:
compare_images******************"<<endl;
	// --Parameters
	const float inlier_threshold = 2.5f; // Distance
threshold to identify inliers
	const float nn_match_ratio = 0.8f;   // Nearest neighbor
matching ratio
	Mat desc_match;
	vector<KeyPoint> matched1, matched2;
	kpts_final.clear();
	desc_final.release();
	vector<Point2f> image1, image2;
	vector<float> dist_temp;

	// --Matching descriptor vectors using BFmatcher
	BFMatcher matcher(NORM_L1);
	vector< vector<DMatch> > nn_matches;
	matcher.knnMatch(desc1, desc2, nn_matches, 2);

	// --Keep the KeyPoints with the small distance
	for(size_t i = 0; i < nn_matches.size(); i++)
	{
		DMatch first = nn_matches[i][0];
		float dist1 = nn_matches[i][0].distance;
		float dist2 = nn_matches[i][1].distance;

		if(dist1 < nn_match_ratio * dist2)
		{
			matched1.push_back(kpts1[first.queryIdx]);
	// <-------
			matched2.push_back(kpts2[first.trainIdx]);

	desc_match.push_back(desc1.row(first.queryIdx)); // <---
----
			dist_temp.push_back(first.distance);  // <---
---

			image1.push_back( kpts1[ first.queryIdx ].pt
);
			image2.push_back( kpts2[ first.trainIdx ].pt
);
		}
	}

	// --Find homography matrix
	Mat homography = findHomography( image1, image2, RANSAC
);
```

```
	//cout<<"homography is:"<<endl<<homography<<endl<<endl;
	if(homography.empty()) {error = true; return;}

	// --Use homography
	for(unsigned i = 0; i < matched1.size(); i++)
	{
		Mat col = Mat::ones(3, 1, CV_64F);
		col.at<double>(0) = matched1[i].pt.x;
		col.at<double>(1) = matched1[i].pt.y;

		col = homography * col;
		col /= col.at<double>(2);
		double dist = sqrt( pow(col.at<double>(0) -
matched2[i].pt.x, 2) + pow(col.at<double>(1) -
matched2[i].pt.y, 2));

		if(dist < inlier_threshold)
		{
			kpts_final.push_back(matched1[i]);	// <---
----
			desc_final.push_back(desc_match.row(i));
	// <-------
			dist_final.push_back(dist_temp[i]);	// <---
----
		}
	}
}
```

Εδώ έχουμε μια από τις πιο σημαντικές συναρτήσεις μας. Της δίνουμε τα features και αυτή ελέγχει εάν στην βάση δεδομένων έχουμε ήδη αποθηκεύσει το συγκεκριμένο σύνολο από features. Τον έλεγχο τον κάνει χρησιμοποιώντας τον BF Matcher. Με μια γρήγορη σάρωση της εικόνας εισόδου με όλα τα στοιχεία στην βάση δεδομένων, ελέγχουμε ποιο από αυτά έχει το καλύτερο μέσο όρο distance (δηλαδή το μικρότερο). Άμα ο μέσος όρος είναι πάνω από μια τιμή που έχουμε δηλώσει θεωρούμε πως είναι καινούρια η εικόνα. Αλλιώς θεωρούμε ότι είναι παλιά και καλούμε την "compare_images()" συνάρτηση.

Εάν είναι καινούρια όμως, αρχικά συγκρίνει την είσοδο με την κοντινότερη εικόνα στην βάση δεδομένων (δηλαδή αυτή με το μικρότερο μέσο όρο distance). Στην συνέχεια βρίσκει τα λίγα κοινά σημεία που έχουν οι δύο εικόνες. Τέλος, τα σημεία που δεν είναι κοινά, τα επιστρέφει με ένα καινούριο ID.

```
void search_in_database(Mat desc_final, vector<KeyPoint>&
kpts_final, vector<float>& dist_final, int& pointID_last,
vector<bool>& old, bool& error)
{
```

```
//    cout<<endl<<"*********************IN:
search_in_database*****************"<<endl;
      vector<Mat> desc_database;
      vector<vector<KeyPoint> > kpts_database;
      int mean_distance = 0;
      int best_distance = 10000;
      int best_id = 0;
      vector<DMatch> matches;
      Mat desc_out;
      vector<KeyPoint> kpts_out;

      load_from_database(desc_database, kpts_database,
"/home/jim/ROS/catkin_ws/src/otacon/xml/database_1.xml");

      BFMatcher matcher(NORM_L1);//NORM_HAMMING
      for(int i=0;i<desc_database.size();i++)
      {
            if(desc_final.rows <=
desc_database[i].rows){matcher.match(desc_final,
desc_database[i], matches); }
            else {matcher.match(desc_database[i], desc_final,
matches); }

            for(int j=0;j<matches.size();j++) {mean_distance
+= matches[j].distance; }  // --?! mipos na kano kai *2 to
kathe "distance"

            mean_distance /= matches.size();
            if(mean_distance < best_distance) {best_distance =
mean_distance; best_id = i; }

            mean_distance = 0;
      }

      cout<<"Best image is No "<<best_id+1<<". The best mean
distance is "<<best_distance<<endl;
      if (best_distance<1500)     // No need for new landmark
      {
            cout<<"Old image. It was found in the
database."<<endl;
            compare_images(desc_database[best_id],
kpts_database[best_id], desc_final, kpts_final, desc_out,
kpts_out, dist_final, error);
            kpts_final = kpts_out;
            desc_final = desc_out.clone();
            for(int
j=0;j<kpts_out.size();j++){old.push_back(true);}
            if(error){cout<<"BIG ERROR!!!!!
HELP!!!!!!!!!!!!!!!!!!!!!!!!!!"<<endl<<endl; return;}
            cout<<" No error"<<endl;
      }
      else
      {
            cout<<"NEW LANDMARK.
YEAAAAAAAAAAAAAAAAAAAAAAAAAY!"<<endl;
            new_landmark(desc_final, kpts_final,
```

```
desc_database[best_id], kpts_database[best_id], dist_final,
pointID_last, old, error);
            if(error){cout<<"BIG ERROR!!!!!
HELP!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"<<endl<<endl; return;}
            cout<<" No error"<<endl;
            add_to_database(desc_final, kpts_final,
pointID_last,
"/home/jim/ROS/catkin_ws/src/otacon/xml/database_1.xml");
      }
//    cout<<endl<<"**********************OUT:
search_in_database******************"<<endl;
}
```

Εδώ δίνουμε δύο vectors. Το ένα είναι τα keypoints και το άλλο είναι οι αποστάσεις που είχαν αυτά τα keypoints μαζί με τα κοντινότερα σε αυτά keypoints στην βάση δεδομένων. Το αποτέλεσμα είναι ένα vector<KeyPoint> που ξεκινάει με τα points που ήταν πολύ κοντά στην σύγκριση και καταλήγει σε αυτό το point που ήταν το πιο μακριά.

Ο λόγος που το κάνουμε αυτό είναι γιατί συνήθως τα points που βρίσκει το front_end είναι πολλά να τα επεξεργαστεί το SLAM και έτσι αναγκαζόμαστε να στείλουμε μόνο μερικά από αυτά. Από τα παλιά points που θα ξαναστείλουμε στο SLAM, θέλουμε να έχουν όσο το λιγότερο δυνατόν distance (δηλαδή να είναι πολύ αξιόπιστα). Γι' αυτό τα βάζουμε στην κορυφή της λίστας (vector) ώστε να είναι εύκολο να ξέρουμε ποια να στείλουμε.

```
vector<KeyPoint> ascending_kpts(vector<KeyPoint> kpts,
vector<float> dist_final)
{
      KeyPoint kpts_temp;
      float dist_temp;
      //vector<KeyPoint> kpts_out;

      for(int i=1;i<kpts.size();i++)
      {

            for(int j=0; j<kpts.size()-i;j++)
            {
                  if(dist_final[j] > dist_final[j+1])
                  {
                        dist_temp = dist_final[j];
                        dist_final[j] = dist_final[j+1];
                        dist_final[j+1] = dist_temp;

                        kpts_temp = kpts[j];
```

```
                        kpts[j] = kpts[j+1];
                        kpts[j+1] = kpts_temp;

                  }
            }
      }
      return kpts;
}
```

Εδώ απλά διαγράφουμε κάποια διπλά points που μπορεί να υπάρχουν

```
vector<KeyPoint> remove_copies(vector<KeyPoint> kpts_in)
{
      vector<int> id;
      for(int i=0;i<kpts_in.size();i++)
      {
            id.push_back(kpts_in[i].class_id);
      }

      vector<KeyPoint> kpts_out;
      for(int i=0;i<kpts_in.size();i++)
      {
            int temp_id = kpts_in[i].class_id;
            if( find( id.begin()+i+1, id.end(),
kpts_in[i].class_id )==id.end() )
            {
                  kpts_out.push_back(kpts_in[i]);
            }
      }
      return kpts_out;
}
```

Γίνεται η σύγκριση ανάμεσα στα points που έχουμε μόλις πάρει (και περάσει από όλα τα φιλτραρίσματα) με αυτά που το SLAM δηλώνει ότι έχει παραλάβει.

Υπάρχει ένα xml αρχείο (θα το δούμε στο τέλος του κεφαλαίου αυτού) το οποίο αποθηκεύει το SLAM.py των αριθμών και το ID των landmarks που έχει ήδη ενσωματώσει στον αλγόριθμο του. Έτσι το front_end ξέρει τι να στείλει στο SLAM.py

```
vector<bool> find_old(vector<KeyPoint> kpts)
{

      int num_id,index=0, temp;
      ostringstream convert;
      string str_num;
```

```
	vector<int> temp_id;
	vector<bool> old;

	FileStorage 
fs("/home/jim/ROS/catkin_ws/src/otacon/xml/id.xml", 
FileStorage::READ);
	fs["num_of_ids"] >> num_id;

	for(int i=0;i<num_id;i++)
	{cout<<"ERROR"<<endl;
		index = i+1;
		convert << index;
		str_num = convert.str();

		str_num = "id_" + str_num;
		fs[str_num] >> temp;
		temp_id.push_back(temp);
	}
	vector<int> id;
	for(int i=0;i<kpts.size();i++)
	{
		id.push_back(kpts[i].class_id);
	}

	for(int j=0;j<id.size();j++)
	{
		if( find(temp_id.begin(),temp_id.end(),id[j]) != 
temp_id.end() )
		{
			old.push_back(true);
		}
		else
		{
			old.push_back(false);
		}
	}

	return old;
}
```

Μια ακόμη πολύ σημαντική συνάρτηση. Εδώ είναι το τελευταίο στάδιο του front_end αλγορίθμου. Παίρνει τα points, βρίσκει τις 3D συντεταγμένες και γεμίζει ένα custom ROS message “feature_coordinates_array.msg” με όλες τις πληροφορίες που χρειάζεται να στείλει στο SLAM.py . Το μισό ROS message το γεμίζει με features που έχει ήδη το SLAM και το άλλο μισό με καινούρια features. Σε περίπτωση που δεν υπάρχουν νέα features, απλά τον γεμίζει όλο με παλιά features. Τέλος, η συνάρτηση επιστρέφει ένα feature_coordinates_array, το οποίο και στέλνεται από το front_end_node στο ανάλογο ROS topic.

```
void get_3D(vector<KeyPoint> kpts_final, vector<bool> old,
otacon::feature_coordinates_array& front_end_msg,
pcl::PointCloud<pcl::PointXYZRGB> cloud_f1, int max_points)
{
	cout<<endl<<"***********************IN: get_3D
function******************"<<endl;
	Mat temp_desc;
	geometry_msgs::Point temp_xyz;
	vector<Point> kpts_final_int =
make_points_int(kpts_final);
	otacon::feature_coordinates msg_tmp;
	int current_point = 0;
	int old_limit = max_points/2, new_limit = max_points/2;
	int old_index = 0, new_index = 0;

	for(int i=0;i<kpts_final_int.size();i++)
	{
		pcl::PointXYZRGB point =
cloud_f1.at(kpts_final_int[i].x, kpts_final_int[i].y);
		if(point.x<10.0 && point.y<10.0 && point.z<10.0)
		{
			if(old[i]==true && old_index < old_limit)
			{
				old_index++;
				msg_tmp.x = point.z;
				msg_tmp.y = point.x;
				msg_tmp.z = point.y;
				msg_tmp.id = kpts_final[i].class_id;
				msg_tmp.old = 1;
				front_end_msg.data.push_back(msg_tmp);
			}
			else if(old[i]==false && new_index <
new_limit)
			{
				new_index++;
				msg_tmp.x = point.z;
				msg_tmp.y = point.x;
				msg_tmp.z = point.y;
				msg_tmp.id = kpts_final[i].class_id;
				msg_tmp.old = 0;
				front_end_msg.data.push_back(msg_tmp);
			}
		}

	}
	//temp_desc.copyTo(desc_final);
	//cout<<"desc_final size is -->
"<<desc_final.rows<<endl;
	cout<<"points found with 3D points -->
"<<front_end_msg.data.size()<<endl;
	cout<<endl<<"***********************OUT: get_3D
function******************"<<endl;

}
```

### 6.2.4. Ανάλυση του fe_ImageTools.cpp

Τώρα θα εξετάσουμε μια-μια τις function του fe_ImageTools.cpp:

Η παράμετρος CV::KeyPoint έχει ένα πεδίο που είναι το ID. Δυστυχώς υπάρχει ένα bug στο xml reader του OpenCV. Όταν φορτώνει τα αποθηκευμένα KeyPoint, στο πεδίο του ID μπαίνουν "σκουπίδια" (όλα τα υπόλοιπα πεδία φορτώνουν κανονικά). Για αυτόν τον λόγο, με την χρήση της βιβλιοθήκης TinyXML, διαβάζουμε με διαφορετικό τρόπο τα IDs. Έτσι, φορτώνουμε το vector<KeyPoints> κανονικά από το XML (με το εργαλείο που δίνει το OpenCV) εκτός του ID πεδίου και φορτώνουμε έτσι τα IDs από το XML.

```
void get_class_id(string location, string kpts, string desc,
vector<int>& class_id)
{
	// Load the XML file
	TiXmlDocument doc;
	doc.LoadFile( location );
	TiXmlElement* root = doc.FirstChildElement();
	//--
	TiXmlElement* titleElement = root->FirstChildElement(
kpts );
	TiXmlText* textNode = titleElement->FirstChild()-
>ToText();
	string title = textNode->Value();
	//--
	TiXmlElement* sizeElement = root->FirstChildElement(
desc );
	TiXmlText* KeyPoint_size = sizeElement-
>FirstChild("rows")->FirstChild()->ToText();
	string size_str = KeyPoint_size->Value();
	istringstream rr(size_str);
	int size;
	rr >> size;


	size_t pos = 0;
	size_t found = 0;

	for(int i=0;i<size;i++)
	{
		// Find the position of the class_id in the string
		bool OK = false;

		do
		{
			found = title.find(" ", pos);
			if( isspace(title[found]) &&
isspace(title[found+2]) ) { OK = true; }
```

```
                else{pos = found+1;}


            }while(OK==false);


            found = found + 2;
            // Add all characters that compose the hole
class_id numbers
            string number_str;
            number_str += title[found];
            found++;
            while(title[found]!=' ')
            {
                  number_str += title[found];
                  found++;
            }


            // Make the string number to a integer number
            istringstream ss(number_str);
            int number;
            ss >> number;
            class_id.push_back(number);

            pos = pos + 60;  //New line at around 119

      }
}
```

Του δίνουμε τα descriptors και τα keypoints από ένα frame, για να τα αποθηκεύσει στην βάση δεδομένων.

```
void add_to_database(Mat desc, vector<KeyPoint> kpts, int&
pointID_last, string location)
{
//cout<<"add_to_database -- 1"<<endl;
      ostringstream convert;
      int num_img=0, index=0;
      Mat temp_desc;
      vector<KeyPoint> temp_kpts;
      string str_num, str_desc, str_kpts;
      vector<string> str_desc_name, str_kpts_name;
      vector<Mat> all_desc;
      vector<vector<KeyPoint> > all_kpts;
//cout<<"add_to_database -- 2"<<endl;
      FileStorage fs(location, FileStorage::READ);
      fs["number_of_images"] >> num_img;
      for(int i=0;i<num_img;i++)
      {
            index = i+1;
```

```
            convert << index;
            str_num = convert.str();

            str_desc = "Descriptor_" + str_num;
            str_kpts = "KeyPoints_" + str_num;
            str_desc_name.push_back(str_desc);
            str_kpts_name.push_back(str_kpts);
            fs[str_desc] >> temp_desc;
            fs[str_kpts] >> temp_kpts;

            // Load class_id of KeyPoints
            vector<int> class_id;
            get_class_id(location, str_kpts, str_desc,
class_id);

            for(int j=0;j<temp_kpts.size();j++){
temp_kpts[j].class_id = class_id[j]; }

            all_desc.push_back(temp_desc);
            all_kpts.push_back(temp_kpts);

            convert.str("");
            convert.clear();
            str_num.clear();
      }
      fs.release();
      num_img++;
      convert << num_img;
      str_num = convert.str();

      str_desc = "Descriptor_" + str_num;
      str_kpts = "KeyPoints_" + str_num;
      str_desc_name.push_back(str_desc);
      str_kpts_name.push_back(str_kpts);
      all_desc.push_back(desc);
      all_kpts.push_back(kpts);
      FileStorage fs2(location, FileStorage::WRITE);
      fs2 << "pointID_last" << pointID_last;
      fs2 << "number_of_images" << num_img;
      for(int i=0;i<num_img;i++)
      {
            fs2 << str_desc_name[i] << all_desc[i];
            fs2 << str_kpts_name[i] << all_kpts[i];
      }
      fs2.release();
}
```

Εδώ είναι που φορτώνουμε features από την βάση δεδομένων με σκοπό να τα συγκρίνουμε με τα features που μόλις πήρε το Kinect

```
void load_from_database(vector<Mat>& all_desc,
vector<vector<KeyPoint> >& all_kpts, string location)
{
```

```
	ostringstream convert;
	int num_img=0, index=0;
	Mat temp_desc;
	vector<KeyPoint> temp_kpts;
	string str_num, str_desc, str_kpts;
	//vector<string> str_desc_name, str_kpts_name;
	all_desc.clear();
	all_kpts.clear();

	FileStorage fs(location, FileStorage::READ);
	fs["number_of_images"] >> num_img;
	for(int i=0;i<num_img;i++)
	{
		index = i+1;
		convert << index;
		str_num = convert.str();

		str_desc = "Descriptor_" + str_num;
		str_kpts = "KeyPoints_" + str_num;
	//str_desc_name.push_back(str_desc);
	//str_kpts_name.push_back(str_kpts);
		fs[str_desc] >> temp_desc;
		fs[str_kpts] >> temp_kpts;

		//temp_desc = fs[str_desc];
		//temp_kpts = fs[str_kpts];

		// Load class_id of KeyPoints
		vector<int> class_id;
		get_class_id(location, str_kpts, str_desc,
class_id);

	//cout<<endl<<endl<<temp_kpts.size()<<endl<<class_id.siz
e()<<endl<<endl;
		for(int j=0;j<temp_kpts.size();j++){
temp_kpts[j].class_id = class_id[j]; }

		all_desc.push_back(temp_desc);
		all_kpts.push_back(temp_kpts);

		convert.str("");
		convert.clear();
		str_num.clear();
	}
	fs.release();
}
```

Τα KeyPoints που έχουν τις συντεταγμένες των σημείων που βρήκαμε στην εικόνα είναι δεκαδικοί. Στο PointCloud (που παίρνουμε από το Kinect) οι συντεταγμένες του πίνακα είναι ακέραιες. Το κάθε σημείο του PointCloud έχει, ανάμεσα και σε άλλες πληροφορίες, την απόσταση x, y και z σε σχέση με το Kinect.

Γι' αυτό το λόγο εδώ στρογγυλοποιούμε τους δεκαδικούς αριθμούς των KeyPoints και τους κάνουμε ακέραιους με σκοπό να βρούμε στο PointCloud τις συντεταγμένες βάθους.

```
vector<Point> make_points_int(vector<KeyPoint> kpts)
{
      // Here we convert the points found from Float to Int,
so we can find there corresponding XYZ values.
      vector<Point> kpts_int;
      for(int j=0;j<=kpts.size()-1;j++)
      {

            float xf=0, yf=0;
            xf = kpts[j].pt.x + 0.5;
            yf = kpts[j].pt.y + 0.5;

            kpts_int.push_back( Point( (int)xf,(int)yf ) );
      }
      return(kpts_int);
}
```

Όταν έχουμε νέο feature, το στέλνουμε εδώ μαζί με την κοντινότερη εικόνα από features που βρέθηκε στο database. Ο σκοπός είναι να δούμε ποια σημεία είναι ίδια και ποια είναι καινούρια.

```
void new_landmark(Mat desc_new, vector<KeyPoint>& kpts_new,
Mat desc_old, vector<KeyPoint> kpts_old, vector<float>&
dist_final, int& pointID_last, vector<bool>& old, bool&
error)
{
      //cout<<endl<<"**********************IN:
new_landmark******************"<<endl;
      // --Parameters
      const float inlier_threshold = 2.5f; // Distance
threshold to identify inliers
      const float nn_match_ratio = 0.8f;   // Nearest neighbor
matching ratio
      vector<KeyPoint> matched1, matched2;
      vector<Point2f> image1, image2;
      vector<int> new_index, old_index;
      vector<float> dist_final_n (kpts_new.size(),0);
      vector<float> dist_temp;

      // --Matching descriptor vectors using BFmatcher
      BFMatcher matcher(NORM_L1);
      vector< vector<DMatch> > nn_matches;
      matcher.knnMatch(desc_new, desc_old, nn_matches, 2);
```

```
	// --Keep the KeyPoints with the small distance
	for(size_t i = 0; i < nn_matches.size(); i++)
	{
		DMatch first = nn_matches[i][0];
		float dist1 = nn_matches[i][0].distance;
		float dist2 = nn_matches[i][1].distance;

		if(dist1 < nn_match_ratio * dist2)
		{
			matched1.push_back(kpts_new[first.queryIdx]);

			matched2.push_back(kpts_old[first.trainIdx]);

			new_index.push_back(first.queryIdx);  // <---
----
			old_index.push_back(first.trainIdx);  // <---
----
			dist_temp.push_back(first.distance);  // <---
----

			image1.push_back( kpts_new[ first.queryIdx
].pt );
			image2.push_back( kpts_old[ first.trainIdx
].pt );
		}
	}

	// --Find homography matrix
	Mat homography = findHomography( image1, image2, RANSAC
);
	cout<<"homography is:"<<endl<<homography<<endl<<endl;
	if(homography.empty()){error = true; return;}
	//cout<<"homography is:"<<endl<<homography<<endl<<endl;

	// --Use homography

	for(unsigned i = 0; i < matched1.size(); i++)
	{
		Mat col = Mat::ones(3, 1, CV_64F);
		col.at<double>(0) = matched1[i].pt.x;
		col.at<double>(1) = matched1[i].pt.y;

		col = homography * col;
		col /= col.at<double>(2);
		double dist = sqrt( pow(col.at<double>(0) -
matched2[i].pt.x, 2) + pow(col.at<double>(1) -
matched2[i].pt.y, 2));

		if(dist < inlier_threshold)
		{
			kpts_new[new_index[i]].class_id =
kpts_old[old_index[i]].class_id; // <-------
			dist_final_n[new_index[i]] = dist_temp[i];
		}
	}
```

```
	for(int i=0;i<kpts_new.size();i++)
	{
//cout<<"Class ID for point "<<i<<" - BEFORE - is -->
"<<kpts_new[i].class_id<<endl;
		if(pointID_last < kpts_new[i].class_id)
		{
			pointID_last++;
			kpts_new[i].class_id = pointID_last;
			old.push_back(false);
			//dist_final[i] = 0;
		}
		else
		{
			old.push_back(true);
		}

//cout<<"Class ID for point "<<i<<" - AFTER - is -->
"<<kpts_new[i].class_id<<endl;
	}
	dist_final = dist_final_n;
	//cout<<endl<<"*********************OUT:
new_landmark*****************"<<endl;
}//------------------------
```

## 6.3. SLAM node

### 6.3.1. Γενικές πληροφορίες

Αυτό το node δέχεται δεδομένα από το front_end και εφαρμόζει το FastSLAM που έχουμε περιγράψει έως τώρα. Σε γενικές γραμμές είναι απλή εφαρμογή του αλγορίθμου στον Πίνακα 6.3.1

Εδώ έχουμε το SLAM.py και το SLAM_extratools.py. Η λογικά είναι παρόμοια με πριν. Το SLAM_extratools.py προσφέρει διάφορα εργαλεία στο SLAM.py. Παρακάτω, θα δούμε αρχικά το SLAM.py. Σε αυτόν το κώδικα έχουμε φτιάξει ένα αντικείμενο, δύο συναρτήσεις και τον κυρίως κώδικα μας. Ας αναλύσουμε το κάθε ένα ξεχωριστά.

### 6.3.2. Ανάλυση του SLAM.py

Στην αρχή έχουμε όλα τα imports, τα οποία χωρίσαμε σε 3 κομμάτια. Το πρώτο είναι για πράγματα σχετικά με το ROS, το δεύτερο για modules και libraries από το Scipy και Sympy, και τέλος έχουμε τα modules που είναι μέσα στο core της python.

```
#!/usr/bin/env python
# ROS imports
import rospy
from otacon.msg import feature_coordinates_array # import
feature_coordinates
from SLAM_extratools import *    # this is my own custome
module
from otacon.srv import *
import otacon.msg as ms

# Scipy and Sympy imports (mathematical imports)
from math import sqrt
import sympy as s
import numpy as np
import numdifftools as ndf
import matplotlib
from sympy.matrices import Matrix
from sympy.interactive.printing import init_printing
#from numpy import *
from sympy import *

# Other Python imports
import time
import copy
import math as m
```

Παρακάτω έχουμε μια κλάση που χαρακτηρίζει ένα particle. Με το που δημιουργούμε το αντικείμενο ορίζουμε την θέση του particle (έχουμε ως default μηδέν στα πάντα), έναν κενό χάρτη, το importance weight σε 1, έναν πίνακα Q που είναι το σφάλμα που χρειαζόμαστε για το EKF update και έναν μοναδιαίο πίνακα τριών θέσεων.

Στο αντικείμενο έχουμε μερικές συναρτήσεις:

→new_landmark – Αυτήν την συνάρτηση καλούμε για κάθε νέο landmark που δέχεται αυτό το particle, ώστε να το ενσωματώσει στον χάρτη του.

→update_landmark – Για όσα landmarks έχουμε ήδη δει, κάνουμε το EKF update εδώ.

→move – Εφαρμόζουμε το motion model στο particle.

```
class particles:
      def __init__(self, _pose=[[0],[0],[0],[0],[0],[0] ]):
            self.Pose = Matrix(_pose) # X - Y - Z - roll -
pitch - yaw
            self.Map = []  # id, mean, sigma
```

```
            self.Weight = 1.0
            self.Q = Matrix([ [1,0,0],[0,1,0],[0,0,1] ])
            self.I = Matrix([ [1,0,0],[0,1,0],[0,0,1] ])


      def new_landmark(self, feature):
            MapMean = L2G(self.Pose, [ feature.x, feature.y,
feature.z ])
            H = Hj(self.Pose, MapMean)
            MapSigma = H.T * self.Q.inv() * H
            MapSigma = MapSigma.inv()
            MapID = feature.id
            Map_element = xartis(MapID, MapMean, MapSigma)
            self.Map.append(Map_element)

      def update_landmark(self, map_item, feature):
            Zo = z_obs(feature.x, feature.y, feature.z) #
measurement observation
            Map_index = self.Map.index(map_item)
            Lmean = self.Map[Map_index].mean
            Lsigma = self.Map[Map_index].sigma

            H = Hj(self.Pose, Lmean)
            Zp = h(self.Pose, Lmean)    # measurement
prediction
            Q = H * Lsigma * H.T + self.Q
            K = Lsigma * H.T * Q.inv()
            new_Lmean = Lmean + K * (Zo - Zp)
            new_Lsigma = (self.I - K * H) * Lsigma
            self.Weight = self.Weight * weight(Zo, Zp, Q)

            self.Map[Map_index].mean = new_Lmean
            self.Map[Map_index].sigma = new_Lsigma

      def move(self, action):     # action = [distance ,angle]
            self.Pose = motion_model(self.Pose, action, 1.0)
```

Αυτή η συνάρτηση μας επιστρέφει μια μέτρηση από το front_end. Κάνει subscribe στο topic που δημιουργεί το front_end για να στέλνει τα δεδομένα του μέσω του ROS δικτύου. Μετά απλά περιμένει μέχρι να σταλούν τα δεδομένα, ώστε να τα λάβουμε μέσω του callback(). Στην τέταρτη γραμμή κώδικα, του ορίζουμε ότι όποτε έρθει νέο ROS message από το front_end θα πρέπει να τρέξει η συνάρτηση callback().

```
def get_measurement():
      global sub
      global feature_data
      sub = rospy.Subscriber("feature_coordinates",
feature_coordinates_array, callback)
      feature_data = []
      while(not feature_data):
```

```
            print("waiting for new features")
            time.sleep(0.3)
        save_ids(feature_data)
        return;
```

Αυτή η συνάρτηση καλείται κάθε φορά που έρχεται καινούριο message από το topic "feature_coordinates". Απλώς ανοίγει το ROS message ώστε να πάρει τα δεδομένα του σε πιο πρακτική μορφή.

```
def callback(features):
        global sub
        global feature_data
        sub.unregister()
        k = len(features.data)
        for i in range(k):
            feature_data.append(features.data[i])
```

Ο αλγόριθμος που εφαρμόζεται πιο κάτω είναι ως εξής:

1. Αρχικοποίηση:
    a. Παίρνω την τωρινή μέτρηση από το front_end
    b. Δημιουργώ έναν αριθμό particles. Στην προκειμένη περίπτωση θα είναι 100
    c. Όποια landmarks βρήκα, τα ενσωματώνω στον αλγόριθμο ως νέα landmarks
2. Στέλνω εντολή κίνησης στο ρομπότ
3. Παίρνω μια μέτρηση
4. Σε κάθε particle:
    a. Εφαρμόζω το motion model
    b. Από τα landmarks που πήρα από το front_end, εντοπίζω ποια έχω ήδη ενσωματώσει στον αλγόριθμο και ποια όχι. Αυτό γίνεται χρησιμοποιώντας το μοναδικό ID που έχει το καθένα. Για καινούρια landmarks καλώ την συνάρτηση new_landmark(), ενώ για παλιά καλώ την συνάρτηση update_landmark().
    c. Κάνω resampling
5. Loop back to step 2.

```
# Initialization =========================
rospy.init_node('SLAM', anonymous=True)
sub = 0
feature_data = []
p=[]


# Get measurement ====================
get_measurement()
print("feature are ", len(feature_data) )

# Create first particles ========================
for i in range(0,100):
	p.append(particles())
	for j in range(len(feature_data)):
		p[i].new_landmark(feature_data[j])

# Main Loop ============================
while(1):
	get_measurement()
	print("feature are ", len(feature_data) )
	for i in range(0,100):
		p[i].Weight = 1.0
		p[i].move([0.1,0])
		for j in range(len(feature_data)):
			is_new_f = [x for x in p[i].Map if x.id ==
feature_data[j].id]

			if(not is_new_f):	#new features
				p[i].new_landmark(feature_data[j])
			else:	#old features

	p[i].update_landmark(is_new_f[0],feature_data[j])#send
also sensor observation
	p=resampling(p)


delete_ids()
rospy.spin()
```

### 6.3.3. Ανάλυση του SLAM_extretools.py

Ας δούμε το SLAM_extretools.py τώρα

Διάφορες βιβλιοθήκες που κάνουμε εισαγωγή για τις παρακάτω συναρτήσεις.

```
#!/usr/bin/env python
import numpy as np
import math as m
import sympy as s
from sympy.matrices import Matrix
```

```
from sympy.interactive.printing import init_printing

from random import *
import xml.etree.ElementTree as ET

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from copy import deepcopy
```

Αυτή η κλάση δημιουργεί τα αντικείμενα για το Map στο particle class. Κάθε ένα landmark είναι τύπου xartis. Κάθε landmark έχει το μοναδικό του id, τη μέση τιμή του (δηλαδή τις συντεταγμένες του) και ένα πίνακα διακύμανσης που δηλώνει πόσο σίγουρα ξέρουμε την τοποθεσία του landmark.

```
class xartis:
	def __init__(self, _id, _mean, _sigma):
		self.id = _id
		self.mean = Matrix(_mean)  #3x1
		self.sigma = Matrix(_sigma)  #3x3
```

Εφαρμογή του vector motion model. Του δίνουμε την θέση του particle και την εντολή κίνησης (θυμίζουμε ότι η εντολή αποτελείται από ταχύτητα και γωνιακή ταχύτητα).

```
def motion_model(pose, action, Dt):
	(x,y,z,roll,pitch,yaw) = 
[pose[0],pose[1],pose[2],pose[3],pose[4],pose[5]]
	(u,w) = action
	(a1,a2,a3,a4,a5,a6) = [0.5,1,0.5,1,0.5,1]
	ue = u + sample(( a1*abs(u) ) + ( a2*abs(w) ))
	we = w + sample(( a3*abs(u) ) + ( a4*abs(w) ))
	ge = sample(( a5*abs(u) ) + ( a6*abs(w) ))
	R = ue/we
	x2 = x -(R*m.sin(yaw)) + (R*m.sin(yaw+(we*Dt)))
	y2 = y +(R*m.cos(yaw)) -(R*m.cos(yaw+(we*Dt)))
	yaw2 = yaw + (we*Dt) + (ge*Dt)

	pose2 = Matrix([[x2],[y2],[0],[0],[0],[yaw2]])
	return(pose2)
```

Μια συνάρτηση που προσεγγίζει μια Gaussian. Το b είναι η τυπική απόκλιση και μας επιστρέφει μια τυχαία τιμή μέσα σε αυτήν την Gaussian.

```
def sample(b):   # b = variance
      z = []
      for i in range(12):
            z.append(uniform(-1,1))
      z = sum(z)*(b/6)
      return z
```

Μεταφορά όλων των points από local frame (δηλαδή το Kinect) σε global.

```
def L2G(Robot, point):
      point.append(1)
      Lpoint = Matrix( point )
      row1 = [s.cos(Robot[5]) , -s.sin(Robot[5]) , 0 , 
Robot[0]]
      row2 = [s.sin(Robot[5]) , s.cos(Robot[5]) , 0 , Robot[1]]
      row3 = [0 , 0 , 1 , Robot[2]]
      row4 = [0 , 0 , 0 , 1]
      Transformation = Matrix([ row1 , row2 , row3 , row4 ])
      Gpoint = Transformation*Lpoint
      Gpoint.row_del(3)
      return Gpoint
```

Εφαρμογή του Jacobian του motion model. Αυτή η συνάρτηση χρησιμοποιείται για την γραμμικοποίηση που χρειάζεται το EKF.

```
def Hj(pose, MapMean):
      x = pose[0]
      y = pose[1]
      z = pose[2]
      row = pose[3]
      pitch = pose[4]
      yaw = pose[5]
      xl = MapMean[0]
      yl = MapMean[1]
      zl = MapMean[2]

      dx = xl-x
      dy = yl-y
      dz = zl-z

      a1 = dx/m.sqrt(dx**2 + dy**2 + dz**2)
      a2 = dy/m.sqrt(dx**2 + dy**2 + dz**2)
      a3 = dz/m.sqrt(dx**2 + dy**2 + dz**2)
      b1 = -dy/(dx**2 + dy**2)
      b2 = dx/(dx**2 + dy**2)
```

```
    b3 = 0
    c1 = -dz/(dx**2 + dz**2)
    c2 = 0
    c3 = dx/(dx**2 + dz**2)
    H = Matrix([[a1,a2,a3],[b1,b2,b3],[c1,c2,c3]])
    return (H)
```

Εδώ υπολογίζουμε το expected prediction. Δηλαδή σε αυτήν τη συνάρτηση δίνουμε την θέση που έδωσε το motion model και την τοποθεσία που είχε ένα landmark στην προηγούμενη μέτρηση. Αυτό σαν αποτέλεσμα μας δίνει που υποθέτουμε ότι θα έπρεπε να μας πει ο αισθητήρας ότι βρίσκεται το landmark.

```
def h(R_pose, L_pose):
    dx = L_pose[0] - R_pose[0]
    dy = L_pose[1] - R_pose[1]
    dz = L_pose[2] - R_pose[2]

    Range = m.sqrt( dx**2 + dy**2 + dz**2 )
    Bearing1 = m.atan2(dy,dx) - R_pose[5]
    Bearing2 = m.atan2(dz,dx) - R_pose[4]
    z_exp = Matrix([ [Range],[Bearing1],[Bearing2] ])
    return(z_exp)
```

Εδώ υπολογίζουμε την θέση που έχει το landmark βασισμένοι αποκλειστικά πάνω στην τωρινή μας μέτρηση. Δίνουμε το XYZ ενός landmark για να πάρουμε το observation μας ($z_t$).

```
def z_obs(x,y,z):
    Range=m.sqrt(x**2 + y**2 + z**2)

    k1=m.sqrt( x**2 + y**2 )
    Bearing1 = m.asin(y/k1)

    k2=m.sqrt( x**2 + z**2 )
    Bearing2 = m.asin(z/k2)

    z_obs = Matrix([ [Range],[Bearing1],[Bearing2] ])
    return (z_obs)
```

Υπολογισμός του importance weight για το particle

```
def weight(Zo, Zp, Q):
```

```
	Zd = Zo-Zp

	temp = -0.5*Zd.T * Q.inv() * Zd
	temp2 =  ((2*m.pi)**3/2)*m.sqrt(Q.det())
	w = ( float(temp2) ) * m.exp( float(temp[0]) )
	return(w)
```

Εφαρμογή του resampling. Ένα πολύ σημαντικό βήμα. Του δίνουμε την λίστα με όλα τα particles και μας επιστρέφει μια νέα λίστα με τα particles που "επιβιώσανε".

```
def resampling(p):
	temp_w=0
	for k in range(len(p)):
		temp_w = temp_w + p[k].Weight
	dis = temp_w/len(p)
	print("distance of U will be",dis,"!!!!!!!!!!!!")
	p_new = []
	r = random()*dis
	Wsum = p[0].Weight
	Windex = 0
	for j in range(len(p)):
		U = r + dis*j
		while U>Wsum:
			Windex = Windex + 1
			Wsum = Wsum + p[Windex].Weight
		p_new.append(deepcopy(p[Windex]))
		print ("I chose you particle", Windex, " with weight = ",
p[Windex].Weight)

	return(p_new)
```

Αποθήκευση σε ένα xml αρχείο των IDs από τα features που έχει ενσωματώσει το SLAM στον εαυτό του. Ταυτόχρονα, το front_end διαβάζει αυτό το αρχείο ώστε να ξέρει ποια landmarks είναι παλιά και ποια είναι καινούρια.

```
def save_ids(features):
	root =
ET.parse("/home/jim/Desktop/python_tests/id.xml").getroot()
	index = int(root[0].text)
	for i in range(len(features)):
		if(features.old=0):
			index = index + 1
			str_tag = "id_" + str(index)
			child = ET.Element(str_tag)
			child.text=str(features[i].id)
			root.append(child)
```

```
	ET.ElementTree(root).write("/home/jim/Desktop/python_tests/id.xml")
```

Όταν τελειώνει την λειτουργία του το SLAM, καλούμε αυτήν την εντολή ώστε να καθαρίσει το xml αρχείο που έχει τα ids των landmarks που έχει ενσωματώσει. Αυτό το κάνουμε διότι για την ώρα δεν αποθηκεύουμε τον χάρτη που έχει χτίσει το SLAM άρα το xml πρέπει να είναι άδειο.

```
def delete_ids():
	root = ET.Element('main')
	child = ET.Element("num_of_ids")
	child.text=str(0)
	root.append(child)
	ET.ElementTree(root).write("/home/jim/ROS/catkin_ws/src/otacon/xml/id.xml")
```

Του δίνουμε την λίστα με όλα τα particles και μας τα εμφανίζει σε ένα 3D plot, εργαλείο που το χρήσιμοποιούμε ώστε να παρακολουθούμε την διαδρομή των particles.

```
def plot_particles(p):
	poseX=[]
	poseY=[]
	poseZ=[]
	weight=[]
	test=[]
	for i in range(len(p)):
		poseX.append(p[i].Pose[0])
		poseY.append(p[i].Pose[1])
		poseZ.append(p[i].Pose[2])
		weight.append(p[i].Weight)
		test.append(250)

	fig = plt.figure()
	ax = fig.add_subplot(111, projection='3d')
	ax.scatter(poseX, poseY, poseZ)
	ax.set_xlabel('X Label')
	ax.set_ylabel('Y Label')
	ax.set_zlabel('Z Label')
	plt.show()
```

Του δίνουμε ένα particle αντικείμενο και εμφανίζει σε 3D την θέση του particle και την θέση των landmarks που έχει δει έως τώρα. Τα landmarks εμφανίζονται με ένα ημιδιαφανές κύκλο, που το μέγεθος του κύκλου αντιπροσωπεύει την αβεβαιότητα του landmark για το συγκεκριμένο particle. Όσο πιο μεγάλος ο κύκλος, τόσο μεγαλύτερη αβεβαιότητα υπάρχει, αντίστοιχα όσο μικρότερος είναι τόσο λιγότερη αβεβαιότητα υπάρχει.

```
def plot_map(p0):
	fig = plt.figure()
	ax = fig.add_subplot(111, projection='3d')
	ax.scatter(p0.Pose[0],p0.Pose[1],p0.Pose[2])

	xl = []
	yl = []
	zl = []
	r = []
	for j in range(len(p0.Map)):
		xl.append(float(p0.Map[j].mean[0]))
		yl.append(float(p0.Map[j].mean[1]))
		zl.append(float(p0.Map[j].mean[2]))
		temp_r = float( (p0.Map[j].sigma[0] + p0.Map[j].sigma[4] +
p0.Map[j].sigma[8])/3 )
		r.append(temp_r*100)

	plt.scatter(xl, yl, zs=zl, s=r, c='r', alpha=0.5)
	ax.set_xlabel('X Label')
	ax.set_ylabel('Y Label')
	ax.set_zlabel('Z Label')
	plt.show()
```

# Κεφάλαιο 7. - Αποτελέσματα

Σε αυτό το κεφάλαιο θα δούμε πως επεξεργαζόμαστε τα δεδομένα που παίρνουμε και θα εμφανίσουμε διαδοχικά πως χτίζονται οι χάρτες και πως κινούνται τα particles.

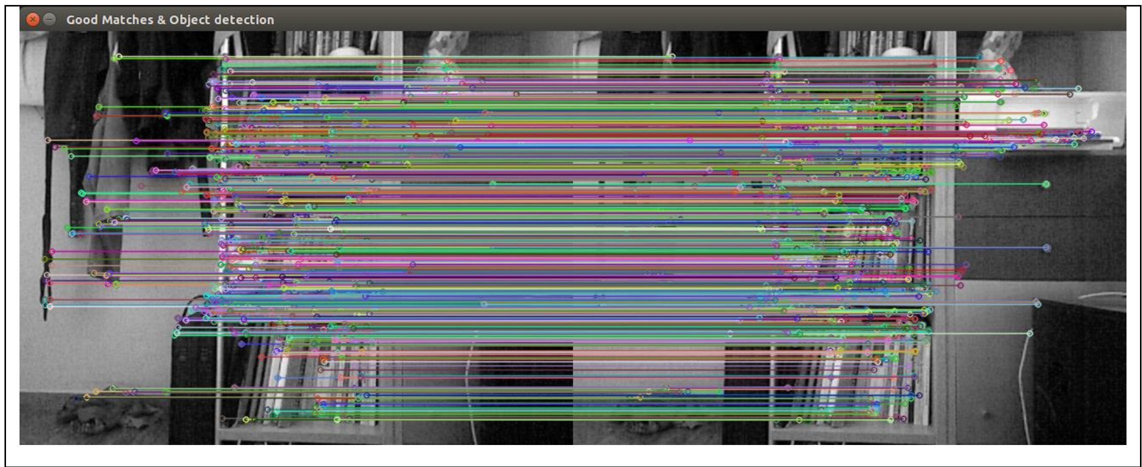
## 7.1. Πως το front_end επεξεργάζεται τα Kinect δεδομένα

Θα ασχοληθούμε με τα δύο δεδομένα που στέλνει το Kinect

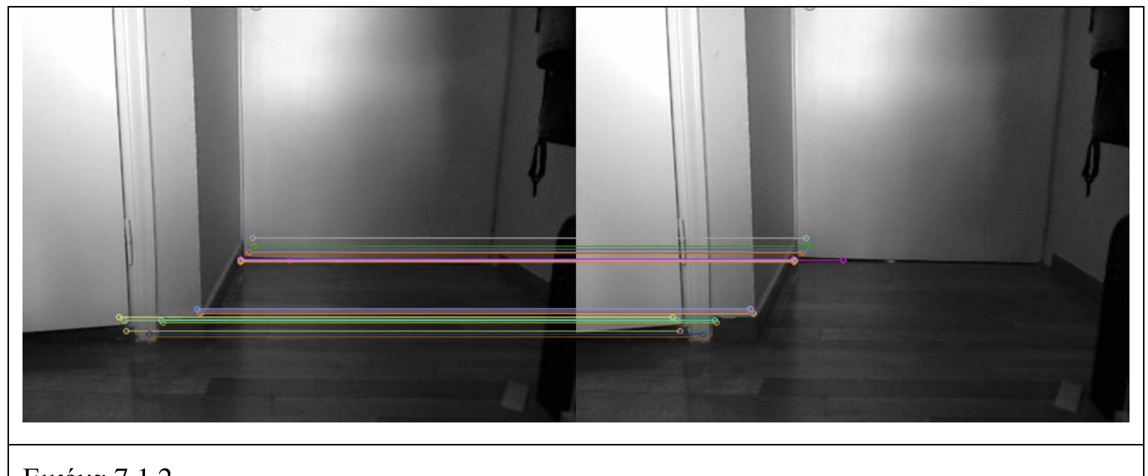
- RGB εικόνα – Κάθε pixel ένα χρώμα
- Depth εικόνα – Κάθε pixel έχει μια τιμή βάθους

Πήραμε μετρήσεις σε δύο περιβάλλοντα. Στο ένα περιβάλλον υπάρχουν πάρα πολλά μοναδικά χαρακτηριστικά ενώ το άλλο είναι σχεδόν άδειο από οποιαδήποτε σημεία.

Στις RGB εικόνες αναφέραμε ότι παίρνουμε δύο frames που απέχουν μόνο λίγα frames μεταξύ τους. Στην συνέχεια βρίσκουμε τα ίδια σημεία. Αυτό μπορούμε να το δούμε να γίνεται στην εικόνα 7.1-1 και εικόνα 7.1-2.




Εικόνα 7.1.1



Εικόνα 7.1.2

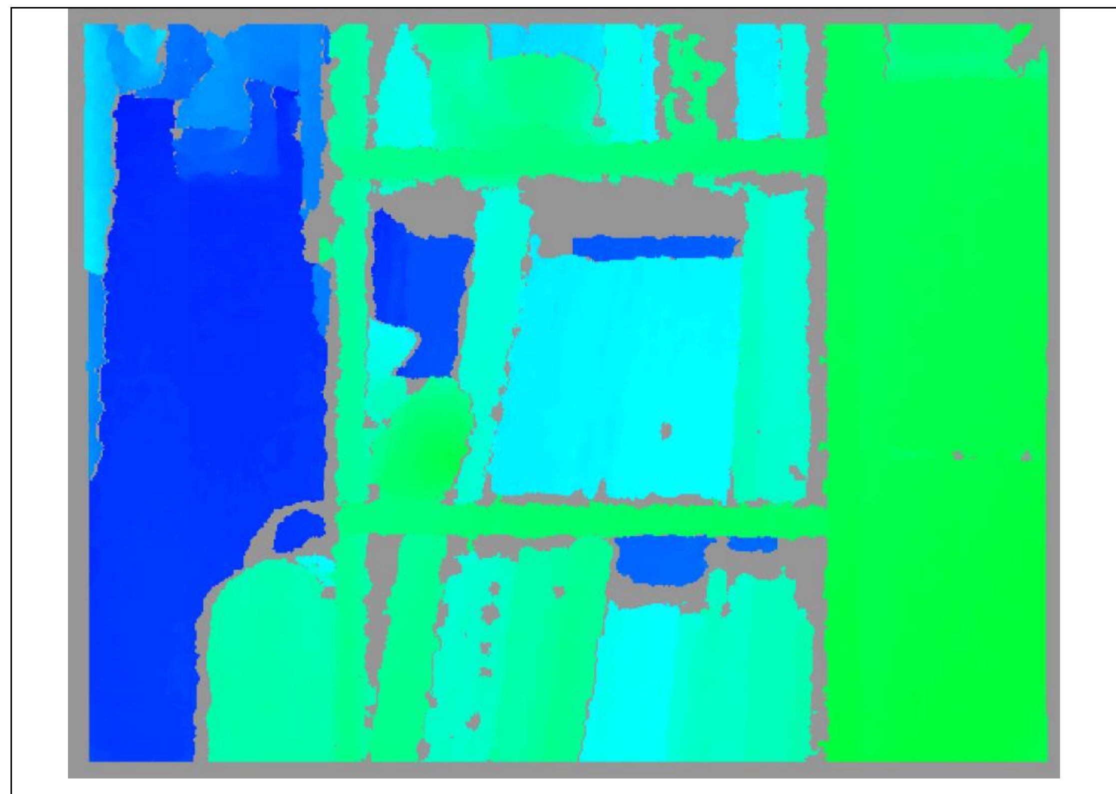
Εμείς κρατάμε το πρώτο από τα frames (αν και δεν έχει κάποια διαφορά επί της ουσίας). Ως αποτέλεσμα έχουμε:

Εικόνα 7.1.3

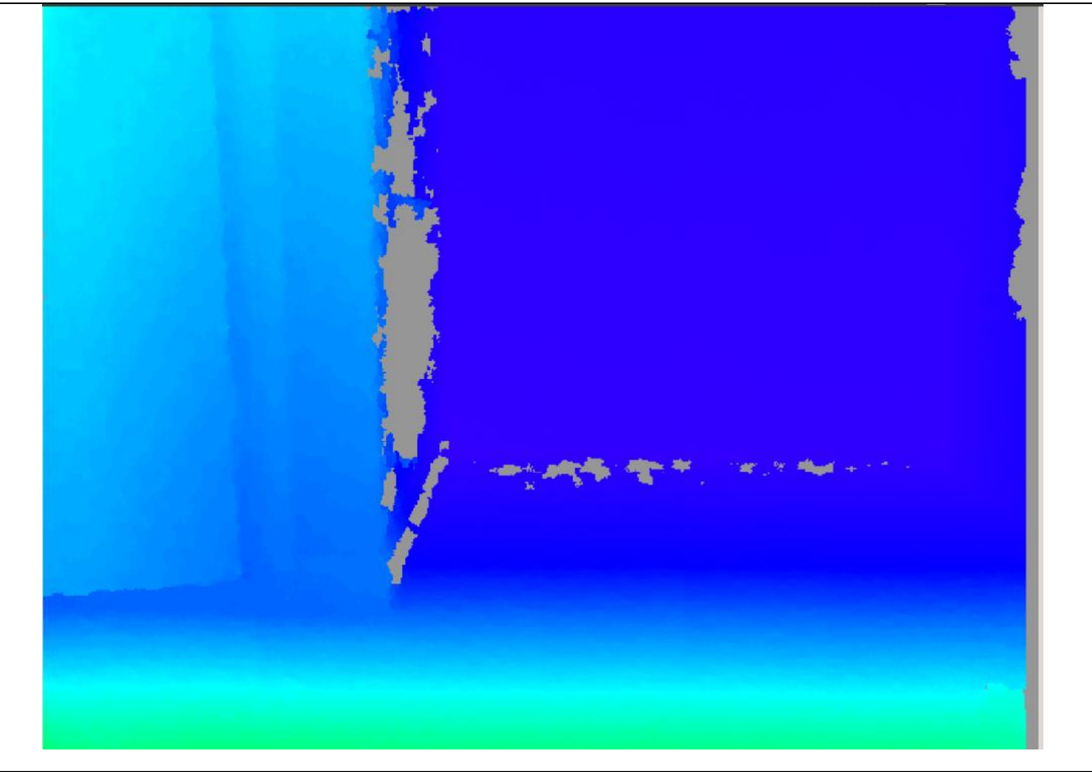
Εικόνα 7.1.4

Αυτά που βλέπουμε στην εικόνα 7.1-3 και 7.1-4 είναι τα KeyPoints. Όπως έχουμε αναφέρει, κάθε ένα από αυτά τα KeyPoints, έχει και ένα descriptor. Αυτά τα KeyPoints τα βρίσκουμε χρησιμοποιώντας τον AKAZE αλγόριθμο, ενώ την σύγκριση την κάνουμε με τον BF Matcher και μερικά φίλτρα που κατασκευάσαμε.

Τώρα πάμε να δούμε τι βλέπει η IR κάμερα του Kinect, δηλαδή το βάθος.



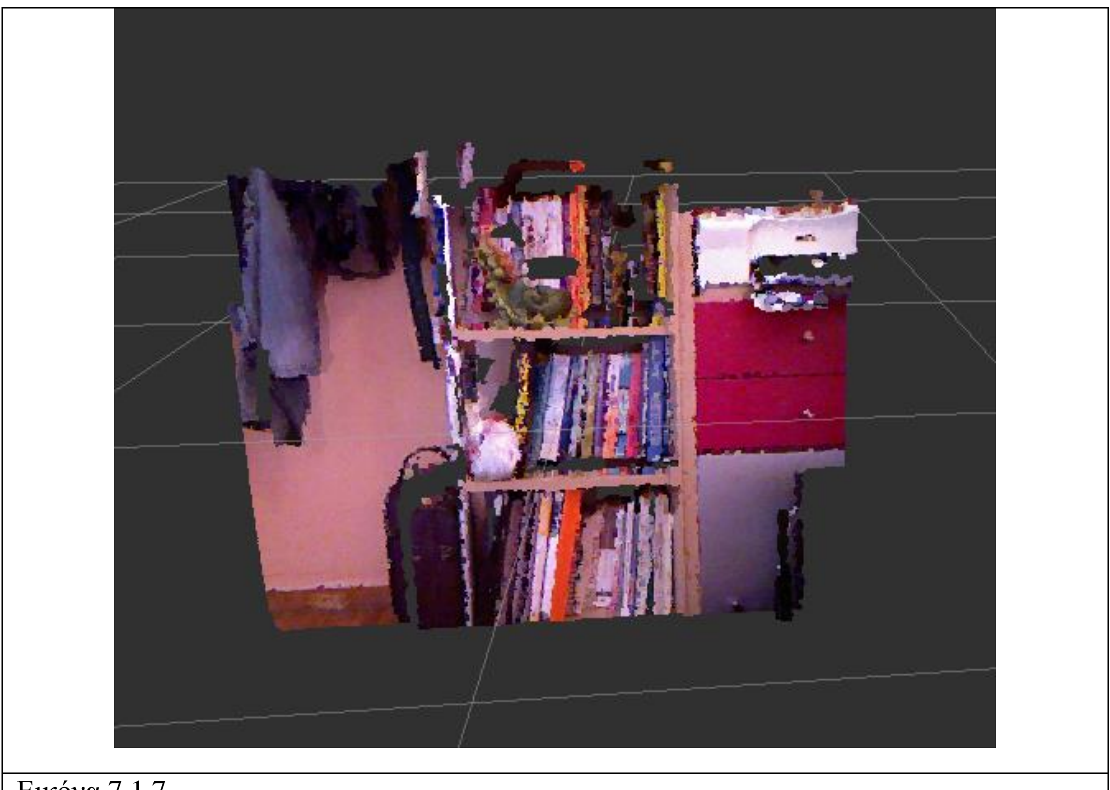
Εικόνα 7.1.5



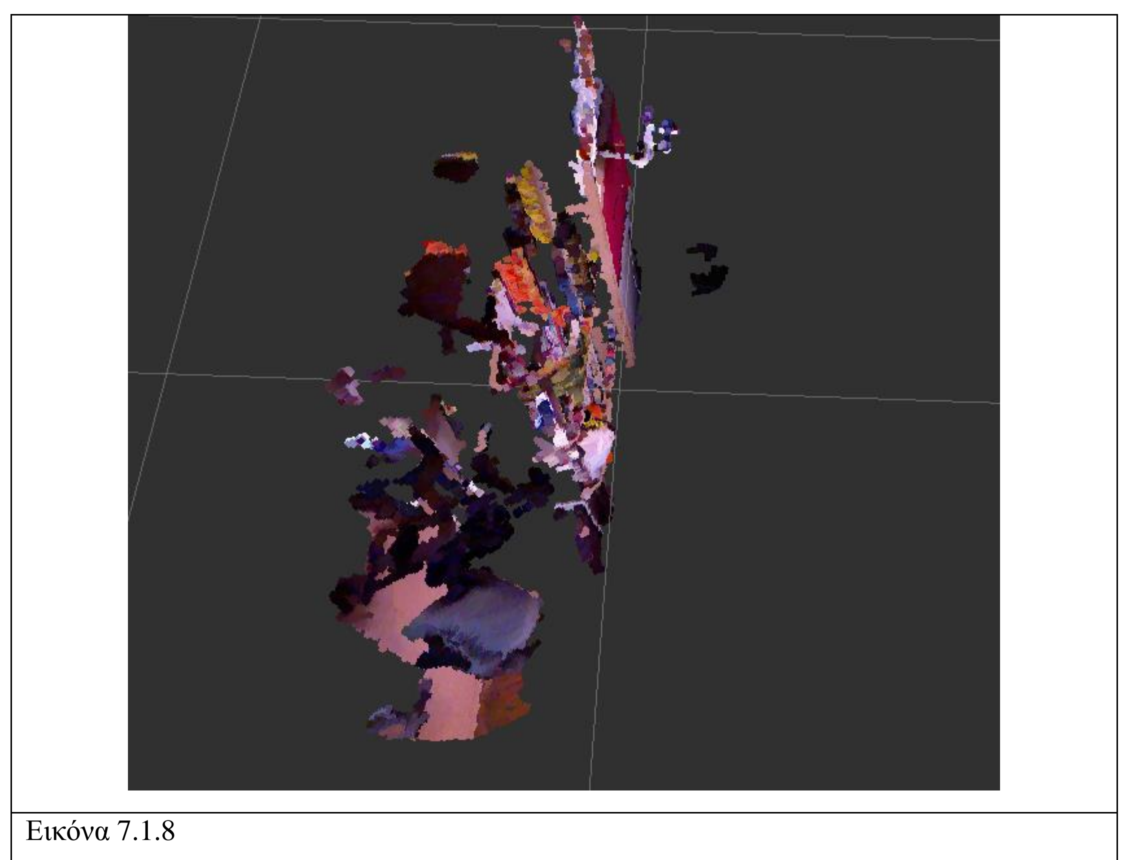
Εικόνα 7.1.6

Στις εικόνες 7.1-5 και 7.1-6 βλέπουμε το frame βάθους. Όσο πιο μπλε το χρώμα, τόσο πιο μακριά είναι. Πράσινο δηλώνει ότι βρίσκετε στην μέση του εύρους που μπορεί το Kinect. Τέλος, όσο πιο κόκκινο, τόσο πιο κοντά βρίσκεται.

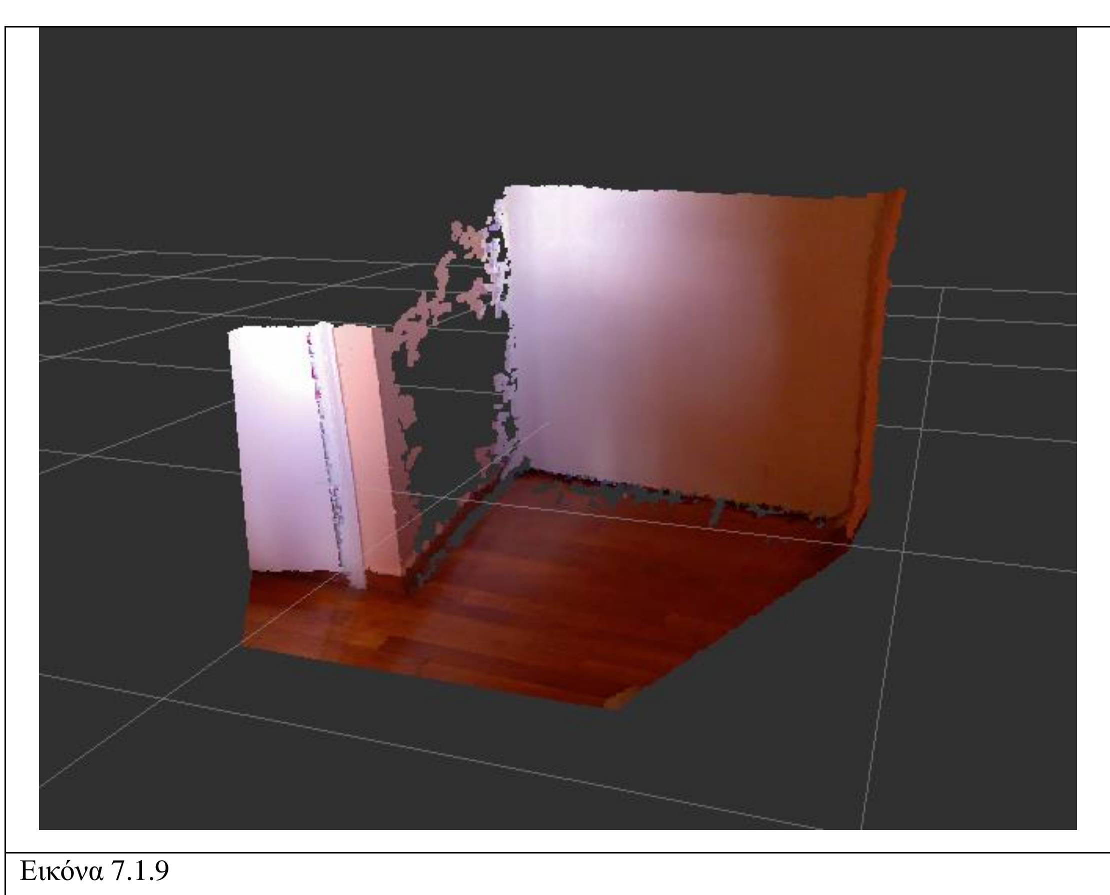
Τέλος πρέπει να συνδυάσουμε αυτά τα δύο frames. Τώρα θα δούμε πώς είναι όταν βάζουμε πάνω στο depth frame, το color frame.



Εικόνα 7.1.7



Εικόνα 7.1.8



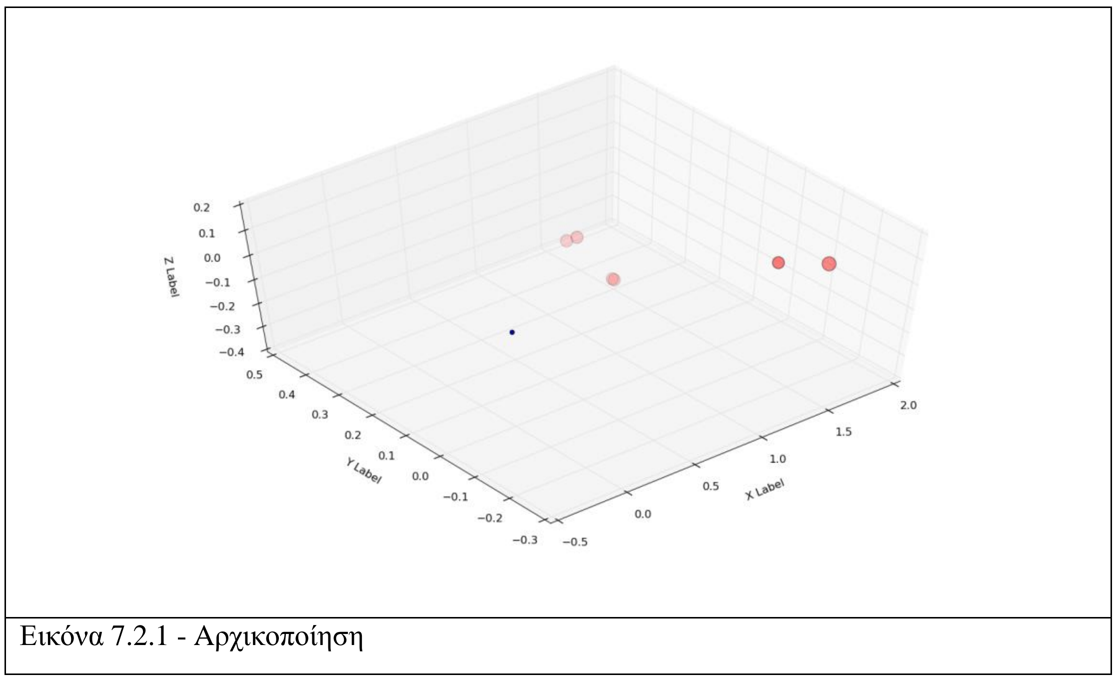
Εικόνα 7.1.9

Στις εικόνες 7.1-7, 7.1-8 και 7.1-9 βλέπουμε τον συνδυασμό των δύο frames. Εφόσον το front_end βρήκε τα KeyPoints, τώρα έχει και το βάθος του κάθε KeyPoints. Το μόνο που μένει είναι να σταλθούν στο SLAM. Οι εικόνες 7.1-7 και 7.1-8 δείχνουν την ίδια σκηνή, απλώς την βλέπουμε από πάνω στην δεύτερη περίπτωση. Αυτό το δείχνουμε ώστε να γίνει πιο κατανοητό πως υπάρχει το βάθος.

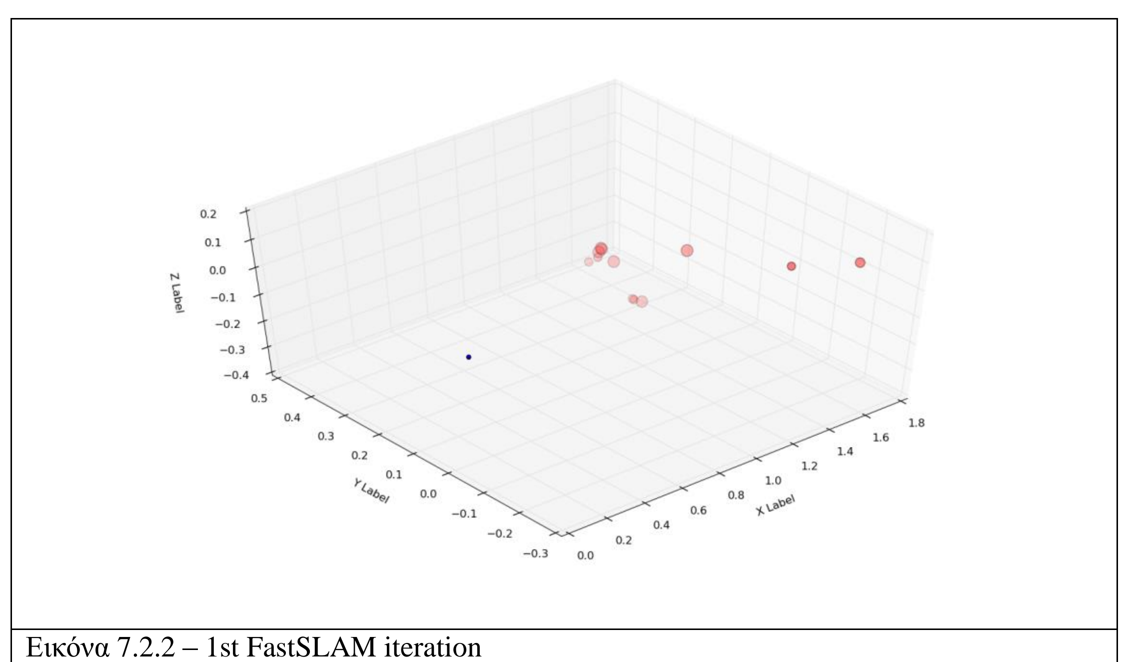
## 7.2. Κατασκευή του χάρτη

Ας δούμε με ποιον τρόπο ένα particle χτίζει το feature-based χάρτη του. Θα δείξουμε πως είναι ο χάρτης στην φάση της αρχικοποίησης και πως πέντε βήματα μετά. Η εντολή είναι να μετακινηθεί ευθεία κατά 0.1 μέτρο. Κάνουμε χρήση της plot_map() συνάρτησης. Να θυμίσουμε ότι το μπλε είναι το particle και τα κόκκινα είναι τα landmarks. Το μέγεθος της αβεβαιότητας των landmarks αντιπροσωπεύεται από το μέγεθος των κόκκινων σφαιρών. Όσο πιο μεγάλο μέγεθος, τόσο μεγαλύτερη αβεβαιότητα. Το αντίστοιχο ισχύει και για μικρά μεγέθη.

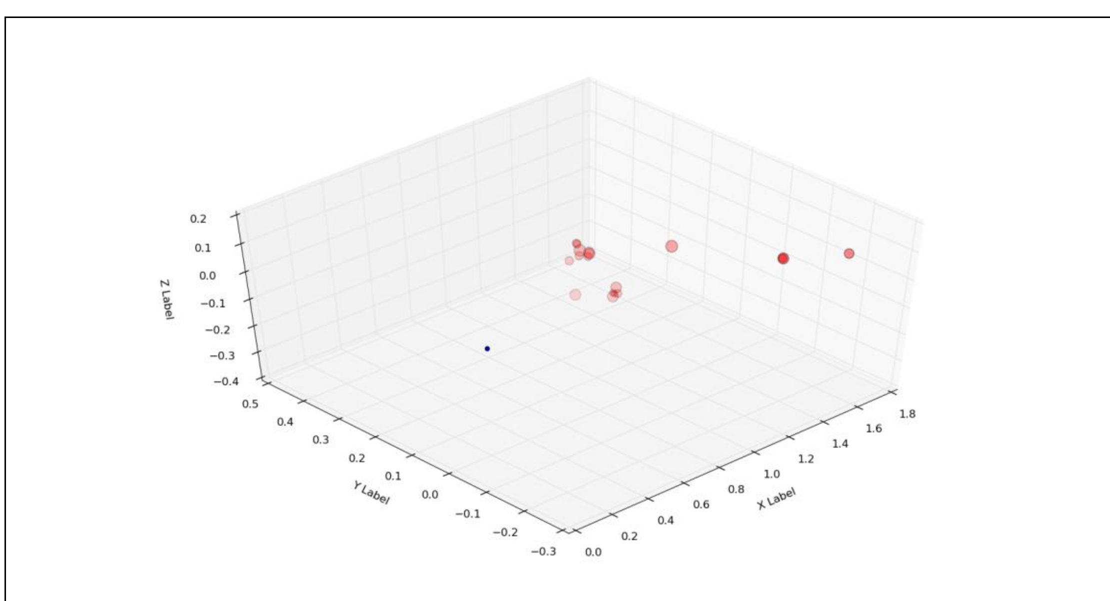
Από πλευρά υλοποίησης, είχαμε το Kinect στερεωμένο πάνω σε ένα σκαμνί, να κοιτάζει την βιβλιοθήκη. Κάθε φορά που τελείωνε ένα γύρο το FastSLAM, κουνούσαμε σταθερά το Kinect 0.1m .



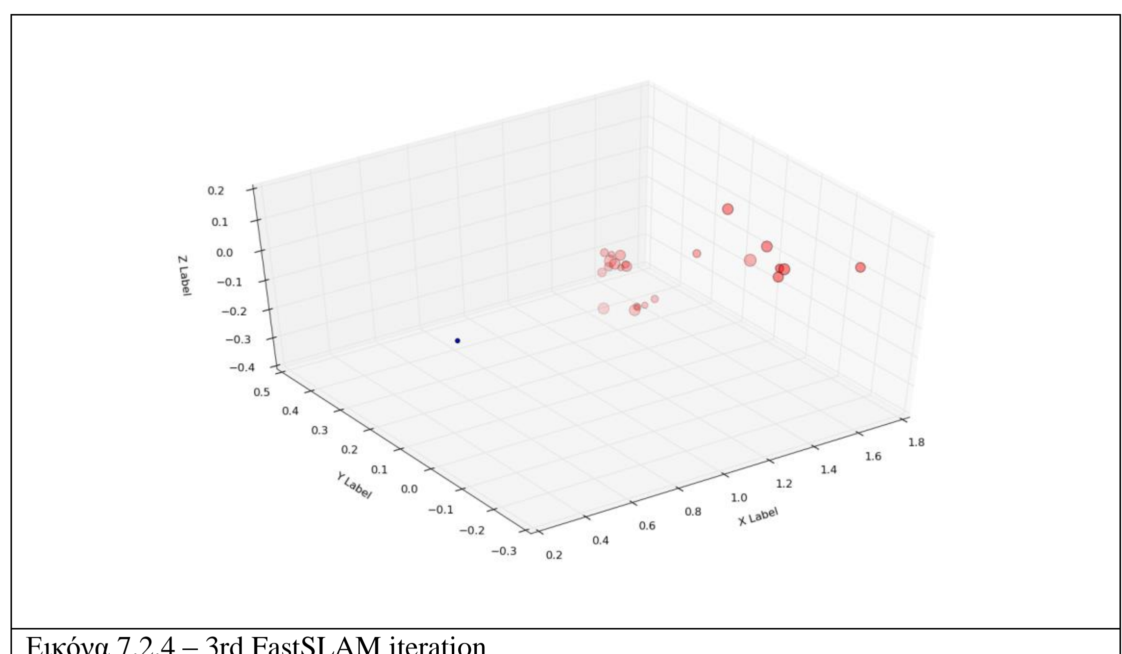

Εικόνα 7.2.1 - Αρχικοποίηση



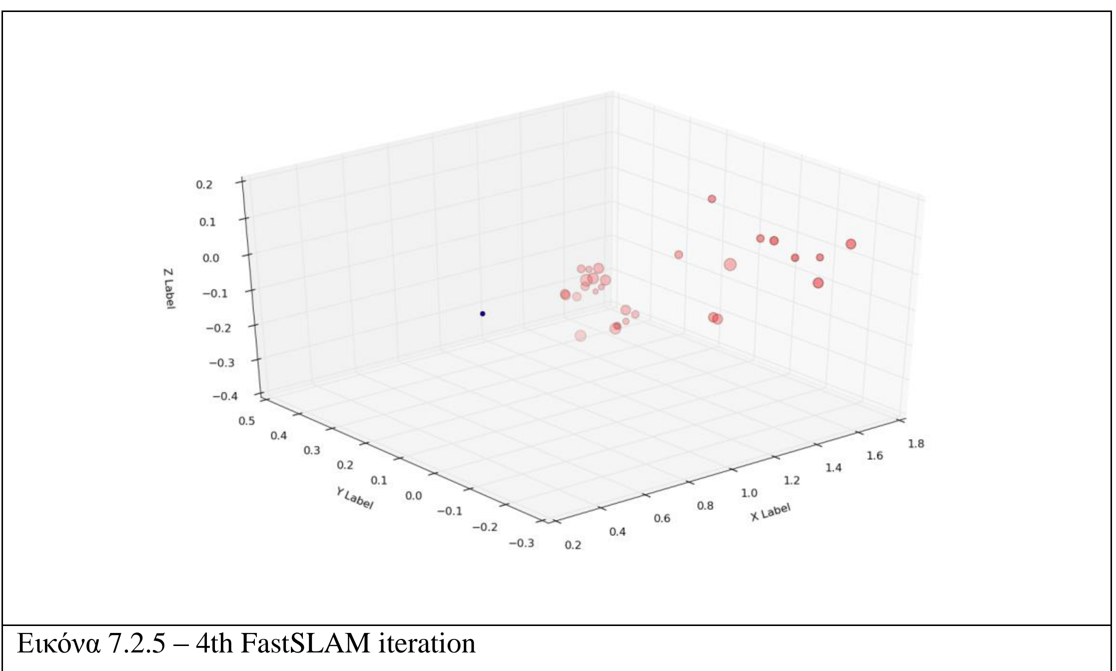

Εικόνα 7.2.2 – 1st FastSLAM iteration



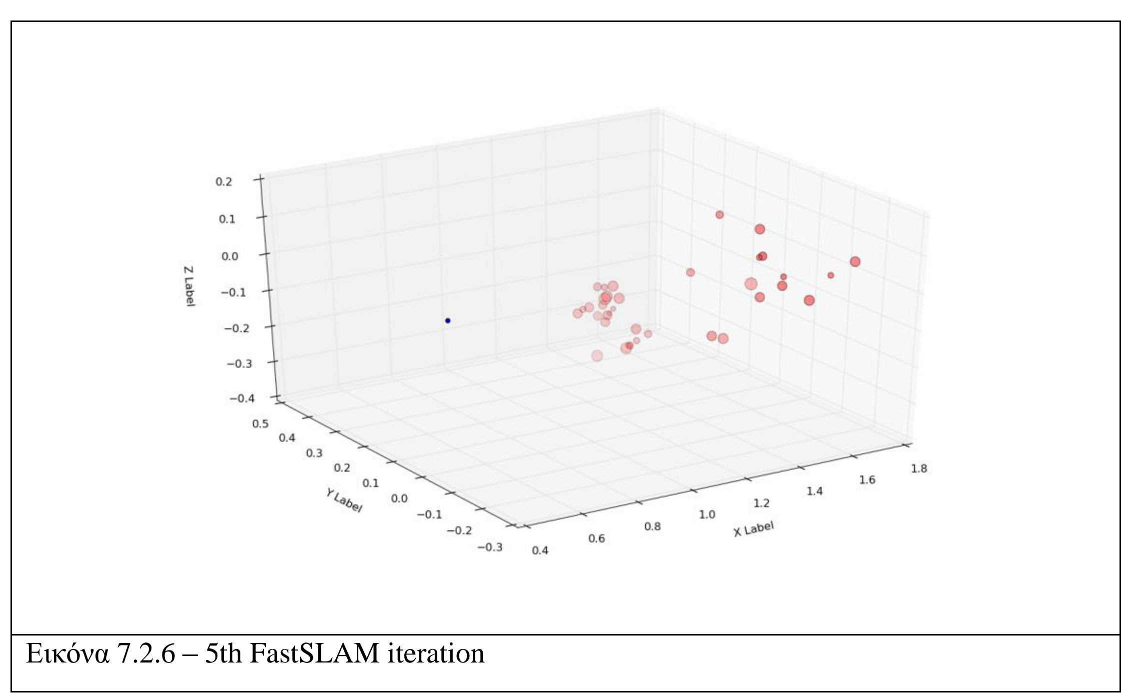

Εικόνα 7.2.3 – 2nd FastSLAM iteration




Εικόνα 7.2.4 – 3rd FastSLAM iteration




Εικόνα 7.2.5 – 4th FastSLAM iteration

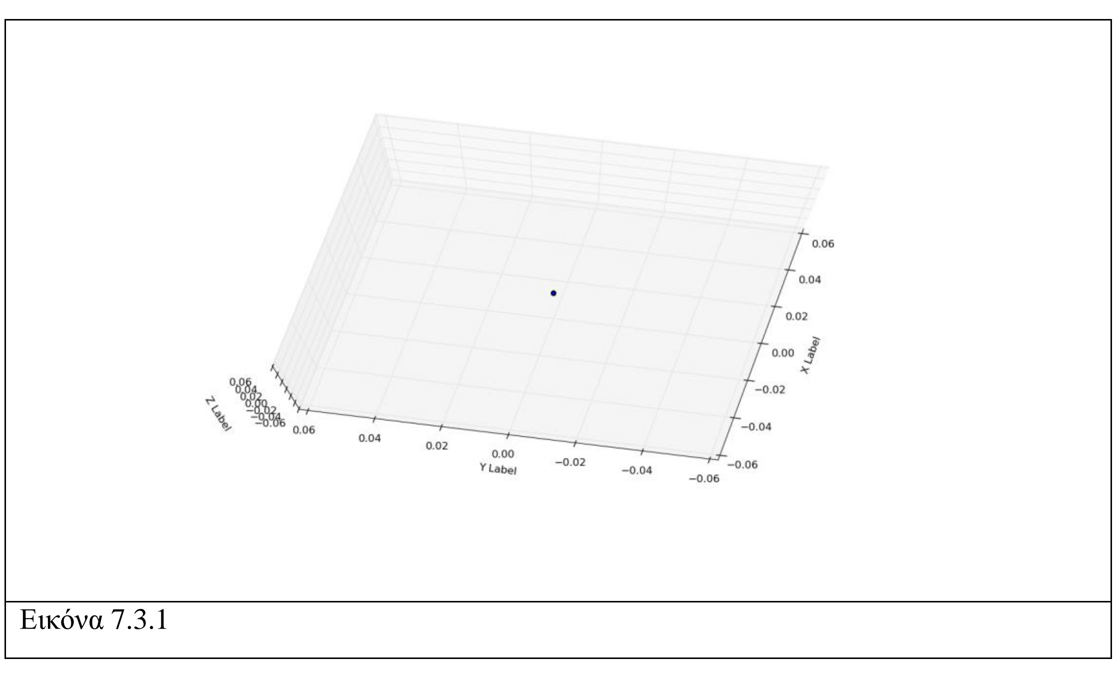



Εικόνα 7.2.6 – 5th FastSLAM iteration

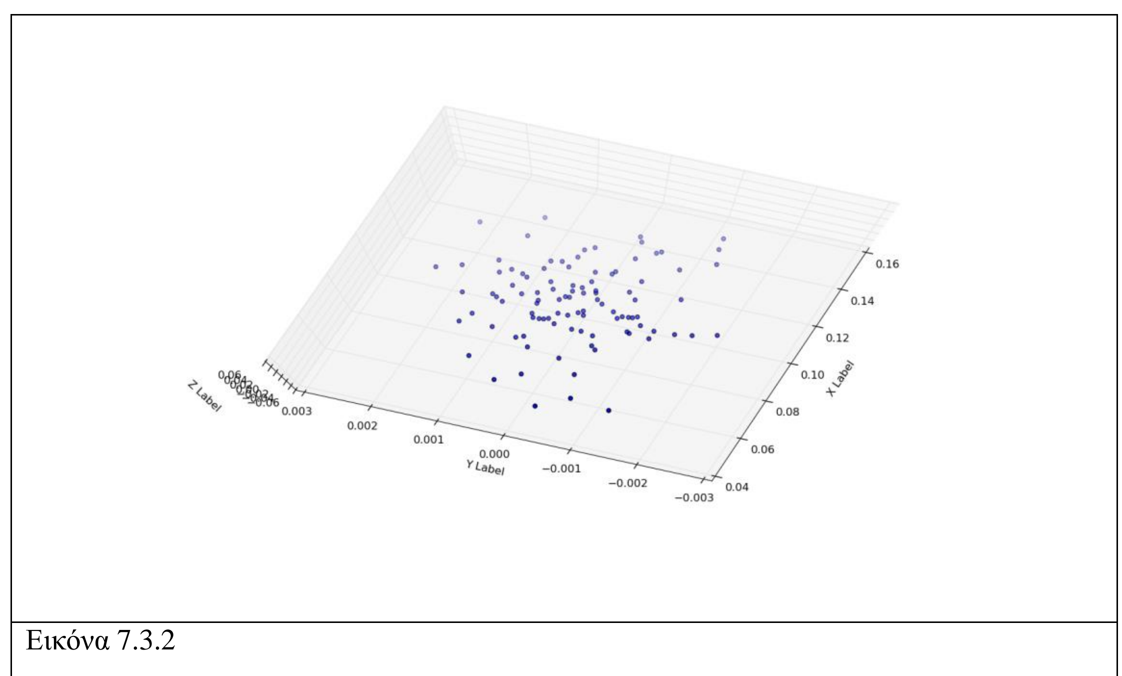
Αν παρακολουθήσετε τον άξονα x, το ρομπότ κινείται ευθεία ενώ τα landmarks παραμένουν με επιτυχία στην θέση τους και η αβεβαιότητα τους μειώνεται συνεχώς. Κάπως έτσι μοιάζει ένα feature-based map.

## 7.3. Κίνηση των particles.

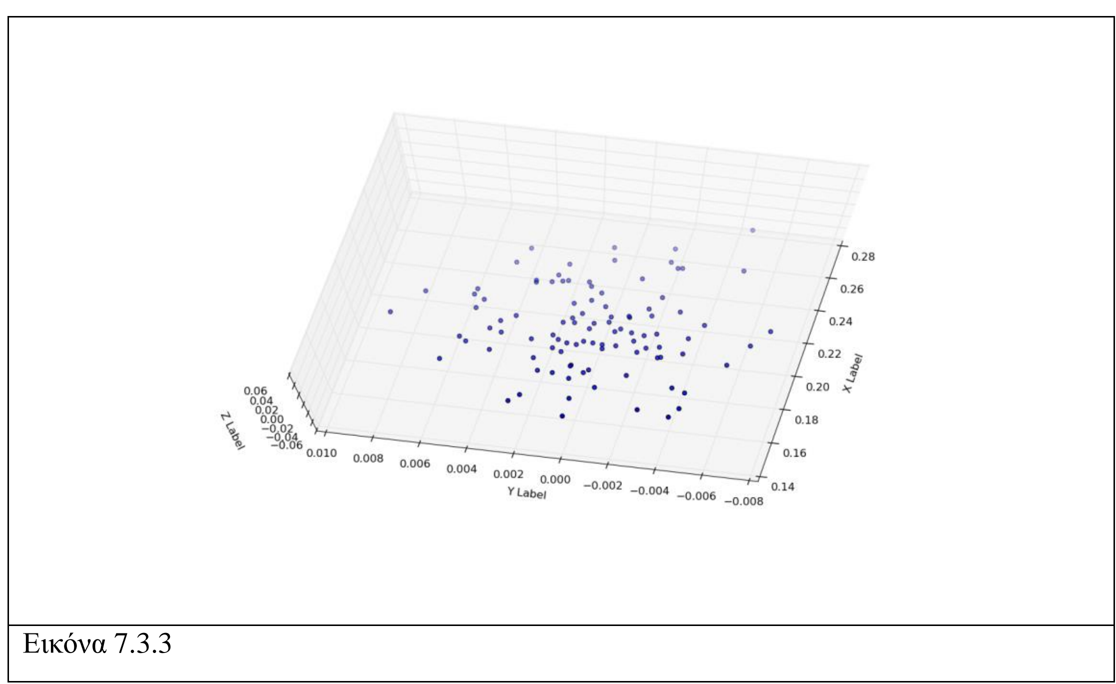
Ας δούμε τώρα πως φαίνονται όλα τα particles μαζί όταν κινούνται. Η πειραματική διαδικασία έγινε με τον ίδιο τρόπο που αναφέρει η ενότητα 7.2 .



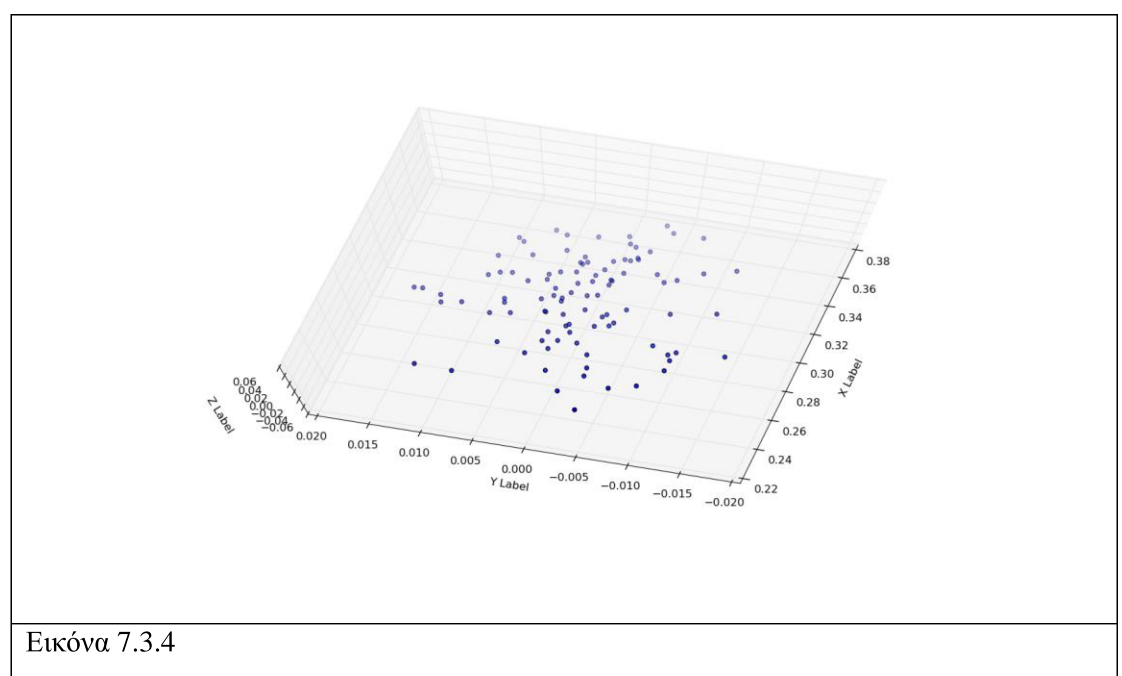

Εικόνα 7.3.1



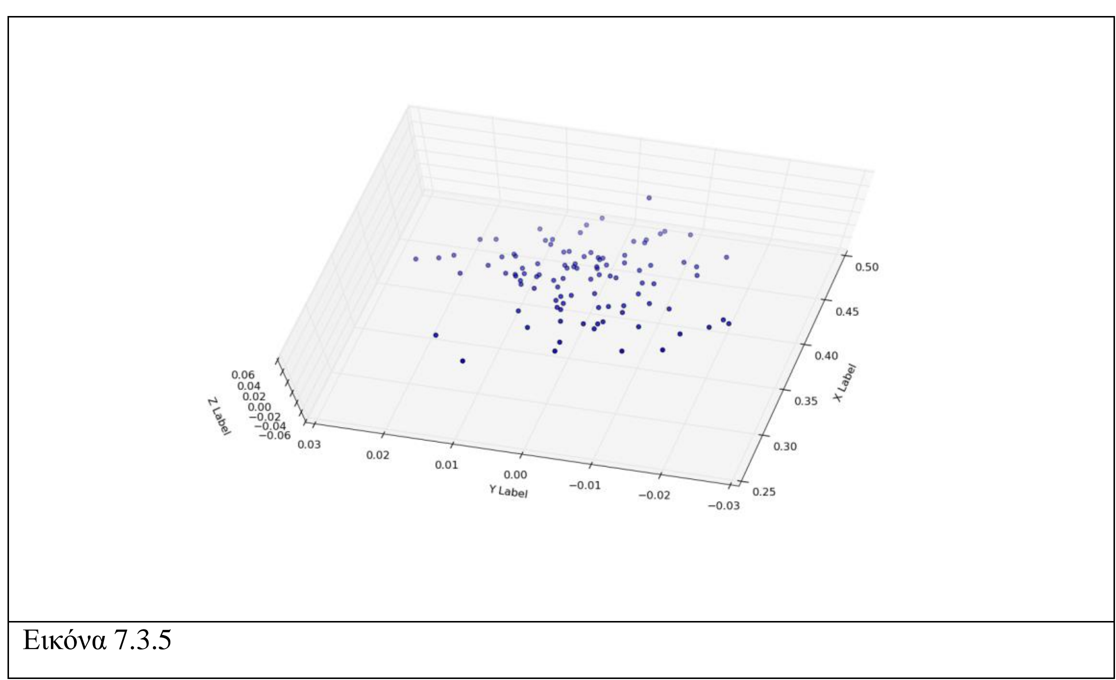

Εικόνα 7.3.2



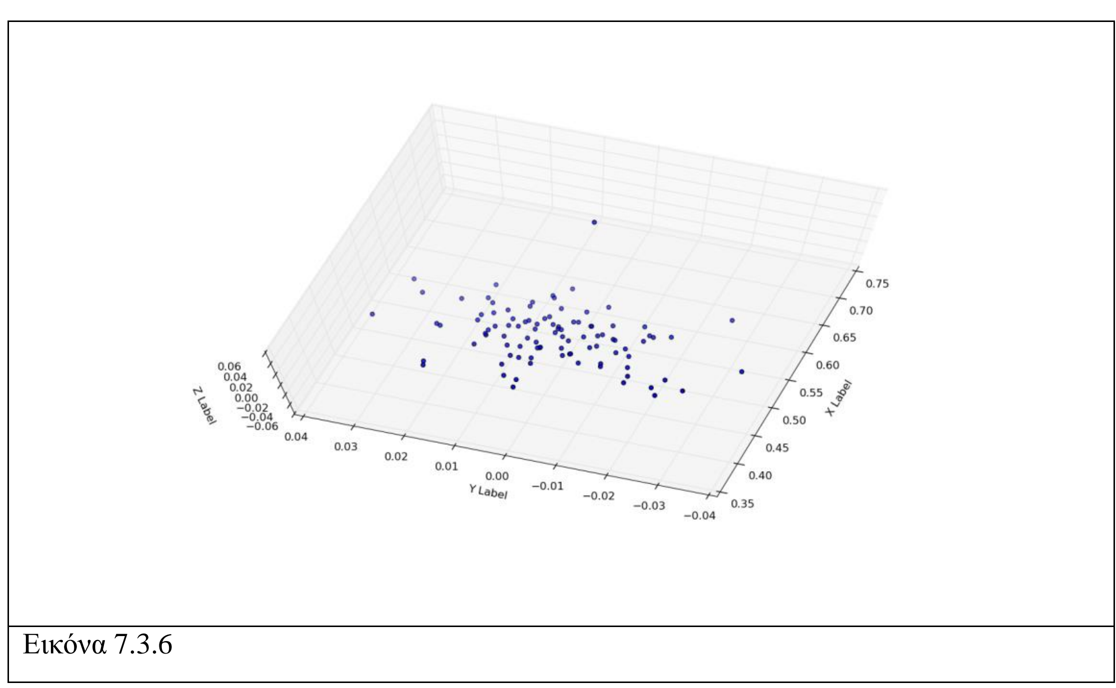

Εικόνα 7.3.3




Εικόνα 7.3.4




Εικόνα 7.3.5




Εικόνα 7.3.6

Όπως αναφέραμε πριν, στέλνουμε εντολή να κουνηθεί μπροστά 0,1m κάθε φορά. Άμα παρατηρήσετε, όλα τα particles βρίσκονται γύρω από το σημείο στο οποίο όντως βρισκόταν το Kinect. Επίσης, αν και φαίνεται μεγάλη η διασπορά των particles, άμα παρατηρηθούν οι άξονες, δεν είναι τόσο μεγάλη η αβεβαιότητα για ένα ρομπότ που κινείται μέσα σε διαδρόμους και δωμάτια.

# Κεφάλαιο 8. - Μελλοντικές βελτιώσεις

Σε αυτό το τελικό κεφάλαιο θα μιλήσουμε για τυχών βελτιώσεις που μπορούν να γίνουν στο ρομποτικό μας σύστημα. Ο στόχος των αλλαγών θα είναι η βελτείωσει των ήδη υπάρχων λειτουργειών αλλά και να δημιουργηθηθεί η υποδομή για προσθήκη νέων.

## 8.1. Συνδυασμός feature-based map με grid-based map

Στο δικό μας σύστημα κατασκευάζουμε feature-based map. Όπως προαναφέρθηκε, η διαδικασία αυτή έχει δύο θετικά:

1. Έχει μοναδικά landmarks, ώστε να γίνεται εύκολα το localization
2. Χρειάζεται λιγότερη επεξεργαστική ισχύς για να κάνει extract landmarks από τα features, σε σχέση με το grid-based map

Όμως το μεγαλύτερο μειονέκτημα που έχει το feature-based map είναι ότι δεν επιτρέπει να γίνει path planning. Για ένα αυτόνομο ρομπότ για παράδειγμα, αυτό είναι μεγάλο μειονέκτημα.

Ένας τρόπος για να ξεπεραστεί αυτό το μειονέκτημα, είναι να χτίσουμε ένα voxel-map σύμφωνα με τις μετρήσεις που λαμβάνουμε απο το Kinect και στην συνέχεια πάνω στον χάρτη που δημιουργήθηκε να τοποθετήσουμε τα features. Αυτό έχει ως αποτέλεσμα να έχουμε την δυνατότητα να κάνουμε path planning αλλά και localization με την χρήση μοναδικών landmarks. Φυσικά για να γίνει αυτό, θα πρέπει να θυσιάσουμε το ένα θετικό στοιχείο του feature-based map όπου είναι η χαμηλή ανάγκη επεξεργαστικής ισχύς .

## 8.2. Εναλλακτικό motion model

Ένα εναλλακτικό motion model που θα μπορούσαμε να χρησιμοποιήσουμε είναι το inverse kinematic model. Το inverse kinematic model έχει σας προϋπόθεση την χρήση μερικών αισθητήρων. Οι αισθητήρες αυτοί θα μπορούσαν να είναι για παράδειγμα ένα γυροσκόποιο και μερικοί encoders . Το γυροσκόποιο θα μπουρούσε να μετράει τις αλλαγές στις γωνίες row, pitch και yaw ενώ οι encoders θα μετράνε την απόσταση μετατόπισης. Συνδυάζοντας τα δεδομένα του γυροσκόποιου και των encoders μπορούμε να υπολογίσουμε την πλήρη μετακίνηση του ρομπότ ως προς x, y, z, row, pitch και yaw.

# Κεφάλαιο 9. Παραρτήματα


| | |
|---|---|
| [0-1] | https://en.wikipedia.org/wiki/The_Steam_Man_of_the_Prairies |
| [0-2] | https://en.wikipedia.org/wiki/Metropolis_(1927_film) |
| [1.2–1] | A. Nuchter, H. Surmann, ”6D SLAM with an Application in Autonomous Mine Mapping”, Proceedings of ICRA, 2004, New Orleans. |
| [1.2-2] | D. Ferguson, A. Morris, D. Hahnel, C. Baker, Z. Omohundro, C. Re- ¨ verte, S. Thayer, W. Whittaker, W. Burgard, S. Thrun, ”An Autononous Robotic System for Mapping Abondoned Mines”, Advances in Neural Information Processing Systems, 2003. |
| [1.2-3] | Lee Hee Gim, F. Fraundorfer, M. Pollefeys, “Structureless pose-graph loop-closure with a multi-camera system on a self-driving car”, IROS, 2013 |
| [1.2-4] | M.J. Milford, G.F. Wyeth, “Single camera vision-only SLAM on a suburban road network”, ICRA 2008, Pasadena, CA. |
| [1.2-5] | F. Caballero, L. Merino, J. Ferruz, A. Ollero, “Vision-Based Odometry and SLAM for Medium and High Altitude Flying UAVs”, Journal of Intelligent and Robotic Systems, March 2009 |
| [1.2-6] | J. Kim, S. Sukkarieh, “Real-time implementation of airborne inertial-SLAM”, Robotics and Automation Systems, Vol 55, Issue 1, January 2007 |
| [1.2-7] | D. Ribas, P. Ridao, J. Neira, J.D. Tardos, “SLAM using an Imaging Sonar for Partially Structured Underwater Environments”, IROS 2006 |
| [1.2-8] | D. Ribas, P. Ridao, J.D. Tardos, J. Neira, “Underwater SLAM in a marine environment”, IROS 2007 |
| [1.2-9] | https://en.wikipedia.org/wiki/Stanley_(vehicle) |
| [1.4-1] | M. Montemerlo, S. Thrun, D. Koller and B. Wegbreit "FastSLAM: A factored solution to the simultaneous localization and mapping problem", Proceedings of the AAAI National Conference on Artificial Intelligence |
| [1.4-2] | J. HyungGi, J. Sungjin, K. Euntai, J. Sewoong, Y. Changyong, “3D FastSLAM algorithm with Kinect sensor”, SCIS, 2014 Joint 7th ISIS, 15th Int.Symposium on |
| [2.2-1] | http://www.ros.org/about-ros/ |
| [2.2-1] | http://pointclouds.org/about/ |
| [2.2-2] | www.pointclouds.org |
| [2.2-3] | http://en.wikipedia.org/wiki/BSD_licenses |
| [2.4-1] | https://en.wikipedia.org/wiki/MATLAB |
| [2.5-1] | https://en.wikipedia.org/wiki/Kinect |
| [3.1.1-1] | http://www.cs.berkeley.edu/~pabbeel/cs287-fa13/slides/bayes-filters.pdf |
| [3.1.2-1] | https://en.wikipedia.org/wiki/Bayes%27_theorem |
| [3.2-1] | https://en.wikipedia.org/wiki/Kalman_filter |
| [3.3-1] | https://en.wikipedia.org/wiki/Extended_Kalman_filter |
| [3.4-1] | https://en.wikipedia.org/wiki/Particle_filter |

| | |
|---|---|
| [4.1-1] | A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, “OctoMap: an efficient probabilistic 3D mapping framework based on octrees,” Autonomous Robots, vol. 34, pp. 189-206, Apr 2013 |
| [4.1-2] | P. F. Alcantarilla, J. Nuevo, A. Bartoli, “Fast Explicit Diffusion for Accelerated in Nonlinear Scale Spaces”, BMVC, September 2013 |
| [4.2-1] | K. Murphy. Bayesian map learning in dynamic environments. In Advances in Neural Information Processing Systems (NIPS). MIT Press, 1999. |
| [4.2-2] | M. Montemerlo, S. Thrun, D. Koller, B. Wgbreit, “FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem” |
| [4.3-1] | http://www.robesafe.com/personal/pablo.alcantarilla/papers/Alcantarilla12eccv.pdf |
| [4.3-2] | http://www.robesafe.com/personal/pablo.alcantarilla/papers/Alcantarilla13bmvc.pdf |
| [4.4-7] | https://en.wikipedia.org/wiki/RANSAC<br>http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.3035&rep=rep1&type=pdf |
| [5.3-1] | http://www.mrpt.org/ |