

**ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ**  
**ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ**  
**ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ ΥΠΟΛΟΓΙΣΤΩΝ**  
**ΤΟΜΕΑΣ:**  
**ΕΡΓΑΣΤΗΡΙΟ**

---

## **Διπλωματική Εργασία**

του φοιτητή του Τμήματος Ηλεκτρολόγων Μηχανικών  
και Τεχνολογίας Υπολογιστών της Πολυτεχνικής Σχολής  
του Πανεπιστημίου Πατρών

**ΑΓΓΕΛΑΚΗΣ ΠΑΝΑΓΙΩΤΗΣ του ΓΕΩΡΓΙΟΥ**

Αριθμός Μητρώου: 227982

Θέμα

**«Αποφυγή εμποδιών με αυτοκινούμενο ρομποτικό οχήμα»**

Επιβλέπων Αν. Καθηγητής  
Ε.ΔΕΡΜΑΤΑΣ

**Αριθμός Διπλωματικής Εργασίας:**

Πάτρα, (Γράφετε Μήνα και Έτος)

## **ΠΙΣΤΟΠΟΙΗΣΗ**

Πιστοποιείται ότι η Διπλωματική Εργασία με θέμα

### **«Αποφυγή εμποδιών με αυτοκινούμενο ρομποτικό οχήμα»**

Του φοιτητή του Τμήματος Ηλεκτρολόγων Μηχανικών και  
Τεχνολογίας Υπολογιστών

ΑΓΓΕΛΑΚΗΣ ΠΑΝΑΓΙΩΤΗΣ του ΓΕΩΡΓΙΟΥ

Αριθμός Μητρώου: 227982

Παρουσιάστηκε δημόσια και εξετάστηκε στο Τμήμα  
Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών  
στις  
...../...../.....

Ο Επιβλέπων

Ε. ΔΕΡΜΑΤΑΣ

Αναπληρωτής καθηγητής

Ο Διευθυντής του Τομέα

Ν. Φακωτακής

Καθηγητής

**Αριθμός Διπλωματικής Εργασίας:**

## Θέμα: «Αποφυγή εμποδίων με αυτοκινούμενο ρομποτικό οχήμα»

Φοιτητής:

Αγγελάκης Παναγιώτης

Επιβλέπων:

Ευάγγελος Δερμάτας  
Αναπληρωτής καθηγητής

### Περίληψη

Η παρούσα εργασία έχει ως αντικείμενο το σχεδιασμό και την κατασκευή ενός ενσωματωμένου συστήματος πλοήγησης ρομποτικού οχήματος, καθώς και την κατασκευή δισδιάστατων και τρισδιάστατων χαρτών σε εσωτερικούς χώρους.

Το ρομποτικό όχημα θα πρέπει να ξέρει που βρίσκεται ανά πασά στιγμή στο χάρτη ώστε να μπορεί να υπολογίσει μια πορεία για να πάει από το σημείο Α στο Β αποφεύγοντας τυχόν νέα εμπόδια στον δρόμο του που δεν υπάρχουν στους χάρτες.

Θα πρέπει να είναι σε θέση να λύσει το πρόβλημα της "απαγωγής του ρομπότ" δηλαδή μετακινώντας το ρομπότ σε μια καινούργια θέση αυτό να μπορεί να προσανατολιστεί ξανά, τρισδιάστατα χαρακτηρίστηκα βοηθάνε πολύ σε αυτόν τον τομέα.

Το ρομπότ είναι ένα τετράτροχο διαφορικό όχημα (δηλαδή οι ρόδες του δεν στρίβουν) οπότε το δυναμικό του μοντέλο είναι σχετικά απλό.

Είναι εξοπλισμένο με quadrature encoders σε κάθε ρόδα ώστε να μετράμε την περιστροφή κάθε ρόδας, περνώντας με αυτό τον τρόπο "τυφλή" οδομετρία και με κλειστό ελέγχο pid να κρατάμε σταθερή την ταχύτητα του οχήματος.

Διαθέτει ένα 2-d lidar για να μετράει την απόσταση παντού γύρω του σε ένα επίπεδο, καθώς και μια 3-d RGB-D κάμερα (kinect) για να φτιάξουμε τούς τρισδιάστατους χάρτες και για 3D αναγνώριση αντικειμένων.

Θα ασχοληθούμε με τεχνικές Simultaneous Localization and Mapping or SLAM και Path Planning ώστε το ρομπότ μας να είναι ικανό να φτιάχνει χάρτες σε περιβάλλοντα που δεν έχει ξαναδεί, και να μετακινείται αυτόνομα σε αυτά.

Επίσης θα ασχοληθούμε με το πως το ρομπότ μπορεί να αναγνωρίζει αντικείμενα στον 3D χώρο, και να βρίσκει την τοποθεσία τους στον τρισδιάστατο χώρο, αυτό θα επιτευχθεί με SVM Classifiers και τα χαρακτηριστικά θα είναι hsv-color histograms και normal-shape histograms, για το training και την αναγνώριση.

Τέλος θα εξοπλίσουμε το ρομπότ με ένα ρομποτικό χέρι 6 βαθμών ελευθέριας, ώστε να είναι σε θέση να αλληλεπιδρά με το περιβάλλον του μετατρέποντας το σε mobile manipulator.

Ο εγκέφαλος είναι ο ενσωματωμένος υπολογιστής Jetson Nano που με 4 cores στα 1.3Ghz και 125 cores gpus μας παρέχει με την υπολογιστική τιχύ που χρειαζόμαστε.

Όλα αυτά θα τα κάνουμε αξιοποιώντας τις δυνατότητες του Robot Operating System (ROS) που μας βοηθάει να προγραμματίζουμε γρήγορα ρομπότ με πακέτα που έχουν σχεδιάσει άλλοι, καθώς και με την επικοινωνία των διάφορων λειτουργιών μεταξύ τους.

Διαβάζοντας αυτή την διπλωματική εργασία θα εξοικειωθείτε με τις δυνατότητες του ROS, και θα είστε σε θέση να φτιάξατε και εσείς παρόμοια ρομπότ με έναν μεθοδικό τρόπο ακόμα και χωρίς να έχετε προηγούμενη εμπειρία με την σχεδίαση και τον προγραμματισμό ρομπότ.

## Abstract

The purpose of the present work is to design and build an integrated robotic vehicle navigation system, as well as to construct two-dimensional and three-dimensional indoor maps.

The robotic vehicle should know where it is at all times on the map so that it can compute a route to go from point A to point B, while avoiding any new obstacles on its way that are not on the maps.

It should also be able to solve the problem of "kidnapping the robot" that is, moving the robot to a new position, the robot can re-localize itself, three-dimensional features are very helpful in this area.

The robot is a 4-wheel-drive differential (meaning that the wheels don't turn) so its dynamic model is relatively simple.

The robot is also equipped with quadrature encoders on each wheel to measure the rotation of each wheel, thereby computing "blind" odometry and giving us the capability to do a close loop pid control to keep the robot's velocity constant.

It has a 2-d lidar to measure the distance everywhere around it on a plane, as well as a 3-d RGB-D camera (kinect) to create the 3D maps.

We will introduce simultaneous localization and mapping or SLAM and Path Planning techniques, so that the robot will be able to map and autonomously navigate unknown areas

Furthermore, we will see how the robot can recognize 3D objects and their relative position in 3D space. This will be achieved with SVM classifiers which will use features of hsv-color and shape-normal in a histogram form for training and classification.

Lastly we will attach to the robot, a robotic hand of 6 degrees of freedom so that it can interact with its environment transforming it into a mobile manipulator.

The brain is the Jetson Nano embedded computer with 4 cores at 1.3 Ghz and 125 cores gpu providing us with the computing power we need.

We will do all of this by taking advantage of the Robot Operating System (ROS) capabilities that help us quickly program robots with packages designed by others, as well as a system to communicate the various operations between them.

Reading this thesis will familiarize you with the capabilities of ROS, and you will be able to build similar robots yourself in a methodical way, without even having previous experience in robot design and programming.

## Ευχαριστίες

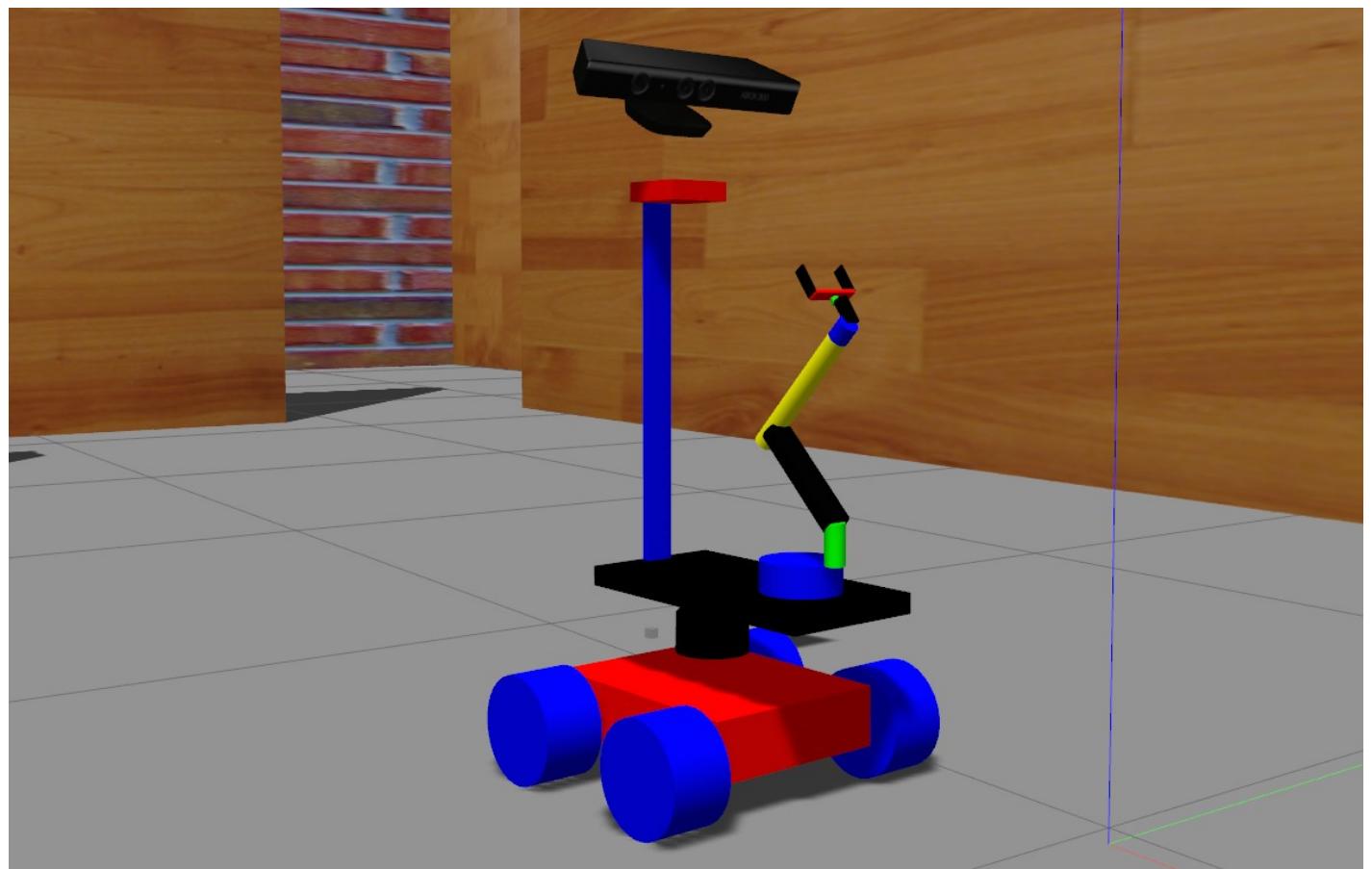
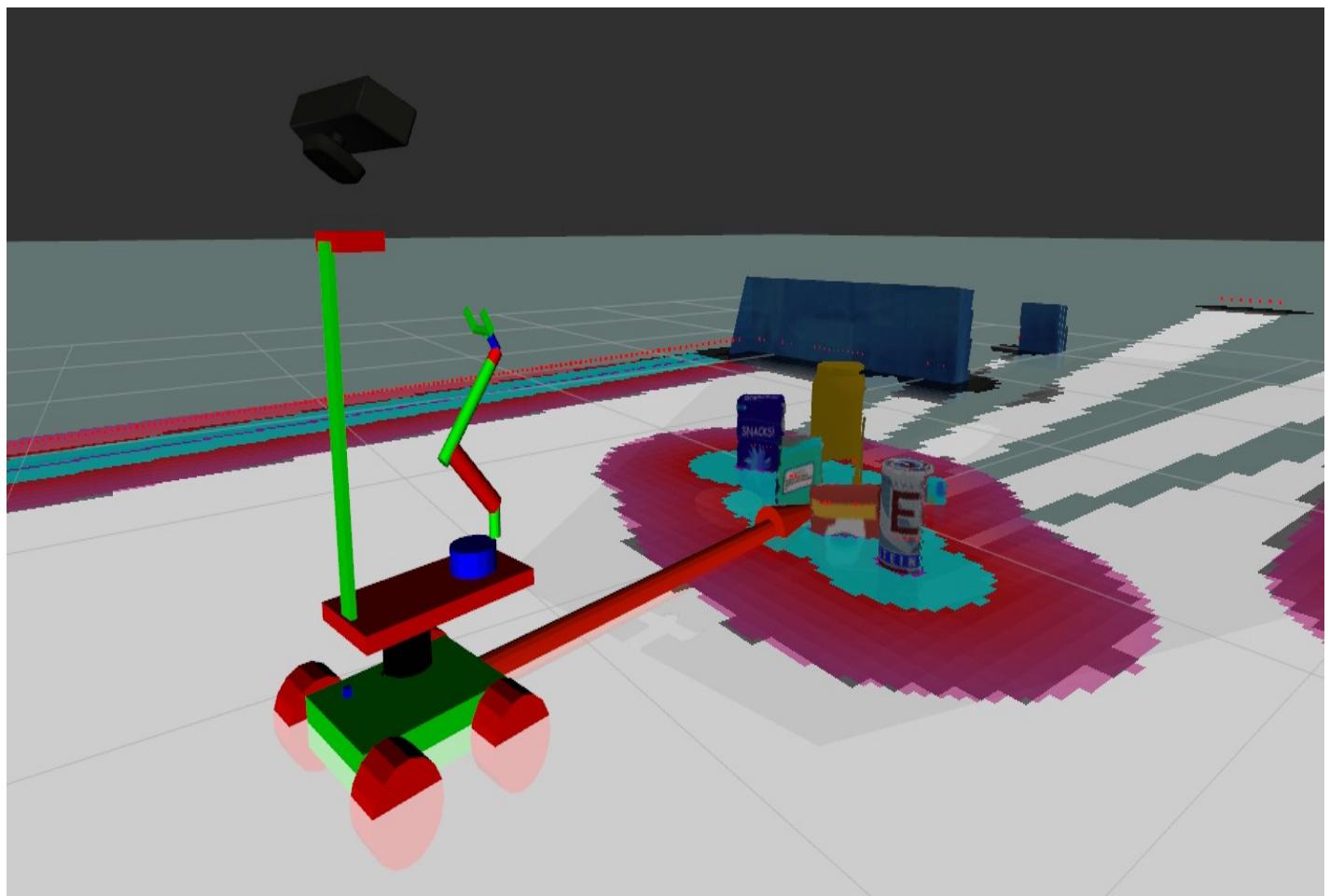
Η παρούσα εργασία εκπονήθηκε κατά το ακαδημαϊκό έτος 2019-2020 στη διάρκεια της φοίτησης μου στο έβδομο έτος σπουδών στο τμήμα Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών του Πανεπιστημίου Πατρών.

Θα ήθελα στο σημείο αυτό να ευχαριστήσω όλους όσους συνέβαλαν στην ολοκλήρωσή της :

Καταρχάς, τον επιβλέποντα καθηγητή μου κ. Ε. Δέρματα που με ενθάρρυνε να ασχοληθώ με το αντικείμενο αυτό καθώς και για τις χρήσιμες συμβουλές του καθ όλη τη διάρκεια της εργασίας και για τον πολύτιμο του χρόνο που αφιέρωσε .

Θα ήθελα ακόμη να ευχαριστήσω την οικογένεια μου και τους κοντινούς μου φίλους για τη στήριξη τους σε όλο το διάστημα των σπουδών μου αλλά και της εκτέλεσης και συγγραφής της παρούσας εργασίας.





# Table of contents

<b>Abstract .....</b>	4
-----------------------	---

## Theory

### 1. Types of Robotic Vehicles

1.1 Holonomic and Non – Holonomic Drive.....	14
1.2 Differential – Non Holonomic Drive.....	15
1.3 Omni-Directional – Mecanum Drive – Holonomic Drive.....	16
1.4 Ackermann steering geometry - Cars.....	17
1.5 My Robot – Mobile Manipulator.....	18

### 2. Hardware and Sensors Used

2.1 Introduction.....	19
2.2 DC motor.....	19
2.3 Stepper motor.....	22
2.4 Quadrature encoder.....	25
2.5 L298N Dual H-Bridge driver.....	26
2.6 A4988 driver.....	29
2.7 Arduino Microcontroller.....	31
2.8 Jetson nano embedded computer.....	32
2.9 Lidar.....	33
2.10 Kinect v1 Time of Flight camera.....	36
2.11 6 Degrees of Freedom Robotic Hand.....	39
2.12 Summary.....	40

### 3. Introduction to Robot Operating System (ROS)

3.1 Introduction.....	41
3.2 Ubuntu operating system.....	41
3.3 What is a robot?.....	41
3.4 Before ROS.....	42
3.5 What is Robot Operating System (ROS).....	42
3.6 Brief history of ROS.....	43
3.7 Nodes and compute graphs.....	43
3.8 Ros Master and Parameter Server.....	45
3.9 Packages and Catkin workspace.....	46
3.10 Running the turtlesim example.....	48
3.11 Launch and configuration files.....	50
3.12 Topics and writing our first c++ node.....	52
3.13 URDF and XACRO.....	56
3.14 Robot state publisher and Joint state publisher.....	61
3.15 Gazebo.....	64
3.16 Writing a Gazebo service.....	71
3.17 Actions.....	72
3.18 Rviz.....	75
3.19 Moveit.....	79
3.20 ROS Control.....	83
3.21 Debugging Tools.....	85

### 4. Localization Techniques

<b>4.1 Introduction.....</b>	88
<b>4.2 What is localization.....</b>	88
<b>4.3 Localization challenges.....</b>	88
<b>4.4 Kalman Filtering</b>	
<b>4.4.1 Overview.....</b>	89
<b>4.4.2 What's a Kalman Filter.....</b>	89
<b>4.4.3 History.....</b>	91
<b>4.4.4 Applications.....</b>	92
<b>4.4.5 Variations.....</b>	92
<b>4.4.6 Robot Uncertainty.....</b>	93
<b>4.4.7 Kalman Filter Advantage.....</b>	96
<b>4.4.8 1-D Gaussian.....</b>	96
<b>4.4.9 Designing 1-D Kalman Filters.....</b>	98
<b>4.4.10 Measurement Update.....</b>	99
<b>4.4.11 State Prediction.....</b>	102
<b>4.4.12 1-D Kalman Filter.....</b>	103
<b>4.4.13 Multivariate Gausians.....</b>	103
<b>4.4.14 Intro to Multidimensional KF.....</b>	105
<b>4.4.15 Design of Multidimensional KF.....</b>	107
<b>4.4.16 Introduction to EKF.....</b>	110
<b>4.4.17 EKF.....</b>	112
<b>4.4.18 Recap.....</b>	116
<b>4.5 Monte Carlo Localization</b>	
<b>4.5.1 Introduction.....</b>	116
<b>4.5.2 What's MCL?.....</b>	117
<b>4.5.3 Power of MCL.....</b>	117
<b>4.5.4 Particle Filters.....</b>	118
<b>4.5.5 Bayes Filtering.....</b>	121
<b>4.5.6 MCL: The Algorithm.....</b>	123
<b>4.5.7 MCK in Action.....</b>	125
<b>4.5.8 Outro.....</b>	126
<b>5. Introduction to Mapping and SLAM</b>	
<b>5.1 Introduction.....</b>	127
<b>5.2 Occupancy Grid Mapping</b>	
<b>5.2.1 Introduction.....</b>	129
<b>5.2.2 Importance of Mapping.....</b>	129
<b>5.2.3 Challenges and Difficulties.....</b>	130
<b>5.2.4 Mapping with Known Poses.....</b>	132
<b>5.2.5 Posterior Probability.....</b>	133
<b>5.2.6 Grid Cells.....</b>	134
<b>5.2.7 Computing the Posterior.....</b>	135
<b>5.2.8 Filtering.....</b>	135
<b>5.2.9 Binary Bayes Filter Algorithm.....</b>	137
<b>5.2.10 Occupancy Grid Mapping Algorithm.....</b>	138
<b>5.2.11 Inverse Sensor Model.....</b>	139
<b>5.2.12 Multi Sensor Fusion.....</b>	141
<b>5.2.13 Introduction to 3D Mapping.....</b>	142
<b>5.2.14 3D Data Representations.....</b>	143
<b>5.2.15 OctoMap Framework.....</b>	145
<b>5.2.16 Outro.....</b>	146
<b>5.3 Grid-based FastSLAM</b>	
<b>5.3.1 Introduction.....</b>	147
<b>5.3.2 Online SLAM.....</b>	147

5.3.3 Full SLAM.....	148
5.3.4 Nature of SLAM.....	150
5.3.5 Correspondence.....	151
5.3.6 SLAM Challenges.....	153
5.3.7 Particle Filter Approach to SLAM.....	154
5.3.8 Introduction to FastSLAM.....	155
5.3.9 FastSLAM Instances.....	155
5.3.10 Adapting FastSLAM to Grid Maps.....	156
5.3.11 Grid-based FastSLAM Techniques.....	157
5.3.12 The Geid-based FastSLAM Algorithm.....	158
5.3.13 <b>Outro</b> .....	159
<b>5.4 GraphSLAM</b>	
5.4.1 <b>Introduction</b> .....	161
5.4.2 Graphs.....	161
5.4.3 Constraints.....	162
5.4.4 Front-End vs Back-End.....	164
5.4.5 <b>Maximum Likelihood Estimation</b> .....	<b>165</b>
5.4.6 Mid-Chapter Overview.....	174
5.4.7 1-D to n-D.....	175
5.4.8 Information Matrix and Vector.....	176
5.4.9 Inference.....	179
5.4.10 Nonlinear Constraints.....	181
5.4.11 <b>Graph-SLAM at a Glance</b> .....	183
<b>5.5 3D SLAM with RTAB-Map</b>	
5.5.1 <b>Intro to 3D SLAM with RTAB-Map</b> .....	184
5.5.2 3D SLAM with RTAB-Map.....	184
5.5.3 Visual Bag-of-Words.....	187
5.5.4 RTAB-Map Memory Management.....	189
5.5.5 RTAB-Map Optimization and Output.....	190
<b>5.6 Outro</b> .....	<b>193</b>

## 6. Path Planning and Navigation

<b>6.1 Introduction</b> .....	<b>194</b>
<b>6.2 Applications</b> .....	<b>195</b>
<b>6.3 Classic Path Planning</b>	
6.3.1 <b>Introduction to Path Planning</b> .....	196
6.3.2 Examples of Path Planning.....	196
6.3.3 Approaches to Path Planning.....	198
6.3.4 Discrete Planning.....	200
6.3.5 Continuous Representation.....	201
6.3.6 Minkowski Sum.....	202
6.3.7 Translation and Rotation.....	204
6.3.8 3D Configuration Space.....	206
6.3.9 Discretization.....	207
6.3.10 Roadmap.....	208
6.3.11 Visibility Graph.....	208
6.3.12 Voronoi Diagram.....	210
6.3.13 Cell Decomposition.....	210
6.3.14 Approximate Cell Decomposition.....	212
6.3.15 Potential Field.....	214
6.3.16 <b>Discretization Wrap-up</b> .....	<b>217</b>
6.3.17 Graph Search.....	218
6.3.18 Breadth-First Search.....	219
6.3.19 Depth-First Search.....	221

6.3.20 Uniform Cost Search.....	223
6.3.21 A* Search.....	226
6.3.22 Overall Concerns Regarding Search.....	231
6.3.23 <b>Graph-Search Wrap-Up</b> .....	232
6.3.24 <b>Discrete planning Wrap-Up</b> .....	232
<b>6.4 Sample-Based and Probabilistic Path Planning</b>	
6.4.1 <b>Introduction to Sample-Based &amp; Probabilistic Path Planning</b> .....	232
6.4.2 Why Sample-Based Planning?.....	233
6.4.3 Weakening Requirements.....	235
6.4.4 Sample-Based Path Planning.....	235
6.4.5 Probabilistic Roadmap (PRM).....	236
6.4.6 Rapidly Exploring Random Tree Method (RRT).....	240
6.4.7 Path Smoothing.....	244
6.4.8 Overall Concerns.....	246
6.4.9 <b>Sample-Based Planning Wrap-Up</b> .....	247
<b>6.5 Probabilistic Path Planning</b>	
6.5.1 <b>Introduction to Probabilistic Path Planning</b> .....	247
6.5.2 Markov Decision Process.....	248
6.5.3 Policies.....	251
6.5.4 State Utility.....	253
6.5.5 Value Iteration Algorithm.....	256
6.5.6 <b>Probabilistic Path Planning Wrap-Up</b> .....	257
<b>7. 3D Perception</b>	
<b>7.1 Introduction Active and Passive Sensors and specs</b> .....	258
<b>7.2 What is a Point Cloud</b> .....	267
<b>7.3 Point Cloud Filtering</b>	
7.3.1 Point Cloud Filtering.....	272
7.3.2 Segmentation in Perception – RANSAC.....	277
7.3.3 Extracting Indices.....	280
7.3.4 Outlier Removal Filter.....	281
7.3.5 <b>Summary</b> .....	282
<b>7.4 Object Segmentation</b>	
7.4.1 <b>Introduction</b> .....	283
7.4.2 Downside of Model Fitting.....	283
7.4.3 Clustering.....	283
7.4.4 K-means Clustering.....	286
7.4.5 DBSCAN Algorithm and comparing it to K-means.....	290
7.4.6 <b>Summary</b> .....	294
<b>7.5 Object Recognition</b>	
7.5.1 <b>Intro to Object Recognition</b> .....	295
7.5.2 Features.....	296
7.5.3 Color Spaces.....	297
7.5.4 Color Histograms.....	301
7.5.5 Surface Normals.....	303
7.5.6 Support Vector Machine.....	305
7.5.7 <b>Summary</b> .....	308
<b>8. Kinematics of Serial Manipulators</b>	
<b>8.1 Intro to Kinematics</b>	
8.1.1 <b>Overview</b> .....	309
8.1.2 Degrees of Freedom.....	310
8.1.3 Two DoF Arm.....	311

8.1.4 Generalized Coordinates.....	313
8.1.5 Rigid Bodies in Free Space.....	314
8.1.6 Joint Types.....	316
8.1.7 Principal Types of Serial Manipulators.....	318
8.1.8 Serial Manipulators Applications.....	319
<b>8.2 Forward and Inverse Kinematics</b>	
8.2.1 Setting up the Problem.....	324
8.2.2 Coordinate Frames and Vectors.....	324
8.2.3 Rotation Matrices.....	325
8.2.4 Composition of Rotations.....	328
8.2.5 Euler Angles from a Rotation Matrix.....	335
8.2.6 Translations.....	335
8.2.7 Homogeneous Transforms and their Inverse.....	336
8.2.8 Composition of Homogeneous Transforms.....	339
8.2.9 Devanit-Hartenberg Parameters.....	342
8.2.10 DH Parameter Assignment Algorithm.....	345
8.2.11 DH Steps and Parameter Table.....	346
8.2.12 Forward and Inverse Kinematics.....	352
<b>8.3 Summary</b> .....	<b>359</b>

## Appendix Experimental Results

### 9. Setting up the ROS Prerequisites

9.1 URDF of the robot.....	360
9.2 Spawning the robot in simulation.....	367

### 10. Localization results

#### 10.1 Kalman Filters

10.1.1 Programming Multidimensional KF in C++.....	371
10.1.2 Implement EKF package in ROS – Sensor Fusion.....	372

#### 10.2 Monte Carlo Localization

10.2.1 Programming Monte Carlo Localization in C++.....	376
10.2.2 Using the Adaptive Monte Carlo Localization with Ros.....	382

### 11. SLAM – gmapping – RTAB-Map Autonomous SLAM

11.1 Programming Occupancy Grid Mapping algorithm in C++.....	392
11.2 Grid-based FastSLAM – gmapping ROS package results.....	395
11.3 Real-Time Appearance-based Mapping, RTAB-Map package—experimental results....	401

### 12. Path Planning – The Navigation Stack and Move Base

12.1 Programming A* in C++.....	416
12.2 The Navigation Stack in ROS.....	421
12.3 Move Base.....	422
12.4 Costmap_2d.....	423
12.5 Configuring the costmaps – global_costmap and local_costmap and local planner.....	426
12.6 Setting up rviz for the navigation stack.....	433
12.7 Modifying parameters with rqt_reconfigure.....	441
12.8 Avoiding obstacles.....	442
12.9 Sending goals programmatically.....	443
<b>12.10 Summary</b> .....	<b>444</b>

### 13. 3D Perception – Experimental Results

13.1 Point Cloud Pre-processing Filtering.....	445
--	-----

13.2 Point Cloud Segmentation.....	449
13.3 Point Cloud Object Recognition.....	452
<b>14. Kinematics – experimental results</b>	
14.1 Creating an IK server for the KUKA KR210.....	460
14.2 Controlling a real robotic arm with MoveIT.....	478
14.3 Hardware Interface – ros control.....	482
14.4 Controlling the arm with MotionPlanning RVIZ display.....	483
<b>15. Project Github Repository.....</b>	<b>487</b>
<b>16. Extra Projects.....</b>	<b>487</b>
<b>17. References.....</b>	<b>487</b>

# 1. Types of Robotic Vehicles

## 1.1 Holonomic and Non - Holonomic Drive

### Robot Locomotion

Robots can be either mobile or stationary. Mobile robots include rolling robots, crawling robots, swimming robots and many more. Stationary robots include robot arm, robot face, industrial robots etc. Although known as stationary, these robots are not actually motionless, but are confined to a small boundary. Each of these robots are designed to work on different platforms and the most common ones work either on Land, Air, Water, space etc. Some of the robots are designed to work on more than one platform and can shift from land to water to air. Based on the way robots move, they can be further classified as "**Holonomic**" or "**Non-Holonomic**" drive Robots

### Holonomic Drive

Holonomic refers to the relationship between controllable and total degrees of freedom of a robot. If the controllable degree of freedom is equal to total degrees of freedom, then the robot is said to be Holonomic. A robot built on castor wheels or Omni-wheels is a good example of Holonomic drive as it can freely move in any direction and the controllable degrees of freedom is equal to total degrees of freedom. The image shows a castor wheel which can rotate in both X-axis and Y-axis making it move in both the directions.



### Non-Holonomic Drive

If the controllable degree of freedom is less than the total degrees of freedom, then it is known as non-Holonomic drive. A car has three degrees of freedom; i.e. its position in two axes and its orientation. However, there are only two controllable degrees of freedom which are acceleration (or braking) and turning angle of steering wheel. This makes it difficult for the driver to turn the car in any direction (unless the car skids or slides).

### Redundant Drive

What if the controllable degrees of freedom are more than the total degrees of freedom? Then the controls are considered to be **redundant**. A robot arm or even a human arm has only six degrees of freedom, but seven controllable degrees of freedom. (Try twisting and rotating your arm and find out what are the seven degrees of freedom, including shoulder, elbow and wrist).

## 1.2 Differential - Non Holonomic Drive

A differential wheeled robot is a mobile robot whose movement is based on two separately driven wheels placed on either side of the robot body. It can thus change its direction by varying the relative rate of rotation of its wheels and hence does not require an additional steering motion.

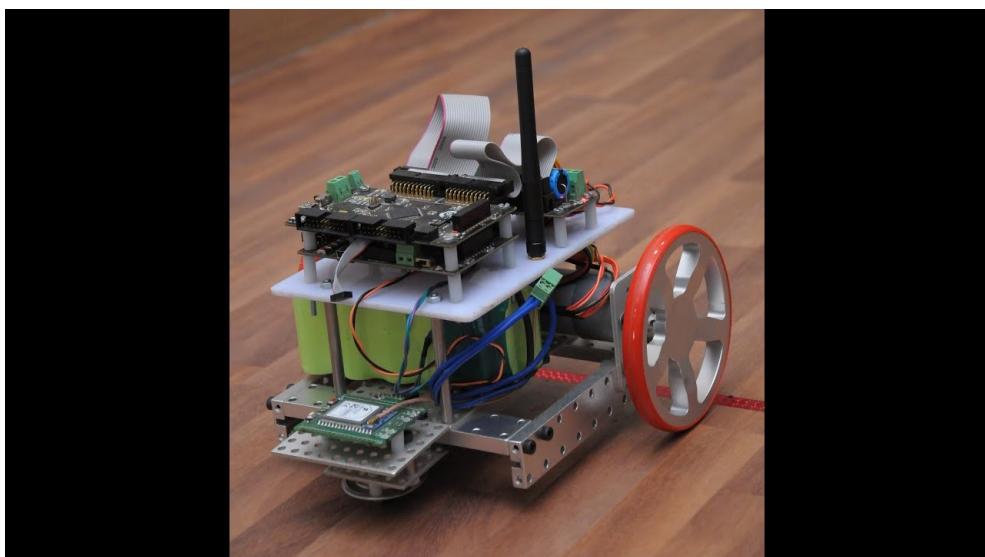
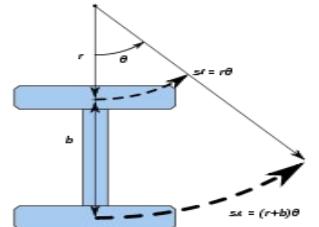
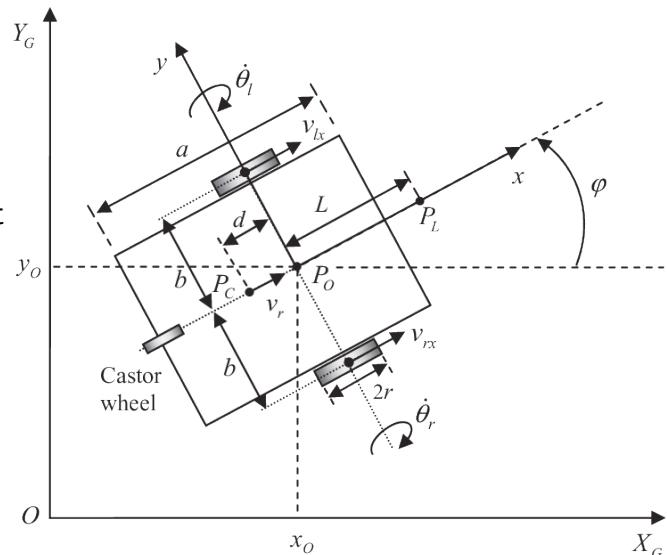
To balance the robot, additional wheels or casters may be added.

### Details

If both the wheels are driven in the same direction and speed, the robot will go in a straight line. If both wheels are turned with equal speed in opposite directions, as

is clear from the diagram shown, the robot will rotate about the central point of the axis. Otherwise, depending on the speed of rotation and its direction, the center of rotation may fall anywhere on the line defined by the two contact points of the tires. While the robot is traveling in a straight line, the center of rotation is an infinite distance from the robot. Since the direction of the robot is dependent on the rate and direction of rotation of the two driven wheels, these quantities should be sensed and controlled precisely.

A differentially steered robot is similar to the differential gears used in automobiles in that both the wheels can have different rates of rotations, but unlike the differential gearing system, a differentially steered system will have both the wheels powered. Differential wheeled robots are used extensively in robotics, since their motion is easy to program and can be well controlled. Virtually all consumer robots on the market today use differential steering primarily for its low cost and simplicity.



## 1.3 Omni-Directional – Mecanum Drive – Holonomic Drive

A robot is **holonomic** if all the constraints that it is subjected to are integrable into positional constraints of the form:

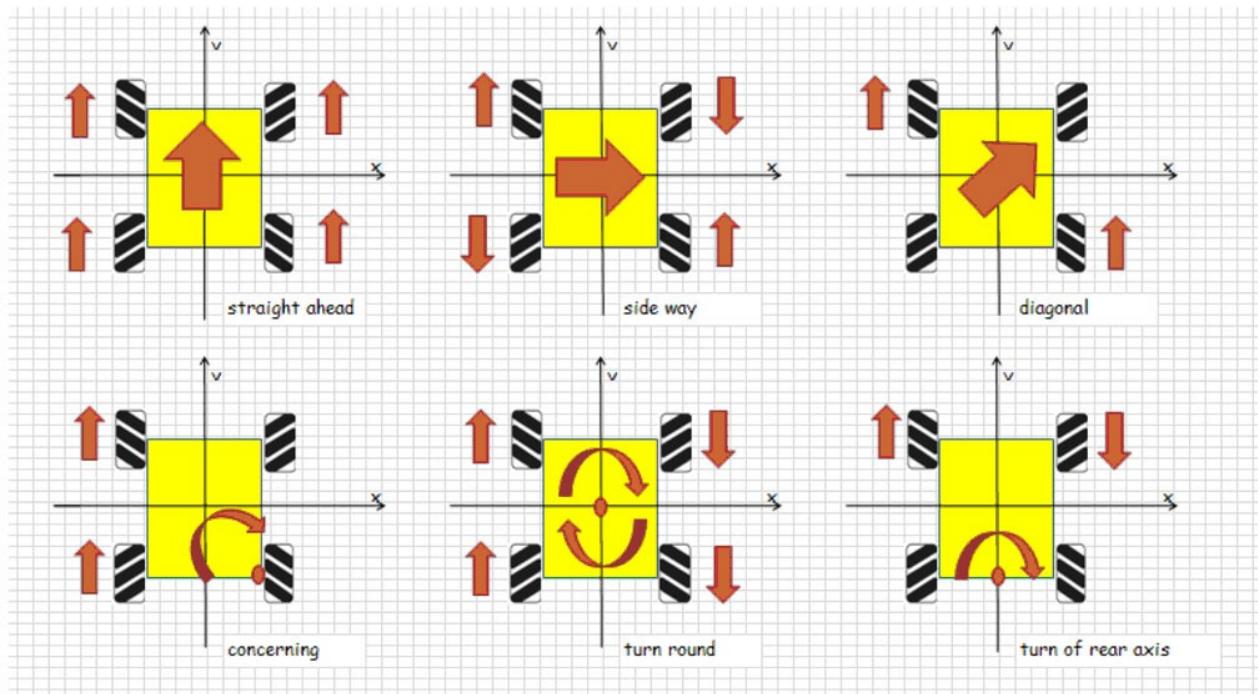
$$f(q_1, q_2, \dots, q_n; t) = 0$$

The variables  $q_i$  are the system coordinates. When a system contains constraints that cannot be written in this form, it is said to be nonholonomic.

In simpler terms, a holonomic system is when the number of controllable degrees of freedom is equal to the total degrees of freedom.



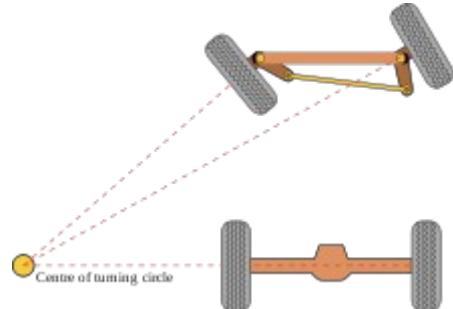
**Mecanum** drive is a type of holonomic drive base; meaning that it applies the force of the wheel at a  $45^\circ$  angle to the robot instead of on one of its axes. By applying the force at an angle to the robot, you can vary the magnitude of the force vectors to gain translational control of the robot; In plain English, the robot can move in any direction while keeping the front of the robot in a constant compass direction. The figure below shows the motions that can be achieved for various combination of wheel rotation.



## 1.4 Ackermann steering geometry - Cars

**Ackermann steering geometry** is a geometric arrangement of linkages in the steering of a car or other vehicle designed to solve the problem of wheels on the inside and outside of a turn needing to trace out circles of different radii.

It was invented by the German carriage builder Georg Lankensperger in Munich in 1817, then patented by his agent in England, Rudolph Ackermann (1764-1834) in 1818 for horse-drawn carriages. Erasmus Darwin may have a prior claim as the inventor dating from 1758.



### Advantages

The intention of Ackermann geometry is to avoid the need for tyres to slip sideways when following the path around a curve. The geometrical solution to this is for all wheels to have their axles arranged as radii of circles with a common centre point. As the rear wheels are fixed, this centre point must be on a line extended from the rear axle. Intersecting the axes of the front wheels on this line as well requires that the inside front wheel be turned, when steering, through a greater angle than the outside wheel.

Rather than the preceding "turntable" steering, where both front wheels turned around a common pivot, each wheel gained its own pivot, close to its own hub. While more complex, this arrangement enhances controllability by avoiding large inputs from road surface variations being applied to the end of a long lever arm, as well as greatly reducing the fore-and-aft travel of the steered wheels. A linkage between these hubs pivots the two wheels together, and by careful arrangement of the linkage dimensions the Ackermann geometry could be approximated. This was achieved by making the linkage *not* a simple parallelogram, but by making the length of the track rod (the moving link between the hubs) shorter than that of the axle, so that the steering arms of the hubs appeared to "toe out". As the steering moved, the wheels turned according to Ackermann, with the inner wheel turning further. If the track rod is placed ahead of the axle, it should instead be longer in comparison, thus preserving this same "toe out".

### Forward and Inverse Kinematics

**It is advised here to search and study the Forward Kinematics and Inverse Kinematics for both Differential, Mecanum and Ackermann Drive Robots that for brevity are not included here.**

## 1.5 My Robot - Mobile Manipulator

Our robot is of the type Differential Drive with 4 Rubber Wheels shown in the picture below, although an effort was made to 3d - print mecanum wheels this was not effective due to the weight of the robot and the small radius of the wheels. In the future I might include 100mm steel mecanum wheels.

It is equipped with multiple sensors, alongside a 6 DOF robotic arm for manipulation, more about the hardware used will be discussed in the next chapter.



## 2. Hardware and Sensors Used

### 2.1 Introduction

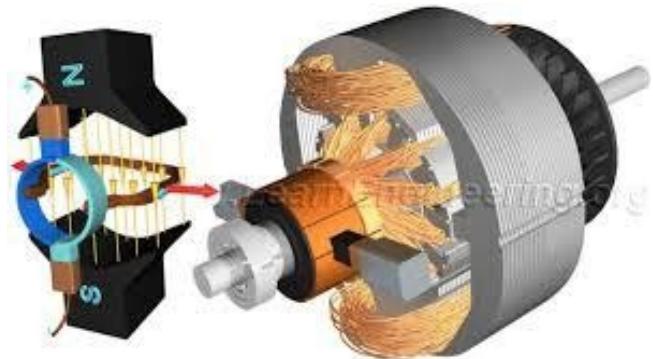
In this chapter we will briefly introduce the main hardware components that are used to any DIY robotic platform, you will become familiar with how they work and how you can control them and integrate them.

A brief list of the hardware that I used to build my robot alongside an indicative cost is presented below.

1. 4 x DC geared motor with encoders ~ 150
2. 4 x L298N motor drivers ~ 20
3. 1 arduino due ~ 30
4. Lipo 5000 mah battery ~ 50
5. Jetson Nano embedded computer ~ 100
6. 2D lidar ~ 100
7. Kinect v1 3D camera ~ 50
8. 3 x DC-DC Boost converters ~ 30
9. I2C servo driver ~ 30
10. DIY 6 DOF Robotic arm (no feedback) ~ 200
11. Total cost 760 - 1000

### 2.2 DC Motor

#### The Basic DC Motor



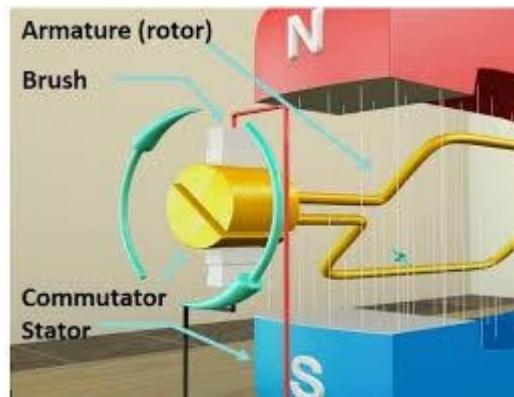
The **DC Motor** or **Direct Current Motor** to give it its full title, is the most commonly used actuator for producing continuous movement and whose speed of rotation can easily be controlled, making them ideal for use in applications where speed control, servo type control, and/or positioning is required. A DC motor consists of two parts, a "Stator" which is the stationary part and a "Rotor" which is the rotating part. The result is that there are basically three types of DC Motor available.

- **Brushed Motor** - This type of motor produces a magnetic field in a wound rotor (the part that rotates) by passing an electrical current through a commutator and carbon brush assembly, hence the term "Brushed". The stators (the stationary part) magnetic field is produced by using either a wound stator field winding or by permanent magnets. **Generally brushed DC motors are cheap, small and easily controlled.**
- **Brushless Motor** - This type of motor produces a magnetic field in the rotor by using permanent magnets attached to it and commutation is achieved electronically. They are generally smaller but more expensive than conventional brushed type DC motors because they use "Hall effect" switches in the stator to produce the required stator field rotational sequence but they have better

- torque/speed characteristics, are more efficient and have a longer operating life than equivalent brushed types.
- **Servo Motor** - This type of motor is basically a brushed DC motor with some form of positional feedback control connected to the rotor shaft. They are connected to and controlled by a PWM type controller and are mainly used in positional control systems and radio controlled models.

## The Brushed DC Motor

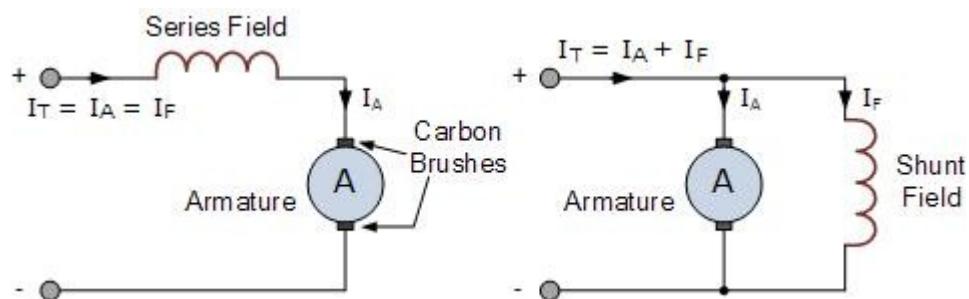
A conventional brushed DC Motor consist basically of two parts, the stationary body of the motor called the Stator and the inner part which rotates producing the movement called the Rotor or "Armature" for DC machines.



The motors wound stator is an electromagnet circuit which consists of electrical coils connected together in a circular configuration to produce the required North-pole then a South-pole then a North-pole etc, type stationary magnetic field system for rotation, unlike AC machines whose stator field continually rotates with the applied frequency. The current which flows within these field coils is known as the motor field current.

These electromagnetic coils which form the stator field can be electrically connected in series, parallel or both together (compound) with the motors armature. A series wound DC motor has its stator field windings connected in *series* with the armature. Likewise, a shunt wound DC motor has its stator field windings connected in *parallel* with the armature as shown.

## Series and Shunt Connected DC Motor

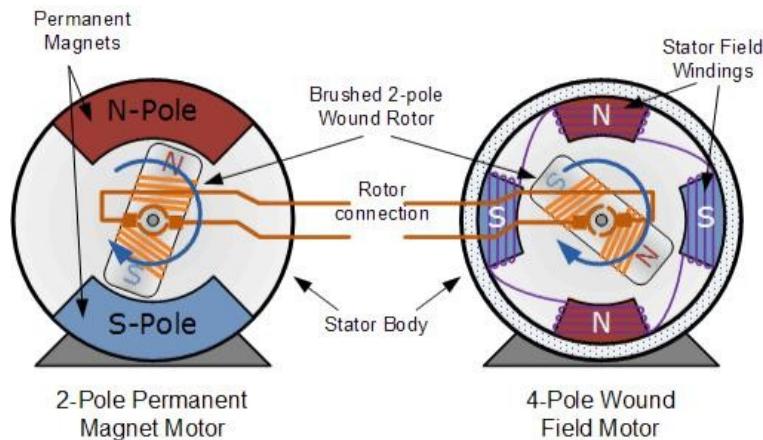


The rotor or armature of a DC machine consists of current carrying conductors connected together at one end to electrically isolated copper segments called the commutator. The commutator allows an electrical connection to be made via carbon brushes (hence the name "Brushed" motor) to an external power supply as the armature rotates.

The magnetic field setup by the rotor tries to align itself with the stationary stator field causing the rotor to rotate on its axis, but can not align itself due to commutation delays. The rotational speed of the motor is dependent on the strength of the rotors magnetic field and the more voltage that is applied to the motor the faster the rotor will rotate. By

varying this applied DC voltage the rotational speed of the motor can also be varied.

## Conventional Brushed DC Motor



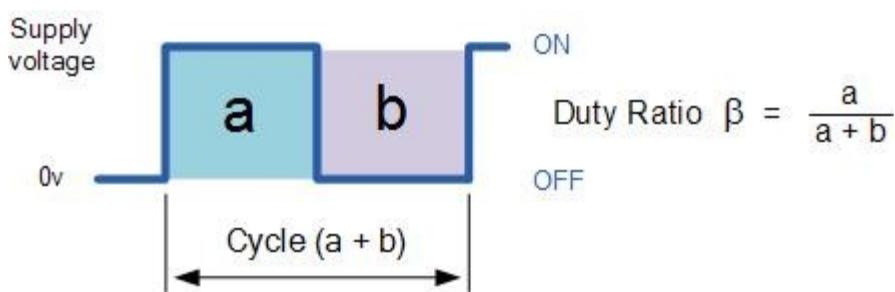
The Permanent magnet (PMDC) brushed DC motor is generally much smaller and cheaper than its equivalent wound stator type DC motor cousins as they have no field winding. In permanent magnet DC (PMDC) motors these field coils are replaced with strong rare earth (i.e. Samarium Cobalt, or Neodymium Iron Boron) type magnets which have very high magnetic energy fields.

The use of permanent magnets gives the DC motor a much better linear speed/torque characteristic than the equivalent wound motors because of the permanent and sometimes very strong magnetic field, making them more suitable for use in models, robotics and servos.

Although DC brushed motors are very efficient and cheap, problems associated with the brushed DC motor is that sparking occurs under heavy load conditions between the two surfaces of the commutator and carbon brushes resulting in self generating heat, short life span and electrical noise due to sparking, which can damage any semiconductor switching device such as a MOSFET or transistor.

## Pulse Width Speed Control

The rotational speed of a DC motor is directly proportional to the mean (average) voltage value on its terminals, and the higher this value, up to maximum allowed motor volts, the faster the motor will rotate. In other words more voltage more speed. By varying the ratio between the "ON" ( $t_{ON}$ ) time and the "OFF" ( $t_{OFF}$ ) time durations, called the "Duty Ratio", "Mark/Space Ratio" or "Duty Cycle", the average value of the motor voltage and hence its rotational speed can be varied. For simple unipolar drives the duty ratio  $\beta$  is given as:

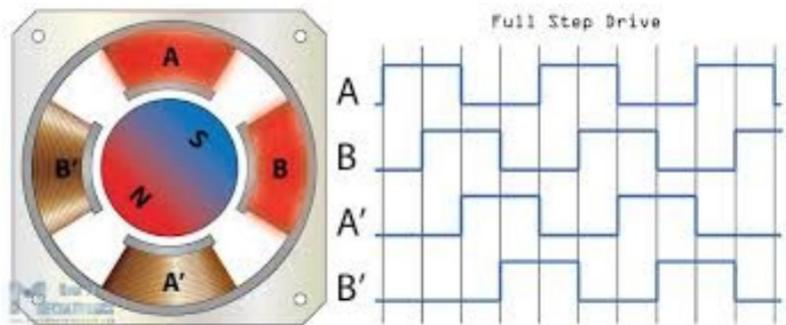
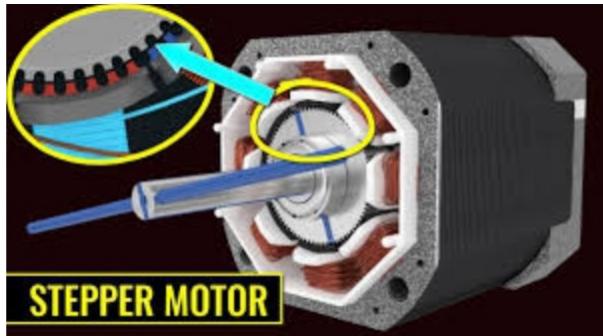


and the mean DC output voltage fed to the motor is given as:  $V_{mean} = \beta \times V_{supply}$ . Then by varying the width of pulse  $\alpha$ , the motor voltage and hence the power applied to the motor can be controlled and this type of control is called Pulse Width Modulation or PWM. Another way of controlling the rotational speed of the motor is to vary the frequency (and hence the time period of the controlling voltage) while the “ON” and “OFF” duty ratio times are kept constant. This type of control is called Pulse Frequency Modulation or PFM.

With pulse frequency modulation, the motor voltage is controlled by applying pulses of variable frequency for example, at a low frequency or with very few pulses the average voltage applied to the motor is low, and therefore the motor speed is slow. At a higher frequency or with many pulses, the average motor terminal voltage is increased and the motor speed will also increase.

Then, Transistors can be used to control the amount of power applied to a DC motor with the mode of operation being either “Linear” (varying motor voltage), “Pulse Width Modulation” (varying the width of the pulse) or “Pulse Frequency Modulation” (varying the frequency of the pulse).

## 2.3 STEPPER MOTOR



Like the DC motor above, Stepper Motors are also electromechanical actuators that convert a pulsed digital input signal into a discrete (incremental) mechanical movement are used widely in industrial control applications. A stepper motor is a type of synchronous brushless motor in that it does not have an armature with a commutator and carbon brushes but has a rotor made up of many, some types have hundreds of permanent magnetic teeth and a stator with individual windings.

As its name implies, the stepper motor does not rotate in a continuous fashion like a conventional DC motor but moves in discrete "Steps" or "Increments", with the angle of each rotational movement or step dependant upon the number of stator poles and rotor teeth the stepper motor has. Because of their discrete step operation, stepper motors can easily be rotated a finite fraction of a rotation at a time, such as 1.8, 3.6, 7.5 degrees etc. So for example, let's assume that a stepper motor

completes one full revolution ( $360^{\circ}$ ) in exactly 100 steps.

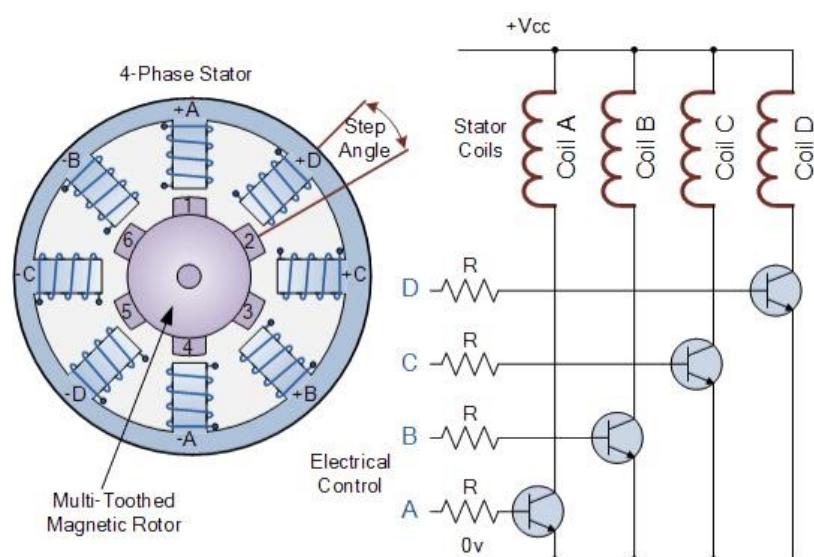
Then the step angle for the motor is given as  $360 \text{ degrees}/100 \text{ steps} = 3.6 \text{ degrees per step}$ . This value is commonly known as the stepper motor's **Step Angle**.

There are three basic types of stepper motor, **Variable Reluctance**, **Permanent Magnet** and **Hybrid** (a sort of combination of both). A **Stepper Motor** is particularly well suited to applications that require accurate positioning and repeatability with a fast response to starting, stopping, reversing and speed control and another key feature of the stepper motor, is its ability to hold the load steady once the required position is achieved.

Generally, stepper motors have an internal rotor with a large number of permanent magnet "teeth" with a number of electromagnet "teeth" mounted on to the stator. The stator's electromagnets are polarized and depolarized sequentially, causing the rotor to rotate one "step" at a time.

Modern multi-pole, multi-teeth stepper motors are capable of accuracies of less than 0.9 degs per step (400 Pulses per Revolution) and are mainly used for highly accurate positioning systems like those used for magnetic-heads in floppy/hard disc drives, printers/plotters or robotic applications. The most commonly used stepper motor being the 200 step per revolution stepper motor. It has a 50 teeth rotor, 4-phase stator and a step angle of 1.8 degrees ( $360 \text{ degs}/(50 \times 4)$ ).

## Stepper Motor Construction and Control



In our simple example of a variable reluctance stepper motor above, the motor consists of a central rotor surrounded by four electromagnetic field coils labelled A, B, C and D. All the coils with the same letter are connected together so that energising, say coils marked A will cause the magnetic rotor to align itself with that set of coils.

By applying power to each set of coils in turn the rotor can be made to rotate or "step" from one position to the next by an angle determined by its step angle construction, and by energising the coils in sequence the rotor will produce a rotary motion.

The stepper motor driver controls both the step angle and speed of the motor by energising the field coils in a set sequence for example, "ADCB, ADCB, ADCB, A..." etc, the rotor will rotate in one direction (forward) and by reversing the pulse sequence to "ABCD, ABCD, ABCD, A..." etc, the rotor will rotate in the opposite direction (reverse).

So in our simple example above, the stepper motor has four coils, making it a 4-phase motor, with the number of poles on the stator being eight ( $2 \times 4$ ) which are spaced at 45 degree intervals. The number of teeth on the rotor is six which are spaced 60 degrees apart.

Then there are 24 (6 teeth  $\times$  4 coils) possible positions or "steps" for the rotor to complete one full revolution. Therefore, the step angle above is given as:  $360^\circ/24 = 15^\circ$ .

Obviously, the more rotor teeth and or stator coils would result in more control and a finer step angle. Also by connecting the electrical coils of the motor in different configurations, Full, Half and micro-step angles are possible. However, to achieve micro-stepping, the stepper motor must be driven by a (quasi) sinusoidal current that is expensive to implement.

It is also possible to control the speed of rotation of a stepper motor by altering the time delay between the digital pulses applied to the coils (the frequency), the longer the delay the slower the speed for one complete revolution. By applying a fixed number of pulses to the motor, the motor shaft will rotate through a given angle.

**The advantage of using time delayed pulse is that there would be no need for any form of additional feedback because by counting the number of pulses given to the motor the final position of the rotor will be exactly known.** This response to a set number of digital input pulses allows the stepper motor to operate in an "**Open Loop System**" making it both easier and cheaper to control.

For example, lets assume that our stepper motor above has a step angle of 3.6 degs per step. To rotate the motor through an angle of say 216 degrees and then stop again at the require position would only need a total of:  $216 \text{ degrees}/(3.6 \text{ degs/step}) = 60 \text{ pulses}$  applied to the stator coils.

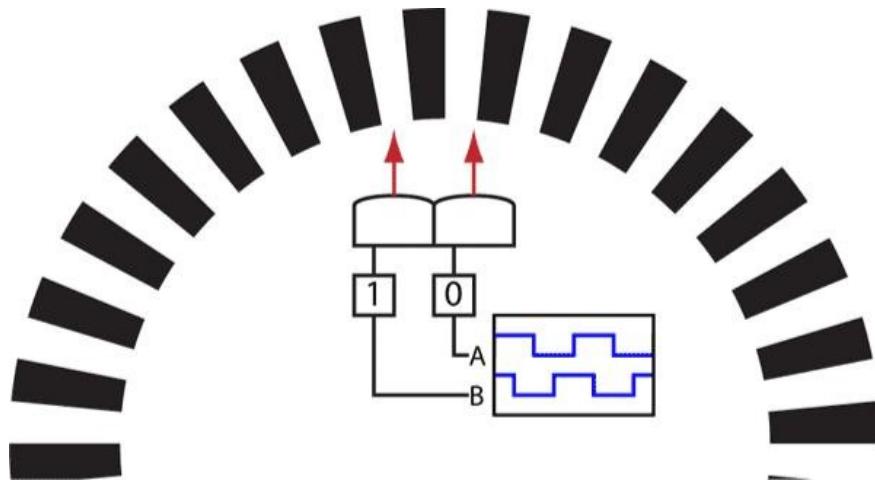
There are many stepper motor controller IC's available which can control the step speed, speed of rotation and motors direction. One such controller IC is the SAA1027 which has all the necessary counter and code conversion built-in, and can automatically drive the 4 fully controlled bridge outputs to the motor in the correct sequence.

The direction of rotation can also be selected along with single step mode or continuous (stepless) rotation in the selected direction, but this puts some burden on the controller. When using an 8-bit digital controller, 256 microsteps per step are also possible

## 2.4 Quadrature encoder

Quadrature Encoders are handy sensors that let you measure the speed and direction of a rotating shaft (or linear motion) and keep track of how far you have moved. They usually use magnetism or light to calculate wheel rotation.

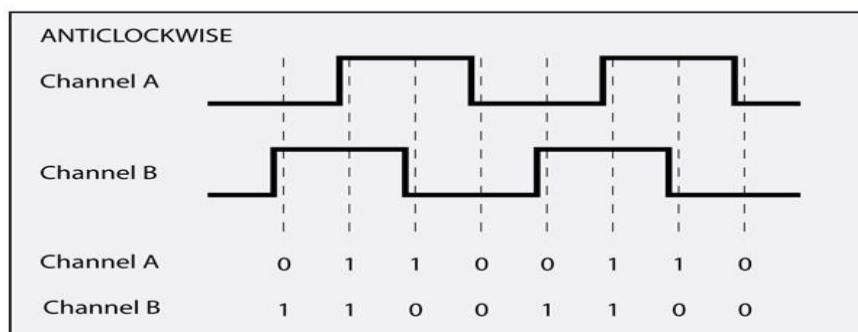
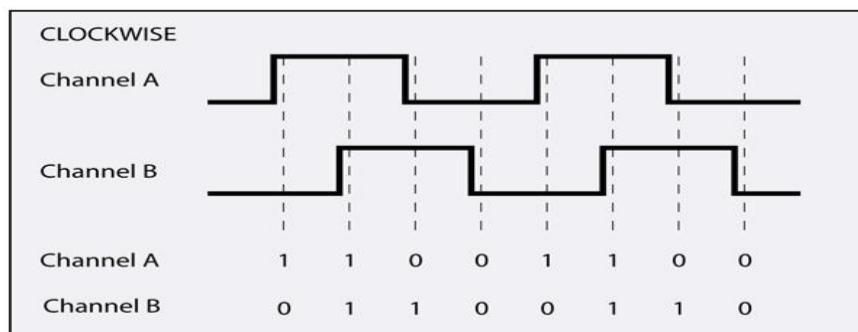
A quadrature encoder normally has at least two outputs - Channel A and B - each of which will produce digital pulses when the thing they are measuring is in motion. These pulses will follow a particular pattern that allows you to tell which direction the thing is moving, and by measuring the time between pulses, or the number of pulses per second, you can also derive the speed.



They have a black and white reflective code wheel inside the wheel rim, and the PCB has a pair of reflective sensors that point at the code wheel. These sensors have just the right spacing between them, and relative to the stripes on the wheel, to produce the pattern of pulses you see in the picture.

In technical terms these pulses are 90 degrees out of phase meaning that one pulse always leads the other pulse by one quarter of a complete cycle (a cycle is a complete transition from low  $\rightarrow$  high  $\rightarrow$  low again).

The order in which these pulses occur will change when the direction of rotation changes. The two diagrams below show what the two pulse patterns look like for clockwise and counter clockwise rotation.



So how do you actually work out the direction?  
Let's start with channel A at the top (clockwise rotation):

1 Channel A goes from LOW to HIGH

2 Channel B is LOW

=>

Clockwise motion

1 Channel A goes from LOW to HIGH

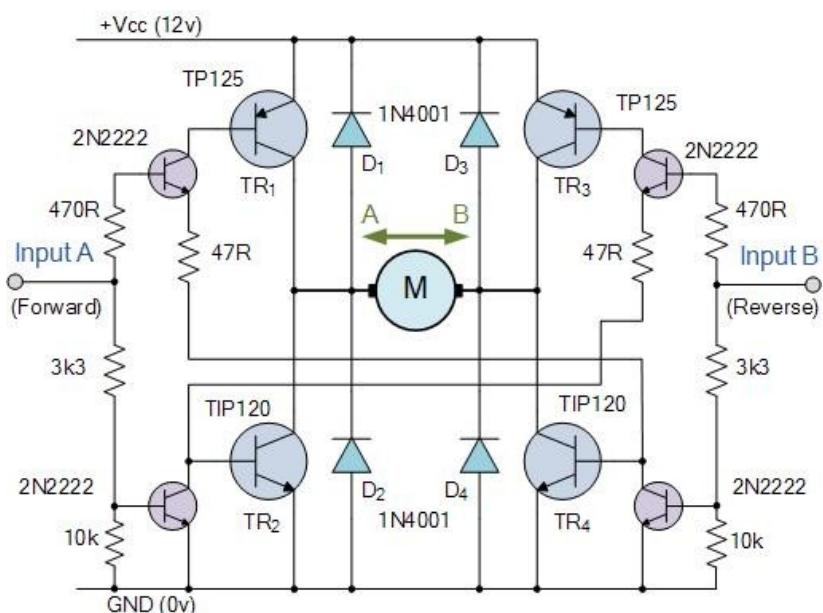
2 Channel B is HIGH

=>

Counter-Clockwise motion

## 2.5 L298N DUAL H-Bridge Motor Driver

### Basic Bi-directional H-bridge Circuit



The **H-bridge circuit** above, is so named because the basic configuration of the four switches, either electro- mechanical relays or transistors resembles that of the letter "H" with the motor positioned on the centre bar. The Transistor or MOSFET H-bridge is probably one of the most commonly used type of bi-directional DC motor control circuits. It uses "complementary transistor pairs" both NPN and PNP in each branch with the transistors being switched together in pairs to control the motor.

Control input A operates the motor in one direction ie, Forward rotation while input B operates the motor in the other direction ie, Reverse rotation. Then by switching the transistors "ON" or "OFF" in their "diagonal pairs" results in directional control of the motor.

For example, when transistor TR1 is “ON” and transistor TR2 is “OFF”, point A is connected to the supply voltage (+Vcc) and if transistor TR3 is “OFF” and transistor TR4 is “ON” point B is connected to 0 volts (GND). Then the motor will rotate in one direction corresponding to motor terminal A being positive and motor terminal B being negative.

If the switching states are reversed so that TR1 is “OFF”, TR2 is “ON”, TR3 is “ON” and TR4 is “OFF”, the motor current will now flow in the opposite direction causing the motor to rotate in the opposite direction.

Then, by applying opposite logic levels “1” or “0” to the inputs A and B the motors rotational direction can be controlled as follows.

### H-bridge Truth Table

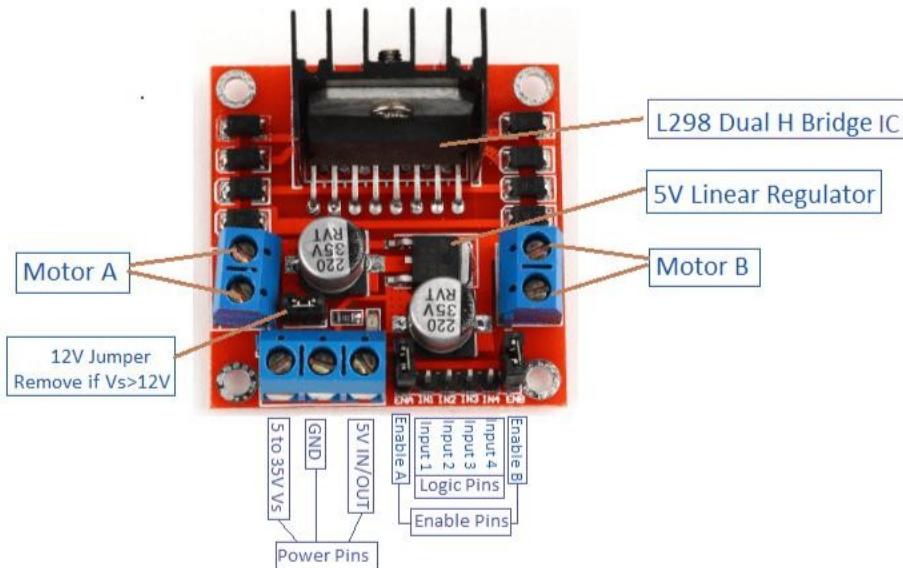
Input A	Input B	Motor Function
TR1 and TR4	TR2 and TR3	
0	0	Motor Stopped (OFF)
1	0	Motor Rotates Forward
0	1	Motor Rotates Reverse

**It is important that no other combination of inputs are allowed as this may cause the power supply to be shorted out, ie both transistors, TR1 and TR2 switched “ON” at the same time, (fuse = bang!).**

As with uni-directional DC motor control as seen above, the rotational speed of the motor can also be controlled using Pulse Width Modulation or PWM. Then by combining H-bridge switching with PWM control, both the direction and the speed of the motor can be accurately controlled.

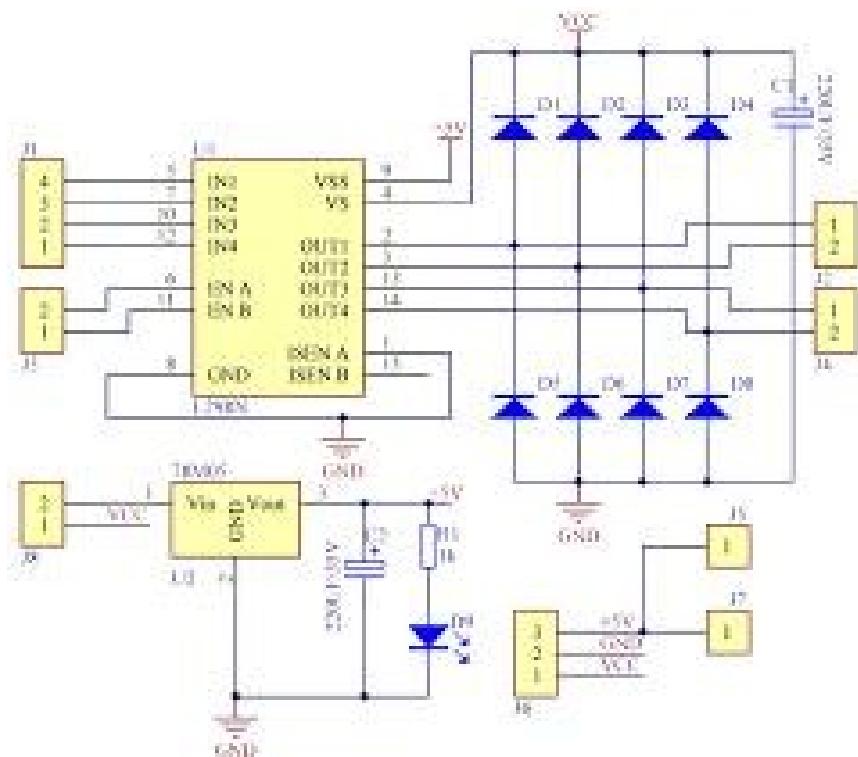
**Commercial off the shelf decoder IC's such as the SN754410 Quad Half H-Bridge IC or the L298N which has 2 H-Bridges are available with all the necessary control and safety logic built in are specially designed for H-bridge bi-directional motor control circuits.**

## L298N

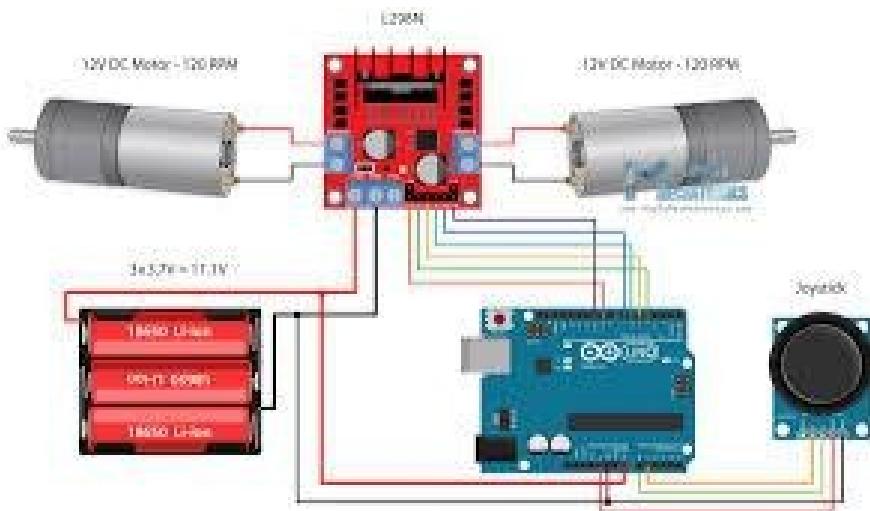


This dual bidirectional motor driver, is based on the very popular L298 Dual H-Bridge Motor Driver Integrated Circuit. The circuit will allow you to easily and independently control two motors of up to 2A each in both directions. It is ideal for robotic applications and well suited for connection to a microcontroller requiring just a couple of control lines per motor. It can also be interfaced with simple manual switches, TTL logic gates, relays, etc. This board equipped with power LED indicators, on-board +5V regulator and protection diodes.

## Schematic Diagram



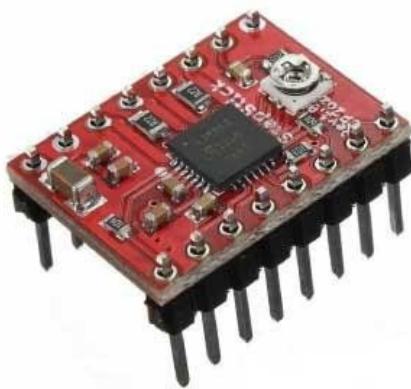
## Connection example



## 2. 6 A4988 driver

## Overview

The A4988 is a microstepping driver for controlling bipolar stepper motors which has built-in translator for easy operation. This means that we can control the stepper motor with just 2 pins from our controller, or one for controlling the rotation direction and the other for controlling the steps.



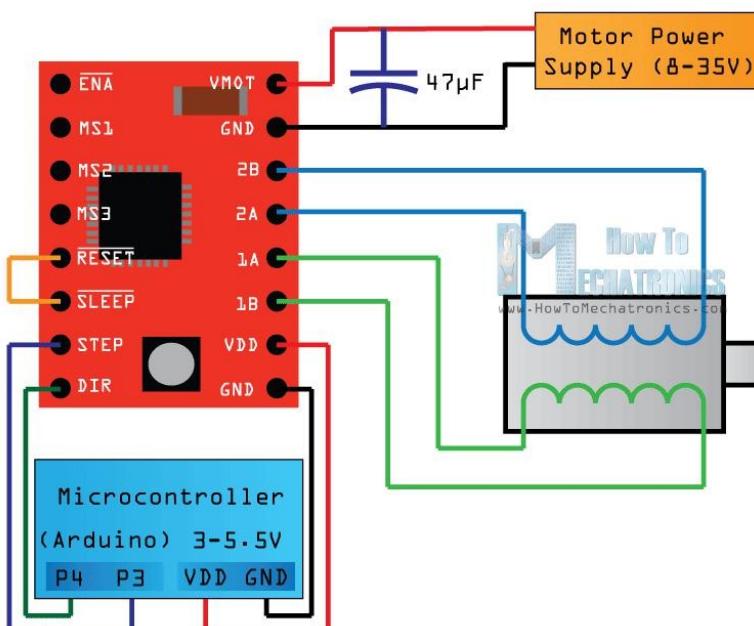
The Driver provides five different step resolutions: full-step, haft-step, quarter-step, eight-step and sixteenth-step. Also, it has a potentiometer for adjusting the current output, over-temperature thermal shutdown and crossover-current protection.

Its logic voltage is from 3 to 5.5 V and the maximum current per phase is 2A if good addition cooling is provided or 1A continuous current per phase without heat sink or cooling.

Minimum Logic Voltage:	3V
Maximum Logic Voltage:	5.5 V
Continuous current per phase:	1 A
Maximum current per phase:	2 A
Minimum Operating Voltage:	8 V
Maximum Operating Voltage:	35 V

### A4988 Stepper Driver Pinout

Now let's close look at the pinout of the driver and hook it up with the stepper motor and the controller. So we will start with the 2 pins on the button right side for powering the driver, the VDD



and Ground pins that we need to connect them to a power supply of 3 to 5.5 V and in our case that will be our controller, the Arduino Board which will provide 5 V. The following 4 pins are for connecting the motor. The 1A and 1B pins will be connected to one coil of the motor and the 2A and 2B pins to the other coil of the motor. For powering the motor we use the next 2 pins, Ground and VMOT that we need to connect them to Power Supply from 8 to 35 V and also we

need to use decoupling capacitor with at least 47  $\mu$ F for protecting the driver board from voltage spikes.

The next two 2 pins, Step and Direction are the pins that we actually use for controlling the motor movements. The Direction pin controls the rotation direction of the motor and we need to connect it to one of the digital pins on our microcontroller.

With the Step pin we control the mirosteps of the motor and with each pulse sent to this pin the motor moves one step. So that means that we don't need any complex programming, phase sequence tables, frequency control lines and so on, because the built-in translator of the A4988 Driver takes care of everything. Here we also need to mention that these 2 pins are not pulled to any voltage internally, so we should not leave them floating in our program.

Next is the SLEEP Pin and a logic low puts the board in sleep mode for minimizing power consumption when the motor is not in use.

Next, the RESET pin sets the translator to a predefined Home state. This Home state or Home Microstep Position can be seen from these Figures from the A4988 Datasheet. So these are the initial positions from where the motor starts and they are different depending on the microstep resolution. If the input state to this pin is a logic low all the STEP inputs will be ignored. The Reset pin is a floating pin so if we don't have intention of controlling it with in our program we need to connect it to the SLEEP pin in order to bring it high and enable the board

MS1	MS2	MS3	Resolution
LOW	LOW	LOW	Full Step
HIGH	LOW	LOW	Half Step
LOW	HIGH	LOW	Quarter Step
HIGH	HIGH	LOW	Eighth step
HIGH	HIGH	HIGH	Sixteenth Step

The next 3 pins (MS1, MS2 and MS3) are for selecting one of the five step resolutions according to the above truth table. These pins have internal pull-down resistors so if we leave them disconnected, the board will operate in full step mode.

The last one, the ENABLE pin is used for turning on or off the FET outputs. So a logic high will keep the outputs disabled.

## 2.7 Arduino Microcontroller



**Arduino** is an open source hardware and software company, project and user community that designs and manufactures single-board microcontrollers and microcontroller kits for building digital devices. Arduino boards are available commercially in preassembled form or as do-it yourself (DIY) kits.

Arduino board designs use a variety of microprocessors and controllers. The boards are equipped with sets of digital and analog input/output (I/O) pins that may be interfaced to various expansion boards or breadboards (*shields*) and other circuits. The boards feature serial communications interfaces, including Universal\_Serial\_Bus (USB) on some models, which are also used for loading programs from personal computers. The microcontrollers can be programmed using C and C++ programming languages. In addition to using traditional compiler toolchains, the Arduino project provides an integrated development environment (IDE) based on the Processing language project.

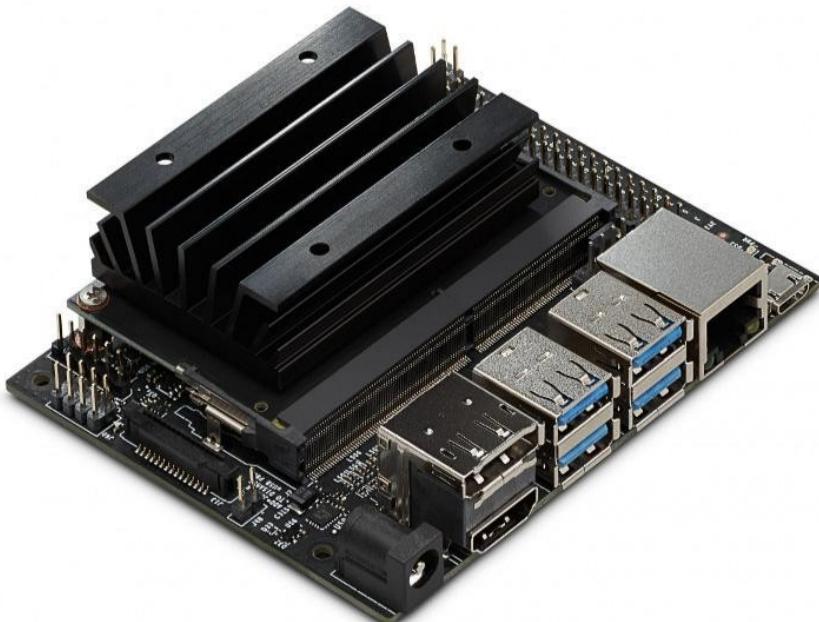
The Arduino project started in 2005 as a program for students at the Interaction Design Institute Ivrea in Ivrea, Italy, aiming to provide a low-cost and easy way for novices and professionals to create devices that interact with their environment using sensors and actuators. Common examples of such devices intended for beginner hobbyists include simple robots, thermostats and motion detectors.

The name *Arduino* comes from a bar in Ivrea, Italy, where some of the founders of the project used to meet. The bar was named after Arduin\_of\_Ivrea, who was the margrave of the March\_of\_Ivrea and King of Italy from 1002 to 1014.

In a later chapter we will see how to use an arduino due for controlling 4 DC motors with encoders using pid control. We will use **rosserial** in order to make a “pseudo” ROS node in the arduino for easy communication with the rest of our system.

Please note that since our embedded computer Jetson Nano provides I2c and gpio interfaces, the use of an arduino might be unnecessary for our project, especially when we use stepper motors, however arduino provides as well hard pwm and an easy way to read the encoder signals reliably.

## 2.8 Jetson Nano



**The brain of our robot is a small factor computer called Jetson Nano Developer Kit.** Jetson Nano runs full desktop Ubuntu out of the box, and is supported by the JetPack 4.2. NVIDIA Jetson Nano **targets AI projects such as mobile robots and drones, digital assistants, automated appliances and more.**

### Jetson Nano Developer kit specifications

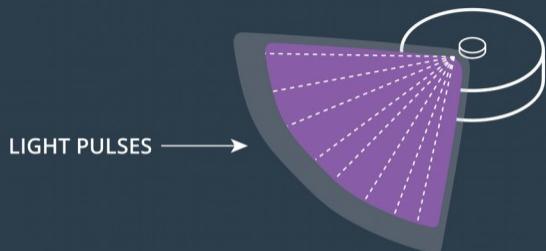
- Jetson Nano CPU Module
  - 128-core Maxwell GPU
  - Quad-core Arm A57 processor @ 1.43 GHz
  - System Memory – 4GB 64-bit LPDDR4 @ 25.6 GB/s
  - Storage – microSD card slot (devkit) or 16GB eMMC flash (production)
  - Video Encode – 4K @ 30 | 4x 1080p @ 30 | 9x 720p @ 30 (H.264/H.265)
  - Video Decode – 4K @ 60 | 2x 4K @ 30 | 8x 1080p @ 30 | 18x 720p @ 30 (H.264/H.265)
  - Dimensions – 70 x 45 mm
- Baseboard
  - 260-pin SO-DIMM connector for Jetson Nano module.
  - Video Output – HDMI 2.0 and eDP 1.4 (video only)
  - Connectivity – Gigabit Ethernet (RJ45) + 4-pin PoE header
  - USB – 4x USB 3.0 ports, 1x USB 2.0 Micro-B port for power or device mode
  - Camera I/F – 1x MIPI CSI-2 DPHY lanes compatible with Leopard Imaging LI-IMX219-MIPI-FF-NANO camera module and Raspberry Pi Camera Module V2
  - Expansion
    - M.2 Key E socket (PCIe x1, USB 2.0, UART, I2S, and I2C) for wireless networking cards
    - 40-pin expansion header with GPIO, I2C, I2S, SPI, UART signals
    - 8-pin button header with system power, reset, and force recovery related signals
  - Misc – Power LED, 4-pin fan header
  - Power Supply – 5V/4A via power barrel or 5V/2A via micro USB port; optional PoE support
  - Dimensions – 100 x 80 x 29 mm

## 2.9 2D Lidar



2D Laser Range Finders or 2D laser scanners are active sensors developed based on the Light Detection and Ranging (Lidar) method. Meaning, these sensors illuminate the target with a pulsed laser and measure the reflected pulses.

Since the laser frequency is a known stable quantity, the distance to objects in the field of view is calculated by measuring the time from when the pulse was sent to when it was received.



Pros	Cons
High spatial resolution in the horizontal plane	Large in size and bulky
High accuracy	High cost
High range	Affected by adverse weather

**YDLIDAR X4 lidar** is a 360-degree two-dimensional laser range scanner (LIDAR). This device uses triangulation principle to measure distance, together with the appropriate optical, electrical, algorithm design, to achieve high-precision distance measurement.

This will be the main sensor for building 2D maps, obstacle avoidance, SLAM and even as a reliable odometry source.

It essentially measure the distance to obstacles all around the robot in a plane. ROS provides a driver package that we will see later.

## Features

- 360-degree scanning distance measurement
- Small distance error; stable distance measurement and high accuracy
- Measuring distance: 0.12-10 m
- Resistance to ambient light interference
- Industrial grade motor drive for stable performance
- Laser-grade: Class I
- 360-degree omnidirectional scanning; 6-12 Hz adaptive scanning frequency
- Range finder frequency: 5000 times/s

## Applications

- Robot navigation and obstacle avoidance
- Robot ROS teaching and research
- Regional security
- Environmental Scan and 3D Reconstruction
- Home service robot/sweeping robot navigation and obstacle avoidance

Range Frequency 5000 Hz  
Scanning Frequency 6-12 Hz  
Range 0.12-10 m  
Scanning angle 0-360°  
Range resolution < 0.5 mm (Range < 2 m)  
                  < 1% of actual distance (Range > 2 m)  
Angle resolution 0.48-0.52°  
Supply Voltage 4.8-5.2 V  
Voltage ripple 0-100 mV  
Starting current 400-480 mA  
Sleep current 280-340 mA  
Working current 330-380 mA  
Baud rate 128000 bps  
Signal high 1.8-3.5 V  
Signal low 0-0.5 V  
PWM frequency 10 kHz  
Duty cycle range 50-100 %  
Laser wavelength 775-795 nm  
Laser Power 3 mW  
FDA Class I  
Working temperature 0-40 °C  
Lighting environment 0-2000 Lux  
Weight 189 g  
Size 102 x 71 x 63 mm

#### Interface

- UART: LVTTL Level standard
- Support scan frequency adjustable: Yes
- Support low power: Yes

#### Pin Configuration

VCC	Power Supply	5 V
Tx	Output	
Rx	Input	
GND	Ground	
M_EN	Motor Enable Control	3.3 V
DEV_EN	Ranging Enable Control	3.3 V
M_SCTP	Motor Speed Control	1.8 V
NC	Reserved Pin	

## 2.10 Kinect v1 Time of Flight camera



### Time of Flight camera

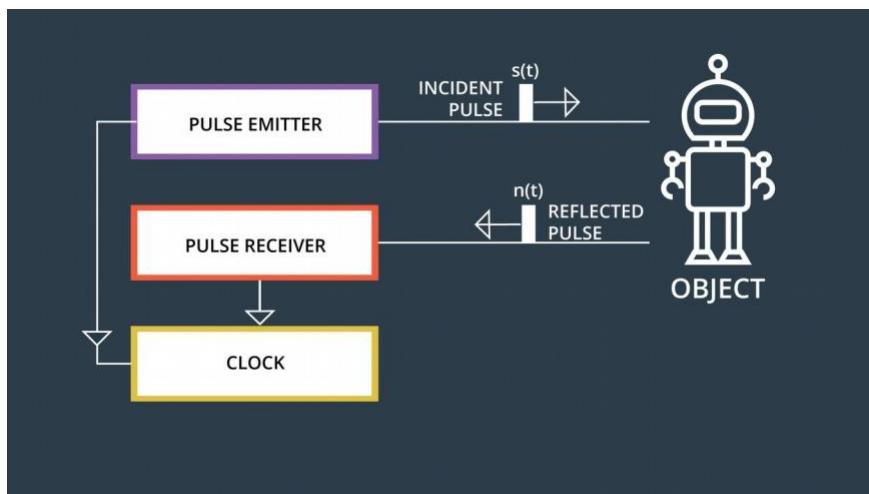
A 3D time of flight camera is an active sensor that performs depth measurements by illuminating an area with an infrared light source and observing the time it takes to travel to the scene and back.

However, unlike a Laser Range Finder, a ToF camera captures entire Field of View with each light pulse without any moving parts. This allows for rapid data acquisition.

Based on their working principle, ToF sensors can be divided into two categories; pulse runtime and phase shift continuous wave

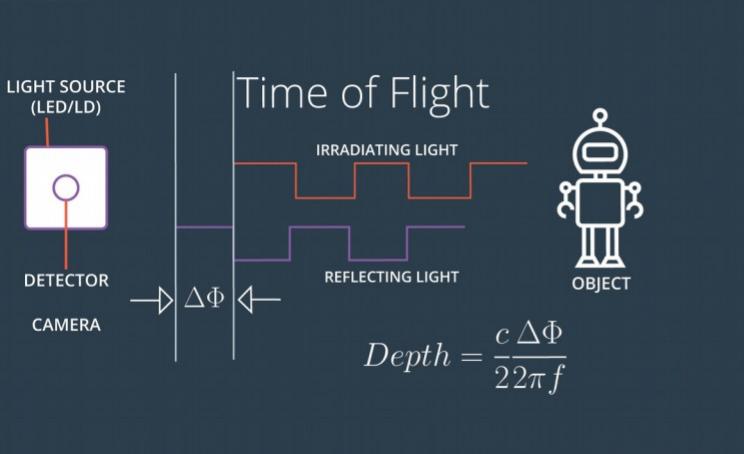
#### Pulse Runtime Sensor

A pulse runtime sensor sends out a pulse of light and starts a timer, then it waits until a reflection is detected and stops the timer, thus directly measuring the light's time of flight. Although intuitive and simple conceptually, this technique requires very accurate hardware for timing.



#### Phase Shift Continuous Wave Sensor

Unlike the Pulse runtime sensor, Phase Shift Continuous Wave sensor emits a continuous stream of modulated light waves. Here, the depth is calculated by measuring the phase shift of the reflected wave, creating a 3D depth map of the scene.

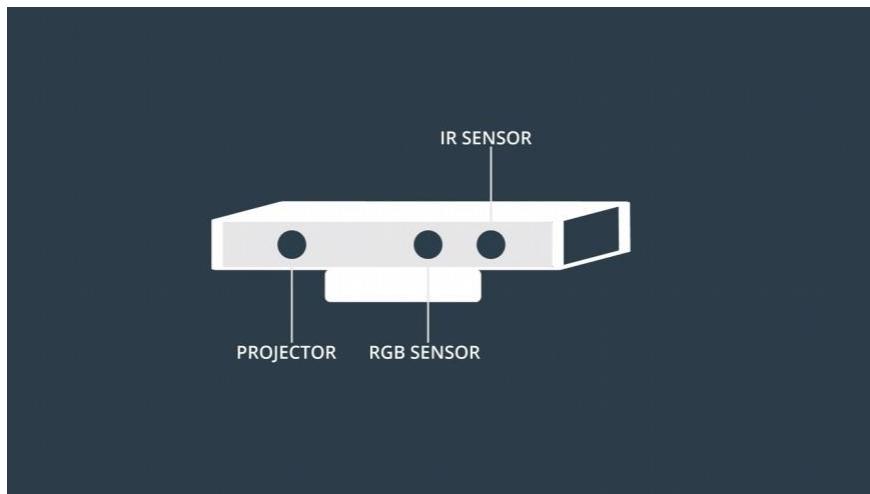


Pros	Cons
Compact size	Secondary reflections
Rapid data acquisition	Ambient light interference (do not work well outdoors)
Ideal for real-time applications	Multiple ToF camera may interfere with one another
	Cannot easily detect glass

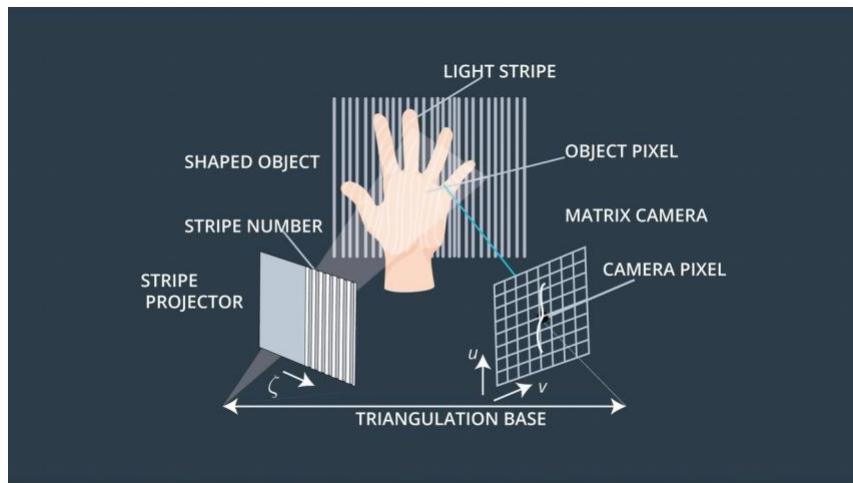
## RGB-D Camera

An RGB-D camera combines best of the active and passive sensor worlds, in that it consists of a passive RGB camera along with an active depth sensor. An RGB-D camera, unlike a conventional camera, provides per-pixel depth information in addition to an RGB image. Traditionally, the active depth sensor is an infrared (IR) projector and receiver. Much like a Continuous Wave Time of Flight sensor, an RGB-D camera calculates depth by emitting a light signal on the scene and analyzing the reflected light, but the incident wave modulation is performed spatially instead of temporally.

Here we can see an example of a standard RGB-D Camera:



This is done by projecting light out of the IR transmitter in a predefined pattern and calculating the depth by interpreting the deformation in that pattern caused by the surface of target objects. These patterns range from simple stripes to unique and convoluted speckle patterns.

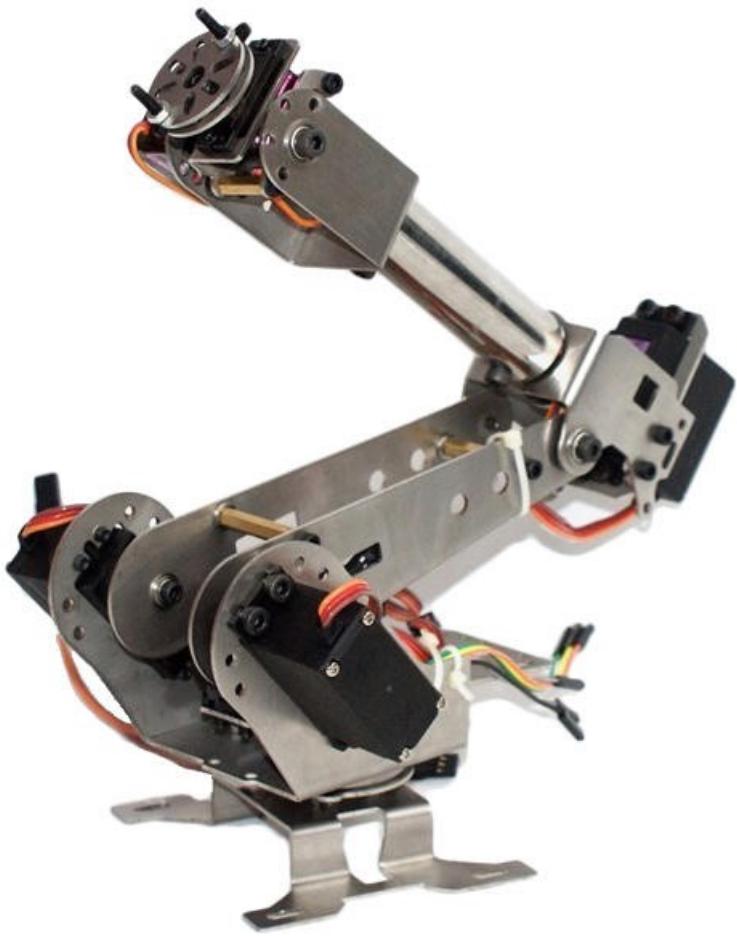


The advantage of using RGB-D cameras for 3D perception is that, unlike stereo cameras, they save a lot of computational resources by providing per-pixel depth values directly instead of inferring the depth information from raw image frames. In addition, these sensors are inexpensive and have a simple USB plug and play interface. RGB-D cameras can be used for various applications ranging from mapping to complex object recognition.

Sensor Specifications		
Cost	Range	Resolution
\$	~0.2 to 5m	Low

We will use the Kinect v1 Time of Flight camera to build 3D Maps of our environment. Although we can we will not use this camera for obstacle avoidance rather for object recognition and map building.

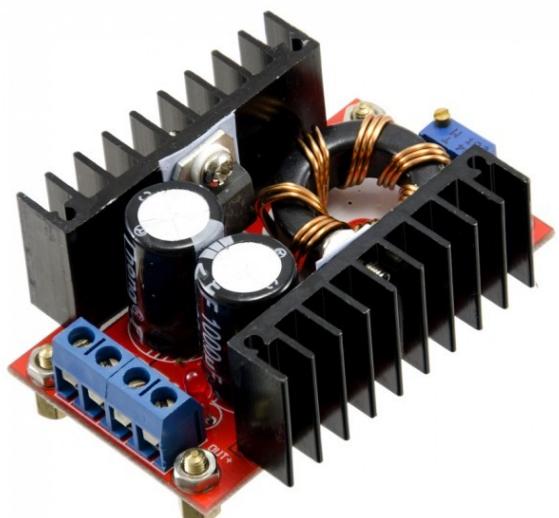
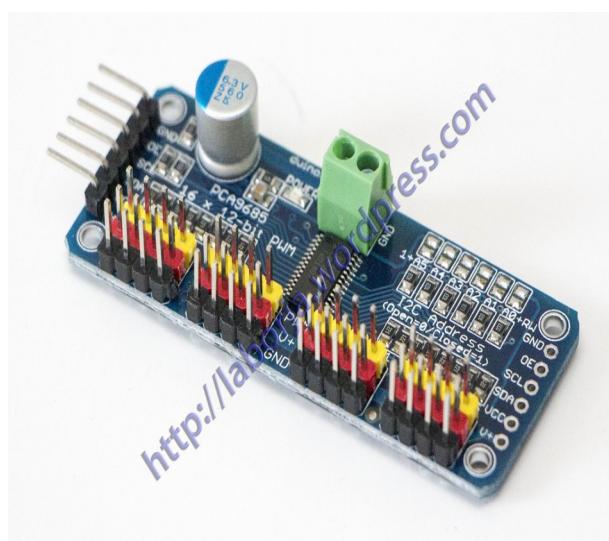
## 2.11 6 Degrees of Freedom Robotic Hand



Finally we equipped our robot with a cheap DIY robotic arm. It is of the anthropomorphic variety meaning it has joints similar to that of the human hand (no prismatic joints).

In order to control it we basically need to control 6 or 7 servos.

This is done easily with the help of an I2C Servo Driver which communicates with the arduino. Of course an external power supply is needed, usually from a battery after we convert the voltage to the appropriate value with an DC-DC boost Converter



## **2.12 Summary**

Now we have briefly seen and explained the main hardware components that are necessary to build your own autonomous robots.

From basic electronics, power management, sensors and embedded computers - micro-controllers to actuators like DC motors and servos.

Any robotics engineer must be familiar with at least these basic components that can be combined to build a variety of simple and complex robots tailored to your specific needs.

# 3. Introduction to Robot Operating System (ROS)

## 3.1 Introduction

In this chapter we will introduce the Robot Operating System also known as ROS. Specifically what it is? Why the need for such a pseudo operating system arose, how difficult it was to program robot software before ROS, what are its main capabilities and functions and the basic concepts of ROS, in order to familiarize ourselves with this powerful robotics tool that any roboticist must become familiar with!!

## 3.2 Ubuntu operating system



Ubuntu is a complete linux operating system based on Debian, which is open sourced with wide community and professional support, it was chosen for its hardware support and its user friendly environment for programmers. Most importantly it offers support for the Robot operating system that needs linux to install. Let's note here that the embedded computer Jetson Nano runs a variant of Ubuntu 18.04 Bionic Beaver for arm processors.

## 3.3 What is a robot?

A robot is made to perform a task, this can be autonomous or semi-autonomous like the surgeon robot Da-vinci or even not physical like Alexa, because answering a question is considered an action. On the other hand a battle bot may not be considered a robot since it's completely manually operated and it has no means of perception or decision making.

So a robot must be able to perceive its environment with sensors, perform a form of decision making and then perform a suitable action.

**What is a *robot*?**

- The word **robot** comes from the Czech word for forced labor, or serf.
- It was introduced by playwright Karel Čapek, whose fictional robotic inventions were created by chemical and biological, rather than mechanical, methods.

A cartoon illustration of a simple robot with a grey metallic body, large red eyes, and a small antenna. It has a screen on its chest displaying a yellow and green logo. It is standing on a white surface with a brown vertical bar to its right.

- **Perception:** a robot uses sensors to perceive its environment. Sensors come in a wide variety (**camera, stereoscopic camera, microphone, temperature and pressure sensors, taste -> chemical analyzers**) alongside non human analogous like (**lidar, “sonar”, gps, barometer, compass**). Each comes with its own advantages and disadvantages and usually a robot uses a multitude of sensors.
- **Decision making:** This can be as simple as Yes or No, or a complex decision tree. Also navigating an unknown environment, extracting information from images, applying Kalman Filters for measurement drift, Motion planning algorithms to go from A to B, Control algorithms to stay on track. Common techniques are simple decision trees, K-means, Deep learning.
- **Action:** This is arguably the most exciting part. Actions can be moving from A to B, speeding up or down, making a sensor measurement or even communicating with humans and other robots. Motors and Actuators can turn a wheel or a propeller or activate a joint in a robotic arm, We can power up a laser scanner to take a specific measurement, emit light or sound and send messages to communicate.

## 3.4 Before ROS?

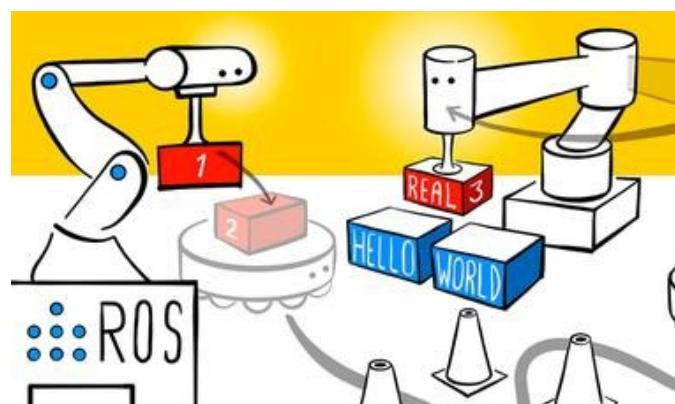
So want to build a robot? Got the hardware?  
What's left to do?

Simply

- Write device drivers for all our sensors and actuators.
- Develop a communication framework that can support different protocols.
- Write algorithms to do perception, navigation and motion planning.
- Implement mechanism to log our data.
- Write control algorithms.
- Error handling.

No wonder building a robot used to be a long and cumbersome process, starting from scratch and constantly reinventing the wheel.

## 3.5 What is Robot Operating System (ROS)

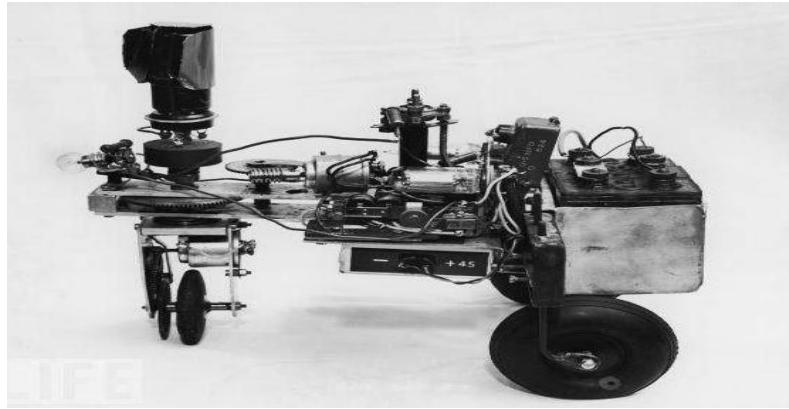


ROS is not an operating system in the typical sense, but like an OS it provides the means of talking to hardware without writing your own device drivers and a way for different processes to communicate with one another via message passing. ROS features a sleek build and package management system allowing you to develop and deploy software with ease.

Lastly ROS also has tools for visualization, simulation and analysis as well as extensive community support and an interface to numerous powerful software libraries.

[checkout this short documentary on ROS](#)

### 3.6 Brief history of ROS

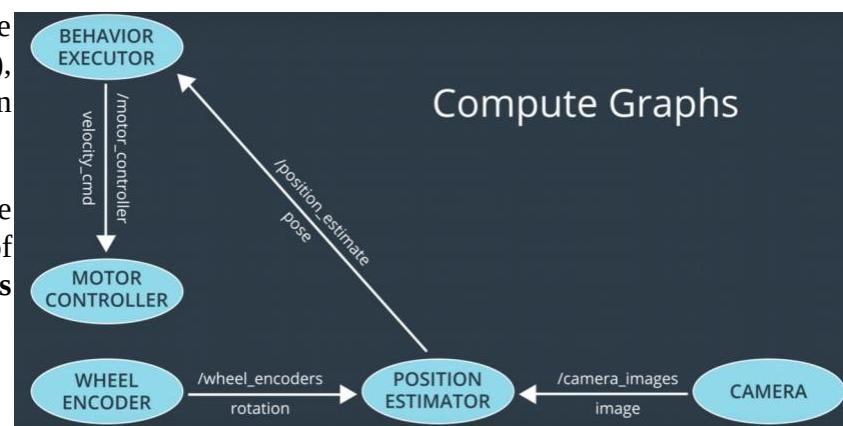


Development of ROS started in the mid-2000s as Project Switchyard in Stanford's Artificial Intelligence Laboratory. In 2007 it became a formal entity with support from Willow Garage. Since 2013 Open Source Robotics Foundation has been maintaining and developing ROS. Primary motivations for developing ROS include the recognition that researchers were constantly reinventing the wheel without many reusable robotics software components that could be used as a starting point for a project. Different groups were all working on custom solutions, so it was difficult for them to share code and ideas or compare results. ROS aims to facilitate the development process by eliminating these problems. Who uses ROS? Lots of people and companies, there are drones, kinematic arms, wheeled robots and even bi-pedal robots that use ROS.

### 3.7 Nodes and compute graphs

As you know all robots perform the same high level tasks of Perception (sensors), Decision Making (software) and Actuation (Motors and Controllers).

On the software side ROS manages these three complex steps **by breaking** each of them down into **many small unix processes called nodes**.



Typically each node is responsible for one small and relatively specific portion of the

robot's overall functionality for example capturing an image or moving the wheels.

ROS provides a powerful communication system, allowing these different nodes to communicate with one another **via message passing using topics, services and actions**.



These diagrams of nodes and topics and how they're all connected are frequently referred to as **compute graphs**. Visualizing the compute graph is very useful for understanding what nodes exist and how they communicate with one another, ROS provides a **tool** called **rqt\_graph** for showing the compute graph of a system. A moderately complex robot will likely have dozens of nodes, even more topics and quite a few services. This tool allows you to zoom in and pan around the graph, as well as choose exactly what information is displayed.



**Topic:** A topic is simply a **named bus** which you can think of as a pipe between nodes through which messages can flow. In order to **send a message** on a topic, we say that a node **must publish to it**, likewise to receive a message on a topic, a node **must subscribe to it**. In the compute graph the arrows represent message flow from publishers to subscribers, each node may simultaneously publish and subscribe to a wide variety of topics. Taking together these network of nodes connected by topics is called a **Publish Subscribe architecture**.

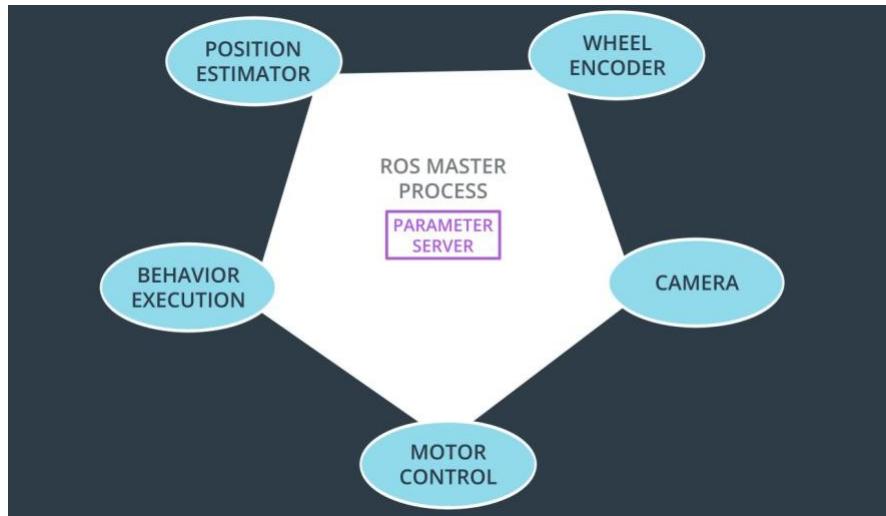
**Message passing:** each ros distribution comes with a wide variety of predefined messages. These can include messages for **Physical Quantities** like (Position, Velocity, Rotations) or for **Sensor readings** like (Laser scans, Images, Point Clouds, Inertial Measurements). But there will come a time where you will want to define your own types. Although messages imply text based content they can in fact contain any kind of data.

**Services:** Passing messages over topics between Publishers and Subscribers is useful but it's not a

**one size fits all communication solution**, there are times when a **request-response pattern** is useful, for these types of interactions ros provides what's called services. Services like topics allow the passing of messages between nodes **but unlike topics** are not a bus hence there are no publishers and subscribers associated with them. Instead **nodes interacting via services do so on 1 to 1 bases using a request response pattern**. For example if we want an image captured every once in a while with specific parameters, a **request (exposure time)** and **response (image)** type of communication is more suitable than a named bus.

**Actions** : Actions are similar to Services but **they also provide feedback and a means to cancel the action or modify it while the robot performs the action**. There are useful when we tell the robot to go from A to B and we want feedback, or a kinematic arm grasping an object.

## 3.8 Ros Master and Parameter Server



**ROS Master:** is at the center of these collection of nodes and acts as a sort of **manager for all the nodes**.

- **Maintains registry** of all the active nodes of the system
- Allows each node to **discover** other nodes in the system and **establish lines of communication with them**

In addition ROS Master also hosts the **Parameter Server** which is a **central repository** typically used to store parameters and configuration values that are shared among the running nodes. Nodes now can **look up** values as needed rather than storing them in different places.

For example the wheel radius might be needed for one node to estimate speed and by another to estimate position.

### 3.9 Packages and Catkin workspaces



- Ros provides a powerful build and package management system called catkin, named after the flowers on the willow trees surrounding their office.
- A **catkin workspace** is a directory where **catkin packages are built, modified and installed**, typically a single workspace holds a wide variety of catkin packages
- All Ros software components are organized into and distributed **as catkin packages**
- Similar to workspaces catkin packages are directories containing a variety of resources, which when considered together constitute some sort of useful module
- Catkin packages may contain **source code for nodes, useful scripts, configuration files and more**

[more information about catkin build system here](#)

#### How to create a catkin workspace

```
$ mkdir -p ~/catkin_ws/src  
$ cd ~/catkin_ws/src  
$ catkin_init_workspace //initialize workspace  
$ cd ..  
$ catkin_make // compile the workspace
```

By initializing the workspace we create a file in the src directory CmakeLists.txt this file is used by catkin **to tell it how to build the packages**. It has **commented** parts that you can uncomment modify and use. The common structure of this file is

- **Required CMake Version** (cmake\_minimum\_required)
- **Package Name** (project())
- **Find other CMake/Catkin packages needed for build** (find\_package())
- **Enable Python module support** (catkin\_python\_setup())
- **Message/Service/Action Generators**

- `(add_message_files(), add_service_files(), add_action_files())`
- **Invoke message/service/action generation** (`generate_messages()`)
- **Specify package build info export** (`catkin_package()`)
- **Libraries/Executables to build** (`add_library()/add_executable()/target_link_libraries()`)
- **Tests to build** (`catkin_add_gtest()`)
- **Install rules** (`install()`)

We will see some examples on how to modify this files to build c++ nodes/executables and **custom** message/service/action messages.

Now we have build our first ros workspace it doesn't do much but it satisfies or the criteria of a workspace. If we **cd** to **catkin\_ws** the workspace and **ls** we can see three files

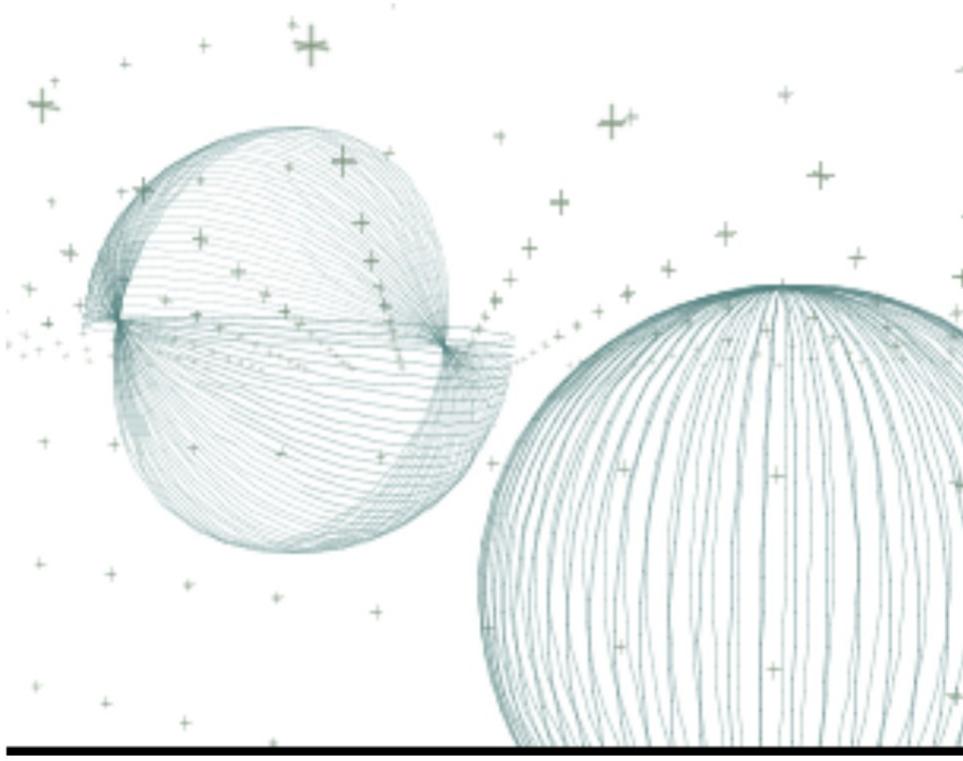
- **build:** The build space is where CMake is invoked to build the catkin packages in the source space. CMake and catkin keep their cache information and other intermediate files here. It is the **built space of the c++ packages and we usually don't interact with it.**
- **src:** The source space contains the **source code of catkin packages**. This is where you can **extract/checkout/clone** source code for the packages you want to build. Each folder within the source space contains one or more catkin packages. This space should remain unchanged by configuring, building, or installing. The root of the source space contains a symbolic link to catkin's boiler-plate 'toplevel' CMakeLists.txt file. This file is invoked by CMake during the configuration of the catkin projects in the workspace. It can be created by calling `catkin_init_workspace` in the source space directory.
- **devel:** The development space is where built targets are placed prior to being installed. The way targets are organized in the development space is the same as their layout when they are installed. This provides a useful testing and development environment which does not require invoking the installation step. This folder **contains the setup.bash** that needs to be **sourced in each terminal** in order for ros to find the packages.

Another file you might want to include in your packages is the **package manifest** which is an **XML file** called **package.xml** that must be included with any **catkin-compliant package's root folder**. This file defines properties about the package such as the **package name, version numbers, authors, maintainers, and dependencies on other catkin packages**.

Your system **package dependencies** are declared in package.xml. If they are missing or incorrect, you **may be able to build from source and run tests on your own machine**, but your package **will not work correctly when released to the ROS community**. Others depend on this information to install the software they need for using your package. The most important thing here are the **build and runtime dependencies** usually other ros packages.

## 3.10 Running the turtlesim example

Distro	Release date	Poster	Tuturtle, turtle in tutorial	EOL date
ROS Lunar Loggerhead	May, 2017	TDB	TDB	May, 2019
ROS Kinetic Kame <i>(Recommended)</i>	May 23rd, 2016			May, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015			May, 2017
ROS Indigo Igloo	July 22nd, 2014			April, 2019 (Trusty EOL)
ROS Hydro Medusa	September 4th, 2013			May, 2015
ROS Groovy Galapagos	December 31, 2012			July, 2014
ROS Fuerte Turtle	April 23, 2012			--
ROS Electric Emys	August 30, 2011			--
ROS Diamondback	March 2, 2011			--



**Turtles and robotics go way back** to the 1940's, early roboticist William Gray created some of the first autonomous devices, turtle robots which he called Elmer and Elsie. And in 1960's at MIT Seymour Papert used turtle robots in **robotics education**. They could move **forward and backward by a given distance** and **rotate by a given angle** or they could **drop a retractable pen and draw complex trajectories** and **sophisticated drawings**. Each **recent version of ROS has been named after some sort of turtle**.

### Environment setup

Before running ROS from a terminal we must ensure that all of the **environment variables** are present, usually by **sourcing the setup script provided by ros source /opt/ros/kinetic/setup.bash**. (We often put this command on **.bashrc** so it will execute automatically when we open a terminal  
**echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc** )

## Environment variables

- **ROS\_ROOT** sets the location where the ROS core packages are installed  
export ROS\_ROOT=/home/user/ros/ros
- **ROS\_MASTER\_URI** a required setting that tells nodes where they can locate the master  
export ROS\_MASTER\_URI=<http://mia:11311/>
- **PYTHONPATH** ROS requires that your PYTHONPATH be updated, **even if you don't program in Python!** Many ROS infrastructure tools rely on Python and need access to the roslib package for bootstrapping.  
export PYTHONPATH=\$PYTHONPATH:\$ROS\_ROOT/core/roslib/src
- and many more!!

**roscore:** by running this command in a terminal we **start the master process**

- Providing naming and registration services to other running nodes
- Tracking all publishers and subscribers
- Aggregating log messages generated by the nodes
- Facilitating connections between nodes

```
... logging to /home/makemedie/.ros/log/b0b144ac-e2ac-11e9-baac-7085c2697a96/roslaunch-makemedie-desktop-6900.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://192.168.2.2:45237/
ros_comm version 1.12.14

SUMMARY
=====

PARAMETERS
* /rosdistro: kinetic
* /rosversion: 1.12.14

NODES

auto-starting new master
process[master]: started with pid [6912]
ROS_MASTER_URI=http://192.168.2.2:11311/

setting /run_id to b0b144ac-e2ac-11e9-baac-7085c2697a96
process[rosout-1]: started with pid [6925]
started core service [/rosout]
```

**Install turtlesim:** \$ sudo apt-get install ros-\$(rosversion -d)-turtlesim

\$ **rosrun package\_name executable/node**

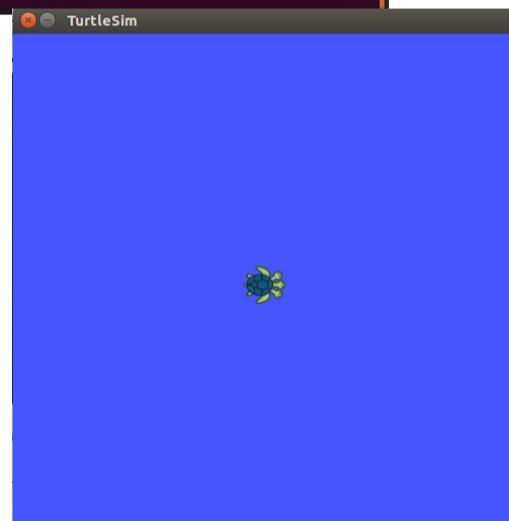
Here package name is turtlesim and the node is turtlesim\_node

\$ **rosrun turtlesim turtlesim\_node**

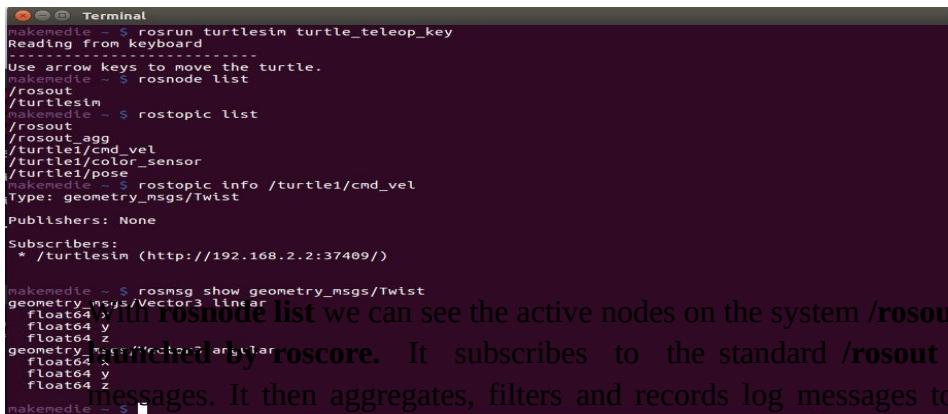
Now a graphical window will appear

And by running the  
turtle\_teleop\_key

\$ **rosrun turtlesim turtle\_teleop\_key**



We can move the turtle around with our arrow keys and draw something.



```
makemedie $ roslaunch turtlesim turtle_teleop_key
Reading from keyboard
-----
Use arrow keys to move the turtle.
makemedie ~ $ rosnode list
/turtle1
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
makemedie ~ $ rostopic list
Type: geometry_msgs/Twist
Publishers: None
Subscribers:
* /turtlesim (http://192.168.2.2:37409/)

makemedie ~ $ rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

With **rosnode list** we can see the active nodes on the system **/rosout** and **/turtlesim**, **/rosout** node is launched by **roscore**. It subscribes to the standard **/rosout topic** where all nodes send log messages. It then aggregates, filters and records log messages to a text file. **/turtlesim** node, it provides a simple simulator for teaching ROS concepts, subscribes to the **turtle1/cmd\_vel topic** and publish **turtle1/pose topic/turtle\_teleop\_key**. It captures the arrows presses and transforms them into a **geometry\_msgs/Twist** message and then publish it to **/turtle1/cmd\_vel** topic.

With **rostopic list** we see what topic are being published

**/rosout\_agg** : aggregated form of messages published to **/rosout**  
**/turtle1/cmd\_vel** : if we publish to this topic the turtle will move

With **rostopic info** we can see which node publish or subscribe to a topic and what type of message is being published. We can see that the **message type is geometry\_msgs/Twist**, turtlesim node subscribes to this topic and currently no node publish to it since **turtle\_teleop\_key** is not activated.

With **rosmsg show geometry\_msgs/Twist** we can see information about the message, here it is comprised of **two geometry\_msgs/Vector3 types** linear and angular and each vector holds 3 float values x , y, z. This type of message is used to publish linear and angular velocities of a free floating object in 3-d space.

Lastly with **rostopic echo /topic\_name** , realtime information of the messages being published to this topic are fed to our terminal.

## 3.11 Launch and configuration files

ROS uses launch files

- launch ROS Master automatically and multiple nodes with one command
- set default paramaters on the paramater server
- automatically respawn processes that have died

In the launch files we can include YAML files that hold configuration values

Here is an example of a lauch file

```
<launch>

<group ns="turtlesim1">
  <node pkg="turtlesim" name="sim" type="turtlesim_node"/> </group>

<group ns="turtlesim2">
  <node pkg="turtlesim" name="sim" type="turtlesim_node"/> </group>
```

```

<node pkg="turtlesim" name="mimic" type="mimic"> <remap
from="input" to="turtlesim1/turtle1"/> <remap from="output"
to="turtlesim2/turtle1"/> </node>

</launch>

```

Which launches 2 turtlesim\_node nodes **under different namespace** and a mimic node.

In the mimic node we can see **a remapping of the name of the topics**, this is useful if your topics don't have the default names expected by a node.

Here is a launch file used later to start our 2d lidar later

### ydlidar.launch

```

<?xml version="1.0"?>
<launch>
  <node name="ydlidar_node" pkg="ydlidar" type="ydlidar_node" output="screen"> <rosparam file="$(find
lynxbot_bringup)/config/ydlidar_params.yaml" command="load" /> </node>
</launch>

```

Here we load a .yaml file which holds the configuration values of the ydlidar\_node which is in the lynxbot\_bringup package/config folder

### ydlidar\_params.yaml

```

port: /dev/ydlidar
baudrate: 115200
frame_id: laser_frame
low_exposure: false
resolution_fixed: true
auto_reconnect: true
reversion: false
angle_min: -180
angle_max: 180
range_min: 0.1
range_max: 6.0
ignore_array: ""
samp_rate: 9
frequency: 8.5

```

we can start the launch file with the command

**\$ rosrun lynxbot\_bringup**

**ydlidar.launch** and the laser should start

scanning the room.

So in conclusion we can run nodes **one by one but this becomes quickly tedious**, with launch files we can launch ros master and multiple nodes with one single command, however since all the nodes are “running” in the same terminal any log messages might be replaced by other nodes making debugging more difficult. For that we can use **script files** that run multiple terminals usually **xterm** with different ros commands.

## 3.12 Topics and writing our first c++ node

```
#include <ros/ros.h>
#include <std_msgs/Int32.h>
int main(int argc, char** argv){

    ros::init(argc,argv,"topic_publisher");
    ros::NodeHandle nh;
    ros::publisher pub = nh.advertise<std_msgs::Int32> ("counter", 1000);
    ros::Rate loop_rate(2);
    std_msgs::Int32 count;
    count.data = 0;

    while (ros::ok()){
        pub.publish(count);
        ROS_INFO("the counter is %d", count.data);
        ros::spinOnce();
        loop_rate.sleep();
        ++ count.data;
    }
    return 0;
}
```

This is the **source code** of a c++ node that just adds 1 to a counter every 2 seconds and then **publish**

**this counter to a topic called /counter.**

**Let's break down the code**

**ros/ros.h** is a convenience include that **includes all the headers necessary** to use the most **common public pieces** of the ROS system.

```
#include <std_msgs/Int32.h>
```

It includes the **std\_msgs/Int32 message** which resides in the **std\_msgs package**

```
ros::init(argc,argv,"topic_publisher");
```

Here we initialize ROS. This is also where we specify the name of our node. This allows ROS to do name remapping through the command line -- not important for now. Names need to be unique in the system.

```
ros::NodeHandle nh;
```

Create a handle to this process' node. The first NodeHandle created will actually do the initialization of the node, and the last one destructed will cleanup any resources the node was using.

```
ros::publisher pub = nh.advertise<std_msgs::Int32> ("counter", 1000);
```

**Tells the master** that we are going to be **publishing a message of type std\_msgs/Int32 on the topic counter**. This lets the master tell any nodes listening on counter that we are going to publish data on that topic. The second argument is the size of **our publishing queue**. If we are publishing too quickly it will buffer up a maximum of 1000 messages before beginning to throw away old ones.

**NodeHandle::advertise() returns a ros::Publisher object, which serves two purposes:**

- 1) It contains a publish() method that lets you publish messages onto the topic it was created with
- 2) when it goes out of scope, it will automatically unadvertise.

**ros::rate loop\_rate(2);**

A ros::Rate object allows you to specify a frequency that you would like to loop at.

**std\_msgs::Int32 count;**

a variable count that holds a std\_msgs::Int32 message

**count.data = 0;**

put 0 in the data field of our message

**while (ros::ok())**

ros::ok() will return false if:

- a SIGINT is received (Ctrl+C)
- we have been kicked off the network by another node with the same namespace
- ros::shutdown() has been called by another part of the application

**pub.publish(count);**

we actually broadcast the message to anyone who is connected

**ROS\_INFO("the counter is %d", count.data);**

the ROS replacement of printf/cout

**ros::spinOnce();**

Calling it here for this simple program is not necessary, because **we are not receiving any callbacks**. However we add it for good measure. Since if it doesn't exist any callback will not be executed.

**loop\_rate.sleep();**

We sleep accordingly to loop\_rate frequency

**++ count.data;**

we increment the counter message data field.

### **Here's the condensed version of what's going on**

- Initialize the ROS system
- Advertise that we are going to be publishing std\_msgs/Int32 messages on the counter topic to the master.
- Loop while publishing messages to counter every 2 seconds

**Now we need to write a node to receive the messages**

```
#include <ros/ros.h>
#include <std_msgs/Int32.h>

Void counterCallback(const std_msgs::Int32::ConstPtr& msg)
{
    ROS_INFO("%d", msg->data);
}

int main(int argc, char** argv){
    ros::init(argc, argv, "topic_subscriber");
    ros::NodeHandle nh;
    ros::Subscriber sub = nh.subscribe("counter", 1000, counterCallback);
    ros::spin()
    return 0;
}
```

lets break it up

```
void counterCallback(const std_msgs::Int32::ConstPtr& msg)
{
    ROS_INFO("%d", msg->data);
}
```

This is the **callback function** that will be called when a new message has arrived on the counter topic.

```
ros::Subscriber sub = nh.subscribe("counter", 1000, counterCallback);
```

Subscribe to the counter topic with the master. ROS will call counterCallback() function whenever a new message arrives. The second argument is the queue size.

**ros::spin()**

enter a loop, calling message callbacks as fast as possible. If there are not messages it won't use much CPU. There are other ways of pumping callbacks too.

Now we wrote the source code but if we do catkin\_make, catkin has no idea how to find or build our source code.

We can create a package with the command (on the src folder)  
catkin\_create\_pkg pkg\_name dependencies

**\$ catkin\_create\_pkg pub\_sub roscpp**

Creates a pub\_sub package with roscpp as dependency

**Now on the created CmakeLists.txt file we need to add or uncomment this lines**

```
add_executable(counter src/counter.cpp)
target_link_libraries(counter ${catkin_LIBRARIES})
```

```

add_dependencies(counter ${simple_EXPORTED_TARGETS} $  

(catkin_EXPORTED_TARGETS))

add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
add_dependencies(listener ${simple_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS))

```

**And with catkin\_make it should build,** we can make a launch file to launch the nodes or

```

$ rosrun pub_sub counter.cpp
$ rosrun pub_sub listener.cpp

```

## Creating a custom message

**The first thing we need to do is to create a msg folder in our package, then we create a file for example**

**Age.msg**

```

float32 years
float32 months
float32 days

```

**Here our costum message will have 3 fields.**

First on the **package.xml** we will have to make sure these lines are uncommented

```

<build_depend>message_generation</build_depend>
<build_export_depend>message_runtime</
build_export_depend>
<exec_depend>message_runtime</exec_depend>

```

Next on the **CmakeLists.txt** we need to add message\_generation

```

dependency find_package(catkin REQUIRED COMPONENTS
    roscpp
    rospy
    std_msgs
    message_generation
)

```

**Also make sure you export the message runtime dependency.**

```

catkin_package(
    CATKIN_DEPENDS message_runtime roscpp std_msgs...
)

```

**Find the following block of code and uncoment it:**

```
add_message_files(
```

```
    FILES  
    Age.msg  
)
```

**Also**

```
generate_messages(
```

```
    Depedencies
```

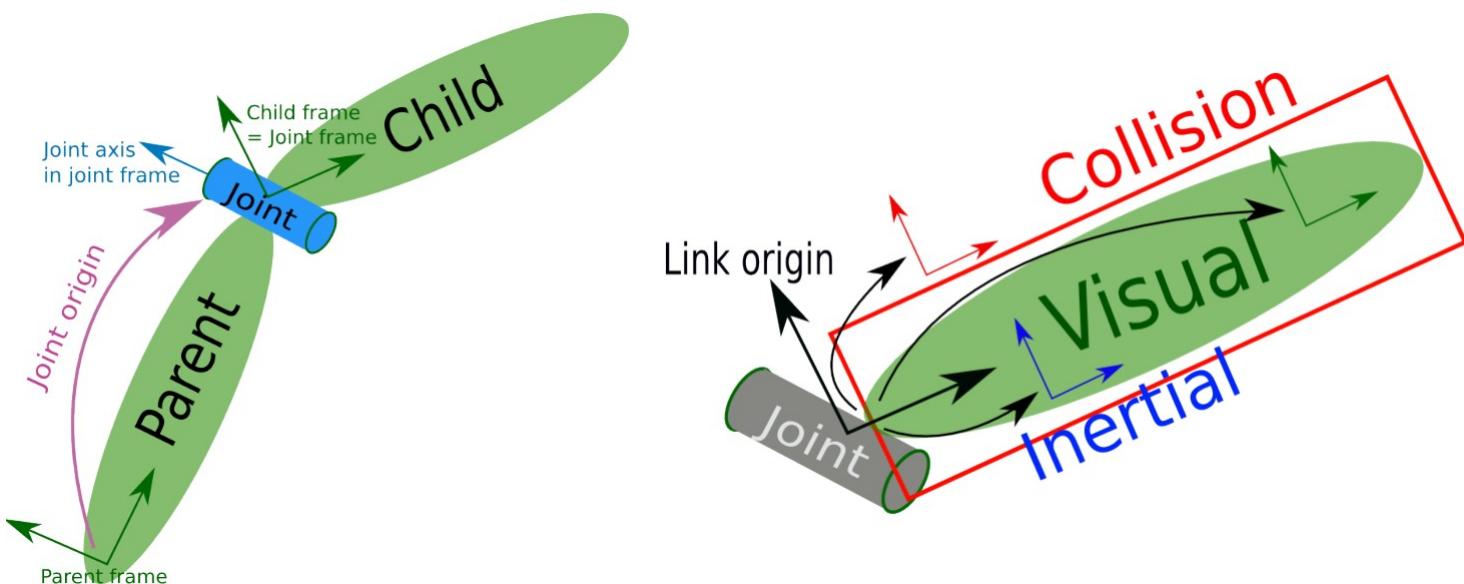
```
    std_msgs
```

```
)
```

Now after building we can check that everything works fine by running:

```
rosmsg show package_name/Age
```

### 3.13 URDF and XACRO



Unified Robot Description Format or urdf, is an XML format used in ROS for representing a robot model. We can use a **urdf file** to define a **robot model**, its **kinodynamic properties**, **visual elements** and even **model sensors for the robot**. URDF can only describe a robot with **rigid links connected by joints in a chain or tree-like structure**. It cannot describe a robot with flexible links or parallel linkage.

A simple robot with two links and a joint can be described using urdf as follows:

```
<?xml version="1.0"?>  
<robot name="two_link_robot">
```

```

<!--Links-->
<link name="link_1">
  <visual>
    <geometry>
      <cylinder length="0.5" radius="0.2"/>
    </geometry>
  </visual>
</link>
<link name="link_2">
  <visual>
    <geometry>
      <box size="0.6 0.1 0.2"/>
    </geometry>
  </visual>
</link>
<!--Joints-->
<joint name="joint_1" type="continuous">
  <parent link="link_1"/>
  <child link="link_2"/>
</joint>
</robot>

```

Since we use urdf files to describe several robot and environmental properties, they **tend to get long and tedious to read through**. This is why we use **Xacro (XML Macros) to divide our single urdf file into multiple xacro files**. While the syntax remains the same, we can now divide our robot description into smaller subsystems.

urdf (and xacro) files are basically XML, so they use tags to define robot geometry and properties. The most important and commonly used tags with their elements are described below:

### <robot> </robot>

This is a top level tag that contains all the other tags related to a given robot.

### <link> </link>

Each rigid link in a robot must have this tag associated with it.

#### **Attributes**

**name:** Requires a unique link name attribute.

#### **Elements**

### <visual> </visual>

This element specifies the appearance of the object for visualization purposes.

NAME	Description
<origin>	The reference frame of the visual element with respect to the reference frame o

	the link.
<geometry>	The shape of the visual object
<material>	The material of the visual element

### **<collision></collision>**

The collision properties of a link. Note that this can be different from the visual properties of a link, for example, simpler collision models are often used to reduce computation time.

NAME	Description
<origin>	The reference frame of the collision element, relative to the reference frame of the link.
<geometry>	See the geometry description in the above visual element.

### **<inertial></inertial>**

The inertial properties of the link are described within this tag.

NAME	Description
<origin>	This is the pose of the inertial reference frame, relative to the link reference frame. The origin of the inertial reference frame needs to be at the center of gravity.
<mass>	The mass of the link is represented by the value attribute of this element.
<inertia>	The 3x3 rotational inertia matrix, represented in the inertia frame. Because the rotational inertia matrix is symmetric, only 6 above-diagonal elements of this matrix are specified here, using the attributes ixx, ixy, ixz, iyy, iyz, izz.

Example snippet for <link> tag with important elements:

```

<link name="link_1">
  <inertial>
    <origin xyz="0 0 0.4" rpy="0 0 0"/>
    <mass value="${mass1}"/>
    <inertia ixx="30" ixy="0" ixz="0" iyy="50" iyz="0" izz="50"/>
  </inertial>
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <mesh filename="package://kuka_arm/meshes/kr210l150/visual/link_1.dae"/>
    </geometry>
    <material name="">
      <color rgba="0.75294 0.75294 0.75294 1"/>
    </material>
  </visual>
</link>

```

```

</material>
</visual>
<collision>
  <origin xyz="0 0 0" rpy="0 0 0"/>
  <geometry>
    <mesh filename="package://kuka_arm/meshes/kr210l150/collision/link_1.stl"/> </geometry>
  </collision>
</link>

```

This `<link>` tag has many more optional elements that can be used to define other properties like color, material, texture, etc. Refer [this link](#) for details on those tags.

## `<joint></joint>`

This tag typically defines a single joint between two links in a robot. The type of joints you can define using this tag include:

NAME	Description
Fixed	Rigid joint with no degrees of freedom. Used to <b>weld</b> links together.
Revolute	A range-limited joint that rotates about an axis.
Prismatic	A range-limited joint that slides along an axis
Continuous	Similar to <b>Revolute</b> joint but has no limits. It can rotate continuously about an axis.
Planar	A 2D <b>Prismatic</b> joint that allows motion in a plane perpendicular to an axis.
Floating	A joint with 6 degrees of freedom, generally used for Quadrotors and UAVs

## Attributes

**name** Unique joint name

**type** Type of joint

## Elements

To define a joint, we need to declare the axis of rotation/translation and the relationship between the two links that form the joint.

NAME	Description
<code>&lt;origin&gt;</code>	This is the transform from the parent link to the child link. The joint is located at the origin of the child link.
<code>&lt;parent&gt;</code>	Name of the Parent link for the respective joint.
<code>&lt;child&gt;</code>	Name of the child link for the respective joint.

<code>&lt;axis&gt;</code>	Defines the axis of rotation for revolute joints, the axis of translation for prismatic joints, and the surface normal for planar joints. Fixed and floating joints do not use the axis field.
---------------------------	--

Example snippet for `<joint>` tag with important elements:

```
<joint name="joint_2" type="revolute">
<origin xyz="0.35 0 0.42" rpy="0 0 0"/>
<parent link="link_1"/>
<child link="link_2"/>
<axis xyz="0 1 0"/>
</joint>
```

**There are many more optional tags and attributes that help to define various dynamic and kinematic properties of a robot along with sensors and actuators. you can refer [this link](#) for more details on those.**

Other optional elements under the `<joint>` tag can be [found here](#).

## Xacro

Xacro is a **macro** language. The xacro program runs all of the macros and outputs the resulting urdf. Usually we generate the urdf automatically during launch in order to stay up to date and for convenience. Typical usage looks like this.

Xacro -- inorder model.xacro > model.urdf

At the top of a URDF file we must specify a namespace in order for the file to parse correctly. For a valid xacro file:

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="firefighter">
```

with xacro we can **define constants, math operations and parameterized macros** as well as divide into many files each representing a specific part of the robot

## Constants

```
<xacro:property name="width" value="0.2" /> <xacro:property
name="bodylen" value="0.6" /> <cylinder radius="${width}"
length="${bodylen}" />
```

## Math

```
<cylinder radius="${wheeldiam/2}" length="0.1"/> <origin
xyz="${reflect*(width+.02)} 0 0.25" />
```

## Parameterized Macro

```

<xacro:macro name="default_inertial" params="mass">

  <inertial>
    <mass value="${mass}" />
    <inertia ixx="1.0" ixy="0.0" ixz="0.0"
             iyy="1.0" iyz="0.0"
             izz="1.0" />
  </inertial>
</xacro:macro>

```

This can be used with the code

```
<xacro:default_inertial mass="10"/>
```

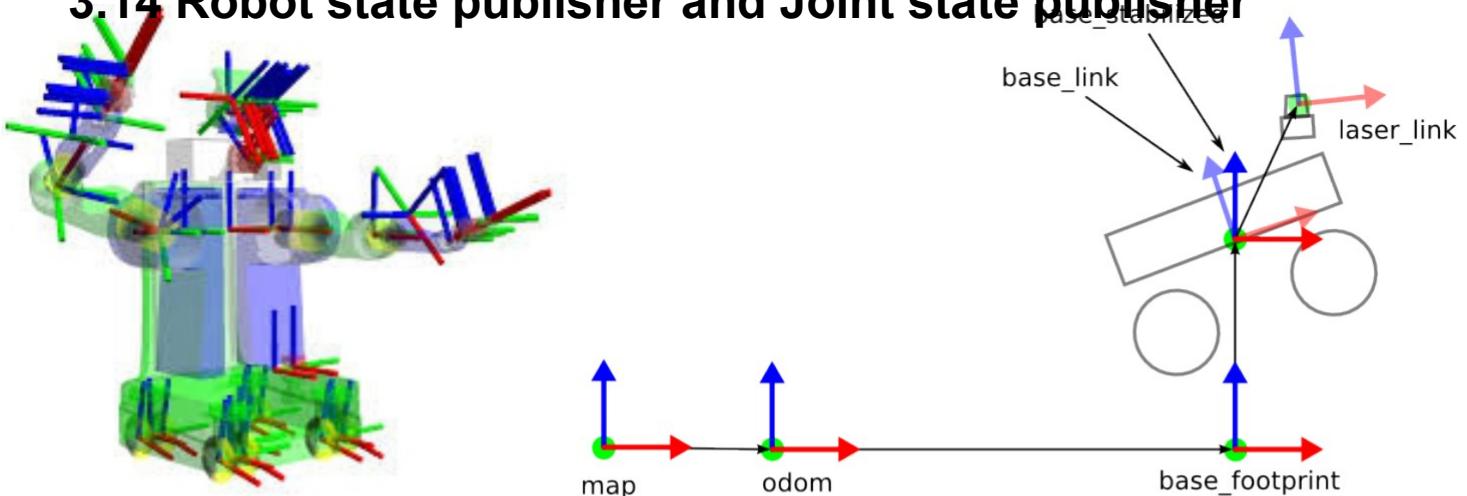
Include a xacro file inside another

```
<xacro:include filename="$(find lynxbot_description)/urdf/common_properties.urdf.xacro" />
```

**Common Trick 1:** Use a name prefix to get two similarly named objects

**Common Trick 2:** Use math to calculate joint origins. In the case that you change the size of your robot, changing a property with some math to calculate the joint offset will save a lot of trouble.

### 3.14 Robot state publisher and Joint state publisher

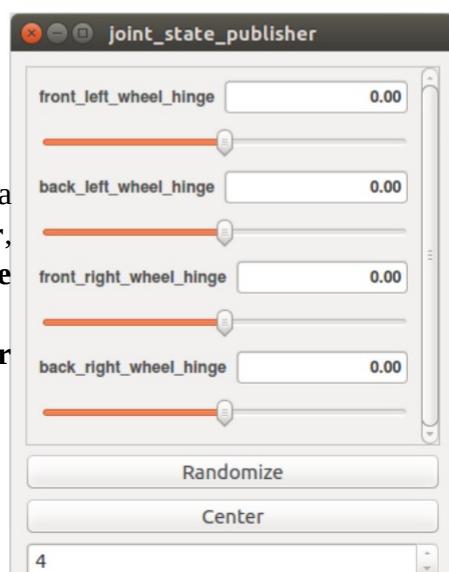


### Joint state publisher

This package **publishes sensor\_msgs/JointState messages** for a robot. The package **reads the robot\_description parameter**, finds **all of the non-fixed joints** and **publishes a JointState message with all those joints defined**.

Can be used **in conjunction with the robot\_state\_publisher node** to also **publish transforms for all joint states**.

There are four possible sources for the value of each JointState:



1. Values directly input through the GUI
2. JointState messages that the node subscribes to
3. The value of another joint
4. The default value

So in essence this node reads the urdf, finds all of the moving joints and **publish the joint state** of each of those joints. We can see the GUI of a 4 wheeled vehicle, **we can change the joint state of each joint by sliding each slider.**

## Robot state publisher

`robot_state_publisher` uses the URDF specified by the parameter `robot_description` and the joint positions from the topic `joint_states` to calculate the forward kinematics of the robot and publish the results via TF.

**Transform Frames (TF):** lets the user keep track of multiple coordinate frames over time. TF maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time.

In the case of a mobile robot we usually have a static frame called map representing a point inside our map, then TF publish the transform between that point and the odometry frame. Odometry calculates the position of the robot from a starting position usually with wheel encoders (**blind odometry**) or external localization techniques like lasers.

By using this launch file

### `robot_description.launch`

```
<?xml version="1.0"?>

<launch>

<!-- send urdf to param server -->

    <param name="robot_description" command="$(find lynxbot_description)/urdf/lynx_rover.urdf.xacro" />

<!-- Send joint values-->

    <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher">
        <param name="use_gui" value="true"/>
    </node>

<!-- Send robot states to tf -->
    <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" respawn="false" output="screen">
        <param name="publish_frequency" type="double" value="10.0" />
    </node>
</launch>
```

```
</node>
```

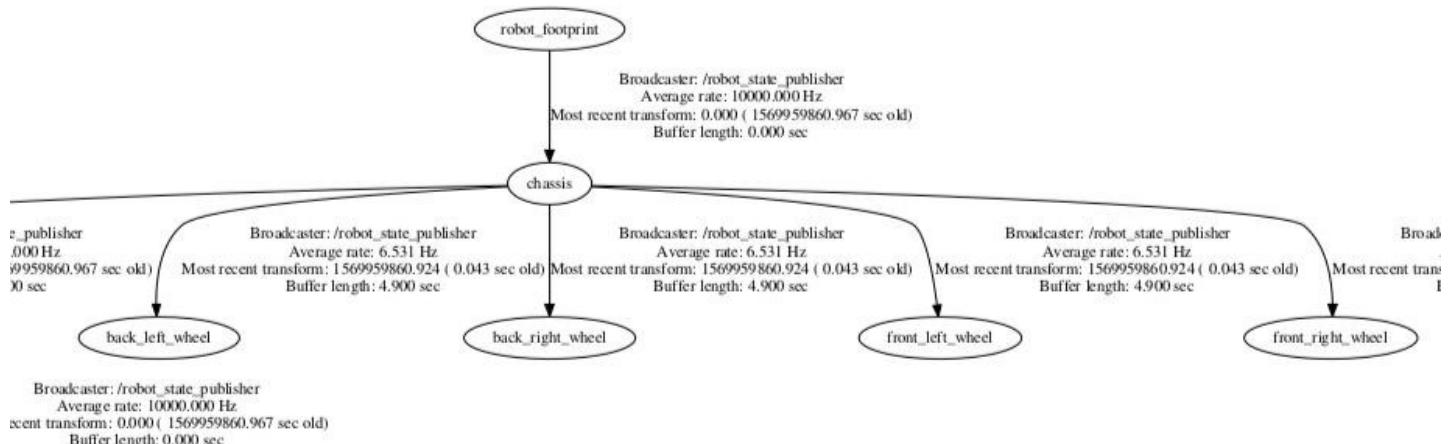
```
</launch>
```

We first **load the URDF to the parameter server** (after converting the xacro in URDF on launch). Then we run the **joint\_state\_publisher** which publishes the joint states of our robot.

Lastly the robot\_state\_publisher **reads the URDF alongside the JointStates**, computes the forward kinematics and publish them as TF.

After launching the nodes (`$ rosrun package_name robot_description.launch`)

We can run the command (`$ rosrun tf view_frames`) that **listens to the published tf for 5 seconds** and **outputs a pdf with all the links, their transforms and the corresponding time**



)

## 3.15 Gazebo



Gazebo is a **physics based** high fidelity **3D simulator for robotics**. Gazebo provides the ability to accurately **simulate** one or more robots in complex **indoor and outdoor** environments filled with **static and dynamic objects, realistic lighting, and programmable interactions**.

Gazebo facilitates **robot design, rapid prototyping & testing, and simulation of real-life scenarios**. While Gazebo is platform agnostic and supports Windows, Mac, and Linux, it is mostly used in conjunction with ROS on Linux systems for robotics development. Simply put, it is an essential tool in every roboticist's arsenal. For more information on the history of Gazebo and a comprehensive list of features visit [their website](#).

### Gazebo components

The two main components involved in running an instance of Gazebo simulation are **gzserver** and **gzclient**.

**gzserver** performs most of the heavy-lifting for Gazebo. It is responsible for parsing the description files related to the scene we are trying to simulate as well as the objects within, it then simulates the complete scene using a physics and sensor engine.

While the server can be launched independently by using the following command in a terminal:

```
$ gzserver
```

it does not have any GUI component. Running **gzserver** in a so-called headless mode can come in handy in certain situations.

**gzclient** on the other hand provides the very essential Graphical Client that connects to the **gzserver** and renders the simulation scene along with useful interactive tools. While you can technically run **gzclient** by itself using the following command:

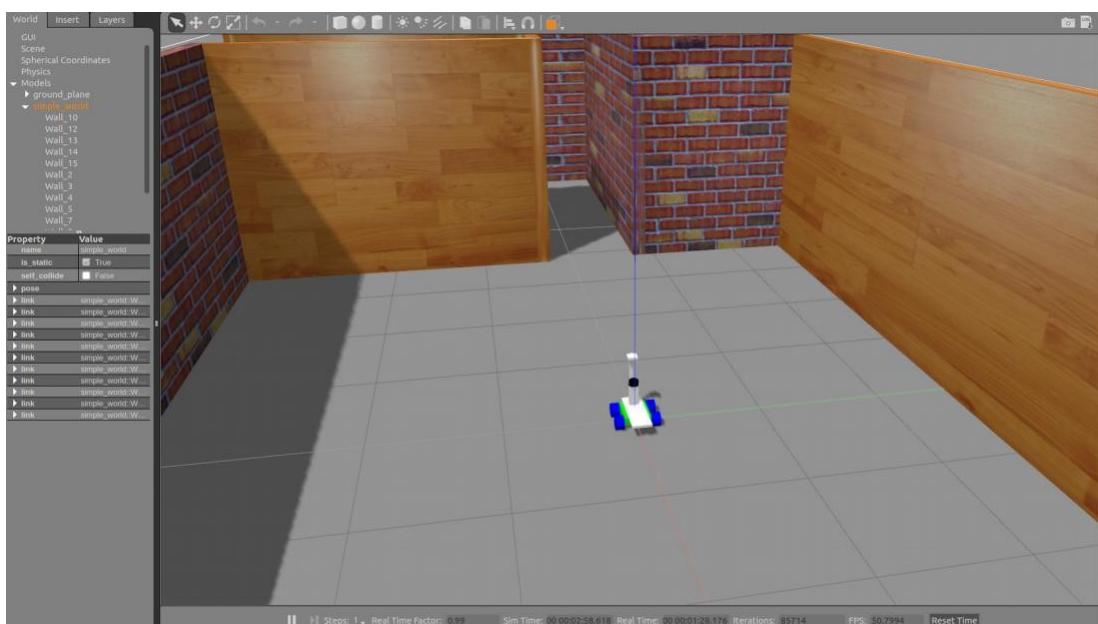
```
$ gzclient
```

it does nothing at all (except consume your compute resources), since it does not have a **gzserver** to connect to.

It is a common practice to run **gzserver** first, followed by **gzclient**, allowing some time to initialize the simulation scene, objects within, and associated parameters before rendering it. To make our lives easier, there is a single intuitive command that necessarily launches both the components sequentially:

```
$ gazebo
```

## Exploring Gazebo User Interface



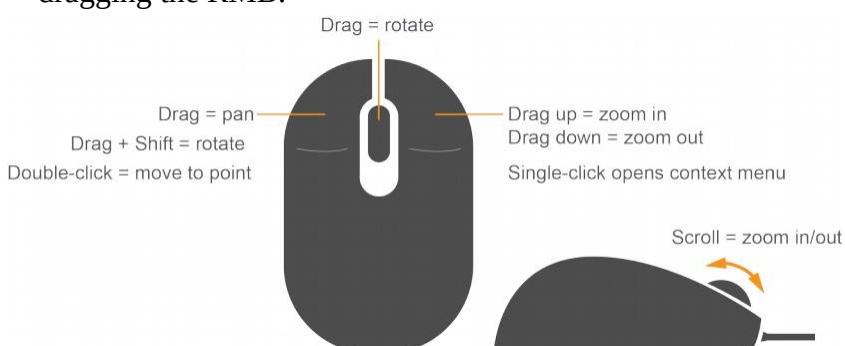
### The Gazebo gui is divided into 4 major sections:

1. Scene
2. Side Panel
3. Toolbars
4. Menu

### Scene

**The scene is where you will be spending most of your time, whether creating a simulation or running one. While you can use a trackpad to navigate inside the scene, a mouse is highly recommended.**

You can pan the scene by pressing LMB and dragging. If you hold down SHIFT in addition, you can now rotate the view. You can zoom in and out by using the mouse scroll or pressing and dragging the RMB.



## **Side panel**

The side panel on the left consists of three tabs: **World**, **Insert**, and **Layers**.

### **World**

This tab displays the lights and models currently in the scene. By clicking on individual model, you can view or edit its basic parameters like position and orientation. In addition, you can also change the physics of the scene like gravity and magnetic field via the Physics option. The GUI option provides access to the default camera view angle and pose.

### **Insert**

This is where you will find objects (models) to add to the simulation scene. Left click to expand or collapse a list/directory of models. To place an object in the scene, simply left click the object of interest under the Insert tab; this will bind the object to your mouse cursor and now you can place it anywhere in the scene by left clicking at that location.

### **Layers**

This is an optional feature, so this tab will be empty in most cases.

## **Top Toolbar**

Next we have a toolbar at the top, it provides quick access to some cursor types, geometric shapes, and views.

### **Select mode**

The most commonly used cursor mode, allows you to navigate the scene.

### **Translate mode**

One way to change an object's position is to select the object in world tab on the side panel and then change its pose via properties. This is cumbersome and also unnatural, the translate mode cursor allows you to change the position of any model in the scene. Simply select the cursor mode and then use proper axes to drag the object around until satisfied.

### **Rotate mode**

Similar to translate mode, this cursor mode allows changing the orientation of any given model.

### **Scale mode**

Scale mode allows changing the scale and hence overall size of any model.

### **Undo/Redo**

Since we humans are best at making mistakes, the undo tool helps us hide our mistakes. On the other hand if you undid something that you did not intend to, redo tool to the rescue.

## Simple shapes

You can insert basic 3D models like cubes, spheres or cylinders into the scene.

## Lights

Add different light sources like spotlight, point light or directional light to the scene.

## Copy/Paste

These tools let you copy/paste models in the scene. On the other hand you can simply press Ctrl+C to copy and Ctrl+V to paste any model.

## Align

This tool allows you to align one model with another along one of the three principle axes.

## Change view

This one is pretty useful. This tool lets you view the scene from different perspectives like topview, sideview, frontview, bottomview? Wonder how useful the bottomview is, anyways moving on.

## Bottom Toolbar

The Bottom Toolbar has a neat play/pause button. This allows you to pause the simulation for conveniently moving objects around. This toolbar also displays data about the simulation, like the simulation time and its relationship to real-life time. An FPS counter can be found here to gauge your systems performance for any given scene.

## Gazebo: Hello, world!

In **the worlds directory** of a package, we save **each individual Gazebo world**. A world is a collection of models such as **your robot**, and a specific environment. Several other physical properties specific to this world can be specified.

To create a simple world, with no objects or

models **simple\_world.world**

```
<?xml version="1.0" ?>
<sdf version="1.4">
<world name="default">
<include>
<uri>model://ground_plane</uri>
</include>
```

```

<!-- Light source -->
<include>
  <uri>model://sun</uri>
</include>

<!-- World camera -->
<gui fullscreen='0'>
  <camera name='world_camera'>
    <pose>4.927360 -4.376610 3.740080 0.000000 0.275643 2.356190</pose>
    <view_controller>orbit</view_controller>
  </camera>
</gui>

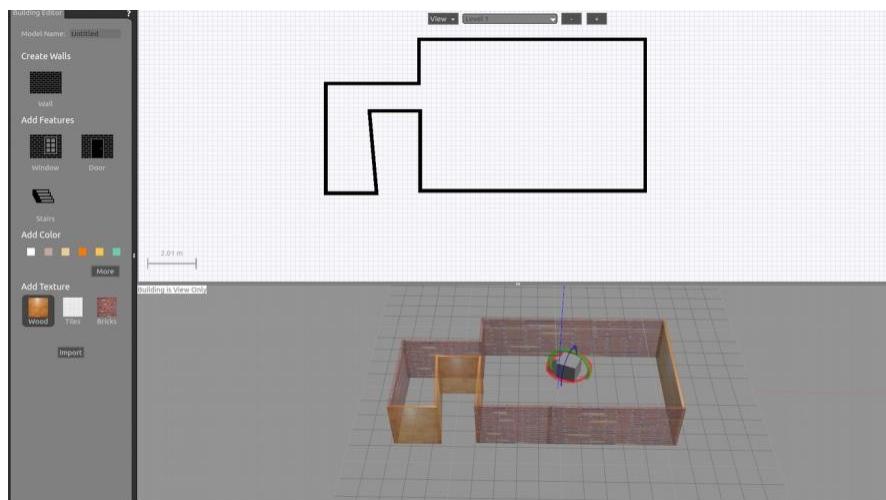
</world>
</sdf>

```

**The .world file uses the XML file format to describe all the elements that are being defined with respect to the Gazebo environment. The simple world that is created above, has the following elements -**

- **<sdf>**: The base element which encapsulates the entire file structure and content.
- **<world>**: The world element defines the world description and several properties pertaining to that world. In this example, you are adding **a ground plane, a light source, and a camera to your world**. Each **model or property** can have **further elements that describe it better**. For example, the **camera has a pose element** which defines its position and orientation.
- **<include>**: The include element, along with the **<uri>** element, provide **a path** to a particular model. In Gazebo there are several models that are included by default, and you can include them in creating your environment.

**Gazebo also provides a building editor to easily create worlds graphically**



**Now lets see a launch file that launches a robot described by URDF in a gazebo world**

```

<?xml version="1.0" encoding="UTF-8"?>
<launch>

```

```

<include file="$(find lynxbot_bringup)/launch/robot_description.launch"/>
<arg name="world" default="empty"/>
<arg name="paused" default="false"/>
<arg name="use_sim_time" default="true"/>
<arg name="gui" default="true"/>
<arg name="headless" default="false"/>
<arg name="debug" default="false"/>

<include file="$(find gazebo_ros)/launch/empty_world.launch">
  <arg name="world_name" value="$(find lynxbot_bringup)/worlds/simple_world.world"/> <arg
  name="paused" value="$(arg paused)"/>
  <arg name="use_sim_time" value="$(arg use_sim_time)"/> <arg
  name="gui" value="$(arg gui)"/>
  <arg name="headless" value="$(arg headless)"/> <arg
  name="debug" value="$(arg debug)"/> </include>

<!--spawn a robot in gazebo world-->

<node name="urdf_spawner" pkg="gazebo_ros" type="spawn _model" respawn="false"
output="screen" args="-urdf -param robot_description -model lynxbot_bringup"/>

</launch>
```

As in the case of the .world file, the **.launch files are also based on XML**. The structure for the file above, is essentially divided into three parts -

- First, you **define certain arguments using the <arg> element**. Each such element will have a **name attribute and a default value**.
- Then, you **include the empty\_world.launch file** from the **gazebo\_ros package**. The empty\_world file includes **a set of important definitions that are inherited by the world that we create**. Using the **world\_name argument and the path to your .world file** passed as the value to that argument, you will be able **to launch your world in Gazebo**.
- Last, we **include the robot\_description.launch** that **loads the URDF on the parameter server along with the joint\_state\_publisher and robot\_state\_publisher**, then with the **urdf\_spawner node** we **spawn the model from the URDF that robot\_description helps generate to the gazebo simulation**.

**As a roboticist or a robotics software engineer, building your own robots is a very valuable skill and offers a lot of experience in solving problems in the domain. Often, there are limitations around building your own robot for a specific task, especially related to available resources or costs.**

That's where simulation environments are quite beneficial. Not only do you have freedom over what kind of robot you can build, but you also get to experiment and test different scenarios with relative ease and at a faster pace. For example, you can have a simulated drone to take in camera data for obstacle avoidance in a simulated city, without worrying about it crashing into a building!

**There are several approaches or design methodologies you can consider when creating your own robot. For a mobile robot, you can break the concept down into some basic details - a robot base, wheels, and sensors.**

**To add a sensor to our robot we first have to add a sensor Link and a corresponding Joint.**

## Gazebo Plugins

Now we successfully added sensors to our robot, allowing it to visualize the world around it! But how exactly does the camera sensor takes those images during simulation? How exactly does your robot move in a simulated environment?

The URDF in itself can't help with that. However, Gazebo allows us to create or use plugins that help utilize all available gazebo functionality in order to implement specific use-cases for specific models.

Note that these plugins or gazebo for that matter are not used with the **real robot**.

Also it is common practice to put these plugins and all gazebo related stuff on a .gazebo file that we include on the xacro of the robot.

On the robot we will use these plugins -

- A plugin for the kinect sensor.
- A plugin for the lidar sensor.
- A plugin for controlling the wheel joints.

Lets see an example of a plugin that implements Differential Drive Controller

```
<gazebo>
  <plugin name="differential_drive_controller" filename="libgazebo_ros_diff_drive.so">
    <legacyMode>false</legacyMode>
    <alwaysOn>true</alwaysOn>
    <updateRate>10</updateRate>
    <leftJoint>left_wheel_hinge</leftJoint>
    <rightJoint>right_wheel_hinge</rightJoint>
    <wheelSeparation>0.4</wheelSeparation>
    <wheelDiameter>0.2</wheelDiameter>
    <torque>10</torque>
    <commandTopic>cmd_vel</commandTopic>
    <odometryTopic>odom</odometryTopic>
    <odometryFrame>odom</odometryFrame>
    <robotBaseFrame>robot_footprint</robotBaseFrame>
  </plugin>
</gazebo>
```

**libgazebo\_ros\_diff\_drive.so** is the shared object file created from compiling some C++ code. The plugin takes in information specific to your robot's model, such as wheel separation, joint names and more, and then calculates and publishes the robot's odometry information to the topics that you are specifying above, like the **odom** topic. It is possible to send velocity commands to your robot to move it in a specific direction. This controller helps achieve that result.

Gazebo already has several of such plugins available for anyone to work with. We will utilize the preexisting plugins for the **kinect\_sensor** and the **plugins for the lidar sensor**.

## 3.16 Writing a Gazebo service

First of all **services are Synchronous** meaning that the program can't continue until it receives a response. On the other hand **Actions are Asynchronous** and the main program can continue, it's like a new thread of execution.

An example of a service for deleting a Gazebo model

```
#include "ros/ros.h"
#include "gazebo_msgs/DeleteModel.h"

int main(int argc, char** argv){

    ros::init(argc, argv, "service_client");
    ros::NodeHandle nh;
    // Create the connection to the service

    ros::ServiceClient delete_model_service =
        nh.serviceClient<gazebo_msgs::DeleteModel>("/gazebo/delete_model");

    gazebo_msgs::DeleteModel srv;
    srv.request.model_name = "bowl_1";
    if (delete_model_service.call(srv)){
        ROS_INFO("%s", srv.response.status_message.c_str());
    }
    else{
        ROS_ERROR("Failed to call service");
        return 1;
    }
    return 0;
}

ros::ServiceClient delete_model_service =
    nh.serviceClient<gazebo_msgs::DeleteModel>("/gazebo/delete_model");
```

This creates a client to call the service DeleteModel. The ros::ServiceClient object is used to call the service later on.

Some useful commands

```
rosservice list
rosservice info /name_of_your_service
rosservice call /the_service_name
rossrv show name_of_pkg/name_of_service
```

**service messages have two parts**

REQUEST  
string model\_name

---

RESPONSE  
bool success  
string status\_message

**Note that with this code we call an existing service.**

Ofcourse in a similar fashion to the topics with some modifications we can create our own costum service messages. Carefull with the 3 dashes it's important to distinguish the variables regarding the Response part of the message to those regarding the Request part.

**Here is an example of creating a simple service that when called displays a message**

```
#include "ros/ros.h"
#include "std_srvs/Empty.h"

bool my_callback(std_srvs::Empty::Request &req, std_srvs::Empty::Response &res)
{
    //e.g req.some_variable= req.some_variable + req.other_variable;
    ROS_INFO("My callback has been called");
}

int main(int argc, char** argv){
    ros::init(argc, argv, "service_server");
    ros::NodeHandle nh;
    // Here the service is created and advertised over ROS.
    ros::ServiceServer my_service = nh.advertiseService("/my_service", my_callback);
    ros::spin()
    return 0;
}
```

**e.g rosservice call /my\_service**

## 3.17 Actions

**Actions unlike Service are asynchronous and can provide feedback.**

**A node that provides the functionality has to contain an action server allowing other nodes to call these functionality. The node that calls has to contain an action client.**

<b>Action</b>	<b>----- goal -----&gt;</b>	<b>Action</b>
<b>Client</b>	<b>----- cancel -----&gt;</b>	<b>Server</b>
	<b>&lt;---- status -----</b>	
	<b>&lt;---- result -----</b>	
	<b>&lt;---- feedback ---</b>	

example of an action message

```
#goal
int32 nseconds
---
```

```
#result
sensor_msgs/CompressedImage[] allPictures
-----
#feedback
sensor_msgs/CompressedImage[] lastImage
```

## Example of a Simple Action Server

```
#include <ros/ros.h>
#include <actionlib/server/simple_action_server.h>
#include <actionlib_tutorials/FibonacciAction.h>

class FibonacciAction
{
protected:

    ros::NodeHandle nh_;
    actionlib::SimpleActionServer<actionlib_tutorials::FibonacciAction> as_; // NodeHandle instance must be created before this line. Otherwise strange error occurs.
    std::string action_name_; //create messages that are used to published feedback/result
    actionlib_tutorials::FibonacciFeedback feedback_;
    actionlib_tutorials::FibonacciResult result_;

public:

    FibonacciAction(std::string name) :
        as_(nh_, name, boost::bind(&FibonacciAction::executeCB, this, _1), false),
        action_name_(name)
    {
        as_.start();
    }

    ~FibonacciAction(void)
    {
    }

    void executeCB(const actionlib_tutorials::FibonacciGoalConstPtr &goal)
    {
        // helper variables
        ros::Rate r(1);
        bool success = true;

        / push_back the seeds for the fibonacci sequence
        feedback_.sequence.clear();
        feedback_.sequence.push_back(0);
        feedback_.sequence.push_back(1);

        / publish info to the console for the user
        ROS_INFO("%s: Executing, creating fibonacci sequence of order %i with seeds %i, %i",
        action_name_.c_str(), goal->order, feedback_.sequence[0], feedback_.sequence[1]);

        // start executing the action
        for(int i=1; i<=goal->order; i++)
        {
            / check that preempt has not been requested by the client
            if (as_.isPreemptRequested() || !ros::ok())
            {
```

```

ROS_INFO("%s: Preempted", action_name_.c_str());
/ set the action state to preempted
as_.setPreempted();
success = false;
break;
}
feedback_.sequence.push_back(feedback_.sequence[i] + feedback_.sequence[i-1]); // publish the
feedback
as_.publishFeedback(feedback_);
/ this sleep is not necessary, the sequence is computed at 1 Hz for demonstration purposes
r.sleep();
}

if(success)
{
    result_.sequence = feedback_.sequence;
    ROS_INFO("%s: Succeeded", action_name_.c_str());
    / set the action state to succeeded
    as_.setSucceeded(result_);
}
};

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

FibonacciAction fibonacci("fibonacci");
ros::spin();
return 0;
}

```

## Writing a simple client

```

#include <ros/ros.h>
#include <actionlib/client/simple_action_client.h>
#include <actionlib/client/terminal_state.h>
#include <actionlib_tutorials/FibonacciAction.h>

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

/ create the action client
/ true causes the client to spin its own thread
actionlib::SimpleActionClient<actionlib_tutorials::FibonacciAction> ac("fibonacci", true);
ROS_INFO("Waiting for action server to start.");

/ wait for the action server to start
ac.waitForServer(); //will wait for infinite time

ROS_INFO("Action server started, sending goal.");
// send a goal to the action
actionlib_tutorials::FibonacciGoal goal;
goal.order = 20;

```

```

ac.sendGoal(goal);

//wait for the action to return
bool finished_before_timeout = ac.waitForResult(ros::Duration(30.0));

if (finished_before_timeout)
{
    actionlib::SimpleClientGoalState state = ac.getState();
    ROS_INFO("Action finished: %s",state.toString().c_str());
}
else
    ROS_INFO("Action did not finish before the time out.");

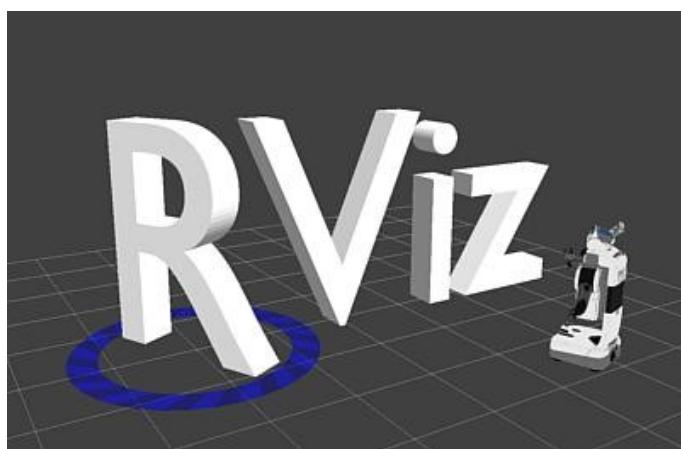
//exit
return 0;
}

```

**Try to see if you understand this code, see the ROS tutorials for more explanation of how this code works. We will use actions, mainly calling them in next chapters.**

## 3.18 RVIZ

### What is Rviz?



RViz (or rviz) stands for ROS Visualization tool or ROS Visualizer. RViz is our one stop tool to **visualize** all three core aspects of a robot: **Perception, Decision Making, and Actuation**. Using rviz, you can visualize any type of sensor data being published over a ROS topic like camera images, point clouds, ultrasonic measurements, Lidar data, inertial measurements, etc. This data can be a live stream coming directly from the sensor or pre-recorded data stored as a bagfile.

You can also visualize live joint angle values from a robot and hence construct a real-time 3D representation of any robot. Having said that, **RVIZ is not a simulator and does not interface with a physics engine, in other words no collisions and no dynamics. RVIZ is not an**

**alternative to Gazebo but a complementary tool to keep an eye on every single process under the hood of a robotic system.**

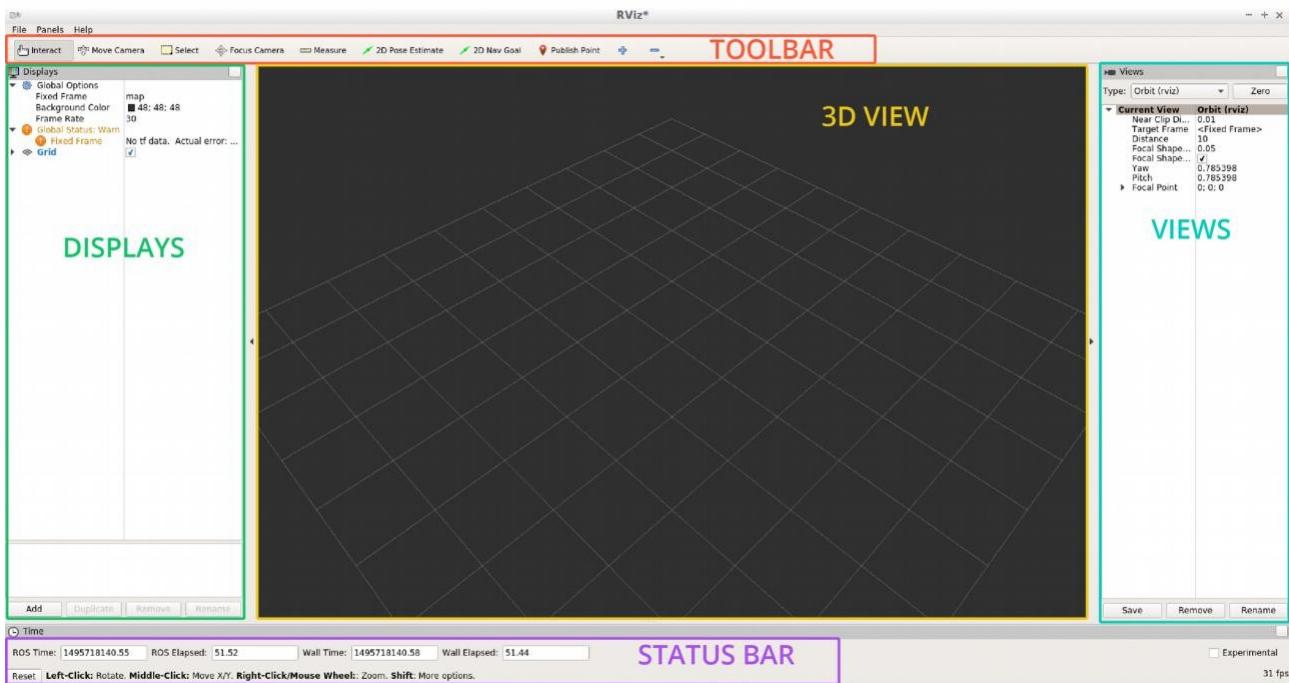
Since rviz is a ROS package, you need roscore running to launch rviz. In a terminal spin up roscore:

**\$ roscore**

In another terminal, run rviz:

**\$ rosrun rviz rviz**

Once properly launched, rviz window should look something like this:



**The empty window in the center is called 3D view, this is where you will spend most of your time observing the robot model, sensor visualization and other meta-data.**

The panel on the left is a list of loaded **Displays**, while the one on the right shows different **Views** available.

On the top we have a number of useful **tools** and bottom bar displays useful information like time elapsed, fps count, and some handy instructions/details for the selected tool.

## Displays

For anything to appear in the **3D view**, you first need to **load a proper display**. A display could be as simple **as a basic 3D shape or a complex robot model**.

Displays can also be used to **visualize sensor data streams like 3D pointclouds, lidar scans, depth images, etc.**

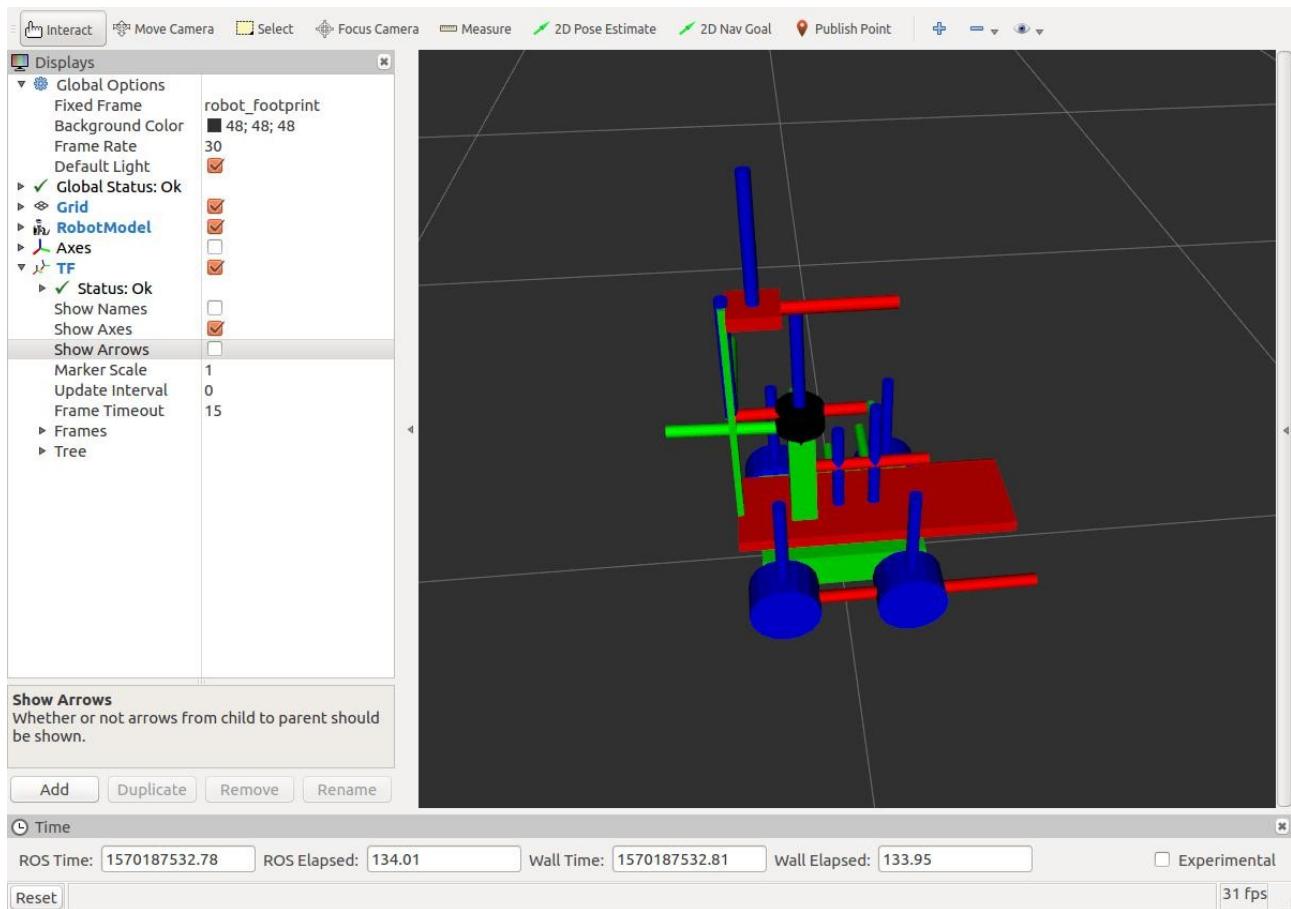
Rviz by default starts up with two fixed property fields that cannot be removed - **Global Options** and **Global Status**. While these are not displays, **one governs simple global settings**, while the other **detects and displays useful status notifications**.

Let us play around with a few basic displays. As you can see we already have a **Grid** display loaded and enabled. To **add a robot model display**, you first need to load the robot description (remember urdf?) into the parameter server and publish transform between all the robot links. Luckily, we have a convenient launch file that does all of this for us, open a new terminal and type in the following command:

```
$ roslaunch lynxbot Bringup robot_description.launch
```

Now let's go back to the rviz window and add a display by clicking the Add button at the bottom. This will bring up a new window with display types, select **RobotModel** and hit Ok.

Next change the Fixed Frame under Global Options from world (or map) to base\_link. If everything works fine, you should be able to see the robot model:



Here we see our robot model along with the TF axes.

You can disable a display type without having to remove it completely by simply unchecking the check-box right next to it. Enable again by checking the check-box.

Remember for most display types to work, you need to load up a corresponding source.

## Views

We have several view types in rviz which basically change the camera type in the 3D view. Remember for each view type, instructions on how to rotate, pan, and zoom using your mouse can be seen at the bottom status bar. We will discuss three widely used camera types here:

### **Orbit**

In the orbit view, you set a Focal point and the camera simply rotates around that focal point always looking at it. While moving, you can see the focal point in the form of a yellow disk.

### **FPS**

FPS view is a first person camera view. Just like a FPS video game, the camera rotates as if rotating about your head.

### **TopDownOrtho**

This camera always looks down along the global Z axis, restricting your camera movement to the XY plane. Mostly used while performing 2D navigation for mobile robots.

## **Toolbar**

Next we will explore some of the tools present in the top toolbar on the rviz window. While most of these tools are generic and can be used for any robot, some are specific to mobile robot navigation.

### **Move camera**

This is the most basic and often default tool used to, as you guessed it, move the camera. Remember that movement control changes from one view type to another.

### **Select**

To demonstrate this tool, let us add a new panel called the “Selection Panel”. To do this, click on “Panels” on the top menu of rviz window and then click on “Selection”, you should now see an empty Selection panel on top of the displays panel. Now select the “Select” tool from the toolbar. Using this tool you can select a single item by left clicking it or box select multiple items by clicking and dragging. Details of your selection are displayed in the Selection panel. e.g. selecting a robot link will provide you with its current Pose (Position + Orientation). This tool comes in handy in complex environments, where you want to determine the properties or status of one or more items.

### **Focus camera**

As the name suggests, this tool allows you to bring any item or part of a robot in ”focus”, meaning at the center of the 3D view panel by left clicking it once.

### **Measure**

Upon activation this tool allows you to measure the distance between two points in the 3D view. You can set the starting point by one left click and then set the end point by second left click. The measured distance between these two points will be shown at the bottom status bar of the rviz

window. Remember though, you cannot measure the distance between two points in the empty space. You must have some object present for you to click.

## Interact

This is a special tool that can be used to interact with interactive markers. We will see some interactive markers on next chapters

## 3.19 MOVEIT

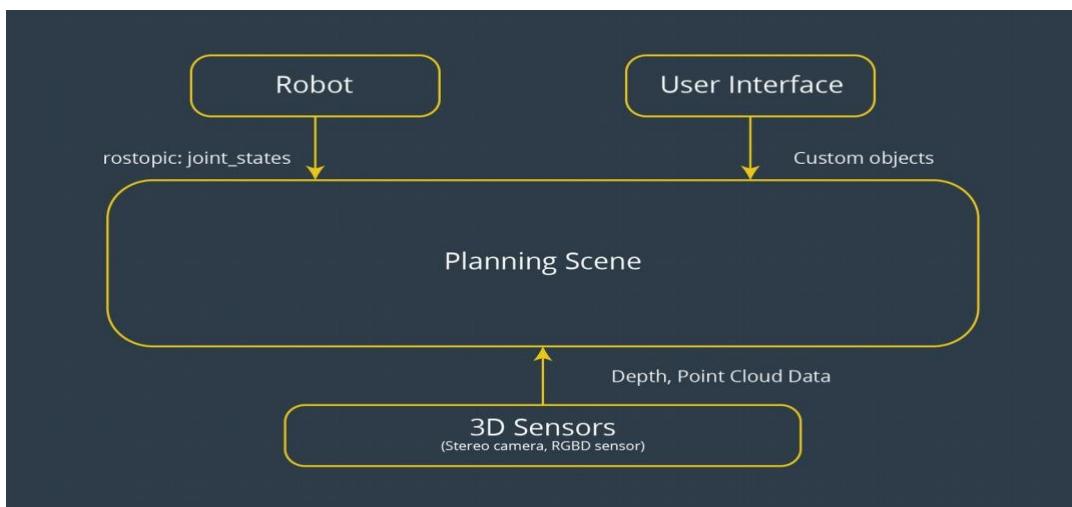


**MoveIt!** is an advanced motion planning framework for manipulation, kinematics, and control. It provides a platform for developing advanced robotics applications, evaluating new robot designs and building integrated robotics products for industrial, commercial, R&D, and other domains.

## The planning Scene

The planning scene represents the state of the robot as well as the state of the world around it.

It tracks the robot state by subscribing to the `joint_states` topic. It integrates information from various sensors like Stereo cameras, and/or RGBD cameras to essentially create a 3D map of the dynamic environment around the robot.



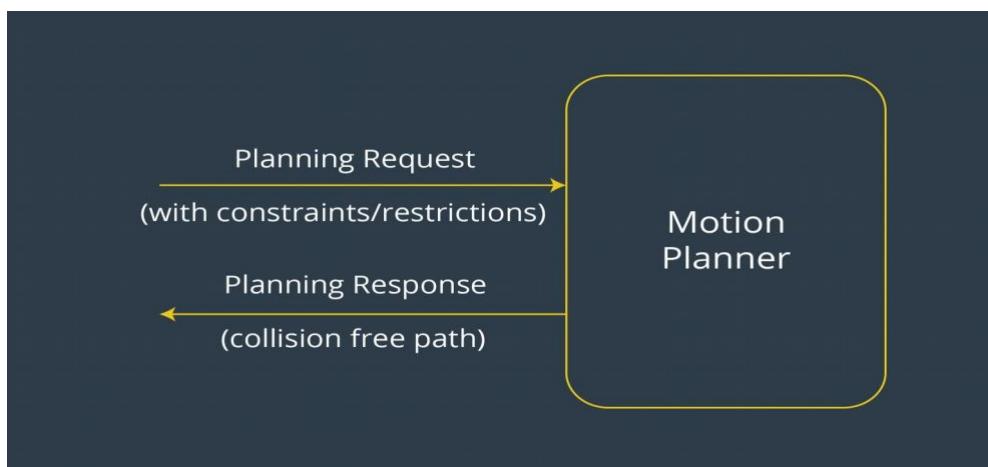
You may also insert abstract or virtual objects into this so called 3D map by publishing them to specific topics.

## Motion Planning

MoveIt! **does not have an inbuilt motion planning algorithm**. Instead it provides a convenient plugin interface to communicate with and use various motion planning libraries. Moveit! utilizes a special ROS Service to establish a request-response relationship with any given motion planner.

As we discussed, unlike pub-sub, request-response relationship relies on very specific messages being passed between two nodes. In this case, once **a motion planner is selected and loaded via the plugin interface**, you can send a request message to the planner with specific instructions on how to generate the plan.

To understand this, imagine you are at a fancy restaurant, you would like to order a specific dish but have dietary constraints/preferences, so you order the dish but also add in instructions like *make it less spicy* or *leave out the nuts*. Moveit! does something similar, in the request message for plan, **you can add instructions like path constraints, trajectory constraints, time allowed for the planner to plan, maximum velocity allowed, etc.**



In response to our request message, Moveit! generates a desired collision free trajectory using the Planning Scene discussed above.

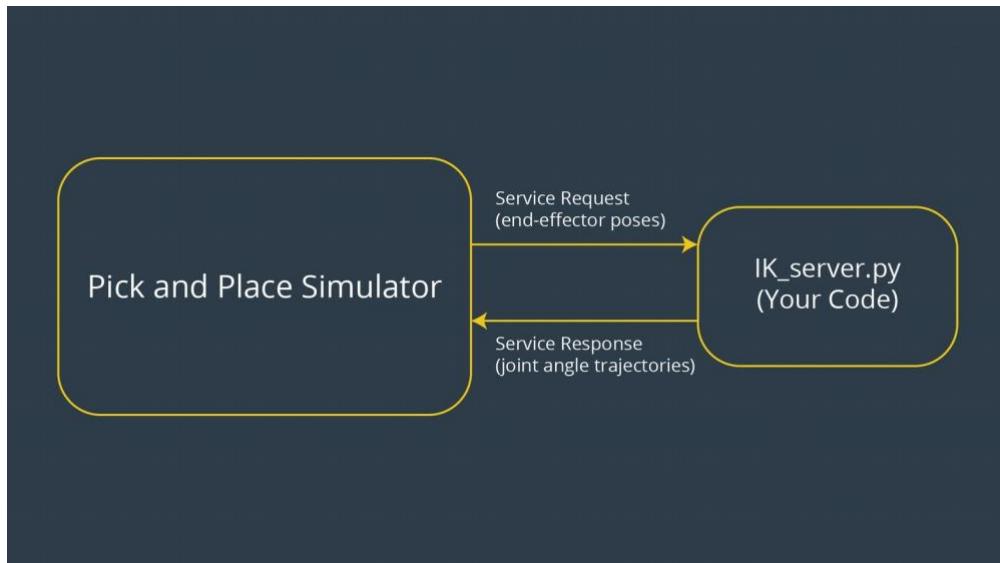
**A point to be noted is that a motion planner only generates a collision free path from the start state of the robot to its end state, and then Moveit! converts this path into a joint space trajectory which obeys velocity and acceleration constraints for joint angles.**

Once you have a valid collision free trajectory, you can **execute it using proper controllers** for your robot.

## Kinematics

At various stages in the process of motion planning, Moveit! needs to convert the joint\_state of the robot into the end-effector pose using Forward Kinematics, and vice-versa using Inverse Kinematics.

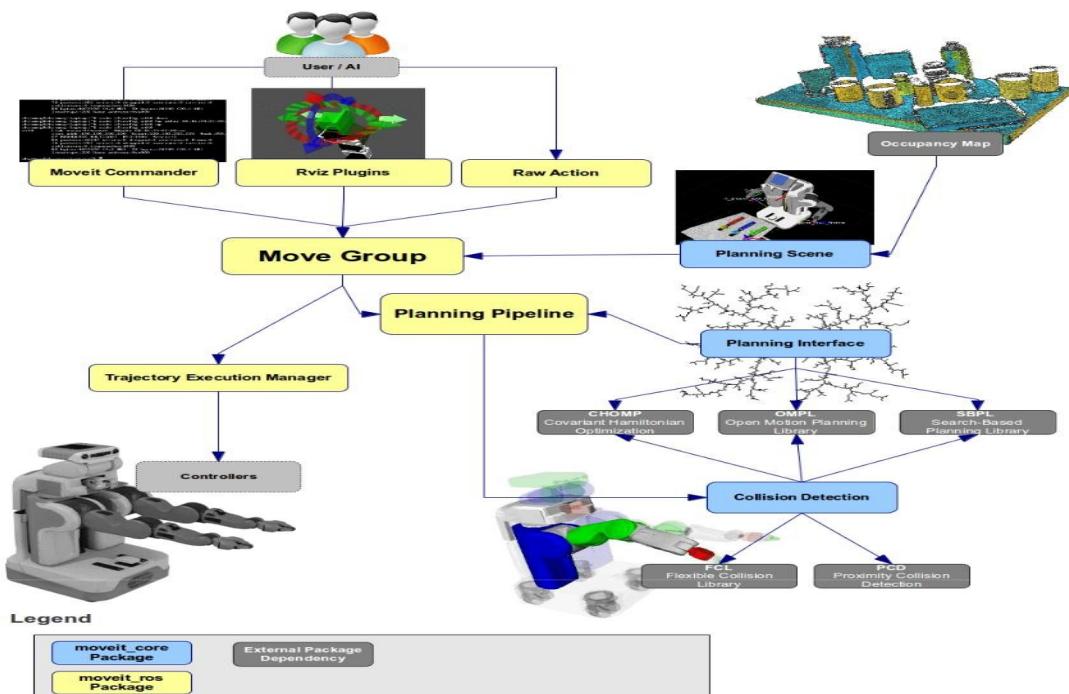
While ROS's **RobotState** class provides access to Forward Kinematics functionality, Moveit! Allows users to write their own Inverse Kinematics implementations.



While your potential python script only solves IK for a specific 6DOF robot arm, there are generalized libraries and plugins which can be used to solve IK for different types of robots. A couple of widely used solutions are:

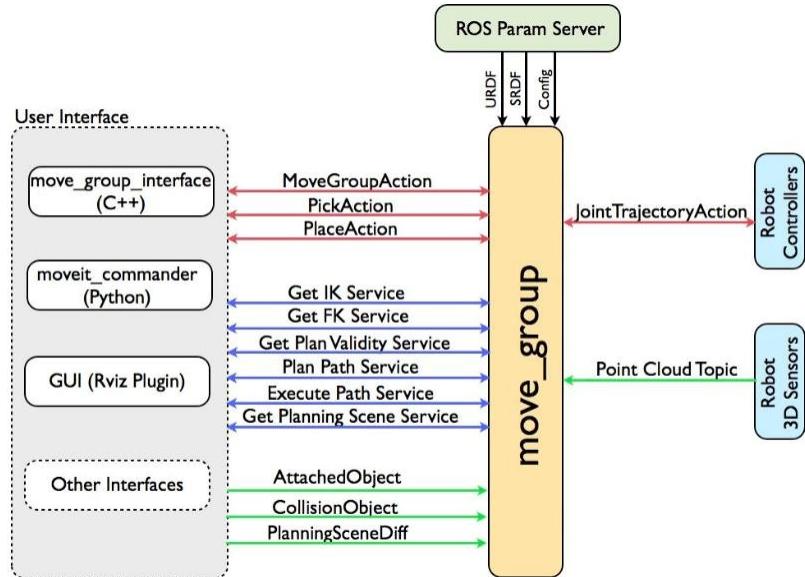
- [The Kinematics and Dynamics Library \(KDL\)](#) developed by orocos
- [IKFast Solver](#) by OpenRAVE

### Quick High level diagram of System Architecture



## The move\_group node

The figure above shows the high-level system architecture for the primary node provided by MoveIt called move\_group. This node serves as an integrator: pulling all the individual components together to provide a set of ROS actions and services for users to use.



We will use MoveIt to control a robotic arm connected to our mobile base and hopefully grab some objects and move them around!!

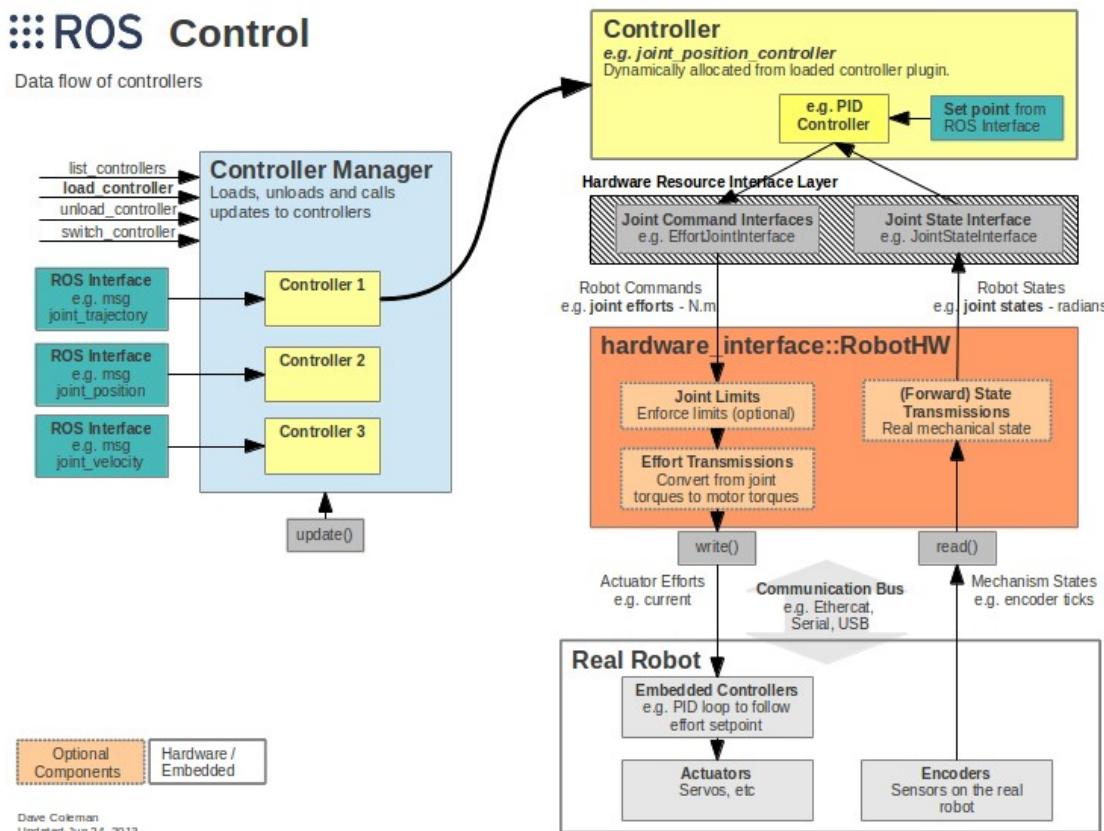
## 3.20 ROS Control

As we saw earlier MoveIT can provide us with the path to move for example a serial manipulator (e.g robotic hand), **but it can not actually implement this movement on the Gazebo simulation or on the real robot**, except only on the internal simulation that MoveIT uses. To do that, we need **to write controllers** that can take a input or setpoint and provided the necessary command or output that will be execute on the simulation or the real robot.

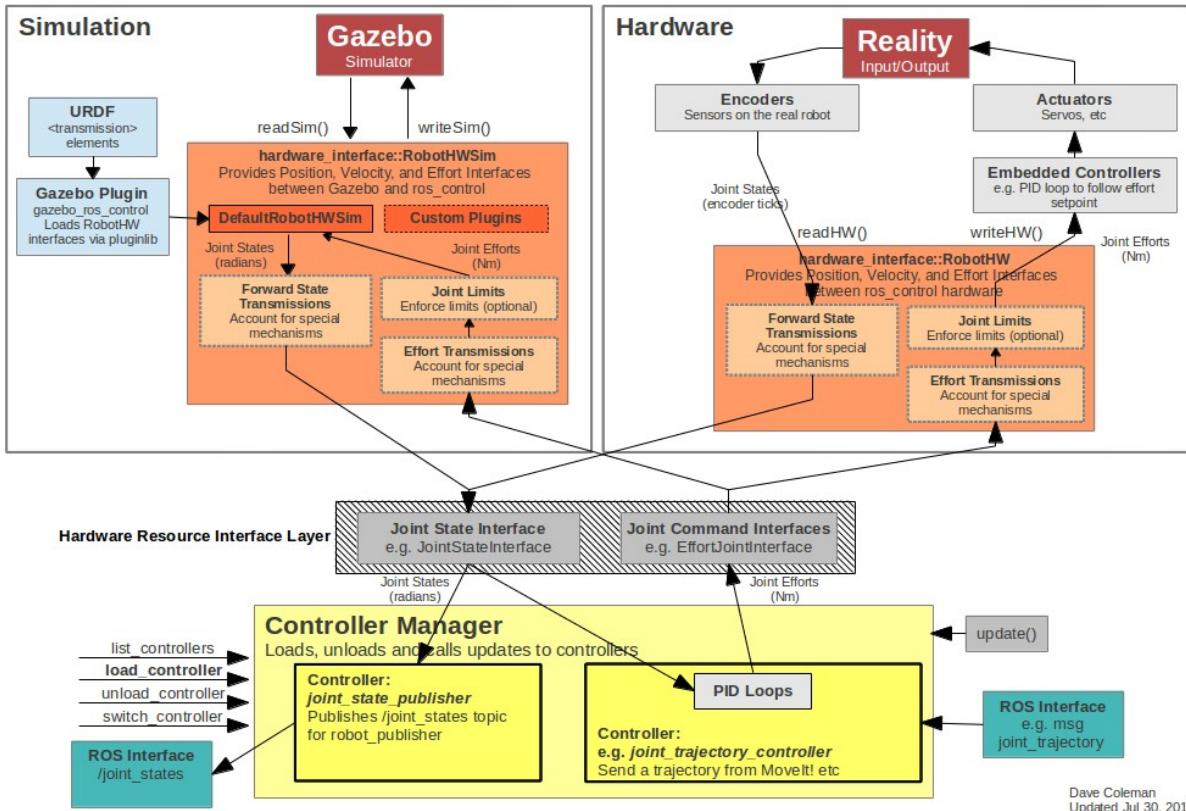
To achieve that ROS provide us with a package called ros\_control

- A set of packages that include controller interfaces, controller managers, transmissions and hardware\_interfaces.
- The ros\_control packages are a rewrite of the [pr2\\_mechanism](#) packages to make controllers generic to all robots beyond just the PR2.
- The ros\_control packages takes as **input the joint state data** from your robot's actuator's encoders and an input set point. It uses a generic control loop feedback mechanism, typically a PID controller, to control the output, typically effort, sent to your actuators. ros\_control gets more complicated for physical mechanisms that do not have one-to-one mappings of joint positions, efforts, etc but these scenarios are accounted for using transmissions.

Below we can see some pictures that defined how ros control is implemented on a macro level.



# GAZEBO + ROS + ros\_control



I will briefly explain what we see above, there are generic controllers that can take as **an input a setpoint effort, velocity or position commands** and provide a output or command to be execute on the robot. The controller manager “manages” these controllers e.g load, unload, switch between different controllers and implements their control circles.

These controllers are **robot agnostic** up to this point and need a **hardware interface** to communicate with the actual hardware on the real robot or in the simulation.

ROS control provide us with a set of tools to help us with that, mainly in the simulation it provide us with the **ros\_control plugin** for an easy and standard way to communicate with the gazebo simulation without writing any code, of course you can write your custom hardware interface if that’s necessary.

For the real robot **you need to write the hardware interface**, but still ros control helps you with that by providing a **class that you can inherit from**, but you need to write the **read and write functions** yourself since that are specific to your robot and **communication protocol** (e.g rosserial, ros industrial, CAN-bus etc..)

**A note here is that ros control provide us with a generic controller namely the joint\_trajectory\_controller that can control several joints together, and is mainly used to control serial manipulators with MoveIT because it provides the action server that MoveIT expects.**

You can read more about ros control at

- [wiki ros control](#)
- [ROSCon talk](#)

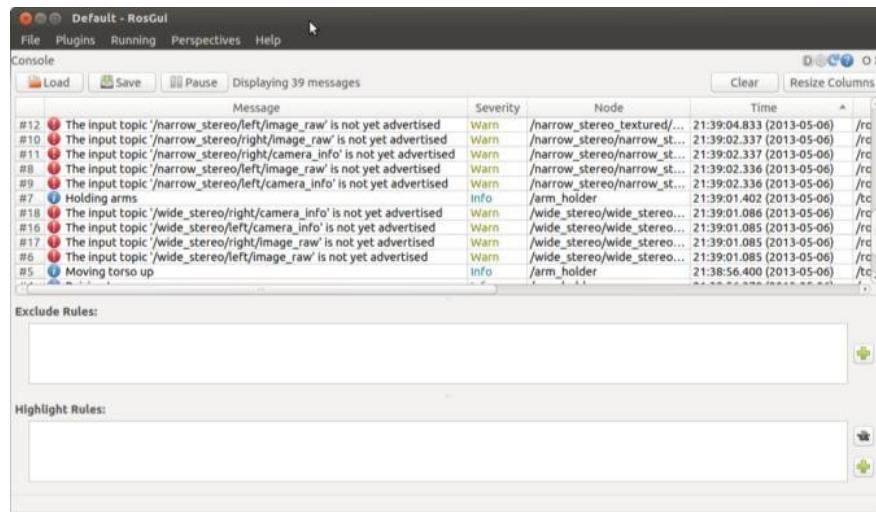
## 3.21 Debugging Tools

Over the years, developers have built an amazing suite of tools for debugging and introspecting the ROS system.

The first step to building a good ROS toolkit is taking care of your terminal situation, but that's just the beginning. For good ROS debugging, you need to use tools that can analyze messages, show you the terminal outputs from various nodes, and just handle the general complexity that ROS brings to the table. Many of the best tools are installed by default with the `ros-*desktop-full` standard installation, so they're available to you on most ROS machines!

### rqt\_console

allows you to quickly filter messages by what node publishes them, what text they contain, their severity (such as only showing errors and ignoring all warnings), and more. If you are working with bugs that only appear when you have many nodes running, it can often be hard to pick out what the exact problems are and their associated error messages. By setting up rqt\_console with heavy filtering, you can ensure that you only see the messages that matter. Run it with a simple `rqt_console` in the terminal.



### rqt\_launchtree

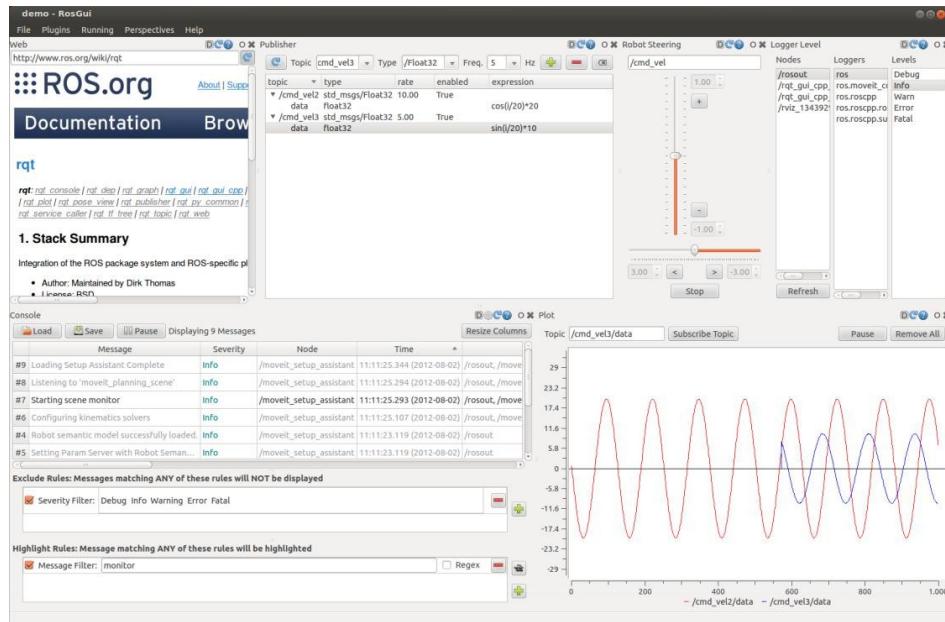
Launch files often end up nested three or four layers deep as you try to keep your code modular and clean. As you start calling launch files in MoveIt or the ROS navigation stack, the launch file include structure can get even more convoluted, making debugging a huge pain. RQT Launchtree is designed to help fix this problem. By parsing launch files, it allows you to drill down through multiple nested launch file includes to see exactly what's going on. It has support for parameters and arguments as well.

## rqt\_plot

It is used to graphically see topics in real time, we will use to see the real time velocity of our robot in real time, and tune the pid parameters accordingly

## rqt

The general-purpose rqt console (run from the terminal with just rqt) can be used to arrange any rqt-compatible GUI elements together. That includes other tools you may have used before, such as rqt\_plot, and you can even add an RViz window or web browser as a panel. You can build your custom layout however you'd like, and then save it for reuse.



## rosnode info

Most commonly used for rosnode list or rosnode kill. If you went through all the ROS tutorials, you may have learned that you can also run rosnode info <node\_name> to get a list of a node's subscribed topics, advertised topics, and services. **It is a good basic diagnostic tool. It's much faster than pulling up another tool such as rqt\_graph and hunting for your node's connections, it also show services too.**

## rqt\_top

rqt\_top performs a similar function to the top command in the Linux terminal, showing what system resources each running node is currently using.

Node	PID	CPU %	Mem %	Num Threads
/rqt_gui_py_node_20010	20010	5.10	1.59	6
/camera_nodelet_manager	5555	2.00	0.95	18
/camera_base_link	6407	1.00	0.11	5
/camera/depth_registered/metric_rect	6104	1.00	0.15	5
/camera_base_link1	6492	1.00	0.11	5
/camera_base_link3	6670	1.00	0.11	5
/camera/depth/points	5915	1.00	0.15	5
/camera/driver	5562	0.00	0.15	5
/camera/depth/metric	5803	0.00	0.15	5
/camera/lr/rectify_ir	5722	0.00	0.15	5
/camera/register_depth_rgb	5971	0.00	0.15	5
/camera/depth/metric_rect	5759	0.00	0.15	5
/camera/depth/rectify_depth	5744	0.00	0.15	5
/camera/depth_registered/rectify_depth	6038	0.00	0.15	5
/camera/rgb/rectify_mono	5643	0.00	0.15	5
/rosout	3007	0.00	0.11	5
/camera/rgb/rectify_color	5680	0.00	0.15	5
/camera/depth_registered/metric	6182	0.00	0.15	5
/camera_base_link2	6599	0.00	0.11	5
/camera/points_xyzrgb_depth_rgb	6227	0.00	0.15	5
/camera/disparity_depth	6292	0.00	0.15	5
/camera/rgb/debayer	5589	0.00	0.15	5
/camera/disparity_depth_registered	6346	0.00	0.15	5

## **roswtf**

Perhaps the most...creatively...named ROS tool, roswtf can be a lifesaver when you're debugging complex ROS-specific issues, such as two nodes that aren't communicating with each other, two nodes publishing to the same topic, or issues with your TF tree. It also diagnoses your environment variables and folder structure. Try it out while your whole constellation of launch files and nodes is running, and see what it comes up with. Run it with roswtf, straight from the terminal.

# 4. Localization Techniques

## 4.1 Introduction

In this chapter we will introduce the concept of localization, what are the challenges in localization and what are the algorithms that are used in localization. Furthermore we will expand on the theory behind the two most popular localization algorithms the Kalman Filter and the Monte Carlo Localization. We will program them in C++ and lastly how to incorporate them with the Robot Operating System.

(Here or in the appendix some parts)

Let's begin.

## 4.2 What is localization

**Localization is the challenge of determining your robot's pose in a pre-mapped environment, we do this by implementing a probabilistic algorithm to filter noisy sensor measurements and track the robot's position and orientation.**

The robot's **starting pose is usually unknown**, as the robot **moves around and takes measurements** it tries to figure out where it can be positioned in the map. Since this is a probabilistic model the robot might have a few guesses as to where it is located. However over time, it should hopefully **narrow down on a precise location**. Now we can say that we identified the robot pose. In our case (**x coordinate, y coordinate and θ orientation**).

### Localization algorithms

- **Extended Kalman Filter** : most common Gaussian filter that helps in estimating the state of non-linear models
- **Markov localization** : a base filter localization algorithm, Markov maintains a probability distribution over the set of all possible position and orientation the robot might be located at
- **Grid localization** : histogram filter, since it is able of estimating the robot pose using grids
- **Monte Carlo localization** : also known as particle filter since it's capable of estimating the robot's pose using virtual particles.

## 4.3 Localization challenges

There are 3 different types of localization problems. The **amount of information present** and the **nature of the environment (static = always matches the ground truth map or dynamic)** that a robot is operating in determine the difficulty of the localization task.

- The easiest is called **Position Tracking** or **Local Localization**. In this case the robot knows its initial pose and the localization challenge entails at estimating the robot's pose as it moves out on the environment, this is not as trivial as you might think since there is always some uncertainty in robot motion, however the uncertainty is limited to regions surrounding the robot.
- A more complicated problem is **Global Localization**, in this case the robot's initial pose is unknown and the robot must determine its pose relative to the ground truth map. The amount of uncertainty is much greater than Position Tracking.

- Finally the hardest is called the **Kidnapped Robot Problem**, this problem is similar to Global Localization except that the robot may be kidnapped at any time and moved into a new location in the map. Although robot's getting kidnapped is quite uncommon, you can think of it as a worst case scenario. Localization algorithms are not free from error and there are instances of a robot miscalculating where it is. If the robot can recover after being "kidnapped" it means that you have a quite robust algorithm.

## 4.4 Kalman Filter

### 4.4.1 Overview

**Kalman Filter**

---

- Background
- Need for a Filter
- 1-D Kalman Filter
- Multidimensional Kalman Filter

- background** : What problem it solves, how it came about and where it is being used.
- Need for a Filter** : The challenges of localization and why we need to use a filter.
- 1-D Kalman Filter** : How the algorithm works + Implementation in C ++
- Multidimensional Kalman Filter** : Implementation in C++
- Extended Kalman Filter** : A speceific variation that extends the reach of applications even further.

### 4.4.2 What's a Kalman Filter

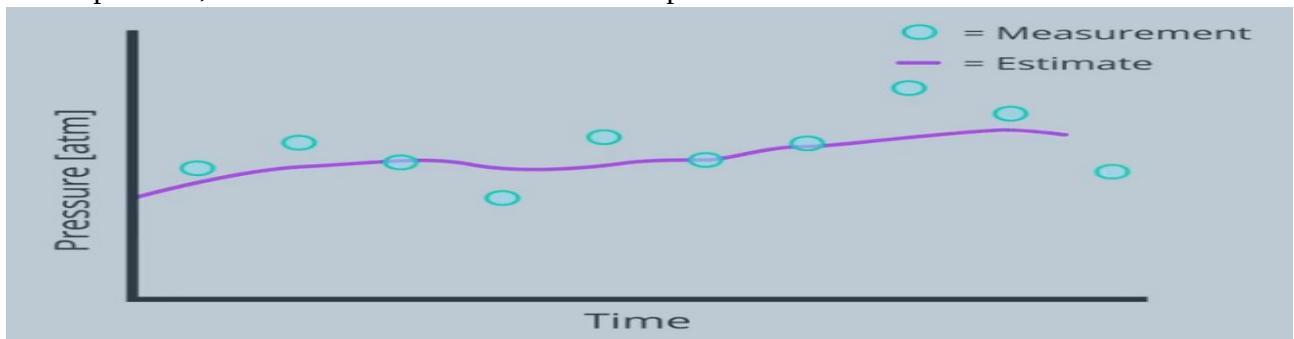
- Firstly it is an estimation algorithm that is very prominent in controls.
- It estimates the value of a variable in real time as the data is being collected (e.g position or velocity of a robot)

- Why it is noteworthy? It can take data with a lot of uncertainty or noise in the measurements and provide a very accurate estimate of the real value. Also it can do so fast, unlike other estimation algorithms it doesn't need a lot of data to come in to produce an accurate estimate.

## Example

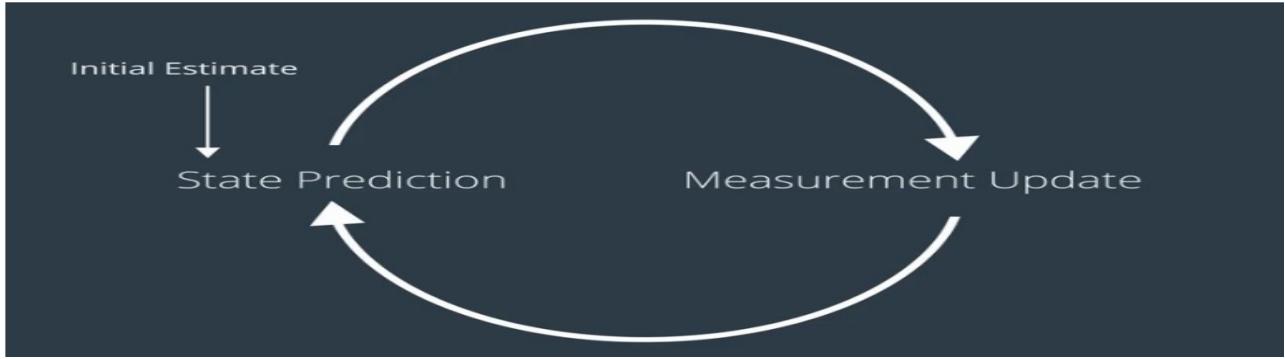


Here we are measuring the pressure that an underwater robot experiences as it swims through the water. Recall that underwater the pressure increases linearly with depth. The pressure measurements are not perfectly accurate though, there maybe be fluctuations in the water resulting in fluctuations on the pressure, as well as electrical noise from the pressure sensor itself.



As soon as the pressure sensors collects data the kalman filter begins to narrow in and estimate the actual pressure. In addition to the sensor readings the kalman filter takes into account the uncertainty of the sensor readings, which is specific to the sensor being used and the environment it is operating in.

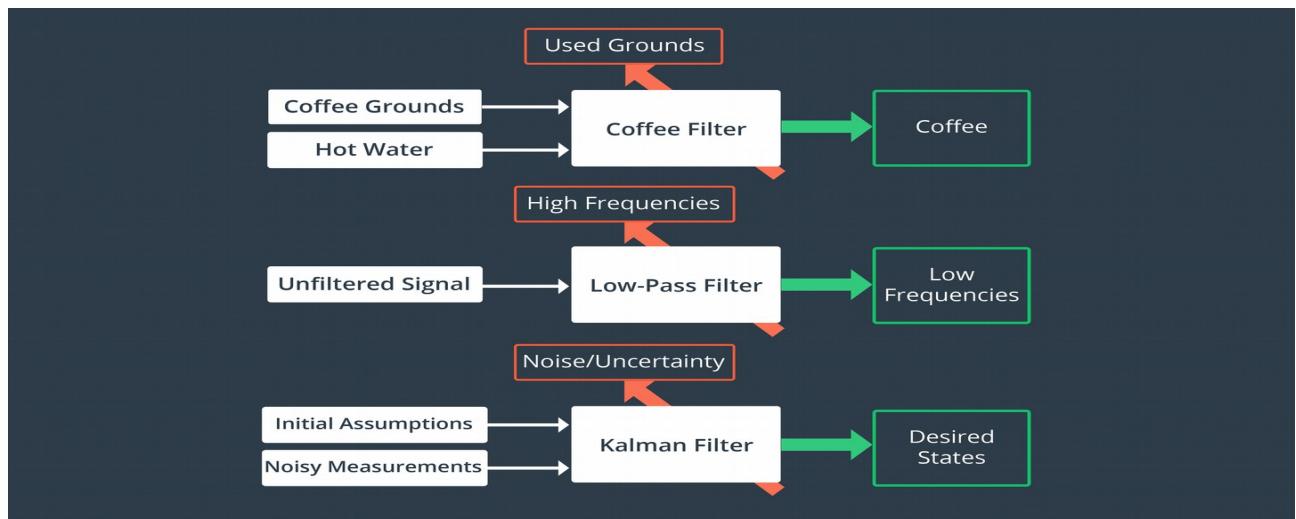
At a high level what is happening every time a measurement is being recorded?  
That's a two step process, Kalman Filter is a continuous iteration of these two steps.



- First step – Measurement Update : We use the recorded measurement to update our state.
- Second step – State Prediction : We use the information that we have about our current state to predict what the future state will be.
- At the start we use an Initial Estimate.

As we continue to iterate through these two steps it doesn't take many iterations for our estimate to converge to the real value.

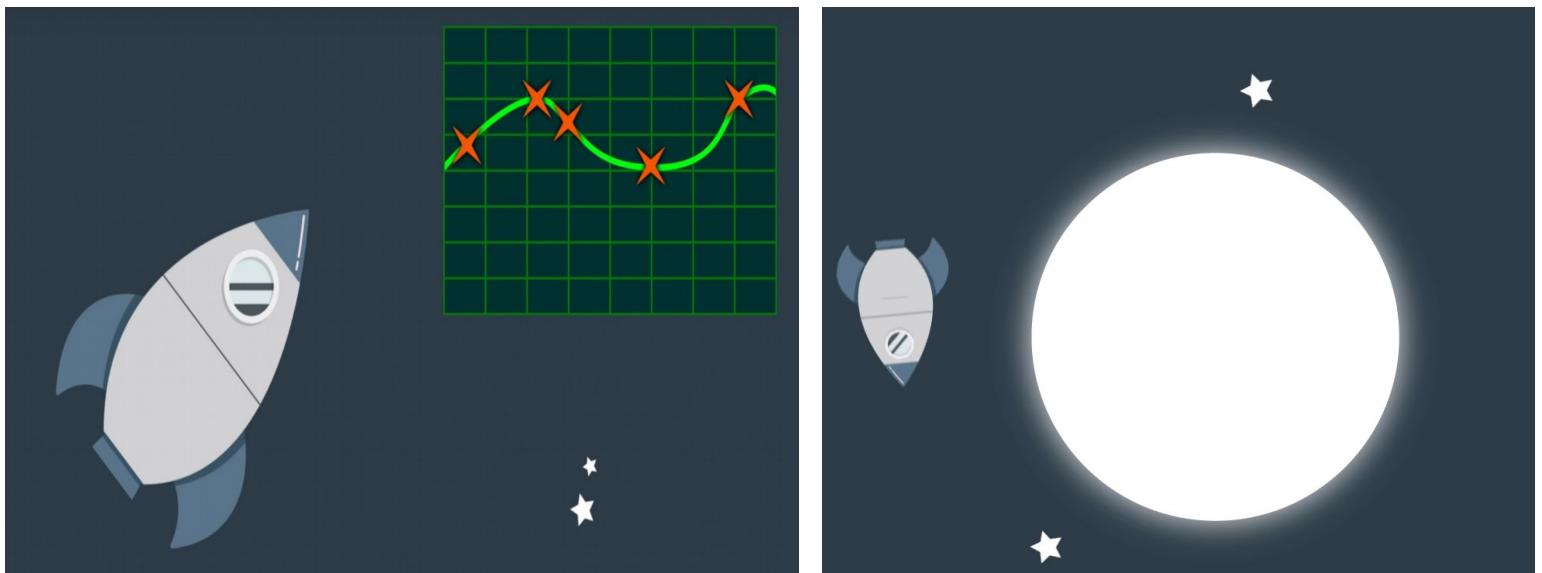
Another way of looking at a Kalman Filter is just like you'd look at any other filter. What does it take as an input, what does it filter out, and what important substance does it let through? The graphic below compares a household coffee filter, an engineering low-pass filter, and a Kalman filter.



### 4.4.3 History

The Kalman Filter was invented by Rudolf kalman at a very convenient time in american history, soon after developing the algorithm Kalman visited his acquaintance Stanley Schmidt at his workplace in NASA and introduced the algorithm to NASA staff, at the time NASA was struggling to apply existing algorithms to the nonlinear problem of trajectory estimation for the Apollo

program the hope was to launch a spacecraft into a trajectory around the moon the challenge lay in being able to create something accurate enough to guide the spacecraft through very narrow corridors of space but also efficient enough to run on an onboard computer in the 1960s at the time computing power was very limited in both the onboard computer and the computer that was used for simulation and testing, another challenge was that during the flight of the spacecraft measurements would be coming in at irregular time intervals, something that the existing algorithms were unable to process, after a lot of work and even some tweaking of the algorithm the kalman filter provided the Apollo mission with the navigational accuracy to successfully enter orbit around the moon.



#### 4.4.4 Applications

Since its success with the Apollo program, the Kalman Filter has become one of the most practical algorithms in the field of controls engineering. Today the Kalman filter is applied in many different disciplines within engineering the Kalman filter is often used to estimate the state of a system when the measurements are noisy. For example, the fluid level in a tank or as we saw earlier the position tracking of a mobile robot. Outside of engineering the kalman filter is very popular in the field of economics, for instance, to estimate the exchange rate of a particular currency or the global domestic product, a measure of a country's economic well being.

Computer vision is another big user of the Kalman filter, for many different applications including feature tracking.

#### 4.4.5 Variations

So far we have been referring to the Kalman filter as one unique entity. However they're actually three common types.

There is the standard Kalman Filter and two variations to it, the Extended Kalman Filter and the Unscented Kalman filter.



- The Kalman filter can be applied to linear systems, ones where the output is proportional to the input.
- The EKF can be applied to non-linear systems which is more applicable in robotics, as real world systems are more often non-linear than linear.
- The UKF is another type of non-linear estimator appropriate for highly non-linear systems where EKF may fail to converge.

#### 4.4.6 Robot Uncertainty

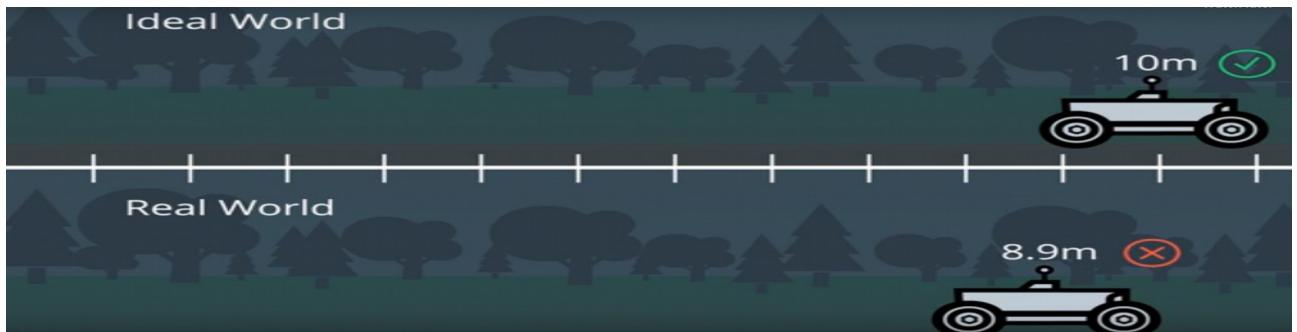
Before we dive into the details of the Kalman filter, it's important to understand a few intricacies of robot operation that will give context to why a Kalman filter works the way it does.

Let's explore two different robot worlds, the Ideal world and the real world



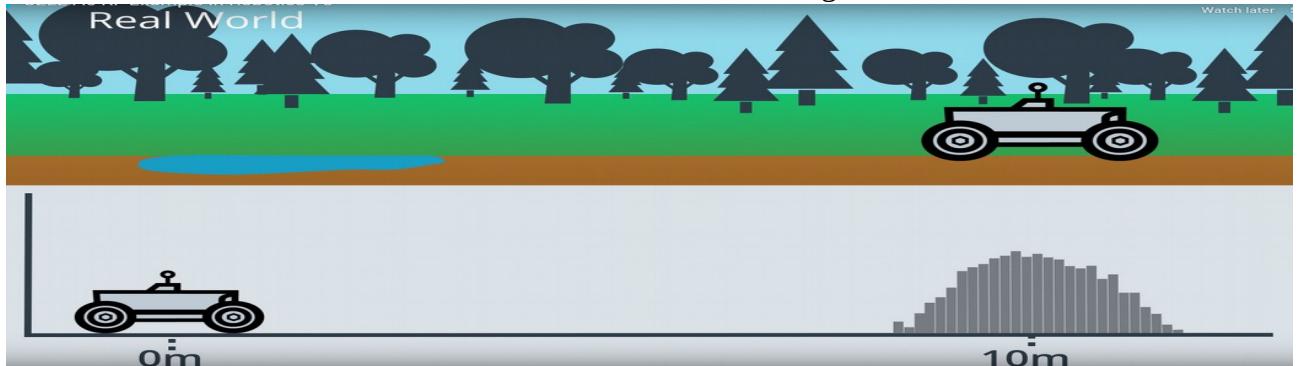
In both worlds, the robots know their starting positions. In the ideal world the robot is instructed to move 10 meters forward, the robot proceeds to do so and stops precisely 10 meters from its starting position. The movement was error-free. The robot in the real world is also asked to move forward 10 meters. **In the real world** however there are few complexities that result in the robot's movement being **imprecise**. The robot may encounter imperfections in the terrain, experience wheel slip, or be adversely affected by other factors in its environment. Upon completion of its movement

the robot may not be at the 10-meter mark precisely, but some distance ahead of or behind of its desired goal.



This error would be different with every movement performed due to the randomness encountered in the environment.

Let's look at this movement on a graph. If we were to record the real world robot moving 100 times forward a total of 100 times, the results would look like the image below.

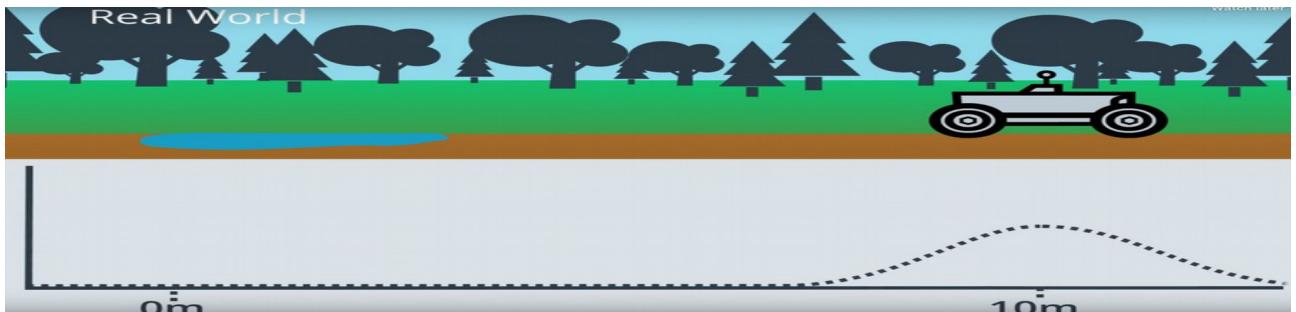


The roboticist with a keen eye may notice that this data **resembles a bell curve, also known as Gaussian**. This graph displays a probability distribution of the robot's final position after multiple iterations of the movement. The X-axis represents the distance traveled by the robot, and the Y-axis represents how often the robot stopped at that distance. This curve shows that if we were to ask a robot to move, it is most likely to stop at the 10-meter mark, but at times, can find itself as far off as the 12-meter mark.

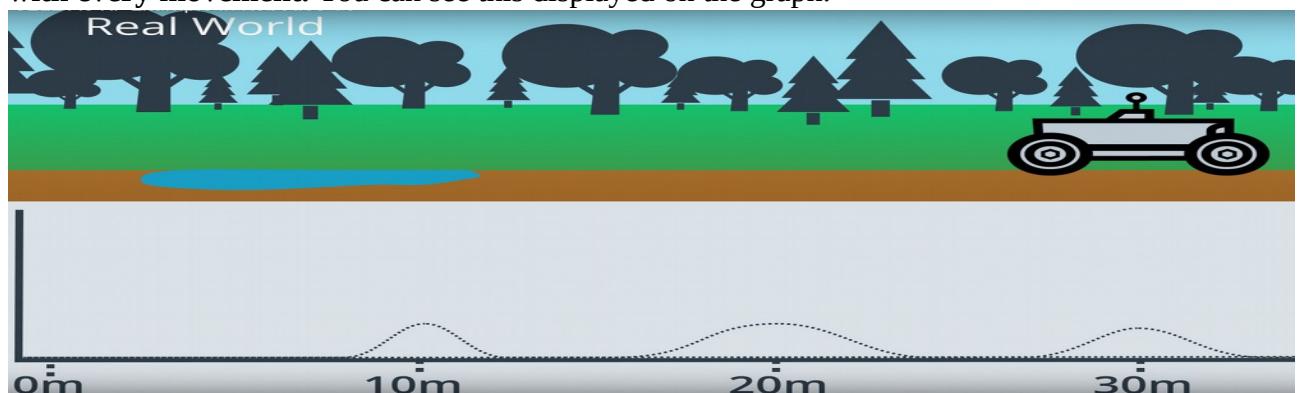
The shape of the Gaussian is specific to the robots and the environment that is operating in. If the robot was driving around a factory floor, there'd be fewer environmental factors to affect it, so its movement would be more precise, and its distribution more narrow.



On the other hand, if the robot is performing rescue missions, it may have any number of environmental factors affect it, such as adverse weather or unexpected movements due to unstable terrain. The result is a much wider Gaussian distribution.



But the distribution of the uncertainty is not the only problem here. **If a robot were to continue to take blind movements one after the other, then its location would become less and less certain with every movement.** You can see this displayed on the graph.



Every movement is uncertain, and these uncertainties stack up over time. With all of these uncertainties in motion, a robot needs a way to sense its own action. **But unfortunately sensory data is often uncertain, too.**

Let's say that we are interested in knowing the speed of a robot, and that we have sensors on-board the robot to measure this. In the **ideal world the sensor will measure the speed of the robot accurately**, and respond to changes in the speed instantaneously. However, in the real world, **sensor measurements will contain some amount of noise**. An expensive encoder may come close to what you would expect in an ideal world, but if you're using hobbyist grade robotic sensors, they will exhibit imperfect measurements.



#### 4.4.7 Kalman Filter Advantage

So now that we know that both movements and sensory measurements are uncertain, how can the Kalman filter help us make better sense of our robot's current state?

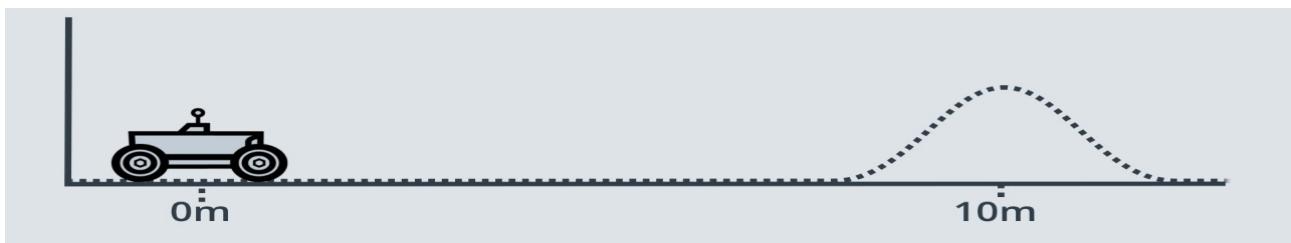
The Kalman filter can very quickly develop a surprisingly accurate estimate of the true value of the variable being measured. For example, your robot's location in a one dimensional 'real' world. Unlike other algorithms that require a lot of data to make an estimate, the Kalman filter is able to do so after just a few sensor measurements. **It does so by using an initial guess and by taking into account the expected uncertainty of a sensor or movement.**

There is another advantageous application of the Kalman filter. Let's say that your robot is using GPS data to identify its location. Today's GPS measurements are only accurate to a few meters. It is possible that by using the GPS alone, you cannot obtain an accurate enough estimate of your robot's location. **However, if you use additional sensors on board the robot, you may be able to combine measurements from all of them to obtain a more accurate estimate. This is called SENSOR FUSION.** Sensor fusion uses the Kalman filter to calculate a more accurate estimate using data from multiple sensors.

Once again, the Kalman filter takes into account the uncertainty of each sensor's measurements. So whether it's making sense of noisy data from one sensor or from multiple, **the Kalman filter is a very useful algorithm to learn!!**

#### 4.4.8 1-D Gaussian

At the basis of the Kalman Filter is the Gaussian distribution, sometimes referred to as a bell curve or normal distribution. Recall the rover example - after executing one motion, the rover's location was represented by a Gaussian. Its exact location was not certain, but the level of uncertainty was bounded. It was unlikely that the rover would be more than a few meters away from its target location, and it would be nearly impossible for it to show up at the 50 meter mark.

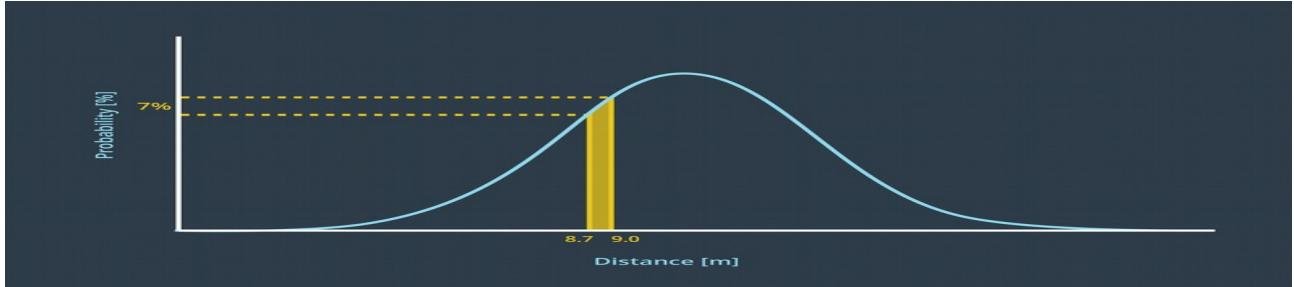


This is the role of a Kalman Filter - after a movement or a measurement update, it outputs a unimodal Gaussian distribution. This is its best guess at the true value of a parameter.

A Gaussian distribution is a probability distribution, which is a continuous function. The probability that a random variable,  $x$ , will take a value between  $x_1$  and  $x_2$  is given by the integral of the function from  $x_1$  to  $x_2$ .

$$p(x_1 < x < x_2) = \int_{x_1}^{x_2} f_x(x) dx$$

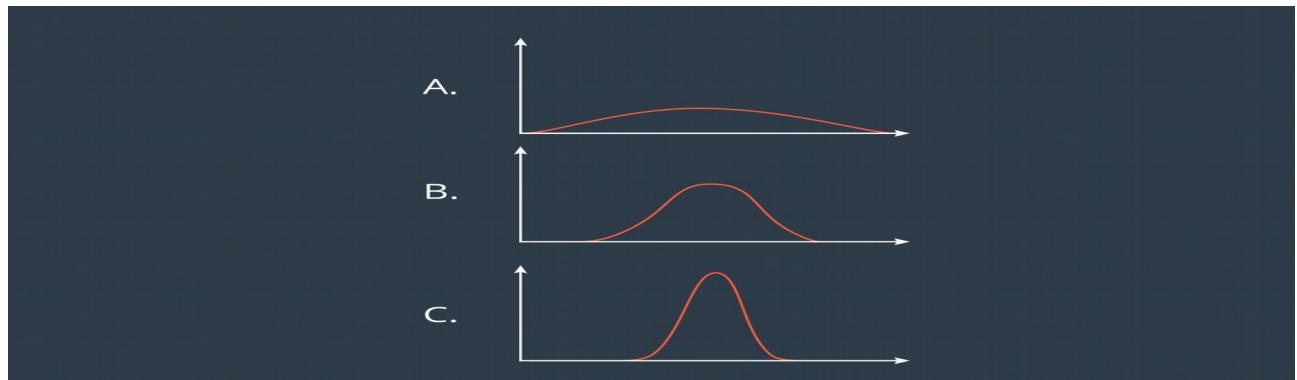
In the image below, the probability of the rover being located between 8.7m and 9m is 7%.



## Mean and Variance

A Gaussian is characterized by two parameters - its mean ( $\mu$ ) and its variance ( $\sigma^2$ ). The mean is the most probable occurrence and lies at the center of the function, and the variance relates to the width of the curve. The term unimodal implies a single peak present in the distribution.

Gaussian distributions are frequently abbreviated as  $N(x: \mu, \sigma^2)$



The most preferred Gaussian distribution to represent the location of a rover is obviously C.

The formula for the Gaussian distribution is printed below. Notice that the formula contains an exponential of a quadratic function. The quadratic compares the value of  $x$  to  $\mu$ , and in the case that  $x=\mu$ , the exponential is equal to 1 ( $e^{0}=1$ ). You'll note here, that the constant in front of the exponential is a necessary normalizing factor.

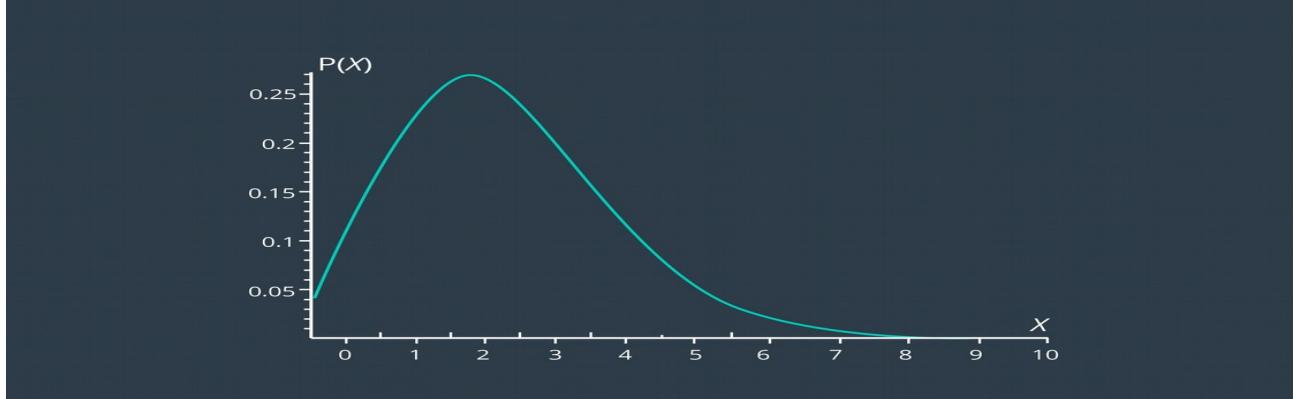
$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Just like with discrete probability, like a coin toss, the probabilities of all the options must sum to one. Therefore, the area underneath the function always sums to one.

$$\int p(x)dx = 1$$

The Gaussian distribution represents **the Predicted Motion, Sensor Measurement and the Estimated State of the Robot**. Furthermore it treats all noise as a unimodal Gaussian. In reality, that's not the case. However the algorithm is optimal if the noise is Gaussian. The term optimal expresses that the algorithm minimizes the mean square error of the estimated parameters.

For example a state with a probability distribution like the image below can't be solved using a Kalman Filter.



#### 4.4.9 Designing 1-D Kalman Filters

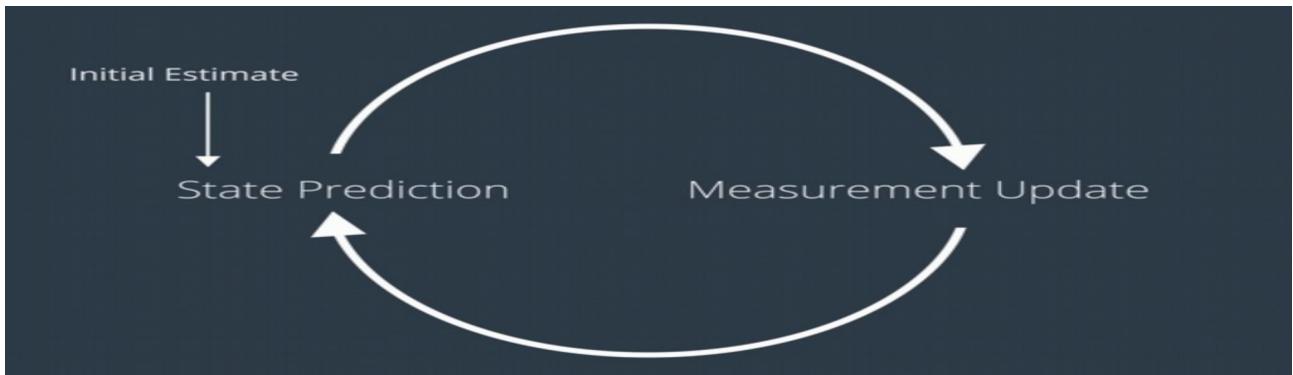
First some naming conventions for the Variables

- $X_t$  : state
- $Z_t$  : measurement
- $U_t$  : control action

Since a robot is unable to sense the world around it with complete certainty, it holds an internal belief, which is its best guess at the state of the environment, including itself.

For example, **a robot constraint to a plane can be described with three state variables**, two coordinates X and Y to identify it's position and one angle yaw to identify its orientation. These are the states variables denoted by  $X_t$  they may change over time so we denote the subscript t.

If you recall the two steps involved in the Kalman filter algorithm, they are a measurement update and a state prediction.



**Measurement update :** We know that a robot can perceive its environment using sensors which produces a measurement. This is denoted with the letter  $Z_t$  which represents the measurement obtained at time  $t$ . Next our **Control Actions** such as movements can change the state of our environment. Control actions are denoted by  $U_t$  represents the change of state that occurred between time  $t - 1$  and  $t$ .

Perfect, now we need to start the circle somewhere and this is usually with an initial estimate of the state. **The estimate does not have to be accurate.** It can be an awful guess and the Kalman filter will still manage to give us good results very quickly. Then we iterate between the **measurement update where we gain knowledge about our environment and the state prediction which causes us to lose knowledge**, due to the uncertainty of robot motion.

Next we will explore the Measurement Update and State Prediction and we will extend our knowledge to a **Multidimensional Kalman Filter and code one in C++ (see Appendix!)**.

Lastly we will see how the Extended Kalman Filter allows us to solve a wide range of problems and help us with real-world robot localization.

## 1D Kalman Filters

- Measurement Update
- State Prediction
- Multidimensional Kalman Filter
- Code in C ++
- **Extended Kalman Filter**

### 4.4.10 Measurement Update

Let's begin our Kalman filter implementation with the measurement update.

Let's take a look at an example back in our one-dimensional robot world.

It seems that the robot has been roaming around however I have an inkling that the robot's current position is near the 20-meter mark, **but we are not incredibly certain so the prior belief Gaussian has a rather wide probability distribution.**

Next the robot takes it's first sensory measurement providing us with data to work with. **The measurement data Z, is more certain, so it appears as a narrow Gaussian with a mean of 30,** hey our intuition was close.

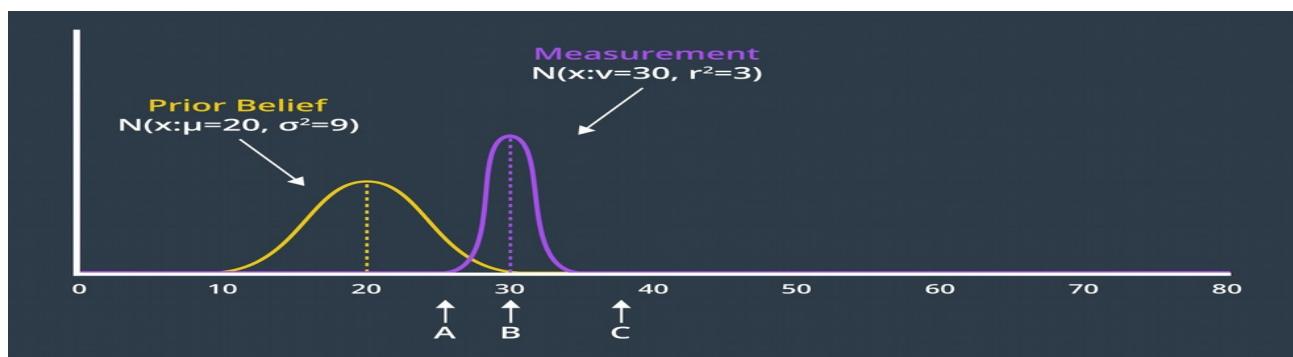
Given our prior belief about the robot's state and the measurement that it collected **where do you think the robot's new belief would be?**

$\mu$ : Mean of the prior belief

$\sigma^2$ : Variance of the prior belief

v: Mean of the measurement

r<sup>2</sup>: Variance of the measurement



The new Belief would have a mean somewhere in between the two Gaussians, since it is combining information from both of them, but where?

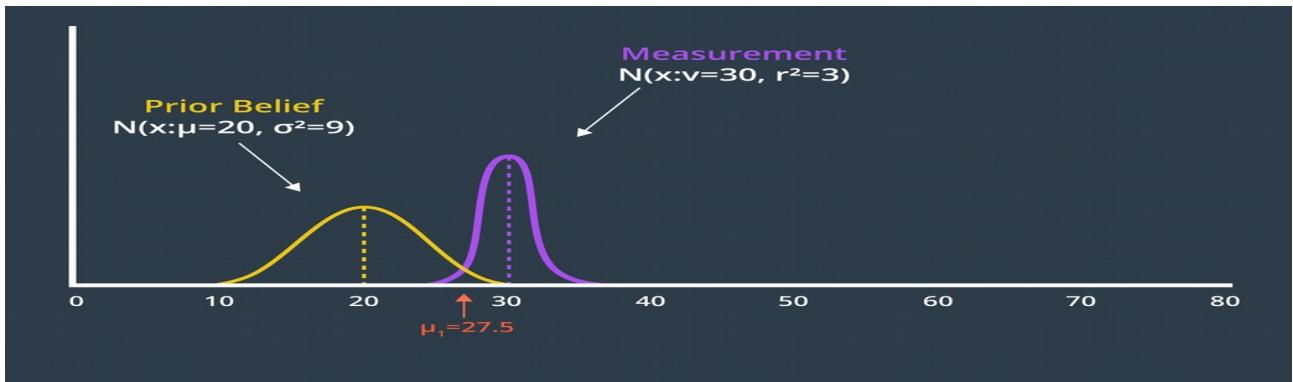
**Well since the measurement update is more certain the new mean will lie closer to the measurement update. Correct answer A.**

The new mean is a weighted sum of the prior belief and measurement means. With uncertainty, a larger number represents a more uncertain probability distribution. However, the new mean should be biased towards the measurement update, which has a smaller standard deviation than the prior. How do we accomplish this?

$$\mu' = \frac{r^2\mu + \sigma^2v}{r^2 + \sigma^2}$$

The answer is - the uncertainty of the prior is multiplied by the mean of the measurement, to give it more weight, and similarly the uncertainty of the measurement is multiplied with the mean of the

prior. Applying this formula to our example generates a new mean of 27.5, which we can label on our graph below.



## Variance Calculation

Next, we need to determine the variance of the new state estimate.

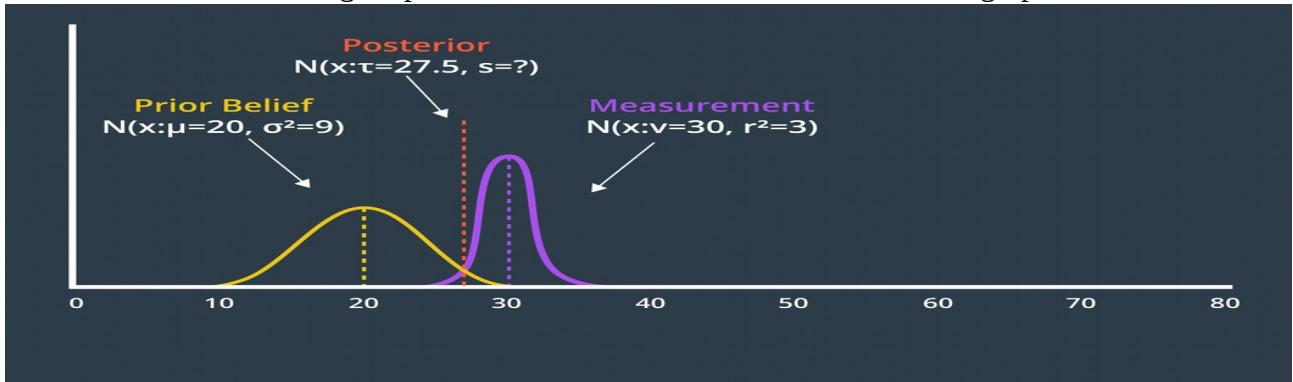
What do you think? Would the variance of the new state estimate be less confident than our prior, in between our prior and measurement, or more confident than our measurement?

**The answer is that it would be More confident than our measurement.**

The two Gaussians provide us with more information together than either Gaussian offered alone.

As a result, our new state estimate is more confident than our prior belief and our measurement.

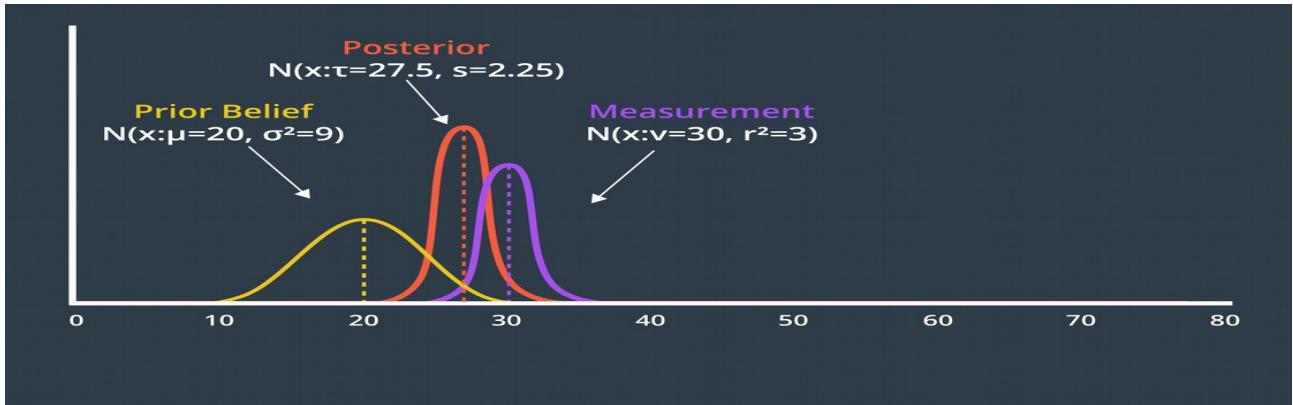
This means that it has a higher peak and is narrower. You can see this in the graph below.



The formula for the new variance is presented below.

$$\sigma^{2'} = \frac{1}{\frac{1}{r^2} + \frac{1}{\sigma^2}}$$

Entering the variances from our example into this formula produces a new variance of 2.25. The new state estimate, often called the posterior, is drawn below.



$\mu$ : Mean of the prior belief

$\sigma^2$ : Variance of the prior belief

$v$ : Mean of the measurement

$r^2$ : Variance of the measurement

$\tau$ : Mean of the posterior

$s^2$ : Variance of the posterior

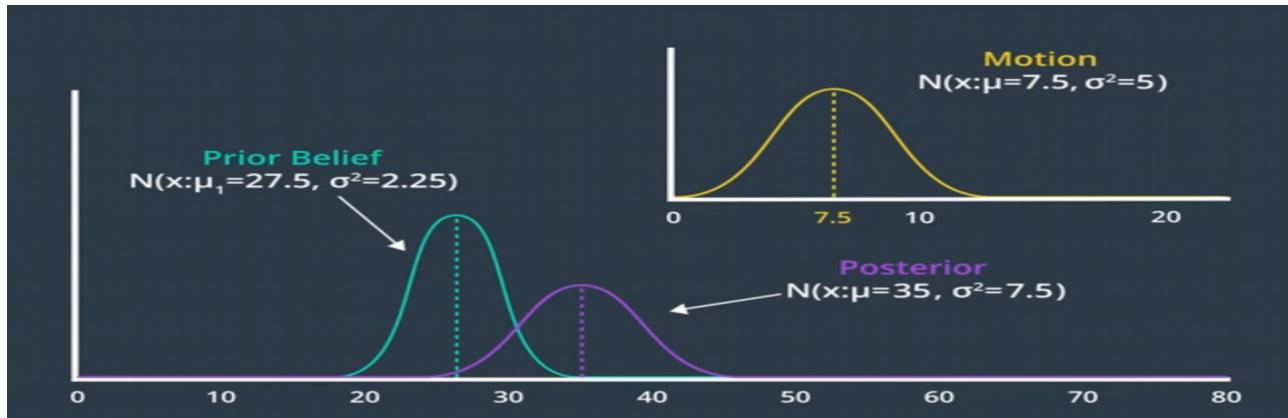
#### 4.4.11 State Prediction

By implementing the measurement update, we have now completed half of the Kalman filter's iterative cycle. Let's move on to State Prediction Next.

Recall that State Prediction is the estimation that takes place after an inevitably uncertain motion. Since the measurement update and state prediction are an iterative cycle, it makes sense to continue where we left off. After taking into account the measurement the posterior distribution was a Gaussian with a mean of 27.5 and a variance of 2.25. However, since we've moved onto the state predictions step in the Kalman filter cycle, this Gaussian is now referred to as the prior belief. This is the robot's best estimate of its current location.

Next the robot executes a command, Move forward 7.5 meters. The result of this motion is a Gaussian distribution centered around 7.5 meters with a variance of five meters.

Calculating the new estimate is as easy as adding the mean of motion to the mean of the prior, and similarly, adding the two variances together to produce the posterior Gaussian.

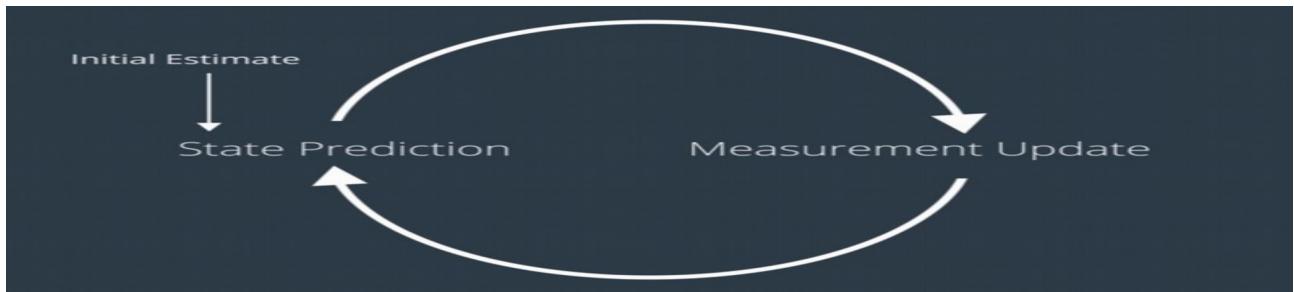


#### State Prediction Formulas

$$\text{Posterior Mean} \quad \mu' = \mu_1 + \mu_2$$

$$\text{Posterior Variance} \quad \sigma'^2 = \sigma_1^2 + \sigma_2^2$$

#### 4.4.12 1-D Kalman Filter



All right. So we've learned a few things. The measurement update is a weighted sum of the prior belief and the measurement and the State Prediction is the addition of the prior belief's mean and variance to the motion's mean and variance. We have done these steps (see C++ code in appendix) now we need to add some support code that will call them one after the other as long as measurement data is available and the robot has motion to implement.

State Prediction	Measurement Update
<pre>double motion[5]={ 1,1,2,1,1 }; double motion_sig=2;</pre>	<pre>double measurements[5]={ 5,6,7,9,10 }; double measurement_sig=4;</pre>
Initial Estimate	
<pre>double mu=35;</pre>	

The index of every value in each list represents its timestamp. The first value in the measurement array is the measurement at time equals zero. The second at time equals 1 and so forth. Here we have 5 values of each, so we will be able to go through five iterations of the Kalman Filter Cycle. At each iteration, the Kalman filter will try to estimate the state of the system. After five iterations, we will see if the Kalman filter converges on the robot's true value. We will also use two variables to represent the variance of the measurements and motions. Lastly we require an initial estimate of the robot's position with a initial variance of 4 (not shown in picture.) but this value is not as critical to a successful outcome as you may initially think.

#### 4.4.13 Multivariate Gausians

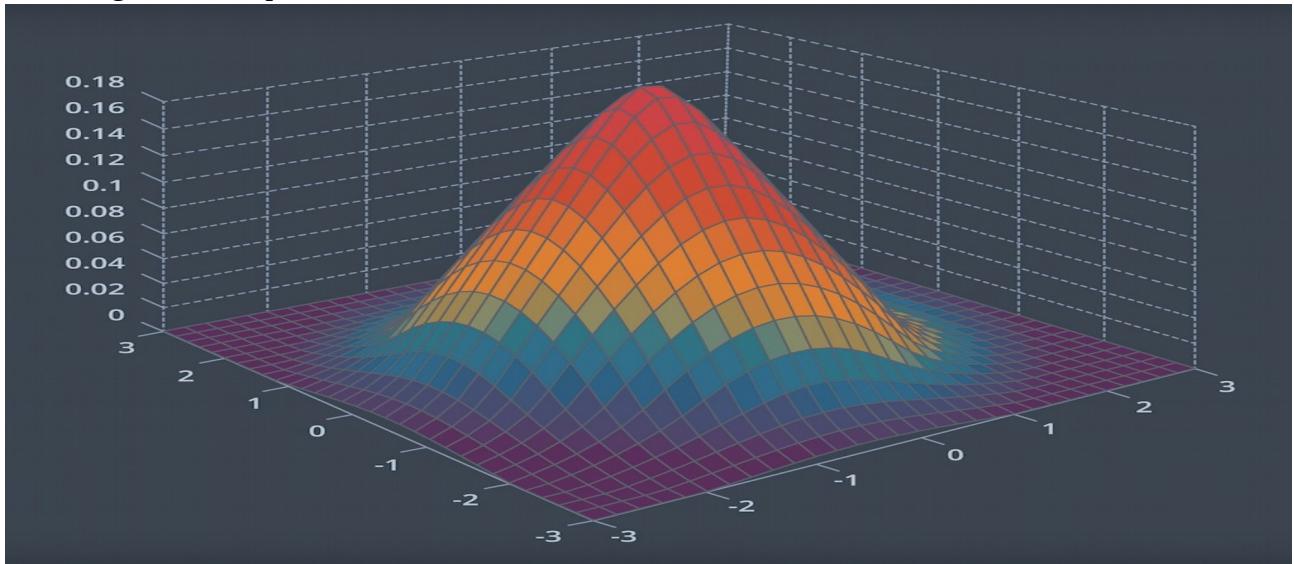
Most robots that we would be interested in modeling are moving in more than one dimension. For instance, a robot on a plane would have an x & y position.

The simple approach to take, would be to have a 1-dimensional Gaussian represent each dimension - one for the x-axis and one for the y-axis.

Do you see any problems with this?

**Yes, There may be correlations between dimensions that we would not be able to model by using independent 1-dimensional Gaussians.**

The image below depicts a two-dimensional Gaussian distribution.



Now it's time to consider systems with higher dimensions, after all most of the real life processes that we'd like to model are multi-dimensional.

A 2-D Gaussian can be represented as so where the contour lines show variations in height.

## Formulas for the Multivariate Gaussian

### Mean

The mean is now a vector,

$$\mu = \begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix}$$

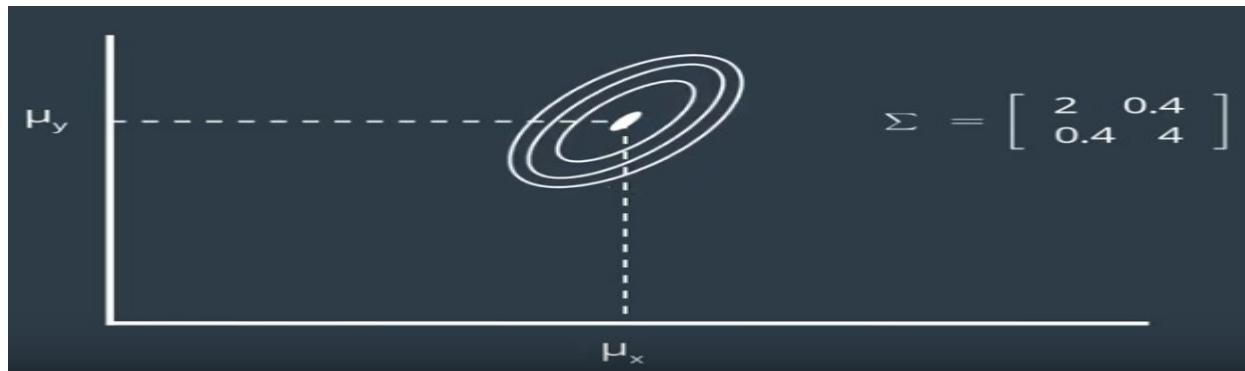
### Covariance

And the multidimensional equivalent of variance is a covariance matrix,

$$\Sigma = \begin{bmatrix} \sigma_x^2 & \sigma_x\sigma_y \\ \sigma_y\sigma_x & \sigma_y^2 \end{bmatrix}$$

Where  $\sigma_x^2$  and  $\sigma_y^2$  represent the variances, while  $\sigma_y\sigma_x$  and  $\sigma_x\sigma_y$  are correlation terms. These terms are non-zero if there is a correlation between the variance in one dimension and the variance in another. When that is the case, the Gaussian function looks 'skewed' when looked at from above.

If we were to evaluate this mathematically, the eigenvalues and eigenvectors of the covariance matrix describe the amount and direction of uncertainty.



### Multivariate Gaussian

Below is the formula for the multivariate Gaussian. Note that  $x$  and  $\mu$  are vectors, and  $\Sigma$  is a matrix.

$$p(x) = \frac{1}{(2\pi)^{\frac{D}{2}} |\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)}$$

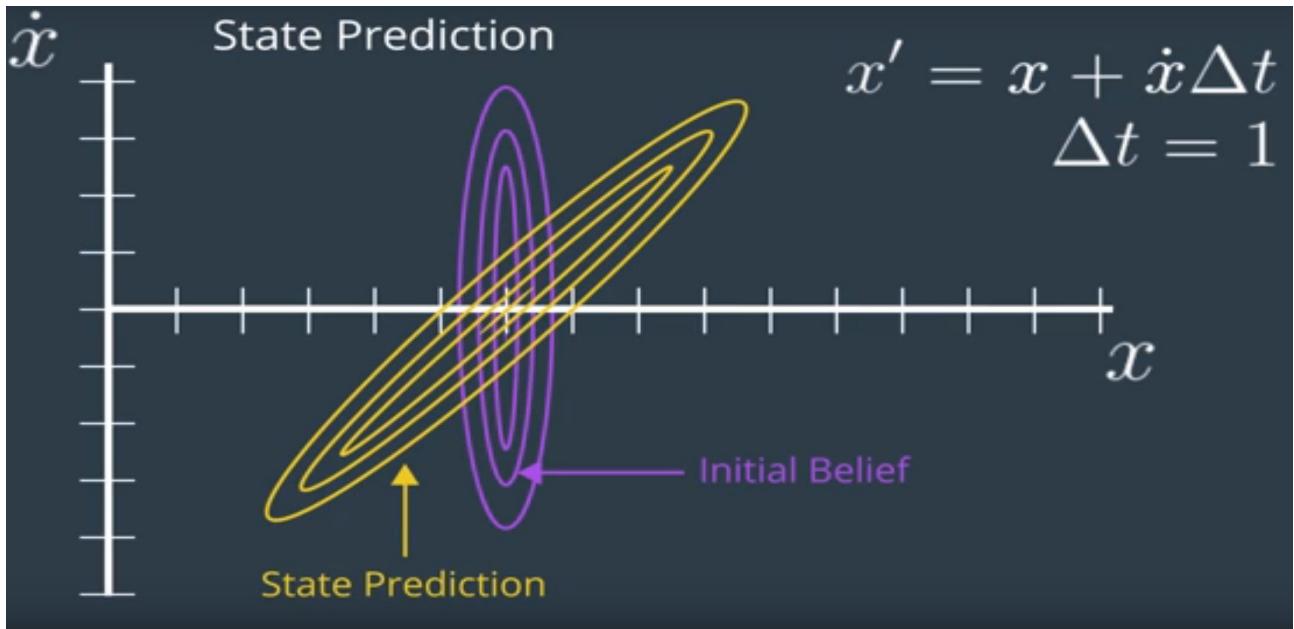
If  $D=1$ , the formula simplifies to the formula for the one-dimensional Gaussian that you have seen before.

### 4.4.14 Intro to Multidimensional KF

In  $N$ -dimensional systems, the state is a vector with  $N$  state variables. A more interesting case is if these states are the position and the velocity of a robot. This leads to another important matter, **when working in 1-Dimensional your state has to be observable**, meaning that it had to be something that can be measured directly. **In multi-dimensional states there may exist hidden state variables**, once that you cannot measure with the sensors available however, you may be able to infer their value from other states and measurements. e.g the velocity is a hidden state.

$$\begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

However the robot's position and velocity over time are linked through a very simple formula. Let's look at this on a graph. If initially, the position of the robot is known, but its velocity is not, the estimate of the robot's state would look like so, a Gaussian that is very narrow in the X dimension representing confidence about the robot's location, and very wide in the X dot dimension, since the robot velocity is completely unknown.



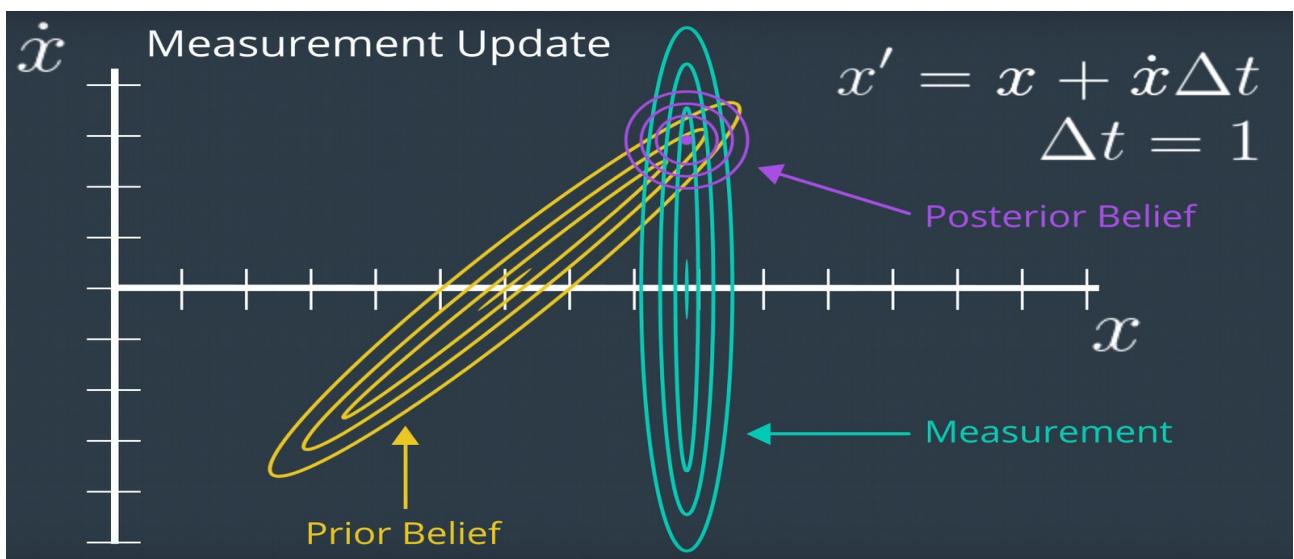
**Next, a state prediction can be calculated, This is where the Kalman filter starts to get exciting.** Knowing the relationship between the hidden variable and observable variable is key.

Let's assume that one iteration of the Kalman filter takes 1 second, Now using the formula we can calculate the posterior state for each possible velocity. For instance, if the velocity is 0 the posterior state would be identical to the prior and if the velocity is one, the posterior would be on the y axis at 1 and so forth, from this we can draw a posterior Gaussian represented by the yellow lines.

This doesn't tell us anything about the velocity, it just graphs the correlation between the velocity and the location of the robot.

However what do we have next? **A measurement update**

Let's see if it will be able to provide us with sufficient information to identify the velocity of the robot. The initial belief was useful to calculate the state prediction but has no additional value and the result of the state prediction can be called a Prior Belief for our measurement update.



If the new measurement suggest a location of X equals 50, Now if we apply the measurement update to our prior, **we will have a very small posterior centered around X equals 50 and X dot equals 50**. This might look like magic, but it's the exact same measurement update that we applied before, implemented in two dimensions, giving us a good estimate of the hidden velocity state of our robot!!

#### 4.4.15 Design of Multidimensional KF

From this point forward we will transition to using linear algebra, as it allows us to easily work with multi-dimensional problems. To begin with, let's write the state prediction in linear algebra form.

##### State Transition

The formula below is the state transition function that advances the state from time  $_t$  to time  $t + 1$ . It is just the relationship between the robot's position,  $x$ , and velocity,  $\dot{x}$ . Here, we will assume that the robot's velocity is not changing.

$$x' = x + \Delta t \dot{x}$$

$$\dot{x}' = \dot{x}$$

We can express the same relationship in matrix form, as seen below. On the left, is the posterior state (denoted with the prime symbol,  $'$ ), and on the right are the state transition function and the prior state. This equation shows how the state changes over the time period,  $\Delta t$ . Note that we are only working with the means here; the covariance matrix will appear later.

$$\begin{bmatrix} x \\ \dot{x} \end{bmatrix}' = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

The State Transition Function is denoted  $F$ , and the formula can be written as so,

$$x' = Fx$$

In reality, the equation should also account for process noise, as its own term in the equation. However, process noise is a Gaussian with a mean of 0, so the update equation for the mean need not include it.

$$x' = Fx + noise$$

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

Now, what happens to the covariance? How does it change in this process?

**Sidenote:** While it is common to use  $\Sigma$  to represent the covariance of a Gaussian distribution in mathematics, it is more common to use the letter P to represent the state covariance in localization.

If you multiply the state,  $x$ , by F, then the covariance will be affected by the square of F. In matrix form, this will look like so:

$$P' = FPF^T$$

However, your intuition may suggest that it should be affected by more than just the state transition function. For instance, additional uncertainty may arise from the prediction itself. If so, you're correct!

To calculate the posterior covariance, the prior covariance is multiplied by the state transition function squared, and Q added as an increase of uncertainty due to process noise. Q can account for a robot slowing down unexpectedly, or being drawn off course by an external influence.

$$P' = FPF^T + Q$$

Now we've updated the mean and the covariance as part of the state prediction.

## Measurement Update

Next, we move onto the measurement update step. If we return to our original example, where we were tracking the position and velocity of a robot in the x-dimension, the robot was taking measurements of the location only (the velocity is a hidden state variable). Therefore the measurement function is very simple - a matrix containing a one and a zero. This matrix demonstrates how to map the state to the observation, z.

$$z = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

This matrix, called the Measurement Function, is denoted H.

For the measurement update step, there are a few formulas. First, we calculate the measurement residual, y. The measurement residual is the difference between the measurement and the expected measurement based on the prediction (ie. we are comparing where the measurement *tells us* we are vs. where we *think* we are). The measurement residual will be used later on in a formula.

$$y = z - Hx'$$

Next, it's time to consider the measurement noise, denoted R. This formula maps the state prediction covariance into the measurement space and adds the measurement noise. The result, S, will be used in a subsequent equation to calculate the Kalman Gain.

$$S = HP'H^T + R$$

These equations need not be memorized, instead they can be referred to in text or implemented in code for use and reuse.

## Kalman Gain

Next, we calculate the Kalman Gain, K. As you will see in the next equation, the Kalman Gain determines how much weight should be placed on the state prediction, and how much on the measurement update. It is an averaging factor that changes depending on the uncertainty of the state prediction and measurement update.

$$K = P'H^T S^{-1}$$

$$x = x' + Ky$$

These equations may look complicated and intimidating, but they do nothing more than calculate an average factor.

The last step in the Kalman Filter is to update the new state's covariance using the Kalman Gain.

$$P = (I - KH)P'$$

## Kalman Filter Equations

These are the equations that implement the Kalman Filter in multiple dimensions.

State Prediction:

$$x' = Fx$$

$$P' = FPF^T + Q$$

Measurement Update:

$$y = z - Hx'$$

$$S = HP'H^T + R$$

Calculation of Kalman Gain:

$$K = P' H^T S^{-1}$$

Calculation of Posterior State and Covariance:

$$x = x' + Ky$$

$$P = (I - KH)P'$$

The Kalman Filter can successfully recover from inaccurate initial estimates, **but it is very important to estimate the noise parameters, Q and R, as accurately as possible** - as they are used to determine which of the estimate or the measurement to believe more.

#### 4.4.16 Introduction to EKF

If you recall the assumptions under which the Kalman filter operates are,

##### Kalman Assumptions

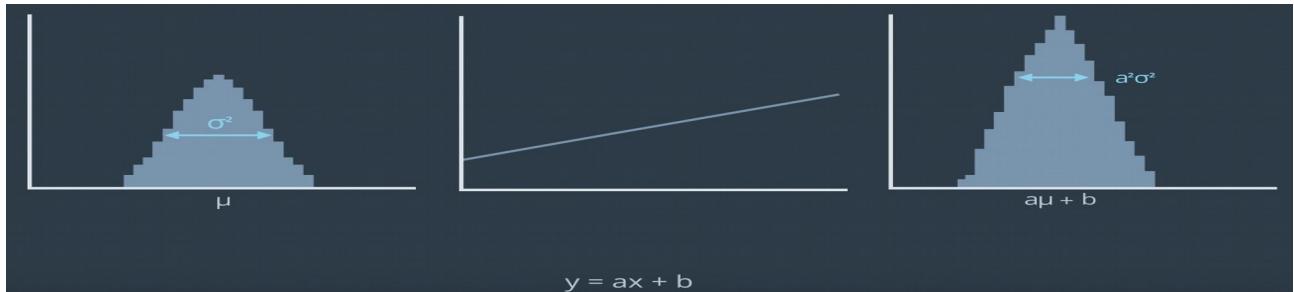
- Motion and measurement models are linear,
- State space can be represented by a unimodal Gaussian distribution.

Well, it turns out that these assumptions are very limiting, and would only suffice for very primitive robots. Most mobile robots will execute non-linear motions. For example, moving in a circle or following a curve.

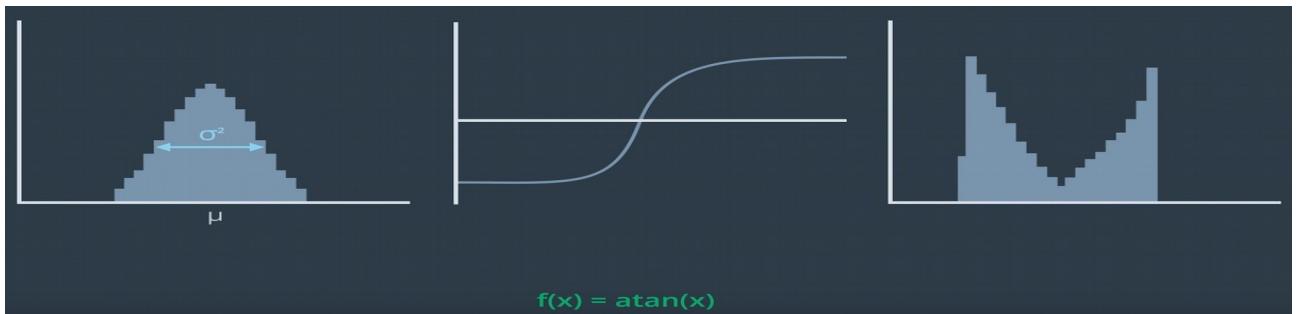
So what can we do if our Kalman filter can't be applied to most robotics problems?

Let's take a look at why we can't apply the Kalman filter in more detail!

With a linear transformation we have

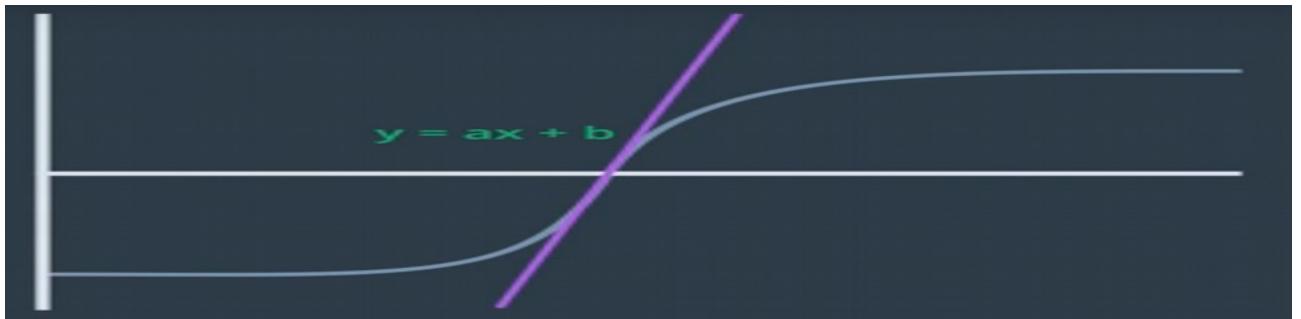


While with a non linear transformation like  $\text{atan}(x)$



Now the function  $f(x)$  is clearly non-linear and the resulting distribution is no longer a Gaussian as can be identified from its shape. In fact the distribution cannot be computed in closed form, meaning in a finite number of operations. Instead to model this distribution, many thousands of samples must be collected according to the prior distribution and pass through the function  $f(x)$  to create this posterior distribution, which is much more computationally expensive losing the responsiveness that the Kalman filter is known for.

So what can we do? If we look very closely the function  $f(x)$  may be approximated by a linear function.



This will allow us to use the Kalman filter on this nonlinear problem. The linear estimate is only good for a small section of the function, but if it is centered on the best estimate, the mean and updated with every step, it turns out that it can produce sufficiently accurate results.

The mean can continue to be updated with a non linear function, but the covariance must be updated by a linearization of the function  $f(x)$



To calculate the local linear approximation, a taylor series can be of help.

$$T(x) = f(a) + \frac{f'(a)}{1!} (x - a) + \frac{f''(a)}{2!} (x - a)^2 + \frac{f'''(a)}{3!} (x - a)^3 + \dots$$

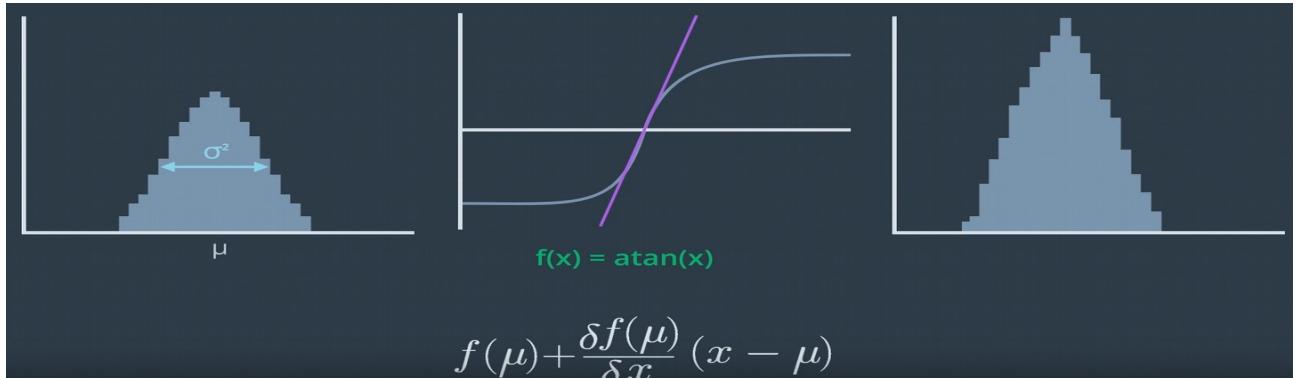
However, an infinite number of terms is not needed.

A linear approximation can be obtained using just the first two terms of a taylor series.

$$f(a) + \frac{f'(a)}{1!} (x - a)$$

This linear approximation centered around the mean, will be used to update the covariance matrix of the prior state Gaussian.

With this approximation the transformation can be applied to the prior, the result is exactly what we wanted a Gaussian



It will inevitably have an error, we can see that the posterior Gaussian distribution is quite different from the distribution that we saw before, **However the tradeoff is in the simplicity and speed of the calculation.**

#### 4.4.17 EKF

### Multi-dimensional Extended Kalman Filter

Now you've seen the fundamentals behind the Extended Kalman Filter. The mechanics are not too different from the Kalman Filter, with the exception of needing to linearize a nonlinear motion or measurement function to be able to update the variance.

You've seen how this can be done for a state prediction or measurement function that is of one-dimension, but now it's time to explore how to linearize functions with multiple dimensions. To do this, we will be using multi-dimensional Taylor series.

### Linearization in Multiple Dimensions

The equation for a multidimensional Taylor Series is presented below.

$$T(x) = f(a) + (x - a)^T Df(a) + \frac{1}{2!} (x - a)^T D^2 f(a) (x - a) + \dots$$

You will see that it is very similar to the 1-dimensional Taylor Series. As before, to calculate a linear approximation, we only need the first two terms.

$$T(x) = f(a) + (x - a)^T Df(a)$$

You may notice a new term,  $Df(a)$ . This is the Jacobian matrix, and it holds the partial derivative terms for the multi-dimensional equation.

$$Df(a) = \frac{\delta f(a)}{\delta x}$$

In its expanded form, the Jacobian is a matrix of partial derivatives. It tells us how each of the components of  $f$  changes as we change each of the components of the state vector.

$$Df(a) = \begin{bmatrix} \frac{\delta f_1}{\delta x_1} & \frac{\delta f_1}{\delta x_2} & \cdots & \frac{\delta f_1}{\delta x_n} \\ \frac{\delta f_2}{\delta x_1} & \frac{\delta f_2}{\delta x_2} & \cdots & \frac{\delta f_2}{\delta x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\delta f_m}{\delta x_1} & \frac{\delta f_m}{\delta x_2} & \cdots & \frac{\delta f_m}{\delta x_n} \end{bmatrix}$$

The rows correspond to the dimensions of the function,  $f$ , and the columns relate to the dimensions (state variables) of  $x$ . The first element of the matrix is the first dimension of the function derived with respect to the first dimension of  $x$ .

The Jacobian is a generalization of the 1-dimensional case. In a 1-dimensional case, the Jacobian would have  $df/dx$  as its only term.

## Example Application

This will make more sense in context, so let's look at a specific example. Let's say that we are tracking the  $x$ - $y$  coordinate of an object. This is to say that our state is a vector  $x$ , with state variables  $x$  and  $y$ .

$$x = \begin{bmatrix} x \\ y \end{bmatrix}$$

However, our sensor does not allow us to measure the  $x$  and  $y$  coordinates of the object directly. Instead, our sensor measures the distance from the robot to the object,  $r$ , as well as the angle between  $r$  and the  $x$ -axis,  $\theta$ .

$$z = \begin{bmatrix} r \\ \theta \end{bmatrix}$$

It is important to notice that our state is using a Cartesian representation of the world, while the measurements are in a polar representation. How will this affect our measurement function?

Our measurement function maps the state to the observation, as so,

$$\begin{bmatrix} x \\ y \end{bmatrix} \xrightarrow{\text{meas. function}} \begin{bmatrix} r \\ \theta \end{bmatrix}$$

Thus, our measurement function must map from Cartesian to polar coordinates. But there is no matrix,  $H$ , that will successfully make this conversion, as the relationship between Cartesian and polar coordinates is nonlinear.

$$r = \sqrt{x^2 + y^2}$$

$$\theta = \tan^{-1}\left(\frac{y}{x}\right)$$

For this reason, instead of using the measurement residual equation  $y=z-Hx'$  that you had seen before, the mapping must be made with a dedicated function,  $h(x')$ .

$$h(x') = \begin{bmatrix} \sqrt{x^2 + y^2} \\ \tan^{-1}\left(\frac{y}{x}\right) \end{bmatrix}$$

Then the measurement residual equation becomes  $y=z-h(x')$ .

Our measurement covariance matrix cannot be updated the same way, as it would turn into a non-Gaussian distribution (as seen in the previous video). Let's calculate a linearization,  $H$ , and use it instead. The Taylor series for the function  $h(x)$ , centered about the mean  $\mu$ , is defined below.

$$h(x) \simeq h(\mu) + (x - \mu)^T Df(\mu)$$

The Jacobian,  $Df(\mu)$ , is defined below. But let's call it  $H$  since it's the linearization of our measurement function,  $h(x)$ .

$$H = \begin{bmatrix} \frac{\partial r}{\partial x} & \frac{\partial r}{\partial y} \\ \frac{\partial \theta}{\partial x} & \frac{\partial \theta}{\partial y} \end{bmatrix}$$

If you were to compute each of those partial derivatives, the matrix would reduce to the following,

$$H = \begin{bmatrix} \frac{x}{\sqrt{x^2+y^2}} & \frac{y}{\sqrt{x^2+y^2}} \\ -\frac{y}{x^2+y^2} & \frac{x}{x^2+y^2} \end{bmatrix}$$

It's this matrix,  $H$ , that can then be used to update the state's covariance.

To summarize the flow of calculations for the Extended Kalman Filter, it's worth revisiting the equations to see what has changed and what has remained the same.

## Extended Kalman Filter Equations

These are the equations that implement the Extended Kalman Filter - you'll notice that most of them remain the same, with a few changes highlighted in red.

State Prediction:

$$\cancel{x' = Fx} \rightarrow x' = f(x)$$

$$P' = \cancel{FPF^T} + Q$$

Measurement Update:

$$\cancel{y = z - Hx'} \rightarrow y = z - h(x')$$

$$S = \cancel{HP'H^T} + R$$

Calculation of Kalman Gain:

$$K = P' \cancel{H^T} S^{-1}$$

Calculation of Posterior State and Covariance:

$$x = x' + Ky$$

$$P = (I - K \cancel{H})P'$$

Highlighted in blue are the Jacobians that replaced the measurement and state transition functions.

The Extended Kalman Filter requires us to calculate the Jacobian of a nonlinear function as part of every single iteration, since the mean (which is the point that we linearize about) is updated.

## Summary

Phew, that got complicated quickly! Here are the key take-aways about Extended Kalman Filters:

- The Kalman Filter cannot be used when the measurement and/or state transition functions are nonlinear, since this would result in a non-Gaussian distribution.
- Instead, we take a local linear approximation and use this approximation to update the covariance of the estimate. The linear approximation is made using the first terms of the Taylor Series, which includes the first derivative of the function.
- In the multi-dimensional case, taking the first derivative isn't as easy as there are multiple state variables and multiple dimensions. Here we employ a Jacobian, which is a matrix of

partial derivatives, containing the partial derivative of each dimension with respect to each state variable.

While it's important to understand the underlying math to employ the Kalman Filter, don't feel the need to memorize these equations. Chances are, whatever software package or programming language you're working with will have libraries that allow you to apply the Kalman Filter, or at the very least perform linear algebra calculations (such as matrix multiplication and calculating the Jacobian).

## 4.4.18 Recap

We have gotten this far!! This content can be a challenge to learn, we have implemented two kalman filters one for a single dimension, and one for a multi-dimensional problem. We then learned how to apply the extended Kalman filter to non-linear problems, **and we are now ready to tackle the challenge of implementing the EKF in ROS (see Appendix for the results)**

# 4.5 Monte Carlo Localization

## 4.5.1 Introduction

Now, it's time to cover another famous localization algorithm.

We will learn the Monte Carlo or MCL, we'll discuss the advantages of MCL over the EKF in localizing robots.

Then we will learn the role of Particle Filters in the MCL algorithm.

Moving on we will learn the Bayes Filter algorithm and we will evaluate the information state or belief.

After that we will learn the Monte Carlo Localization Pseudo code.

Lastly we will see how the MCL can be applied to a robot to estimate its pose.

## Monte Carlo Localization

- o MCL Concept
- o MCL vs. EKF
- o Particle Filters
- o Bayes Filtering
- o MCL Pseudo Code
- o MCL in Action

**Check the paper of Robust Monte Carlo Localization for Mobile Robots [paper](#) by Sebastian Thrun.**

### 4.5.2 What's MCL?

As a roboticist, you will certainly be interested in mapping your home, office, or backyard with the help of your robot. Then, you'll want to operate your robot inside the mapped environment by keeping track of its position and orientation. To do so, we have a wide choice of localization algorithms we can implement ranging from EKF, to Marco, to Monte Carlo, and finally, Grid.

The Monte Carlo Localization algorithm, or MCL, is the most popular localization algorithm in robotics. So, you might want to choose MCL and deploy it to your robot to keep track of its pose. Now, your robot will be navigating inside its known map and collecting sensory information using **range finder sensors**. MCL will use these sensor measurements to keep track of your robot pose. Many scientists refer to MCL as particle filter localization algorithm, since it uses particles to localize your robot. In robotics, you can think of a particle as a virtual element that resembles the robot. Each particle has a position and orientation and represents a guess of where your robot might be located. These particles are re-sampled each time your robot moves and senses its environment. **Keep in mind that MCL is limited to local and global localization problem only. So, you lose sight of your robot if someone hacks into it, meaning it can not solve the Kidnapped Robot localization problem.**

### 4.5.3 Power of MCL?

With the EKF localization algorithm, you can estimate the pose of almost any robot with accurate onboard sensors. So, you might be wondering why it's necessary to master another localization algorithm. Well, MCL presents many advantages over EKF. **First, MCL is easy to program compared to EKF (we will implement the MCL algorithm in C++ on the appendix).**

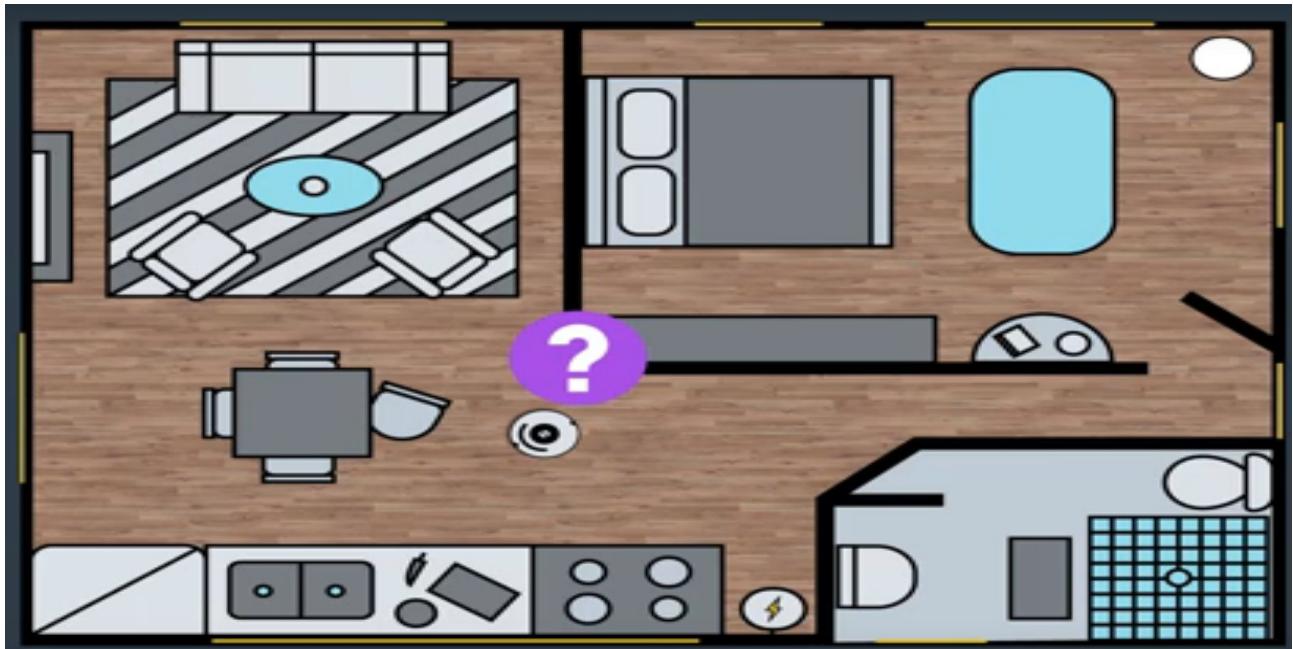
Second, MCL represents **non-Gaussian distributions** and can approximate any other practical important distribution, this means that MCL is unrestricted by a linear Gaussian state based assumption as is the case for EKF. This allows MCL to model a much greater variety of environments especially since you can't always model the real world with Gaussian distributions. Third, in MCL, you can **control the computational memory and resolution** of your solution by changing the number of particles distributed uniformly and randomly throughout the map.

For a full comparison of EKF and MCL check the table below.

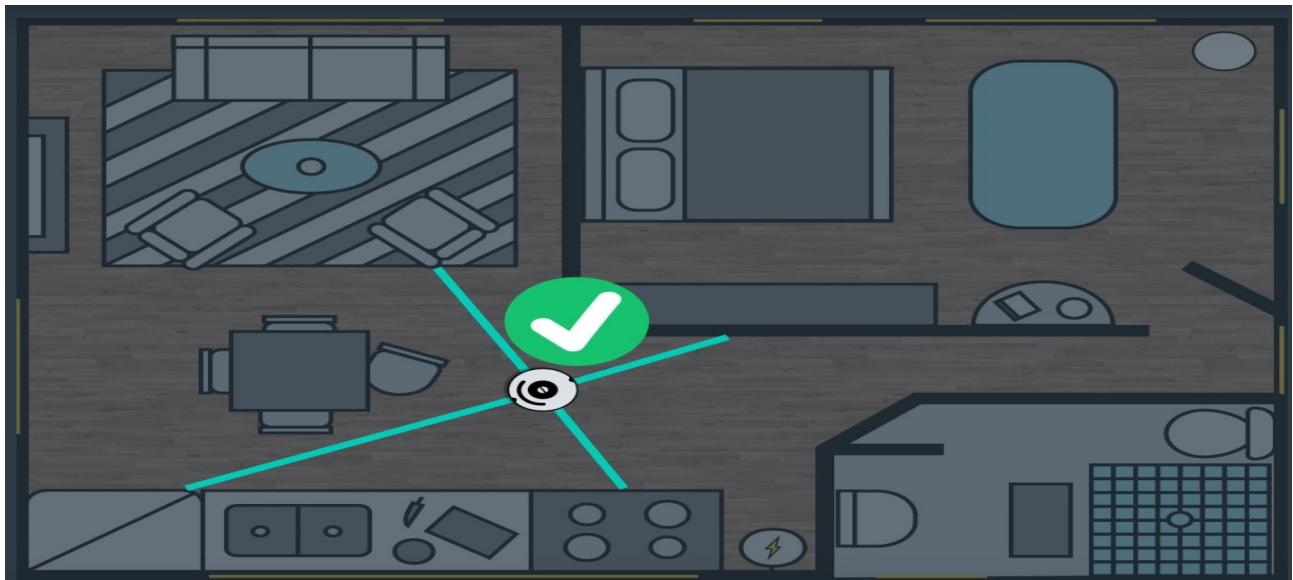
	MCL	EKF
Measurements	Raw Measurements	Landmarks
Measurement Noise	Any	Gaussian
Posterior	Particles	Gaussian
Efficiency(memory)	✓	✓✓
Efficiency(time)	✓	✓✓
Ease of Implementation	✓✓	✓
Resolution	✓	✓✓
Robustness	✓✓	x
Memory & Resolution Control	Yes	No
Global Localization	Yes	No
State Space	Multimodel Discrete	Unimodal Continuous

#### 4.5.4 Particle Filters

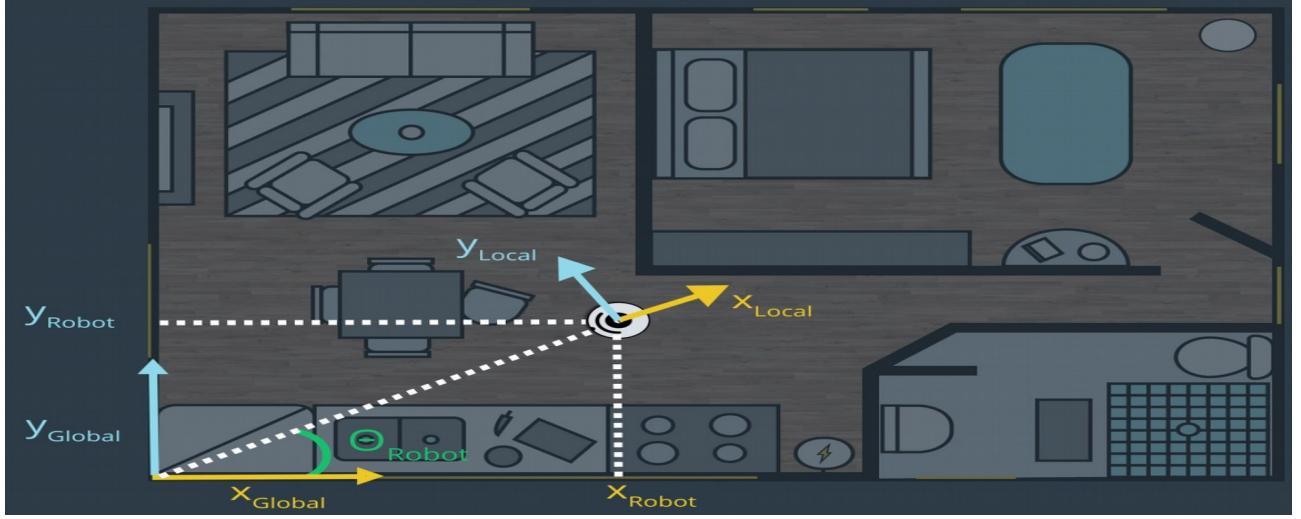
In this particular two-dimensional map, the robot has no idea where it's located at. Since the initial state is unknown, the robot is trying to estimate its pose by solving the global localization problem.



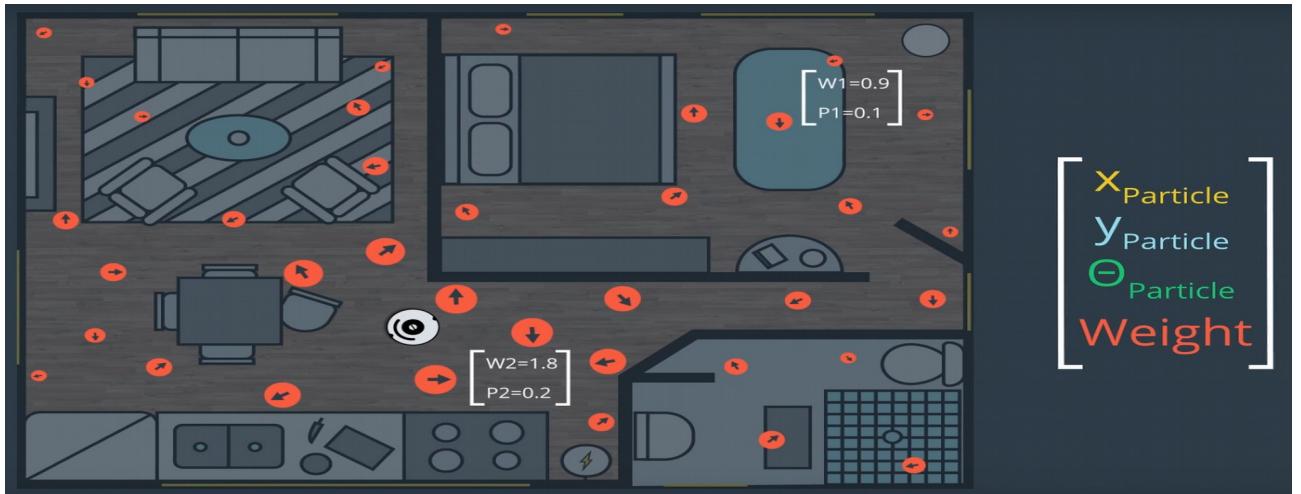
The robot has on-board sensors which permits it to sense obstacles such as objects, walls, and ultimately determine where it's located.



Here, the current robot pose is represented by an x coordinate, y coordinate and the orientation vector, all with respect to the global coordinate frame.



With the Monte Carlo localization algorithm, particles are initially spread randomly and uniformly throughout the entire map. These particles do not physically exist and are just shown in simulation. Every red circle represents a single particle, and just like the robot each particle has an x coordinate, y coordinate and orientation  $\theta$ . So, each of these particles represents a hypothesis of where the robot might be. In addition to the three-dimensional vector, particles are each assigned a weight. The weight of a particle is the difference between the robot's actual pose and the particle's predicted pose. The importance of a particle depends on its weight, and the bigger the particle, the more accurate it is. Particles with large weights are more likely to survive during the resampling process. For example, P2 will be twice as more likely to survive than P1, since it has larger weight and thus a larger probability of survival. After the resampling process, particles with significant weight are more likely to survive whereas the others are more likely to die.



Finally, after several iterations of the Monte Carlo localization algorithm, and after different stages of resampling, particles will converge and estimate the robot's pose.



#### 4.5.5 Bayes Filtering

The powerful Monte Carlo localization algorithm estimates the posterior distribution of a robot's position and orientation based on sensory information. This process is known as a recursive Bayes filter.

Using a Bayes filtering approach, roboticists can estimate the **state** of a **dynamical system** from sensor **measurements**.

In mobile robot localization, it's important to be acquainted with the following definitions:

- **Dynamical system:** The mobile robot and its environment
- **State:** The robot's pose, including its position and orientation.
- **Measurements:** Perception data(e.g. laser scanners) and odometry data(e.g. rotary encoders)

The goal of Bayes filtering is to estimate a probability density over the state space conditioned on the measurements. The probability density, or also known as **posterior** is called the **belief** and is denoted as:

$$Bel(X_t) = P(X_t | Z_{1..t})$$

Where,

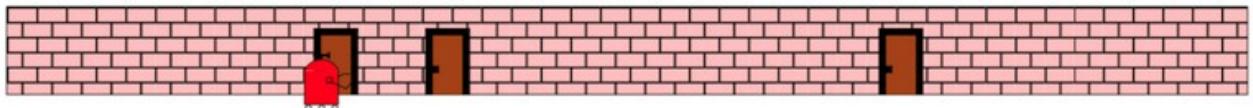
- $X_t$ : State at time t
- $Z_{1..t}$  : Measurements from time 1 up to time t

### Probability:

Given a set of probabilities,  $P(A|B)$  is calculated as follows:

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)} = \frac{P(B|A) * P(A)}{P(A) * P(B|A) + P(\neg A) * P(B|\neg A)}$$

### Quiz:



Source: Thrun, 2005 - Probabilistic Robotics

This robot is located inside of a 1D hallway which has three doors. The robot doesn't know where it is located in this hallway, but it has sensors onboard that can tell it, with some amount of precision, whether it is standing in front of a door, or in front of a wall. The robot also has the ability to move around - with some precision provided by its odometry data. Neither the sensors nor the movement is perfectly accurate, but the robot aims to locate itself in this hallway.

The mobile robot is now moving in the 1D hallway and collecting odometry and perception data. With the odometry data, the robot is keeping track of its current position. Whereas, with the perception data, the robot is identifying the presence of doors.

In this quiz, we are aiming to calculate the state of the robot, given its measurements. This is known by the belief:  $P(X_t|Z)$ !

### Given:

- $P(POS)$ : The probability of the robot being at the actual position
- $P(DOOR|POS)$ : The probability of the robot seeing the door given that it's in the actual position
- $P(DOOR|\neg POS)$ : The probability of the robot seeing the door given that it's not in the actual position

### Compute:

- **P(POS|DOOR)**: The belief or the probability of the robot being at the actual position given that it's seeing the door.

```
#include <iostream>
using namespace std;

int main() {

    //Given P(POS), P(DOOR|POS) and P(DOOR|¬POS)
    double a = 0.0002; //P(POS) = 0.002
    double b = 0.6; //P(DOOR|POS) = 0.6
    double c = 0.05; //P(DOOR|¬POS) = 0.05

    //TODO: Compute P(¬POS) and P(POS|DOOR)
    double d = 1-a; //P(¬POS)
    double e = (b*a) / ((a*b)+(d*c)); //P(POS|DOOR)

    //Print Result
    cout << "P(POS|DOOR) = " << e << endl;

    return 0;
}
```

### 4.5.6 MCL: The Algorithm

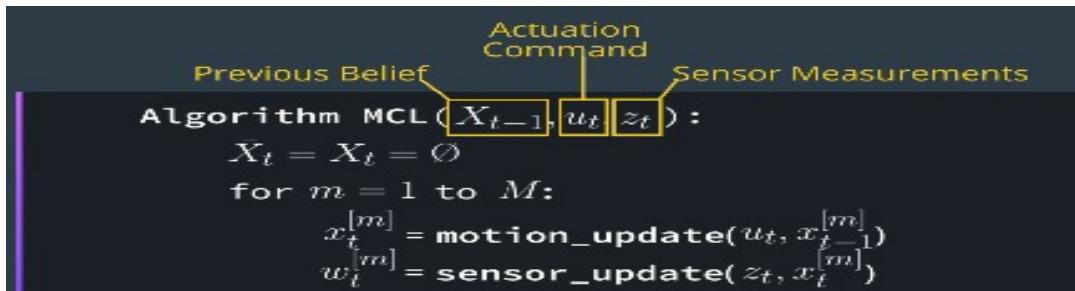
```
Algorithm MCL( $X_{t-1}, u_t, z_t$ ):
```

 $\bar{X}_t = X_t = \emptyset$ 
**for**  $m = 1$  **to**  $M$ :

 $x_t^{[m]} = \text{motion\_update}(u_t, x_{t-1}^{[m]})$   
 $w_t^{[m]} = \text{sensor\_update}(z_t, x_t^{[m]})$ 
 $\bar{X}_t = \bar{X}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$ 
**endfor**
**for**  $m = 1$  **to**  $M$ :

**draw**  $x_t^{[m]}$  **from**  $\bar{X}_t$  **with probability**  $\propto w_t^{[m]}$ 
 $X_t = X_t + x_t^{[m]}$ 
**endfor**
**return**  $X_t$ 

The Monte Carlo Localization Algorithm is composed of two main sections represented by two for loops. The first section is the motion and sensor update, and the second one is the resampling process. Given a map of the environment, the goal of the MCL is to determine the robot's pose represented by the belief. At each iteration, the algorithm takes the previous belief, the actuation command and the sensor measurements as input.



Initially the belief is obtained by randomly generating  $m$  particles. Then, in the first for loop, the hypothetical state is computed whenever the robot moves. Following, the particles weight is computed using the latest sensor measurements.

Now, motion and measurement are both added to the previous state.

Moving on to the second section of the MCL where a re-sampling process happens. Here the particles with high probability survive and are re-drawn in the next iteration, while the others die.

Finally, the algorithm outputs the new belief and another cycle of iteration starts implementing the next motion by reading the new sensor measurements.

```

Algorithm MCL( $X_{t-1}, u_t, z_t$ ):
     $\bar{X}_t = X_t = \emptyset$ 
    for  $m = 1$  to  $M$ :
         $x_t^{[m]} = \text{motion\_update}(u_t, x_{t-1}^{[m]})$ 
         $w_t^{[m]} = \text{sensor\_update}(z_t, x_t^{[m]})$ 
         $\bar{X}_t = \bar{X}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$ 
    endfor
    for  $m = 1$  to  $M$ :
        draw  $x_t^{[m]}$  from  $\bar{X}_t$  with probability  $\propto w_t^{[m]}$ 
         $X_t = X_t + x_t^{[m]}$ 
    endfor
    return  $X_t$ 

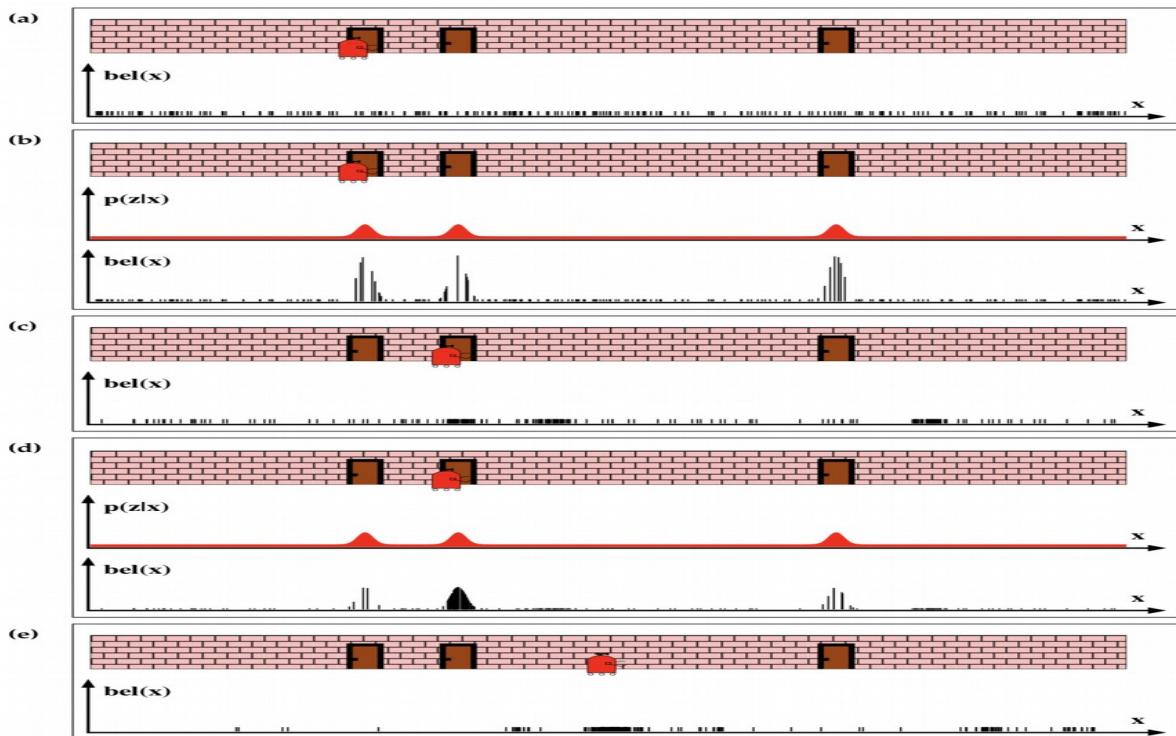
```

New Cycle of Iteration

#### 4.5.7 MCL in Action

##### MCL vs EKF in Action

###### 1- MCL:

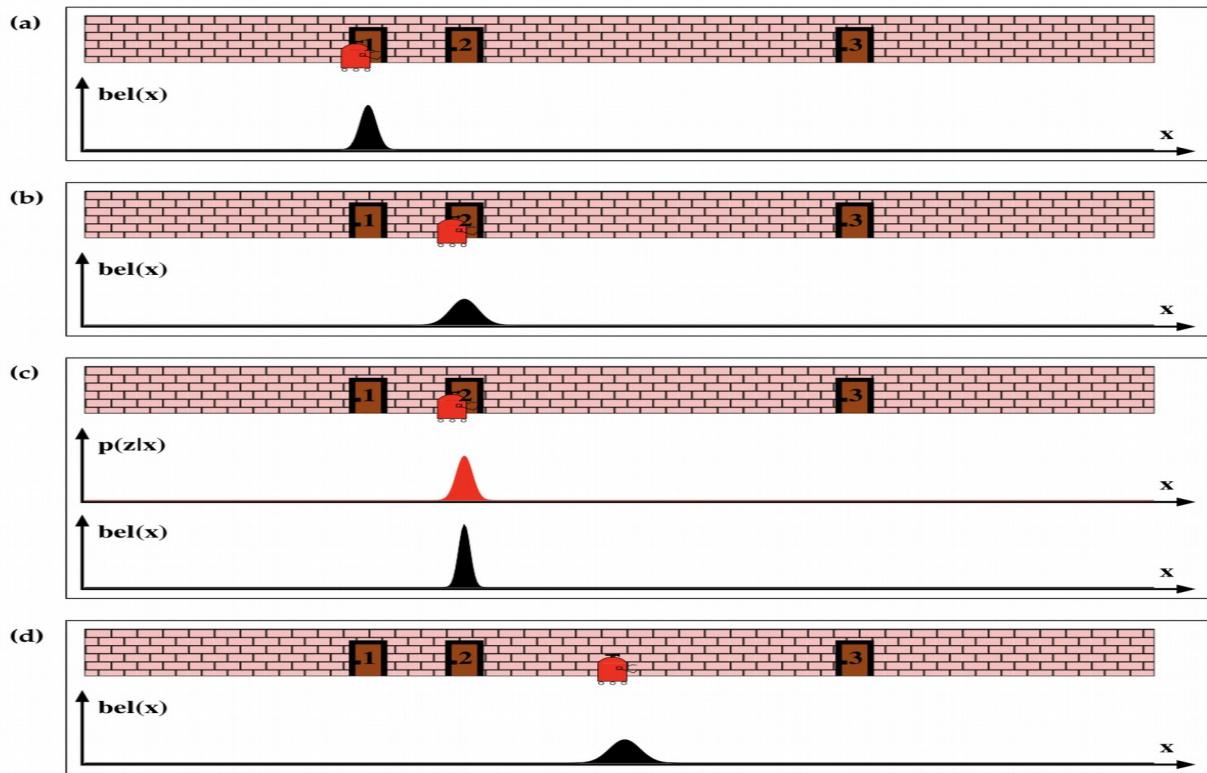


**At time:**

- **t=1**, Particles are drawn randomly and uniformly over the entire pose space.
- **t=2**, Measurement is updated and an importance weight is assigned to each particle.

- **t=3**, Motion is updated and a new particle set with uniform weights and high number of particles around the three most likely places is obtained in resampling.
- **t=4**, Measurement assigns non-uniform weight to the particle set.
- **t=5**, Motion is updated and a new resampling step is about to start.

## 2- EKF:



**At time:**

- **t=1**, Initial belief represented by a Gaussian distribution around the first door.
- **t=2**, Motion is updated and the new belief is represented by a shifted Gaussian of increased weight.
- **t=3**, Measurement is updated and the robot is more certain of its location. The new posterior is represented by a Gaussian with a small variance.
- **t=4**, Motion is updated and the uncertainty increases.

## 4.5.8 Outro

Congratulations, we just mastered the Monte Carlo Localization algorithm, we are now experienced in the two most common localization algorithm in robotics. **In the appendix you will find an implementation of the MCL algorithm in C++ in a simple environment with visualizations (images), furthermore you will see the experimental result of the MCL algorithm applied with ROS on the real robot!!!**

# 5. Introduction to Mapping and SLAM

## 5.1 Introduction

In this chapter we will learn how robots build maps of their environment and how they perform Simultaneous Localization and Mapping or SLAM, we have already examined Localization so we are halfway there and after learning Mapping, we will be prepared to take on the bigger challenge of SLAM.

As you recall, in localization

### **Localization.**

Estimate the robot's pose given the map of the environment.

The robot is provided the map of its environment, the robot also has access to its movement and sensor data and using the three of these together, it must estimate its pose.

But what if the map of the environment doesn't exist? There are many applications that there isn't a known map either because the area is unexplored or because the surroundings change often and the map may not be up to date.

In such a case, the robot will have to construct a map, this leads us to robotic mapping.

### **Mapping.**

Produce a map of the environment given the robot's pose(s).

Mapping assumes that a robot knows its pose, and as usual has access to its movements and sensor data. The robot must produce a map of the environment using the known trajectory and measurement data.

However, even such a case can be quite uncommon in the real world. Often, you neither have a map nor know the robot's pose.

This is where SLAM comes in, with access only to robot's own movement and sensory data,

### **SLAM**

Simultaneous Localization & Mapping

Simultaneously estimate the robot's pose and produce a map of the environment.

The robot must build a map of its environment while simultaneously localizing itself relative to the map.

## Mapping

Robot mapping sounds quite similar to localization, instead of assuming a known map and estimating your pose, you're assuming a known path and estimating your environment.

So what can be so hard?

Well, your pose can be defined with some finite number of state variables such as the robot's x and y position. In most applications, there are only a handful.

On the other hand, a map generally lies in a continuous space, and as a result, there are infinitely many variables used to describe it. Couple this with the uncertainties present in perception and the mapping task becomes even more challenging.

Furthermore, there are other challenges too depending on the nature of the space that you are mapping. For instance, the geometry and whether the space is repetitive in the sense that many different places look alike. For example, if you're driving through a forest which only contains one type of tree, the kind where they are neatly planted in rows, you're going to have a lot of trouble distinguishing between trees and establishing correspondences. Thus, making it challenging to map the environment.

In the next chapter, we will learn how to map an environment, there are a number of different mapping algorithms and we will be focusing on the **Occupancy Grid Mapping Algorithm**. With this algorithm you can map any arbitrary environment by dividing it into a finite number of grid cells, by estimating the sets of each individual cell, we will end up with an estimated map of the environment.

## SLAM

In SLAM, a robot must construct a map of the environment, while simultaneously localizing itself relative to this map. This problem is more challenging than localization and mapping, since neither the map nor the robots poses are provided. With noise in the robot's motions and measurements, the map and robot's pose will be uncertain, and the errors in the robot's pose estimates and map will be correlated. The accuracy of the map depends on the accuracy of the localization and vice versa. SLAM is often called the chicken or the egg problem (**Fun Fact The egg came first**) because the map is needed for localization, and the robot's pose needed for mapping. **SLAM truly is a challenge, but it's fundamental to mobile robotics.**

For robots to be useful to society, they must be able to move in environments they've never seen before. For instance, one of SLAM's applications is the robot vacuum cleaner, such a vacuum comes equipped with sensors that help it map the room and estimate its pose as it vacuums. Other applications of SLAM include self driving vehicles, be on the roads, or underground mines, aerial surveillance, and even robots mapping the surface of mars.

SLAM algorithms generally fall into five categories seen in the image below.

## SLAM Algorithms

- Extended Kalman Filter SLAM (EKF)
- Sparse Extended Information Filter (SEIF)
- Extended Information Form (EIF)
- FastSLAM
- GraphSLAM

Later we will be introduced to the FastSLAM algorithm which uses the particle filter approach along with a low dimensional EKF to solve the SLAM problem, we will then adapt this algorithm to grid maps, which will result in a grid-based fastSLAM algorithm.

Finally, we will learn graphSLAM, which uses constraints to represent relationships between robot poses and the environment, and then tries to resolve all of these constraints to create the most likely map given the data.

**We will also learn and see experimental result of a graphSLAM implementation called Real Time Appearance Based Mapping, or RTABmap.**

## 5.2 Occupancy Grid Mapping

### 5.2.1 Introduction

#### Localization:

- *Assumption:* Known Map
- *Estimation:* Robot's Trajectory

#### Mapping:

- *Assumption:* Robot's Trajectory
- *Estimation:* Map

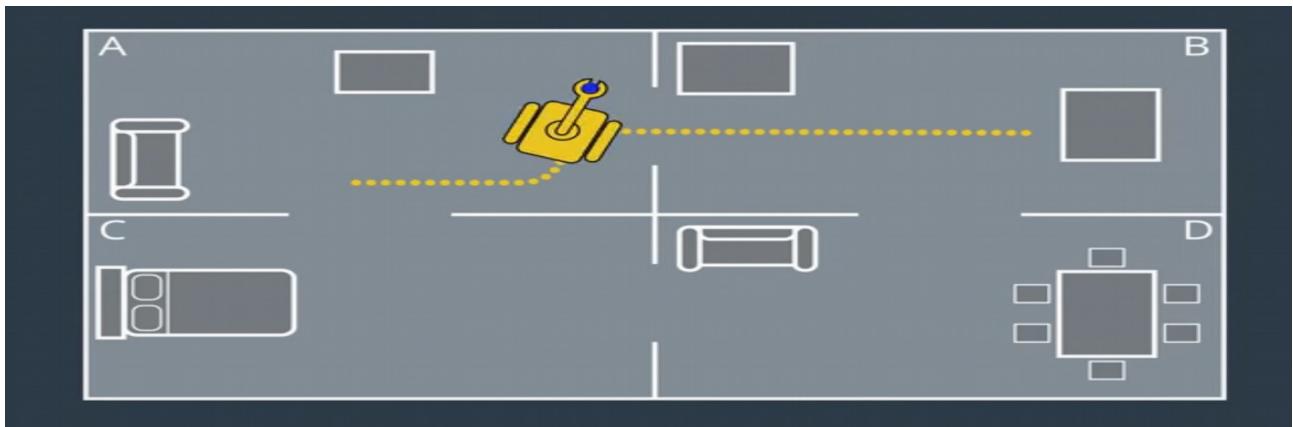
In this chapter, we'll learn how to **map** an environment with the **Occupancy Grid Mapping** algorithm!

### 5.2.2 Importance of Mapping

We just saw that mapping is required in undiscovered areas since the priority map is unavailable, but what if you just want to deploy a robot in your home?

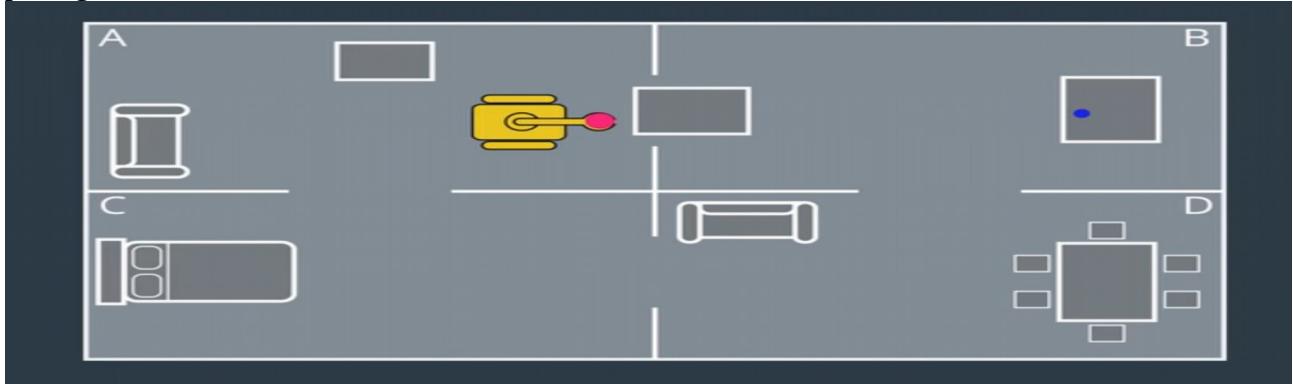
Well, you can almost always provide the robot with the map of your home and expect it to perform and navigate accurately.

In this example, we'll test if a robot with a priority map will navigate as expected. Here's an example of a mobile robot with a robotic arm deployed inside the home, the map of this home is uploaded to the robot which is unable to perform SLAM. Now the robot is being tasked to pick up a cup from room A and deliver it to room B. It has computed its trajectory and starts navigating from room A to room B.



Currently, the robot is only performing localization. Great! It has successfully moved from room A to room B and delivered the cup.

Let's try it again, but this time the resident of the house has rearranged the furniture, as a result, the passage between room A and B is now blocked.



The robot is once again commanded to deliver an item to room B, it picks up the object and starts navigating and suddenly hits one of the pieces of furniture and gets stopped. So why this happened and how we can keep this from happening again?

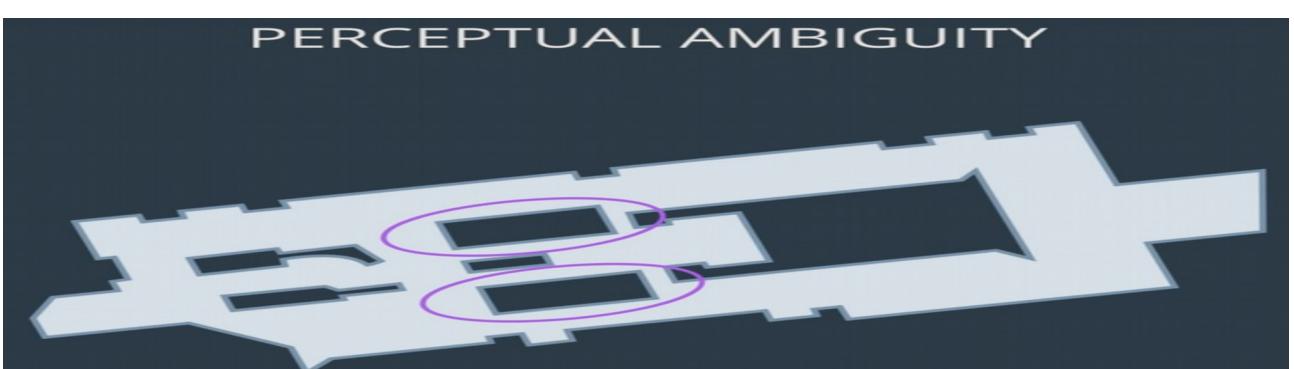
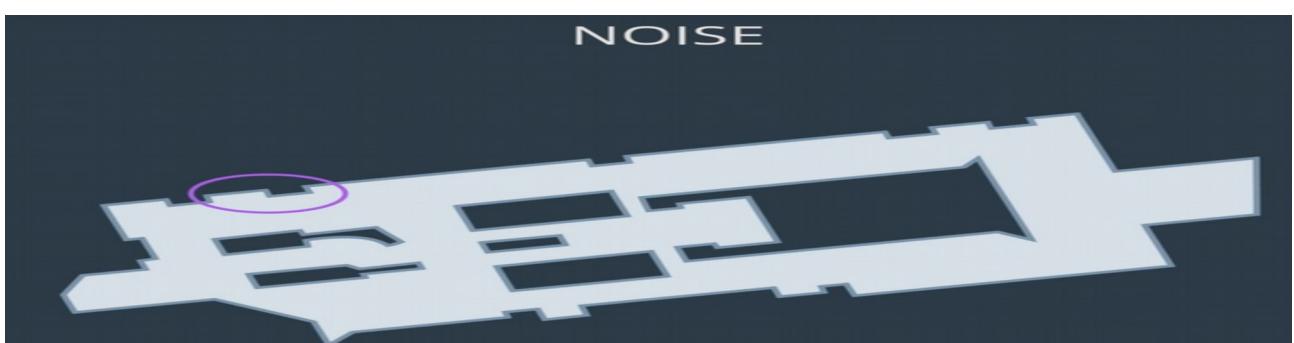
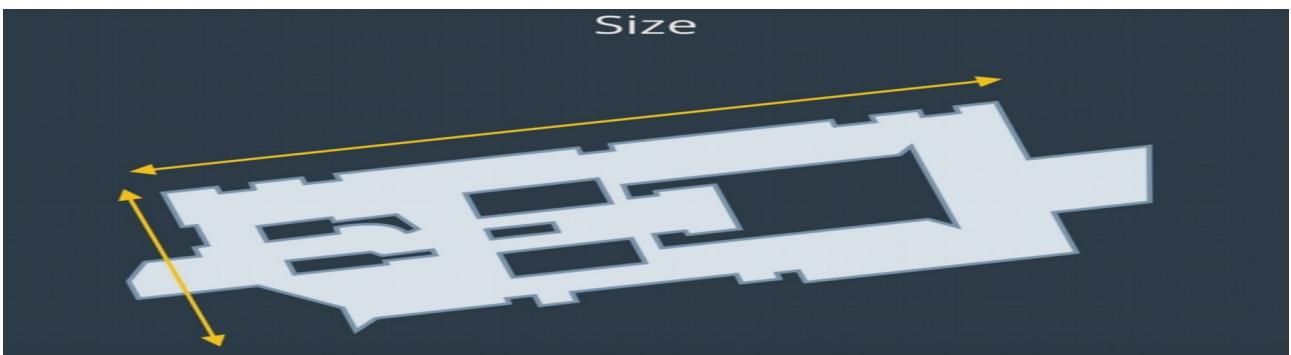
Well, the robot is unable to adapt to changes and update its own map. Mapping is critical in dynamic environments where objects may shift over time, if the map had been updated after rearranging the walls the robot would have taken a different path.

Thus, in changing environments, the robot should always perform instantaneous mapping in both discovered and undiscovered areas, even if the priority map is available.

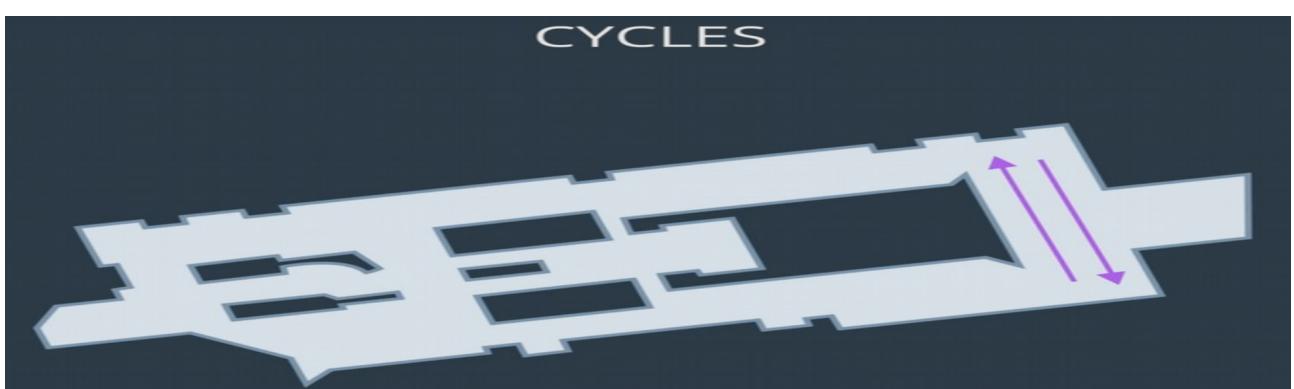
### 5.2.3 Challenges and Difficulties

A mobile robot is mapping a large environment while traveling in cycles and correlating between different objects seen at different points in time. What are the challenges and difficulties faced by this robot?

- Unknown Map
- Huge Hypothesis Space
- Size
- Noise
- Perceptual Ambiguity
- Cycles



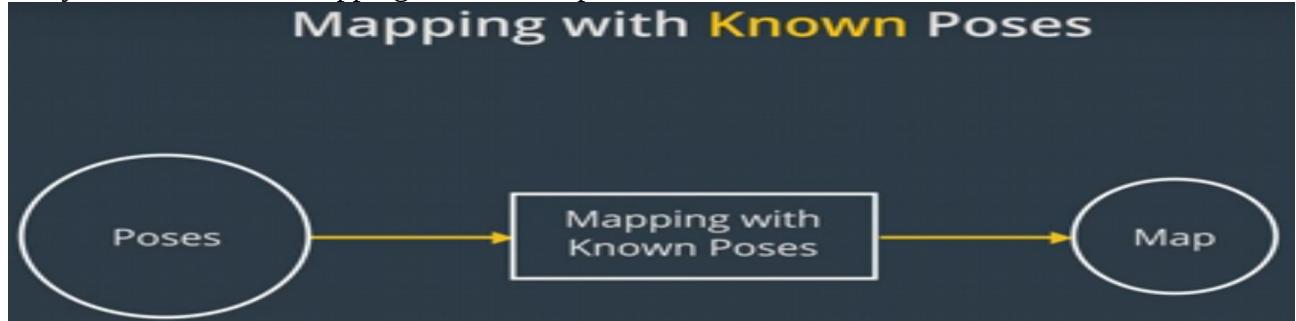
- **Discrete Data:** You obtain this data by counting it. This data has finite values. *Example:* Number of robots in a room



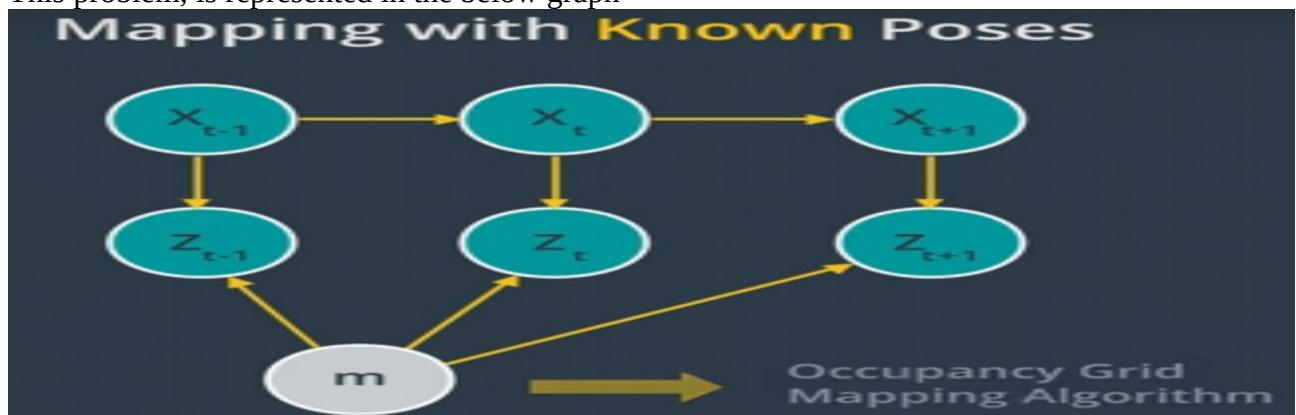
- **Continuous Data:** You obtain this data by measuring it. This data has an infinite number of steps, which form a continuum. *Example: Weight* of a robot

## 5.2.4 Mapping with Known Poses

The problem of generating a map under the assumption that the robot poses are known and non-noisy, is referred to as mapping with known poses.

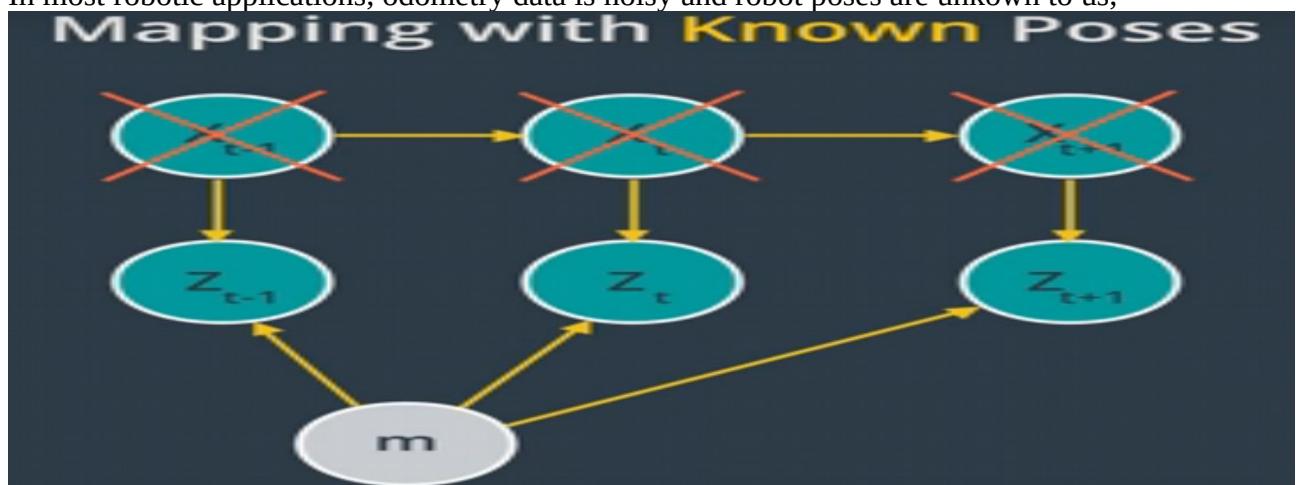


This problem, is represented in the below graph

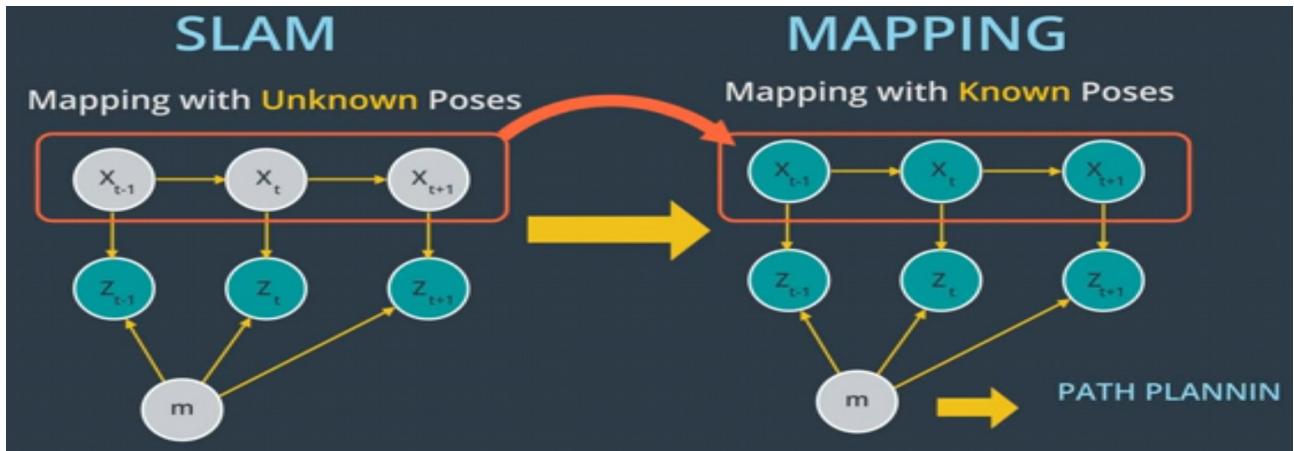


Where X represents the poses, Z the measurements and M the map. **The Occupancy Grid Mapping Algorithm can estimate the posterior map giving noisy measurements and known poses.**

In most robotic applications, odometry data is noisy and robot poses are unknown to us,



So you might be wondering why mapping is necessary under such a situation.  
Well, mapping usually happens after SLAM,

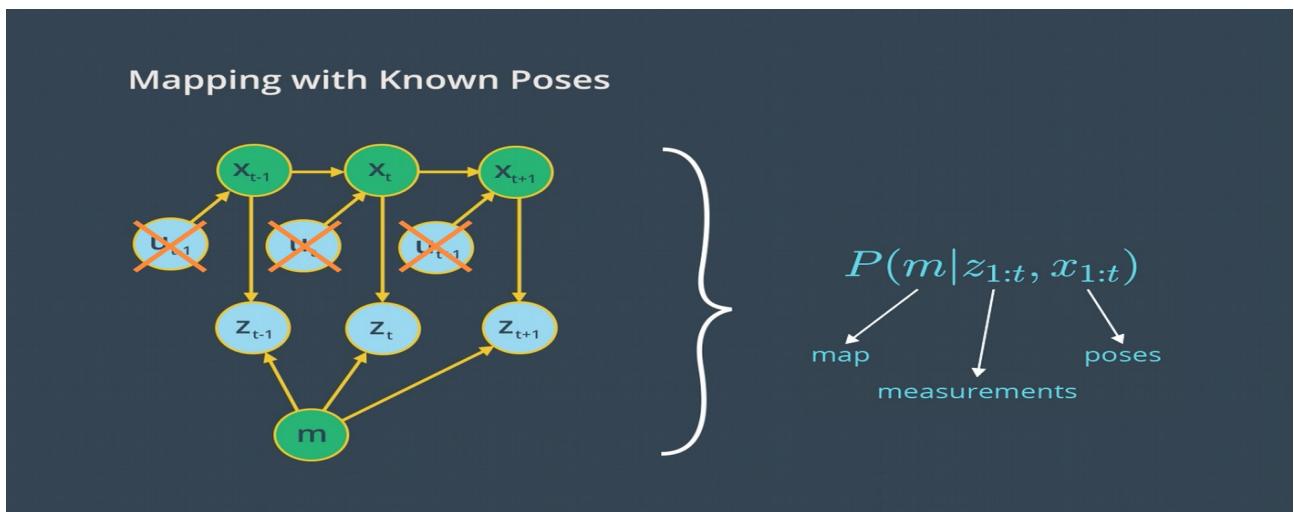


thus, the power of mapping is in its post-processing. In SLAM, the problem changes from mapping with known poses to mapping with unknown poses. During SLAM, the robot built a map of the environment and localize itself relative to it. After SLAM, the Occupancy Grid Mapping Algorithm uses the exact robot poses filtered from SLAM, **then with the known poses from SLAM and noisy measurements, the mapping algorithm can generate a map fit for path planning and navigation.**

What are the inputs and outputs to the Mapping with Known Poses problem?

- **Inputs:** Poses + Measurements, **Output:** Map

### 5.2.5 Posterior Probability



### Posterior Probability

Going back to the graphical model of mapping with known poses, our goal is to implement a mapping algorithm and estimate the map given noisy measurements and assuming known poses.

The Mapping with Known Poses problem can be represented with  $P(m|z_{1:t}, x_{1:t})$  function. With this function, we can compute the posterior over the map given all the measurements up to time  $t$  and all the poses up to time  $t$  represented by the robot trajectory.

In estimating the map, we'll exclude the controls  $\mathbf{u}$  since the robot path is provided to us from SLAM. However, keep in mind that the robot controls will be included later in SLAM to estimate the robot's trajectory.

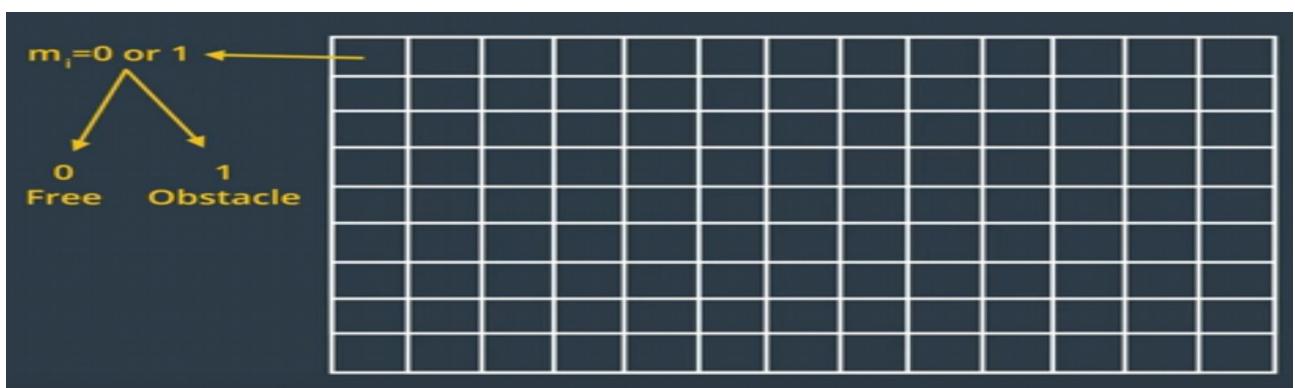
## 2D Maps

For now, we will only estimate the posterior for two-dimensional maps. In the real world, a mobile robot with a two-dimensional laser rangefinder sensor is generally deployed on a flat surface to capture a slice of the 3D world. Those two-dimensional slices will be merged at each instant and partitioned into grid cells to estimate the posterior through the occupancy grid mapping algorithm. Three-dimensional maps can also be estimated through the occupancy grid algorithm, but at much higher computational memory because of the large number of noisy three-dimensional measurements that need to be filtered out.

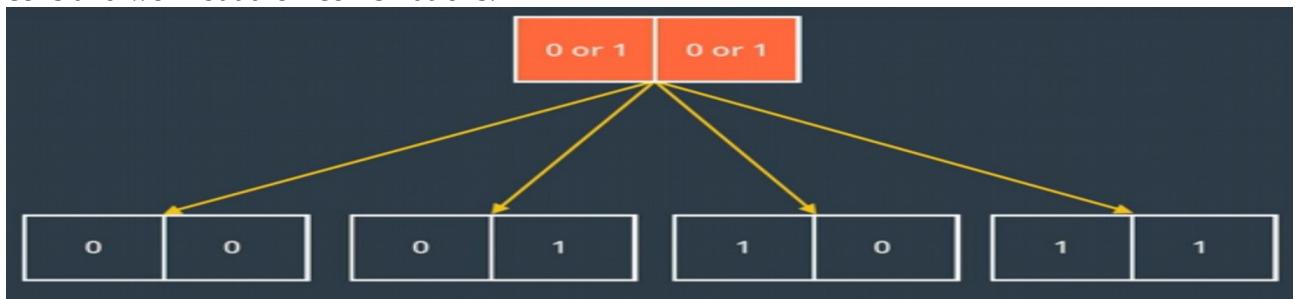
- **Localization** :  $P(x_{1:t} | u_{1:t}, m, z_{1:t})$
- **Mapping** :  $P(m | z_{1:t}, x_{1:t})$
- **SLAM** :  $P(x_{1:t}, m | z_{1:t}, u_{1:t})$

### 5.2.6 Grid Cells

To estimate the posterior map, the Occupancy Grid Mapping Algorithm will uniformly partition the two dimension space in a finite number of grid cells. Each of these grid cells will hold a binary random value that corresponds to the location it covers. Based on the measurement data, this grid space will be filled with zeros and ones, if the laser range finder sensor detects an obstacle, the cell will be considered as occupied and its value will be one and for free spaces the cell will be considered unoccupied and its value will be zero.



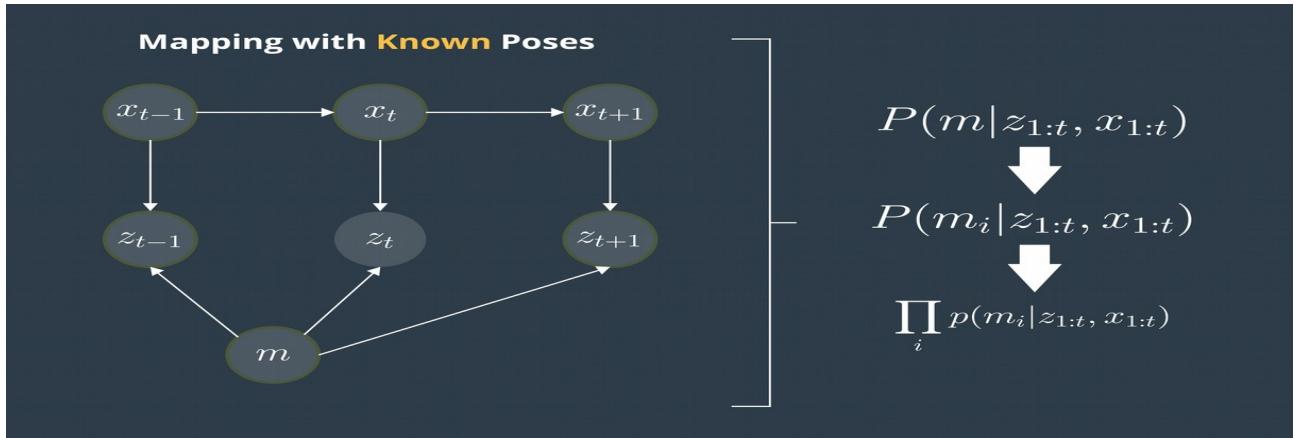
Now, to find out the number of maps that can be generated out of grid cells, let's pick the first two cells and work out their combinations.



Thus, the total number of maps that can be formed out of these two cells is equal to four. To generalize, the number of maps that can be formed out of grid cells is equal to  $2^n$  where n is the number of grid cells.

Now, imagine if you had a 2D space with tens of thousands of grid cells, the number of possible maps would be huge!!

### 5.2.7 Computing the Posterior



#### First Approach: $P(m|z_{1:t}, x_{1:t})$

We just saw that maps have high dimensionality so it will be too pricey in terms of computational memory to compute the posterior under this first approach.

#### Second Approach: $P(m_i|z_{1:t}, x_{1:t})$

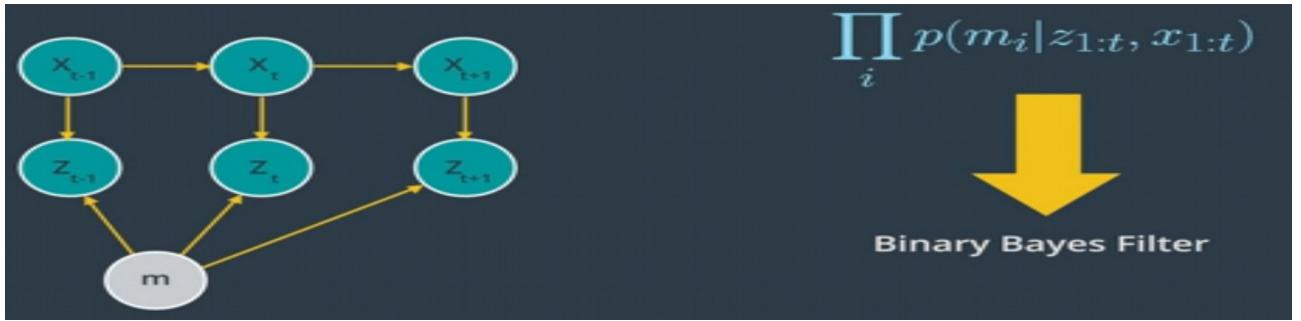
A second or better approach to estimating the posterior map is to decompose this problem into many separate problems. In each of these problems, we will compute the posterior map  $m_i$  at each instant. However, this approach still presents some drawbacks because we are computing the probability of each cell independently. Thus, we still need to find a different approach that addresses the dependencies between neighboring cells.

#### Third Approach: $\prod_i P(m_i|z_{1:t}, x_{1:t})$

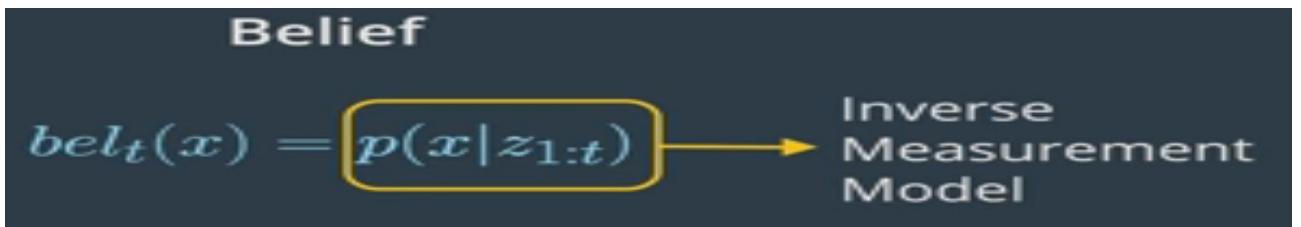
Now, the third approach is the best approach to computing the posterior map by relating cells and overcoming the huge computational memory, is to estimate the map with the product of marginals or factorization.

### 5.2.8 Filtering

So far we have managed to represent the probability of grid cells using the factorization method, due to factorization we are now solving the binary estimation problem in which grid cells holds a static state that do not change over time. By static, I mean that the state of the system does not change during sensing. Luckily, a filter to this problem exists, and is known as the Binary Bayes Filter.

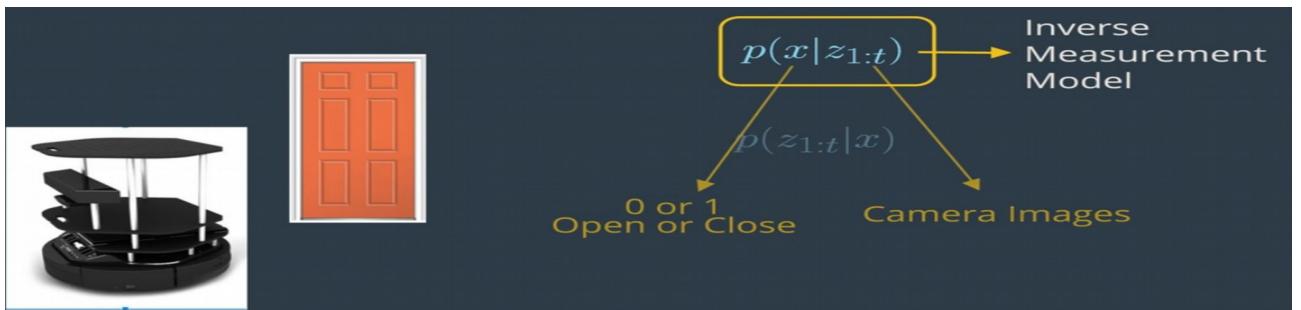


The Binary Bayes Filter solves the static problems by taking the log odds ratio of the belief. With static state, the belief is now a function of the measurements only. Depending on the measurement values reflected, the state of the grid cell is updated, and this belief is known by the inverse measurement model which represents the binary state of grid cells with respect to measurements.



**The Inverse Measurement Model is generally used when measurements are more complex than the binary static state.**

Let's take an example of a mobile robot equipped with an RGB-D camera. Here's the robot mission is to estimate if the door is open or closed. The field of measurements represented by the camera images is huge compared to a simple binary state of the door either open or close. In such situation, it's always easier to use an inverse sensor model than a forward sensor model.



As previously mentioned the Binary Bayes Filter will solve the Inverse measurement model with the log odds ratio representation.



To represent the belief in log odds ratio for it, the Binary Bayes Filter divides the belief by its negate and then take the logarithmic form of this expression. The advantage of using a log odds ratio representation is to avoid probability instabilities near zero or one.. Another advantage relates to system speed, accuracy, and simplicity. Check out these two sources for more information on log probability and numerical stability:

1. [Log Probability](#)
2. [Numerical Stability](#)

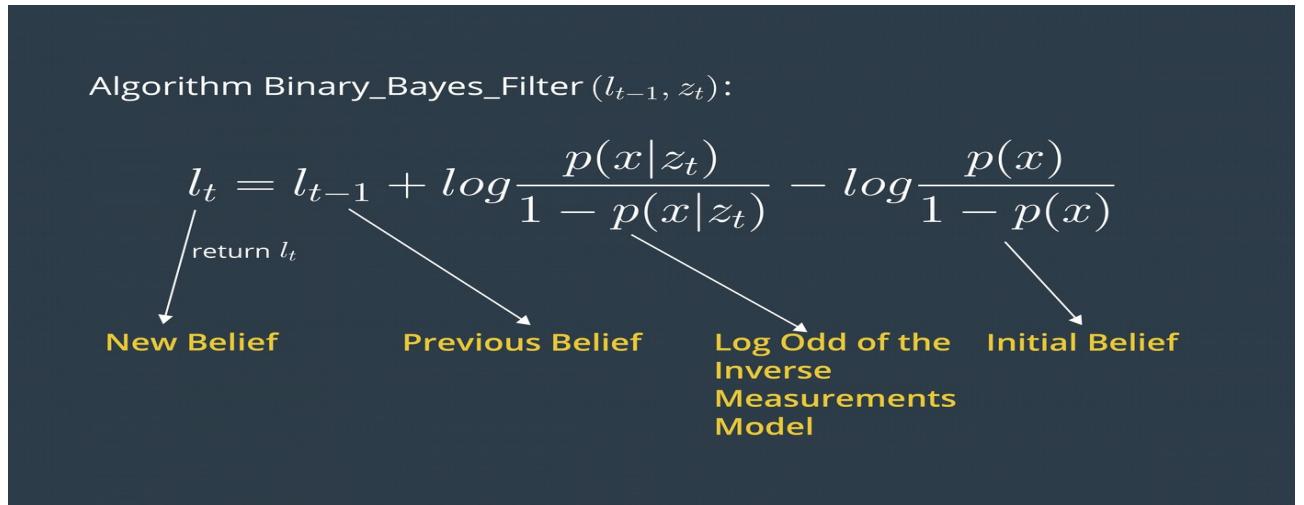
## Forward vs. Inverse Measurement Model

**Forward Measurement Model** -  $P(z_{1:t} | x)$ : Estimating a posterior over the measurement given the system state.

**Inverse Measurement Model** -  $P(x | z_{1:t})$ : Estimating a posterior over the system state given the measurement.

The inverse measurement model is generally used when measurements are more complex than the system's state.

### 5.2.9 Binary Bayes Filter Algorithm



#### Input

The binary Bayes filter algorithm computes the log odds of the posterior belief denoted by  $l_t$ .

Initially, the filter takes the previous log odds ratio of the belief  $l_{t-1}$  and the measurements  $z_t$  as parameters.

#### Computation

Then, the filter computes the new posterior belief of the system  $l_t$  by adding the previous belief  $l_{t-1}$  to the log odds ratio of the inverse measurement model  $\log \frac{p(x|z_t)}{1 - p(x|z_t)}$  and subtracting the prior probability state also known by initial belief  $\log \frac{p(x)}{1 - p(x)}$ . The initial belief represents the initial state of the system before taking any sensor measurements into consideration.

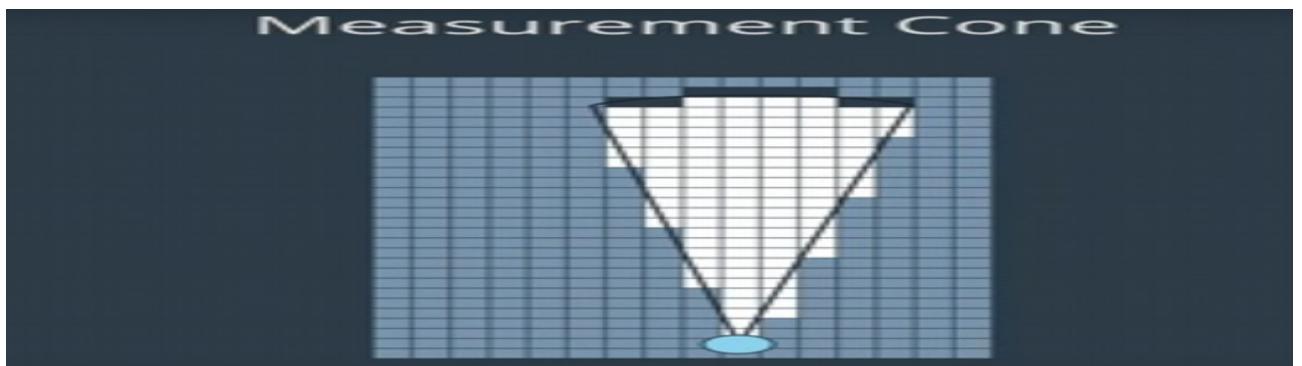
## Output

Finally, the algorithm returns the posterior belief of the system  $\mathbf{l}_t$ , and a new iteration cycle begins.

### 5.2.10 Occupancy Grid Mapping Algorithm

The occupancy grid mapping algorithm implements a binary base filter to estimate the occupancy value of each cell. Initially the algorithm takes the previous occupancy values of the cells, the poses and the measurements as parameters. The algorithm now loops through all the grid cells, for each of the cells, the algorithm tests if the cells are currently being sensed by the range finder sensors.

To better understand this condition lets take a look at an example. Here, we can see a mobile robot sensing the environment. The cells that fall under the measurement cone are highlighted in white and black color.



While looping through all the cells, the algorithm will consider those white and black cells as cells falling under the perceptual field of the measurements. Now, the algorithm will update the cells that fall under the measurement cones by computing their new belief using the binary base filter algorithm. As we previously said the new state of the cell is computed by adding the previous belief to the inverse sensor model. The Inverse Sensor model represents the probability of the posterior map giving the measurements and the poses and subtracting the initial belief which is the initial state of the map in its log odd form.

```

Algorithm Occupancy_Grid_Mapping ( $\{\mathbf{l}_{t-1,i}\}, \mathbf{x}_t, \mathbf{z}_t$ ):
    for all cells  $m_i$  do
        if  $m_i$  in perceptual field of  $z_t$  then
             $l_{t,i} = l_{t-1,i} + \text{Inverse_Sensor_Model}(m_i, x_t, z_t) - l_0$ 
    endfor
    Previous Belief
    
$$\log \frac{p(m_i|z_t, x_t)}{1 - p(m_i|z_t, x_t)}$$

    
$$\log \frac{p(m_i)}{1 - p(m_i)}$$


```

For the cells that do not fall under the measurement cone, their occupancy value remains unchanged. Now the algorithm returns the updated occupancy values of the cells and another cycle of iteration starts.

```

Algorithm Occupancy_Grid_Mapping ({l_{t-1,i}}, x_t, z_t):
    for all cells m_i do
        if m_i in perceptual field of z_t then
            l_{t,i} = l_{t-1,i} + Inverse_Sensor_Model(m_i, x_t, z_t) - l_0
        else
            l_{t,i} = l_{t-1,i}
        endif
    endfor

```

### 5.2.11 Inverse Sensor Model

In the Occupancy Grid Mapping Algorithm we saw the inverse sensor model which is the probability of the map giving the measurements and the poses in its log odds representation. To demonstrate that, let's recover the probability of the map giving the data from the log odds representation.

```

Algorithm Occupancy_Grid_Mapping ({l_{t-1,i}}, X_t, Z_t):
    for all cells m_i do
        if m_i in perceptual field of z_t then
            l_{t,i} = l_{t-1,i} + Inverse_Sensor_Model(m_i, x_t, z_t) - l_0
        else
            l_{t,i} = l_{t-1,i}
        endif
    endfor
    return {l_{t,i}}

```

$$\log \frac{p(m_i|z_t, x_t)}{1 - p(m_i|z_t, x_t)} \rightarrow p(m_i|z_{1:t}, x_{1:t}) = \underbrace{\frac{1}{1 + \exp(l_{t,i})}}_{\text{Range : } ]0,1[}$$

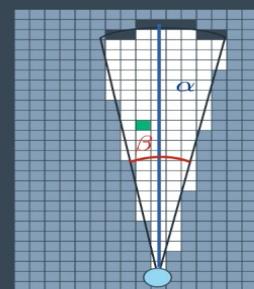
Now, we can see how the probability of the map is bounded between zero and one. Mobile robots equipped with range finder sensors instantaneously detect obstacles and return measurements in a conical field of view. At the borders of this cone robots can only sense portions of these cells and thus, probabilities are weak. Due to weak probabilities it's hard and complex to compute the state of the cells at the border of the cone. **To simplify this problem and estimate the state of the cells within, at the border and outside of the measurement cone an inverse sensor model is introduced.** With this algorithm, you can estimate if the current cells is free, occupied or unknown.

The inverse sensor model algorithm takes each cell the poses and measurements as parameters.

```

Algorithm Inverse_Sensor_Model(m_i, x_t, z_t):
    Let x_i, y_i be the center of mass of m_i
    r =  $\sqrt{(x_i - x)^2 + (y_i - y)^2}$ 
     $\Phi = \text{atan2}(y_i - y, x_i - x) - \theta$ 
    k = argmin_j |Φ - θ_{j,sens}|
    if  $r > \min(z_{max}, z_t^k + \alpha/2)$  or  $|\Phi - \theta_{k,sens}| > \beta/2$  then
        return l_0
    if  $z_t^k < z_{max}$  and  $|r - z_t^k| < \alpha/2$ 
        return l_occ
    if  $r \leq z_t^k$ 
        return l_free

```



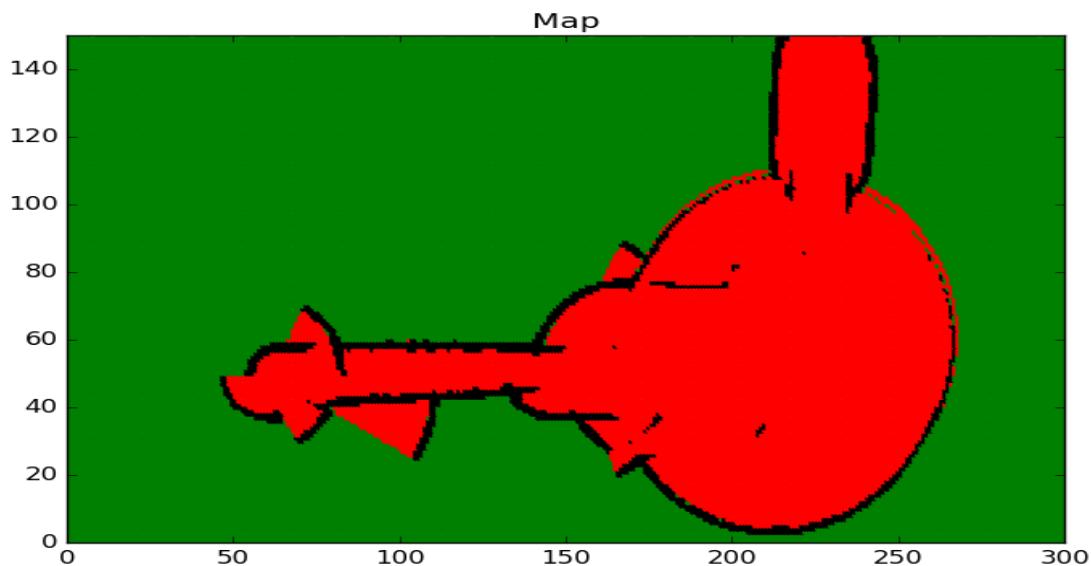
- First, the range for the center of mass of the cell is computed, here  $x_i$  and  $y_i$  are the center of mass of the cell  $m_i$ ,  $r$  is the range for the center of mass, computed with respect to the center of mass and robot position.
- Next, the beam index  $k$  is computed with respect to the center of mass of the cell and the robot pose
- $\beta$  is the opening angle of the mean and  $\alpha$  is the width of a cell
- if a cell lies outside the measurement range of the sensor beam or more than  $\alpha$  over two behind its detected range, then the state of this cell is considered unknown and its initial or prior state  $l_0$  and its log odds form is returned.
- Now, the cell is considered as occupied if it ranges between negative and positive  $\alpha$  over two of the detected range.
- Finally, the cell is considered as free if the range of this cell is shorter than the measurement range by more than  $\alpha$  over two.

### **Summary of notations for the sonar rangefinder inverse sensor model:**

- $m_i$ : Map at instant  $i$  or current cell that is being processed
- $x_i, y_i$ : Center of mass of the current cell  $m_i$
- $r$ : Range of the center of mass computed with respect to robot pose and center of mass
- $k$ : The sonar rangefinder cone that best aligns with the cell being considered computed with respect to the robot pose  $(x, y, \theta)$ , center of mass  $(x_i, y_i)$ , and sensor angle.
- $\beta$ : Opening angle of the conical region formed out of the measurement beams.
- $\alpha$ : Width of obstacles which is almost equal to the size of a cell.

**To see how to program the Occupancy Grid Mapping algorithm in C++ with visualization check the appendix and the corresponding github repo.**

### **Generated Map**



## Map Legend

- Green: Unknown/Undiscovered zone
- Red: Free zone
- Black: Occupied zone

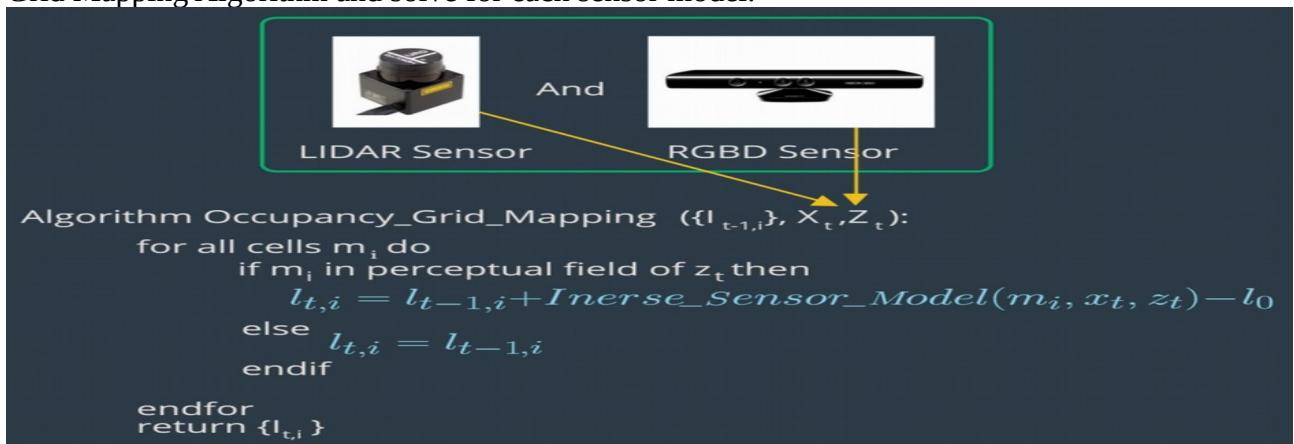
### 5.2.12 Multi Sensor Fusion

So far, we've covered mapping for robots equipped with only one sensor, such as mobile robots equipped with an ultrasonic sensor or LIDAR sensor or even an RGB-D sensor. Sometimes the mobile robot might be equipped with a combination of these sensors.



Mapping with a mobile robot equipped with a combination of these sensors leads to a more precise map.

But how would you combine the information from a LIDAR and RGB-D sensor or any other combination into a single map. Well, an intuitive approach would be to implement the Occupancy Grid Mapping Algorithm and solve for each sensor model.



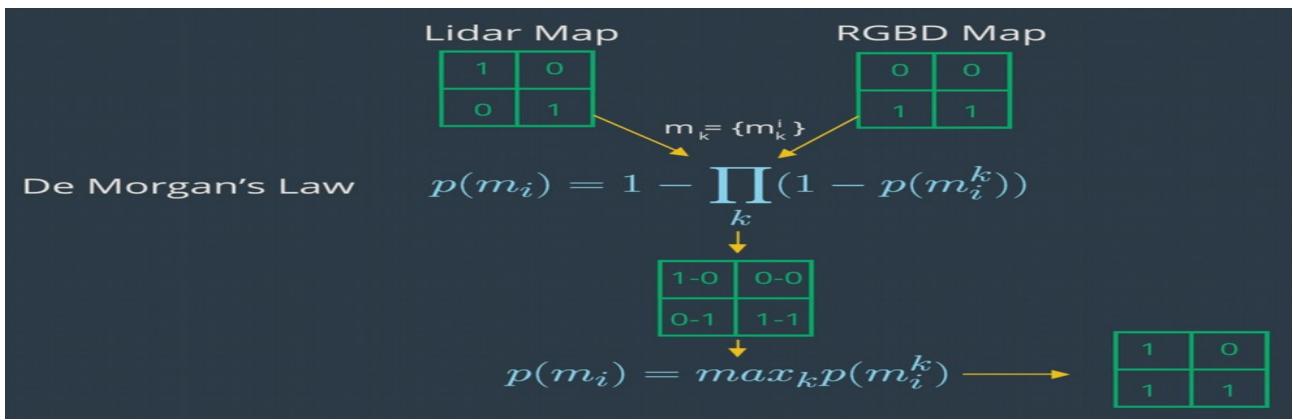
But this will fail since each and every sensor has different characteristics. In addition to that, sensors are sensitive to different obstacles. For example, an RGB-D camera might detect an obstacle at a particular location but this object might not be seen by the laser beams of a LIDAR, and thus the LIDAR will process it as a free space.

The best approach to the multi sensor fusion problem is to build separate maps for each sensor type independently of each other and then integrate them.

Here you can see two independent 2x2 maps created by each sensor. Within the same environment both maps do not look the same since each sensor has a different sensitivity.



Now let's denote the map built by the Kth sensor as  $M_k$  and combine these maps.



If the measurements are independent of each other, we can easily combine the maps using De Morgan's Law. The resultant map now combines the estimated occupancy values of each cell. To obtain the most likely map, we need to compute the maximum value of each cell, another approach would be to perform a null operation between values of each cell. The resulting map now perfectly combines the measurement from different sensors.

### 5.2.13 Introduction to 3D Mapping

So far, you've heard about two dimensional maps, describing a slice of the 3D world. In resource constrained systems, it can be very computationally expensive to build and maintain these maps. 3D representations are even more costly. That being said, robots live in the 3D world, and we want to represent that world and the 3D structures within it as accurately and reliably as possible. 3D mapping would give us the most reliable collision avoidance, and motion and path planning, especially for flying robots **or mobile robots with manipulators**.

First, let's talk briefly about how we collect this 3D data, then we will move on to how it is represented. To create 3D maps, robots sense the environment by taking 3D range measurements. This can be done using numerous technologies.

3D lidar can be used, which is a single sensor with an array of laser beams stacked horizontally. Alternatively, a 2D lidar can be tilted (horizontally moving up and down) or rotated (360 degrees) to obtain 3D coverage.

An RGBD camera is a single visual camera combined with a laser rangefinder or infrared depth sensor, and allows for the determination of the depth of the image, and ultimately the distance from

an object. A stereo camera is a pair of offset cameras, and can be used to directly infer the distance of close objects, in the same way as humans do with their two eyes.

A single camera system is cheaper and smaller, but the software algorithms needed for monocular SLAM are much more complex. Depth cannot be directly inferred from the sensor data of a single image from a single camera. Instead, it is calculated by analysing data from a sequence of frames in a video.

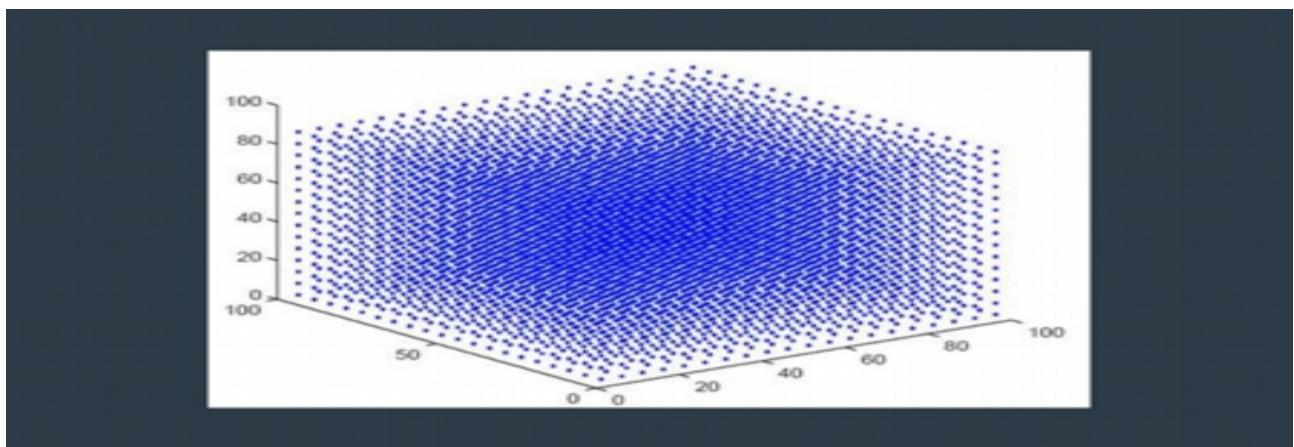
### 5.2.14 3D Data Representations

#### Some of the desired characteristics of an optimal representation:

- Probabilistic data representations can be used to accommodate for sensor noise and dynamic environments.
- It is important to be able to distinguish data that represents an area that is free space versus an area that is unknown or not yet mapped. This will enable the robot to plan an unobstructed path and build a complete map.
- Memory on a mobile robot is typically a limited resource, so memory efficiency is very important. The map should also be accessible in the robot's main memory, while mapping a large area over a long period of time. To accomplish this, we need a data representation that is compact and allows for efficient updates and queries.

#### Point Clouds

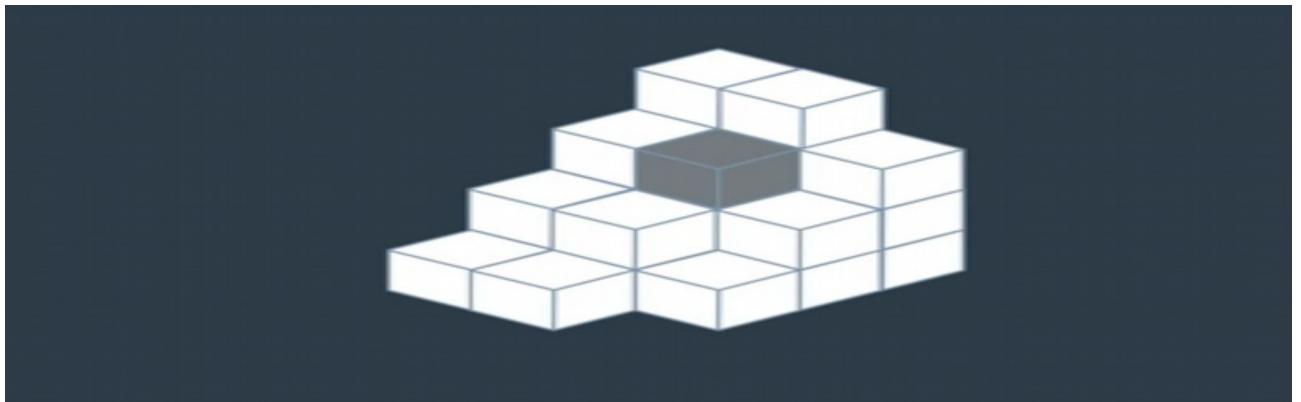
A 3D point cloud is a set of data points corresponding to range measurements, each at defined x y z coordinates. A disadvantage with point cloud data is that information only exists about where things are in the world, the data looks the same whether the space is unoccupied or unknown. Point clouds also store a large amount of measurement points and with each scan you need to allocate more memory, so they are not memory efficient.



#### 3D Voxel Grid

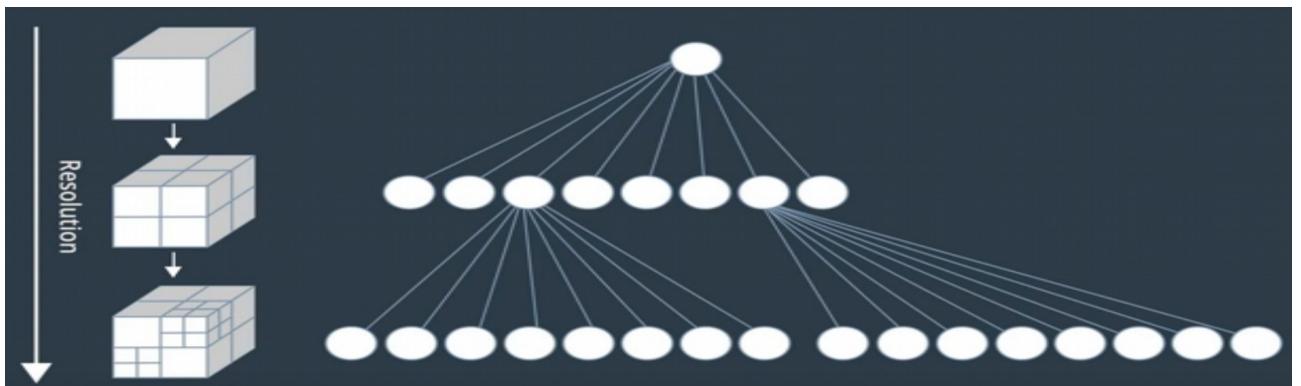
A 3D Voxel Grid is a volumetric representation using a grid of cubic volumes of equal size. This is a probabilistic representation, so you can estimate whether the voxel is occupied, free or unknown space. One drawback of a 3D voxel grid is that the size of the area must be known or approximated

before measurement,, which may not always be possible. A second drawback is that the complete map must be allocated in memory, so the overall memory requirement is high.



## Octrees

Octrees are a memory efficient tree based representation, on the left you will see the volumetric representation and on the right, the tree-based representation. These trees can be dynamically expanded to different resolutions and different areas, where every voxel can be subdivided into eight voxels recursively. The size of the map doesn't need to be known beforehand, because map volumes aren't initialized until you need to add new measurements. Octrees have been used to adapt occupancy grid mapping from 2D to 3D, introducing probabilistic representation of occupied versus freespace.



**2.5D maps**, also known as height maps, store the surface of the entire environment as the maximum height measured at every point. They are memory efficient, with constant access time. This type of mapping is not very useful if you have terrain with trees or overhang structures, where the robot could move underneath. Also, height maps are non-probabilistic. Similar to point clouds, there is also no distinction between free and unknown space.

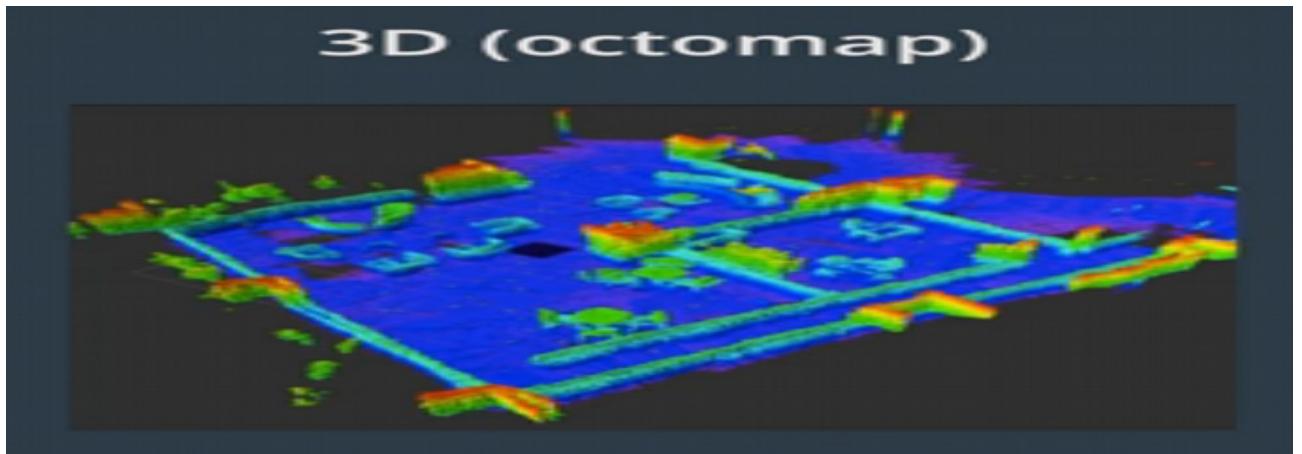
**Elevation maps** are 2D grids that store an estimated height, or elevation, for each cell. A Kalman filter is used to estimate the height, and can also incorporate the uncertainty of the measurement process itself, which typically increases with the measured distance. One problem with elevation maps is the vertical wall - you can tell there is a vertical object but don't know exactly how tall it is.

**Extended elevation maps** store a set of estimated heights for every cell, and include cells that contain gaps. You can check whether the variance of the height of all data points within each cell is large. If so, you can investigate whether the corresponding set of points contains a gap exceeding the height of the robot (known as a “gap cell”), and ultimately use gap cells to determine traversability.

In **multi-level surface (MLS)** map representations, each 2D cell stores “patches”, of which there can be multiple per cell. Each patch contains 3 key pieces of information - the height mean, the height variance, and the depth value. The height mean is the estimated height of the individual vertical area, also referred to as an interval. The uncertainty of the height is stored as the height variance, with the assumption that the error is represented by a Gaussian distribution. The depth value is defined by the difference between height of the surface patch and the height of the lowest measurement that is considered as belonging to that vertical object (ex the depth of the floor would be 0). Individual surfaces can be directly calculated, allowing the robot to deal with vertical and overhanging objects. This method also works very well with multi-level traversable surfaces, such as a bridge that you could travel over top of, or underneath, or a structure like a parking garage. An MLS map isn’t a volumetric representation, but a discretization in the vertical dimension. Unknown areas are not represented, and localization for this method is not straightforward.

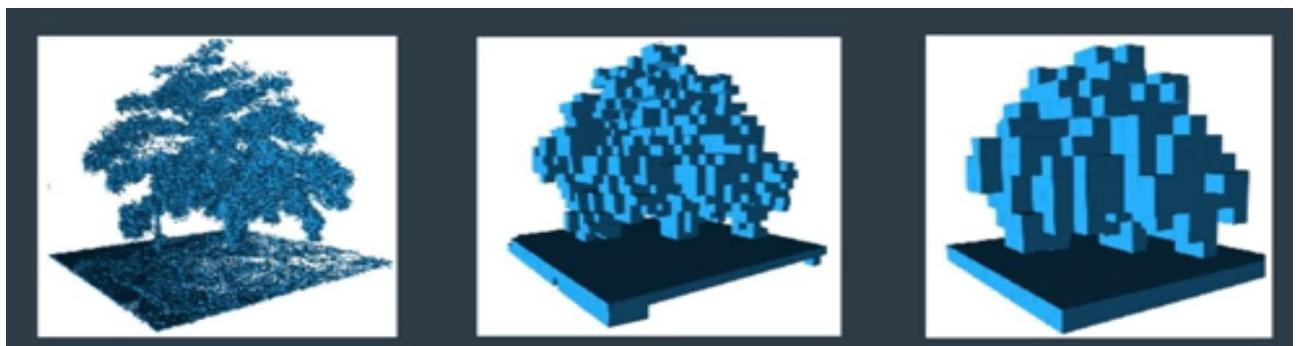
### 5.2.15 OctoMap framework

Now that we’ve learned about some 3D data representation, lets move on to an implementation.



The OctoMap framework is an open-source C++ library and ROS package based on Octrees, and it can be used to generate volumetric 3D models. OctoMap is not a 3D SLAM solution, it is the mapping framework and requires a pose estimate. It converts and integrates point clouds into 3D occupancy maps.

- OctoMap uses a probabilistic occupancy estimation modeled as a recursive binary Bayes filter. It is a static state filter, which assumes the environment doesn't change.
- Efficient updates are achieved using the log odds notation
- Occupancy is represented volumetrically with modelling of free, occupied and unmapped areas. Upper and lower bounds are placed on the log odds value of the occupancy estimate. This [p;icy limits the number of updates required to change the state of the voxel.



OctoMap supports multi-resolution map queries where the minimum voxel size determines the resolution.

- Tree pruning is also used to reduce redundant information between discrete occupancy states. Pruning is accomplished by defining a threshold probability that the voxel is occupied or free. Children that are identical to the parent in the tree can be pruned.
- Memory efficient representation is accomplished using a compression method that produces compact map files. Coherent map volumes are locally combined, including both mapped free areas and occupied space.

### 5.3.16 Outro

So far, we have assumed exact robot poses and learn how a static environment can be mapped with the occupancy grid mapping algorithm, this algorithm decompose the world into grid cells and implement a binary Bayes filter to estimate the occupancy of each cell individually. We've also learned how to fuse measurements from different sensors using De Morgan's Law to build a single map. The mapping with known poses problem is a challenging one, but unfortunately, by solving this problem you are still unable to generate maps in the real world. That's because both the poses and the map will be unknown to you. So, get ready to learn how to map real-world environments by solving a much more challenging problem in the upcoming chapters.

# 5.3 Grid-based FastSLAM

## 5.3.1 Introduction

In this chapter we are going to tackle a much more challenging problem of estimating both the map and the robot poses in real-world environments. This problem is called Simultaneous Localization and Mapping or SLAM. Some roboticist referre to it as CLAM or Concurrent Localization and Mapping.

While solving localization problems you assume a known map and estimate the robot poses, whereas in solving mapping problems, you are provided with the exact robot poses and estimated the map of the environment.

Now we will combine our knowledge from both localization and mapping to solve the most fundamental problem in robotics.

In SLAM, we will map the environment, giving the noisy measurements and localize the robot relative to its own map giving the controls. This makes it much more challenging problem than localization or mapping since both the map and the poses are now unknown to us.

In real-world environments, we will primarily be faced with SLAM problems and we will aim to estimate the map and localize the robot.

An example of a robot solving its SLAM problem is the robotic vacuum cleaner that uses the measurements provided by its laser finder sensors and data from the encoders to estimate the map and localize itself relative to it.

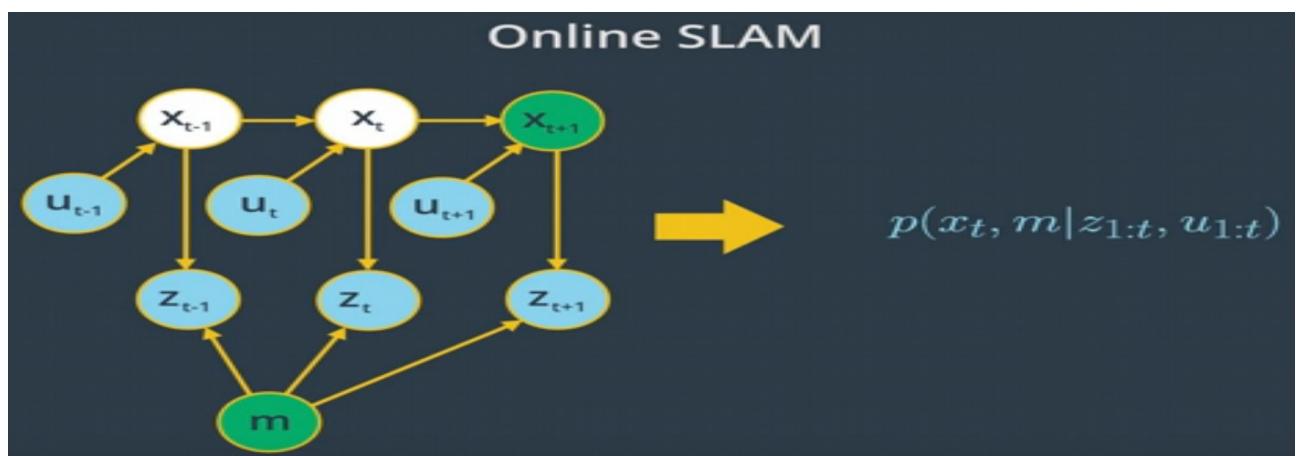
### What are the inputs and outputs to the SLAM problem?

**Input: Measurements+Controls**

**Output: Map+Trajectory**

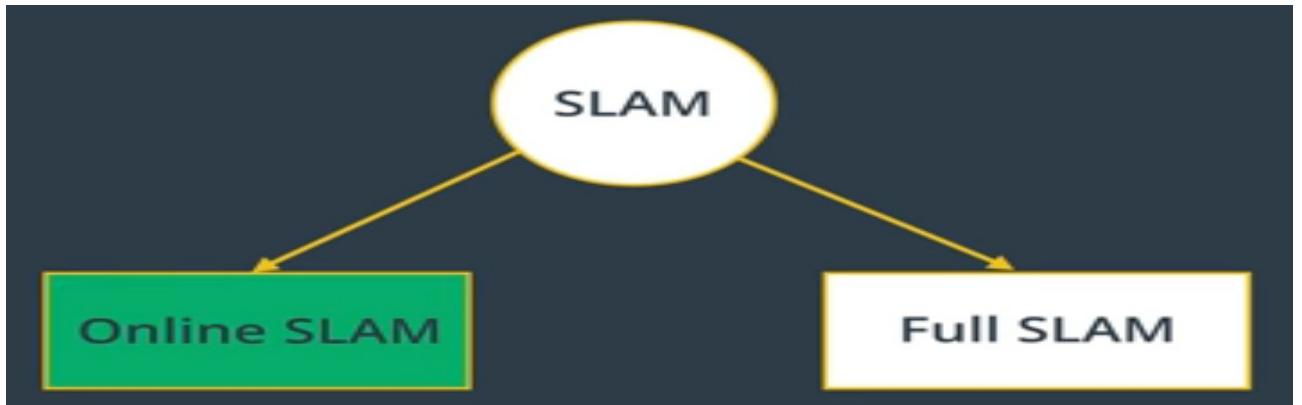
## 5.3.2 Online SLAM

Here's the graphical representation of the SLAM problem.



Giving the measurements Z and the controls U, we will learn how to solve the posterior map m and along with the poses X.

The SLAM problem comes in two different forms, the Online SLAM and the Full SLAM.



Since we will encounter SLAM algorithms that solve either the Online SLAM problem or the Full SLAM problem, some algorithms will even solve both problems.

Let's start with the Online SLAM problem.

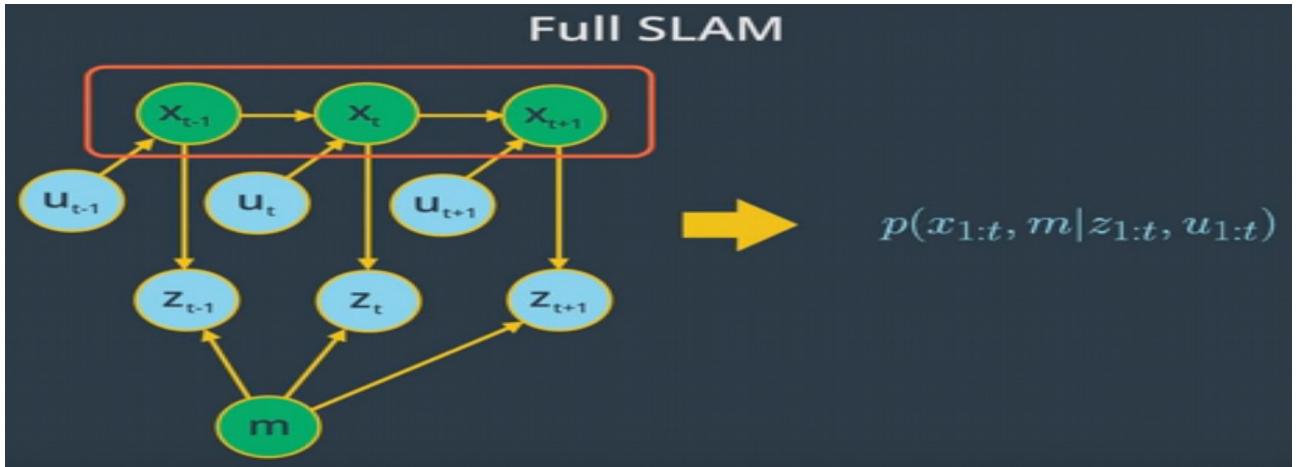
### Online SLAM Problem

- At time  $t-1$ , the robot will estimate its current pose  $\mathbf{x}_{t-1}$  and the map  $\mathbf{m}$  given its current measurements  $\mathbf{z}_{t-1}$  and controls  $\mathbf{u}_{t-1}$ .
- At time  $t$ , the robot will estimate its new pose  $\mathbf{x}_t$  and the map  $\mathbf{m}$  given only its current measurements  $\mathbf{z}_t$  and controls  $\mathbf{u}_t$ .
- At time  $t+1$ , the robot will estimate its current pose  $\mathbf{x}_{t+1}$  and the map  $\mathbf{m}$  given the measurements  $\mathbf{z}_{t+1}$  and controls  $\mathbf{u}_{t+1}$ .

This problem can be modeled with the probability equation  $p(\mathbf{x}_t, \mathbf{m} | \mathbf{z}_{1:t}, \mathbf{u}_{1:t})$  where we solve the posterior represented by the instantaneous pose  $\mathbf{x}_t$  and the map  $\mathbf{m}$  given the measurements  $\mathbf{z}_{1:t}$  and controls  $\mathbf{u}_{1:t}$ . Thus, with online SLAM we estimate variables that occur at time  $t$  only.

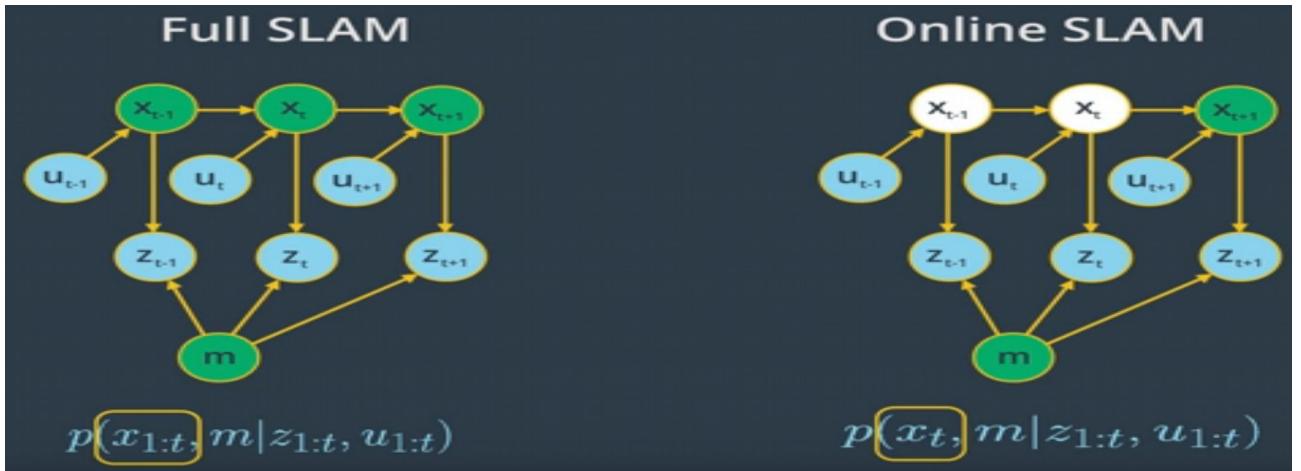
### 5.3.3 Full SLAM

So far, we've covered the Online SLAM problem, now it's time to dive deeper in the Full SLAM problem also known as off-line SLAM. While solving the Full SLAM problem we will estimate the entire path up to time  $t$  instead of an instantaneous pose given all the measurements and controls. Now let's model this graphical representation of the full SLAM problem in a probability function. Here's the result.



This function estimates the posterior over the entire path along the map given all the measurements and controls up to time t.

Now that we've seen both the Online and Full SLAM problem let's compare them, and see if a relation exists between them.



Clearly, the difference between the two forms of SLAM is in their poses. One represents a posterior over the entire path, while the other represents a posterior over the current pose.

Now, is there any relation between these two forms of SLAM?

Yes, a relation between the Online SLAM problem, and the Full SLAM exists by integrating the previous poses from the Full SLAM problem once at a time, we obtain the Online SLAM problem.

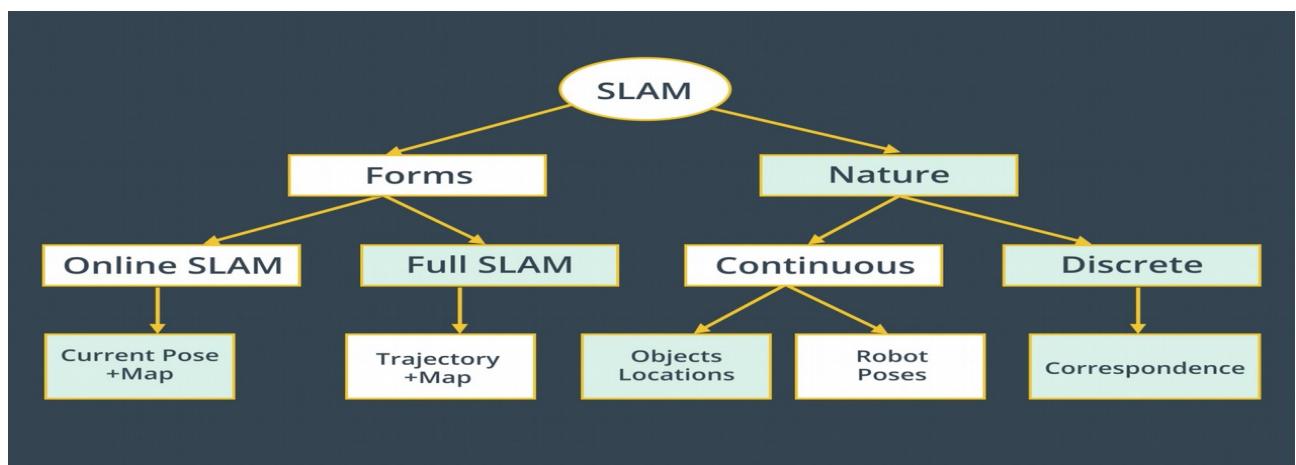
$$p(x_t, m | z_{1:t}, u_{1:t}) = \underbrace{\int \int \dots \int}_{\text{Online Slam}} \underbrace{p(x_{1:t}, m | z_{1:t}, u_{1:t}) dx_1 dx_2 \dots dx_{t-1}}_{\text{Full Slam}}$$

## Full SLAM Problem

- At time  $t-1$ , the robot will estimate the robot pose  $\mathbf{x}_{t-1}$  and map  $\mathbf{m}$ , given the measurements  $\mathbf{z}_{t-1}$  and controls  $\mathbf{u}_{t-1}$ .
- At time  $t$ , the robot will estimate the entire path  $\mathbf{x}_{t-1:t}$  and map  $\mathbf{m}$ , given all the measurements  $\mathbf{z}_{t-1:t}$  and controls  $\mathbf{u}_{t-1:t}$ .
- At time  $t+1$ , the robot will estimate the entire path  $\mathbf{x}_{t-1:t+1}$  and map  $\mathbf{m}$ , given all the measurements  $\mathbf{z}_{t-1:t+1}$  and controls  $\mathbf{u}_{t-1:t+1}$ .

This problem can be modeled with the probability equation  $p(\mathbf{x}_{1:t}, \mathbf{m} | \mathbf{z}_{1:t}, \mathbf{u}_{1:t})$ , where we solve the posterior represented by the robot's trajectory  $\mathbf{x}_{1:t}$  and the map  $\mathbf{m}$  given all the measurements  $\mathbf{z}_{1:t}$  and controls  $\mathbf{u}_{1:t}$ . Thus, with full SLAM problem we estimate all the variables that occur throughout the robot travel time.

### 5.3.4 Nature of SLAM



#### Forms

You've just learned the first key feature of the SLAM problem which has to do with its two forms. The online SLAM problem computes a posterior over the current pose along with the map and the full SLAM problem computes a posterior over the entire path along with the map.

#### Nature

Now, the second key feature of the SLAM problem relates to its nature. SLAM problems generally have a continuous and a discrete element.

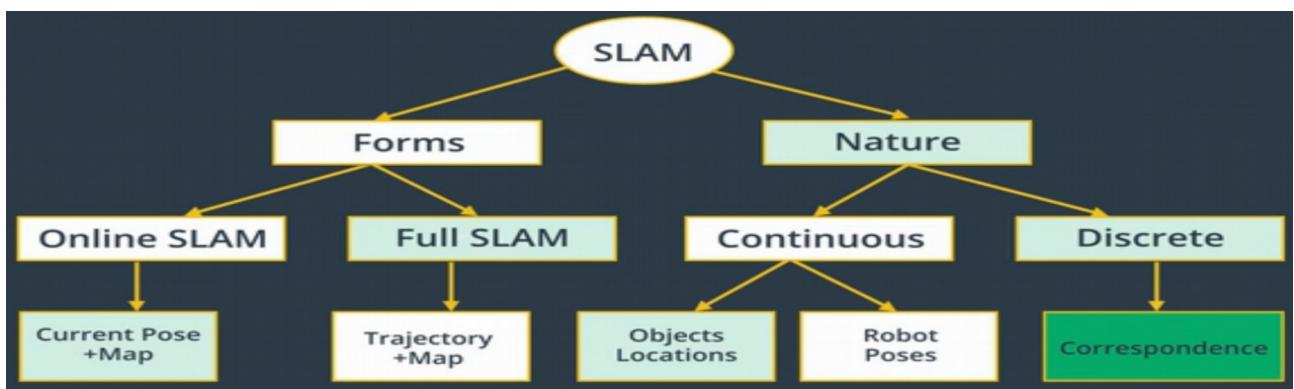
#### Nature - Continuous

Let's start with the continuous component of the SLAM problem. During SLAM, a robot continuously collects odometry information to estimate the robot poses and continuously senses the environment to estimate the location of the object or landmark. Thus, both robots poses and object location are continuous aspects of the SLAM problem.

## Nature - Discrete

Now, moving to the second component of the SLAM problem. As I mentioned earlier, robots continuously sense the environment to estimate the location of the objects, when doing so SLAM algorithms have to identify if a relation exists between any newly detected objects and previously detected ones. This helps the robot understand if it has been in this same location before. At each moment, the robot has to answer the question, "Have I been here before?". The answer to this question is binary - either yes or no - and that's what makes the relation between objects a discrete component of the SLAM problem. This discrete relation between objects is known by correspondence.

### 5.3.5 Correspondence

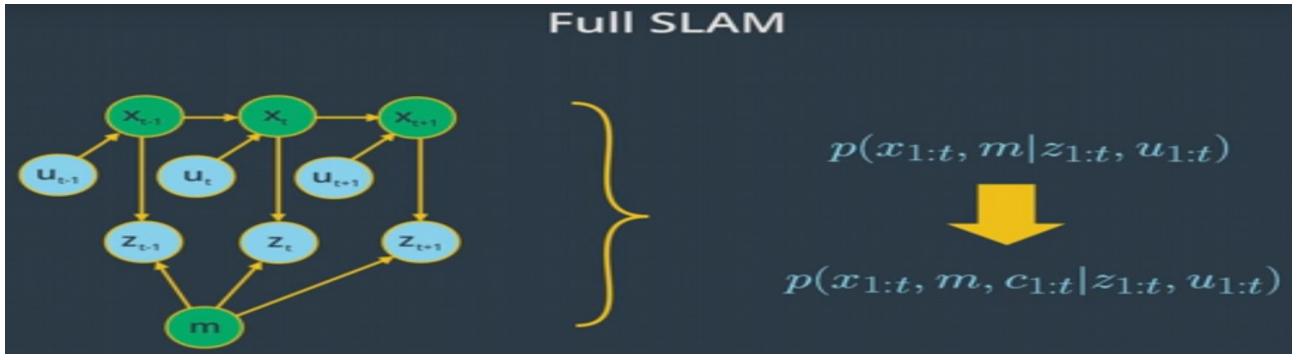


While studying the nature of the SLAM problem, we concluded that SLAM is a continuous discrete estimation problem. The discrete component of the SLAM problem has to do with correspondence and since it's an important aspect, we should explicitly include it in our estimation problem. Meaning that the posterior should now include the correspondence values in both the Online and Full SLAM problems.

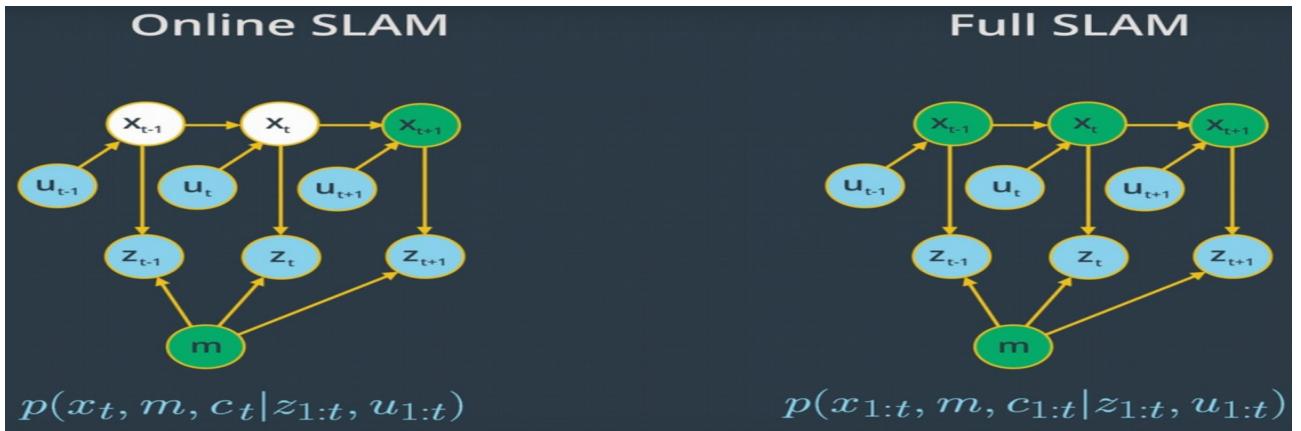
When solving the Online SLAM problem we will now aim to estimate the posterior over the current pose, map and the current correspondence value  $c_t$  giving the measurements and controls up to time  $t$ .



On the other hand, while solving the full SLAM problem, we will now aim to estimate a posterior over the entire path, map and all of the correspondence values  $c_1$  up to time  $t$  giving all the measurements and controls up to time  $t$ .



Now that we've added the correspondence values to both the posterior of the online and full SLAM problem, our ultimate goal of estimating the map and the robot pose or trajectory remains the same.



**The advantage of adding correspondence values to the posterior is to have the robot better understand where it is located by establishing a relation between objects.**

But what about the relationship between the Online and Full SLAM problem, now that we've added the correspondence values, is it still the same?

The online SLAM problem is now the result of integrating the robot poses once at a time and summing over each previous correspondence value.

$$p(x_t, m, c_t | z_{1:t}, u_{1:t}) = \underbrace{\int \int \dots \int}_{\text{Online SLAM}} \sum_{c_1} \sum_{c_2} \dots \sum_{c_{t-1}} \underbrace{p(x_{1:t}, m, c_{1:t} | z_{1:t}, u_{1:t}) dx_1 dx_2 \dots dx_{t-1}}_{\text{Full SLAM}}$$

Now that we've learned the two **key features** of the SLAM problem, let's summarize them:

## 1-Forms

- **Online SLAM:** Robot estimates its current pose and the map using current measurements and controls.
- **Full SLAM:** Robot estimates its entire trajectory and the map using all the measurements and controls.

## 2- Nature

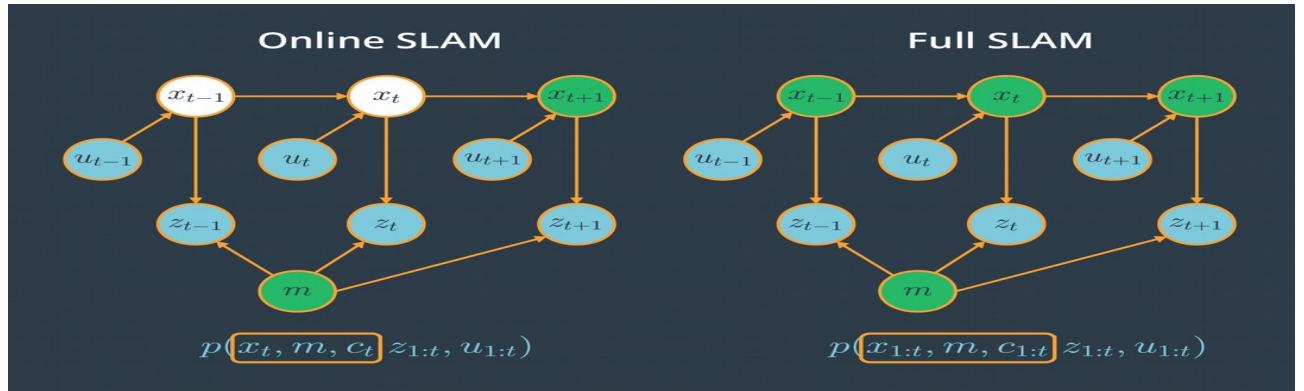
- **Continuous:** Robot continuously senses its pose and the location of the objects.

- **Discrete:** Robot has to identify if a relation exists between any newly detected and previously detected objects.

For the **Full SLAM** problem, what's the posterior under **unknown correspondences**?

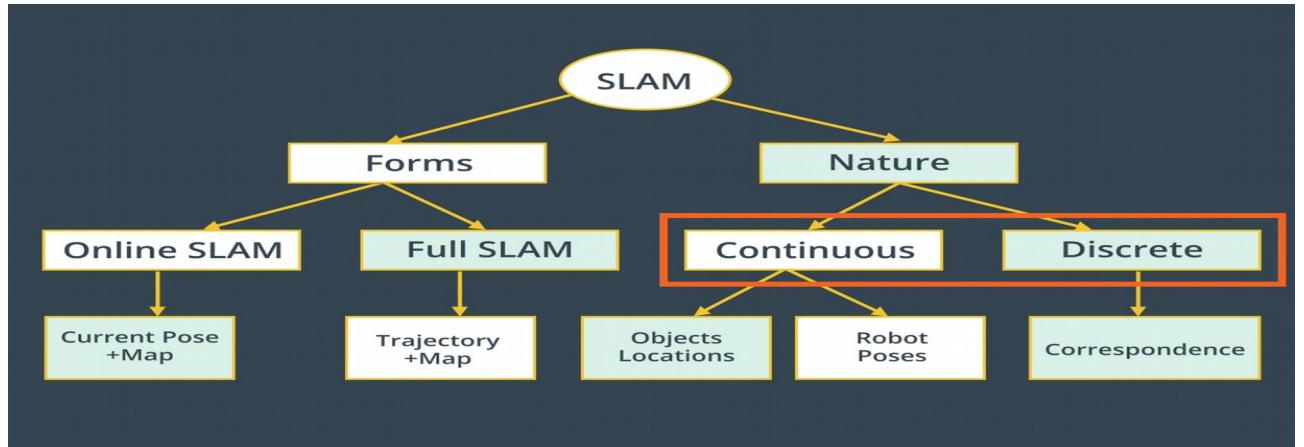
Answer:  $x_{1:t}, m, c_{1:t}$

### 5.3.6 SLAM Challenges



### SLAM Challenges

Computing the full posterior composed of the robot pose, the map and the correspondence under SLAM poses a big challenge in robotics mainly due to the **continuous** and **discrete** portion.



### Continuous

The continuous parameter space composed of the robot poses and the location of the objects is highly dimensional. While mapping the environment and localizing itself, the robot will encounter many objects and have to keep track of each one of them. Thus, the number of variables will increase with time, and this makes the problem highly dimensional and challenging to compute the posterior.

### Discrete

Next, the discrete parameter space is composed out of the correspondence values, and is also highly dimensional due to the large number of correspondence variables. Not only that, the correspondence

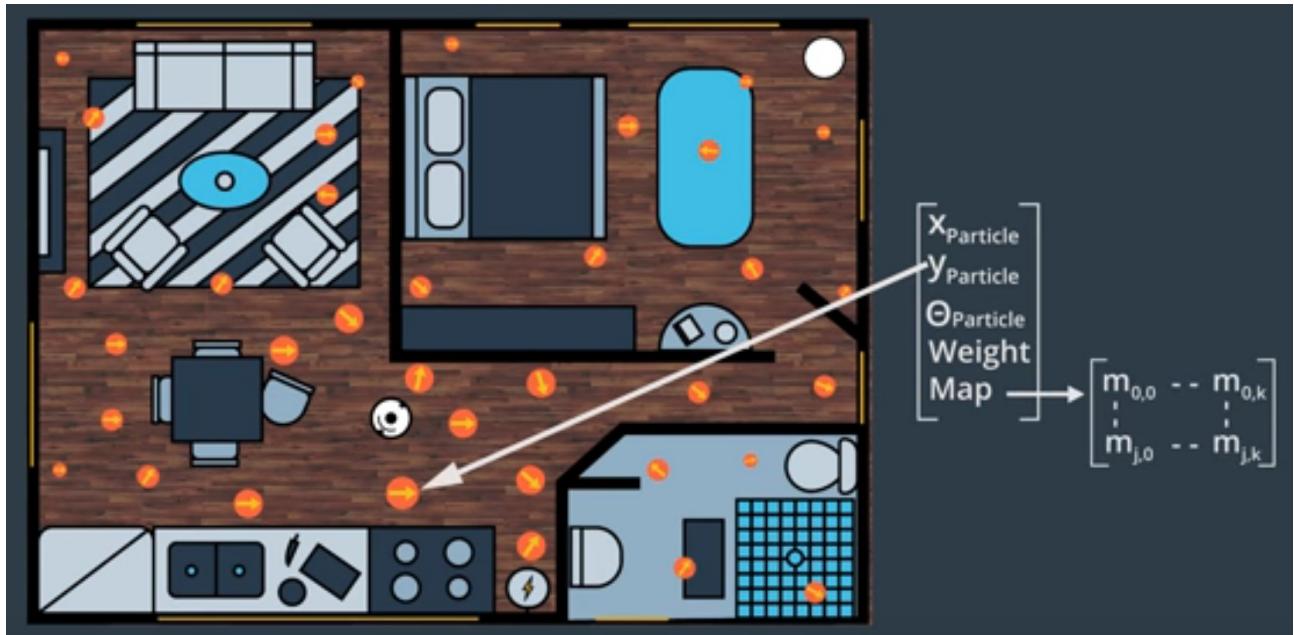
values increase exponentially over time since the robot will keep sensing the environment and relating the newly detected objects to the previously detected ones. Even if you assume known correspondence values, the posterior over maps is still highly dimensional as we saw in the mapping lessons.

You can now see why it's infeasible to compute the posterior under unknown correspondence. Thus, SLAM algorithms will have to rely on approximation while estimating a posterior in order to conserve computational memory.

### 5.3.7 Particle Filter Approach to SLAM

So far, we've learned the key features and challenges of the SLAM problem. Now, we will learn how to overcome these challenges and solve the online and full SLAM problems.

But before we dive into any state-of-the-art SLAM algorithm, let's recap what we have learned from Localization. While solving the localization problem we used different approaches to estimate a robot's pose inside an environment. One of these approaches was the particle filter approach implemented in the Monte Carlo Localization algorithm.



Each of these particles held the robot pose along with the importance weight of the particle. Using this approach, we were able to accurately estimate any robot's pose.

Now let's imagine that we modify the particle filter approach by adding another dimension to each particle, the new particle will now hold the robot's pose, as well as the map.

Will that solve the SLAM problem? Not really. This approach will fail and that's because the map is modeled with many variables resulting in high dimensionality. Thus, the particle filter approach to SLAM in this current form will scale exponentially and be doomed to fail.

### 5.3.8 Introduction to FastSLAM

The fastSLAM algorithm, uses a costum particle filter approach to solve the full SLAM problem with known correspondences. Using particles, fastSLAM estimates a posterior over the robot path along with the map. Each of these particles holds the robot trajectory which will give an advantage to SLAM to solve the problem of mapping with known poses. In addition to the robot trajectory, each particle holds a map and each feature of the map is represented by a local Gaussian.

With the fastSLAM algorithm, the problem is now divided into seperate independent problems. Each of which aims to solve the problem of estimating features of the map.

To solve these independent mini problems, fastSLAM will use a low dimensional extended Kalman Filter.

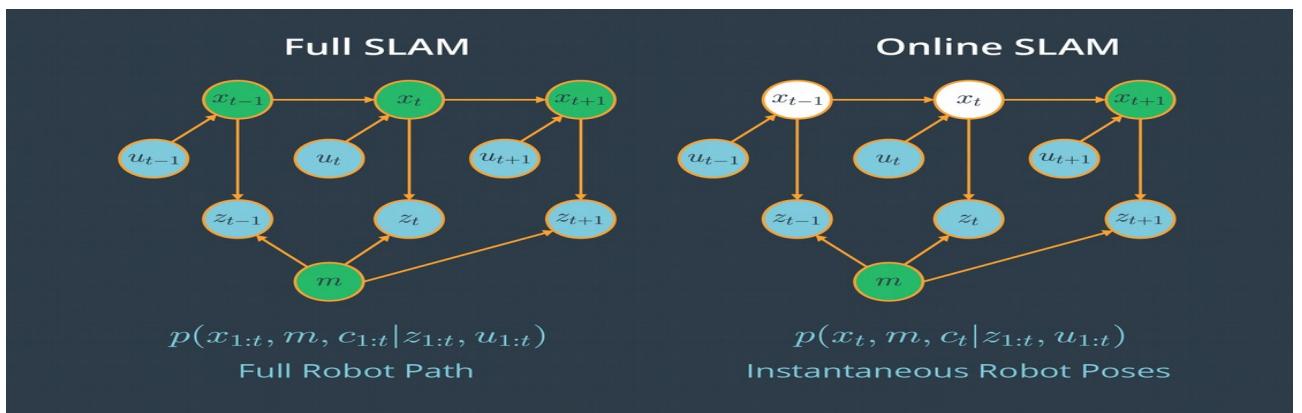
While map features are treated independently, dependency only exists between robot pose uncertainty.

The **FastSLAM** algorithm solves the Full SLAM problem with known correspondences.

- **Estimating the Trajectory:** FastSLAM estimates a posterior over the trajectory using a particle filter approach. This will give an advantage to SLAM to solve the problem of mapping with known poses.
- **Estimating the Map:** FastSLAM uses a low dimensional Extended Kalman Filter to solve independent features of the map which are modeled with local Gaussian.

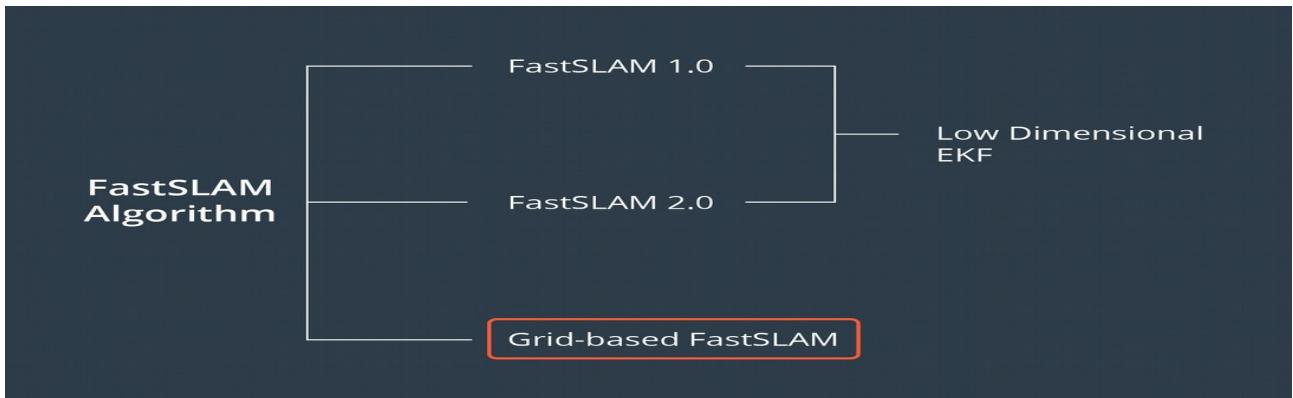
These custom approach of representing the posterior with particle filter and Gaussian is known by the Rao-Blackwellized Particle Filter One.

### 5.3.9 FastSLAM Instances



We've seen that the FastSLAM algorithm can solve the full SLAM problem with known correspondences. Since FastSLAM uses a particle filter approach to solve SLAM problems, some roboticists consider it a powerful algorithm capable of solving both the **Full SLAM** and **Online SLAM** problems.

- FastSLAM estimates the full robot path, and hence it solves the **Full SLAM** problem.
- On the other hand, each particle in FastSLAM estimates instantaneous poses, and thus FastSLAM also solves the **Online SLAM** problem.



Now, three different instances of the FastSLAM algorithm exist.

### **FastSLAM 1.0**

The FastSLAM 1.0 algorithm is simple and easy to implement, but this algorithm is known to be inefficient since particle filters generate sample inefficiency.

### **FastSLAM 2.0**

The FastSLAM 2.0 algorithm overcomes the inefficiency of FastSLAM 1.0 by imposing a different distribution, which results in a low number of particles. Keep in mind that both of the FastSLAM 1.0 and 2.0 algorithms use a low dimensional Extended Kalman filter to estimate the posterior over the map features.

### **Grid-based FastSLAM**

The third instance of FastSLAM is really an extension to FastSLAM known as the grid-based FastSLAM algorithm, which adapts FastSLAM to grid maps. In this chapter, we will learn grid-based FastSLAM.

## **5.3.10 Adapting FastSLAM to Grid Maps**



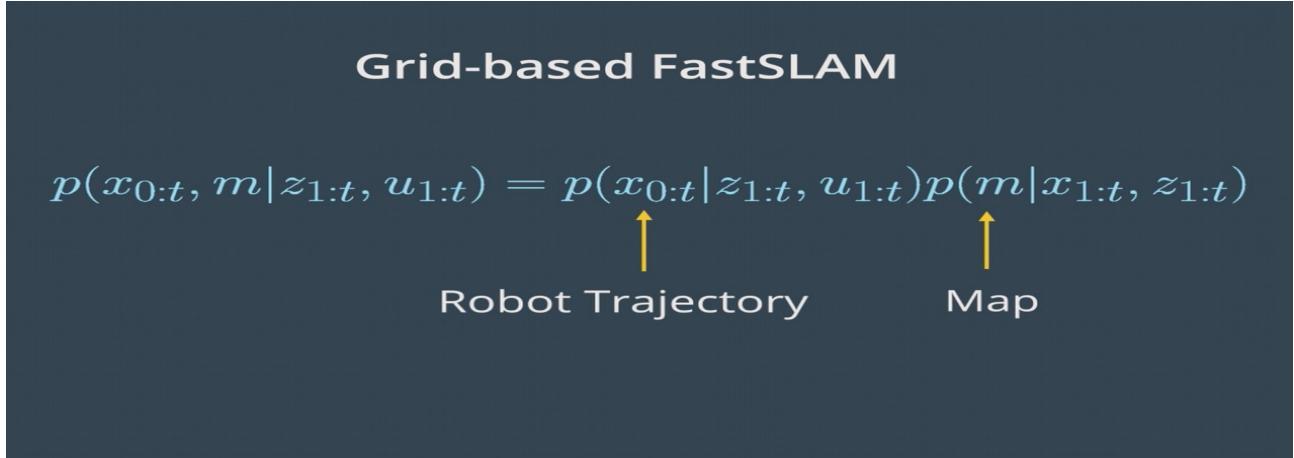
### **FastSLAM 1.0 & 2.0**

The main advantage of the FastSLAM algorithm is that it uses a particle filter approach to solve the SLAM problem. Each particle will hold a guess of the robot trajectory, and by doing so, the SLAM problem is reduced to mapping with known poses. But, in fact, this algorithm presents a **big disadvantage since it must always assume that there are known landmark positions**, and thus

with FastSLAM we are not able to model an arbitrary environment. Now, what if landmark positions are unavailable to us? Are we still able to solve the SLAM problem?

### Grid-based FastSLAM

Yes, with the grid mapping algorithm you can model the environment **using grid maps without predefining any landmark position**. So by extending the FastSLAM algorithm to occupancy grid maps, you can now solve the SLAM problem in an arbitrary environment. While mapping a real-world environment, **you will mostly be using mobile robots equipped with range sensors**. We'll then extend the FastSLAM algorithm and solve the SLAM problem in term of grid maps.



### Robot Trajectory

Just as in the FastSLAM algorithm, with the grid-based FastSLAM each particle holds a guess of the robot trajectory.

### Map

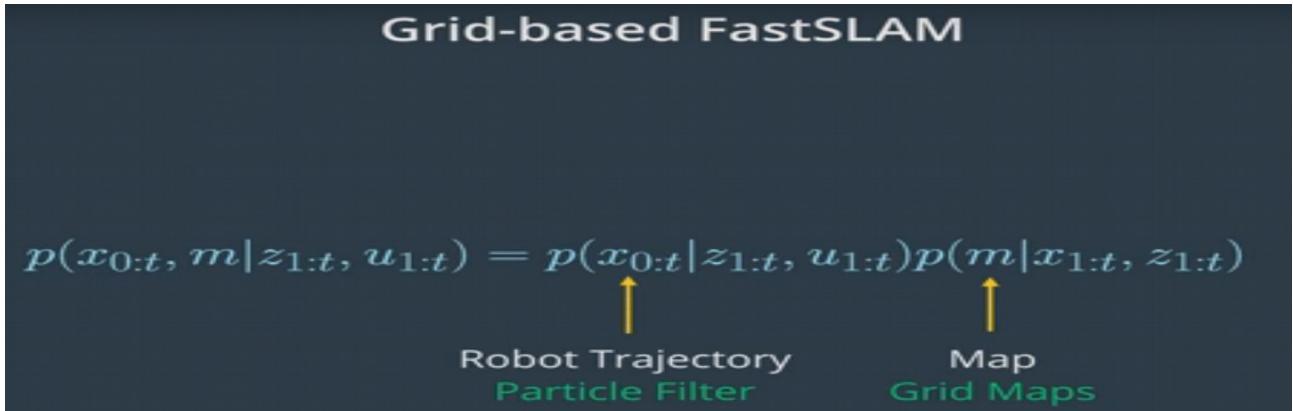
In addition, each particle maintains its own map. The grid-based FastSLAM algorithm will update each particle by solving the mapping with known poses problem using the occupancy grid mapping algorithm.

The Grid-based FastSLAM algorithm uses the following algorithms

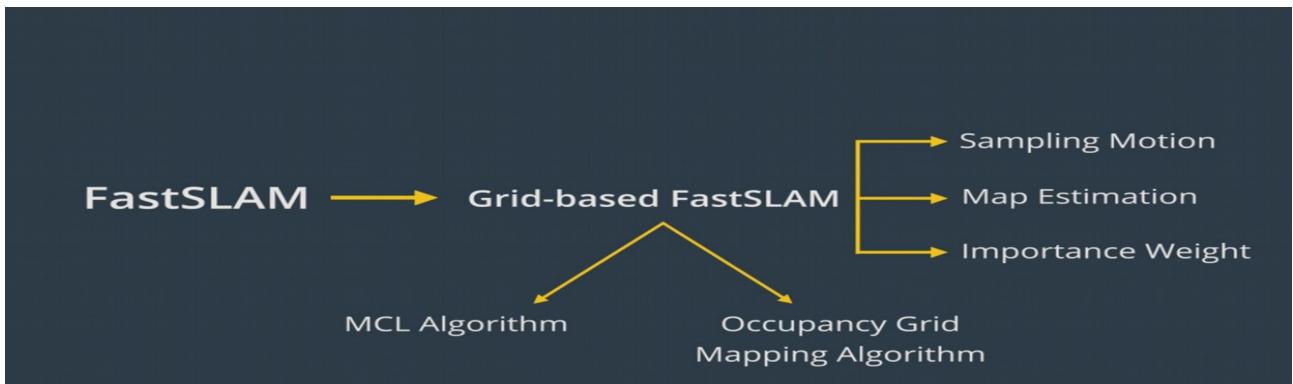
- MCL
- Occupancy Grid Mapping
- **Not a** Low-Dimensional EKF like FastSLAM 1 & 2

### 5.3.11 Grid-based FastSLAM Techniques

Adapting the FastSLAM algorithm to grid maps is resulted in the grid-based FastSLAM algorithm. Since the grid-based Fast SLAM algorithm uses a particle filter approach and represents the world in terms of grid maps.



We will need to combine what we learned from MCL and the Occupancy Grid Mapping algorithm.

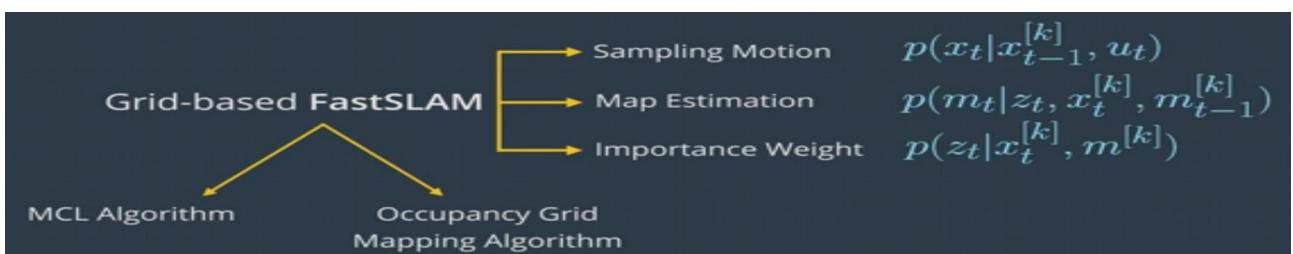


To adapt FastSLAM to grid mapping, we need three different techniques:

- Sampling Motion**- $p(x_t | x_{t-1}[k], u_t)$ : Estimates the current pose given the k-th particle previous pose and the current controls  $u$ , with the MCL algorithm.
- Map Estimation**- $p(m_t | z_t, x_t[k], m_{t-1}[k])$ : Estimates the current map given the current measurements, the current k-th particle pose, and the previous k-th particle map, with the Occupancy Grid Mapping algorithm.
- Importance Weight**- $p(z_t | x_t[k], m[k])$ : Estimates the current likelihood of the measurement given the current k-th particle pose and the current k-th particle map, with the MCL algorithm.

### 5.3.12 The Grid-based FastSLAM Algorithm

The sampling motion, map estimation and Importance weight techniques are the essence of the Grid-based FastSLAM algorithm.



It implements them to estimate both the map and the robot trajectory giving the measurements and controls.

Here's the Grid-based FastSLAM algorithm which looks very similar to the MCL algorithm with some additional statements concerning the map estimation.

## Algorithm Grid-based FastSLAM( $X_{t-1}, u_t, z_t$ )

```

 $\bar{X}_t = X_t = \emptyset$ 
for k=1 to M do
     $x_t^{[k]} = \text{sample\_motion\_model } (u_t, x_{t-1}^{[k]})$ 
     $w_t^{[k]} = \text{measurement\_model\_map } (z_t, x_t^{[k]}, m_{t-1}^{[k]})$ 
     $m_t^{[k]} = \text{updated\_occupancy\_grid } (z_t, x_t^{[k]}, m_{t-1}^{[k]})$ 
     $\bar{X}_t = \bar{X}_t + \langle x_t^{[k]}, m_t^{[k]}, w_t^{[k]} \rangle$ 
endfor
for k=1 to M do
    draw i with probability  $\propto w_t^{[i]}$ 
    add  $\langle x_t^{[i]}, m_t^{[i]} \rangle$  to  $X_t$ 
endfor
return  $X_t$ 
```

In fact, this algorithm is the result of combining the MCL algorithm and the Occupancy Grid Mapping one. As in the MCL algorithm the Grid-based FastSLAM algorithm is composed of two sections represented by two for loops.

The first section includes the motion, sensor and map updates steps, and the second one includes the re-sampling process

- **Step 1 : Previous Belief**
- **Step 2 : Sampling Motion**
- **Step 3 : Importance Weight**
- **Step 4 : Map Estimation**
- **Step 5 : Resampling**
- **Step 6 : New Belief**

### 5.3.13 Outro

In this chapter, we've learned the two essential key features of the SLAM problem.

The first feature lies in its two forms which are the Online SLAM and the FullSLAM problem.

The second feature to the SLAM problem is related to its continuous and discrete nature.

Then, we were exposed to the challenges of SLAM, the continuous nature of the SLAM problem poses a big challenge because of the high dimension of posterior, in addition, the discrete nature of

SLAM problem poses another challenge because of the large number of correspondence values that exists.

Moving on, we were introduced to the FastSLAM algorithm which uses a particle filter approach along with a low-dimensional extended Kalman Filter estimation to solve the SLAM problem.

We've also learned how to overcome some drawbacks to the FastSLAM algorithm by adapting it to grid maps which resulted in the grid based FastSLAM algorithm.

We saw how the Grid-based Fast SLAM algorithm implements parts of the MCL algorithm along with the occupancy grid mapping algorithm.

**And finally the gmapping ROS package uses the Grid-based FastSLAM algorithm to map the environment and estimate the robot's path.**

**See the appendix on how to use this package and the experimental results.**

## 5.4 GraphSLAM

### 5.4.1 Introduction

In this chapter will examine another flavor of SLAM called GraphSLAM.

GraphSLAM is a slam algorithm that solves the full slam problem, this means that the algorithm recovers the entire path and map, instead of just the most recent pose and map. This difference allows it to consider dependencies between current and previous poses.

One example where GraphSLAM would be applicable is in underground mining. Large machines called bores, spent every day cutting away at the rockface, the environment changes rapidly, and it's important to keep an accurate map of the workspace. One way to map the space would be to drive a vehicle with a LIDAR around the environment that collects data about the surroundings, then after the fact, the data can be analyzed to create an accurate map of the environment.

As we learn about GraphSLAM, we'll understand some of the benefits of the algorithm, including the reduced need for significant onboard processing capability.

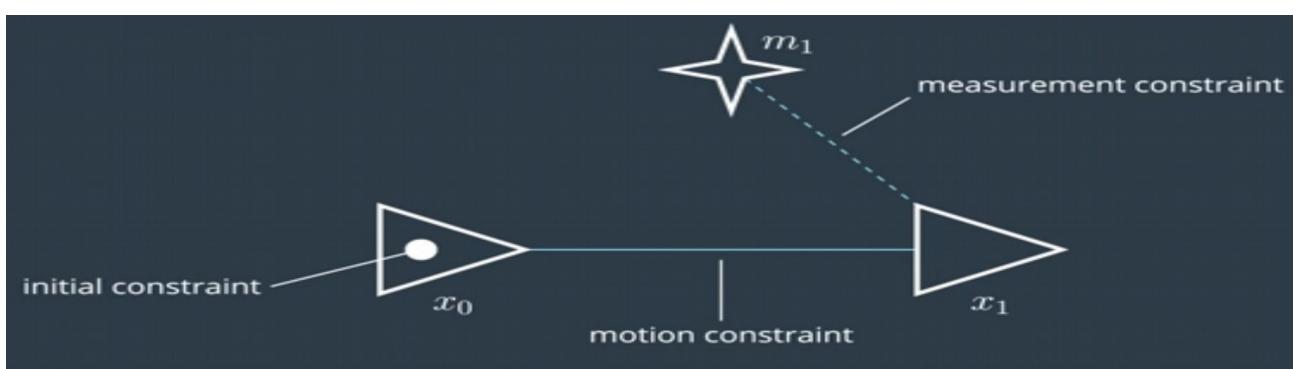
One advantage that can be immediately appreciated is GraphSLAM's improved accuracy over FastSLAM. FastSLAM uses particles to estimate the robot's most likely pose, however at any point in time, it's possible that there isn't a particle in the most likely location. In fact, the chances are slim to none especially, in large environments.

Since GraphSLAM solves the full slam problem, this means that it can work with all of the data at once to find the optimal solution. FastSLAM uses tidbits of information with a finite number of particles, so there is room for error.

In this chapter we will learn the fundamentals of GraphSLAM, how to construct the graphs, how to optimize within their constraints to create the most likely set of poses and map and how to overcome some of GraphSLAM limitations.

### 5.4.2 Graphs

GraphSLAM has graph in it's name because it uses graphs to represent the robot's poses and the environment. Let's build a very simple graph to better understand how that works.



A robot pose, it's position and orientation can be represented by a node like so. This is it's pose at time step 0. Usually, the first node is arbitrary constrained to (0,0) or it's equivalent in higher dimensions.

The robot pose at timestep 1 can be represented by another node and the two would be connected by an edge, sometimes called an arc.

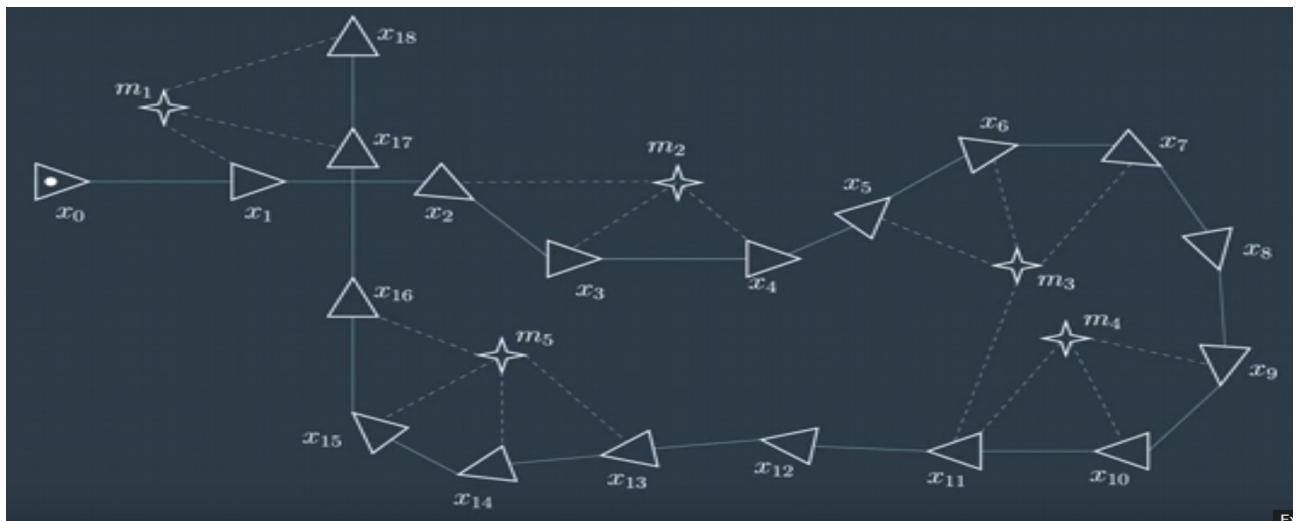
This edge is a soft spatial constraint between the two robot poses, these constraints are called soft because, as we have seen before, motion and measurement data are uncertain. The constraints will have some amount of error present.

Soft constraints come in two forms, motion constraints between two successive robot poses and measurement constraints between a robot pose and a feature in the environment. The constraint seen between X0 and X1 is a motion constraint.

If the robot were to sense it's environment and encounter a feature m1, a soft measurement constraint would be added. Here, the star represents a feature in the environment. This could be a landmark specifically placed for the purpose of localization and mapping, or an identifiable element of the environment such as a corner or an edge.

Note that the notation that I have chosen to use has solid edges representing motion constraints and dashed edges representing measurement constraints. I find that this notation makes the graph easier to comprehend.

As the robot moves around more and more nodes are added to the graph and over time the graph constructed by a mobile robot becomes very large in size. Luckily, GraphSLAM is able to handle large number of features.



## Summary of Notation

- **Poses** are represented with triangles.
- **Features** from the environment are represented with stars.
- **Motion constraints** tie together two poses, and are represented by a solid line.
- **Measurement constraints** tie together a feature and a pose, and are represented by a dashed line.

### 5.4.3 Constraints

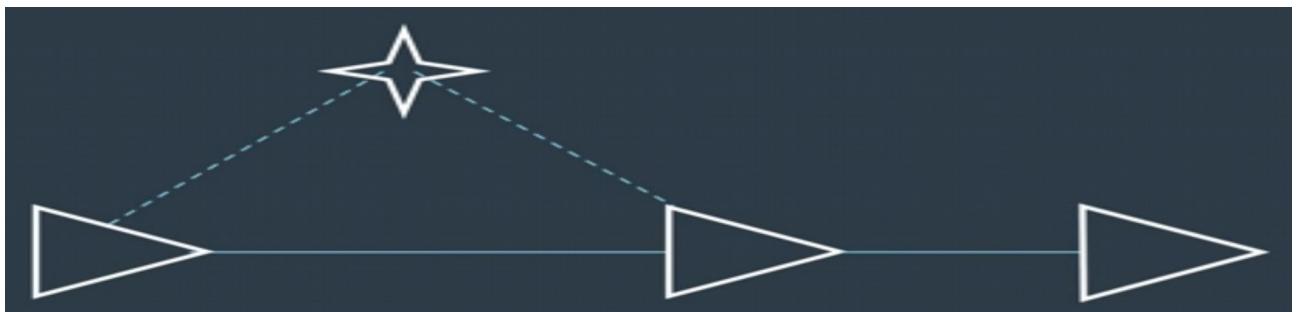
One way to look at soft constraints is to interpret them as masses connected by rubber bands or springs.



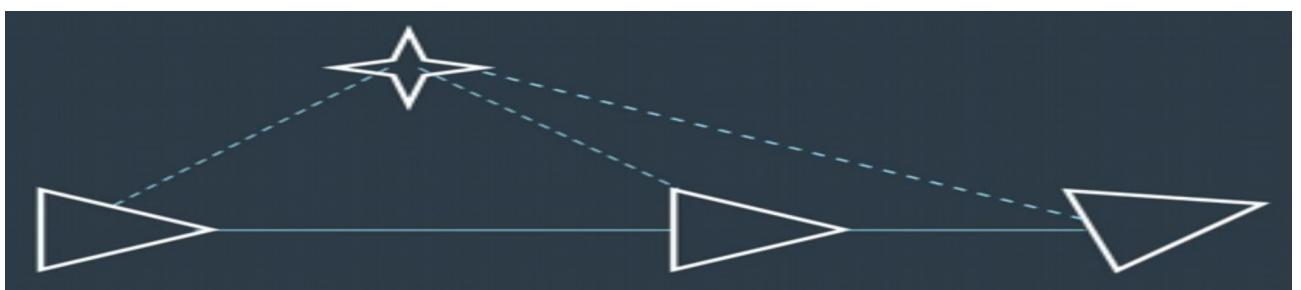
When no external forces are acting on them, the springs will naturally bring the system into a configuration, where the forces experienced by all of the springs are minimized. When the nodes are connected by a linear function as so, the resting configuration is easy to find.



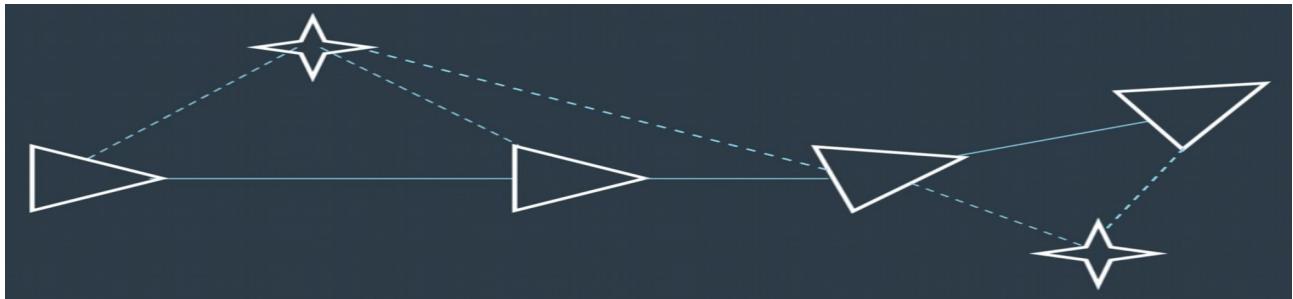
But the process becomes more challenging as nodes become more and more interconnected. The springs will all try to push or pull the system in their own ways. This is very similar to how constraints work in a graph.



Every motion or measurement constraint pulls the system closer to the constraint desired state.



Since the measurements and motions are uncertain, constraints will conflict with each other and there will always be some error present. In the end, the goal is to find the node configuration that minimizes the overall error present in all the constraints.



#### 5.4.4 Front-End vs Back-End

The goal of GraphSLAM is to create a graph of all robot poses and features encountered in the environment and find the most likely robot's path and map of the environment. This task, can be broken up into two sections, the front-end and the back-end.

The front-end of GraphSLAM looks at how to construct the graph using the odometry and sensory measurements collected by the robot.

##### Front-End

**The front-end of GraphSLAM looks at how to construct the graph using the odometry and sensory measurements collected by the robot**

This includes interpreting sensory data, creating the graph, and continuing to add nodes and edges to it as the robot traverses the environment. Naturally, the front-end can differ greatly from application to application depending on the desired goal, including accuracy, the sensor used, and other factors.

##### Front-End



For instance, the front-end of a mobile robot applying SLAM in the office using a laser range finder would differ greatly from the front-end of a vehicle operating on a large outdoor environment which uses a stereo camera.

The front-end of GraphSLAM also has the challenge of solving the data association problem. In simpler terms, this means accurately identifying whether features in the environment have been previously seen. We will explore this on greater detail later.

The back-end of GraphSLAM is where the magic happens

## Back-End Graph Optimization

The input to the back-end is the completed graph with all of the constraints, and the output is the most probable configuration of robot poses and map features. The back-end is an optimization process that takes all of the constraints and finds the system configuration that produces the smallest error. The back-end is a lot more consistent across applications. The front-end and the back-end can be completed in succession or can be performed iteratively with the back-end feeding an updated graph to the front-end for further processing.

### 5.4.5 Maximum Likelihood Estimation

#### Maximum Likelihood Estimation

At the core of GraphSLAM is graph optimization - the process of minimizing the error present in all of the constraints in the graph. Let's take a look at what these constraints look like, and learn to apply a principle called *maximum likelihood estimation* (MLE) to structure and solve the optimization problem for the graph.

#### Likelihood

Likelihood is a complementary principle to probability. While probability tries to estimate the outcome given the parameters, likelihood tries to estimate the parameters that best explain the outcome. For example,

- **Probability:** What is the probability of rolling a 2 on a 6-sided die? Answer: 1/6
- **Likelihood:** I've rolled a die 100 times, and a 2 was rolled 10% of the time, how many sides does my die have? Answer: 10 sides

When applied to SLAM, likelihood tries to estimate the most likely configuration of state and feature locations given the motion and measurement observations.

#### Feature Measurement Example

Let's look at a very simple example - one where our robot is taking repeated measurements of a feature in the environment. This example will walk you through the steps required to solve it, which can then be applied to more complicated problems.

The robot's initial pose has a variance of 0 - simply because this is its start location. Recall that wherever the start location may be - we call it location 0 in our relative map. Every action pose and measurement hereafter will be uncertain. In GraphSLAM, we will continue to make the assumption that motion and measurement data has Gaussian noise.

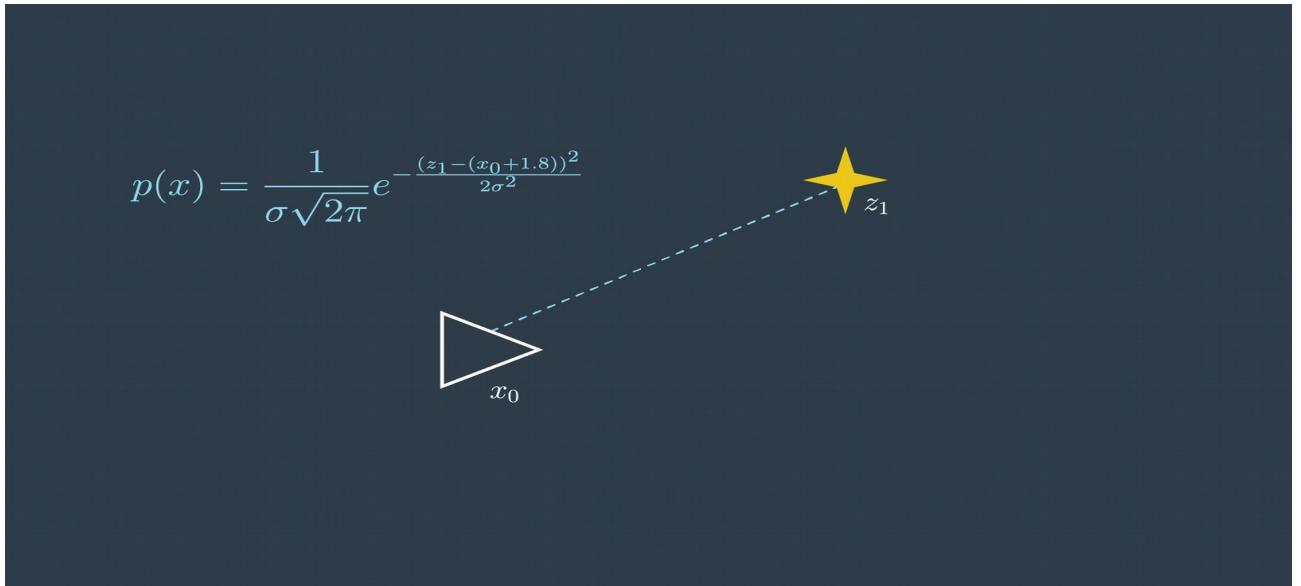
The robot takes a measurement of a feature, m1, and it returns a distance of 1.8 metres.

If we return to our spring analogy - 1.8m is the spring's resting length. This is the spring's most desirable length; however, it is possible for the spring to be compressed or elongated to accommodate other forces (constraints) that are acting on the system.

This probability distribution for this measurement can be defined as so,

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2} \frac{(z_1 - (x_0 + 1.8))^2}{\sigma^2}}$$

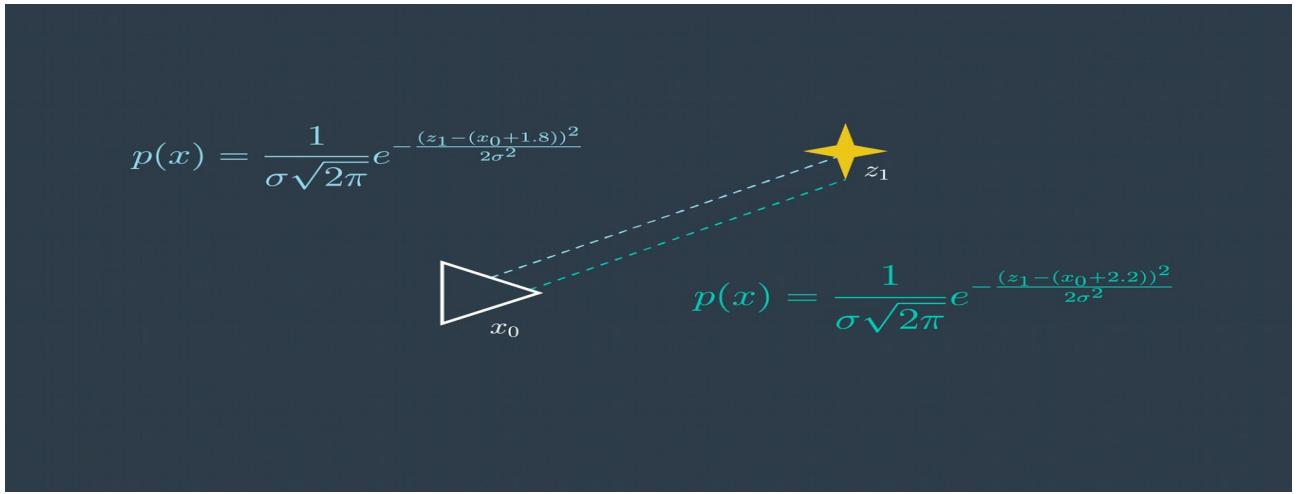
In simpler terms, the probability distribution is highest when  $z_1$  and  $x_0$  are 1.8 meters apart.



However, since the location of the first pose,  $x_0$  is set to 0, this term can simply be removed from the equation.

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2} \frac{(z_1 - 1.8)^2}{\sigma^2}}$$

Next, the robot takes another measurement of the same feature in the environment. This time, the data reads 2.2m. With two conflicting measurements, this is now an overdetermined system - as there are more equations than unknowns!



With two measurements, the most probable location of the feature can be represented by the product of the two probabilities.

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2} \frac{(z_1 - 1.8)^2}{\sigma^2}} * \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2} \frac{(z_1 - 2.2)^2}{\sigma^2}}$$

In this trivial example, it is probably quite clear to you that the most likely location of the feature is at the 2.0 meter mark. However, it is valuable to go through the maximum likelihood estimation process to understand the steps entailed, to then be able to apply it to more complicated systems.

To solve this problem analytically, a few steps can be taken to reduce the equations into a simpler form.

### Remove Scaling Factors

The value of  $m$  that maximizes the equation does not depend on the constants in front of each of the exponentials. These are scaling factors, however in SLAM we are not usually interested in the absolute value of the probabilities, but finding the maximum likelihood estimate. For this reason, the factors can simply be removed.

$$\cancel{\frac{1}{\sigma\sqrt{2\pi}}} e^{-\frac{1}{2} \frac{(z_1 - 1.8)^2}{\sigma^2}} * \cancel{\frac{1}{\sigma\sqrt{2\pi}}} e^{-\frac{1}{2} \frac{(z_1 - 2.2)^2}{\sigma^2}}$$

### Log-Likelihood

The product of the probabilities has been simplified, but the equation is still rather complicated - with exponentials present. There exists a mathematical property that can be applied here to convert this product of exponentials into the sum of their exponents.

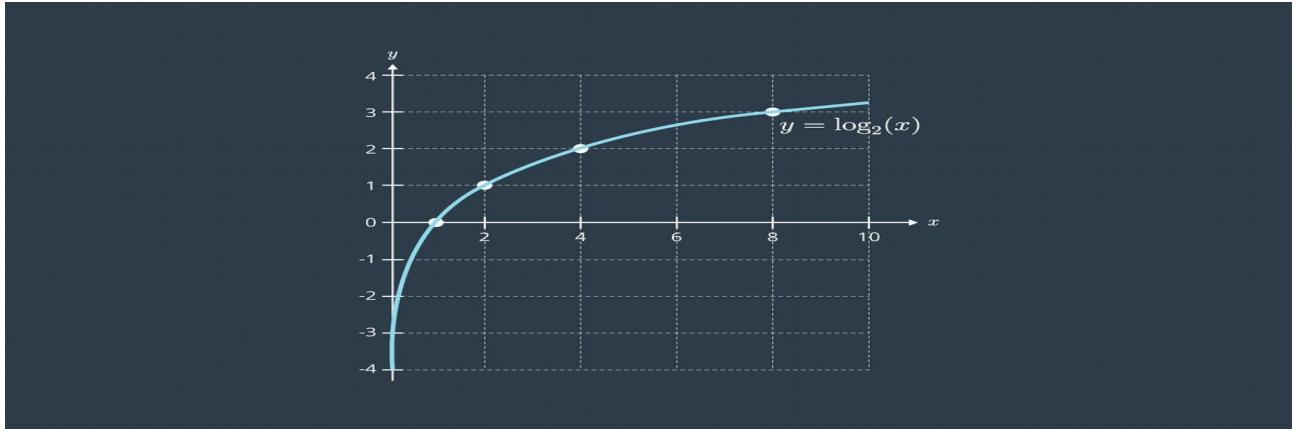
First, we must use the following property,  $e^{a+b} = e^a e^b$ , to combine the two exponentials into one.

$$\begin{aligned} & e^{-\frac{1}{2} \frac{(z_1 - 1.8)^2}{\sigma^2}} * e^{-\frac{1}{2} \frac{(z_1 - 2.2)^2}{\sigma^2}} \\ &= e^{-\frac{1}{2} \frac{(z_1 - 1.8)^2}{\sigma^2} - \frac{1}{2} \frac{(z_1 - 2.2)^2}{\sigma^2}} \end{aligned}$$

Next, instead of working with the likelihood, we can take its natural logarithm and work with it instead.

$$\begin{aligned} \ln(e^{-\frac{1}{2} \frac{(z_1 - 1.8)^2}{\sigma^2}} - \frac{1}{2} \frac{(z_1 - 2.2)^2}{\sigma^2}) \\ = -\frac{1}{2} \frac{(z_1 - 1.8)^2}{\sigma^2} - \frac{1}{2} \frac{(z_1 - 2.2)^2}{\sigma^2} \end{aligned}$$

The natural logarithm is a [monotonic function](#) - in the log's case - it is always increasing - as can be seen in the graph below. There can only be a one-to-one mapping of its values. This means that optimizing the logarithm of the likelihood is no different from maximizing the likelihood itself.



One thing to note when working with logs of likelihoods, is that they are always negative. This is because probabilities assume values between 0 and 1, and the log of any value between 0 and 1 is negative. This can be seen in the graph above. For this reason, when working with log-likelihoods, optimization entails *minimizing* the *negative* log-likelihood; whereas in the past, we were trying to maximize the likelihood.

Lastly, as was done before, the constants in front of the equation can be removed without consequence. As well, for the purpose of this example, we will assume that the same sensor was used in obtaining both measurements - and will thus ignore the variance in the equation.

$$(z_1 - 1.8)^2 + (z_1 - 2.2)^2$$

## Optimization

At this point, the equation has been reduced greatly. To get it to its simplest form, all that is left is to multiply out all of the terms.

$$2z_1^2 - 8z_1 + 8.08$$

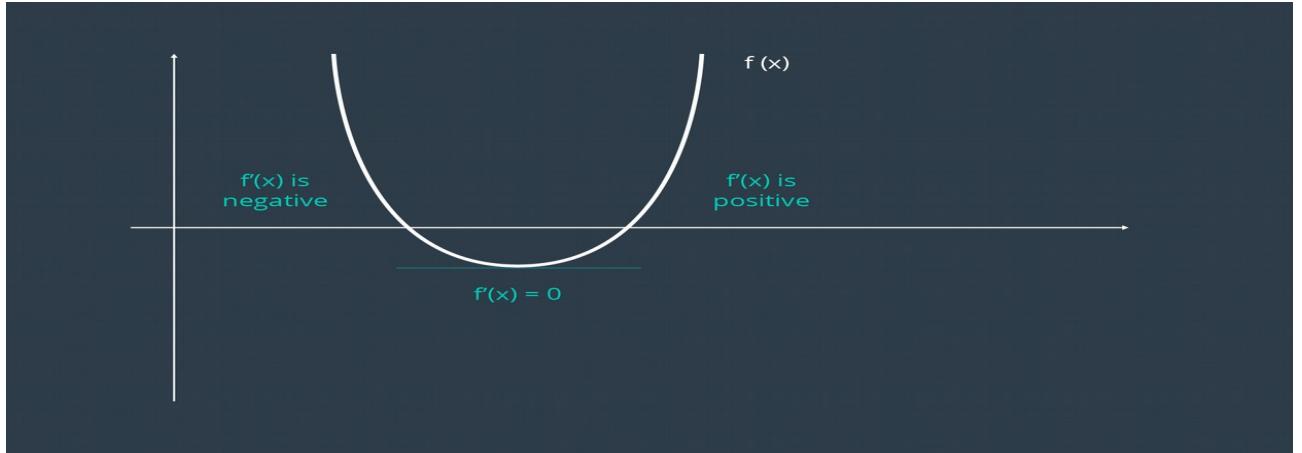
To find the minimum of this equation, you can take the first derivative of the equation and set it to equal 0.

$$4z_1 - 8 = 0$$

$$4z_1 = 8$$

$$z_1 = 2$$

By doing this, you are finding the location on the curve where the slope (or *gradient*, in multi-dimensional equations) is equal to zero - the trough! This property can be visualized easily by looking at a graph of the error function.



In more complex examples, the curve may be multimodal, or exist over a greater number of dimensions. If the curve is multimodal, it may be unclear whether the locations discovered by the first derivative are in fact troughs, or peaks. In such a case, the second derivative of the function can be taken - which should clarify whether the local feature is a local minimum or maximum.

## Overview

The procedure that you executed here is the *analytical* solution to an MLE problem. The steps included,

- Removing inconsequential constants,
- Converting the equation from one of *likelihood estimation* to one of *negative log-likelihood estimation*, and
- Calculating the first derivative of the function and setting it equal to zero to find the extrema.

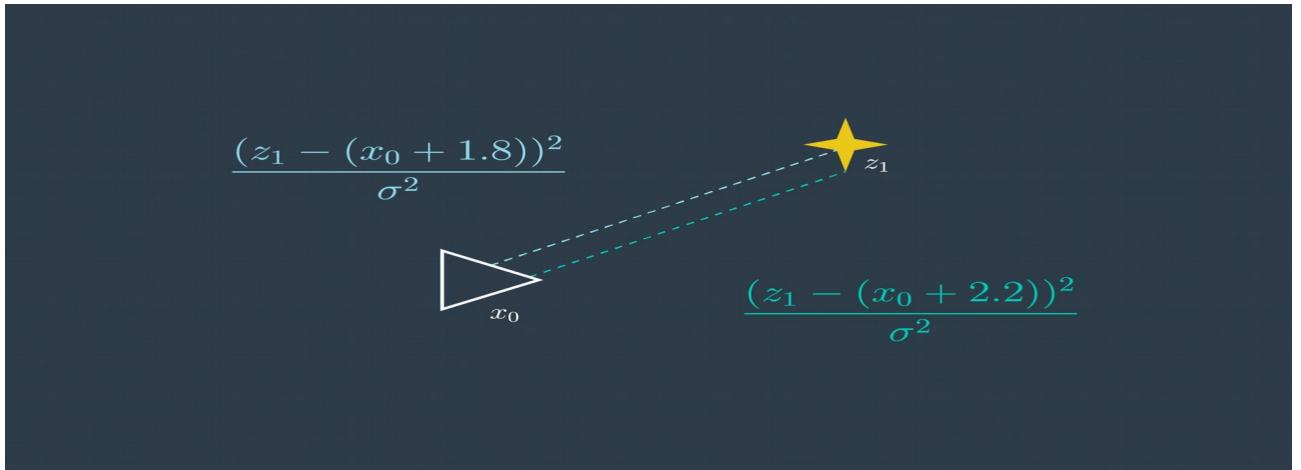
In GraphSLAM, the first two steps can be applied to *every* constraint. Thus, any measurement or motion constraint can simply be labelled with its negative log-likelihood error. For a measurement constraint, this would resemble the following,

$$\frac{(z_t - (x_t + m_t))^2}{\sigma^2}$$

And for a motion constraint, the following,

$$\frac{(x_t - (x_{t-1} + u_t))^2}{\sigma^2}$$

Thus, from now on, constraints will be labelled with their negative log-likelihood error,



with the estimation function trying to minimize the sum of all constraints,

$$J_{GraphSLAM} = \sum_t \frac{(x_t - (x_{t-1} + u_t))^2}{\sigma^2} + \sum_t \frac{(z_t - (x_t + m_t))^2}{\sigma^2}$$

Next, we will work through a more complicated estimation example to better understand maximum likelihood estimation, since it really is the basis of GraphSLAM.

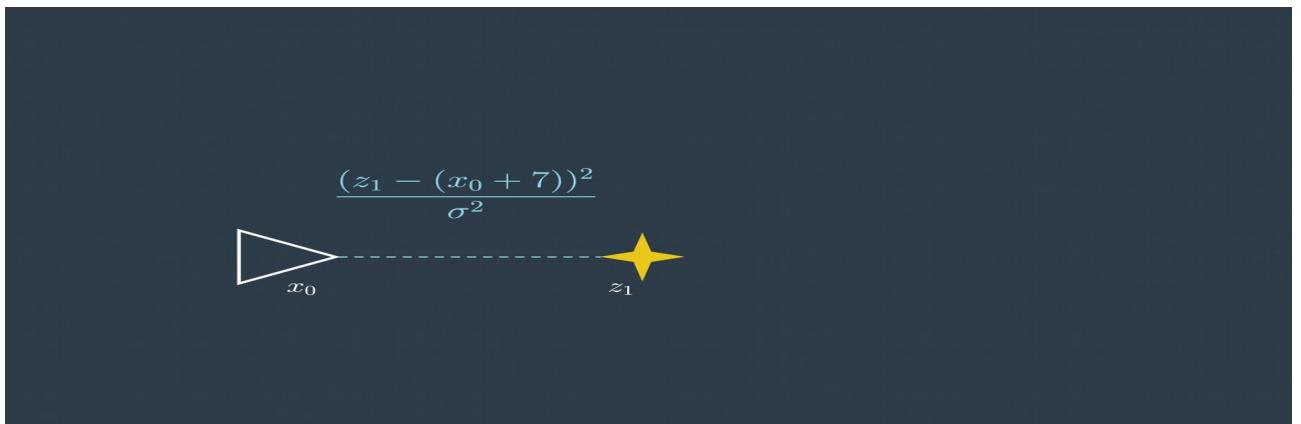
### MLE Example

In the previous example you looked at a robot taking repeated measurements of the same feature in the environment. This example demonstrated the fundamentals of maximum likelihood estimation, but was very limited since it was only estimating one parameter -  $z_1$ .

In this example, we will look on a more complicated 1-dimensional estimation problem.

### Motion and Measurement Example

The robot starts at an arbitrary location that will be labeled 0, and then proceeds to measure a feature in front of it - the sensor reads that the feature is 7 meters way. The resultant graph is shown in the image below.

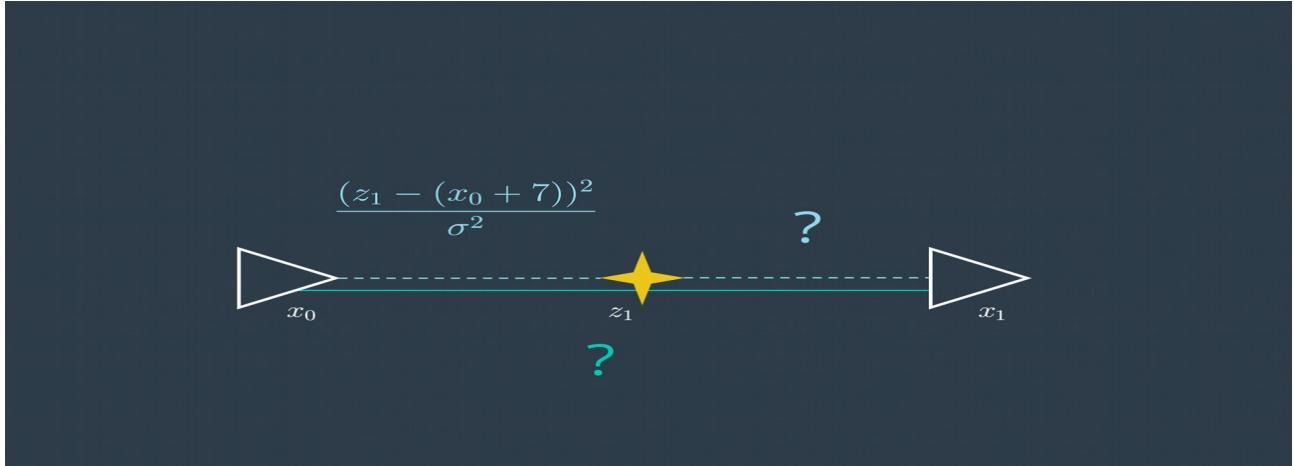


After taking its first measurement, the following Gaussian distribution describes the robot's most likely location. The distribution is highest when the two poses are 3 meters apart.

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(z_1 - (x_0 + 7))^2}{2\sigma^2}}$$

Recall that since we constrained the robot's initial location to 0,  $x_0$  can actually be removed from the equation.

Next, the robot moves forward by what it records to be 10 meters, and takes another measurement of the same feature. This time, the feature is read to be 4 meters behind the robot. The resultant graph looks like so,



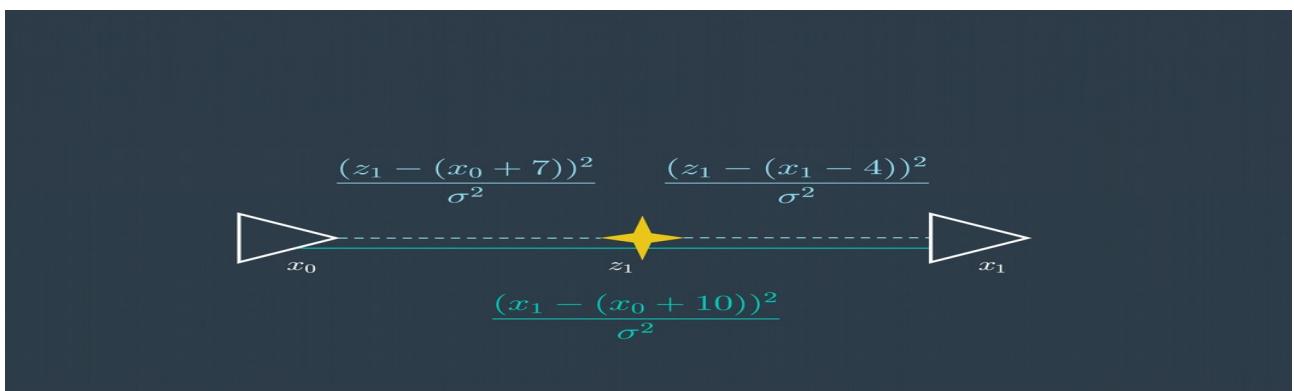
Now let's look at what the two new constraints look like!

The correct constraint for the robot's motion from  $x_0$  to  $x_1$  is :  $\frac{(x_1 - (x_0 + 10))^2}{\sigma^2}$

The correct constraint for the robot's measurement from  $x_1$  to  $z_1$  is:  $\frac{(z_1 - (x_1 - 4))^2}{\sigma^2}$

## Sum of Constraints

The completed graph, with all of its labeled constraints can be seen below.



Now, the task at hand is to minimize the sum of all constraints:

$$J_{GraphSLAM} = \frac{(z_1 - 7)^2}{\sigma^2} + \frac{(x_1 - (x_0 + 10))^2}{\sigma^2} + \frac{(z_1 - (x_1 - 4))^2}{\sigma^2}$$

To do this, you will need to take the first derivative of the function and set it to equal zero. Seems easy, but wait - there are two variables! You'll have to take the derivative with respect to each, and then solve the system of equations to calculate the values of the variables.

For this calculation, assume that the measurements and motion have equal variance.

See if you can work through this yourself to find the values of the variables

What are the estimated locations of z1 and x1 based on your calculations?

Z1 = 6.67, X1 = 10.33

If you've gotten this far, you've figured out that in the above example you needed to take the derivative of the error equation with respect to two different variables - z1 and x1 - and then perform variable elimination to calculate the most likely values for z1 and x1. This process will only get more complicated and tedious as the graph grows.

### Optimization with Non-Trivial Variances

To make matters a little bit more complicated, let's actually take into consideration the variances of each measurement and motion. Turns out that our robot has the fanciest wheels on the market - they're solid rubber (they won't deflate at different rates) - with the most expensive encoders. But, it looks like the funds ran dry after the purchase of the wheels - the sensor is of very poor quality.

Redo your math with the following new information,

- Motion variance: 0.02,
- Measurement variance: 0.1.

Z1 = 6.54 X1 = 10.09

That seemed to be a fair bit more work than the first example! At this point, we just have three constraints - imagine how difficult this process would be if we had collected measurement and motion data over a period of half-an hour, as may happen when mapping a real-life environment. The calculations would be tedious - even for a computer!

Solving the system analytically has the advantage of finding *the* correct answer. However, doing so can be very computationally intensive - especially as you move into multi-dimensional problems with complex probability distributions. In this example, the steps were easy to perform, but it only takes a short stretch of the imagination to think of how complicated these steps can become in complex multidimensional problems.

Well, *what is the alternative?* you may ask. Finding the maximum value can be done in two ways - *analytically* and *numerically*. Solving the problem numerically allows for a solution to be found rather quickly, however its accuracy may be sub-optimal. Next, you will look at how to solve complicated MLE problems numerically.

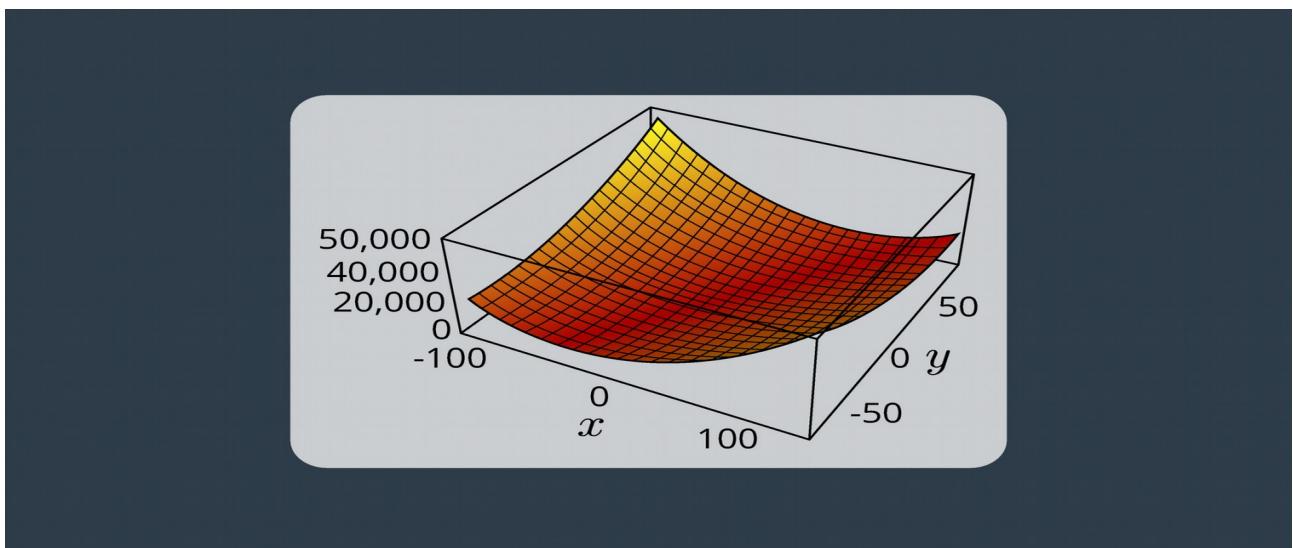
## Numerical Solution to MLE

The method that you applied in the previous two examples was very effective at finding a solution quickly - but that is not always the case. In more complicated problems, finding the analytical solution may involve lengthy computations.

Luckily there is an alternative - numerical solutions to maximum likelihood problems can be found in a fraction of the time. We will explore what a numerical solution to the previous example would look like.

### Numerical solution

The graph of the error function from the previous example is seen below. In this example, it is very easy to see where the global minimum is located. However, in more complicated examples with multiple dimensions this is not as trivial.



This MLE can be solved numerically by applying an optimization algorithm. The goal of an optimization algorithm is to *speedily* find the optimal solution - in this case, the local minimum. There are several different algorithms that can tackle this problem; in SLAM, the [gradient descent](#), [Levenberg-Marquardt](#), and [conjugate gradient](#) algorithms are quite common. A brief summary of gradient descent.

### Quick Refresher on Gradient Descent

Recall that the gradient of a function is a vector that points in the direction of the greatest rate of change; or in the case of an extrema, is equal to zero.

In gradient descent - you make an initial guess, and then adjust it incrementally in the direction *opposite* the gradient. Eventually, you should reach a minimum of the function.

This algorithm does have a shortcoming - in complex distributions, the initial guess can change the end result significantly. Depending on the initial guess, the algorithm converges on two different local minima. The algorithm has no way to determine where the global minimum is - it very naively moves down the steepest slope, and when it reaches a local minima, it considers its task complete.

One solution to this problem is to use stochastic gradient descent (SGD), an iterative method of gradient descent using subsamples of data.

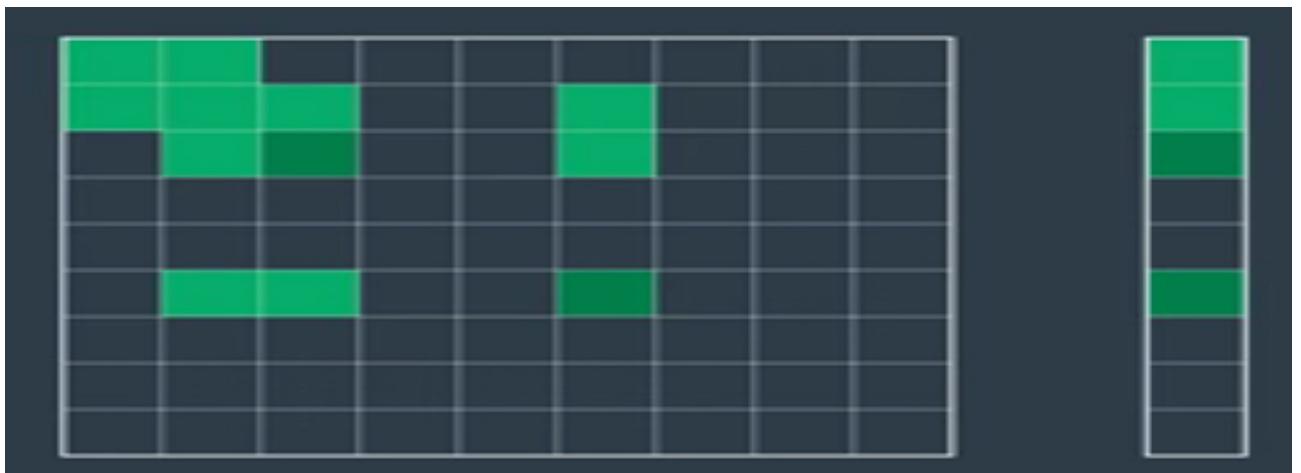
### 5.4.6 Mid-Chapter Overview

At this point, you might be thinking, I could really use a break from performing math calculations or there must be a better way. Well there is!!

We'll briefly introduce the notation used to represent multidimensional measurement and motion constraints.



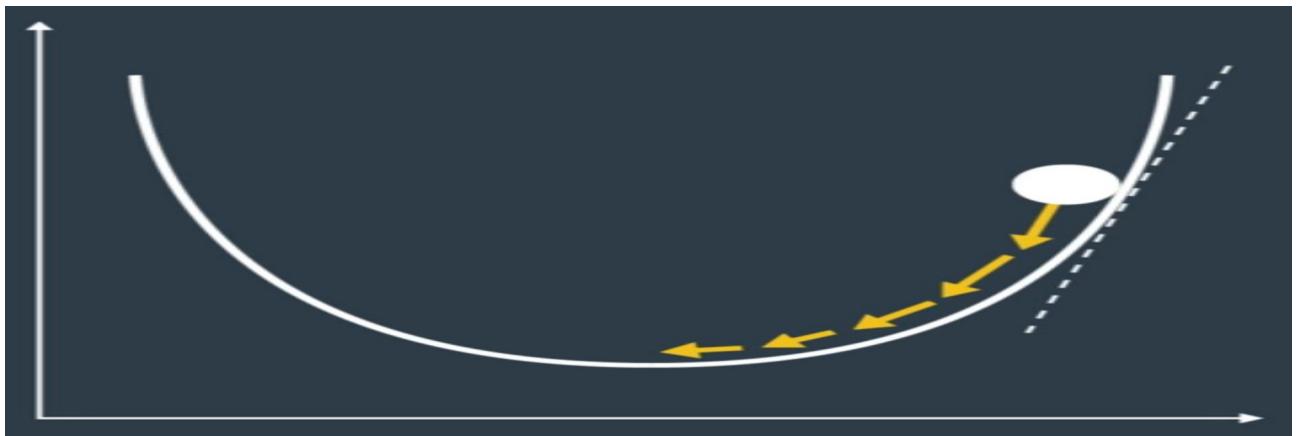
And then we'll learn about information matrices and information vectors. These are data structures that will store constraint information and simplify the process of solving the linear system of equations. In a linear system, adding a constraint is as easy as adding several values to the matrix and vector, then, when we've added all the information, we can solve it.



After that, we're going to face the unfortunate truth that much of what robots experience in the real world just isn't linear. We're going to look at how we can cope with this by once again, linearizing the nonlinear motion and measurement models using Jacobian's.



Lastly, it is not always efficient to solve a system of equations analytically, so we will learn a numerical technique that we briefly explored previously and apply it to graph optimization.



#### 5.4.7 1-D to n-D

##### 1-Dimensional Graphs

In the previous examples, you were working with 1-dimensional graphs. The robot's motion and measurements were limited to one dimension - they could either be performed forwards or backwards.

In such a case, each constraint could be represented in the following form,

$$\textbf{1-D Measurement constraint: } \frac{(z_t - (x_t + m_t))^2}{\sigma^2}$$

$$\textbf{1-D Motion constraint: } \frac{(x_t - (x_{t-1} + u_t))^2}{\sigma^2}$$

##### n-Dimensional Graphs

In multi-dimensional systems, we must use matrices and covariances to represent the constraints. This generalization can be applied to systems of 2-dimensions, 3-dimensions, and really any n-number of dimensions. The equations for the constraints would look like so,

##### n-D Measurement constraint:

$$(z_t - h(x_t, m_j))^T Q_t^{-1} (z_t - h(x_t, m_j))$$

##### n-D Motion constraint:

$$(x_t - g(u_t, x_{t-1}))^T R_t^{-1} (x_t - g(u_t, x_{t-1}))$$

Where  $h()$  and  $g()$  represent the measurement and motion functions, and  $Q_t$  and  $R_t$  are the covariances of the measurement and motion noise. These naming conventions should be familiar to you, as they were all introduced in the Localization chapter.

The multidimensional formula for the sum of all constraints is presented below.

$$J_{GraphSLAM} = x_0^T \Omega x_0 + \sum_t (x_t - g(u_t, x_{t-1}))^T R_t^{-1} (x_t - g(u_t, x_{t-1})) \\ + \sum_t (z_t - h(x_t, m_j))^T Q_t^{-1} (z_t - h(x_t, m_j))$$

The first element in the sum is the initial constraint - it sets the first robot pose to equal to the origin of the map. The covariance,  $\Omega_0$ , represents complete confidence. Essentially,

$$\Omega_0 = \begin{bmatrix} \infty & 0 & 0 \\ 0 & \infty & 0 \\ 0 & 0 & \infty \end{bmatrix}$$

Now that we are working with multi-dimensional graphs and multi-dimensional constraints, it makes sense to use a more intelligent data structure to work with our data. The information matrix and information vector are just that!

#### 5.4.8 Information Matrix and Vector

Now that we know what multi-dimensional constraints look like, it's time to learn a more elegant solution to solving the system of linear equations produced by a graph of constraints.

The information matrix, and information vector, are two data structures that we will use to store information from our constraints. The information matrix is denoted Omega, and the information vector  $\xi$ .



Fundamentally, the information matrix is the inverse of the co-variance matrix. This means that higher certainty is represented with larger values in the information matrix, the opposite of the covariance matrix, where complete certainty was represented by 0.

The matrix and vector holds over all of the poses and all of the features in the environment. Every off-diagonal cell in the matrix is a link between two poses, a pose and a feature or two features. When no information is available about a link the cell has a value of 0.

$x_0$	$x_1$	$x_2$	$x_3$	$x_4$	$m_1$	$m_2$	$m_3$	$m_4$
$x_0$								
$x_1$								
$x_2$								
$x_3$								
$x_4$								
$m_1$								
$m_2$								
$m_3$								
$m_4$								

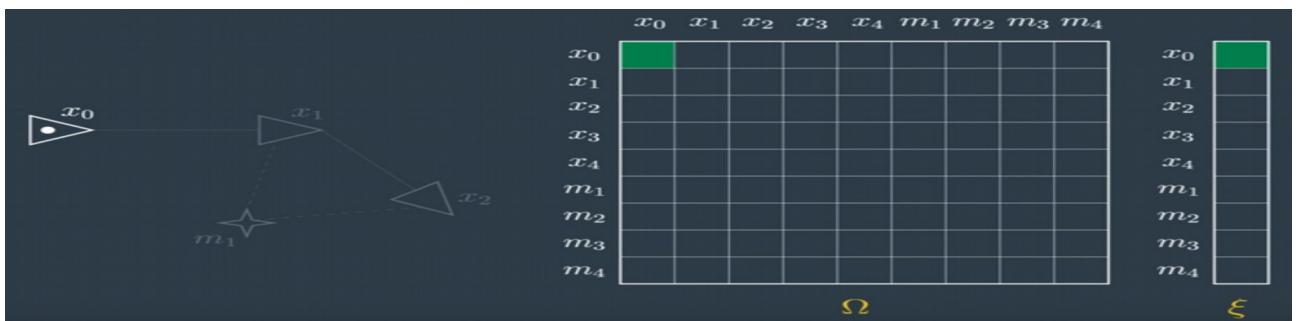
$\Omega$

$\xi$

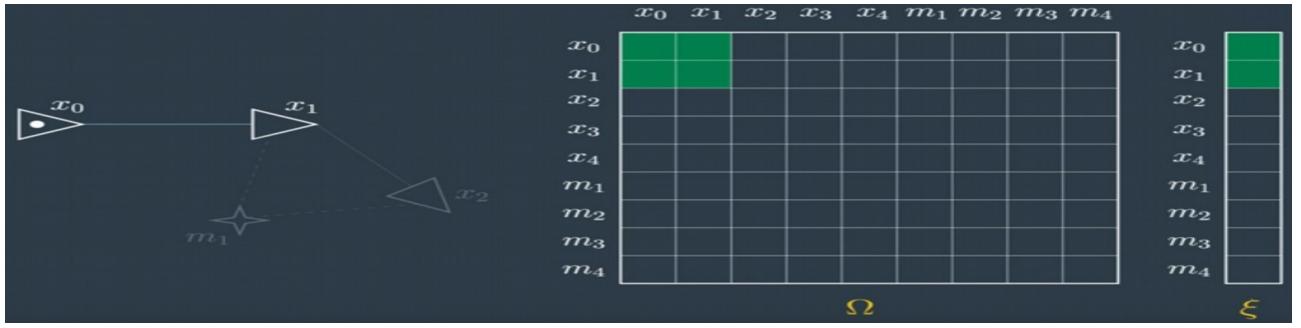
The information matrix and information vector exploit the additive property of the negative log likelihoods of constraints. For a system with linear measurements and motion models, the constraints can be populated into the information matrix and vector in an additive manner.



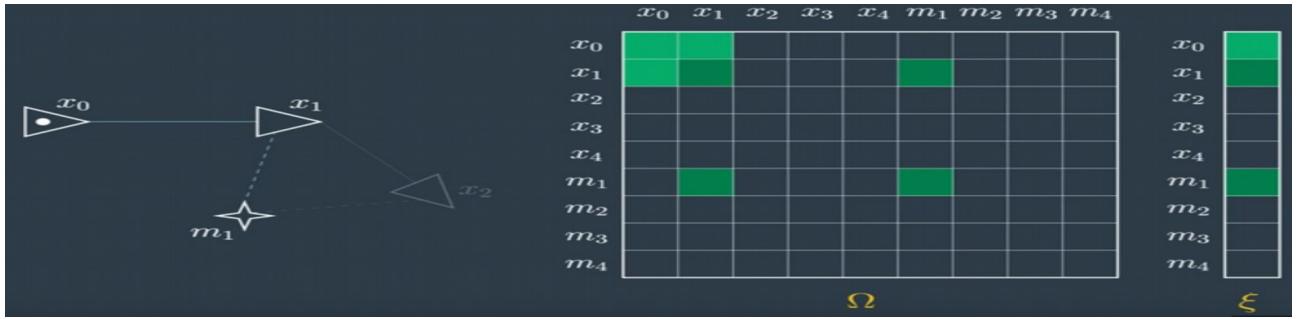
On the left you can see the start of graph that a robot constructed as it moved around some environment. The graph contains five constraints. One initial constraint, two motion constraints and two measurement constraints. This graph will be used to demonstrate at a high level how the information matrix and vector can be populated.



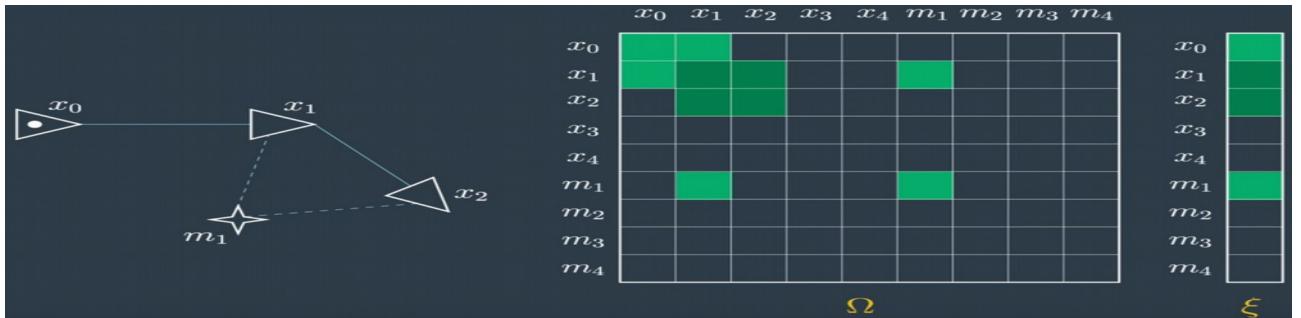
The initial constraint will tie the pose  $X_0$  to a value usually 0, this constraint will populate 1 cell in the information matrix and one in the information vector.



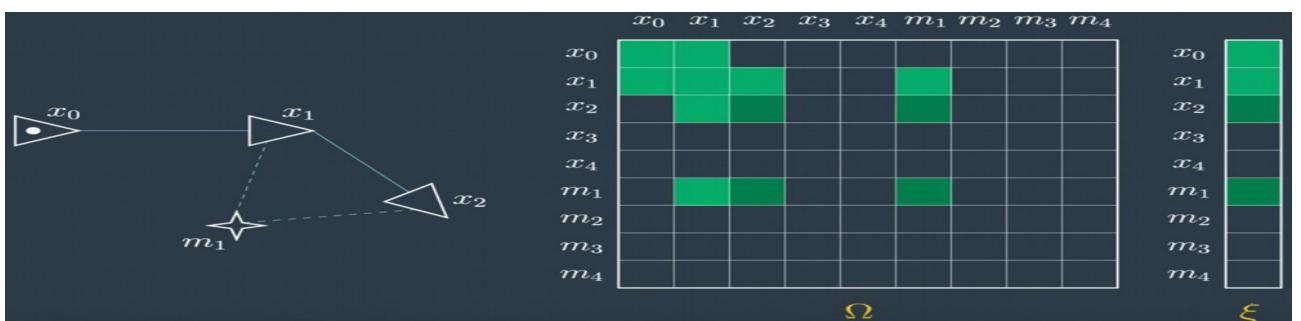
Next, let's look at the motion constrain between  $X0$  and  $X1$ . A motion constraint will tie together two robot poses populating 4 cells in the matrix and two in the vector, these are the cells that relate  $X0$  and  $X1$  to each other.



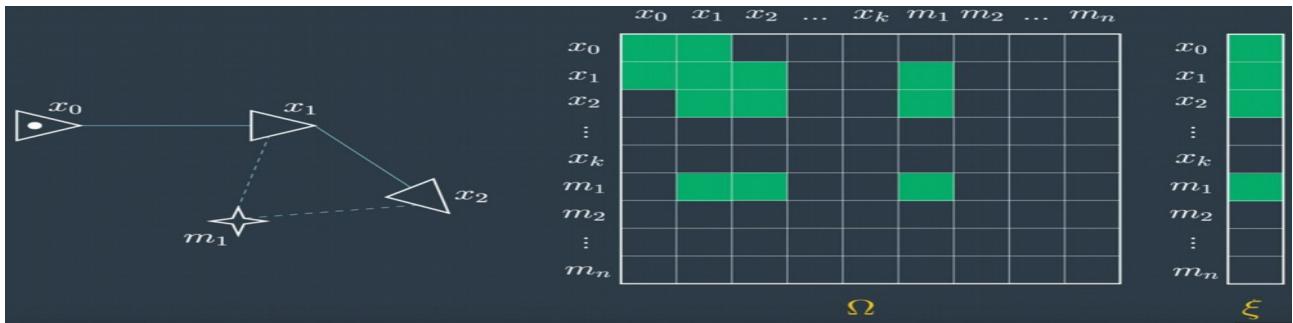
Similarly, a measurement constraint will update 4 cells in the matrix and two in the vector. These are the cells that relate the feature to the pose from which it was measured. In this case  $m_1$  and  $x_1$ .



Recall that any new information supplied to the matrix and vector is additive. This process continues. The next constraint ties together poses  $x_1$  and  $x_2$ .



And the following constraint ties together feature  $m_1$  to the pose  $x_2$ .



Once every constraint has been added to the information matrix and vector, it is concerned to be populated.

It is common for the number of poses and features to be in the thousands or even ten of thousands. The information matrix is considered sparse because most off-diagonal elements are zero, since there is no relative information to tie them together. This sparsity is very valuable when it comes to solving the system of equations that is embedded in the information matrix and vector, and this is what we will explore next.

## Summary

- A **motion constraint** ties together two poses,
- A **measurement constraint** ties together the feature and the pose from which is was measured,
- Each operation updates 4 cells in the information matrix and 2 cells in the information vector,
- All other cells remain 0. Matrix is called ‘sparse’ due to large number of zero elements,
- **Sparsity** is a very helpful property for solving the system of equations.

## 5.4.9 Inference

### Inference

Once the information matrix and information vector have been populated, the path and map can be recovered by the following operation,

$$\mu = \Omega^{-1} * \xi$$

The result is a vector,  $\mu$  defined over all poses and features, containing the best estimate for each. This operation is *very* similar to what you encountered before in the simple one-dimensional case, with a bit of added structure. Just as before, all constraints are considered when computing the solution.

Completing the above operation requires solving a system of equations. In small systems, this is an easily realizable task, but as the size of the graph and matrix grows - efficiency becomes a concern.

The efficiency of this operation, **specifically the matrix inversion**, depends greatly on the topology of the system.

## Linear Graph

If the robot moves through the environment once, without ever returning to a previously visited location, then the topology is linear. Such a graph will produce a rather sparse matrix that, with some effort, can be reordered to move all non-zero elements to near the diagonal. This will allow the above equation to be completed in linear time.

## Cyclical Graph

A more common topology is cyclical, in which a robot revisits a location that it has been to before, after some time has passed. In such a case, features in the environment will be linked to multiple poses - ones that are not consecutive, but spaced far apart. The further apart in time that these poses are - the more problematic, **as such a matrix cannot be reordered to move non-zero cells closer to the diagonal**. The result is a matrix that is more computationally challenging to recover.

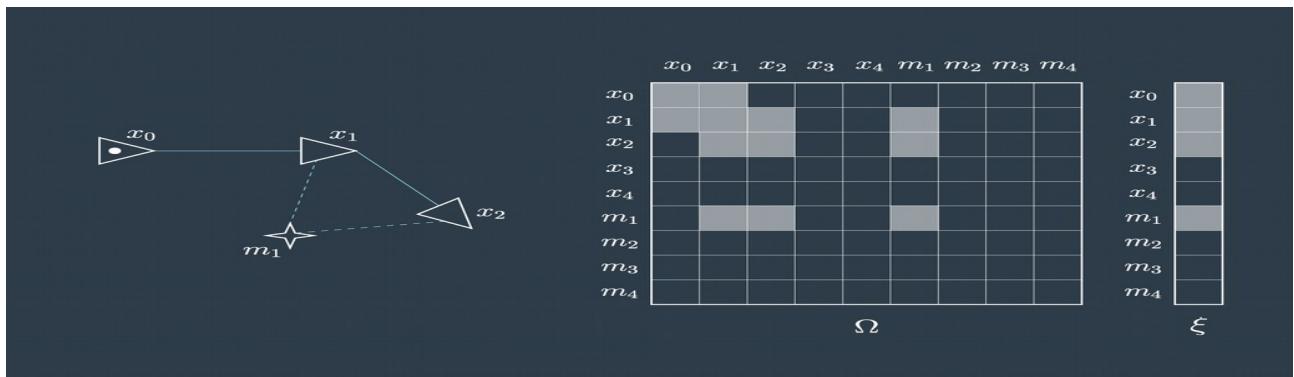
However, all hope is not lost - a variable elimination algorithm can be used to simplify the matrix, allowing for the inversion and product to be computed quicker.

## Variable Elimination

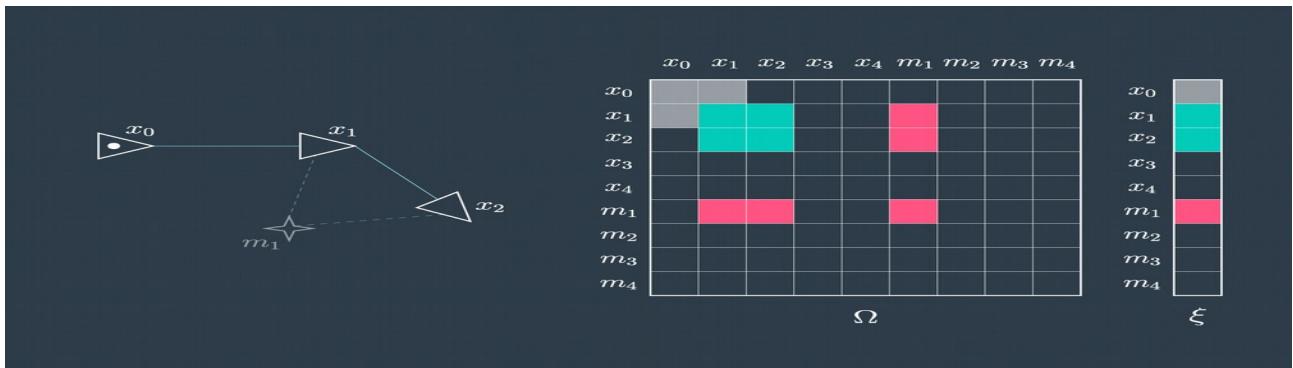
Variable elimination can be applied iteratively to remove all cyclical constraints. Just like it sounds, variable elimination entails removing a variable (ex. feature) entirely from the graph and matrix. This can be done **by adjusting existing links or adding new links to accommodate for those links that will be removed**.

If you recall the spring analogy, variable elimination **removes features, but keeps the net forces in the springs unaltered by adjusting the tension on other springs or adding new springs where needed**.

This process is demonstrated in the following two images. The first image shows the graph, matrix, and vector as they were presented previously.



The second image shows the elimination of  $m_1$ . You can see that in this process the matrix will have five cells reset to zero (indicated in red), and four cells will have their values adjusted (indicated in green) to accommodate the variable elimination. Similarly, the information vector will have one cell removed and two adjusted.



This process is repeated for all of the features, and in the end the matrix is defined over all robot poses. At this point, the same procedure as before can be applied,  $\mu = \Omega^{-1} \xi$ .

Performing variable elimination on the information matrix/vector prior to performing inference is less computationally intense than attempting to solve the inference problem directly on the unaltered data.

In practice, the **analytical inference method described above is seldom applied, as numerical methods are able to converge on a sufficiently accurate estimate in a fraction of the time**. More will be said on this topic later, **but first it is important to explore how nonlinear constraints are handled in GraphSLAM**.

### 5.4.10 Nonlinear Constraints

#### Nonlinear Constraints

In the Localization chapter, you were introduced to nonlinear motion and measurement models. The idea that a robot only moves in a linear fashion is quite limiting, and so it became important to understand how to work with nonlinear models. In localization, nonlinear models couldn't be applied directly, as they would have turned the Gaussian distribution into a much more complicated distribution that could not be computed in closed form (analytically, in a finite number of steps). The same is true of nonlinear models in SLAM - most motion and measurement constraints are nonlinear, and must be linearized before they can be added to the information matrix and information vector. Otherwise, it would be impractical, if not impossible, to solve the system of equations analytically.

Luckily, we will be able to apply the same procedure that we learned in the EKF chapter of Localization to linearize nonlinear constraints for SLAM.

If you recall, a Taylor Series approximates a function using the sum of an infinite number of terms. A linear approximation can be computed by using only the first two terms and ignoring all higher order terms. In multi-dimensional models, the first derivative is replaced by a Jacobian - a matrix of partial derivatives.

### Linearizing Constraints

A linearization of the measurement and motion constraints is the following,

$$g(u_t, x_{t-1}) \simeq g(u_t, \mu_{t-1}) + G_t(x_{t-1} - \mu_{t-1})$$

$$h(y_t) \simeq h(\mu_t) + H_t^i(y_t - \mu_t)$$

To linearize each constraint, you need a value for  $\mu_{t-1}$  or  $\mu_t$  to linearize about. This value is quite important since the linearization of a nonlinear function can change significantly depending on which value you choose to do so about. So, what  $\mu_{t-1}$  or  $\mu_t$  is a reasonable estimate for each constraint?

Well, when presented with a completed graph of nonlinear constraints, you can apply only the motion constraints to create a pose estimate,  $[x_0 \dots x_T]^T$ , and use this primitive estimate in place of  $\mu$  to linearize all of the constraints. Then, once all of the constraints are linearized and added to the matrix and vector, a solution can be computed as before, using  $\mu = \Omega^{-1}\xi$ .

This solution is unlikely to be an accurate solution. The pose vector used for linearization will be erroneous, since applying just the motion constraints will lead to a graph with a lot of drift, as errors accumulate with every motion. Errors in this initial pose vector will propagate through the calculations and affect the accuracy of the end result. This is especially so because the errors may increase in magnitude significantly during a poorly positioned linearization (where the estimated  $\mu_t$  is far from reality, or the estimated  $\mu_t$  lies on a curve where a small step in either direction will make a big difference).

To reduce this error, we can repeat the linearization process several times, each time using a better and better estimate to linearize the constraints about.

## Iterative Optimization

The first iteration will see the constraints linearized about the pose estimate created using solely motion constraints. Then, the system of equations will be solved to produce a solution,  $\mu$ .

The next iteration will use this solution,  $\mu$ , as the estimate used to linearize about. The thought is that this estimate would be a little bit better than the previous; after all, it takes into account the measurement constraints too.

This process continues, with all consequent iterations using the previous solution as the vector of poses to linearize the constraints about. Each solution incrementally improves on the previous, and after some number of iterations the solution converges.

## Summary

Nonlinear constraints can be linearized using Taylor Series, but this inevitably introduces some error. To reduce this error, the linearization of every constraint must occur as close as possible to the true location of the pose or measurement relating to the constraint. To accomplish this, an iterative solution is used, where the point of linearization is improved with every iteration. After several iterations, the result,  $\mu$ , becomes a much more reasonable estimate for the true locations of all robot poses and features.

The workflow for GraphSLAM with nonlinear constraints is summarized below:

- Collect data, create graph of constraints,
- Until convergence:
  - Linearize all constraints about an estimate,  $\mu$ , and add linearized constraints to the information matrix & vector,
  - Solve system of equations using  $\mu = \Omega^{-1} \cdot \xi$ .

### 5.4.11 Graph-SLAM at a Glance

Congratulation on learning GraphSLAM! This algorithm is quite challenging to learn, there is a fair bit of math to swift through and there are a lot of intricacies to its operation. Since the start of the chapter we learn

- How to **construct a graph** of poses and features
- How to **define the constraints** in both in 1-D and multi-D systems
- How to **solve the system of equations** to determine the most likely set of poses given the observations
- How to **handle non-linear measurements and motion constraints by linearizing them.**
- How to **solve a system of equations iteratively until convergence** to achieve the best results.

Next, we'll be learning a specific graph-based SLAM approach called RTAB-Map.

If you'd like to dive deeper into the mathematics of GraphSLAM, feel free to explore the following resources:

[A Tutorial on Graph-Based SLAM, Grisetti](#)

[The GraphSLAM Algorithm with Applications to Large-Scale Mapping of Urban Structures, Thrun](#)

## 5.5 3D SLAM With RTAB-Map

### 5.5.1 Intro to 3D SLAM With RTAB-Map

Now, we're going to dive into a Graph-Based SLAM approach called Real-Time Appearance-Based or RTAB-Map.

Appearance-Based SLAM means that the algorithm uses data collected from vision sensors to localize the robot and map the environment.

In Appearance-Based methods, a process called loop closure is used to determine whether the robot has seen a location before. As the robot travels to new areas in its environment, the map is expanded, and the number of images that each new image must be compared to increases, this causes the loop closure to take longer with the complexity increasing linearly.

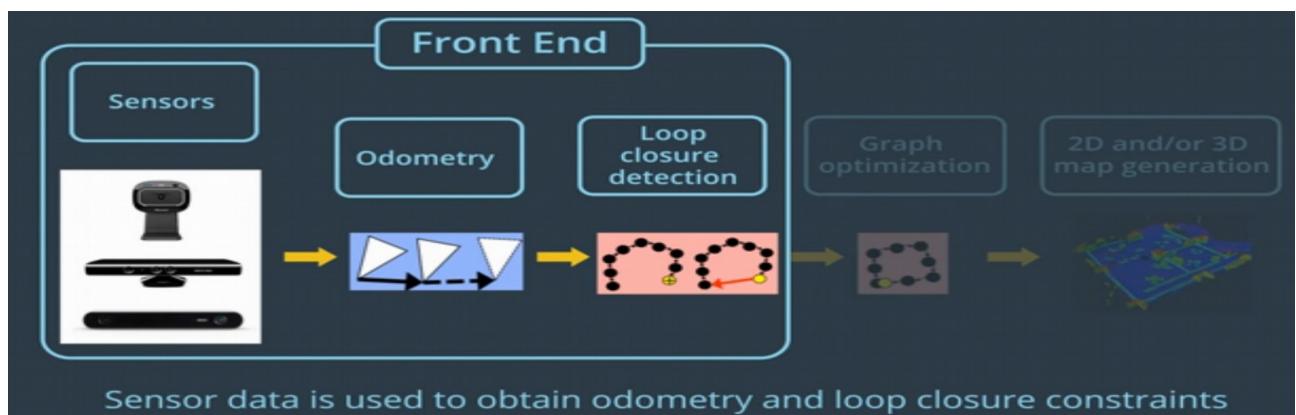
RTAB-Map is optimized for large-scale and long-term SLAM by using multiple strategies to allow for loop closure to be done in real-time.

In this context, this means that the loop closure is happening fast enough that the result can be obtained before the next camera images are acquired.

In the remainder of this chapter, we will learn about how the front-end and back-end structure of RTAB-Map differs from traditional graph SLAM, the importance of loop closure, how the process is being optimized using bag of words and memory management and some practical implications of RTAB-Map.

### 5.5.2 3D SLAM With RTAB-Map

Now, we will discuss the front-end and back-end specific to RTAB-Map.



The front-end of RTAB-Map focuses on sensor data used to obtain the constraints that are used for feature optimization approaches. Although **landmark constraints** are used for other graph SLAM methods like the 2D graph SLAM we saw earlier, RTAB-Map **does not use them**. Only **odometry constraints and loop closure constraints are considered here**.

The odometry constraints can come from wheel encoders, IMU, Lidar or Visual odometry. Visual odometry is accomplished using 2D features such as **Speeded Up Robust Features or SURF**.

**Appearance based**  
**No metric distance information**



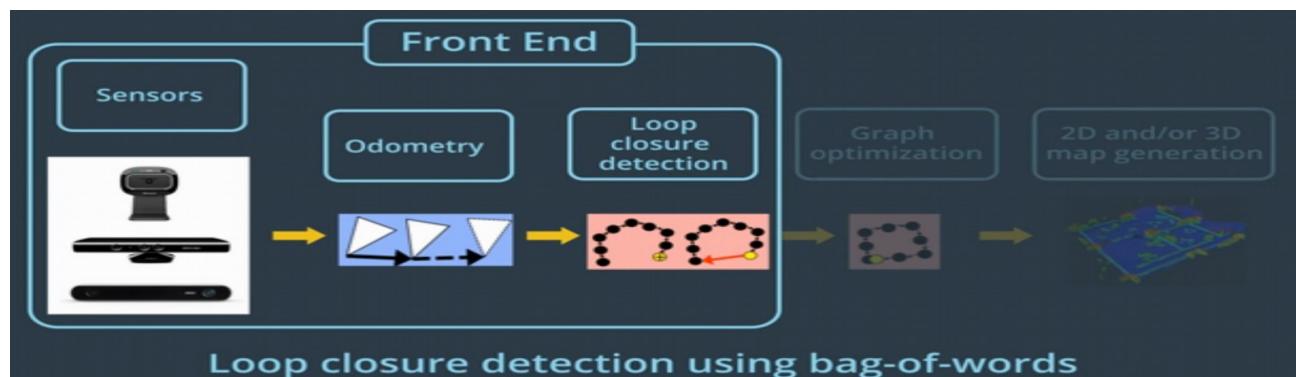
**Appearance based**  
**Metric graph slam**



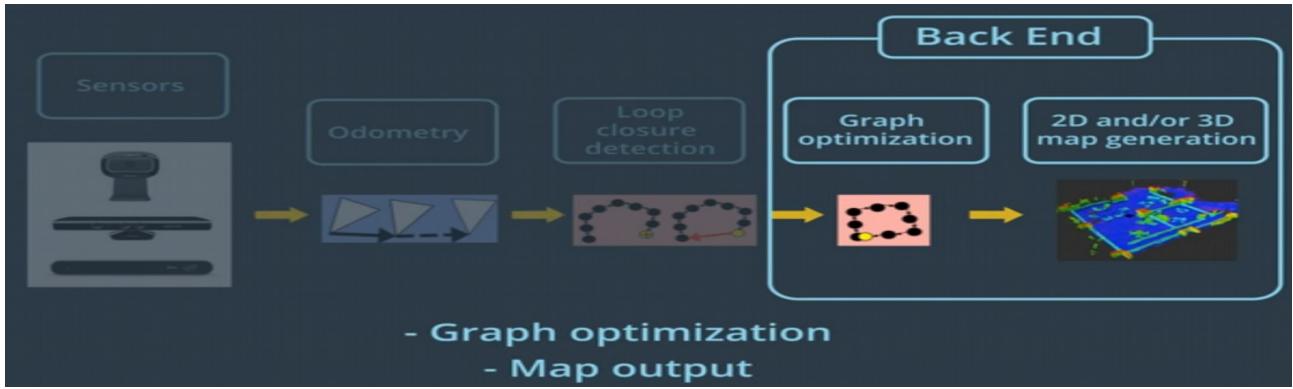
Remember that RTAB-Map is appearance based with no metric distance information. RTAB-Map can use a single monocular camera to detect loop closure.

For metric graph SLAM, RTAB-Map requires an RGB-D camera or a stereo camera to compute the geometric constraint between the images of a loop closure.

A laser range finder can also be used to improve or refine this geometric constraint by providing a more precise localization.



The front-end also involves graph management, which includes node creation and loop closure detection using **bag-of-words**.



The back-end of RTAB-Map includes graph optimization, and assembly of an Occupancy grid from the data of the graph.

## Local loop closure

Loop closure detection using positions from limited map region  
Depends on robots position (odometry)

We will discuss these components in more detail starting with loop closure. As we said, loop closure detection is the process of finding a match between the current and previously visited locations in SLAM. There are two types of loop closure detection: local and global.

Many probabilistic SLAM methods use local loop closure detection where matches are found between a new observation and a limited graph region. The size and location of this limited map region is determined by the uncertainty associated with the robot's position. This type of approach fails if the estimated position is incorrect. As we already seen, it is likely that the events in the real world that the robot is operating in **will cause errors in the estimated position**.

## Global loop closure

Independent of the position estimation (odometry)

In a global loop closure approach, a new location is compared with previously viewed locations. If no match is found, the new location is added to memory. As the map grows and more locations are added, the amount of time to check whether the location has been previously seen increases linearly. If the time it takes to search and compare new images to the ones stored in memory becomes larger than the acquisition time, the map becomes ineffective.

**RTAB-Map uses a global loop closure approach combined with other techniques to ensure that the loop closure process happens in real time.**

# Importance of loop closure

Loop closure detection disabled



Loop closure detection enabled



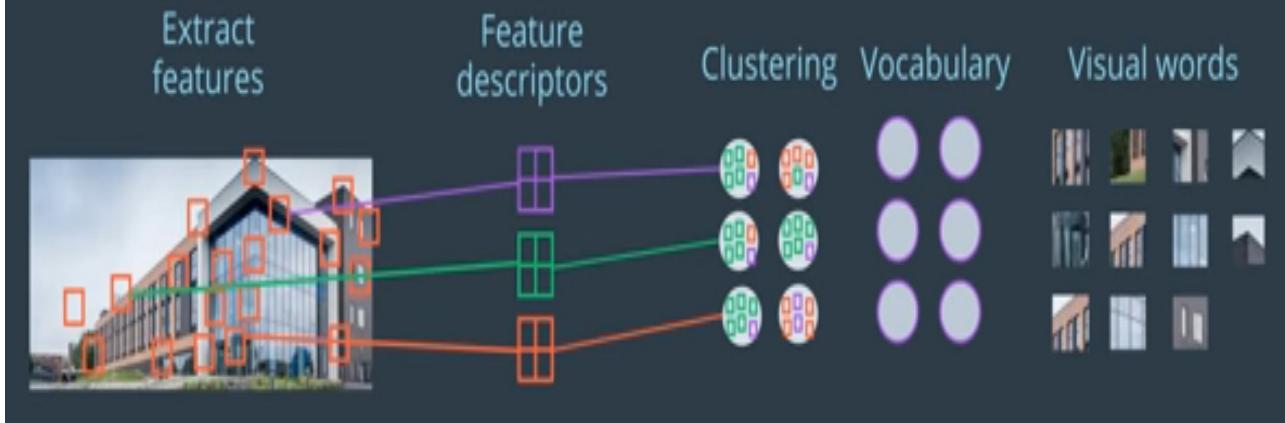
**The importance of loop closure is best understood by seeing a map result without it!**

When loop closure is disabled, you can see parts of the map output that are **repeated**, and the resulting map looks a lot more choppy. It is not an accurate representation of the environment. This is caused by the robot not using loop closure to compare new images and locations to ones that are previously viewed, and instead it registers them as new locations. When loop closure is enabled, the map is significantly smoother and is an accurate representation of the room.

For example, on the left, where loop closure is disabled, you'll see highlighted where the door is represented as multiple corners and parts of a door, where on the right, you see a single clearly defined door.

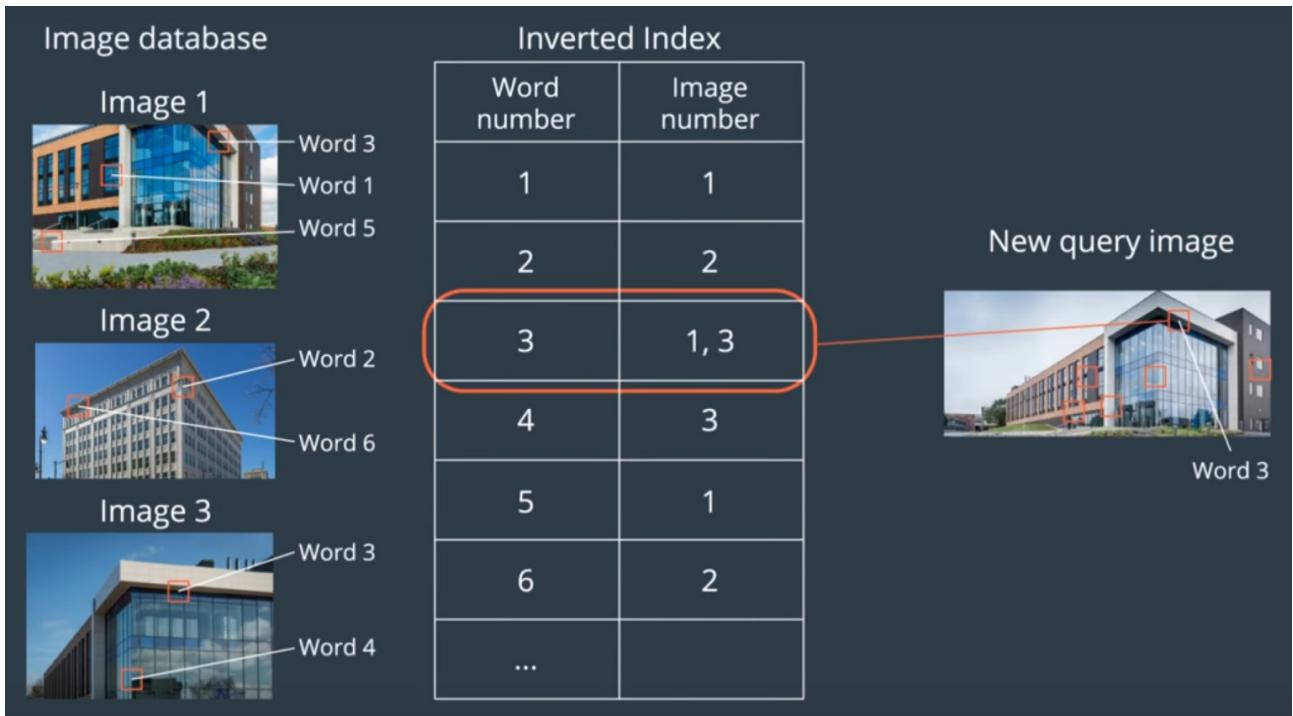
## 5.5.3 Visual Bag-of-Words

# Feature



In RTAB-Mapping **loop closure is detected using a bag-of-words approach**. Bag-of-Words is commonly used in vision-based mapping.

- A **feature is a very characteristic of an image** like a patch with a complex texture, or a well defined edge or corner. In RTAB-Map the default method for extracting features from an image is called Speeded Up Robust Features or **SURF**
- Each feature has a descriptor associated with it. A feature descriptor **is a unique and robust representation of the pixels that make up a feature**. In SURF, the point of interests where the feature is located is split into smaller square sub-regions. From these sub-regions, the pixel intensities in regions of regularly spaced sample points are calculated and compared. The difference between the sample points are used to categorize the sub-regions of the image.
- Comparing feature descriptors directly is time consuming, so a **vocabulary is used for faster comparison**. This is where similar features or synonyms are clustered together.
- The collection of these clusters represent the vocabulary. Once a feature descriptor is matched to one of the vocabulary, it is called **quantization**.
- At this point, the feature is now linked to a word and can be referred to as a **visual world**. When all features in an image are **quantized**, the image is now a **bag-of-words**. Each word keeps **a link to images that is associated with**, making image **retrieval more efficient** over a large data set.



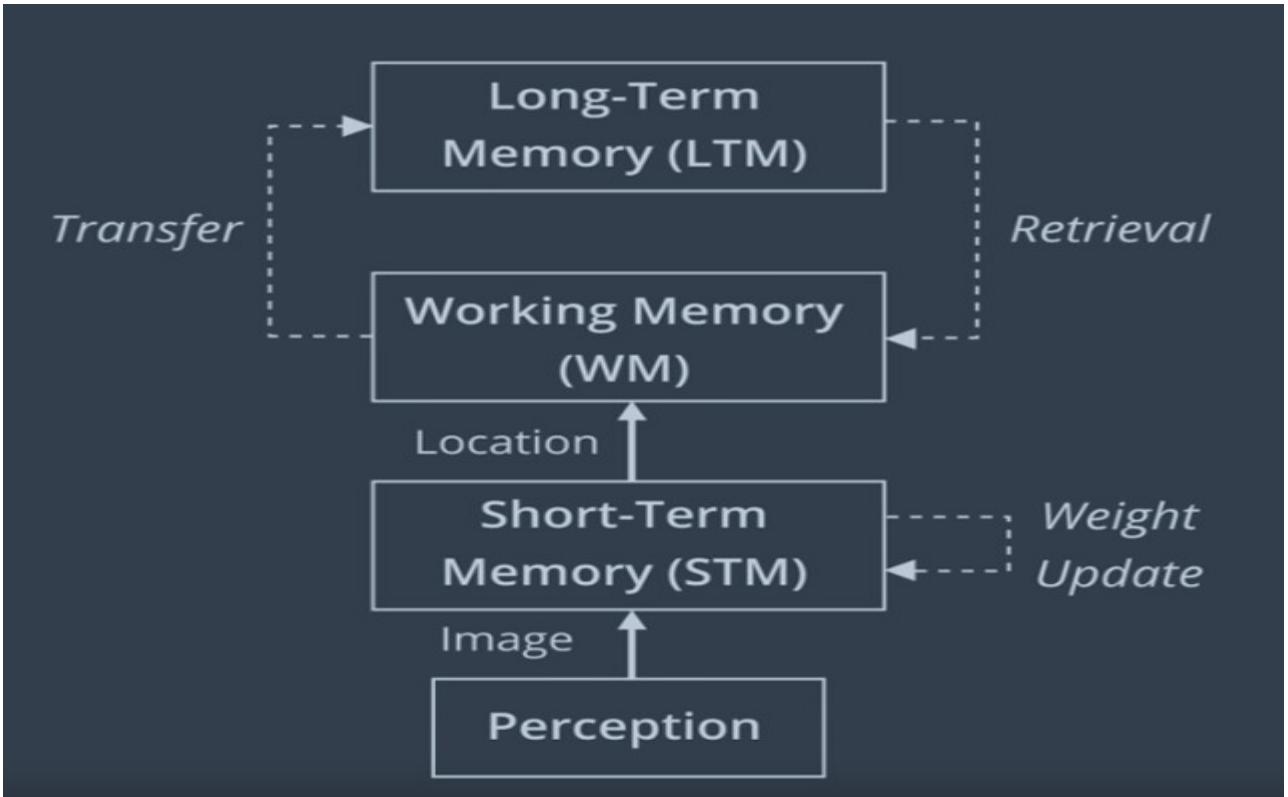
To compare an image with our previous images, **a matching score** is given to all images containing the same words.

Each word keeps track of which image it has been seen in, so similar images can be found. This is called an **inverted index**.

If a word is seen in an image, the score of this image will increase. If an image shares many visual words with the query image, it will score higher.

A **Bayesian filter** is used to evaluate the scores. This is the hypothesis that an image has been seen before. When the hypothesis reaches a predefined threshold  $H$ , a loop closure is detected.

#### 5.5.4 RTAB-Map Memory Management



RTAB-Map uses a memory management technique to limit the number of locations considered as candidates during loop closure detection. This technique is a key feature of RTAB-Map and allows for loop closure to be done in real time.

The overall strategy is to keep the most recent and frequently observed locations in the robot's **Working Memory (WM)**, and transfer the others into **Long-Term Memory (LTM)**.

- When a new image is acquired, a new node is created in the **Short Term Memory (STM)**.
- When creating a node, recall that features are extracted and compared to the vocabulary to find all of the words in the image, creating a bag-of-words for this node.
- Nodes are assigned a weight in the STM based on how long the robot spent in the location (heuristic) - where a longer time means a higher weighting.
- The STM has a fixed size of  $S$ . When STM reaches  $S$  nodes, the oldest node is moved to WM to be considered for loop closure detection. Nodes in STM are not considered during loop closure detection, because they are generally very similar to one another.
- Loop closure happens in the WM.
- WM size depends on a fixed time limit  $T$ . When the time required to process new data reaches  $T$ , some nodes of graph are transferred from WM to LTM - as a result, WM size is kept nearly constant. WM is made up of nodes seen for a longer period of time.
- Oldest and less weighted nodes in WM are transferred to LTM before others, so WM is made up of nodes seen for longer periods of time.
- LTM is not used for loop closure detection and graph optimization.
- If loop closure is detected, neighbours in LTM of an old node can be transferred back to the WM (a process called retrieval).

## 5.5.5 RTAB-Map Optimization and Output

### RTAB-Map Optimization and Output

Here we will discuss graph and map optimization, as well as time complexity considerations.

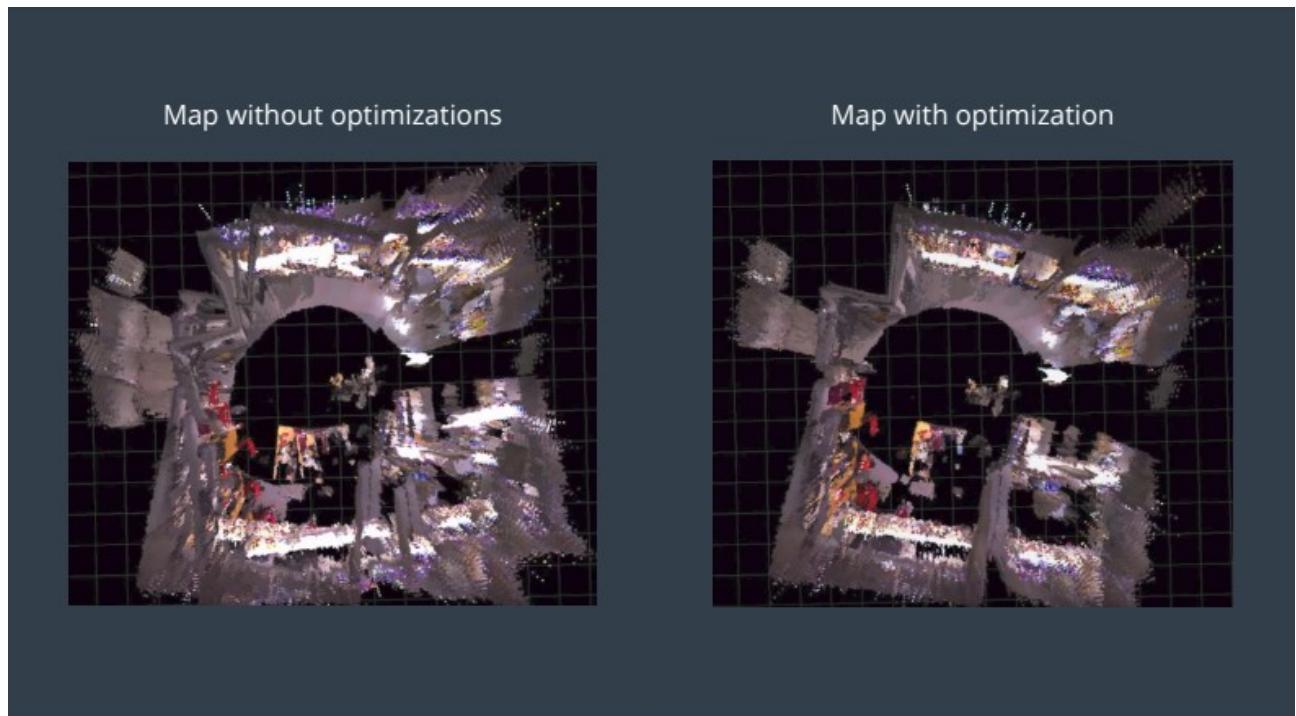
### Graph Optimization

When a loop closure hypothesis is accepted, a new constraint is added to the map's graph, then a graph optimizer minimizes the errors in the map. RTAB-Map supports 3 different graph optimizations: Tree-based network optimizer, or TORO, General Graph Optimization, or G2O and GTSAM (Smoothing and Mapping).

All of these optimizations use node poses and link transformations as constraints. When a loop closure is detected, errors introduced by the odometry can be propagated to all links, correcting the map.

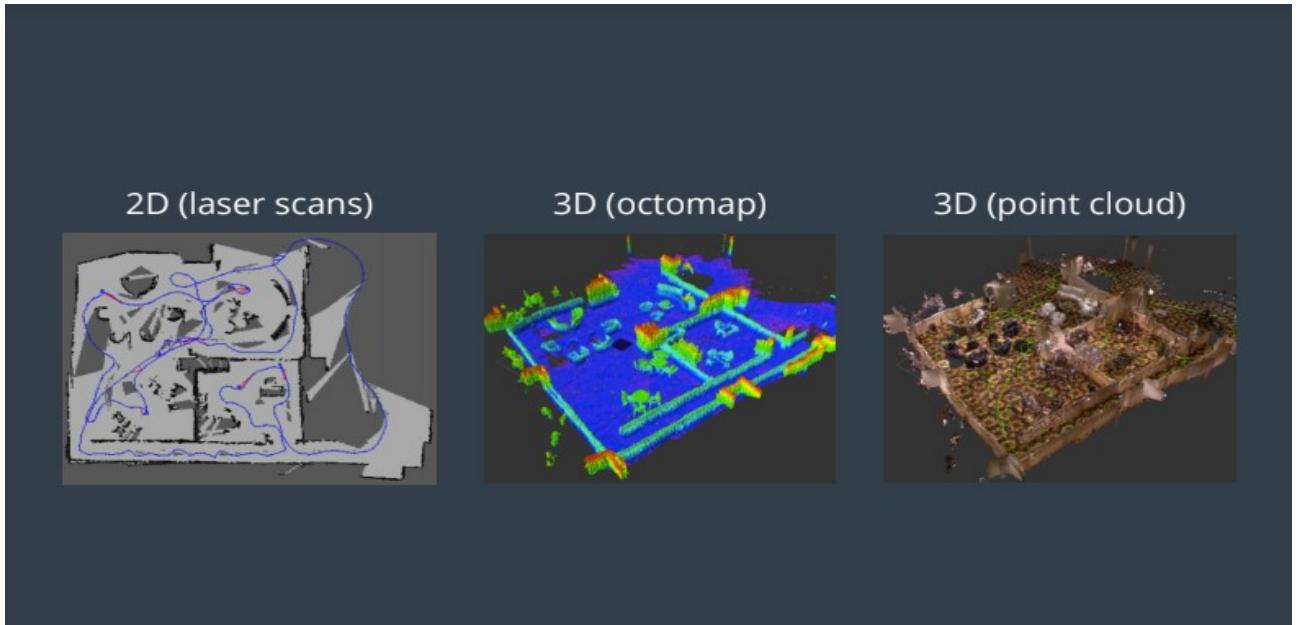
Recall that Landmarks are used in the graph optimization process for other methods, whereas RTAB-Map doesn't use them. Only odometry constraints and loop closure constraints are optimized.

You can see the impact of graph optimization in the comparison below.

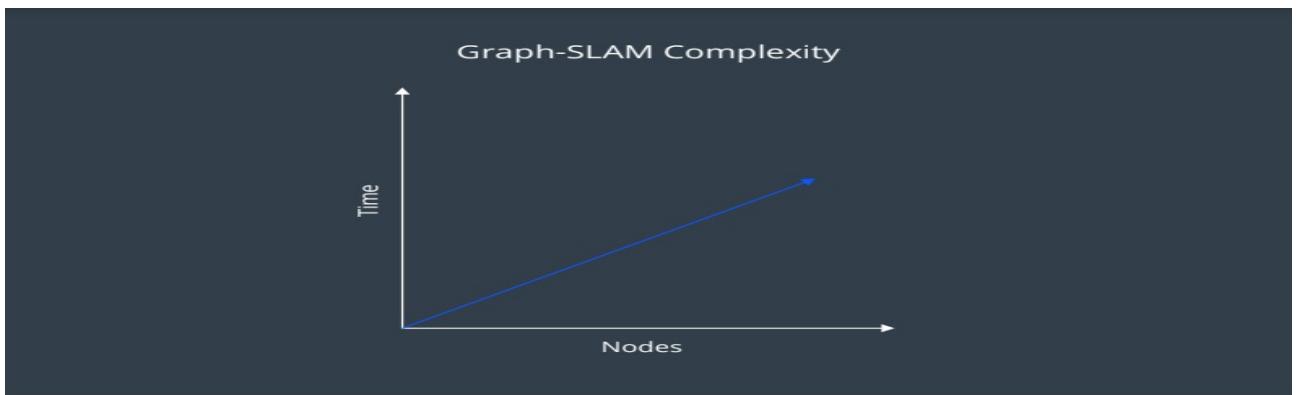


### Map assembly and Output

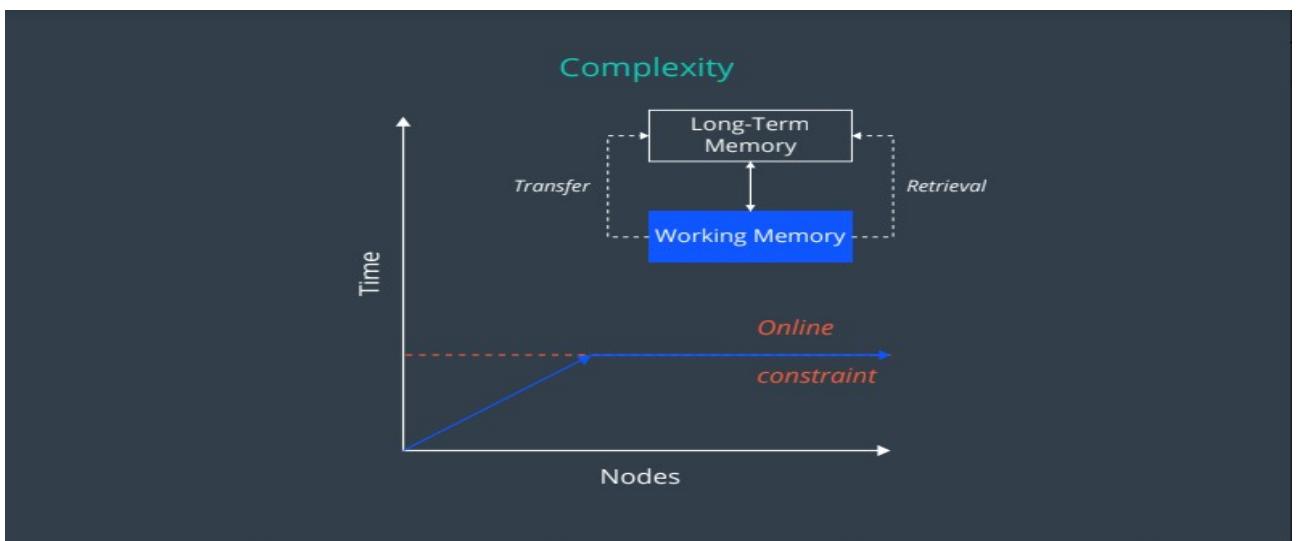
The possible outputs of RTAB-Map are a 2d Occupancy grid map, 3d occupancy grid map (3d octomap), or a 3D point cloud.



## Graph SLAM Complexity and the Complexity of RTAB-Map



Graph-SLAM complexity is linear, according to the number of nodes, which increases according to the size of the map.



By providing constraints associated with how many nodes are processed for loop closure by memory management, the time complexity becomes constant in RTAB-Map.

## 5.6 Outro

In the first half of this chapter, we learned the fundamentals of graph SLAM, how a graph is constructed, how do we evaluate the constraints and how graph optimization is done.

We also learned about a specialized SLAM approach, called RTAB-Map, the importance of loop closure and how the process is optimized using bag of words and memory management.

We haven't fully deconstructed all of the algorithms of this complex SLAM approach, but this knowledge will take us forward and equip us to utilize RTAB-Map as an advanced ROS package in our projects.

**For experimental results regarding RTAB-Map and a video you can see the appendix!!!!**

# 6. Path Planning and Navigation

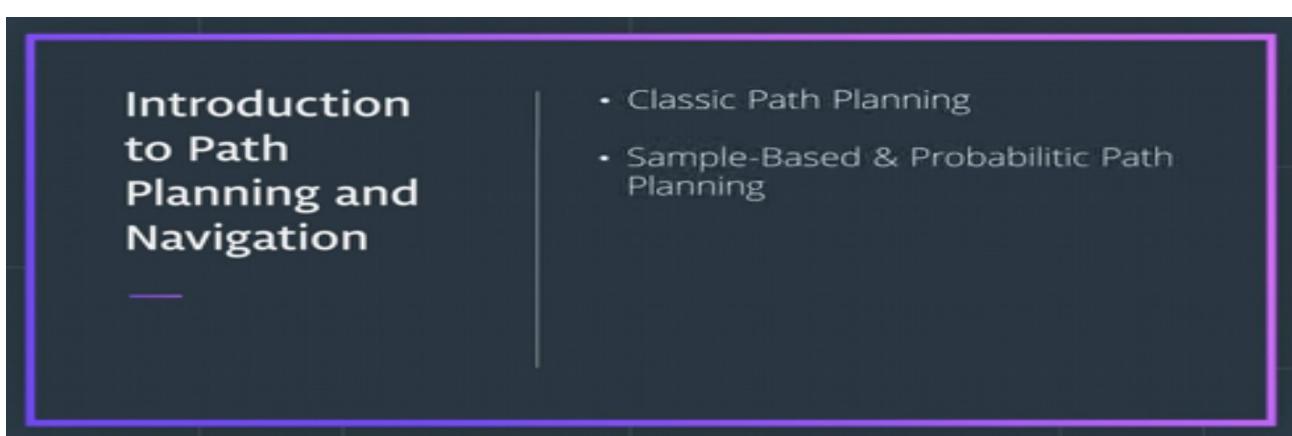
## 6.1 Introduction

So far, we've focused on the questions of, where a robot is with respect to a map in localization and how to build a map of the environment in mapping and slam. These are both critical elements for robotic mobility.

Now we'll turn our attention to the decision-making aspects of mobile robotics. Path planning and navigation. Given a map and a goal location, path planning is the calculation of the path that gets the robot where it needs to go. Once a path is in place, the robot must navigate to the goal and may encounter unmapped obstacles along the way that conflict with the planned path. With real time sensor readings, the robot can use obstacle avoidance techniques to modify the original path. While **path planning is a strategic solution to the problem of, how do I get there, obstacle avoidance is a series of tactical decisions the robot must make as it moves along the path.**



There are multiple approaches for accomplishing both. In this chapter you will learn a number of path planning approaches both **classical and probabilistic** and we will program the **A\* algorithm in C++ on the appendix** alongside how to implement path planning on ROS.



## 6.2 Applications

Before we dive into the details - let's look at where path planning can be applied!

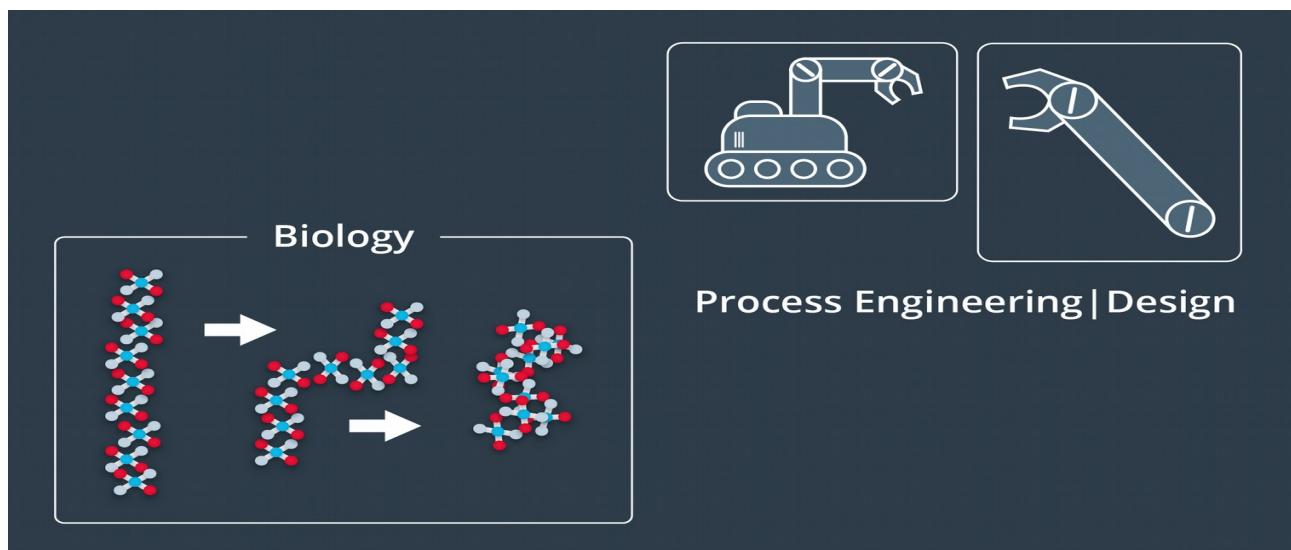
Sitting in your home or office, some environment-specific examples come to mind right away - vacuum robots plan their paths around a house to ensure that every square inch of space gets cleaned. Self-driving cars are starting to appear around us. These vehicles can accept a destination as an input from a human and plan an efficient path that avoids collisions and obeys all traffic regulations.

More peculiar applications of path planning in robotics include assistive robotics. Whether working with the disabled or elderly, robots are starting to appear in care homes and hospitals to assist humans with their everyday tasks. Such robots must be mindful of their surroundings when planning paths - some obstacles stay put over time, such as walls and large pieces of furniture, while others may move around from day to day. Path planning in dynamic environments is undoubtedly more difficult.

Another robotic application of path planning is the planning of paths by exploratory rovers, such as Curiosity on Mars. The rover must safely navigate the surface of Mars (which is between 55 and 400 million kilometers away!). Accurate problem-free planning that avoids risks is incredibly important.

Path planning is not limited to robotics applications, in fact it is widely used in several other disciplines. Computer graphics and animation use path planning to generate the motion of characters. While computational biology applies path planning to the folding of protein chains.

With many different applications, there are naturally many different approaches. Next you will gain the knowledge required to implement several different path planning algorithms.



### Roadmap:

- Classical Path Planning
- Sample-Based and Probabilistic Path Planning

### Outcomes:

- Understand real-world applications of path planning.
- Select the appropriate path planning algorithm for a given application.

- Be able to successfully implement path planning algorithms in code.
- Interface a path planning package with ROS

## 6.3 Classic Path Planning

### 6.3.1 Introduction to Path Planning

Welcome to path planning, the final piece of the puzzle, as it relates to teaching our robot friends to operate in the real world. Just like in localization and SLAM there isn't one correct way to accomplish the task of path planning.

A path planning algorithm takes in as inputs the provided environment geometry, the robots geometry, and the robots start and goal poses and uses this information to produce a path from start to goal.



**The workings of this box is what we'll spend the rest of the chapter examining.**

By the end of this chapter you will be able to –

#### Outcomes

- Recognize different types of path planning algorithms
- Comprehend the inner workings of a collection of algorithms
- Evaluate the suitability of an algorithm for a particular application
- Be ready to implement search algorithms in c++

### 6.3.2 Examples of Path Planning

An exploratory robot may find itself dropped off at a starting position and need to traverse the land, water or air to get to its goal position. In between its start and goal locations, there will inevitably be some obstacles. Let's assume that our rover on land, from the map the robot knows that there's rubble present along it's path. Using GPS data and photographs the rover must plan a path through the rubble to get to its destination.

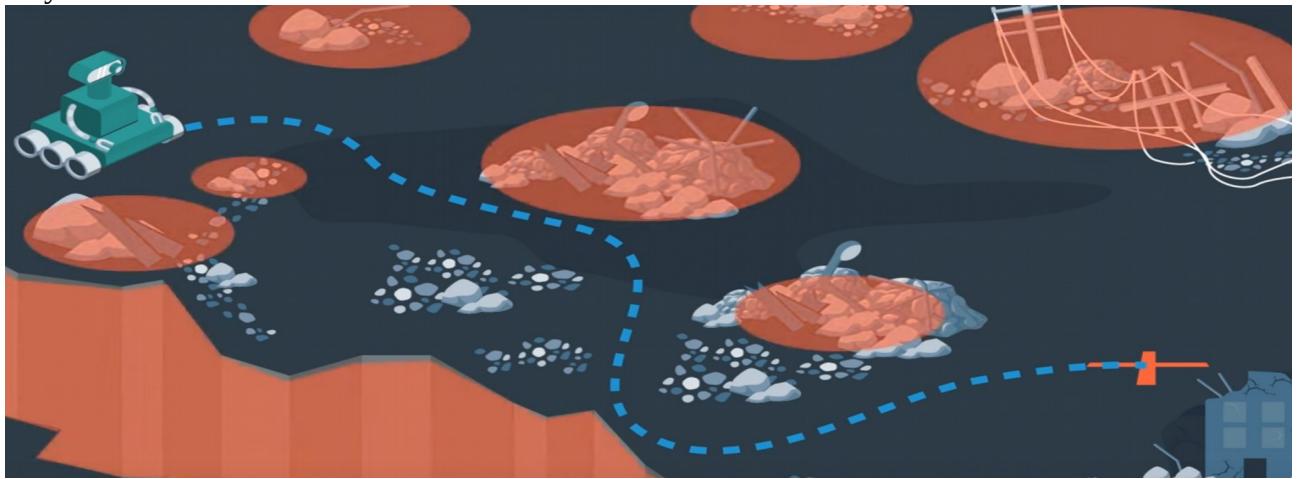
One option is to follow the shortest path, the straight line between a start and goal locations. However, due to the large amount of rubble present this is suboptimal or impossible.

One algorithm, that the robot can apply would have the rover traverse any encountered obstacles clockwise until it reaches its intended path once again.

**This algorithm is often referred to as the bug algorithm.**



An alternate route altogether would be to go around as much rubble as possible. To accomplish this, the path planning algorithm would need a way to evaluate how long it takes to traverse different types of land and to take this information into account when planning a path. The resulting path may look like this.



Although it is longer the rover would get to the goal location faster because it can move through flat terrain faster. More sophisticated algorithms may take into account the risk that the rover faces, avoiding unstable terrain or moving too closely to a cliff.

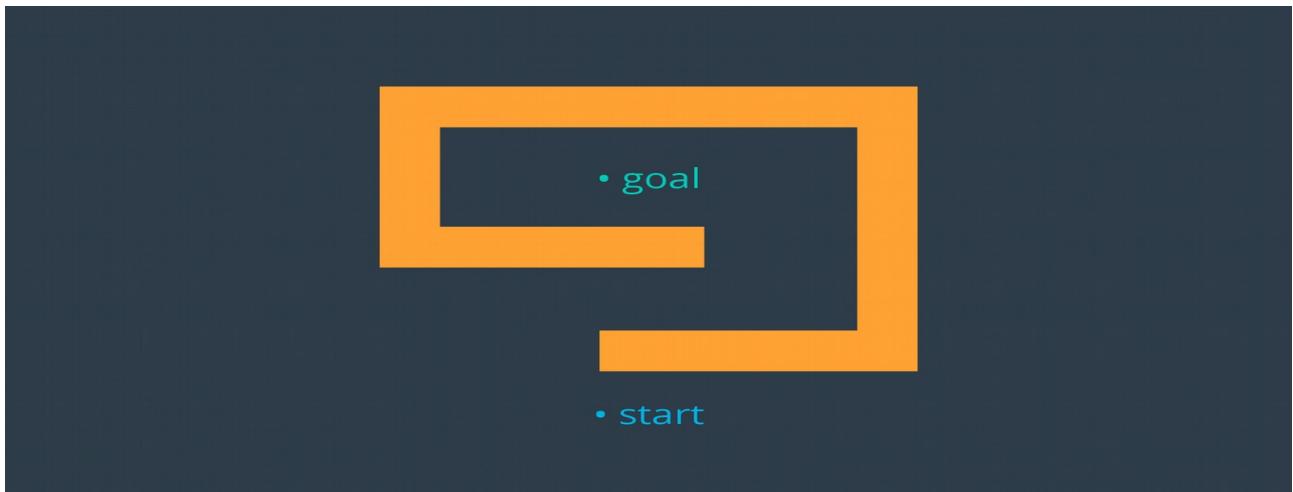
## Terminology

**Complete** - An algorithm is complete if it is able to find a path between the start and the goal when one exists.

**Optimal** - An algorithm is optimal if it is able to find the best solution.

**For example the bug algorithm is neither complete or optimal.**

The problem below will demonstrate one instance where a solution exists, but the bug algorithm is unable to find it.



In the above example, the robot would end up traversing the outer wall of the obstacle endlessly. There exist variants to the bug algorithm that will remedy this error, but the bulk of path planning algorithms rely on other principles that you will be introduced to throughout this chapter. In studying new algorithms, we will revisit the notion of Completeness and Optimality in analyzing the applicability of an algorithm to a task.

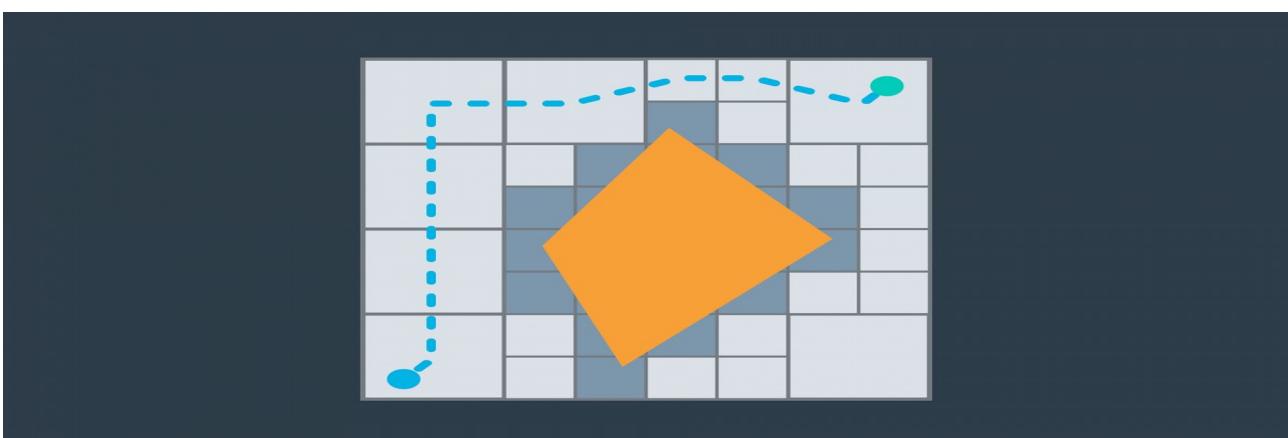
### 6.3.3 Approaches to Path Planning

In this chapter, you will be studying three different approaches to path planning. The first, called discrete (or combinatorial) path planning, is the most straightforward of the three approaches. The other two approaches, called sample-based path planning and probabilistic path planning, will build on the foundation of discrete planning to develop more widely applicable path planning solutions.

#### Discrete Planning

Discrete planning looks to explicitly discretize the robot's workspace into a connected graph, and apply a graph-search algorithm to calculate the best path. This procedure is very precise (in fact, the precision can be adjusted explicitly by changing how fine you choose to discretize the space) and very thorough, as it discretizes the *complete* workspace. As a result, discrete planning can be very computationally expensive - possibly prohibitively so for large path planning problems.

The image below displays one possible implementation of discrete path planning applied to a 2-dimensional workspace.



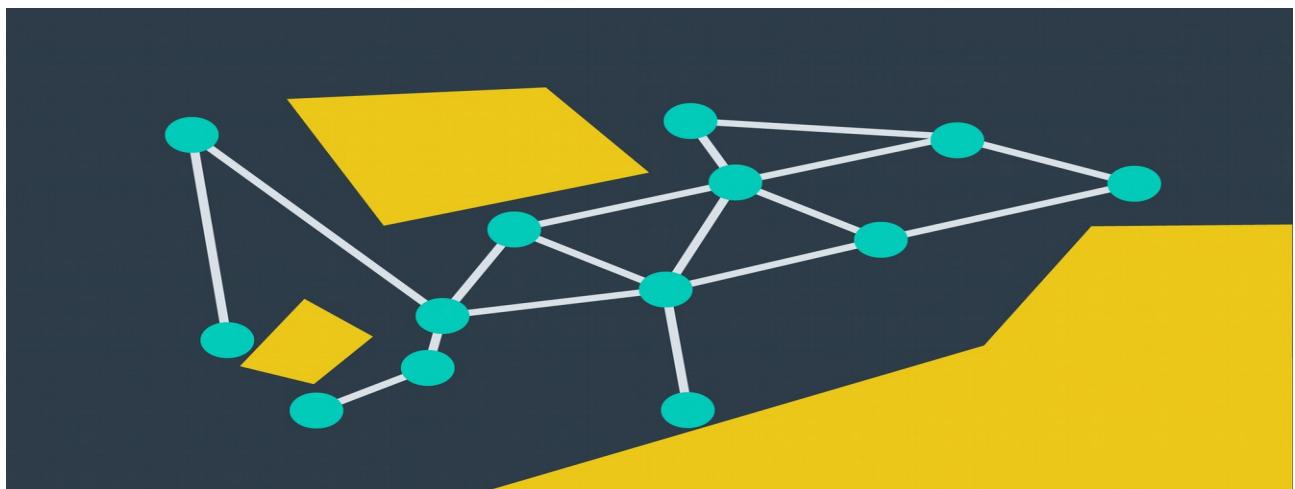
Discrete path planning is elegant in its precision, but is best suited for low-dimensional problems. For high-dimensional problems, sample-based path planning is a more appropriate approach.

## Sample-Based Planning

Sample-based path planning probes the workspace to incrementally construct a graph. Instead of discretizing *every* segment of the workspace, sample-based planning takes a number of samples and uses them to build a discrete representation of the workspace. The resultant graph is not as precise as one created using discrete planning, but it is much quicker to construct because of the relatively small number of samples used.

A path generated using sample-based planning may not be the *best* path, but in certain applications - it's better to generate a feasible path quickly than to wait hours or even days to generate the optimal path.

The image below displays a graph representation of a 2-dimensional workspace created using sample-based planning.

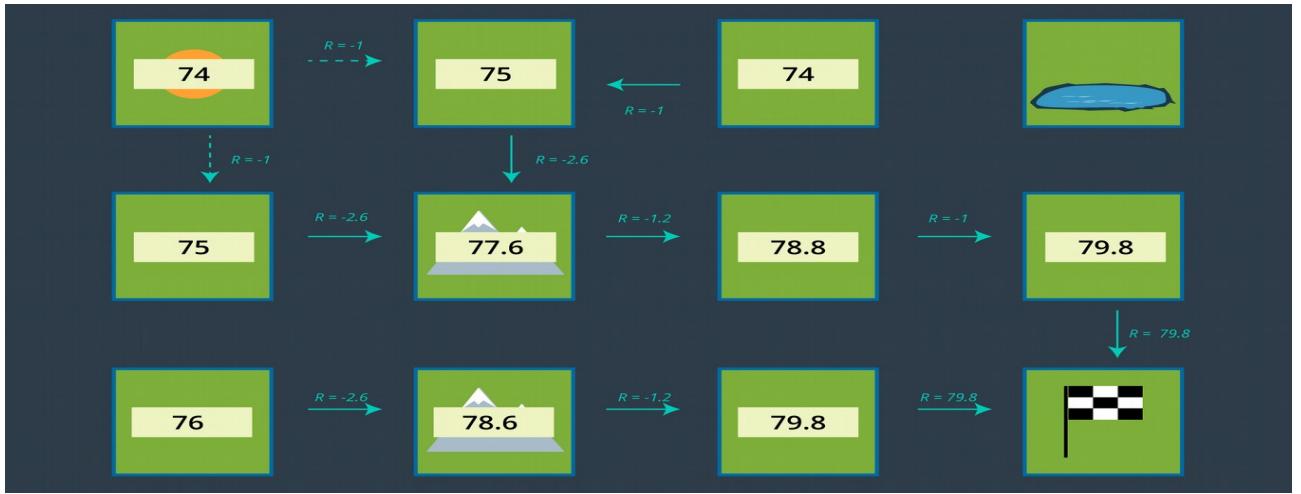


## Probabilistic Path Planning

The last type of path planning that you will learn about in this module is probabilistic path planning. While the first two approaches looked at the path planning problem generically - with no understanding of who or what may be executing the actions - probabilistic path planning takes into account the uncertainty of the robot's motion.

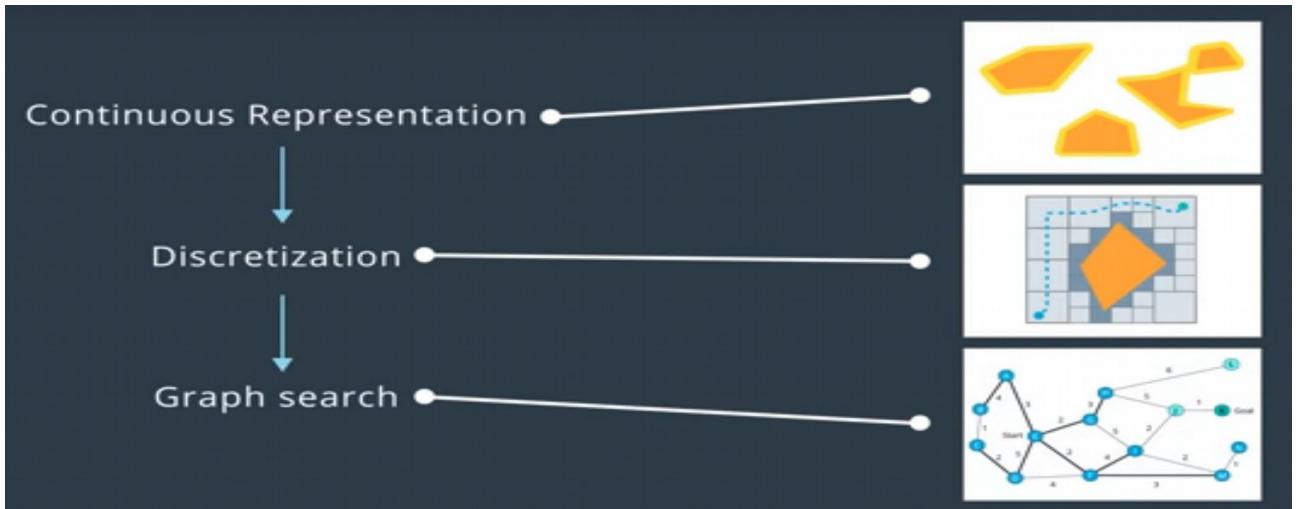
While this may not provide significant benefits in some environments, it is especially helpful in highly-constrained environment or environments with sensitive or high-risk areas.

The image below displays probabilistic path planning applied to an environment containing a hazard (the lake at the top right).



### 6.3.4 Discrete Planning

Solving the path planning problem through discrete planning, otherwise known as combinatorial planning, can be broken down into three distinct steps.



The first is to develop a convenient continuous representation, this can be done by representing the problem space as the configuration space also known as C space. The C space is an alternate way of representing the problem space. The C space takes into account the geometry of the robot and makes it easier to apply discrete search algorithms.

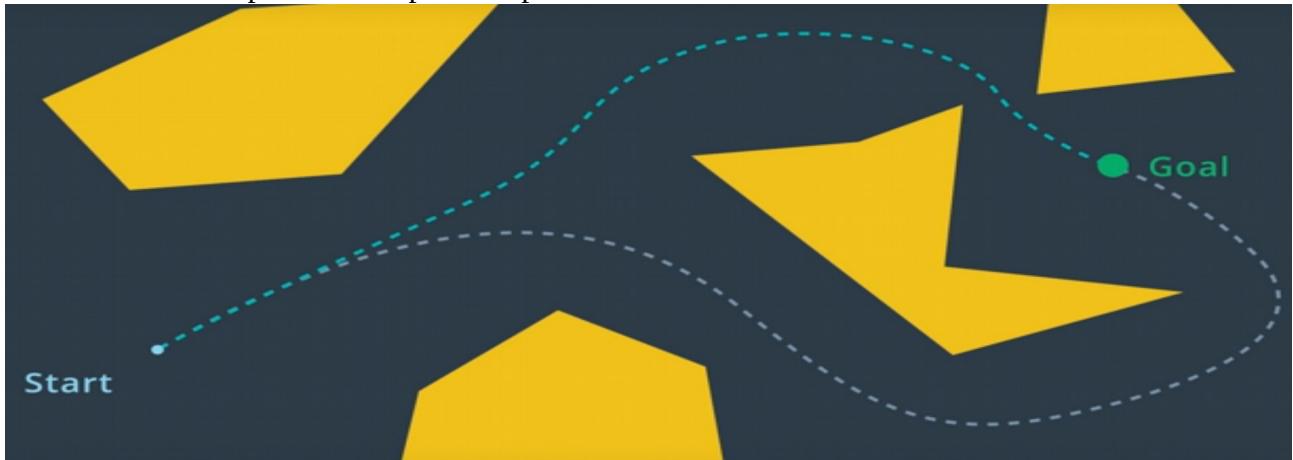
Next the C space must be discretized into a representation that is more easily manipulated by algorithms. The discretized space is represented by a graph.

Finally a search algorithm can be applied to the graph to find the best path from the start node to the goal node.

For each of these steps there is a variety of methods that can be applied to accomplish the desired outcomes, each with their own advantages and disadvantages.

### 6.3.5 Continuous Representation

If we treat the robot as a single point then the task of path planning is quite simple, within this workspace the robot can move anywhere in the free space. The robot can even travel along the wall of an obstacle. In such a case the path planning is relatively simple, find a curved or piece wise linear path connecting the robot start pose to the goal pose that does not collide with any obstacles. Here are two examples of such possible paths.



However, in reality robots have more dimensions than a point. If we model the robot as a 2-D disc, and try to attempt to follow the same paths as before we may run into trouble, some of the paths may have the robot collide with obstacles.



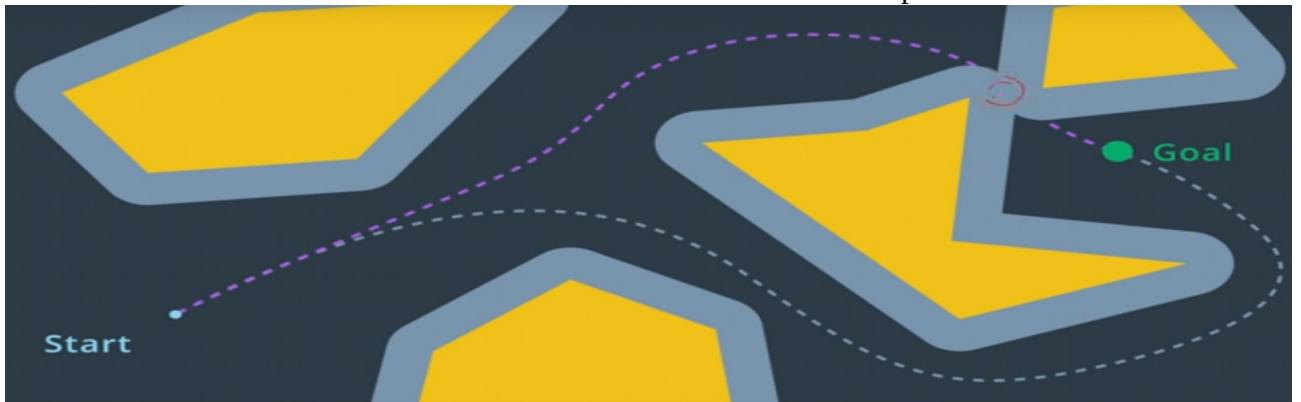
So, what do we do?

For every step of the path we could compute the distance from the robot center to every obstacle, and ensure that the space is greater than the radius of the robot.

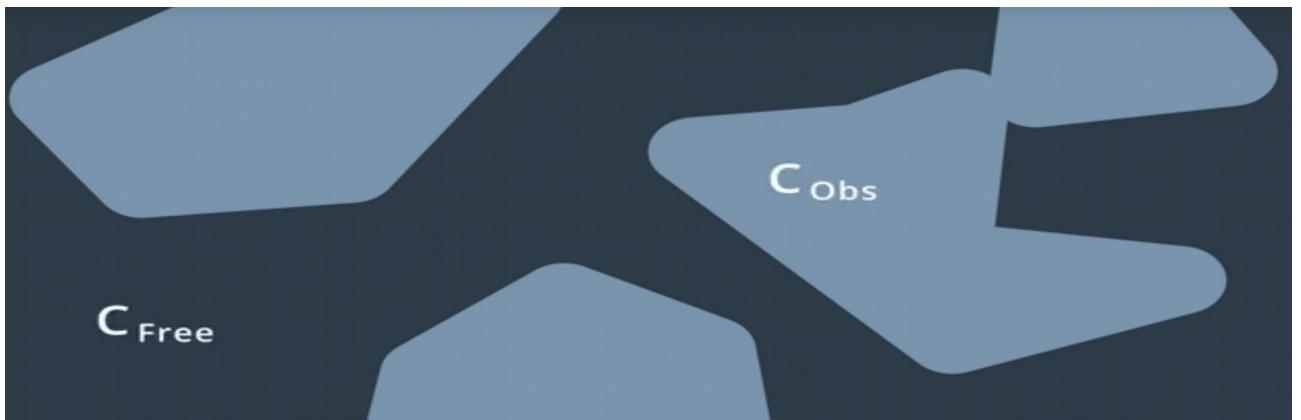


**But that would be a lot of work. The same can be accomplished in an easier manner.**

We can inflate every single obstacle by the radius of the robot, and then treat the robot as a point. Doing so, has shown us that the upper path is no longer an option for the robot of this size. The robot would not be able to fit in between the two obstacles. The other path is still viable.



This representation of the environment is called the configuration space or the C space for short. **A configuration space is a set of all robot poses. The C space is divided into C Free and C Obstacle.**

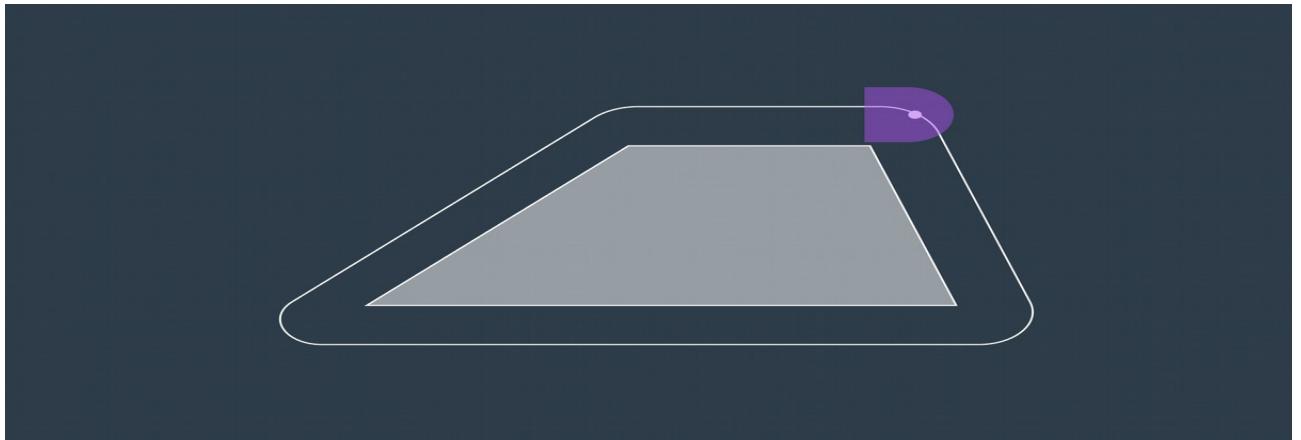


C Free represents the set of poses in the free space that do not collide with any obstacles, seen here in dark blue, and C obstacle is the compliment to C Free, representing the set of robot poses that are in collision with obstacles or walls, represented here in gray.

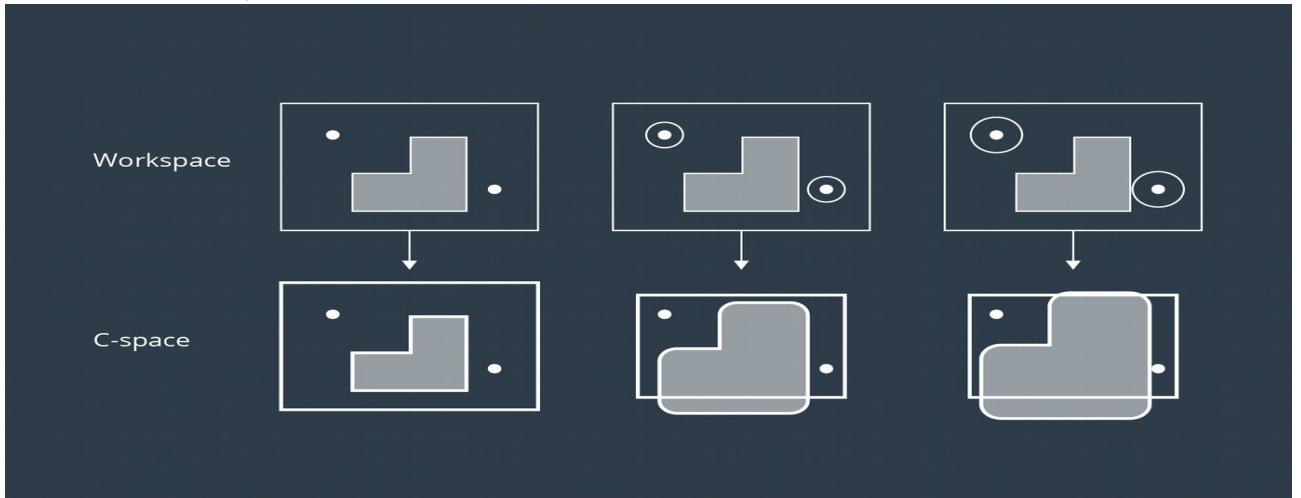
### 6.3.6 Minkowski Sum

The Minkowski sum is a mathematical property that can be used to compute the configuration space given an obstacle geometry and robot geometry.

The intuition behind how the Minkowski sum is calculated can be understood by imagining to paint the outside of an obstacle using a paintbrush that is shaped like your robot, with the robot's origin as the tip of the paintbrush. The painted area is Cobs. The image below shows just this.



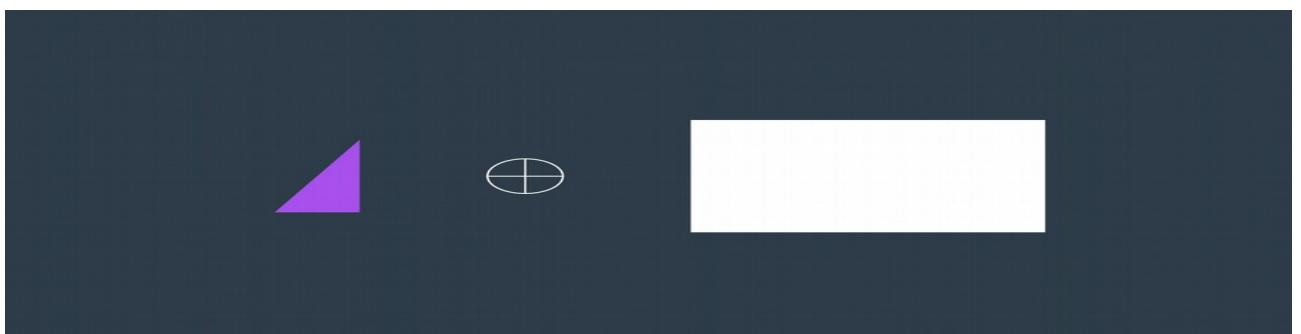
To create the configuration space, the Minkowski sum is calculated in such a way for every obstacle in the workspace. The image below shows three configuration spaces created from a single workspace with three different sized robots. As you can see, if the robot is just a dot, then the obstacles in the workspace are only inflated by a small amount to create the C-space. As the size of the robot increases, the obstacles are inflated more and more.



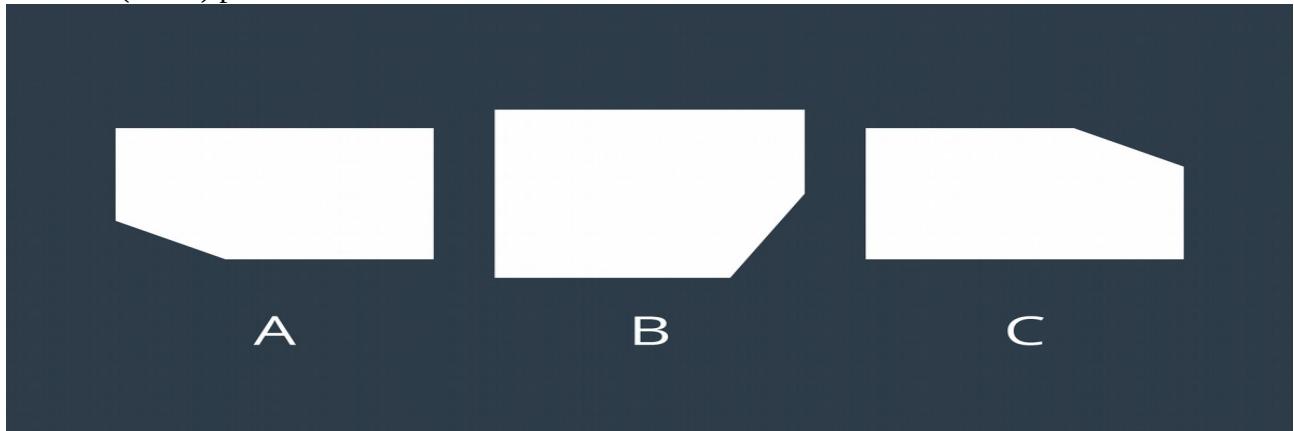
For convex polygons, computing the convolution is trivial and can be done in linear time - however for non-convex polygons (i.e. ones with gaps or holes present), the computation is much more expensive.

[A blog post on Minkowski sums and differences](#)

### Quiz: Minkowski Sum



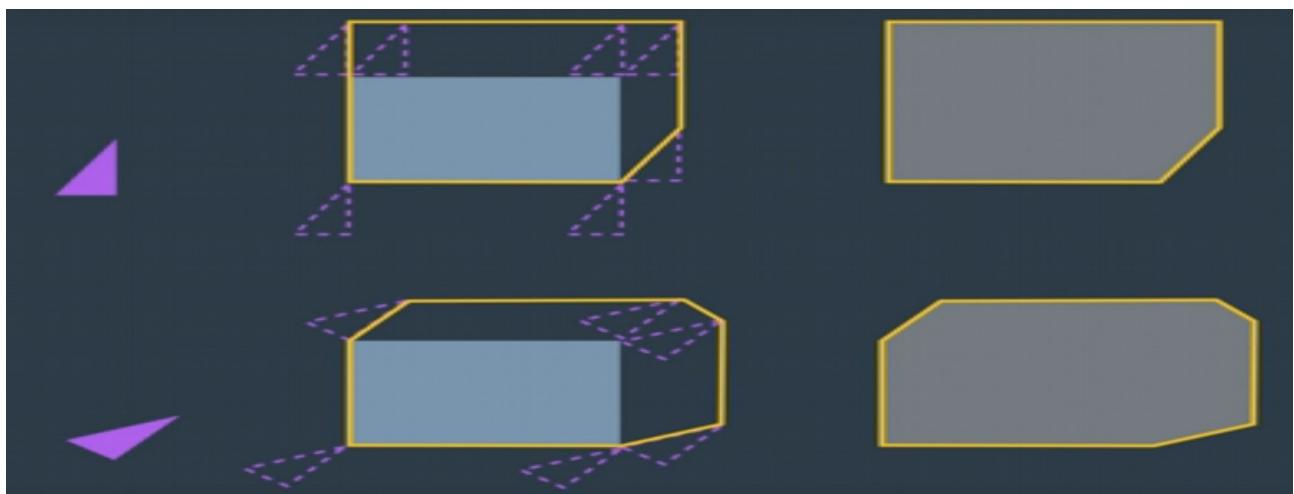
Which of the following images represents the Configuration Space for the robot (purple) and obstacle (white) presented above?



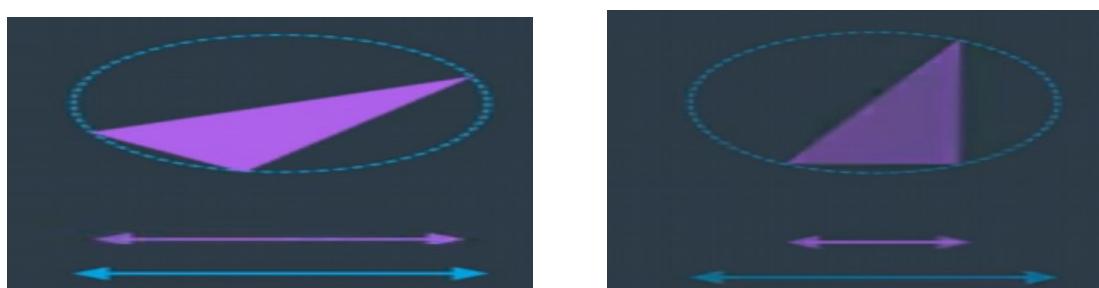
The correct answer is B.

### 6.3.7 Translation and Rotation

Now we're going to take things to the next level, before the robot was represented by a circle. **This was a very easy shape to use because its rotation did not affect the geometry of the configuration space.** If our robot was a triangle the configuration space of a triangular robot moving around a rectangular shape looks like so the upper section **while if we rotate the robot by say 38 degrees the configuration space changes depending on the orientation of the robot.**



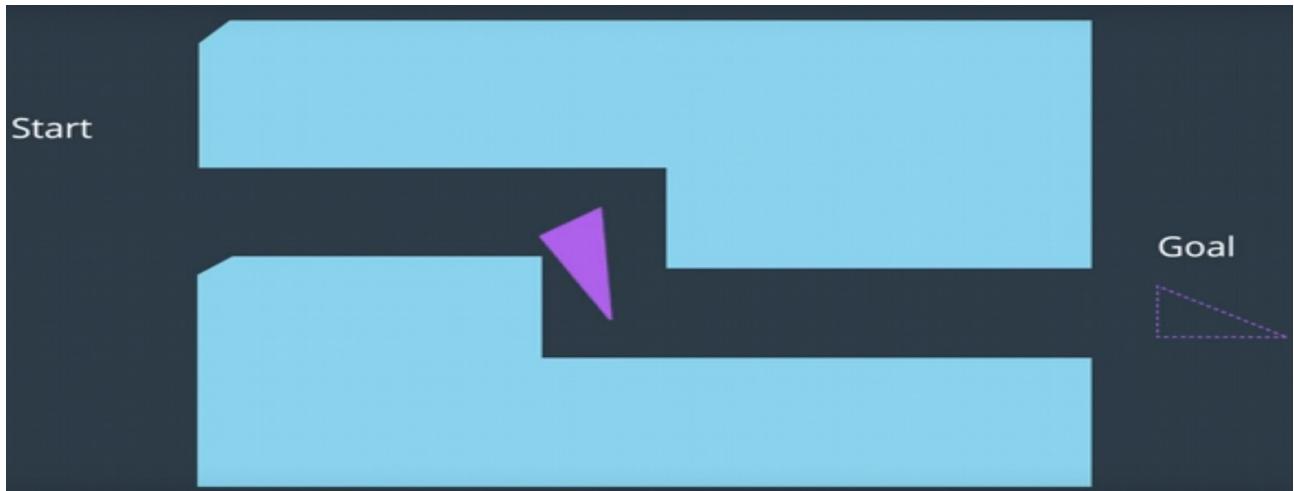
One way to standardize the configuration space for an odd shape robot, would be to enclose the robot in a bounding circle. **The circle represents the worst case scenario.** For some orientations of the robot it is a relatively accurate representation of the bounds of its vertices, but for other it may be a significant exaggeration.



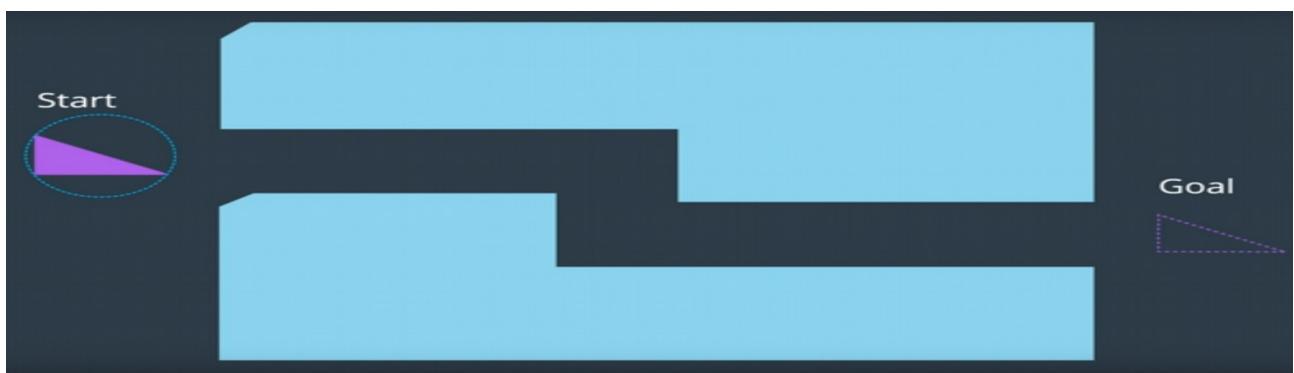
However, at all times the bounding circle is equal to or larger than the robot and its current configuration. So, if a path is found for the bounding circle, it will work for the robot.

This method is simple. **But it does come with a significant drawback. An algorithm applied to this generalization would not be complete.**

For example in this tight corridor the task is possible, and with two rotations the robot can navigate the corner and make it to the other end of the corridor.



However, if we were to enclose a triangular robot representation into a bounding circle, the algorithm would not be able to find a solution, since the circle's diameter is larger than the width of the corridor. Any algorithm applied here will return no solution found.



**So, while bounding circles can be an acceptable solution to some path planning problems, for instance ones in wide open environments, they are not a complete solution.**



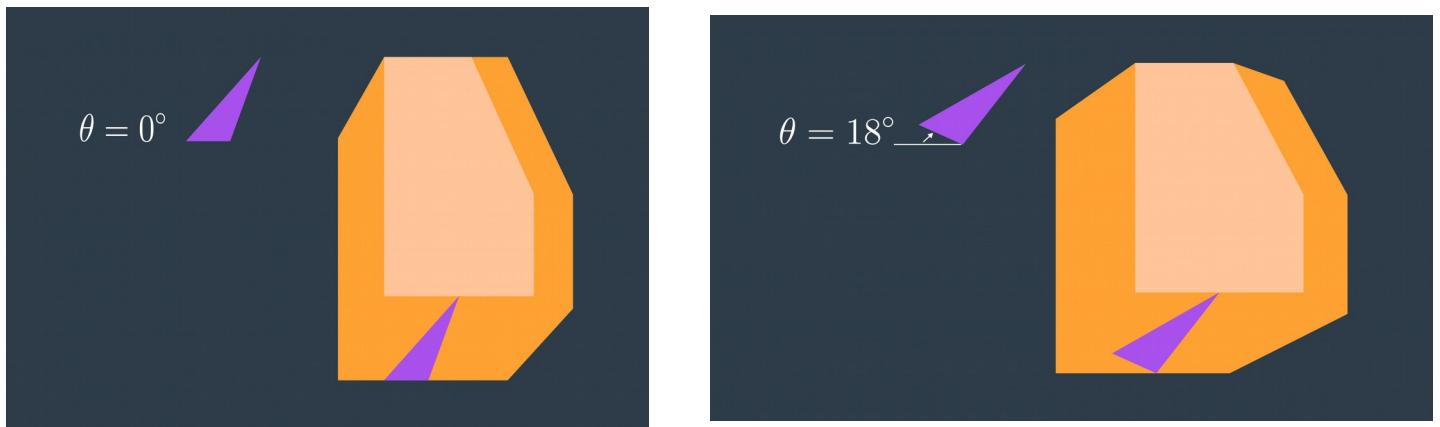
**Really, when you add the ability for the robot to rotate, you are adding a degree of freedom, the appropriate way to represent this in the configuration space is to add a dimension. The x-y plane would continue to represent the translation of the robot in the workspace, while the vertical axis would represent rotation of the robot.**

### 6.3.8 3D Configuration Space

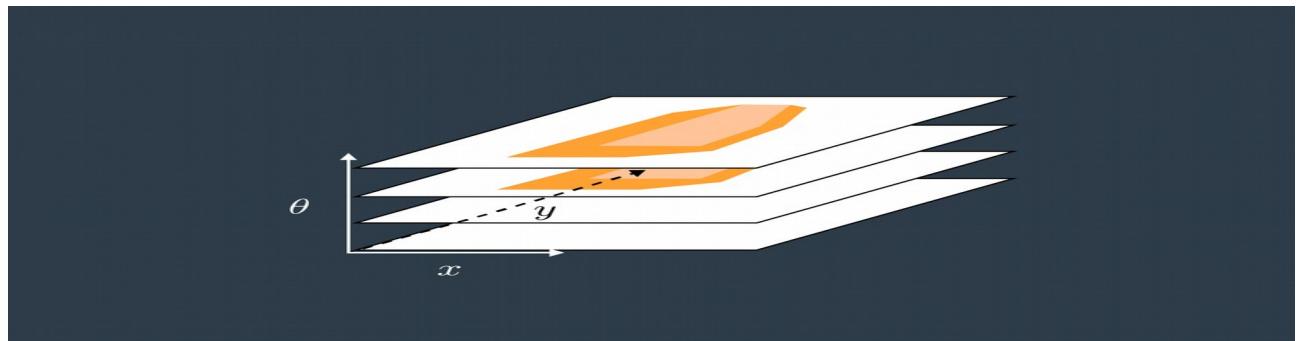
As you saw, the configuration space for a robot changes depending on its rotation. Allowing a robot to rotate adds a degree of freedom - so, sensibly, it complicates the configuration space as well. Luckily, this is actually very simple to handle. The dimension of the configuration space is equal to the number of degrees of freedom that the robot has.

While a 2D configuration space was able to represent the x- and y-translation of the robot, a third dimension is required to represent the rotation of the robot.

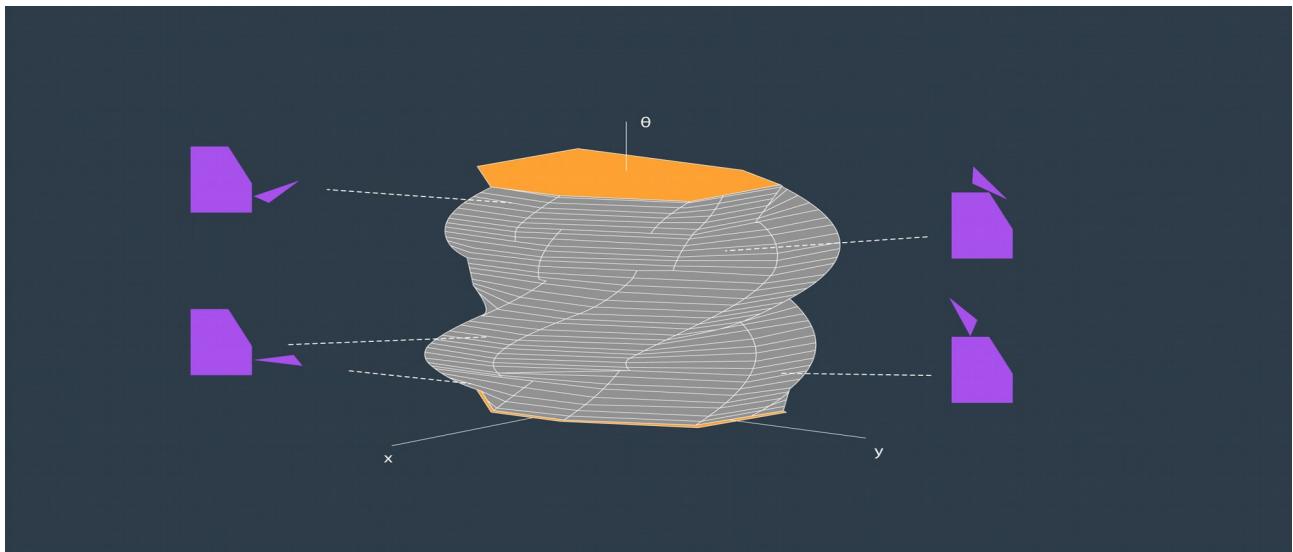
Let's look at a robot and its corresponding configuration space for two different rotations. The first will have the robot at  $0^\circ$ , and the second at  $18^\circ$ .



A three-dimensional configuration space can be generated by stacking two-dimensional configuration spaces as layers - as seen in the image below.



If we were to calculate the configuration spaces for infinitesimally small rotations of the robot, and stack them on top of each other - we would get something that looks like the image below.



The image above displays the configuration space for a triangular robot that is able to translate in two dimensions as well as rotate about its z-axis. While this image looks complicated to construct, there are a few tricks that can be used to generate 3D configuration spaces and move about them. The following video from the Freie Universität Berlin is a wonderful visualization of a 3D configuration space. The video will display different types of motion, and describe how certain robot motions map into the 3D configuration space.

[Configuration Space Visualization](#) - This is a *must* watch!

**Quiz:** When a robot rotates about one of it's boundary points (edges or vertices), how is this motion represented in the 3D configuration space?

**Answer: Helix**

### 6.3.9 Discretization

To be able to apply a search algorithm, the configuration space must be reduced to a finite size that an algorithm can traverse in a reasonable amount of time, as it searches for a path from start to the goal. This reduction in size can be accomplished by discretization. **Discretization is the process of breaking down a continuous entity, in this case a configuration space into discrete segments.** There are different methods that can be applied to discretize a continuous space.  
We will explore three different types of discretization



Each has its own advantages and disadvantages, **balancing trade-offs such as time and the level of detail**. Once we're equipped with that knowledge, we will explore **graph search** which can be applied to find a path from a start node to the goal node.

### 6.3.10 Roadmap

The first group of discretization approaches that we will learn is referred to by the name Roadmap. These methods represent the configuration space using a simple connected graph - similar to how a city can be represented by a metro map.



Roadmap methods are typically implemented in two phases:

- The **construction phase** builds up a graph from a continuous representation of the space. This phase usually takes a significant amount of time and effort, but the resultant graph can be used for multiple queries with minimal modifications.
- The **query phase** evaluates the graph to find a path from a start location to a goal location. This is done with the help of a search algorithm.

In this Discretization section, we will only discuss and evaluate the construction phase of each Roadmap method. Whereas the query phase will be discussed in more detail in the Graph Search section, following Discretization.

The two roadmap methods that you will learn next are the Visibility Graph, and Voronoi Diagram methods.

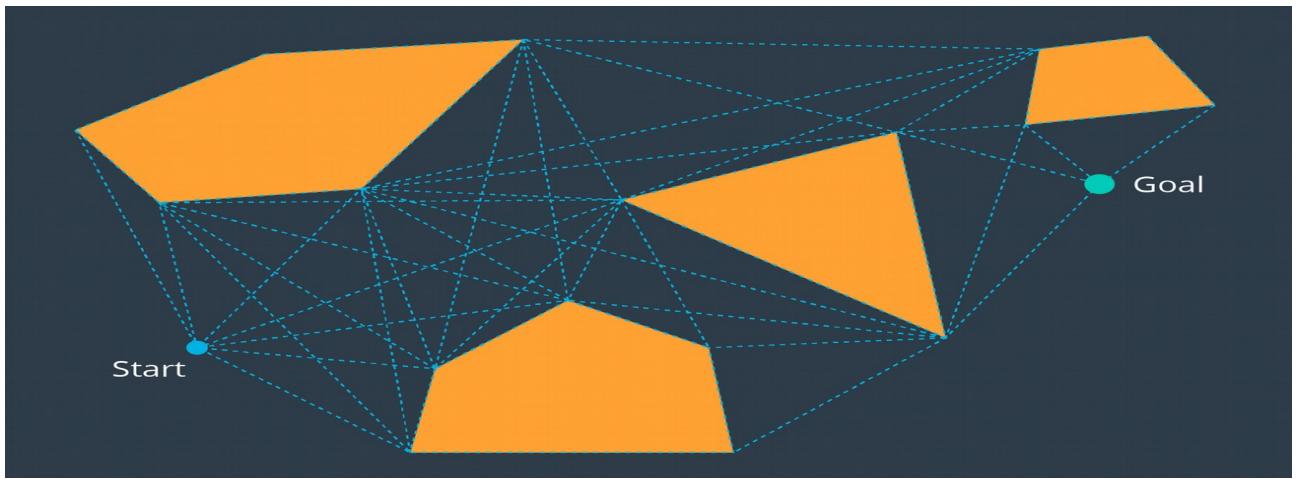
### 6.3.11 Visibility Graph

The Visibility Graph builds a roadmap by connecting the start node, all of the obstacles' vertices, and goal node to each other - except those that would result in collisions with obstacles. The Visibility Graph has its name for a reason - it connects every node to all other nodes that are 'visible' from its location.

**Nodes:** Start, Goal, and all obstacle vertices.

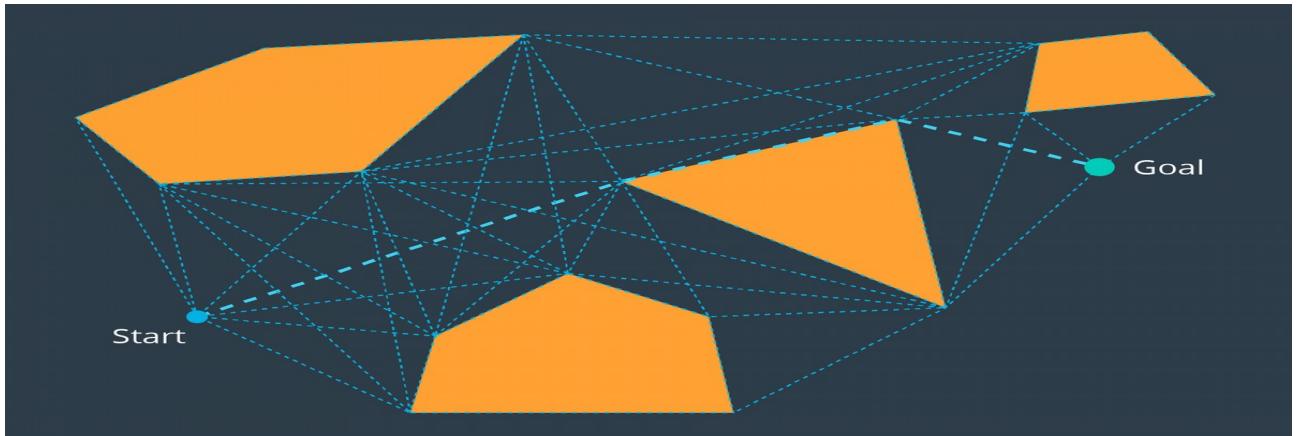
**Edges:** An edge between two nodes that does not intersect an obstacle, including obstacle edges.

The following image illustrates a visibility graph for a configuration space containing polygonal obstacles.



The motivation for building Visibility Graphs is that the shortest path from the start node to the goal node will be a piecewise linear path that bends only at the obstacles' vertices. This makes sense intuitively - the path would want to hug the obstacles' corners as tightly as possible, as not to add any additional length.

Once the Visibility Graph is built, a search algorithm can be applied to find the shortest path from Start to Goal. The image below displays the shortest path in this visibility graph.



## Quiz

Although the algorithms used to search the roadmap have not yet been introduced - it is still worth analysing whether *any* algorithm would be able to find a path from start to goal, and whether the optimal path lies within the roadmap.

### Quiz Question

Is the visibility graph complete? Does it contain the optimal path?

**It is Complete and Optimal.**

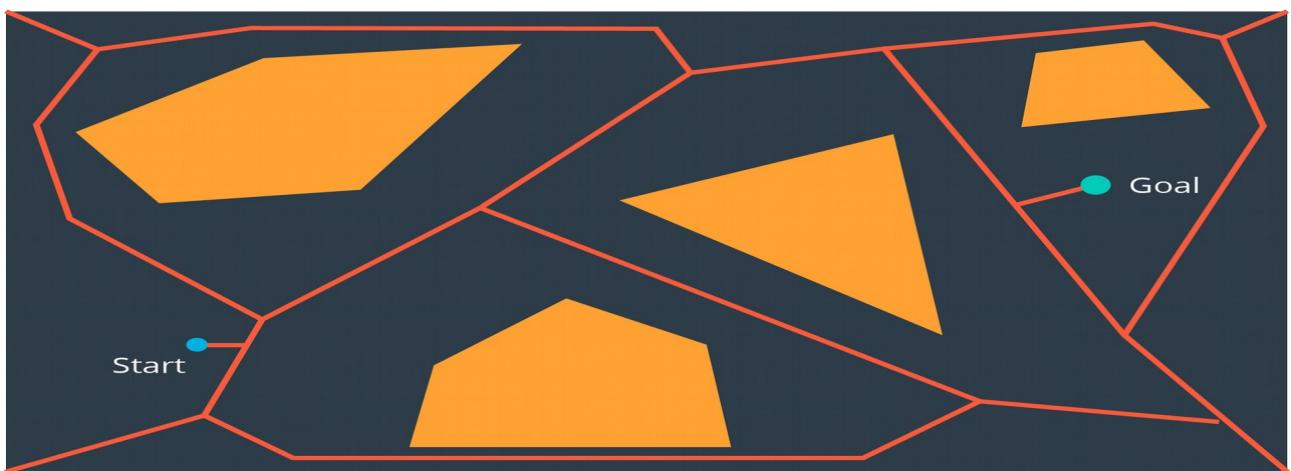
Having completed the quiz, you should have by now seen the advantages of the Visibility Graph method. **One disadvantage to the Visibility Graph is that it leaves no clearance for error.** A

robot traversing the optimal path would have to pass incredibly close to obstacles, increasing the risk of collision significantly. In certain applications, such as animation or path planning for video games, this is acceptable. However the uncertainty of real-world robot localization makes this method impractical.

### 6.3.12 Voronoi Diagram

Another discretization method that generates a roadmap is called the Voronoi Diagram. Unlike the visibility graph method which generates the shortest paths, Voronoi Diagrams maximize clearance between obstacles.

A Voronoi Diagram is a graph whose edges bisect the free space in between obstacles. Every edge lies equidistant from each obstacle around it - with the greatest amount of clearance possible. You can see a Voronoi Diagram for our configuration space in the graphic below.



Once a Voronoi Diagram is constructed for a workspace, it can be used for multiple queries. Start and goal nodes can be connected into the graph by constructing the paths from the nodes to the edge closest to each of them.

Every edge will either be a straight line, if it lies between the edges of two obstacles, or it will be a quadratic, if it passes by the vertex of an obstacle.

## Quiz

Once again, it is worth investigating - will the roadmap built by the voronoi diagram contain a path from start to goal, and will it contain the optimal path.

Quiz Question

Is the Voronoi Diagram complete? Does it contain the optimal path?

**It is only Complete.**

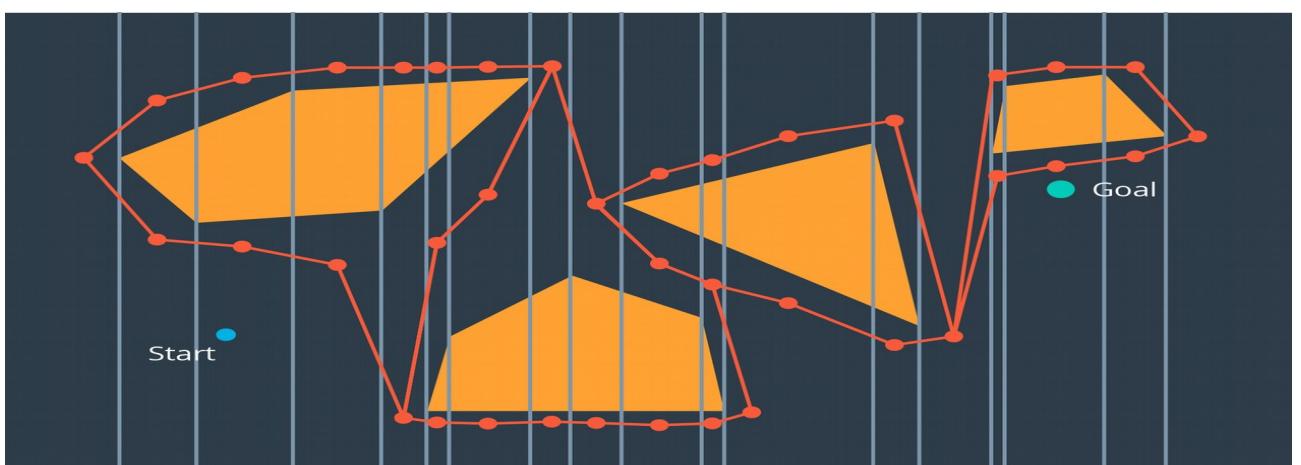
### 6.3.13 Cell Decomposition

Another discretization method that can be used to convert a configuration space into a representation that can easily be explored by a search algorithm is cell decomposition. Cell decomposition divides the space into discrete cells, where each cell is a node and adjacent cells are connected with edges. There are two distinct types of cell decomposition:

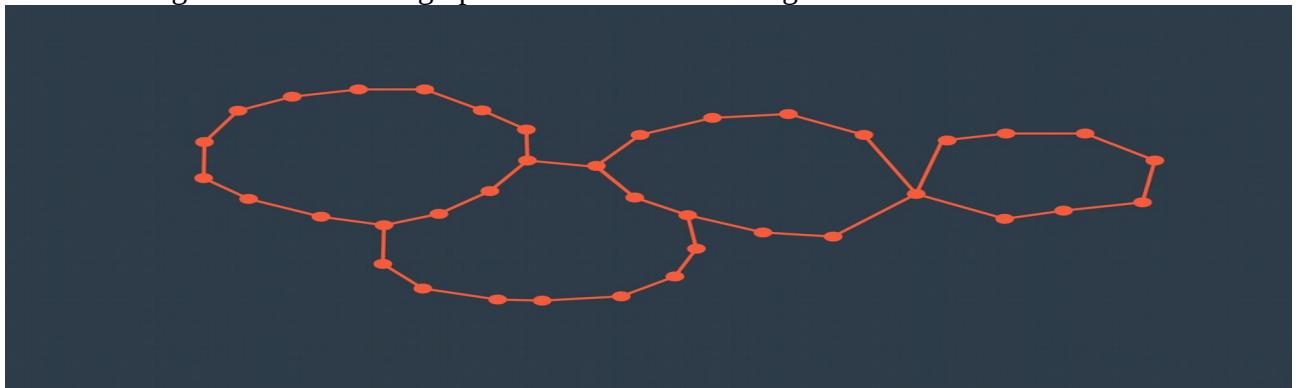
- Exact Cell Decomposition
- Approximate Cell Decomposition.

## Exact Cell Decomposition

Exact cell decomposition divides the space into *non-overlapping* cells. This is commonly done by breaking up the space into triangles and trapezoids, which can be accomplished by adding vertical line segments at every obstacle's vertex. You can see the result of exact cell decomposition of a configuration space in the image below.



Once a space has been decomposed, the resultant graph can be used to search for the shortest path from start to goal. The resultant graph can be seen in the image below.



Exact cell decomposition is elegant because of its precision and completeness. Every cell is either ‘full’, meaning it is completely occupied by an obstacle, or it is ‘empty’, meaning it is free. And the union of all cells exactly represents the configuration space. If a path exists from start to goal, the resultant graph *will* contain it.

To implement exact cell decomposition, the algorithm must order all obstacle vertices along the x-axis, and then for every vertex determine whether a new cell must be created or whether two cells should be merged together. Such an algorithm is called the Plane Sweep algorithm.

Exact cell decomposition results in cells of awkward shapes. Collections of uniquely-shaped trapezoids and triangles are more difficult to work with than a regular rectangular grid. This results in an added computational complexity, especially for environments with greater numbers of dimensions. It is also difficult to compute the decomposition when obstacles are not polygonal, but of an irregular shape.

For this reason, there is an alternate type of cell decomposition, that is much more practical in its implementation.

### 6.3.14 Approximate Cell Decomposition

#### Approximate Cell Decomposition

Approximate cell decomposition divides a configuration space into discrete cells of simple, regular shapes - such as rectangles and squares (or their multidimensional equivalents). Aside from simplifying the computation of the cells, this method also supports hierarchical decomposition of space (more on this below).

#### Simple Decomposition

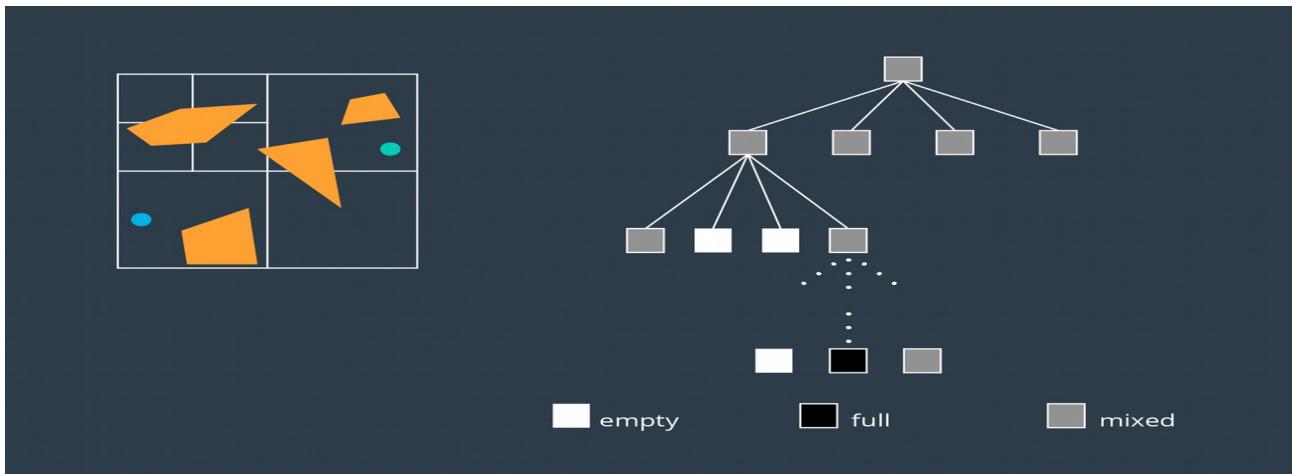
A 2-dimensional configuration space can be decomposed into a grid of rectangular cells. Then, each cell could be marked full or empty, as before. A search algorithm can then look for a sequence of free cells to connect the start node to the goal node.

Such a method is more efficient than exact cell decomposition, **but it loses its completeness**. It is possible that a particular configuration space contains a feasible path, but the resolution of the cells results in some of the cells encompassing the path to be marked ‘full’ due to the presence of obstacles. A cell will be marked ‘full’ whether 99% of the space is occupied by an obstacle or a mere 1%. Evidently, this is not practical.

#### Iterative Decomposition

An alternate method of partitioning a space into simple cells exists. Instead of immediately decomposing the space into *small* cells of equal size, the method *recursively* decomposes a space into four quadrants. Each quadrant is marked full, empty, **or a new label called ‘mixed’** - used to represent cells that are somewhat occupied by an obstacle, but also contain some free space. If a **sequence of free cells cannot be found from start to goal, then the mixed cells will be further decomposed into another four quadrants**. Through this process, more free cells will emerge, eventually revealing a path if one exists.

The 2-dimensional implementation of this method is called quadtree decomposition. It can be seen in the graphic below.



## Algorithm

The algorithm behind approximate cell decomposition is much simpler than the exact cell decomposition algorithm. The pseudocode for the algorithm is provided below.

Decompose the configuration space into four cells, label cells free, mixed, or full.

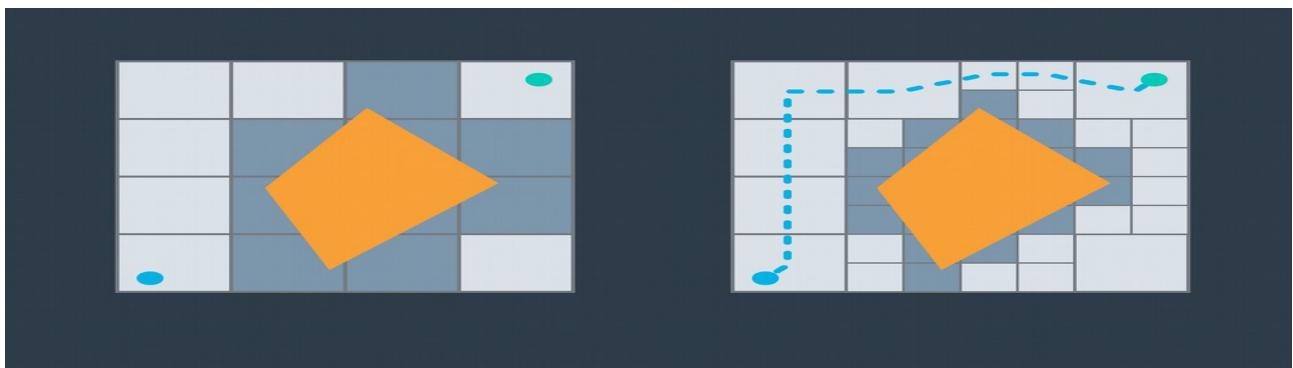
Search for a sequence of free cells that connect the start node to the goal node.

If such a sequence exists:

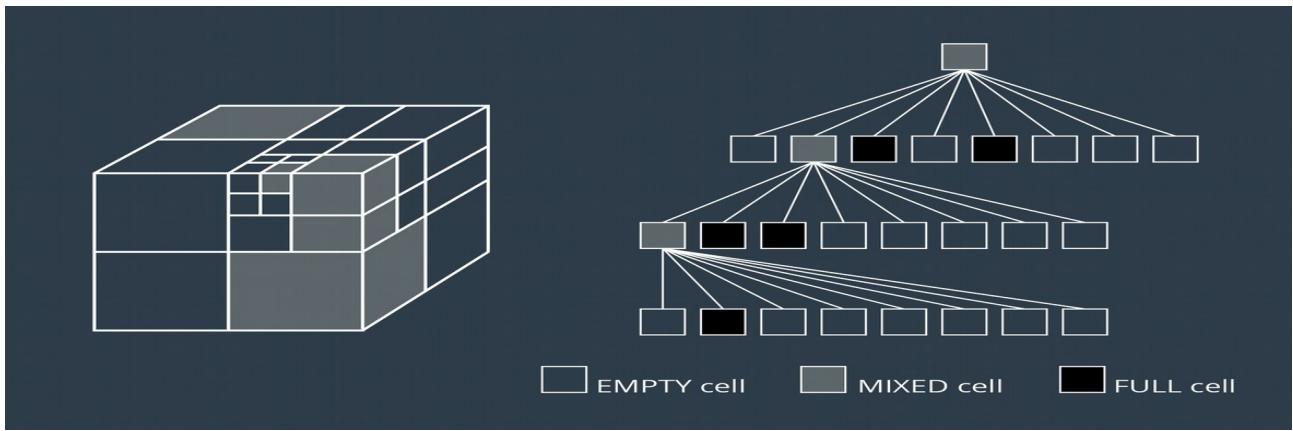
Return path

Else:

Further decompose the mixed cells



**The three dimensional equivalent of quadtrees are octrees, depicted in the image below.** The method of discretizing a space with any number of dimensions follows the same procedure as the algorithm described above, but expanded to accommodate the additional dimensions.



Although exact cell decomposition is a more elegant method, it is much more computationally expensive than approximate cell decomposition for non-trivial environments. **For this reason, approximate cell decomposition is commonly used in practice.**

**With enough computation, approximate cell decomposition approaches completeness.**

However, **it is not optimal** - the resultant path depends on how cells are decomposed. Approximate cell decomposition finds the obvious solution quickly. It is possible that the optimal path squeezes through a minuscule opening between obstacles, but the resultant path takes a much longer route through wide open spaces - **one that the recursively-decomposing algorithms would find first.**

Approximate cell decomposition is functional, but like all discrete/combinatorial path planning methods - **it starts to be computationally intractable for use with high-dimensional environments.**

- In practice, approximate cell decomposition is preferred due to its more manageable computation.
- Approximate cell decomposition is not optimal because obvious (wide/open) paths are found first.
- The quadtree and octree methods recursively decompose mixed cells until they find a sequence of free cells from start to goal.

### 6.3.15 Potential Field

Unlike the methods discussed thus far that discretize the continuous space into a connected graph, the potential field method performs a different type of discretization.

**To accomplish its task, the potential field method generates two functions - one that attracts the robot to the goal and one that repels the robot away from obstacles.** The two functions can be summed to create a discretized representation. By applying an optimization algorithm such as gradient descent, a robot can move toward the goal configuration while steering around obstacles. Let's look at how each of these steps is implemented in more detail.

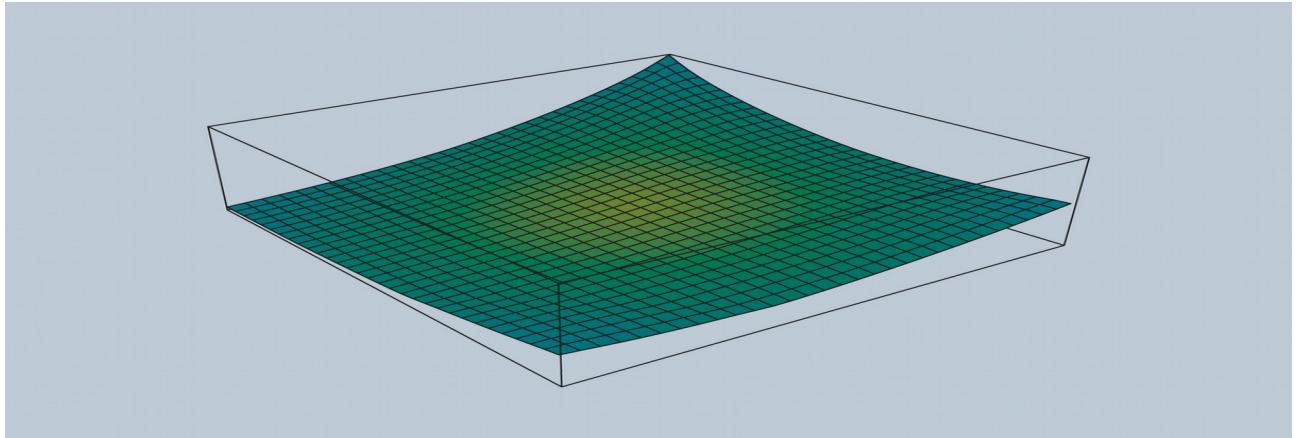
#### Attractive Potential Field

The attractive potential field is a function with the global minimum at the goal configuration. If a robot is placed at any point and required to follow the direction of steepest descent, it will end up at the goal configuration. This function does not need to be complicated, a simple quadratic function can achieve all of the requirements discussed above.

$$f_{att}(\mathbf{x}) = \nu_{att}(||\mathbf{x} - \mathbf{x}_{goal}||)^2$$

**Where  $\mathbf{x}$  represents the robot's current position, and  $\mathbf{x}_{goal}$  the goal position.  $\nu$  is a scaling factor.**

A fragment of the attractive potential field is displayed in the image below.



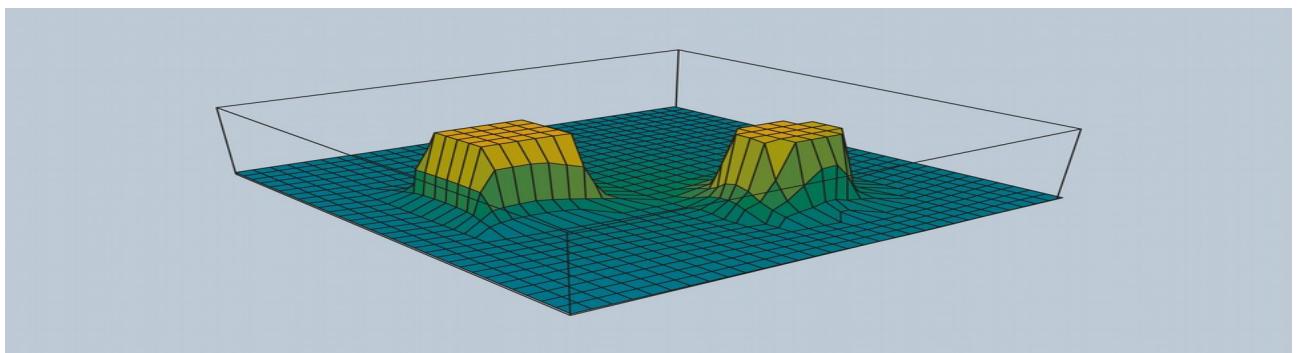
## Repulsive Potential Field

The repulsive potential field is a function that is equal to zero in free space, and grows to a large value near obstacles. One way to create such a potential field is with the function below.

$$f_{rep} = \begin{cases} \nu_{rep} \left( \frac{1}{\rho(\mathbf{x})} - \frac{1}{\rho_0} \right)^2 & \text{if } \rho \leq \rho_0, \\ 0 & \text{if } \rho > \rho_0 \end{cases}$$

Where the function  $\rho(\mathbf{x})$  returns the distance from the robot to its nearest obstacle,  $\rho_0$  is a scaling parameter that defines the reach of an obstacle's repulsiveness, and  $\nu$  is a scaling parameter.

An image of a repulsive potential field for an arbitrary configuration space is provided below.

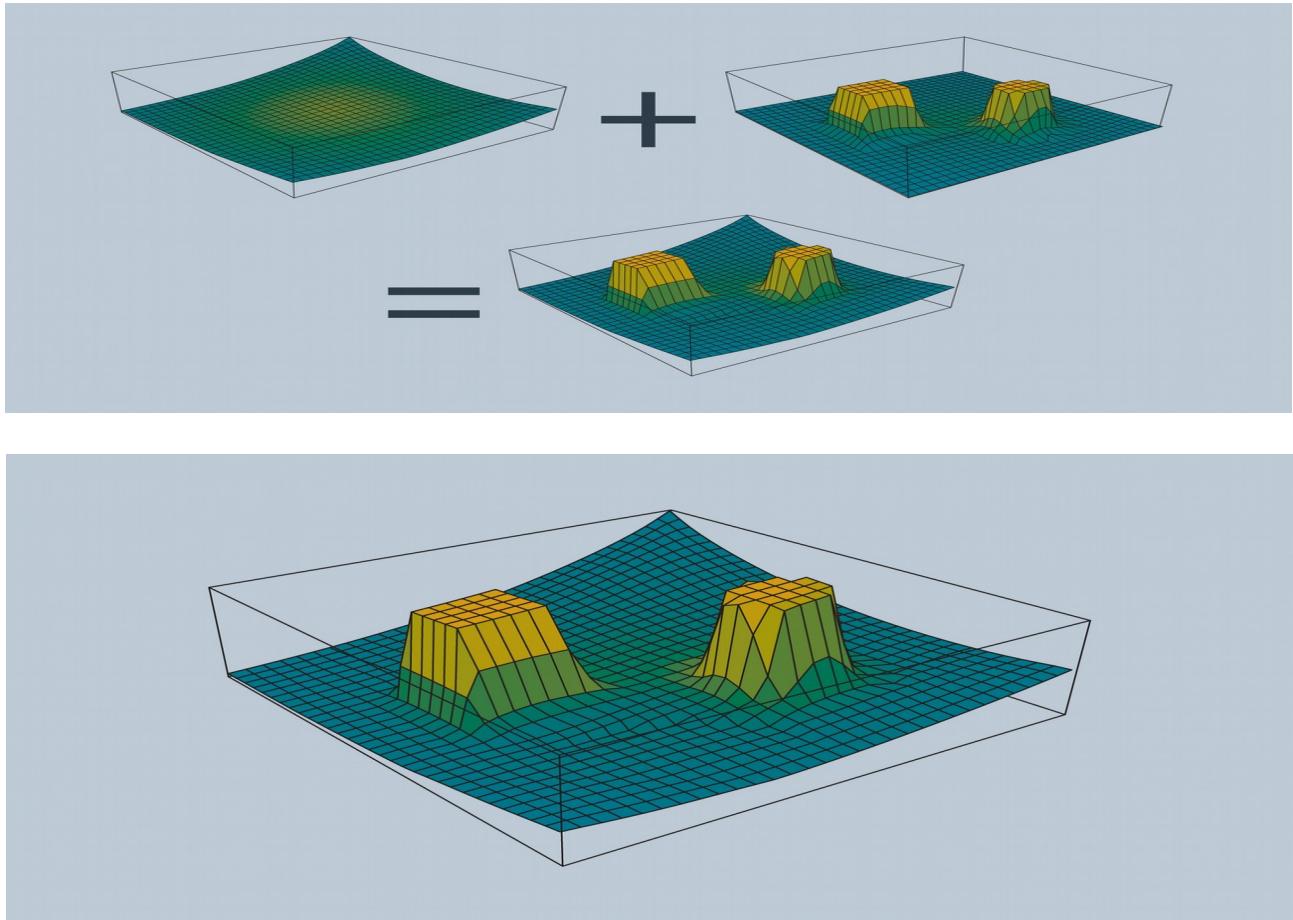


The value  $\rho_0$  controls how far from an obstacle the potential field will be non-zero, and how steep the area surrounding an obstacle will be.

Past  $\rho_0$ , the potential field is zero. Within a  $\rho_0$  distance of the obstacle, the potential field scales with proximity to the obstacle.

## Potential Field Sum

The attractive and repulsive functions are summed to produce the potential field that is used to guide the robot from anywhere in the space to the goal. The image below shows the summation of the functions, and the image immediately after displays the final function.



Imagine placing a marble onto the surface of the function - from anywhere in the field it will roll in the direction of the goal without colliding with any of the obstacles (as long as  $\rho_0$  is set appropriately)!

The gradient of the function dictates which direction the robot should move, and the speed can be set to be constant or scaled in relation to the distance between the robot and the goal.

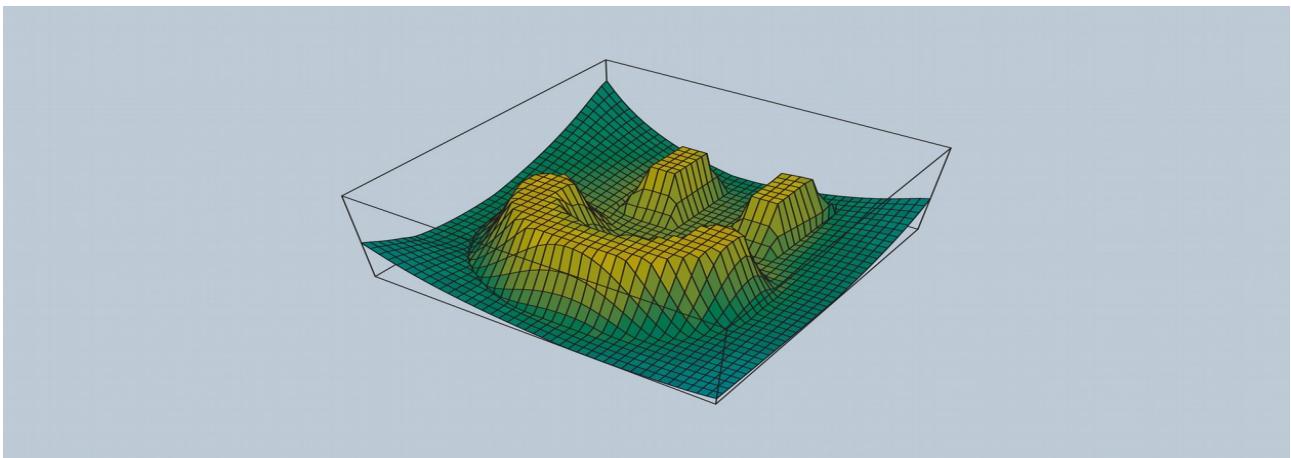
## Problems with the Potential Field Method

The potential field method is not without its faults - the method is neither complete nor optimal. In certain environments, the method will lead the robot to a **local minimum**, as opposed to the global minimum. The images below depict one such instance. Depending on where the robot commences, it may be led to the bottom of the smile.

The image below depicts the configuration space, and the following image displays the corresponding potential field.



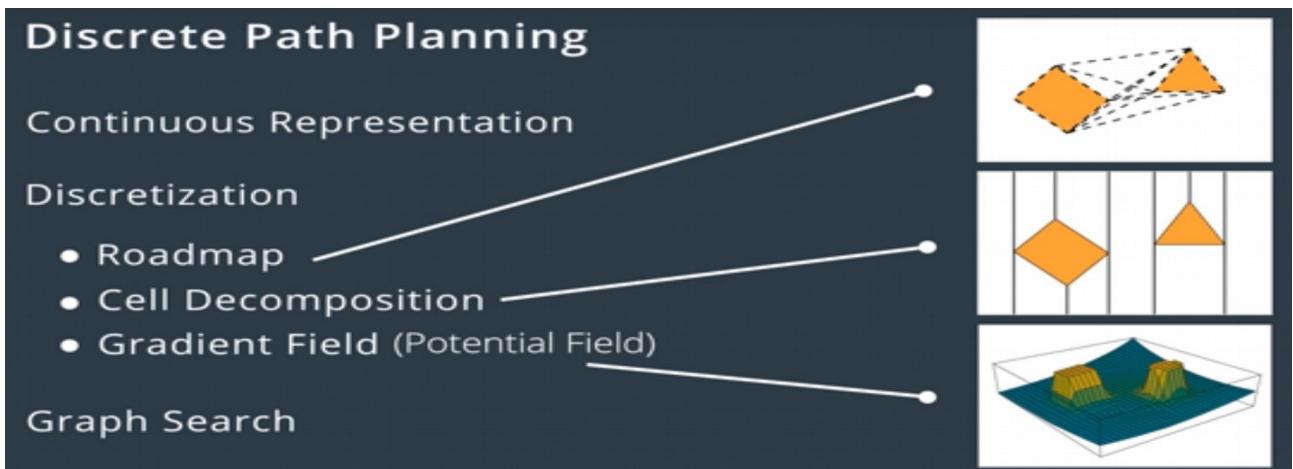
● Goal



The problem of a robot becoming stuck in a local minimum can be resolved by adding random walks, and other strategies that are commonly applied to gradient descent, but ultimately the method is not complete.

The potential field method isn't optimal either, as it may not always find the shortest (or cheapest) path from start to goal. The shortest path may not follow the path of steepest descent. In addition, potential field does not take into consideration the cost of every step.

### 6.3.16 Discretization Wrap-up



In continuous representation, we learned how to create a configuration space, and in our study of discretization, we learned about three different types of methods that can be used to represent a configuration space with discrete segments.

- **Roadmap:** Here, we modeled the configuration space as a simple graph by either connecting the vertices of the obstacles or building a Voronoi diagram.
- **Cell decomposition:** broke the space into a finite number of cells, each of which was assessed to be empty, full, or mixed. The empty cells were then linked together to create a graph.
- **Gradient Field:** Is a method that models the configuration space using a 3D function that has the goal as global minimum and obstacles as tall structures

**Now, most of these methods left us with a graph representation of the space.**

**Next on the section of graph search we will learn how to traverse the graph to find the best path for our robot.**

### 6.3.17 Graph Search

Great. We're on to the last step of discrete planning, graph search. Graph search is used to find a finite sequence of discrete actions to connect a start state to a goal state. It does so by searching, visiting states sequentially asking every state, "Hey, are you the goal state?"

As a human, you may find it easy to pick out a solution to a search problem when the search problem is 2-D and manageable in size. For instance this one. However a computer must conduct the search manually, it goes through node by node and doesn't see and doesn't see the goal node until it is one node away, like so.



As the size of the space grows and dimensions are added, the problem naturally becomes less trivial. It becomes imperative to choose the appropriate algorithm for the task to achieve satisfactory results.

**There are two types of search algorithms Informed and Uninformed**

#### Uninformed vs Informed Search

Uninformed search algorithms are not provided with any information about the whereabouts of the goal, and thus search blindly. The only difference between different uninformed algorithms is the order in which they expand nodes. Several different types of uninformed algorithms are listed below:

- Breadth-first Search

- Depth-first Search
- Uniform Cost Search

Informed searches, on the other hand, are provided with information pertaining to the location of the goal. As a result, these search algorithms are able to evaluate some nodes to be more promising than others. This makes their search more efficient. The informed algorithm that you will be learning in this lesson is,

- A\* Search

Several variations on the above searches exist, and will be briefly discussed.

**In the appendix we will see how to program the A\* algorithm in C++**

## Terminology

You are already familiar with two terms that can be used to describe an algorithm - completeness and optimality. However, there are a few others that you should know before starting to learn individual graph search algorithms.

The **time complexity** of an algorithm assesses how long it takes an algorithm to generate a path, usually with respect to the number of nodes or dimensions present. It can also refer to the trade-off between quality of an algorithm (ex. completeness) vs its computation time.

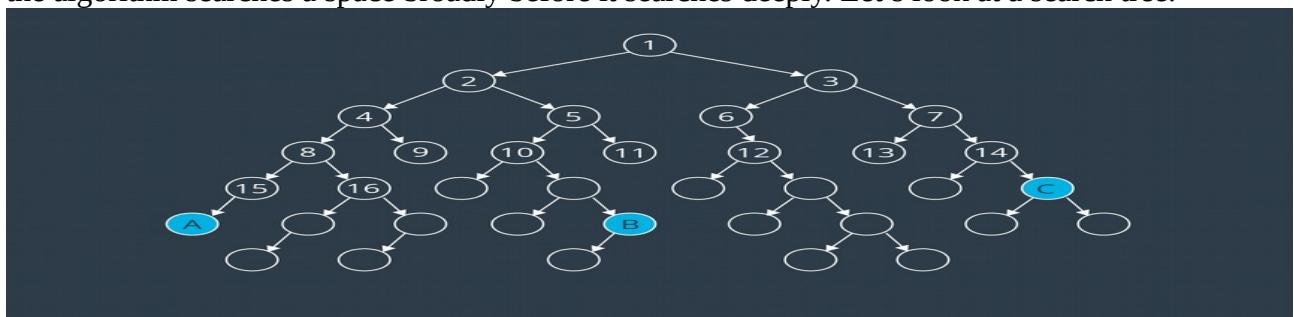
The **space complexity** of an algorithm assesses how much memory is required to execute the search. Some algorithms must keep significant amounts of information in memory throughout their run-time, while others can get away with very little.

The **generality** of an algorithm considers the type of problems that the algorithm can solve - is it limited to very specific types of problems, or will the algorithm perform well in a broad range of problems?

Keep these concepts in mind as you learn about each search algorithm. Let's dive into the algorithms!

### 6.3.18 Breadth-First Search

One of the simplest types of search is called breadth-first search or BFS, it has its name because the algorithm searches a space broadly before it searches deeply. Let's look at a search tree.



Here, we have a number of interconnected nodes with the start node at the top of the tree. Breadth-first search traverses the tree exploring one level at a time.

How the algorithm breaks ties changes from implementation to implementation, but on a search tree like this it is usually implied that you move from left to right. So after a few steps searching through this tree the nodes would have been search in this order as depicted in the image above.

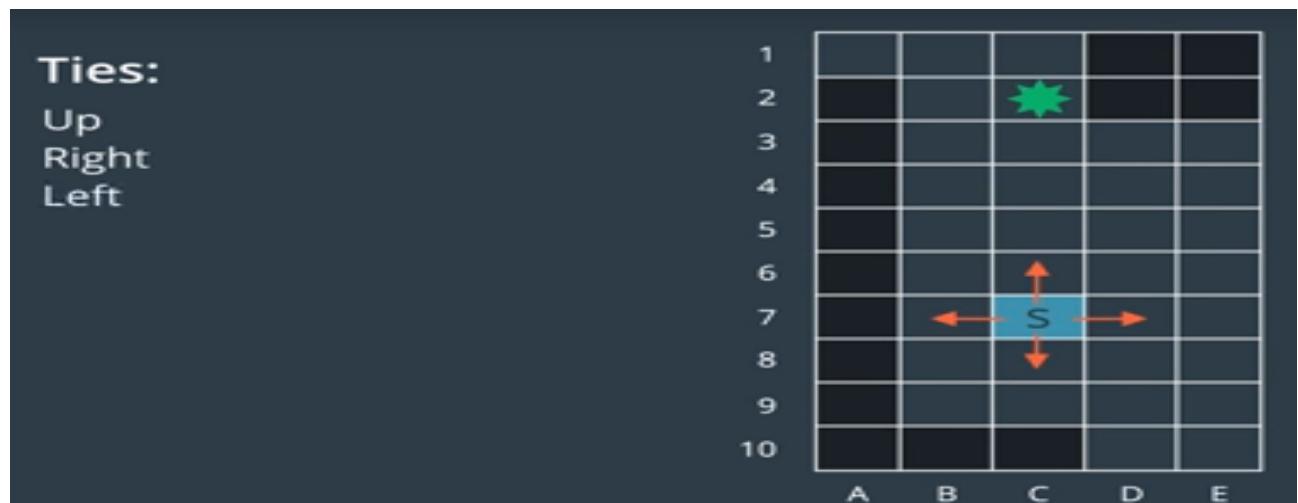
At which steps the algorithm will reach the nodes A B and C?

A: 23            B: 27            C: 22

As you may have guessed Breadth-first search is an uniformed search algorithm, this means that it searches blindly without any knowledge of the space it's traversing or where the goal might be. **For this reason it isn't the most efficient in its operation.**

Let's look at a more complicated example.

Here is a discretized map of an environment, the robot starts at the S state and it would like to find a path to the goal State marked in green, let's assume that the space is four connected meaning that the robot can move up, down, left and right but not diagonally. Thus from a start location the robot has four options to where to explore next. We can't explore all four at once, **so we are going to add each of these options to something called the frontier.**

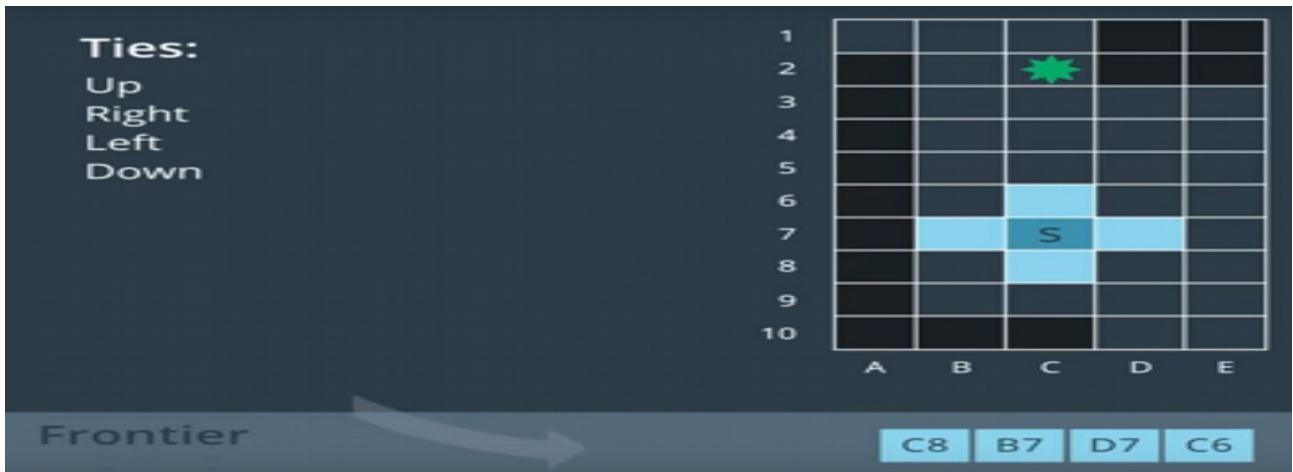


The frontier is the collection of all nodes that we have seen but not yet explored, when the times comes each of these nodes will be removed from the frontier and explored.

Before we add these nodes to the frontier let's set a standard, in our example we will break ties in the following manner, when we have new nodes to add to the frontier, we will choose to add the top node first, then the one on the right, then on the left and finally if no other options are available then the node directly below.

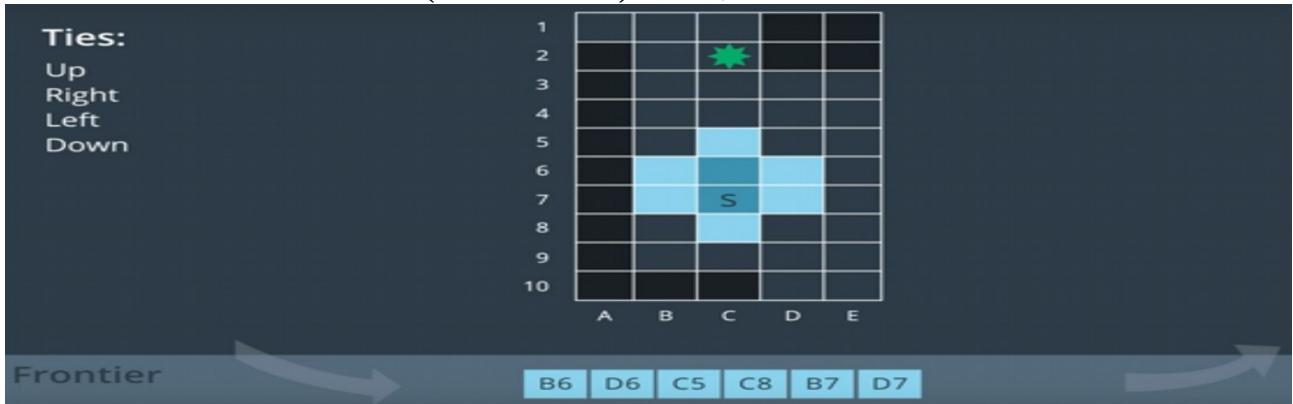
Now we can add the four nodes to the frontier.

For breadth-first search, the data structure underlying the frontier is a queue.



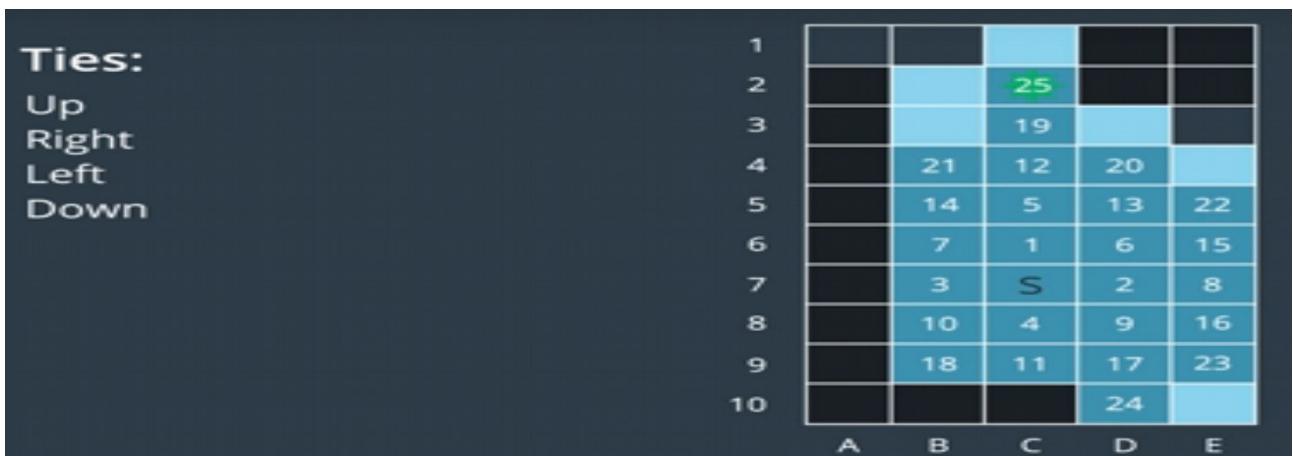
In a queue the first element to enter will be the first to leave the queue.

We will mark previously explored nodes in dark blue. We will explore the C6 node first and we will add 3 more nodes to the frontier (C5 B6 and D6) like so,



The next node to be explored is D7, D7 is removed from the frontier and E7 and D8 are added.

If we follow this logic we can see the result below.



The explored area radiates outward from the start node, breadth-search first searches broadly visiting the closest nodes first, for this reason it takes the algorithm a long time to travel a certain distance because it is radiating in all directions.

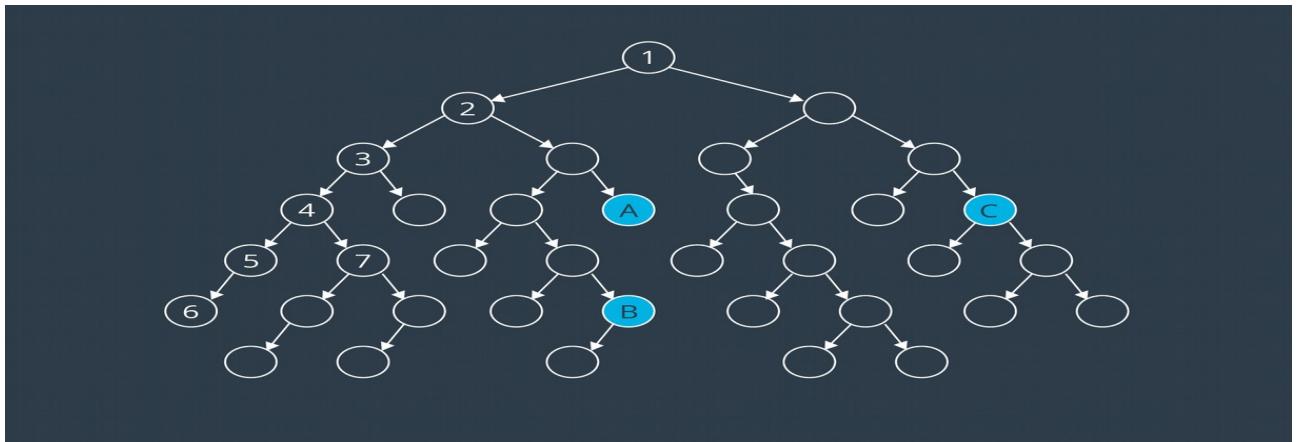
**Eventually the algorithm will find the goal node at step 25 what do you think of that?**

**The algorithm is Complete and Optimal but NOT EFFICIENT.**

### 6.3.19 Depth-First Search

Depth-First Search or (DFS) is another uninformed search algorithm like the name suggest it **searches deeply before it searches broadly**.

If we go back to our search tree



you can see this in action, instead of commencing at the top node and searching level by level, **DFS will explore the start nodes first child and then that nodes child and so on, until it hits the lowest leaf in this branch, only then will DFS backup to a node which had more than one child and explore this node second child**

On which steps of its search would the DFS algorithm reach the nodes labelled A, B, and C?

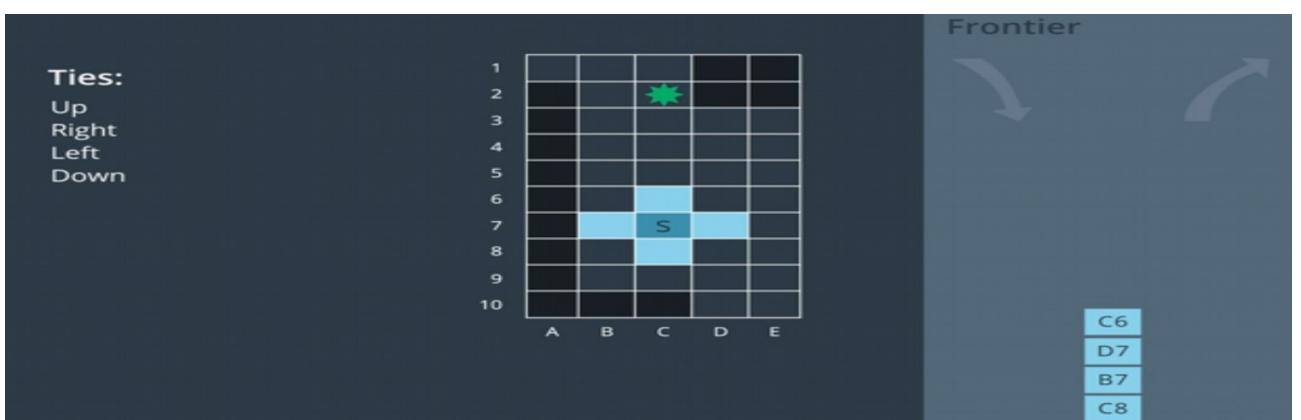
A: 20

B: 18

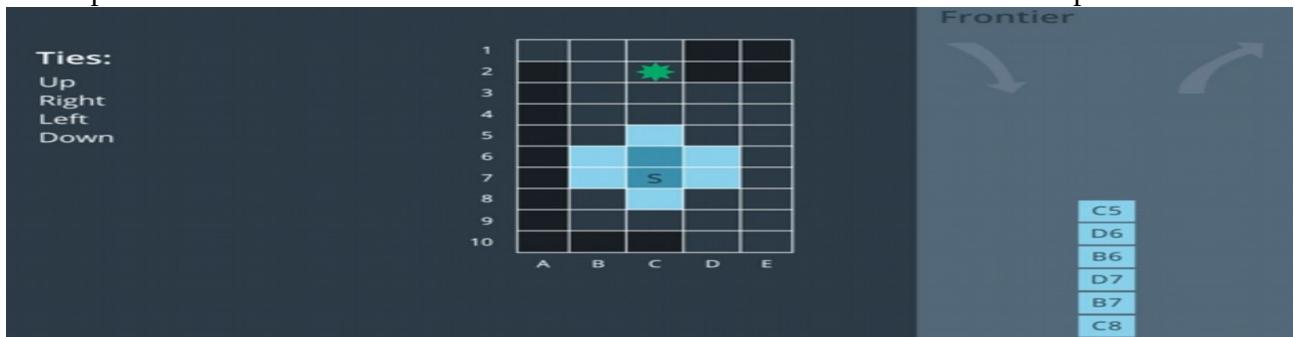
C: 32

let's see the more complicated example now.

The difference now is that our FRONTIER will change, in BFS we used a queue for our frontier which supported expanding the oldest nodes first. In depth-first search, we wish to expand newly visited nodes first. To accommodate this, the data structure underlying the frontier will be a STACK and so the four nodes visible from the start location will be added to the frontier stack.



They are ordered in a way that would have the node above expanded before the right node, left node and node below. After adding these nodes to the frontier DFS will pop the top element off the stack and explore it next. From C6 three more nodes are visible so we add them to the top of the stack.



This process continues. DFS is exploring deep in the upward direction simply because that is how ties are broken. The DFS algorithm continues searching and soon enough find itself at the goal at 5 steps.



Much better than BFS will that always be the case???

What if the goal node was placed at the bottom right corner?

**It will take DFS 30 moves to find the goal in comparison BFS would have find it in 29, seems like neither of this algorithms are too EFFICIENT.**

1					
2			25		
3		27	19	26	
4		21	12	20	28
5		14	5	13	22
6		7	1	6	15
7		3	S	2	8
8		10	4	9	16
9		18	11	17	23
10				24	29

Breadth-First Search: 29

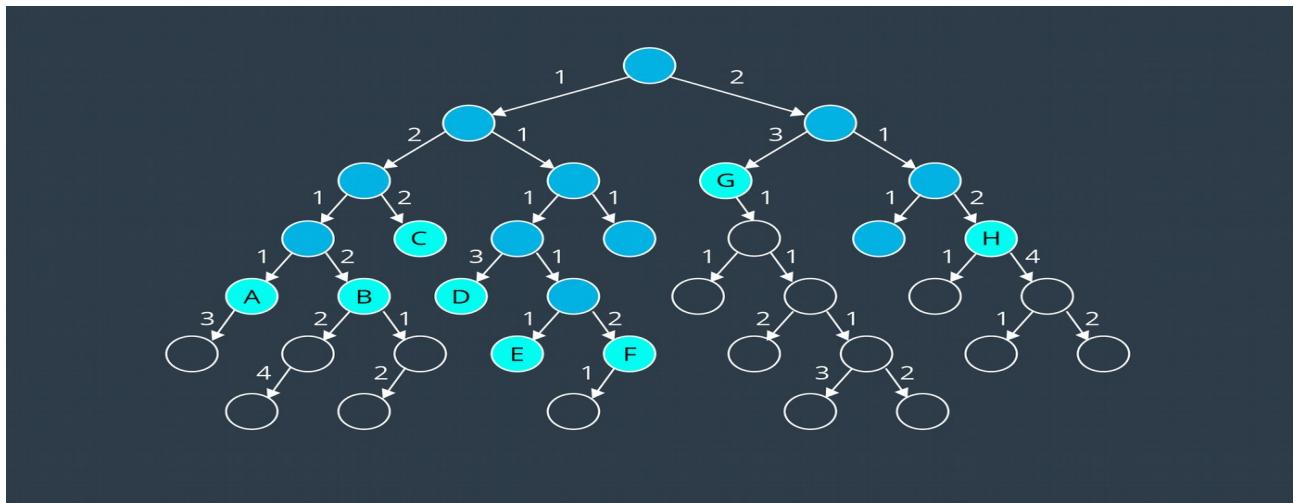
1	8	7	6		
2		9	5		
3		10	4	22	23
4		11	3	21	24
5		12	2	20	25
6		13	1	19	26
7		14	S	18	27
8		15	16	17	28
9					29
10					30

Depth-First Search: 30

DFS specifically is neither Complete Optimal unlike BFS and ofcourse not very efficient

### 6.3.20 Uniform Cost Search

If you recall, BFS is optimal because it expands the shallowest unexplored node with every step. However, BFS is limited to graphs where all step costs are equal. This next algorithm, Uniform Cost Search builds upon BFS to be able to search graphs with differing edge costs. UFS is also optimal because it expands nodes in order of increasing path cost.



In certain environments, you can assign a cost to every edge, the cost may represent one of many things. For instance, the time it takes a robot to move from one node to another, a robot may have to slow down to turn corners or to move across rough terrain. The associated delay can be represented with a higher cost to that edge.



UFS explores nodes on the frontier starting with the node that has the lowest path cost. **Path cost refers to the sum of all edge costs leading from start to that node.** In the image above we have added to the frontier all nodes with a total cost of 1 2 3 and 4.

## Question 1

If you are to expand the search to include all nodes with a path cost of 5, which of the following nodes will not be explored?

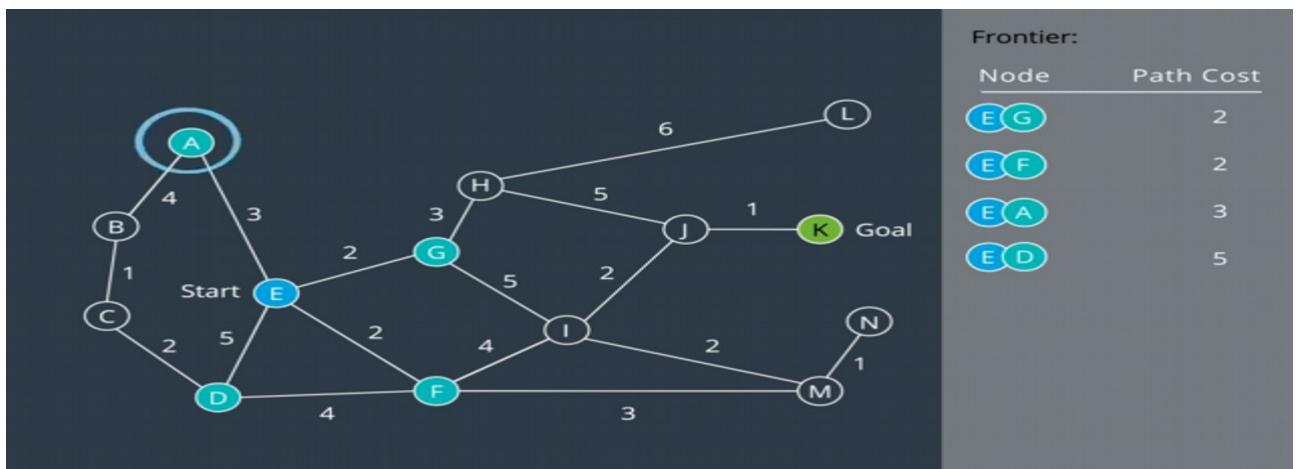
**Answer:** B D F all have a path cost of 6 which will be explored in the following steps along other nodes.

**Now let's explore a more complicated example.**

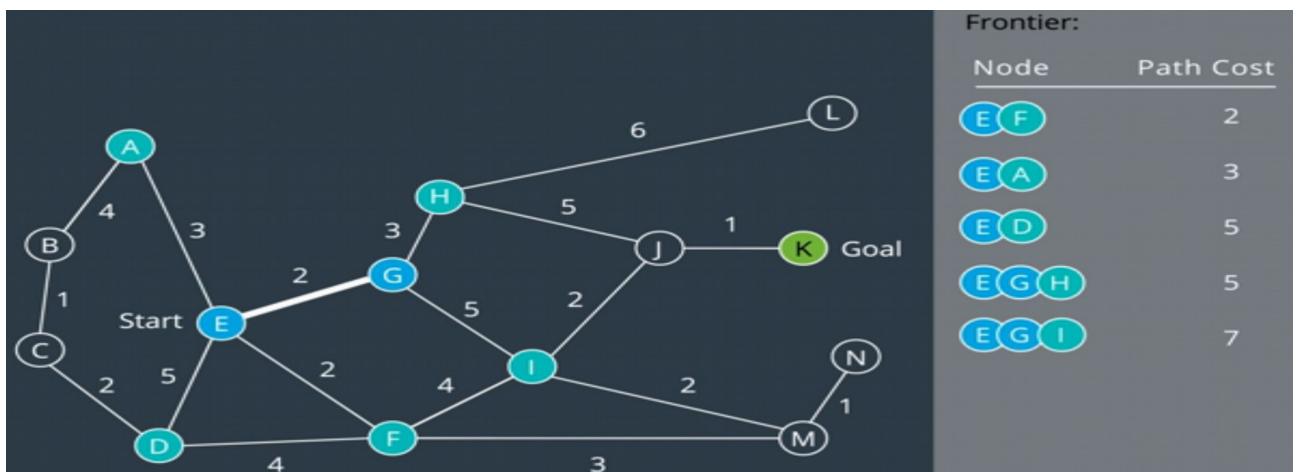
we'd like to apply uniform cost search to find a path from the start node E to the goal node K. Recall that in BFS the frontier was represented by a queue while in DFS it was represented by a stack, each accommodated the corresponding algorithms desired search order, first-in-first-out or last-in-first-out.

In UFS we wish to explore nodes with lowest path cost first. **To accommodate this we can use a priority queue, that is a queue that is organized by the path cost.**

At the start there are four nodes and their corresponding paths are on the frontier, they are organized as such with nodes G and F at the top of the queue as they have the lowest path cost followed by nodes A and D

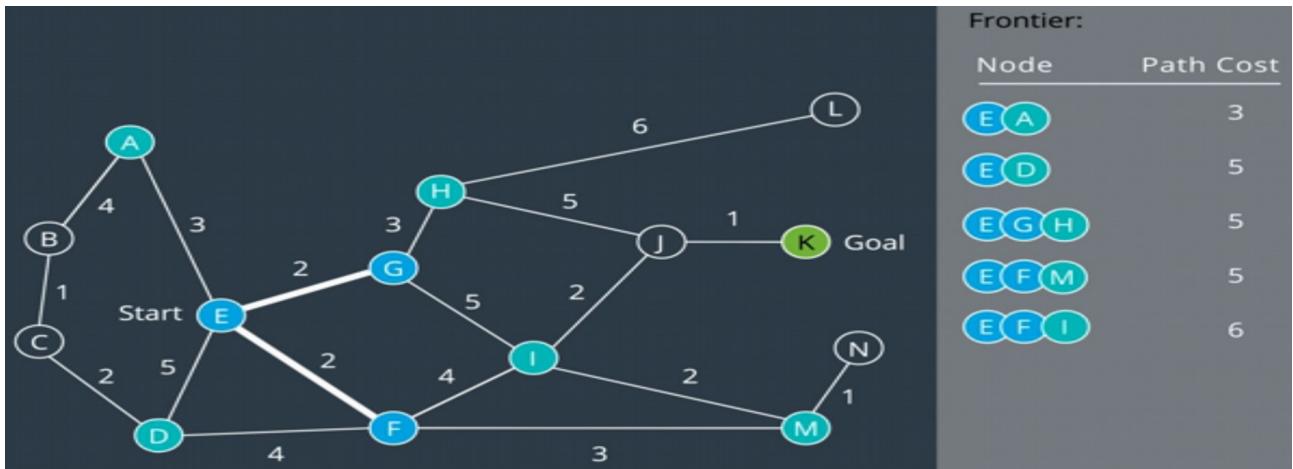


Next we remove the top node from the frontier to explore it, as a result two more nodes are added to the frontier, they assume their appropriate positions in the prioritize queue.



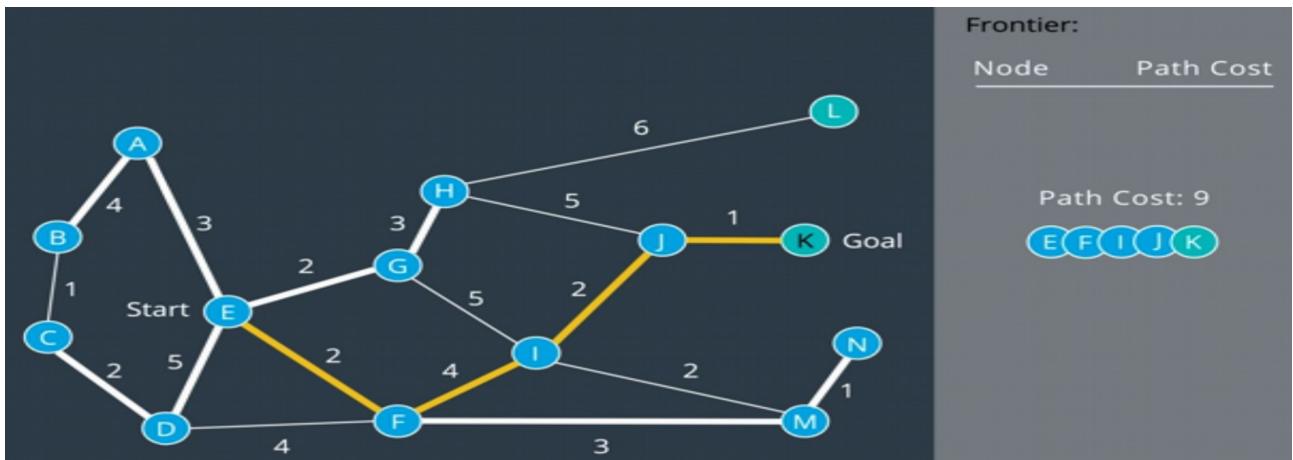
These are the shortest paths to these nodes that we are aware of at this time.

Next we explore node F.



In this process, node M is added to the queue and another interesting thing happens. **The path cost of node I is reduced to six**, this is because a shorter route has been found for this node.

The algorithm continues searching, exploring the node on the frontier that has the lowest path cost, adding new nodes to the frontier and updating the path costs for a node if a shorter path has been found. After some time the goal has been found, it has a path cost of 9 following the path E->F->I->J->K.



**The UFS is Complete and Optimal but still not very Efficient.**

### Terminology Note

In some of the literature for these algorithms you may come across the terms "LIFO" and "FIFO":

**LIFO** stands for **Last In First Out**

**FIFO** stands for **First In First Out**

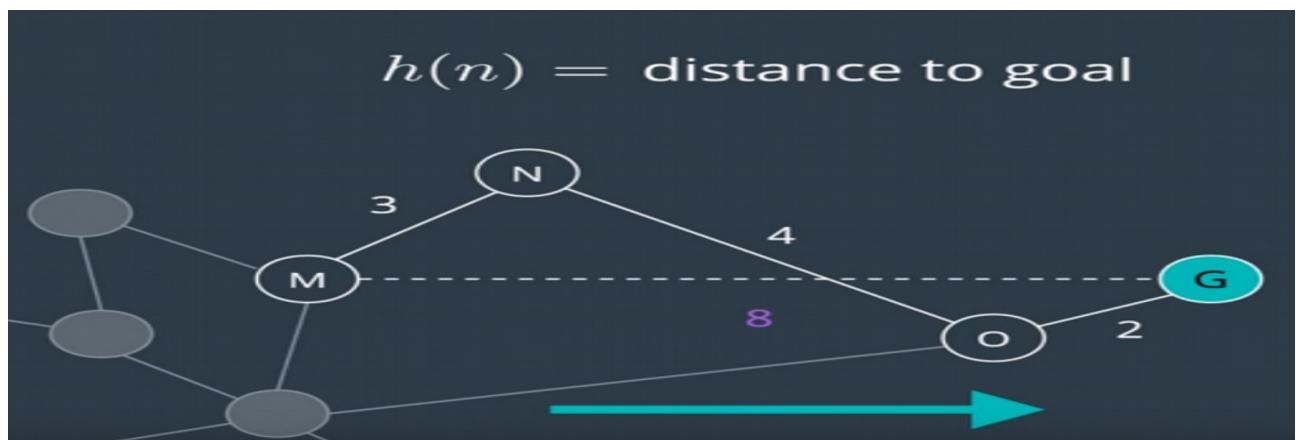
### 6.3.21 A\* Search

The algorithms we examined thus far **have been uninformed**, their search was sprawled in all directions because they **lacked any information regarding the whereabouts of the goal**.

Uninformed	Informed
<ul style="list-style-type: none"> <li>• Breadth-First Search</li> <li>• Depth-First Search</li> <li>• Uniform Cost Search</li> </ul>	<ul style="list-style-type: none"> <li>• A* Search</li> </ul>

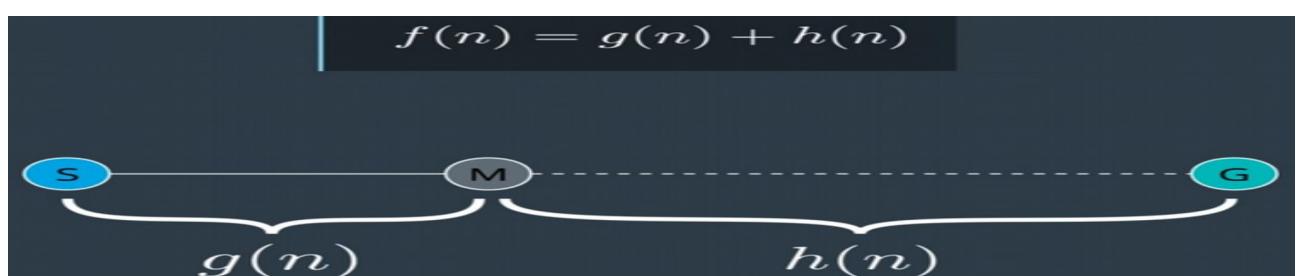
The mightest of them all is A\* Search which is an informed search. This means that it takes into account information about the goal's location as it goes about its search.

It does so by using something called a **heuristic function**. A heuristic function represents the distance from a node to the goal.  $h(n)$  is only an estimate of the distance, as the only way to know the true distance would be to traverse the graph.



However, even an estimate is beneficial as it steers a search in the appropriate direction.

A\* uses more than just a heuristic function in its search strategy. It also takes into account the path cost, A\* chooses the path that minimize the sum of the path cost and the heuristic function, the sum is denoted  $f(n)$ . **By doing so, it accomplishes two things at once, minimizing  $g(n)$  favors shorter paths and minimizing  $h(n)$  favors paths in the direction of the goal.**

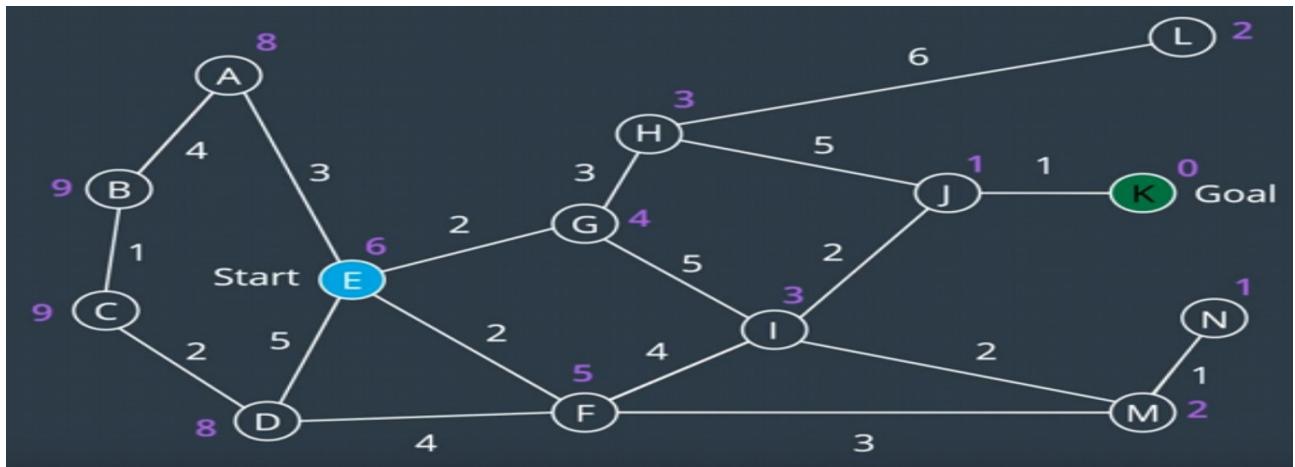


A\* searches for the shortest path in the direction of the goal.

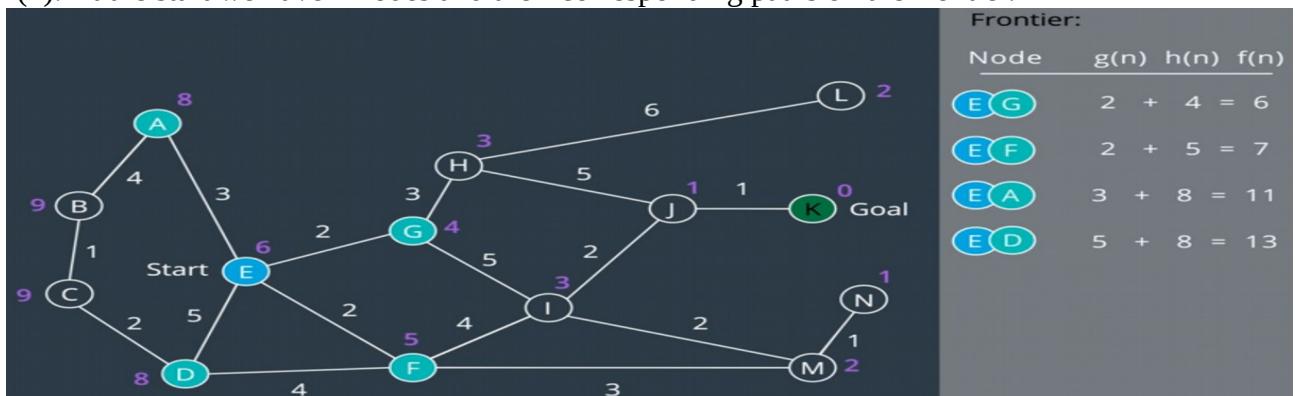
Let's go back to the graph to see this in action.

The objective we have is to find the shortest path from node E to node K, let's try to search this graph once more this time with the help of the heuristic to guide the algorithm to the goal.

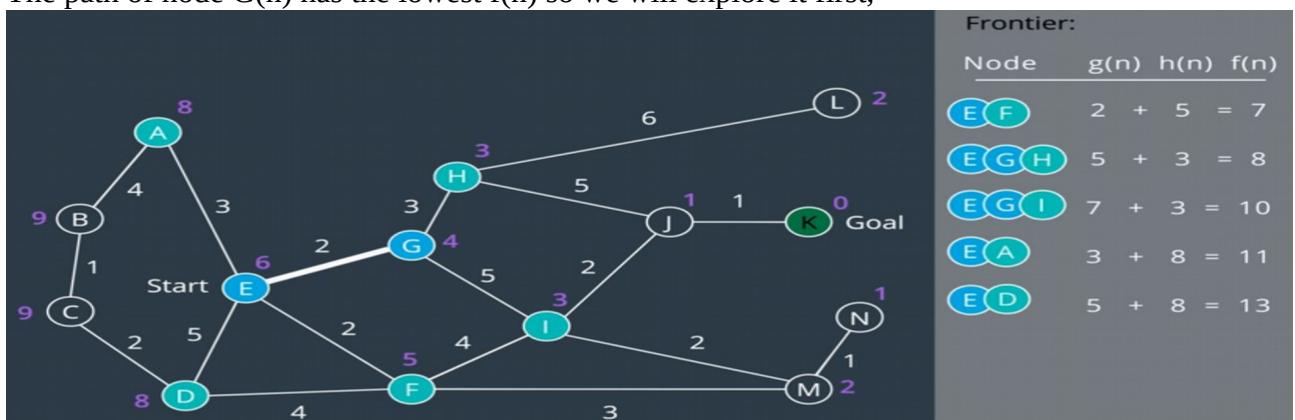
For a 2-D graph like below a valid heuristic would be the Euclidean distance from a node to the goal (nodes close to the goal have a low heuristic value nodes further away have a larger value and the goal itself has a heuristic of 0).



Just like in UFS we will be using a priority queue for the frontier, for A\* Search, we will order it by  $f(n)$ . At the start we have 4 nodes and their corresponding paths on the frontier.



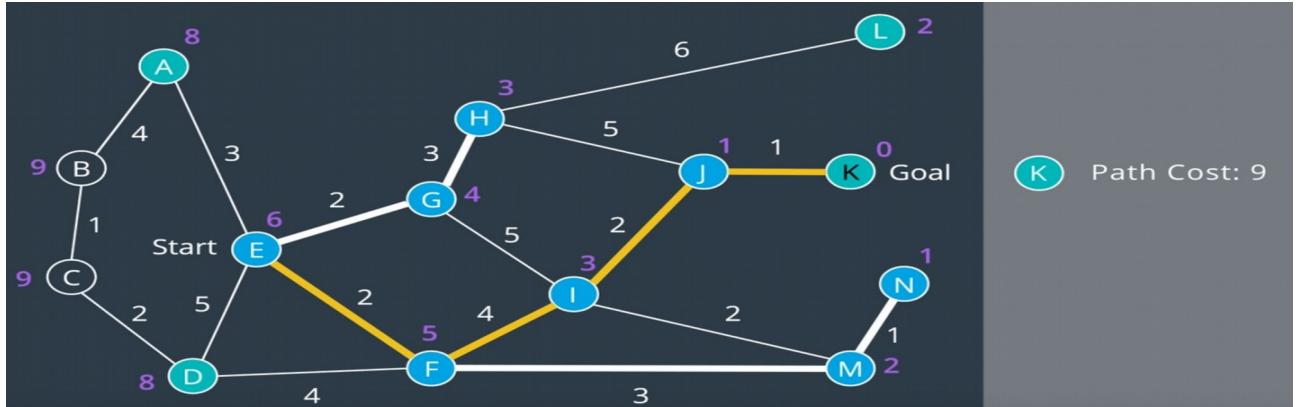
The path of node G(n) has the lowest  $f(n)$  so we will explore it first,



As new nodes are added to the frontier, they are inserted into the appropriate location on the priority queue based of  $f(n)$ . Once again if a shorter path is found to a node the path and value  $f(n)$  of that node will be updated.

Unlike UFS, A\* is directed towards the goal, the nodes on the left are not easily explored.

However A\* would still explore what it believes are promising sections like the dead end at node N.



A\* used less steps than UFS, as it used the heuristic to direct itself towards the goal.

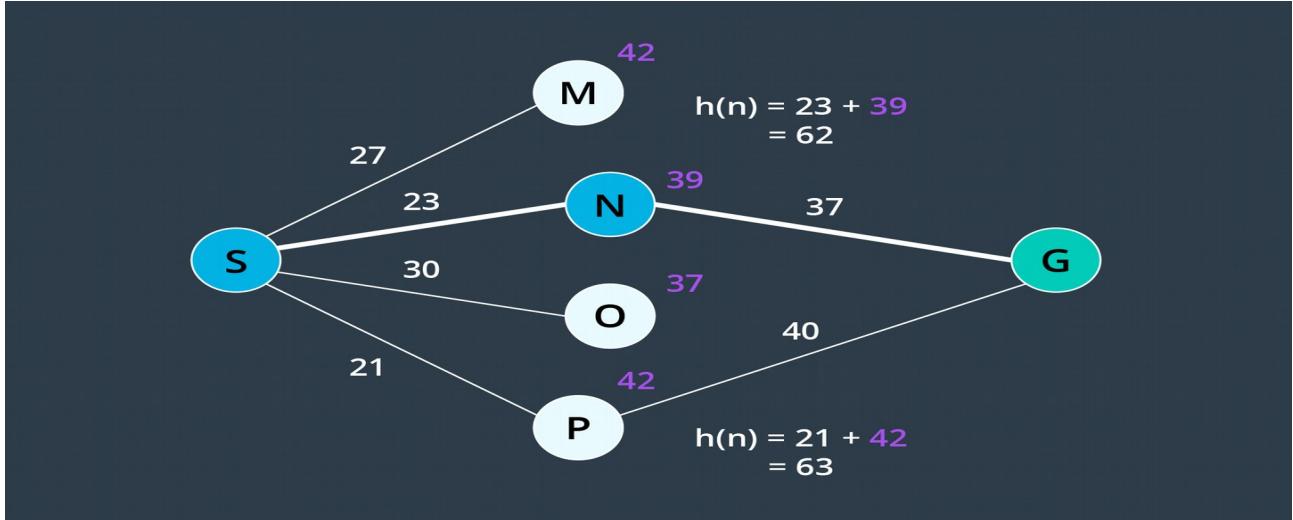
- A heuristic function provides the robot with knowledge about the environment, guiding it in the direction of the goal.
- A\* uses the sum of the path cost and the heuristic function to determine which nodes to explore next.
- A\* uses a priority queue as the data structure underlying the frontier.
- A\* is not optimal
- Choosing an appropriate heuristic function (ex. Euclidean distance, Manhattan distance, etc.) is important. The order in which nodes are explored will change from one heuristic to another.

A\* search orders the frontier using a priority queue, ordered by  $f(n)$ , the sum of the path cost and the heuristic function. This is very effective, as it requires the search to keep paths short, while moving towards the goal. However, as you may have discovered- A\* search is not guaranteed to be optimal. Let's look at why this is so!

A\* search will find the optimal path *if* the following conditions are met,

- Every edge must have a cost greater than some value,  $\epsilon$ , otherwise, the search can get stuck in infinite loops and the search would not be complete.
- The heuristic function must be consistent. This means that it must obey the triangle inequality theorem. That is, for three neighboring points  $(x_1, x_2, x_3)$ , the heuristic value for  $x_1$  to  $x_3$  must be less than the sum of the heuristic values for  $x_1$  to  $x_2$  and  $x_2$  to  $x_3$ .
- The heuristic function must be admissible. This means that  $h(n)$  must always be less than or equal to the true cost of reaching the goal from every node. In other words,  $h(n)$  must never overestimate the true path cost.

To understand where the admissibility clause comes from, take a look at the image below. Suppose you have two paths to a goal where one is optimal (the highlighted path), and one is not (the lower path). Both heuristics overestimate the path cost. From the start, you have four nodes on the frontier, but Node N would be expanded first because its  $h(n)$  is the lowest - it is equal to 62. From there, the goal node is added to the frontier - with a cost of  $23 + 37 = 60$ . This node looks more promising than Node P, whose  $h(n)$  is equal to 63. In such a case, A\* finds a path to the goal which is not optimal. If the heuristics never overestimated the true cost, this situation would not occur because Node P would look more promising than Node N and be explored first.



As you saw in the image above, admissibility is a requirement for A\* to be optimal. For this reason, common heuristics include the Euclidean distance from a node to the goal, or in some applications the Manhattan distance. When comparing two different types of values - for instance, if the path cost is measured in hours, but the heuristic function is estimating distance - then you would need to determine a scaling parameter to be able to sum the two in a useful manner.

If you are interested in learning more about heuristics, visit [Amit's Heuristics Guide](#) on Stanford's website.

While A\* is a much more efficient search in most situations, there will be environments where it will not outperform other search algorithms. This happens if the path to the goal happens to go in the opposite direction first.

Variants of A\* search exist - some accommodate the use of A\* search in dynamic environments, while others help A\* become more manageable in large environments.

## Additional Resources

The following visualization is a great tool that allows you to draw your own obstacles, set your own rules, and perform search using different algorithms.

[Path Finding Visualization](#)

For more information on A\* variants, take a look at:

[MovingAI A\\* Variants](#)

[Variants of A\\* - Stanford](#)

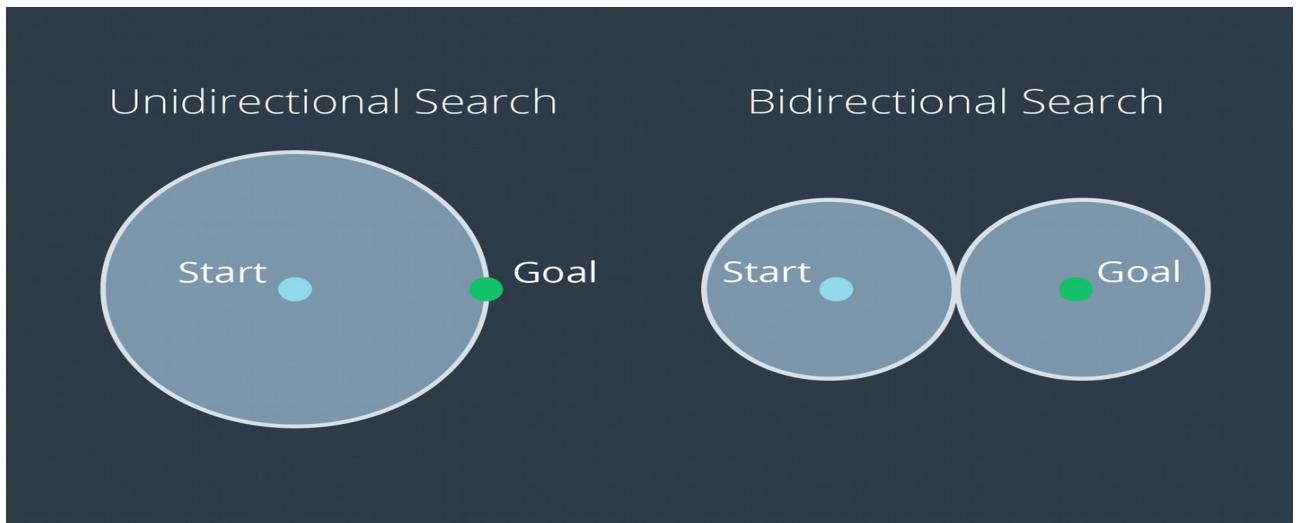
Take some time to investigate the efficiency of A\* over BFS in different scenarios! And if you're feeling extra adventurous, research some of the other algorithms that are provided in the simulation and compare their results to those of BFS & A\*.

### 6.3.22 Overall Concerns Regarding Search

#### Bidirectional Search

One way to improve a search's efficiency is to conduct two searches simultaneously - one rooted at the start node, and another at the goal node. Once the two searches meet, a path exists between the start node and the goal node.

The advantage with this approach is that the number of nodes that need to be expanded as part of the search is decreased. As you can see in the image below, the volume swept out by a unidirectional search is noticeably greater than the volume swept out by a bidirectional search for the same problem.



#### Path Proximity to Obstacles

Another concern with the search of discretized spaces includes the proximity of the final path to obstacles or other hazards. When discretizing a space with methods such as cell decomposition, empty cells are not differentiated from one another. The optimal path will often lead the robot very close to obstacles. In certain scenarios this can be quite problematic, as it will increase the chance of collisions due to the uncertainty of robot localization. The optimal path may not be the best path. To avoid this, a map can be 'smoothed' prior to applying a search to it, marking cells near obstacles with a higher cost than free cells. Then the path found by A\* search may pass by obstacles with some additional clearance.

#### Paths Aligned to Grid

Another concern with discretized spaces is that the resultant path will follow the discrete cells. When a robot goes to execute the path in the real world, it may seem funny to see a robot zig-zag its way across a room instead of driving down the room's diagonal. In such a scenario, a path that is optimal in the discretized space may be suboptimal in the real world. Some careful path smoothing, with attention paid to the location of obstacles, can fix this problem.

### 6.3.23 Graph-Search Wrap-UP

Awesome, we've learned some very applicable search algorithms for path planning in a discretized space. Using this knowledge, we'll be able to direct our robot to get from one point on a map to another and hopefully prove itself useful in one way or another. The search algorithms are also relevant in many other fields, from planning routes for your next drive to finding your friends on social media sites. Graph search is widely used in today's digital society. In addition to learning the algorithms themselves, we've also learned some essential vocabulary and you should now be able to assess an algorithms completeness, optimality, efficiency, and other qualities.

### 6.3.24 Discrete planning Wrap-UP

## Discrete Path Planning

- o Continuous Representation
- o Discretization
- o Graph Search

Well, that's all the risks to Discrete Path Planning. We've learned the three steps that compose the process, and we've learned different approaches to accomplishing every step. With this knowledge, we are now equipped to choose the appropriate approach for an application. For instance, if you were working on a 2-D robot simulation on a powerful computer, you may decide to create a configuration space and discretize it using approximate cell decomposition. Then, you'd apply the A\* algorithm to search the space for a path. On the contrary, if you are doing path planning online on a mobile robot with limited computing power, you may select a less precise approach in favor of a reduced runtime. As of many aspects of engineering, choose the right tools for the job. In the next chapter we will build on what we have learned here and study sample-based planning, to see how it can be applied to more complex path planning problems.

## 6.4 Sample-Based and Probabilistic Path Planning

### 6.4.1 Introduction to Sample-Based & Probabilistic Path Planning

The examples that we investigated in discrete or combinatorial planning were quite simple. They were two-dimensional examples of limited size. There are times when you can simplify your environment and robot to a two-dimensional representation. For instance, a vacuum robot traversing somebody's house, in such a case, there isn't a need for a complex three-dimensional representation as a little robot can't do anything more than translate and rotate on the 2D plane that is the floor.

But there would also be times where you will find yourself limited by the two-dimensional representation and will need to work in a three-dimensional world, with robots that have six degrees of freedom or more.

Such a scenario is a lot more challenging to perform path planning in. It may be possible to apply the previous algorithms however, the efficiency of these algorithms becomes more and more critical. Performing a complete discretization of the entire space and applying a graph search algorithm to the space may be too costly.

To tackle the path planning problem of larger size and greater dimension, there exists alternate algorithms that fall under the umbrella of sample-based path planning. Instead of conducting a complete discretization of the configuration space, **these algorithms randomly sample the space hoping that the collection of samples will adequately represent the configuration space.**

- Sample based path planning is the first topic we will explore in this chapter
- Probabilistic path planning is the second topic, which will look at how we can **explicitly consider the uncertainty of robot motion when planning paths.**

## 6.4.2 Why Sample-Based Planning?

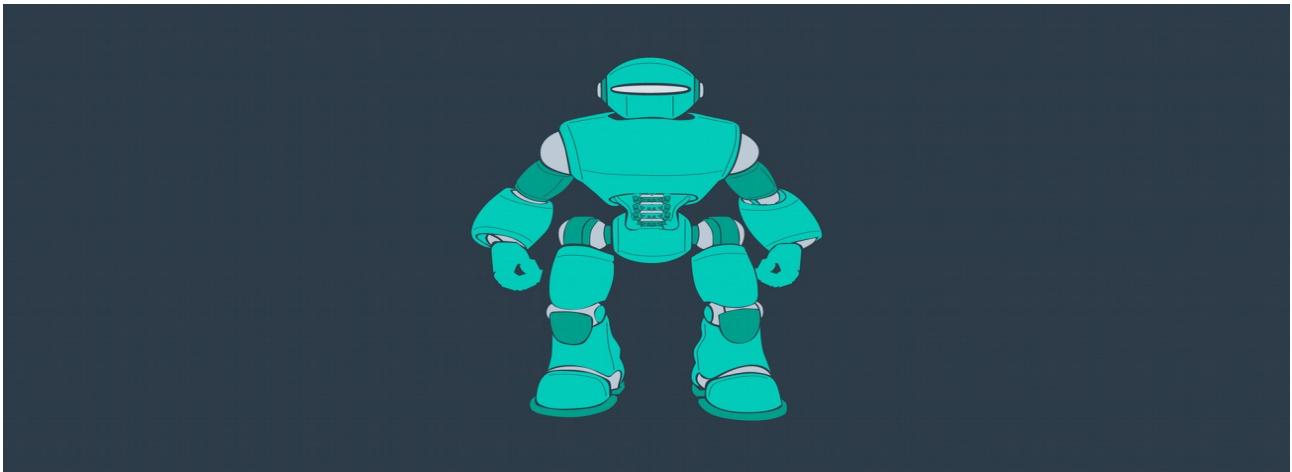
### Why Sample-Based Planning?

So why exactly can't we use discrete planning for higher dimensional problems? Well, it's incredibly hard to discretize such a large space. The complexity of the path planning problem increases exponentially with the number of dimensions in the C-space.

### Increased Dimensionality

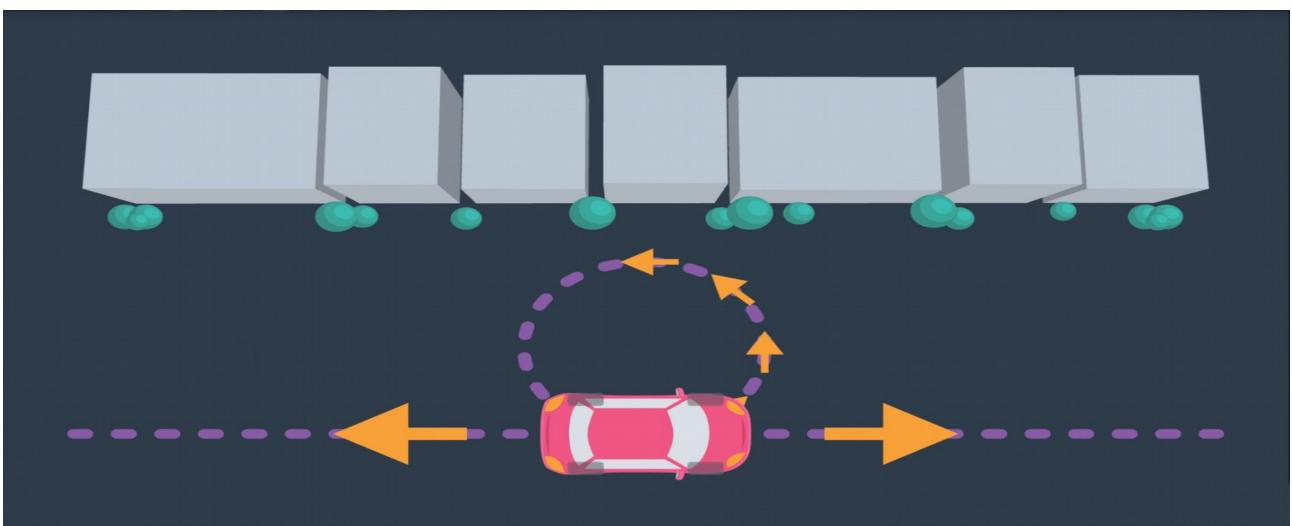
For a 2-dimensional 8-connected space, every node has 8 successors (8-connected means that from every cell you can move laterally or diagonally). Imagine a 3-dimensional 8-connected space, how many successors would every node have? 26. As the dimension of the C-space grows, the number of successors that every cell has increases substantially. In fact, for an n-dimensional space, it is equal to  $3n-1$ .

It is not uncommon for robots and robotic systems to have large numbers of dimensions. For example robotic arm has 6-DOF. If multiple 6-DOF arms work in a common space, the computation required to perform path planning to avoid collisions increases substantially. Then, think about the complexity of planning for humanoid robots such as the one depicted below. Such problems may take intolerably long to solve using the combinatorial approach.

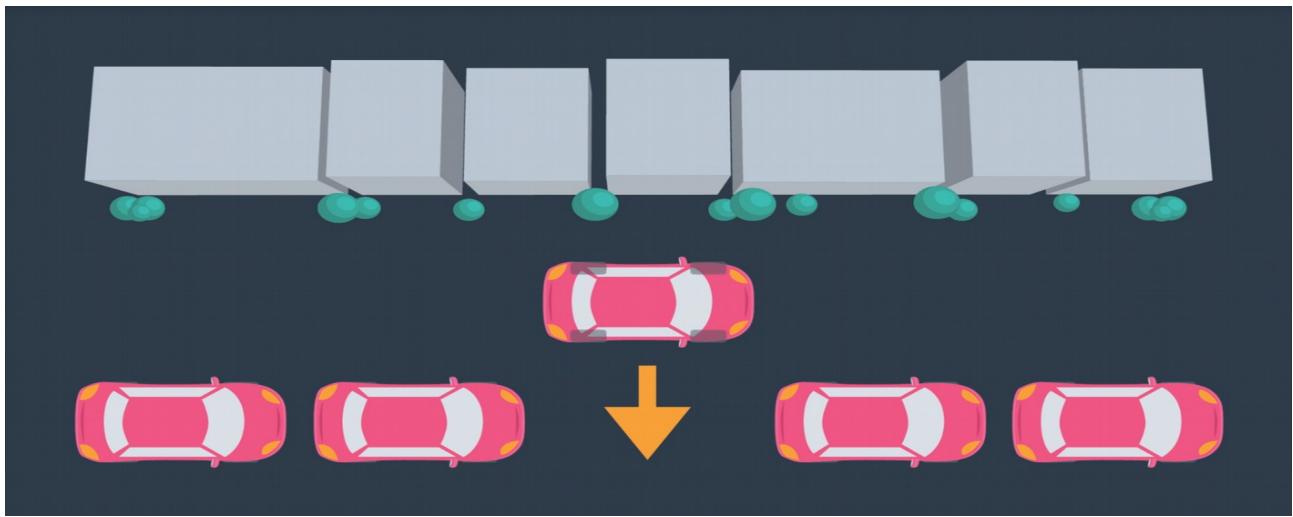


## Constrained Dynamics

Aside from robots with many degrees of freedom and multi-robot systems, another computational difficulty involves working with robots that have constrained dynamics. For instance, a car is limited in its motion - it can move forward and backward, and it can turn with a limited turning radius - as you can see in the image below.



However, the car is *not* able to move laterally - as depicted in the following image. (*As unfortunate as it is for those of us that struggle to parallel park!*)



In the case of the car, more complex motion dynamics must be considered when path planning - including the derivatives of the state variables such as velocity. For example, a car's safe turning radius is dependent on its velocity.

Robotic systems can be classified into two different categories - holonomic and non-holonomic. **Holonomic systems** can be defined as systems where every constraint depends exclusively on the current pose and time, and not on any derivatives with respect to time. **Nonholonomic systems**, on the other hand, are dependent on derivatives. Path planning for nonholonomic systems is more difficult due to the added constraints.

In this section, you will learn two different path planning algorithms, and understand how to tune their parameters for varying applications.

### 6.4.3 Weakening Requirements

Combinatorial path planning algorithms are too inefficient to apply in high-dimensional environments, which means that some practical compromise is required to solve the problem! Instead of looking for a path planning algorithm that is both complete and optimal, what if the requirements of the algorithm were weakened?

Instead of aspiring to use an algorithm that is complete, the requirement can be weakened to use an algorithm that is probabilistically complete. A **probabilistically complete** algorithm is one who's probability of finding a path, if one exists, increases to 1 as time goes to infinity.

Similarly, the requirement of an optimal path can be weakened to that of a feasible path. A **feasible path** is one that obeys all environmental and robot constraints such as obstacles and motion constraints. For high-dimensional problems with long computational times, it may take unacceptably long to find the optimal path, whereas a feasible path can be found with relative ease. Finding a feasible path proves that a path from start to goal exists, and if needed, the path can be optimized locally to improve performance.

Sample-based planning is probabilistically complete and looks for a feasible path instead of the optimal path.

### 6.4.4 Sample-Based Path Planning

Sample-based path planning differs from combinatorial path planning in that it does not try to systematically discretize the entire configuration space. Instead, it samples the configuration space randomly (or semi-randomly) to build up a representation of the space. The resultant graph is not as precise as one created using combinatorial planning, but it is much quicker to construct because of the relatively small number of samples used.

Such a method is probabilistically complete because as time passes and the number of samples approaches infinity, the probability of finding a path, if one exists, approaches 1.

Such an approach is very effective in high-dimensional spaces, however it does have some downfalls. Sampling a space uniformly is not likely to reach small or narrow areas, such as the passage depicted in the image below. Since the passage is the only way to move from start to goal, it is critical that a sufficient number of samples occupy the passage, or the algorithm will return ‘no solution found’ to a problem that clearly has a solution.



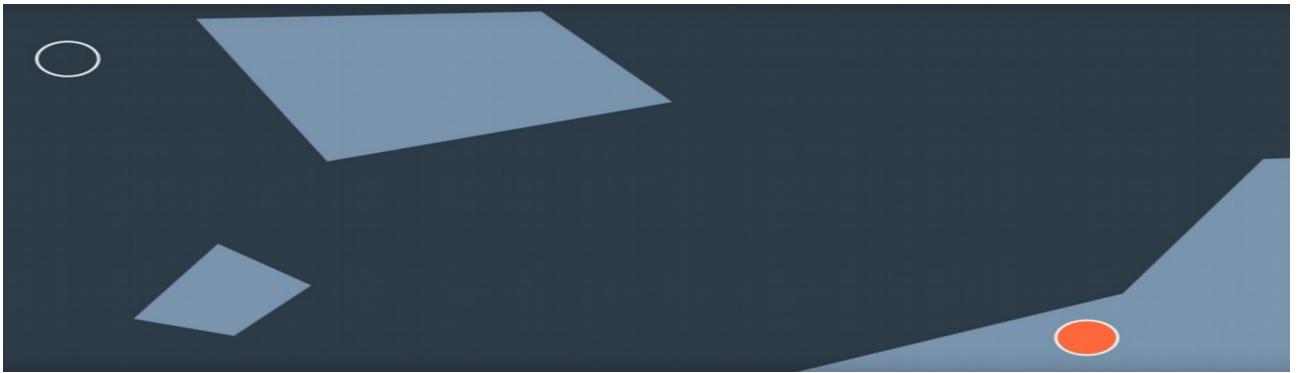
Different sample-based planning approaches exist, each with their own benefits and downfalls. In the next few pages you will learn about,

- Probabilistic Roadmap Method
- Rapidly Exploring Random Tree Method

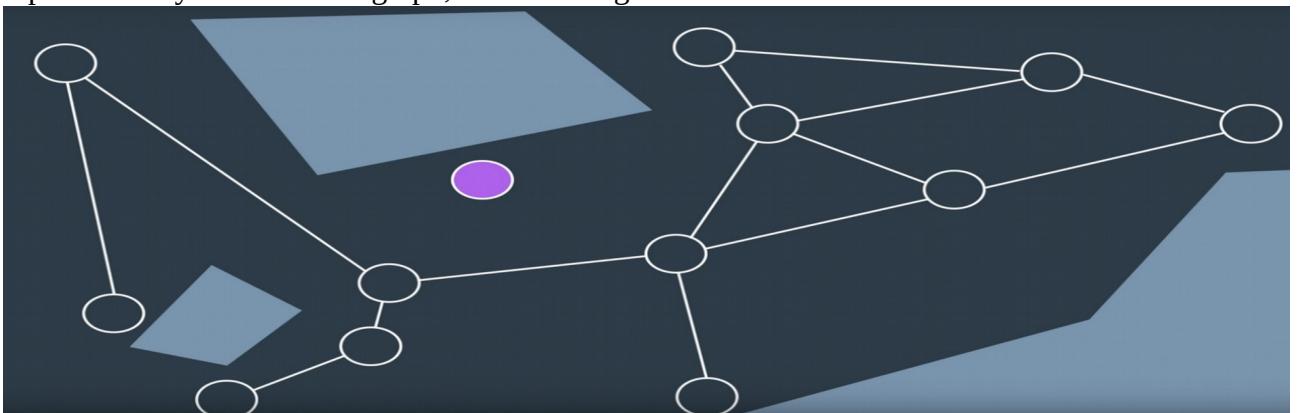
We will also learn about Path Smoothing - one improvement that can make resultant paths more efficient.

#### 6.4.5 Probabilistic Roadmap (PRM)

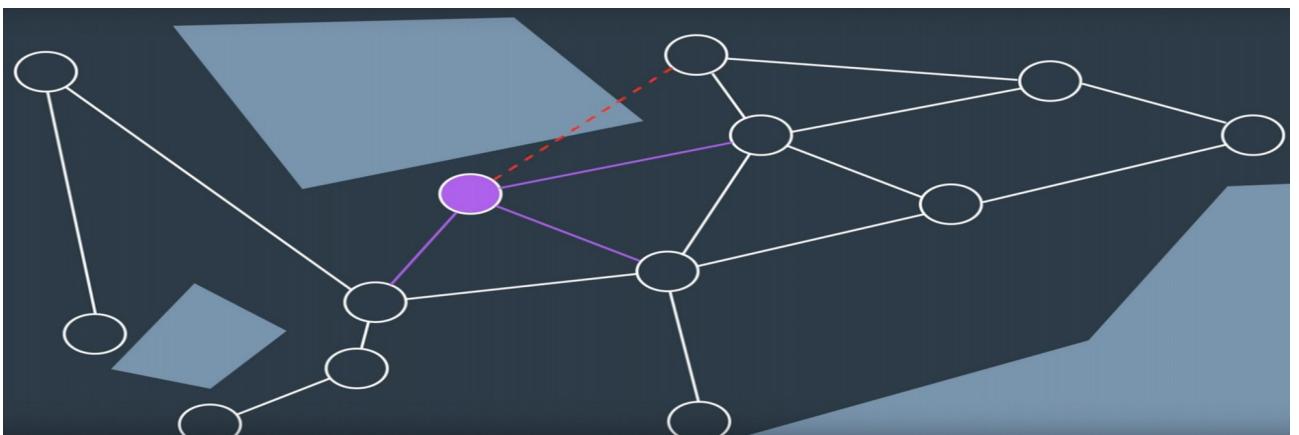
One common sample-based path planning method, is called the probabilistic roadmap or PRM for short. PRM randomly samples the workspace building up a graph to represent the free space. It does so without needing to construct the C space or discretize it. **All that PRM requires is a collision check function**, to test whether a randomly generated node lies in the freespace or is in collision with an obstacle.



Let's add a few more random samples to this workspace, and see how PRM works. The process of building up a graph is called the learning phase. Since that what PRM does by sampling random configurations and adding them to the graph. It does so by generating a new random configuration represented by a node in the graph, and checking to see if it is in collision.



If it is not like the node you see above then PRM, then PRM will try to connect the node to its neighbors. There are a few different ways of doing so. PRM can look for any number of neighbors within a certain radius of the node or it could look for the nodes K nearest neighbors. Once the neighbors have been selected, PRM will see if it can successfully create an edge to each of its neighbors.

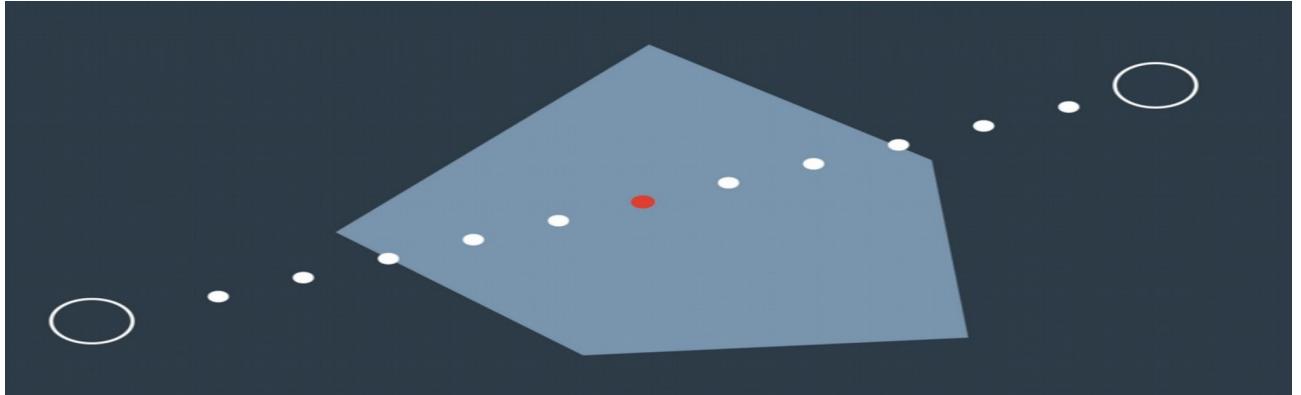


As you see above, one edge is in collision with an obstacle while the other three are safe to add. This node has been added to the graph and now the process can be repeated for another randomly generated node.

The local planner must find a path between two nodes or return that such a path does not exist, and it must do so quickly since this step is repeated for every neighbor of every new node. One easy

way to accomplish this, is to draw a straight line between two nodes, and then check if any part of it collides with an obstacle. To do this, you can place a number of evenly spaced samples on the line and see whether any one of them is in collision.

You can work incrementally starting on one side of the edge and moving toward the other, or you can take a binary approach, checking the sample at the mid point first. In this case, that returns a collision right away.

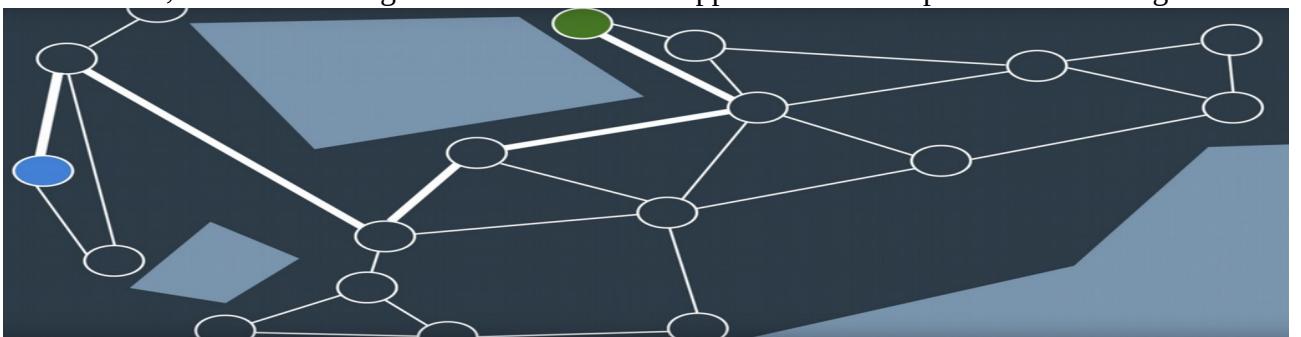


But if it didn't, you could continue breaking the edge up into segments, checking each midpoint sample for a collision, if all the samples returned no collision found, then the edge can be added to the graph.



The process of adding new nodes and connecting them to the graph continues, eventually a certain criteria is met, such as a specific number of nodes or edges have been created, or a particular amount of time has elapsed. **At this point the learning phase is over.**

**Then, PRM enters the query phase. Where it uses the resulting graph to find a path from start to goal.** First, it must connect each of these to the graph, PRM does so by looking for the nodes closest to the start and goal and using the local planner to try and build a connection. If this process is successful, then a search algorithm like A\* can be applied to find the path from start to goal.



**The resulting path may not be optimal, but it proves that moving from start to goal is feasible.**

## Algorithm

The pseudocode for the PRM learning phase is provided below.

### Initialize an empty graph

**For** n iterations:

**Generate** a random configuration.

**If** the configuration is collision free:

**Add** the configuration to the graph.

**Find** the k-nearest neighbours of the configuration.

**For** each of the k neighbours:

**Try to find** a collision-free path between  
the neighbour and original configuration.

**If** edge is collision-free:

**Add** it to the graph.

**After the learning phase, comes the query phase.**

## Setting Parameters

There are several parameters in the PRM algorithm that require tweaking to achieve success in a particular application. Firstly, the **number of iterations** can be adjusted - the parameter controls between how detailed the resultant graph is and how long the computation takes. For path planning problems in wide-open spaces, additional detail is unlikely to significantly improve the resultant path. However, the additional computation is required in complicated environments with narrow passages between obstacles. Beware, setting an insufficient number of iterations can result in a ‘path not found’ if the samples do not adequately represent the space.

Another decision that a robotics engineer would need to make is **how to find neighbors** for a randomly generated configuration. One option is to look for the k-nearest neighbors to a node. To do so efficiently, a k-d tree can be utilized - to break up the space into ‘bins’ with nodes, and then search the bins for the nearest nodes. Another option is to search for any nodes within a certain distance of the goal. Ultimately, knowledge of the environment and the solution requirements will drive this decision-making process.

The choice for what type of **local planner** to use is another decision that needs to be made by the robotics engineer. The local planner demonstrated above is an example of a very simple planner. For most scenarios, a simple planner is preferred, as the process of checking an edge for collisions is repeated many times ( $k*n$  times, to be exact) and efficiency is key. However, more powerful planners may be required in certain problems. In such a case, the local planner could even be another PRM.

## Probabilistically Complete

As discussed before, sample-based path planning algorithms are probabilistically complete. Now that you have seen one such algorithm in action, you can see why this is the case. As the number of iterations approaches infinity, the graph approaches completeness and the optimal path through the graph approaches the optimal path in reality.

## Variants

The algorithm that you learned here is the vanilla version of PRM, but many other variations to it exist. The following link discusses several alternative strategies for implementing a PRM that may produce a more optimal path in a more efficient manner.

- [A Comparative Study of Probabilistic Roadmap Planners](#)

## PRM is a Multi-Query Planner

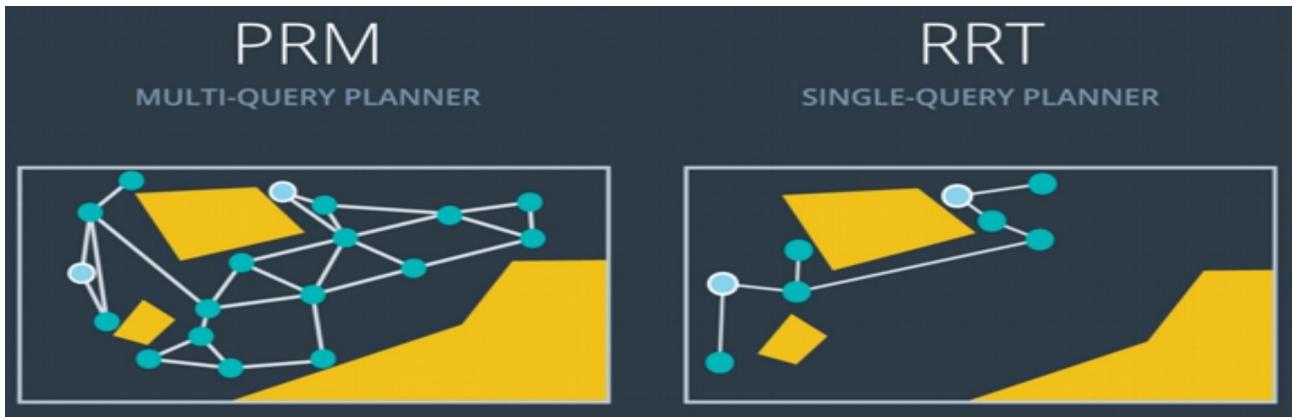
The Learning Phase takes significantly longer to implement than the Query Phase, which only has to connect the start and goal nodes, and then search for a path. However, the graph created by the Learning Phase can be reused for many subsequent queries. For this reason, PRM is called a **multi-query planner**.

This is very beneficial in static or mildly-changing environments. However, some environments change so quickly that PRM's multi-query property cannot be exploited. In such situations, PRM's additional detail and computational slow nature is not appreciated. A quicker algorithm would be preferred - one that doesn't spend time going in *all* directions without influence by the start and goal.

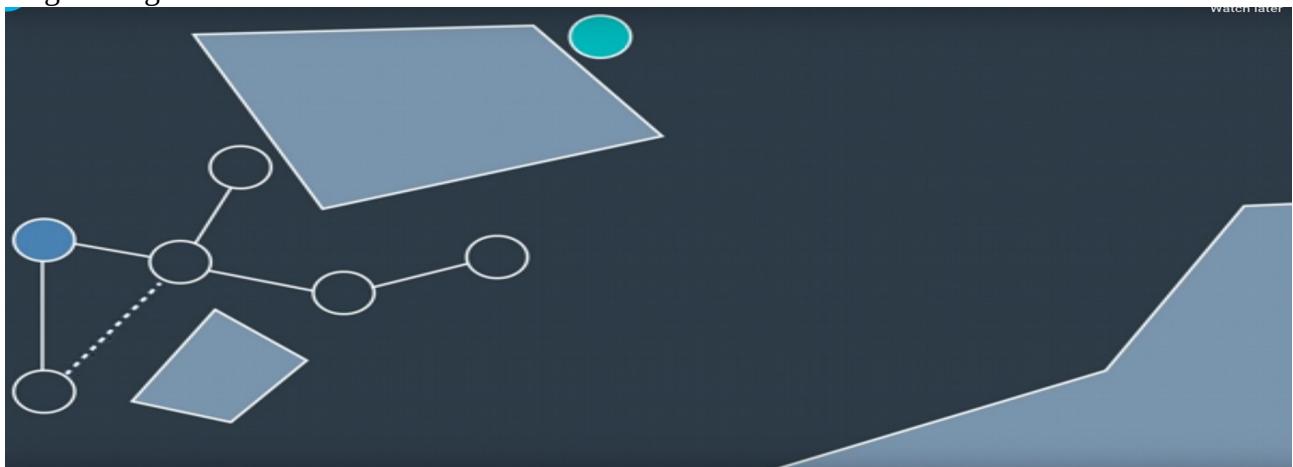
- **PRM is a multi-query method (ie. the resultant graph can be used for multiple queries).**
- **If an insufficient iterations of the PRM algorithm are run, there is a risk of finding an inefficient path or not finding a path at all.**
- **When A\* is applied to the graph generated by RPM, the optimal path is NOT found.**

## 6.4.6 Rapidly Exploring Random Tree Method (RRT)

Another commonly utilized sample-based path planning method, is the randomly exploring random tree method, or RRT for short. RRT differs from PRM in that it is a single query planner. If you recall, PRM spent its learning phase building up a representation of the entire workspace. This was computationally expensive, but the resultant graph can be used for multiple queries. RRT disregards the need for a comprehensive graph, and builds one new for each individual query, taking into account the start and goal positions as it does so. This results in a much smaller, but more directed graph with a faster computation time. PRM is great for static environments, where you can reuse the graph, but certain environments change too quickly, and the RRT method serves these environments well.

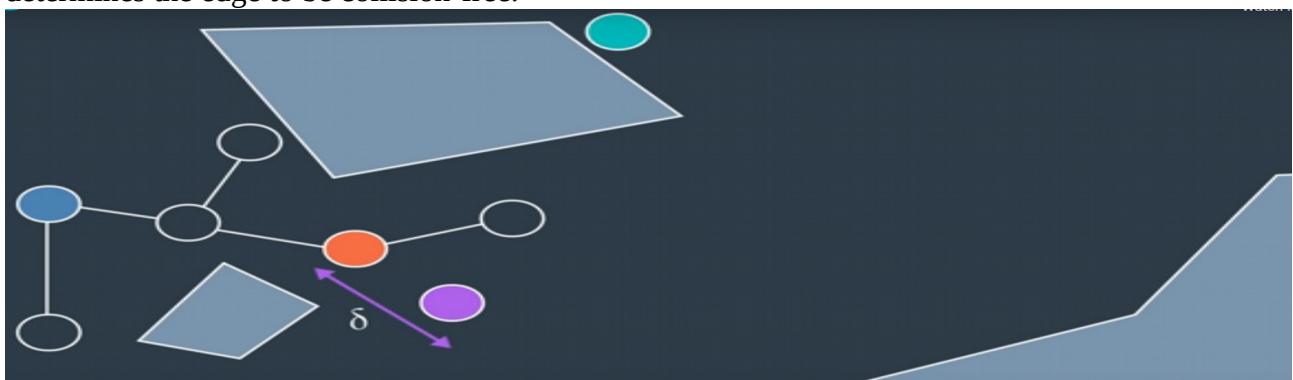


Let's see what the RRT method looks like, we're path planning at the same environment as before with the same start and goal configurations. **However, instead of adding these in the learning phase, they will be explicitly considered from the start.** Then, we start to build up a representation of the workspace, while the PRM built up a graph, RRT will build a tree, that is a type of graph where each node only has one parent. In a single query planner, you are only concerned about getting from start to goal and the lack of lateral connections between seemingly neighboring nodes is less of a concern.



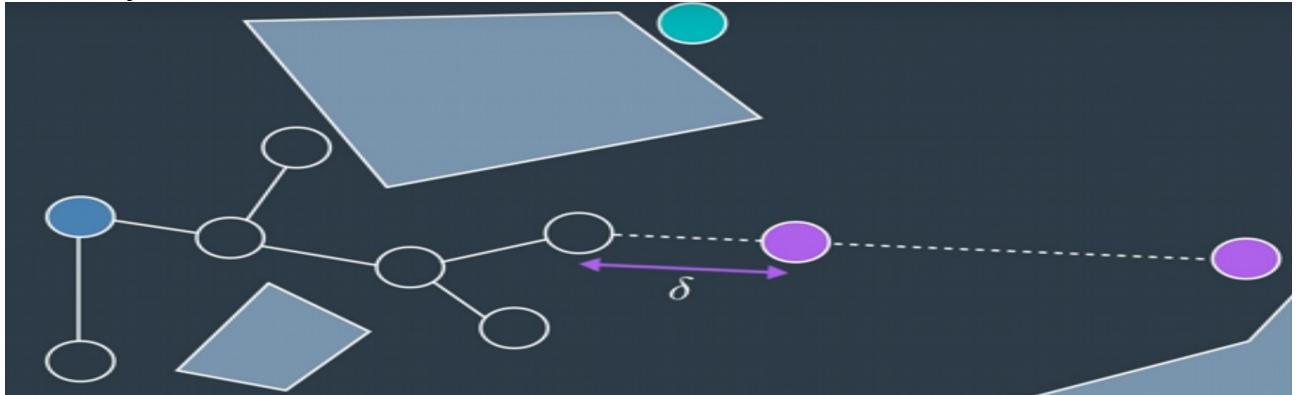
So what is the algorithm?

RRT will randomly generate a node, then it will find its closest neighbor if the node is within a certain distance  $\delta$  of the neighbor, then it can be connected directly. Of course if the local planner determines the edge to be collision-free.



However, if a newly generated node is a far distance away from all other nodes, then the chance of the edge between the node and its nearest neighbor be collision-free is unlikely. In such a case,

instead of connecting to this node, RRT will create a new node in the same direction, but a distance delta away.

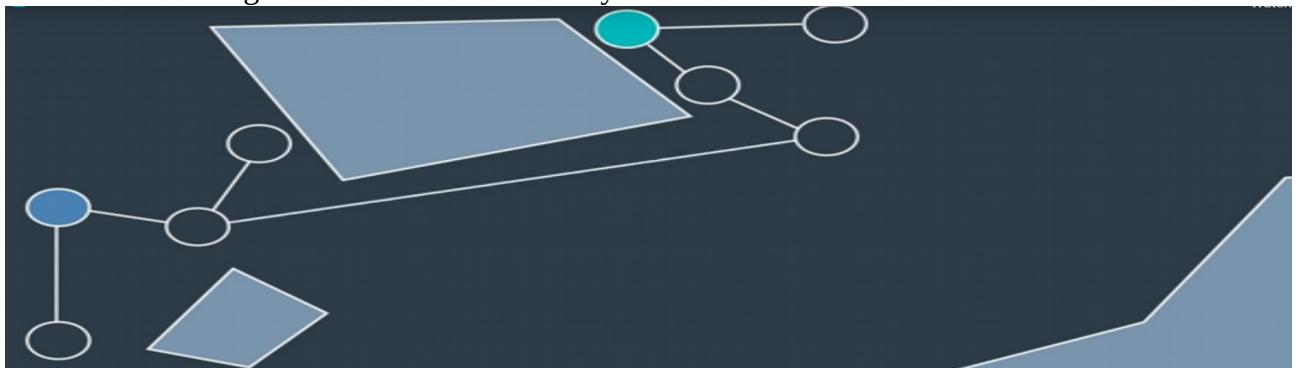


Then this edge is checked for collisions, and if it's in the clear the node is added to the tree.

Nodes can be generated by uniformly sampling the search space, which would favor wide unexplored spaces, or alternatively, some greediness can be introduced by increasing the probability of sampling near the goal, which would bias new samples in the direction of the goal. **Since RRT is a single query planner, slight biasing is often favorable to introduce.**

One variation of the RRT method is one that grows two trees.

One from the start and one from the goal. RRT alternates growing each tree, and at every step, It tries to build an edge between the most recently added node and the other tree.

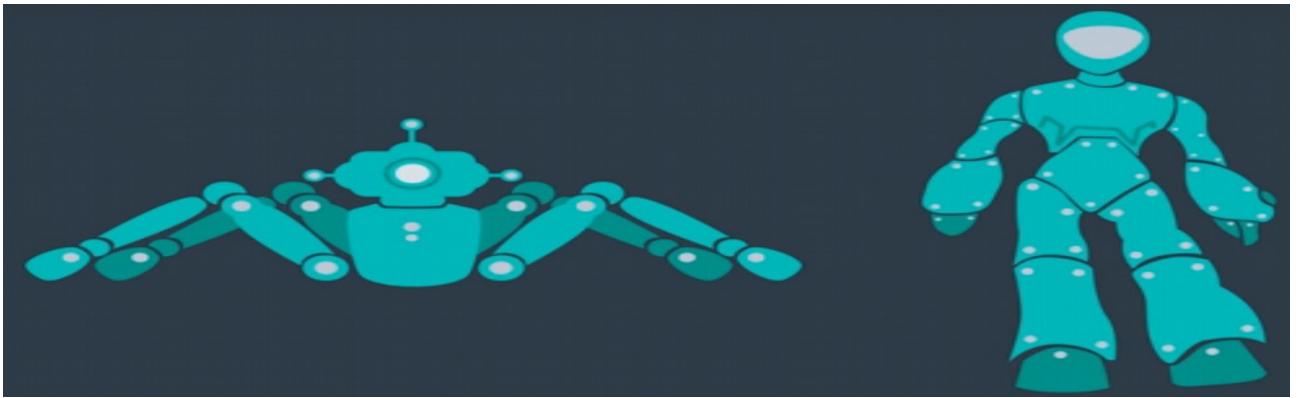


Eventually, it succeeds. RRT knows that a path has been found from start to goal.

This is a very simplified example of a workspace and not a true representation of a problem where PRM or RRT would be tasked with solving.

PRM and RRT are exemplary performers in multidimensional spaces for robots with many degrees of freedom.

In fact, they have been able to solve problems that traditional path planning algorithms are unable to solve.



## Algorithm

The pseudocode for the RRT learning phase is provided below.

**Initialize** two empty trees.

**Add** start node to tree #1.

**Add** goal node to tree #2.

**For** n iterations, or until an edge connects trees #1 & #2:

**Generate** a random configuration (alternating trees).

**If** the configuration is collision free:

**Find** the closest neighbour on the tree to the configuration

**If** the configuration is less than a distance  $\delta$  away from the neighbour:

**Try to connect** the two with a local planner.

**Else:**

**Replace** the randomly generated configuration

        with a new configuration that falls along the same path,  
        but a distance  $\delta$  away from the neighbour.

**Try to connect** the two with a local planner.

**If** node is added successfully:

**Try to connect** the new node to the closest neighbour.

## Setting Parameters

Just like with PRM, there are a few parameters that can be tuned to make RRT more efficient for a given application.

The first of these parameters is the **sampling method** (ie. how a random configuration is generated). As discussed above, you can sample uniformly - which would favour wide unexplored spaces, or you can sample with a bias - which would cause the search to advance greedily in the direction of the goal. Greediness can be beneficial in simple planning problems, however in some environments it can cause the robot to get stuck in a local minima. It is common to utilize a uniform sampling method with a *small* hint of bias.

The next parameter that can be tuned is  $\delta$ . As RRT starts to generate random configurations, a large proportion of these configurations will lie further than a distance  $\delta$  from the closest configuration in the graph. In such a situation, a randomly generated node will dictate the direction of growth, while  $\delta$  is the growth rate.

Choosing a small  $\delta$  will result in a large density of nodes and small growth rate. On the other hand, choosing a large  $\delta$  may result in lost detail, as well as an increasing number of nodes being unable to connect to the graph due to the greater chance of collisions with obstacles.  $\delta$  must be chosen carefully, with knowledge of the environment and requirements of the solution.

## Single-Query Planner

Since the RRT method explores the graph starting with the start and goal nodes, the resultant graph cannot be applied to solve additional queries. RRT is a single-query planner.

RRT is, however, much quicker than PRM at solving a path planning problem. This is so because it takes into account the start and end nodes, and limits growth to the area surrounding the existing graph instead of reaching out into all distant corners, the way PRM does. RRT is more efficient than PRM at solving large path planning problems (ex. ones with hundreds of dimensions) in dynamic environments.

Generally speaking, RRT is able to solve problems with 7 dimensions in a matter of milliseconds, and may take several minutes to solve problems with over 20 dimensions. In comparison, such problems would be impossible to solve with the combinatorial path planning method.

## RRT & Non-holonomic Systems

While we will not go into significant detail on this topic, the RRT method supports planning for non-holonomic systems, while the PRM method does not. This is so because the RRT method can take into consideration the additional constraints (such as a car's turning radius at a particular speed) when adding nodes to a graph, the same way it already takes into consideration how far away a new node is from an existing tree.

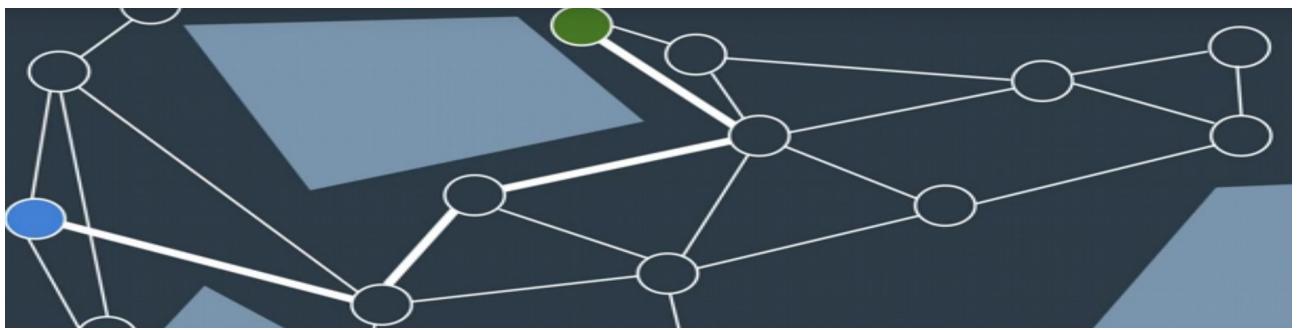
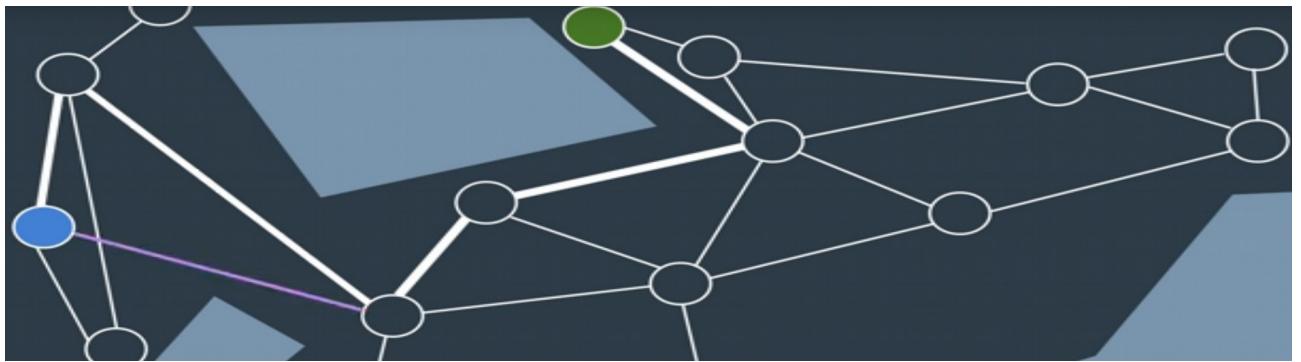
- **RRT can be applied to path planning with non-holonomic systems.**
- **If  $\delta$  is set to a large value, the algorithm's efficiency will drop, as the local planner is more likely to encounter collisions along a longer path.**

## 6.4.7 Path Smoothing

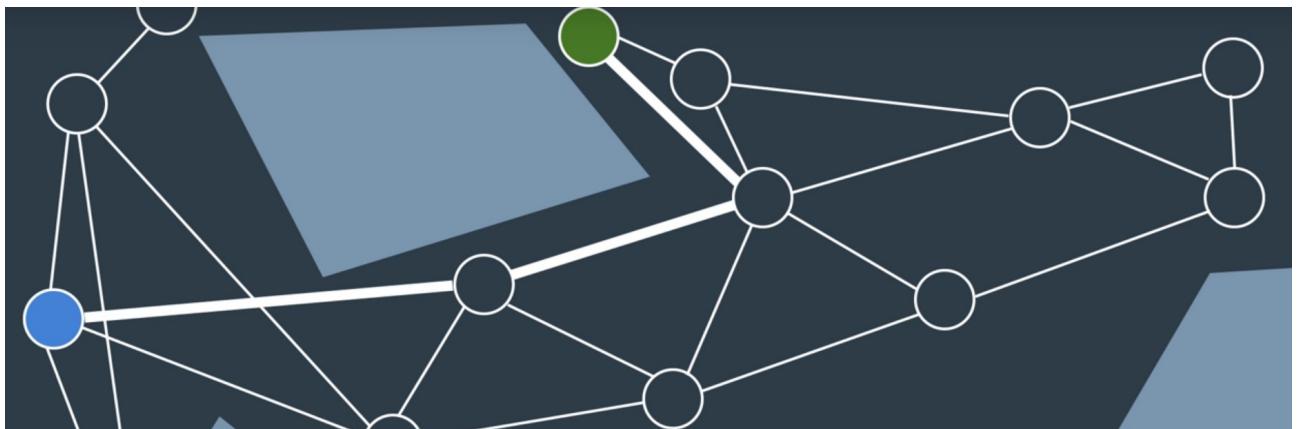
When you look at the paths produced by PRM and RRT, there are by no means optimal and can be quite jerky.

Instead of using these paths directly some post-processing can be applied to smooth out the paths and improve the results. One simple algorithm that can be used, is often **referred to as a path shortcutter**, it looks for ways to shorten the resulting path, by connecting two non-neighboring nodes together.

If it is able to find a pair of nodes whose edge is collision free, then the original path between the two nodes is replaced with the shortcut edge.



If this process is successful, then a search algorithm like A\* can be applied to find the path from start to goal. The resultant path may not be optimal, but it proves that moving from start to goal, is feasible.



## **Algorithm:**

The following algorithm provides a method for smoothing the path by shortcutting.

**For** n iterations:

**Select** two nodes from the graph

**If** the edge between the two nodes is shorter than the existing path  
between the nodes:

**Use local planner** to see if edge is collision-free.

**If** collision-free:

**Replace** existing path with edge between the two nodes.

Keep in mind that the path's distance is not the only thing that can be optimized by the Path Shortcutter algorithm - it could optimize for path smoothness, expected energy use by the robot, safety, or any other measurable factor.

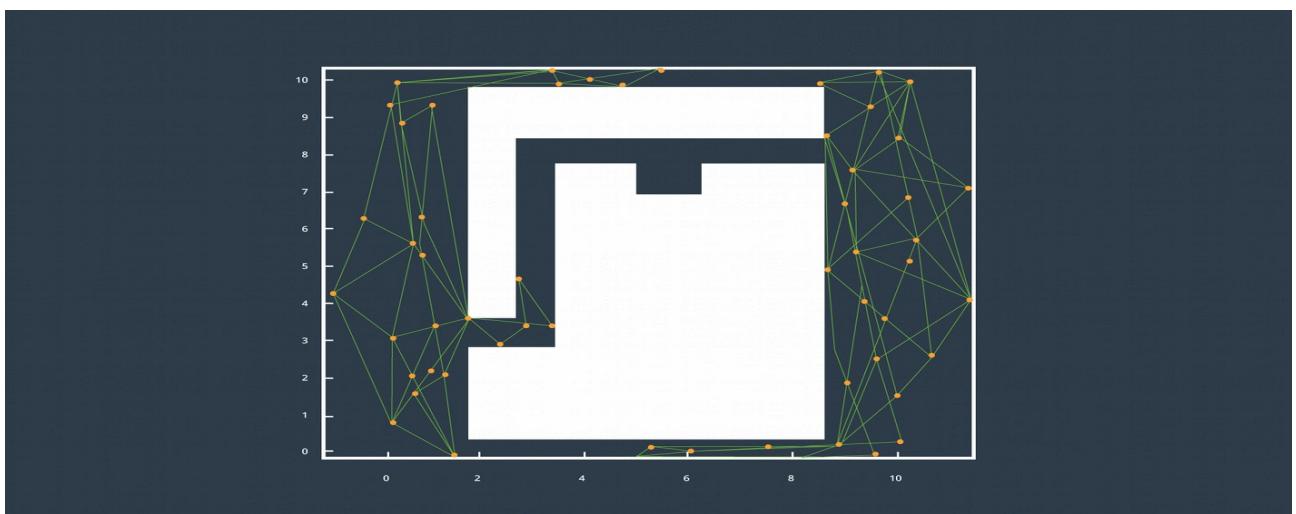
After the Path Shortcutting algorithm is applied, the result is a more optimized path. It may still not be *the optimal path*, but it should have at the very least moved towards a local minimum. There exist more complex, informed algorithms that can improve the performance of the Path Shortcutter. These are able to use information about the workspace to better guide the algorithm to a more optimal solution.

**For large multi-dimensional problems, it is not uncommon for the time taken to optimize a path to exceed the time taken to search for a feasible solution in the first place.**

## 6.4.8 Overall Concerns

### Not Complete

Sample-based planning is not complete, it is probabilistically complete. In applications where decisions need to be made quickly, PRM & RRT may fail to find a path in difficult environments, such as the one shown below.



To path plan in an environment such as the one presented above, alternate means of sampling can be introduced (such as Gaussian or Bridge sampling). Alternate methods bias their placement of samples to obstacle edges or vertices of the open space.

### Not Optimal

Sample-based path planning isn't optimal either - while an algorithm such as A\* will find the most optimal path within the graph, the graph is not a thorough representation of the space, and so the true optimal path is unlikely to be represented in the graph.

## Conclusion

Overall, there is no silver bullet algorithm for sample-based path planning. The PRM & RRT algorithms perform acceptably in most environments, while others require customized solutions. An algorithm that sees a performance improvement in one application, is not guaranteed to perform better in others.

**Ultimately, sample-based path planning makes multi-dimensional path planning feasible!**

### 6.4.9 Sample-Based Planning Wrap-Up

This concludes our discussion of sample-based path planning, as we saw in previous few concepts probabilistic roadmaps and rapidly exploring random trees are two alternate algorithms for path planning which are especially applicable in large high-dimensional spaces. Although the algorithms are not complete, they are considered to be probabilistically complete. As their completeness grows exponentially with the number of samples collected. The algorithms that we learned are incredibly applicable in real world robotics.

### Extended Reading

At this point, you have the knowledge to read through a paper on path planning. The following paper, [Path Planning for Non-Circular Micro Aerial Vehicles in Constrained Environments](#), addresses the problem of path planning for a quadrotor.

It is an enjoyable read that culminates the past two sections of path planning, as it references a number of planning methods that you have learned, and introduces a present-day application of path planning. Reading the paper will help you gain an appreciation of this branch of robotics, as well as help you gain confidence in the subject.

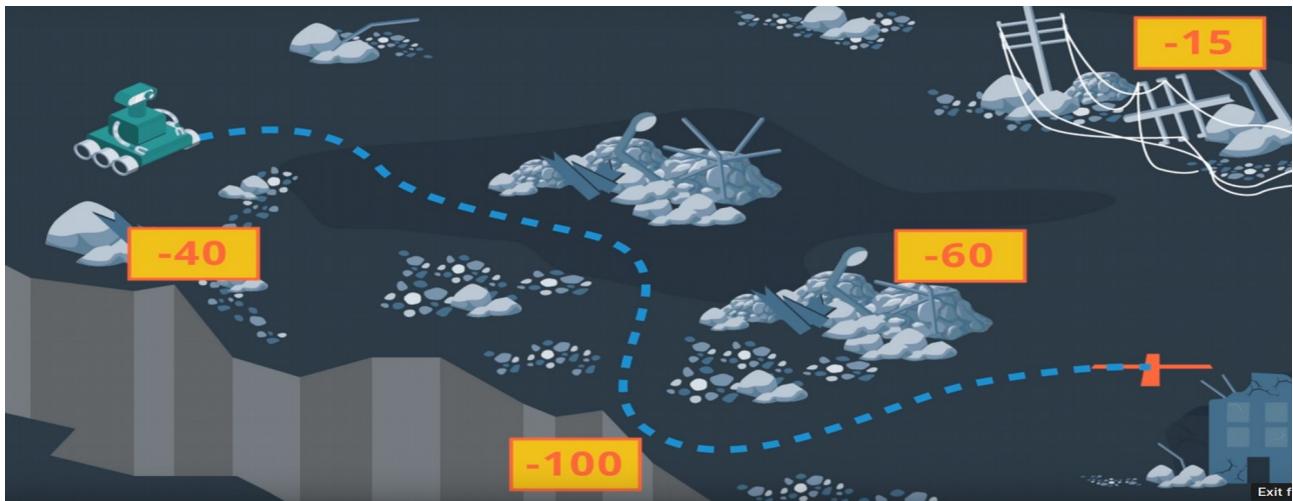
Some additional definitions that you may find helpful while reading the paper:

- **Anytime algorithm:** an anytime algorithm is an algorithm that will return a solution even if its computation is halted before it finishes searching the entire space. The longer the algorithm plans, the more optimal the solution will be.
- **RRT\*:** RRT\* is a variant of RRT that tries to smooth the tree branches at every step. It does so by looking to see whether a child node can be swapped with its parent (or its parent's parent, etc) to produce a more direct path. The result is a less zig-zaggy and more optimal path.

## 6.5 Probabilistic Path Planning

### 6.5.1 Introduction to Probabilistic Path Planning

Recall the exploratory rover introduced at the start of this chapter, its task was to find a path from its drop-off location to the goal location that would be safe for humans to follow.



The terrain contains a lot of different hazards, the operator of the rover is willing to take whatever risk is necessary but would naturally want to minimize it as much as possible. **The algorithms we have introduced thus far are unable to adequately model the risk.** For instance, a combinatorial path planning algorithm would have no difficulty finding this path to the goal location, the path may be the best by all other means but due to the uncertainty of the rover motion there is a chance that the rover would meet its demise along this path.

It is possible to inflate the size of the rover to ensure that there is enough room to maneuver, but as we have seen, the algorithm would no longer be complete.

Another idea is to give negative rewards to dangerous areas of the map so that the search algorithm is more likely to select alternative paths. Similar to reinforcement learning.

This is a step in the right direction, and would cause the rover to avoid dangerous areas, but it does not actually consider the uncertainty of the rover motion.

What we'd really like is to model the uncertainty by considering a non-deterministic transition model.

For instance, the following one



**Since path execution is uncertain, an algorithm that takes the uncertainty into account explicitly is more likely to produce realistic paths. a.k.a Markov Decision Processes.**

### 6.5.2 Markov Decision Process

A recycling robot goal is to drive around its environment and pick up as many cans as possible. It has a set of states that it could be in, and a set of actions that it could take. The robot will receive a reward for picking up cans, however, it will also receive a negative reward (a penalty) if it were to run out of battery and get stranded.

The robot had a non-deterministic **transition model** (sometimes called the *one-step dynamics*). This means that an action cannot guarantee to lead a robot from one state to another state. Instead, there is a probability associated with resulting in each state.

Say at an arbitrary time step  $t$ , the state of the robot's battery is high ( $S_t=\text{high}$ ). In response, the agent decides to search for cans ( $A_t=\text{search}$ ). In such a case, there is a 70% chance of the robot's battery charge remaining high and a 30% chance that it will drop to low.

Let's see the definition of an MDP before moving forward.

## MDP Definition

A Markov Decision Process is defined by:

- A set of states:  $S$ ,
- Initial state:  $s_0$ ,
- A set of actions:  $A$ ,
- The transition model:  $T(s,a,s')$ ,
- A set of rewards:  $R$ .

The transition model is the probability of reaching a state  $s'$  from a state  $s$  by executing action  $a$ . It is often written as  $T(s,a,s')$ .

The Markov assumption states that the probability of transitioning from  $s$  to  $s'$  is only dependent on the present state,  $s$ , and not on the path taken to get to  $s$ .

One notable difference between MDPs in probabilistic path planning and MDPs in reinforcement learning, is that in path planning the robot is fully aware of all of the items listed above (state, actions, transition model, rewards). Whereas in RL, the robot was aware of its state and what actions it had available, but it was not aware of the rewards or the transition model.

## Mobile Robot Example

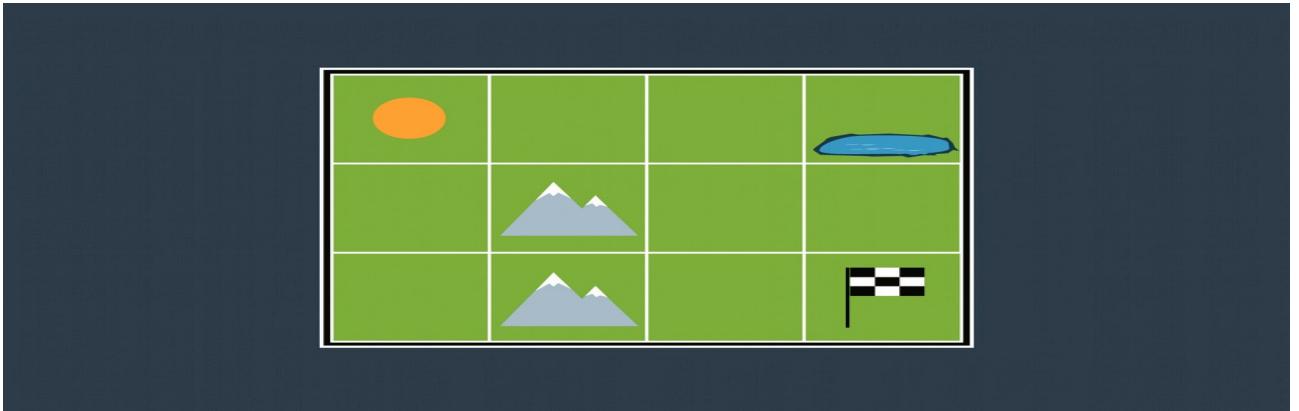
In our mobile robot example, movement actions are non-deterministic. Every action will have a probability less than 1 of being successfully executed. This can be due to a number of reasons such as wheel slip, internal errors, difficult terrain, etc. The image below showcases a possible transition model for our exploratory rover, for a scenario where it is trying to move forward one cell.



As you can see, the intended action of moving forward one cell is only executed with a probability of 0.8 (80%). With a probability of 0.1 (10%), the rover will move left, or right. Let's also say that bumping into a wall will cause the robot to remain in its present cell.

Let's provide the rover with a simple example of an environment for it to plan a path in. The environment shown below has the robot starting in the top left cell, and the robot's goal is in the bottom right cell. The mountains represent terrain that is more difficult to pass, while the pond is a

hazard to the robot. Moving across the mountains will take the rover longer than moving on flat land, and moving into the pond may drown and short circuit the robot.



## Combinatorial Path Planning Solution

If we were to apply A\* search to this discretized 4-connected environment, the resultant path would have the robot move right 2 cells, then down 2 cells, and right once more to reach the goal (or R-R-D-R-D, which is an equally optimal path). This truly is the shortest path, however, it takes the robot right by a very dangerous area (the pond). There is a significant chance that the robot will end up in the pond, failing its mission.

If we are to path plan using MDPs, we might be able to get a better result!

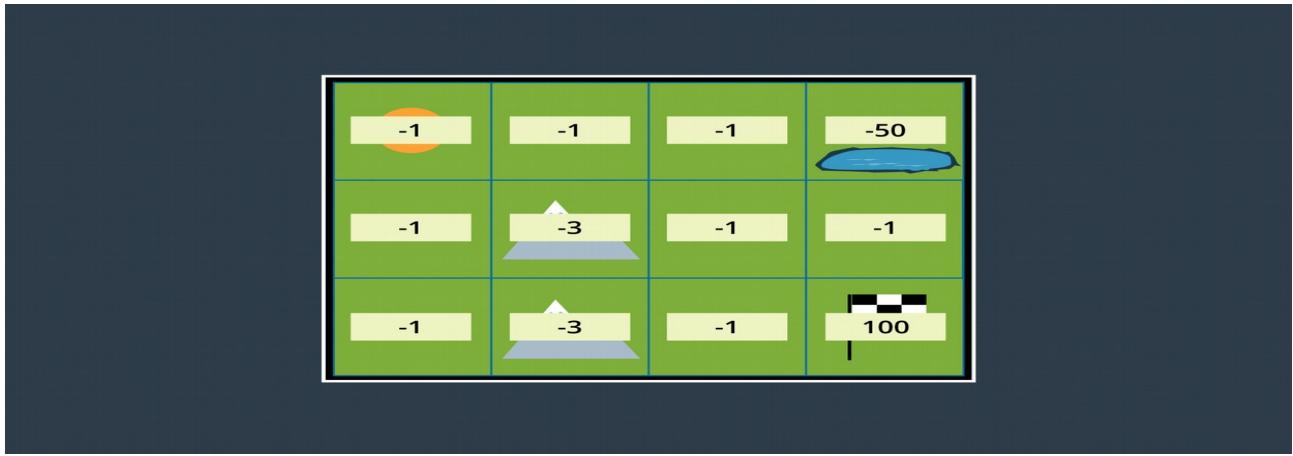
## Probabilistic Path Planning Solution

In each state (cell), the robot will receive a certain reward,  $R(s)$ . This reward could be positive or negative, but it cannot be infinite. It is common to provide the following rewards,

- small negative rewards to states that are not the goal state(s) - to represent the cost of time passing (a slow moving robot would incur a greater penalty than a speedy robot),
- large positive rewards for the goal state(s), and
- large negative rewards for hazardous states - in hopes of convincing the robot to avoid them.

These rewards will help guide the rover to a path that is efficient, but also safe - taking into account the uncertainty of the rover's motion.

The image below displays the environment with appropriate rewards assigned.



As you can see, entering a state that is not the goal state has a reward of -1 if it is a flat-land tile, and -3 if it is a mountainous tile. The hazardous pond has a reward of -50, and the goal has a reward of 100.

With the robot's transition model identified and appropriate rewards assigned to all areas of the environment, we can now construct a policy. Read on to see how that's done in probabilistic path planning!

### 6.5.3 Policies

#### Policies

A solution to a Markov Decision Process is called a policy, and is denoted with the letter  $\pi$ .

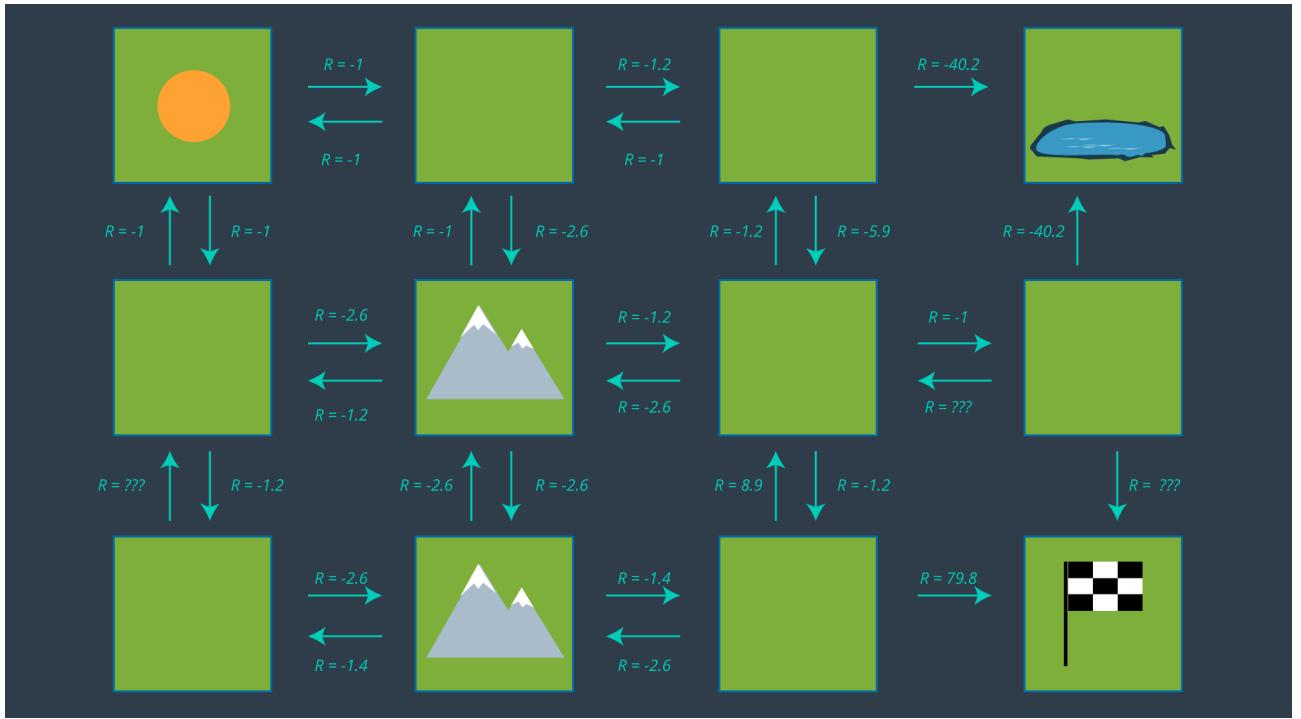
#### Definition

A **policy** is a mapping from states to actions. For every state, a policy will inform the robot of which action it should take. An **optimal policy**, denoted  $\pi^*$ , informs the robot of the *best* action to take from any state, to maximize the overall reward. We'll explore optimal policies in more detail below.

If you aren't comfortable with policies, it is highly recommended to study a Gridworld Example, State-Value Functions, and Bellman Equations. These demonstrate what a policy is, how state-value is calculated, and how the Bellman equations can be used to compute the optimal policy.

#### Developing a Policy

The image below displays the set of actions that the robot can take in its environment. Note that there are no arrows leading away from the pond, as the robot is considered DOA (dead on arrival) after entering the pond. As well, no arrows leave the goal as the path planning problem is complete once the robot reaches the goal - after all, this is an *episodic task*.



From this set of actions, a policy can be generated by selecting one action per state. Before we visit the process of selecting the appropriate action for each policy, let's look at how some of the values above were calculated. After all, -5.9 seems like quite an odd number!

## Calculating Expected Rewards

Recall that the reward for entering an empty cell is -1, a mountainous cell -3, the pond -50, and the goal +100. These are the rewards defined according to the environment. However, if our robot wanted to move from one cell to another, it is not guaranteed to succeed. Therefore, we must calculate the **expected reward**, which takes into account not just the rewards set by the environment, but the robot's transition model too.

Let's look at the bottom mountain cell first. From here, it is intuitively obvious that moving right is the best action to take, so let's calculate that one. If the robot's movements were deterministic, the cost of this movement would be trivial (moving to an open cell has a reward of -1). However, since our movements are non-deterministic, we need to evaluate the *expected* reward of this movement. The robot has a probability of 0.8 of successfully moving to the open cell, a probability of 0.1 of moving to the cell above, and a probability of 0.1 of bumping into the wall and remaining in its present cell.

$$\text{Expected reward} = 0.8 * (-1) + 0.1 * (-3) + 0.1 * (-3) \text{ expected reward} = -1.4$$

All of the expected rewards are calculated in this way, taking into account the transition model for this particular robot.

You may have noticed that a few expected rewards are missing in the image above. Can you calculate their values?

## Selecting a Policy

Now that we have an understanding of our expected rewards, we can select a policy and evaluate how efficient it is. Once again, a policy is just a mapping from states to actions. If we review the set of actions depicted in the image above, and select just one action for each state - i.e. exactly one arrow leaving each cell (with the exception of the hazard and goal states) - then we have ourselves a policy.

However, we're not looking for *any* policy, we'd like to find the *optimal* policy. For this reason, we'll need to study the utility of each state to then determine the *best* action to take from each state. That's what the next concept is all about!

### 6.5.4 State Utility

#### Definition

The **utility of a state** (otherwise known as the **state-value**) represents how attractive the state is with respect to the goal. Recall that for each state, the state-value function yields the expected return, if the agent (robot) starts in that state and then follows the policy for all time steps. In mathematical notation, this can be represented as so:

$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} R(s_t) | \pi, s_0 = s\right]$$

The notation used in path planning differs slightly from what is used in Reinforcement Learning. But the result is identical.

Here,

- $U^\pi(s)$  represents the utility of a state  $s$ ,
- $E$  represents the *expected* value, and
- $R(s)$  represents the reward for state  $s$ .

The utility of a state is the sum of the rewards that an agent would encounter if it started at that state and followed the policy to the goal.

#### Calculation

We can break the equation down, to further understand it.

$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} R(s_t) | \pi, s_0 = s\right]$$

Let's start by breaking up the summation and explicitly adding all states.

$$U^\pi(s) = E[R(s_0) + R(s_1) + R(s_2) + \dots | \pi, s_0 = s]$$

Then, we can pull out the first term. The expected reward for the first state is independent of the policy. While the expected reward of all future states (those between the state and the goal) depend on the policy.

$$U^\pi(s) = E[R(s_0)|s_0 = s] + E[R(s_1) + R(s_2) + \dots | \pi]$$

Re-arranging the equation results in the following. (Recall that the prime symbol, as on  $s'$ , represents the next state - like  $s_2$  would be to  $s_1$ ).

$$U^\pi(s) = R(s) + E[\sum_{t=0}^{\infty} R(s_t) | \pi, s_0 = s']$$

Ultimately, the result is the following.

$$U^\pi(s) = R(s) + U^\pi(s')$$

As you see here, calculating the utility of a state is an iterative process. It involves all of the states that the agent would visit between the present state and the goal, as dictated by the policy.

As well, it should be clear that the utility of a state depends on the policy. If you change the policy, the utility of each state will change, since the sequence of states that would be visited prior to the goal may change.

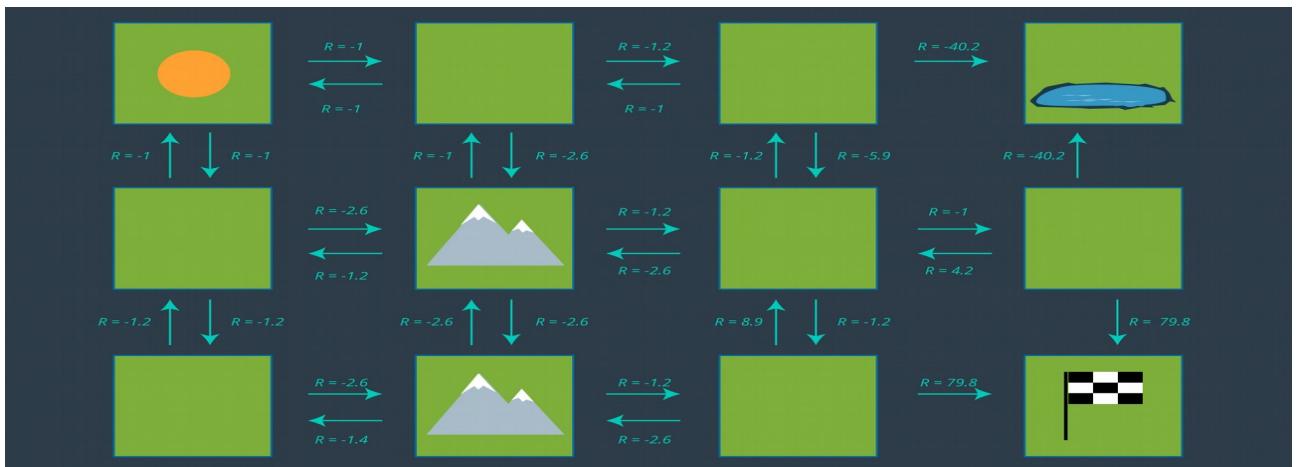
## Determining the Optimal Policy

Recall that the **optimal policy**, denoted  $\pi^*$ , informs the robot of the *best* action to take from any state, to maximize the overall reward. That is,

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} E[U^\pi(s)]$$

In a state  $s$ , the optimal policy  $\pi^*$  will choose the action  $a$  that maximizes the utility of  $s$  (which, due to its iterative nature, maximizes the utilities of all future states too).

While the math may make it seem intimidating, it's as easy as looking at the set of actions and choosing the best action for every state. The image below displays the set of all actions once more.



It may not be clear from the get-go which action is optimal for every state, especially for states far away from the goal which have many paths available to them. It's often helpful to start at the goal and work your way backwards.

If you look at the two cells adjacent to the goal, their best action is trivial - go to the goal! The goal state's utility is 0. This is because if the agent starts at the goal, the task is complete and no reward is received. Thus, the expected reward from either of the goal's adjacent cells is 79.8. Therefore, the state's utility is,  $79.8 + 0 = 79.8$  (based on  $U\pi(s) = R(s) + U\pi(s')$ ).

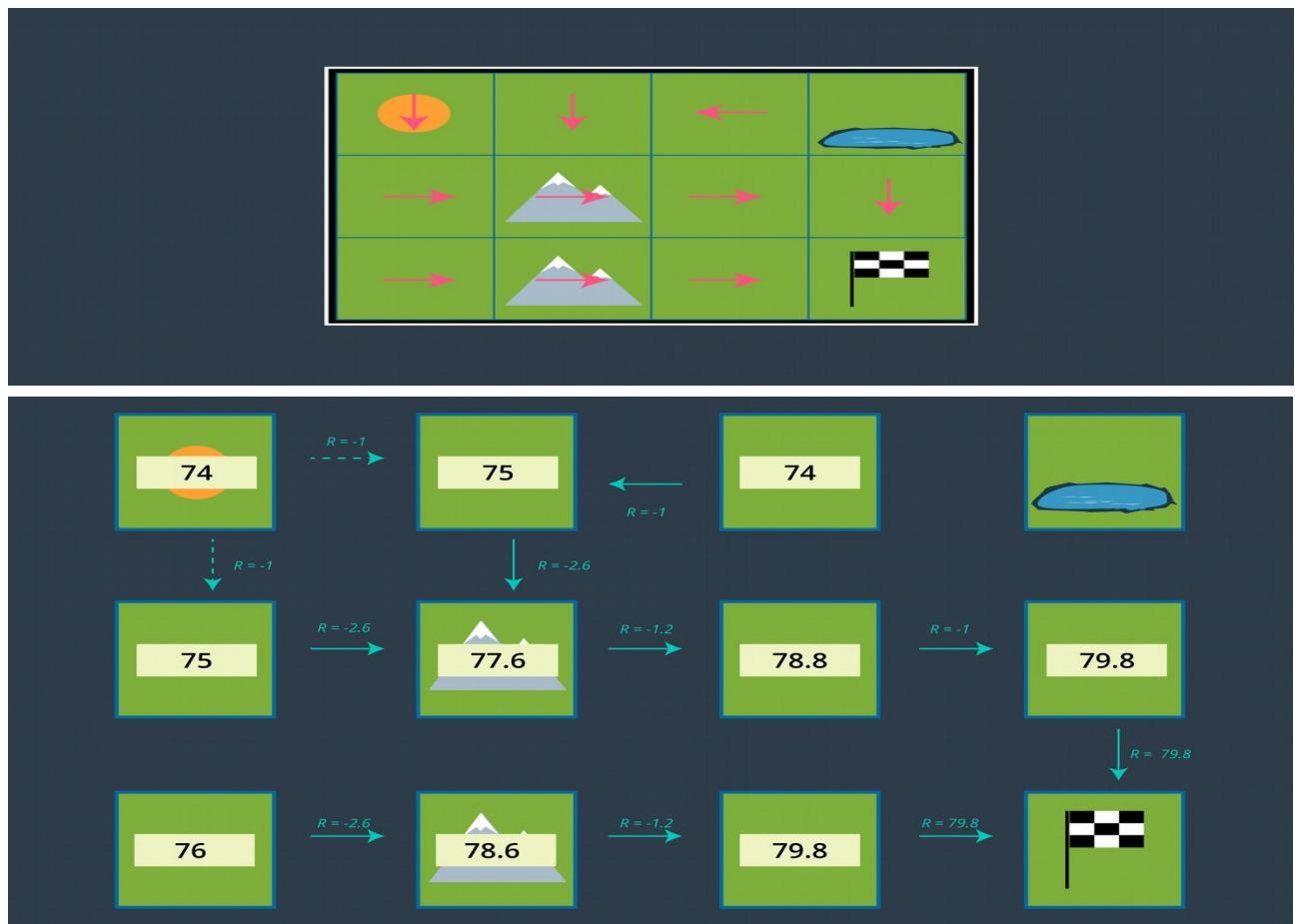
If we look at the lower mountain cell, it is also easy to guess which action should be performed in this state. With an expected reward of -1.2, moving right is going to be much more rewarding than taking any indirect route (up or left). This state will have a utility of  $-1.2 + 79.8 = 78.6$ .

The process of selecting each state's most rewarding action continues, until every state is mapped to an action. These mappings are precisely what make up the policy.

## Applying the Policy

Once this process is complete, the agent (our robot) will be able to make the best path planning decision from every state, and successfully navigate the environment from any start position to the goal. The optimal policy for this environment and this robot is provided below.

The image below shows the set of actions with just the optimal actions remaining. Note that from the top left cell, the agent could either go down or right, as both options have equal rewards.



## Discounting

One simplification that you may have noticed us make, is omit the discounting rate  $\gamma$ . In the above example,  $\gamma=1$  and all future actions were considered to be just as significant as the present action. This was done solely to simplify the example.

In reality, discounting is often applied in robotic path planning, since the future can be quite uncertain. The complete equation for the utility of a state is provided below:

$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi, s_0 = s\right]$$

### 6.5.5 Value Iteration Algorithm

The process that we went through to determine the optimal policy for the mountainous environment was fairly straightforward, but it did take some intuition to identify which action was optimal for every state. In larger more complex environments, intuition may not be sufficient. In such environments, an algorithm should be applied to handle all computations and find the optimal solution to an MDP. One such algorithm is called the Value Iteration algorithm. *Iteration* is a key word here, and you'll see just why!

The Value Iteration algorithm will initialize all state utilities to some arbitrary value - say, zero. Then, it will iteratively calculate a more accurate state utility for each state, using

$$U(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U(s')$$

#### Algorithm

$U' = 0$

loop until *close-enough*( $U$ ,  $U'$ )

$U = U'$

for  $s$  in  $S$ , do:

$$U(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U(s')$$

return  $U$

With every iteration, the algorithm will have a more and more accurate estimate of each state's utility. The number of iterations of the algorithm is dictated by a function close-enough which detects convergence. One way to accomplish this is to evaluate the root mean square error,

$$RMS = \frac{1}{|S|} \sqrt{\sum_s (U(s) - U'(s))^2}$$

Once this error is below a predetermined threshold, the result has converged sufficiently.

$$RMS = \frac{1}{|S|} \sqrt{\sum_s (U(s) - U'(s))^2}$$

This algorithm finds the optimal policy to the MDP, regardless of what  $U'$  is initialized to (although the efficiency of the algorithm will be affected by a poor  $U'$ ).

### 6.5.6 Probabilistic Path Planning Wrap-Up

Great, we've seen how Markov decision processes can be applied to path planning, to generate a policy, and how the policy can be used to guide the rover from any state to the goal.

We've learned about the value iteration algorithm that is able to find the optimal policy, and now we're ready to put an algorithm into action.

# 7. 3D Perception

## 7.1 Introduction Active and Passive Sensors and specs

We perceive the world around us in three dimensions, we are able to do this partly because we have two eyes and our brain is constantly performing a stereo vision comparison between what we see in one eye versus the other, but also because we have prior knowledge of what the world should look like.

Robots don't necessarily know what the world should look like but they can be outfitted with cameras that function like our eyes to perceive the world, but inferring depth from 2-D images is tricky. The most robust way to achieve 3-D vision in robotics is to directly measure the distance to objects in the field of view, something that we cannot do with our eyes.

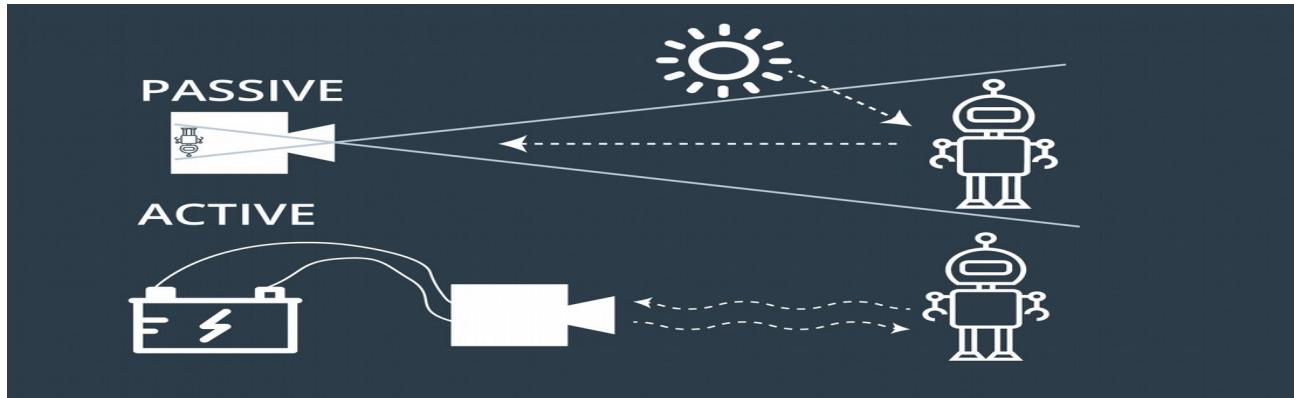
That can be measured by a variety of sensors including Radar, Lidar and even Sonar, But on this chapter we will focus on using data from **RGB-D cameras** to achieve feature rich perception in three dimensions.

Next, we'll be making a quick comparison of the trade-offs between these various 3-D sensors and then we'll explore how RGB-D cameras work and how to optimally handle the data.

### Active vs. Passive Sensors

Before we jump into active sensors right away, lets take a look at both active and passive sensors. This way we can get an idea of each and carry that information with us as we learn about both!

Active Sensing	Passive Sensing
Active Sensing techniques involve the use of an energy source to probe the environment. The sensors that utilize active sensing emit some form of energy (light or sound) into the scene and measure the reflected energy as a way of understanding the environment.	Contrary to its active counterpart, Passive Sensing refers to the measurement of energy already present in the environment. For instance, in the case of cameras that energy is light from the sun or other source reflected off objects in the scene and then into the camera



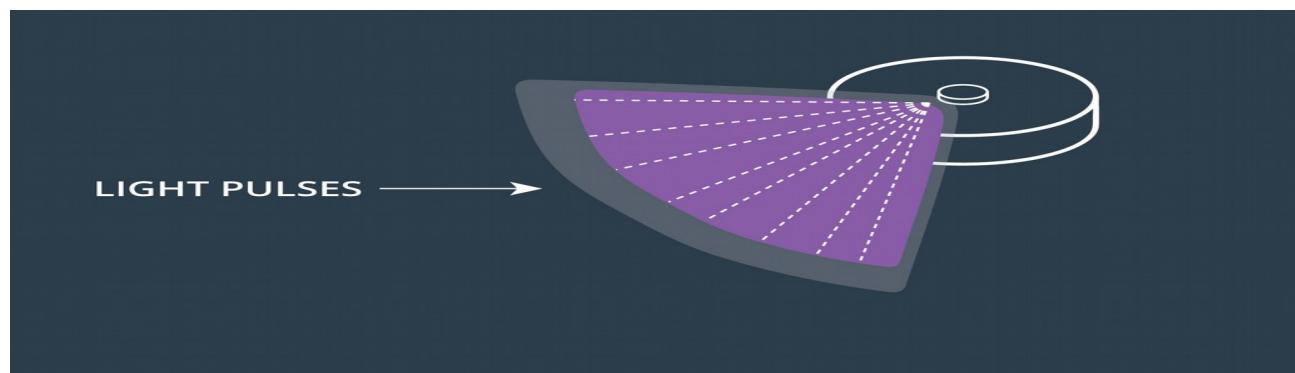
To better understand these two modalities, let us take a quick look at a few sensors. We will be exploring not only how each sensor functions, but also some of the pros and cons associated with each.

## Active Sensors

### Laser Range Finder (Lidar)

3D Laser Range Finders or 3D laser scanners are active sensors developed based on the Light Detection and Ranging ([Lidar](#)) method. Meaning, these sensors illuminate the target with a pulsed laser and measure the reflected pulses.

Since the laser frequency is a known stable quantity, the distance to objects in the field of view is calculated by measuring the time from when the pulse was sent to when it was received.



Pros	Cons
High spatial resolution in the horizontal plane	Large in size and bulky
High accuracy	High cost
High range	Affected by adverse weather

## Sensor Specifications

Cost	Range	Resolution
\$\$\$	~100m	High

### Time of Flight camera

A 3D time of flight camera is an active sensor that performs depth measurements by illuminating an area with an infrared light source and observing the time it takes to travel to the scene and back.

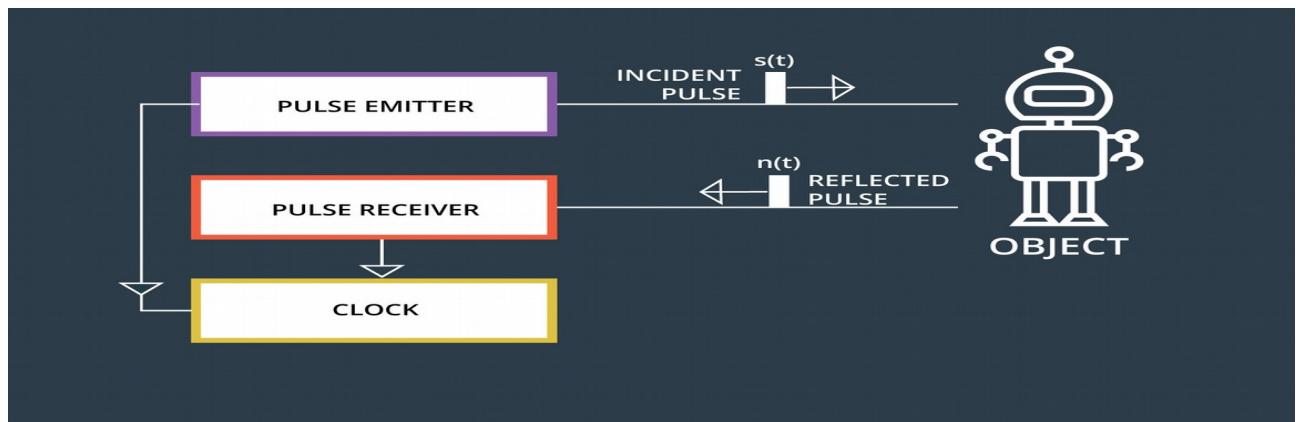
However, unlike a Laser Range Finder, a **ToF camera captures entire Field of View** with each light pulse without any moving parts. This allows for rapid data acquisition.

Based on their working principle, ToF sensors can be divided into two categories; **pulse runtime** and **phase shift continuous wave** (for a more detailed discussion check out [this slide deck](#)).

#### Pulse Runtime Sensor

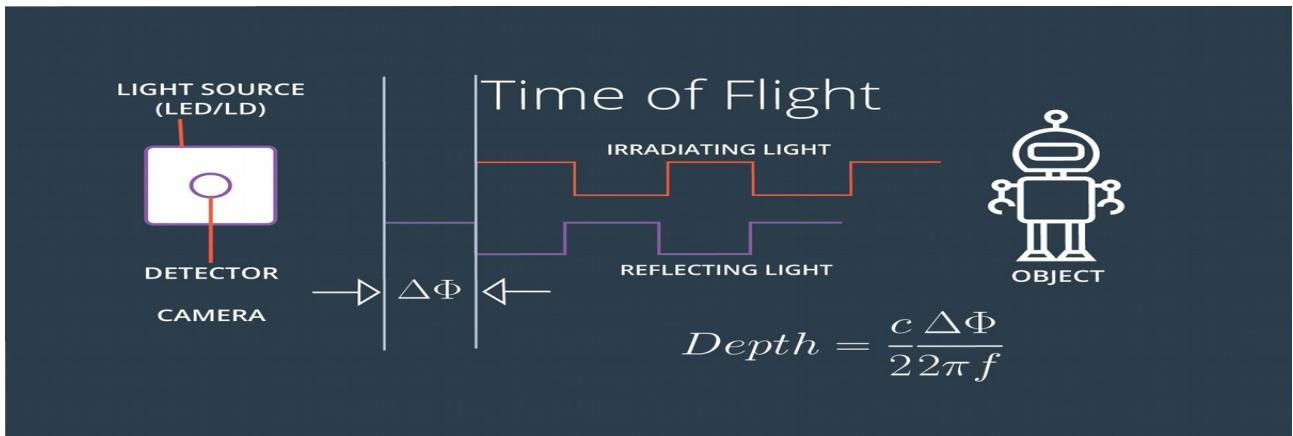
A pulse runtime sensor sends out a pulse of light and starts a timer, then it waits until a reflection is detected and stops the timer, thus directly measuring the light's time of flight.

Although intuitive and simple conceptually, this **technique requires very accurate hardware for timing**.



#### Phase Shift Continuous Wave Sensor

Unlike the Pulse runtime sensor, Phase Shift Continuous Wave sensor emits a continuous stream of modulated light waves. Here, the depth is calculated by measuring the phase shift of the reflected wave, creating a 3D depth map of the scene.



## Sensor Specifications

Cost	Range	Resolution
\$\$	~10m	High

Pros	Cons
Compact size	Secondary reflections
Rapid data acquisition	Ambient light interference (do not work well outdoors)
Ideal for real-time applications	Multiple ToF camera may interfere with one another
	Cannot easily detect glass

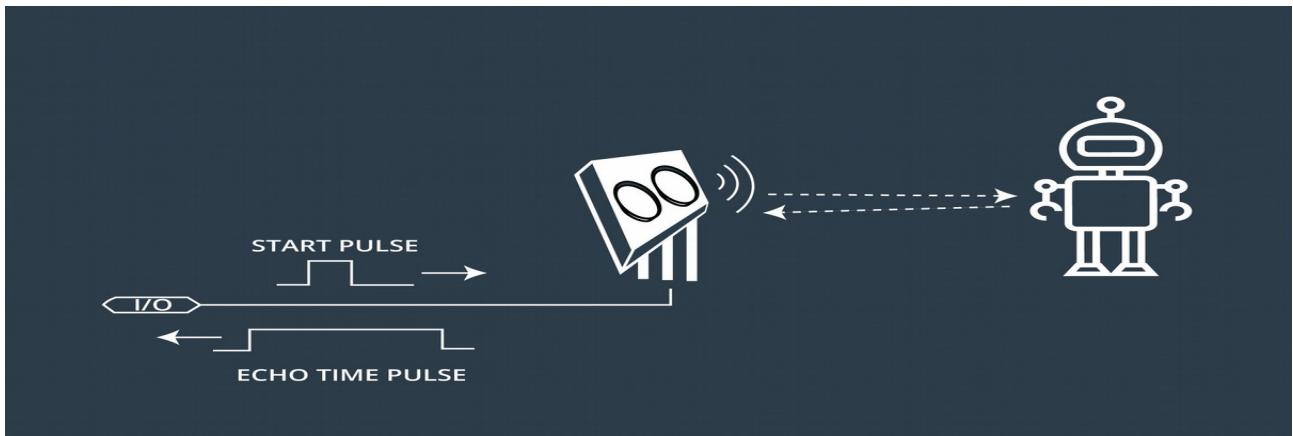
## Ultrasonic sensor

Much like a Laser Scanner uses laser pulses for distance measurement, an ultrasonic sensor uses high frequency sound pulses.

The Ultrasonic Sensor sends out a high-frequency sound pulse and then measures the time it takes for the echo of the sound to reflect back.

The sensor typically consists of two piezoelectric crystals, one acts as a transmitter (like a speaker) and another as a receiver (like a microphone).

Since the speed of sound in air is known (about 343 meters per second), the ultrasonic sensor calculates the distance from a target by timing how long it takes for the reflected pulse to reach the receiver.



## Sensor Specifications

Cost	Range	Resolution
\$	~0.2 to 5m	Low

Pros	Cons
High accuracy	Low spatial resolution
Relatively inexpensive	Short Range
Can detect glass	Interference with ambient noise or other sound sources
	Secondary reflections
	Certain object textures may cause sound absorption leading to faulty measurements
	Variation in temperature and/or pressure changes speed of sound resulting in imprecise measurements

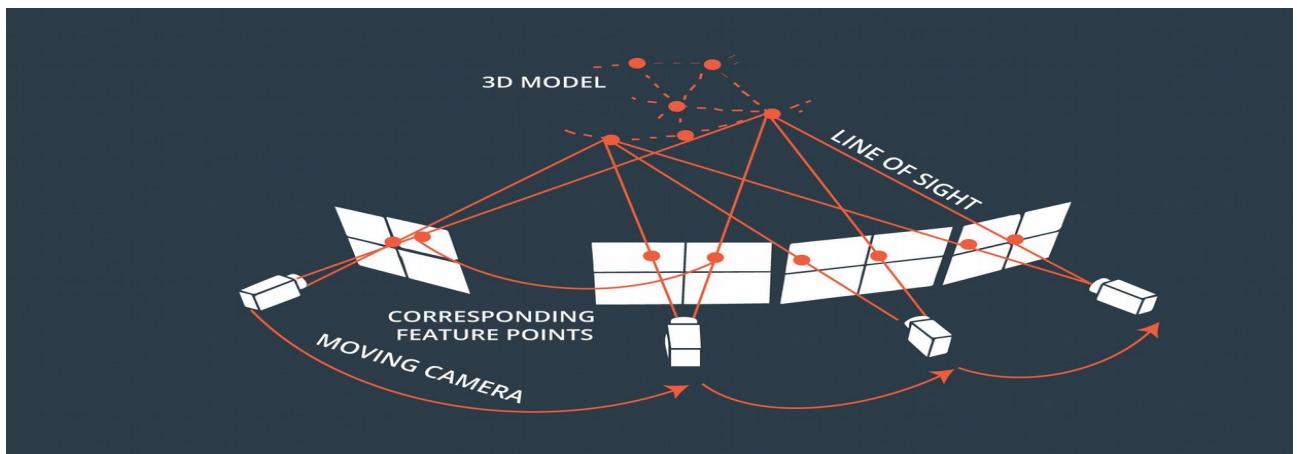
## Passive Sensors

### Monocular camera

You might be surprised to know that a simple monocular color camera can be used to infer depth data of a given scene. While non-trivial, it is possible to use software techniques to calculate depth from multiple 2D images.

One such technique is [Structure from Motion](#). In this approach, multiple images of a given object or scene are taken from a single moving camera to reconstruct a 3D model from the resulting video stream. Depth is calculated via triangulation technique, which requires accurate measurement of camera pose throughout its movement.

Turns out it's possible to improve the Structure from Motion result by combining these triangulation techniques with depth inference from single images. Check out this awesome paper by [Saxena, Chung and Ng 2007](#) for the details.

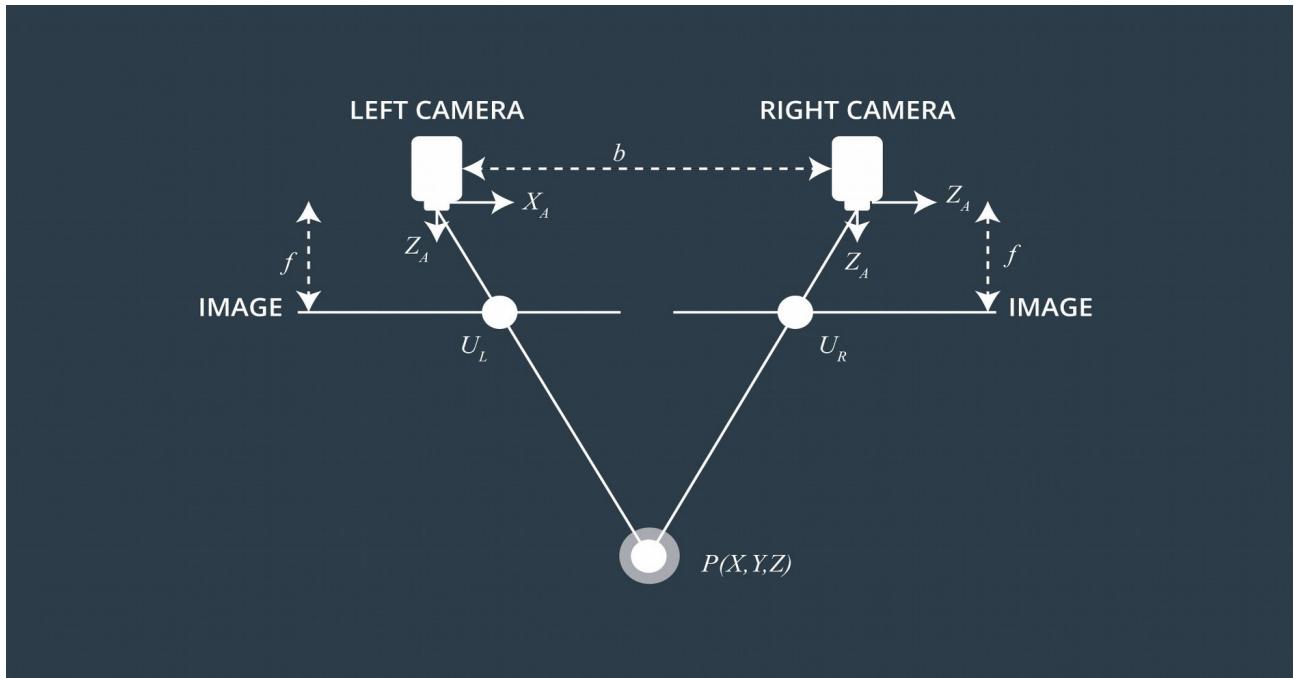


Pros	Cons
Provides feature rich color data	Inferring depth is computationally heavy
Texture	Performance varies with ambient light
Shape	Limited range
High spatial resolution	
Low cost	
Portable	

## Stereo Camera

A [stereo camera](#) system consists of two monocular cameras separated by an accurately known distance. Depth information is obtained by comparing image frames obtained from both cameras viewing the same object or scene.

The difference between position of a given object in the scene as perceived by the two cameras is called disparity. Stereo cameras, much like human eyes, leverage this disparity to calculate the depth data associated with a given object.



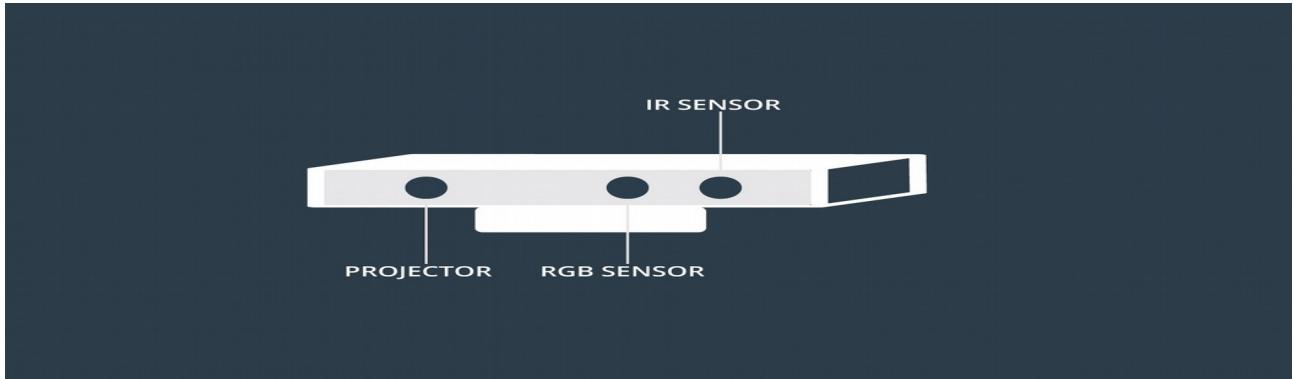
Pros	Cons
High spatial resolution in the horizontal plane	Inferring depth is computationally heavy
May provide color info in addition to depth	Performance varies with ambient light
Low cost compared to it's active counterparts	Limited range
Portable	Depth inference with this sensor usually depends upon image containing shapes and texture
Feature rich data	

## RGB-D Cameras

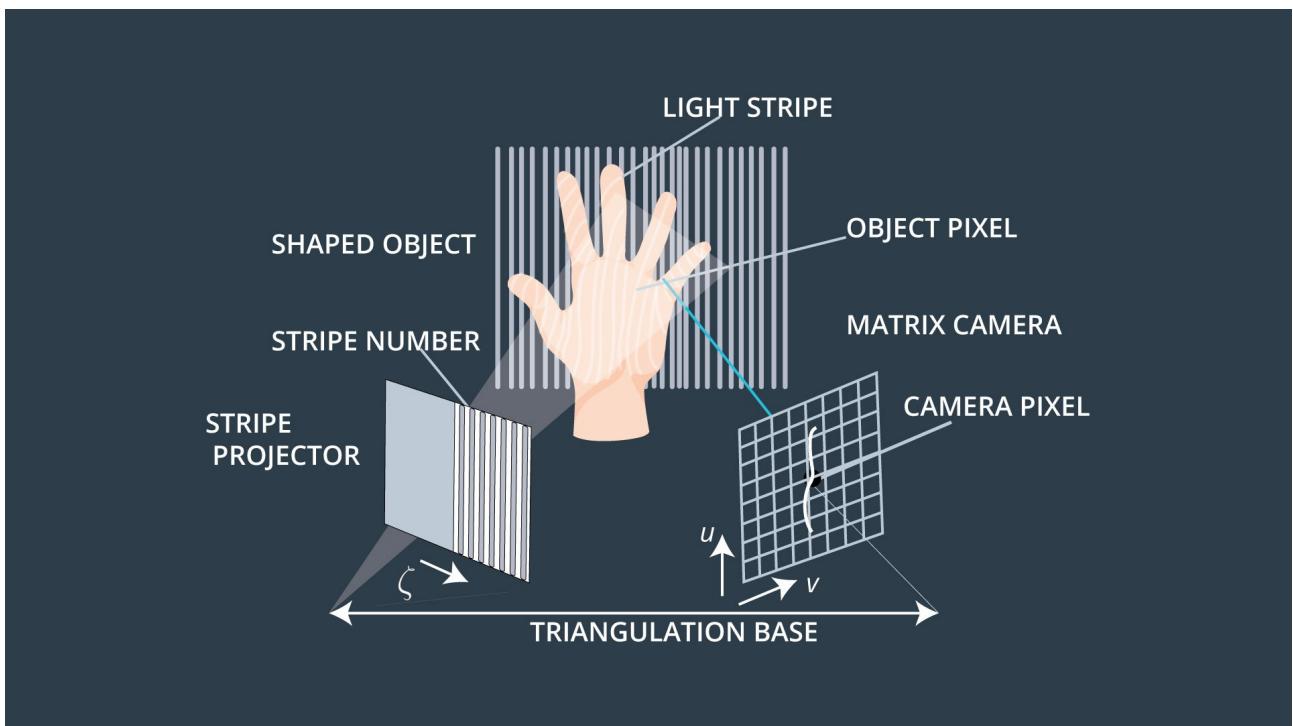
An RGB-D camera **combines best of the active and passive sensor worlds**, in that it consists of a passive RGB camera along with an active depth sensor. An RGB-D camera, unlike a conventional camera, provides **per-pixel depth information** in addition to an RGB image.

Traditionally, the active depth sensor is an infrared (IR) projector and receiver. Much like a Continuous Wave Time of Flight sensor, an RGB-D camera calculates depth by emitting a light signal on the scene and analyzing the reflected light, but the incident wave modulation is performed spatially instead of temporally.

Here we can see an example of a standard RGB-D Camera:



This is done by projecting light out of the IR transmitter in a predefined pattern and calculating the depth by interpreting the deformation in that pattern caused by the surface of target objects. These patterns range from simple stripes to unique and convoluted speckle patterns.



The advantage of using RGB-D cameras for 3D perception is that, unlike stereo cameras, they save a lot of computational resources by providing per-pixel depth values directly instead of inferring the depth information from raw image frames.

In addition, these sensors are inexpensive and have a simple USB plug and play interface. RGB-D cameras can be used for various applications ranging from mapping to complex object recognition.

## Sensor Specifications

Cost	Range	Resolution
\$	~0.2 to 5m	Low

### Adding Depth

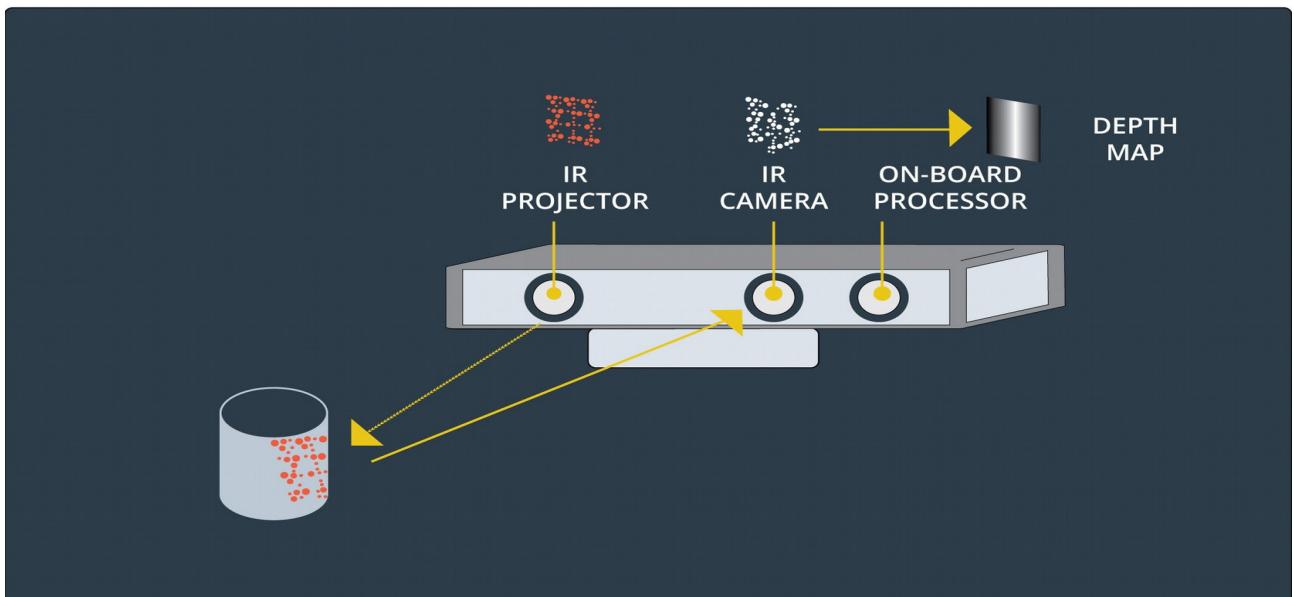
**Let's now have a look at the depth sensor to see how it enables an RGB-D camera to perceive the world in three dimensions.**

Most RGB-D cameras use a technique called Structured Light to obtain depth information from a scene.

This setup functions very similar to a stereo camera setup but there are a couple of major differences.

In a stereo setup, we calculate depth by comparing the disparity or difference between the images captured by Left and Right cameras. In contrast to that, a structured light setup contains one projector and one camera .

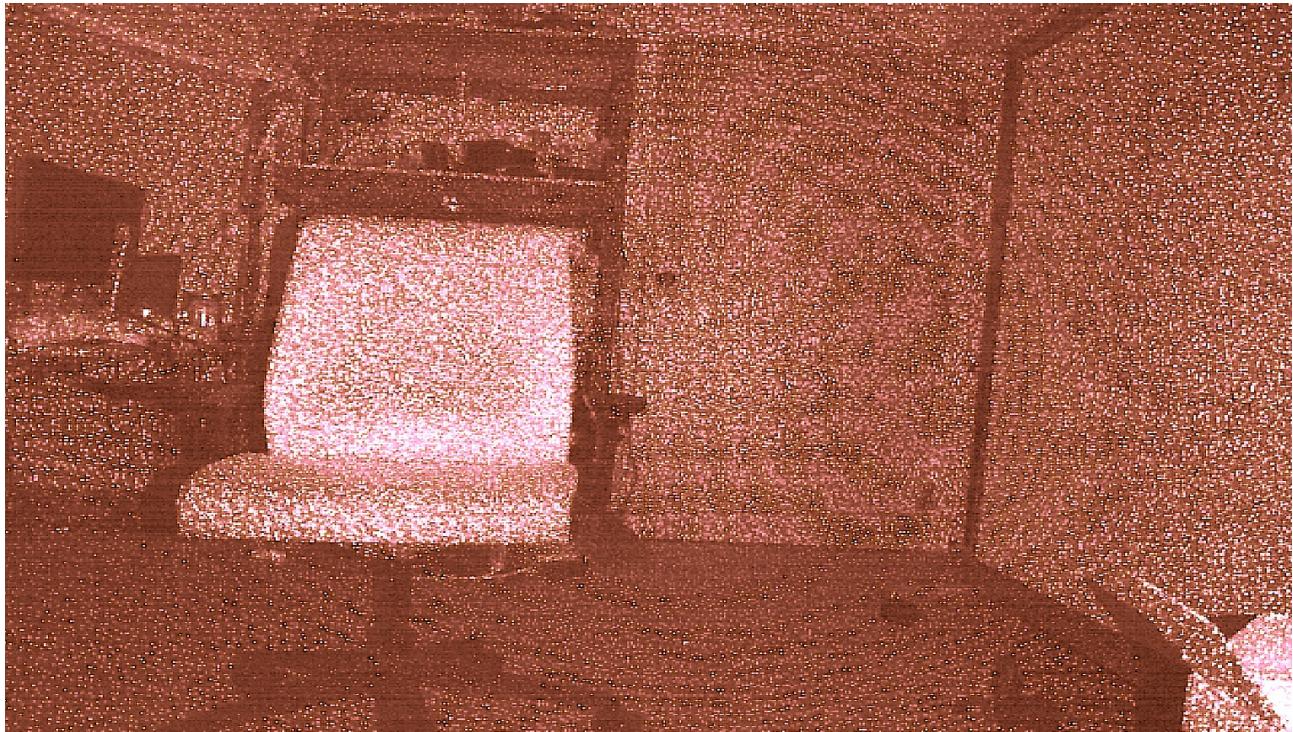
Here we can see how the depth map is generated from the RGB-D camera:



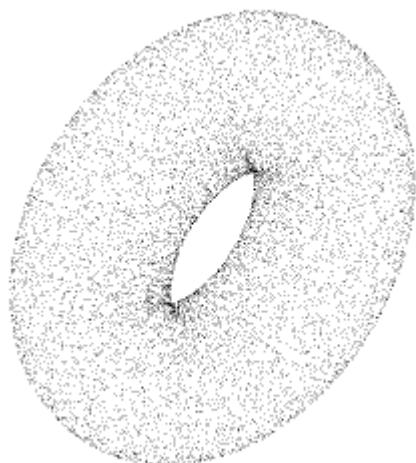
**The projector projects a known pattern on the scene. This pattern can be as simple as a series of stripes or a complex speckled pattern.**

The camera then captures the light pattern reflected off of the objects in the scene. The perceived pattern is distorted by the shape of objects in the scene. By performing a comparison between the known projected pattern (which is saved on the sensor hardware) and the reflected pattern, a depth map is generated much like a stereo camera.

Below is an example of a projection map that is placed over a scene in order to map out the correct pixel depths.



## 7.2 What is a Point Cloud



In the previous pages we've explored various techniques used by 3D sensors to infer depth information from a scene. Here, we will look at how we can represent that information as a point cloud, which is an abstract data type responsible for storing data from our RGB-D sensor.

Point clouds are digital representations of three dimensional objects. In practical implementations, they can also contain additional metadata for each point, as well as a number of useful methods for operating on the point cloud.

Examples of additional metadata might include

- RGB values for each point
- Intensity values
- Local curvature information

Additional methods provided might include

- Iterators for traversing points spatially
- Methods for filtering the cloud based on particular properties
- Operators for performing statistical analyses on the cloud



Nearly all 3D scanners or Lidars output data as high accuracy Point Clouds. Data from stereo cameras and RGB-D cameras can also be easily converted into Point Clouds. Point Clouds are used in numerous applications where 3D spatial information is a key component of the data. Some example applications include:

- Depth sensor measurements
- Models of real-world objects
- The extent of a robot's workspace
- Environment maps

As you can see, the Point Clouds are a convenient and useful way to represent spatial data! As for how point clouds are represented computationally, here is an example of the datatypes associated with each point of data from an RGB-D camera:

Attributes	Commonly used data type
x-coordinate	float
y-coordinate	float
z-coordinate	float
Red color value	Unsigned 8-bit int
Blue color value	Unsigned 8-bit int
Green color value	Unsigned 8-bit int

## Point Cloud Types

Every single point in a 3D Point Cloud contains certain attributes. While the x, y, and z coordinate information of a given point may be the most obvious attributes, points may also contain special attributes associated with the shape, color, or texture of the objects in the scene, depending on what type of sensor was used to record the data.

Various factors must be taken into consideration before choosing a point type for any given application. These include,

- Goals of your application
- Type of sensor used
- Nature of the objects in your scene
- Perception algorithms being used

In the upcoming lessons, you'll be using the powerful [Point Cloud Library \(PCL\)](#) to manipulate your point clouds, so let's explore some of the more widely used point types in the PCL.

### PointXYZ

This is the most commonly used point type. It represents a simple point in 3D space with no additional attributes.

Attributes	Commonly used data type
x-coordinate	float
y-coordinate	float
z-coordinate	float

## PointXYZI

Deriving from the PointXYZ class, this point type also includes an intensity attribute. Intensity is a measure of the amount of light reflected off from a point on the surface of an object in the scene.

Attributes	Commonly used data type
x-coordinate	float
y-coordinate	float
z-coordinate	float
Intensity	float

## PointXYZRGB

Another derivative of the simple PointXYZ class, which contains information from three color channels.

Attributes	Commonly used data type
x-coordinate	float
y-coordinate	float
z-coordinate	float
Red color value	Unsigned 8-bit int
Blue color value	Unsigned 8-bit int
Green color value	Unsigned 8-bit int

## Normal

One of the other widely used data types, this point type represents the surface normal at a given point, and a measure of curvature. More information about surface normals and curvature can be found [here](#).

Attributes	Commonly used data type
normal_x	float
normal_y	float
normal_z	float
curvature	float

### PointNormal

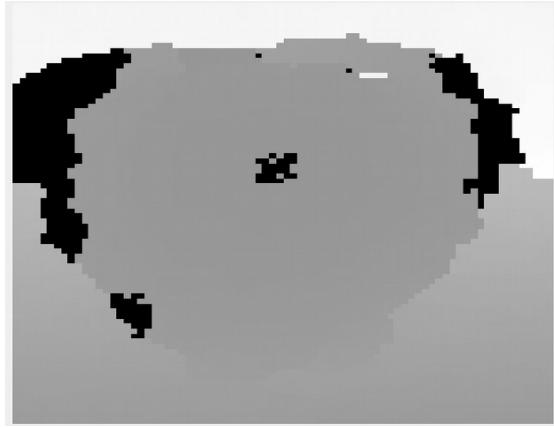
A point structure that holds XYZ data, together with surface normals and curvatures.

Attributes	Commonly used data type
x-coordinate	float
y-coordinate	float
z-coordinate	float
normal_x	float
normal_y	float
normal_z	float
curvature	float

### Point Cloud from RGB-D Data example

This is the RGB image:

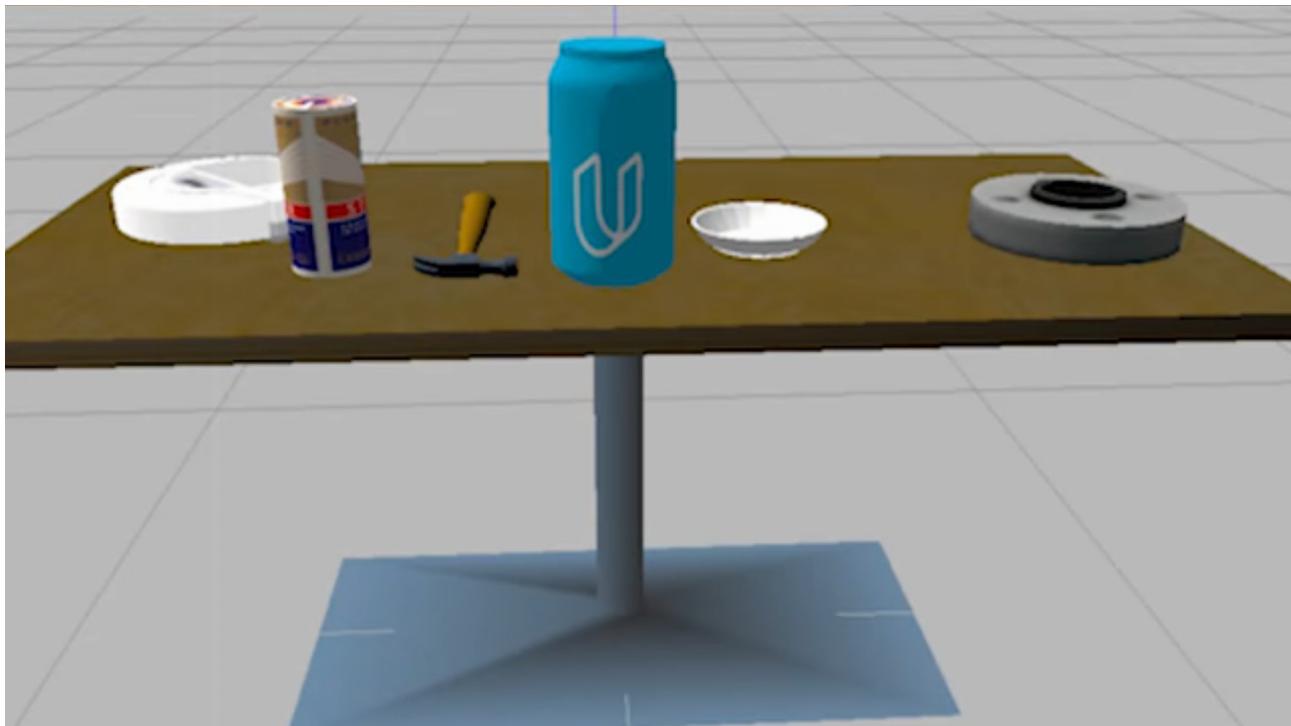
And here is the depth map:



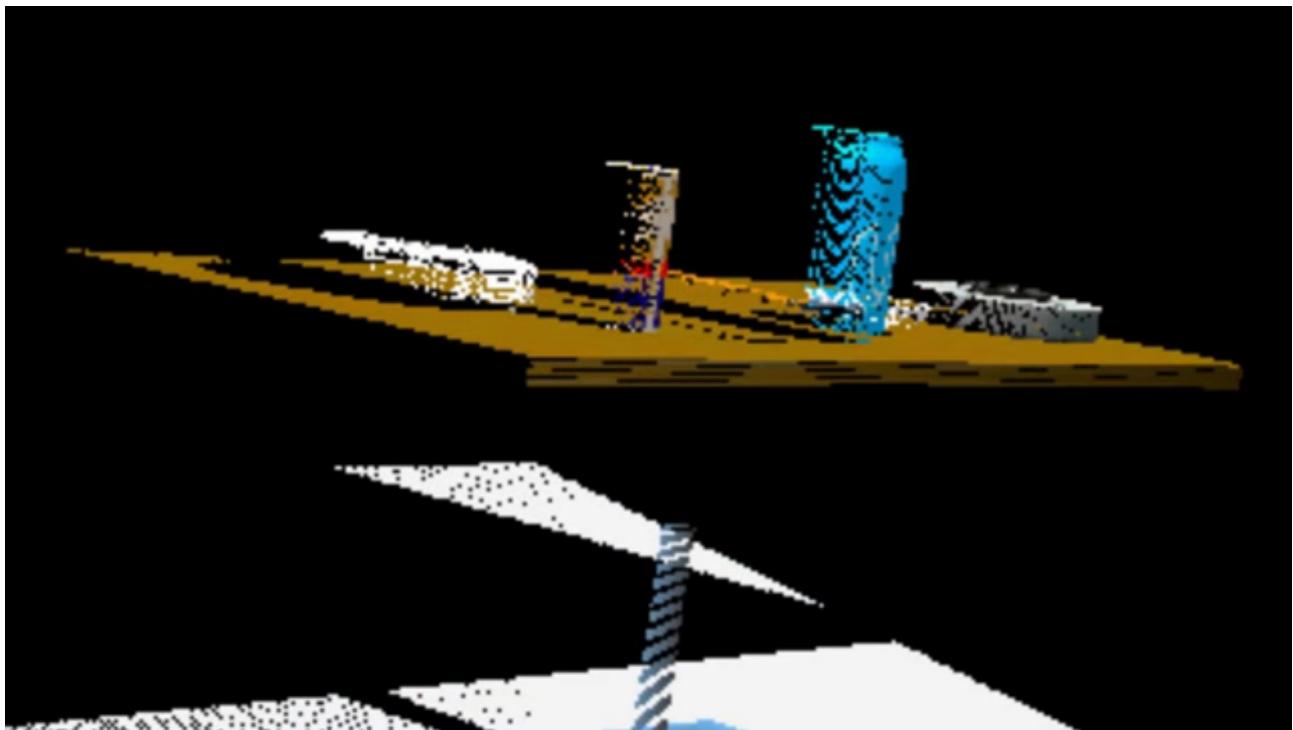
### 7.3 Point Cloud Filtering

To understand the series of actions we need to perform in order to be successful, let's start by examining the task at hand.

We'll implement a perception pipeline to identify our target object in a cluttered tabletop environment. Our robot's workspace will look something like this.



And this is what the point cloud representation of our robot's workspace would look like

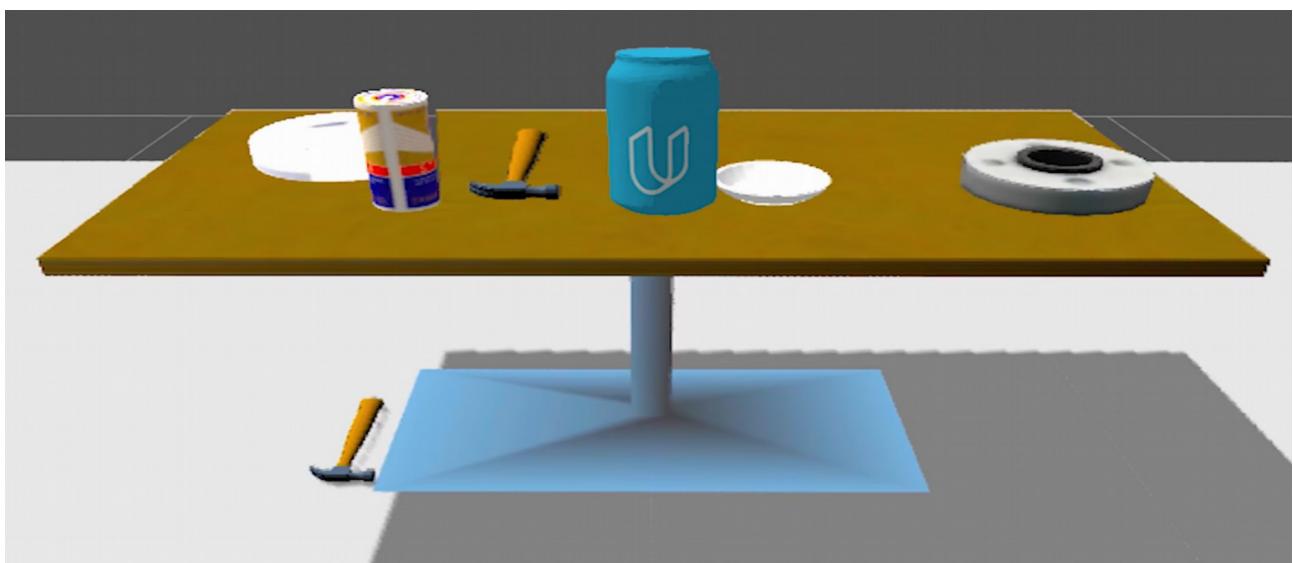


Let's say for example that our target object is the hammer on the table. Upon taking a closer look at this point cloud data, we'll discover that the majority of the data are not useful for identifying the target. Things like the ground, the table or the objects in the background do not aid in recognizing our target and, hence, can be regarded as noise for the most part.

Processing the complete point cloud including this excessive data is inefficient and leads to wastage of compute cycles. To alleviate this problem, we can use some simple techniques called point cloud filtering to remove the additional data points.

Filtering not only removes useless and excessive data but also the data that is deemed adversarial for a given application.

Adversarial data, in this case,



might correspond to an object on the ground that resembles your target object on the table. The presence of such an object in your point cloud data could induce false positives if it weren't filtered out.

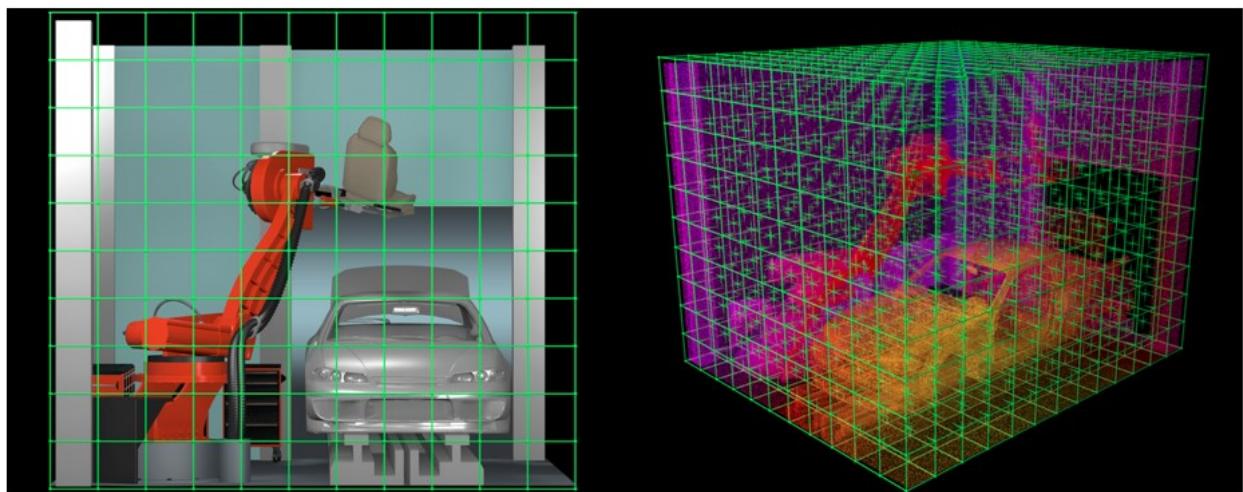
Next up, we'll explore some of the most commonly used filters from the Point Cloud Library. These filters are:

- VoxelGrid Downsampling Filter
- ExtractIndices Filter
- PassThrough Filter
- RANSAC Plane Fitting
- Outlier Removal Filter

While some of these are actual filters that use an underlying algorithm to improve the quality of your point cloud data, some are simple tools that are required at certain stages in your perception pipeline to aid in the extraction of a subset of the input point cloud.

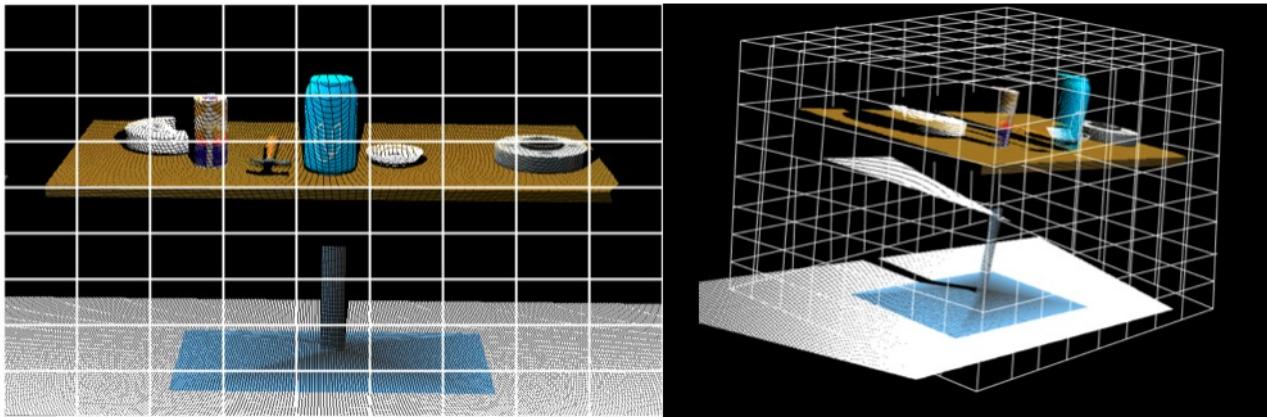
**For experimental result you can refer to the appendix  
Now let's take a closer look at this filters!**

## Voxel Grid Downsampling



RGB-D cameras provide feature rich and particularly dense point clouds, meaning, more points are packed in per unit volume than, for example, a Lidar point cloud. Running computation on a full resolution point cloud can be slow and may not yield any improvement on results obtained using a more sparsely sampled point cloud.

So, in many cases, it is advantageous to downsample the data. In particular, we are going to use a VoxelGrid Downsampling Filter to derive a point cloud that has fewer points but should still do a good job of representing the input point cloud as a whole.



The word "pixel" is short for "picture element". Similarly, the word "voxel" is short for "volume element". Just as you can divide the 2D image into a regular grid of area elements, as shown in the image on the left above, you can divide up your 3D point cloud, into a regular 3D grid of volume elements as shown on the right. Each individual cell in the grid is now a voxel and the 3D grid is known as a "voxel grid".

A voxel grid filter allows you to downsample the data by taking a spatial average of the points in the cloud confined by each voxel. You can adjust the sampling size by setting the voxel size along each dimension. The set of points which lie within the bounds of a voxel are assigned to that voxel and statistically combined into one output point.

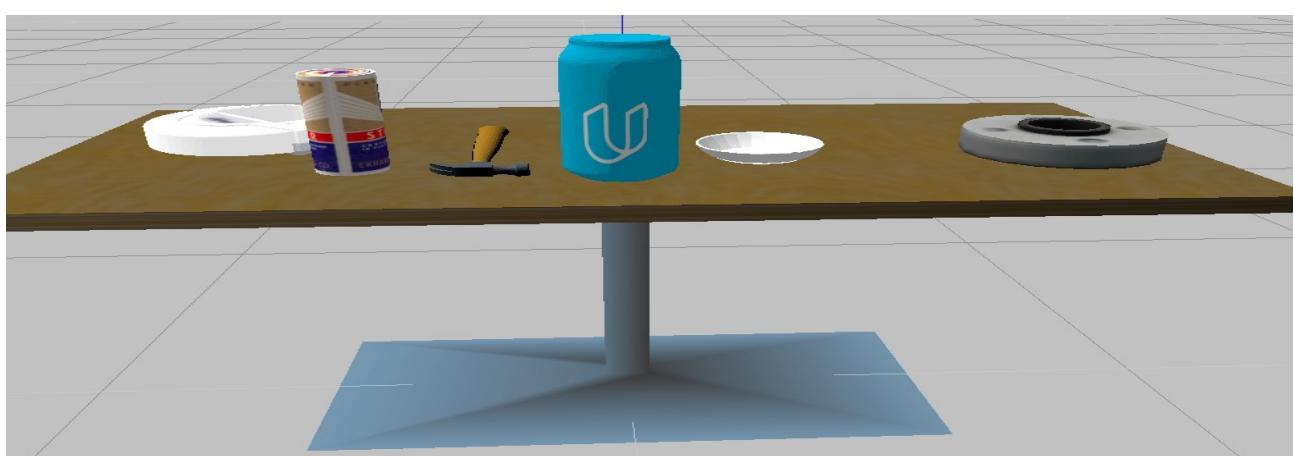
```
# Create a VoxelGrid filter object for our input point cloud
vox = cloud.make_voxel_grid_filter()

# Choose a voxel (also known as leaf) size
# Note: this (1) is a poor choice of leaf size
# Experiment and find the appropriate size!
LEAF_SIZE = 0.01

# Set the voxel (or leaf) size
vox.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)

# Call the filter function to obtain the resultant downsampled point cloud
cloud_filtered = vox.filter()
```

## Pass Through Filtering

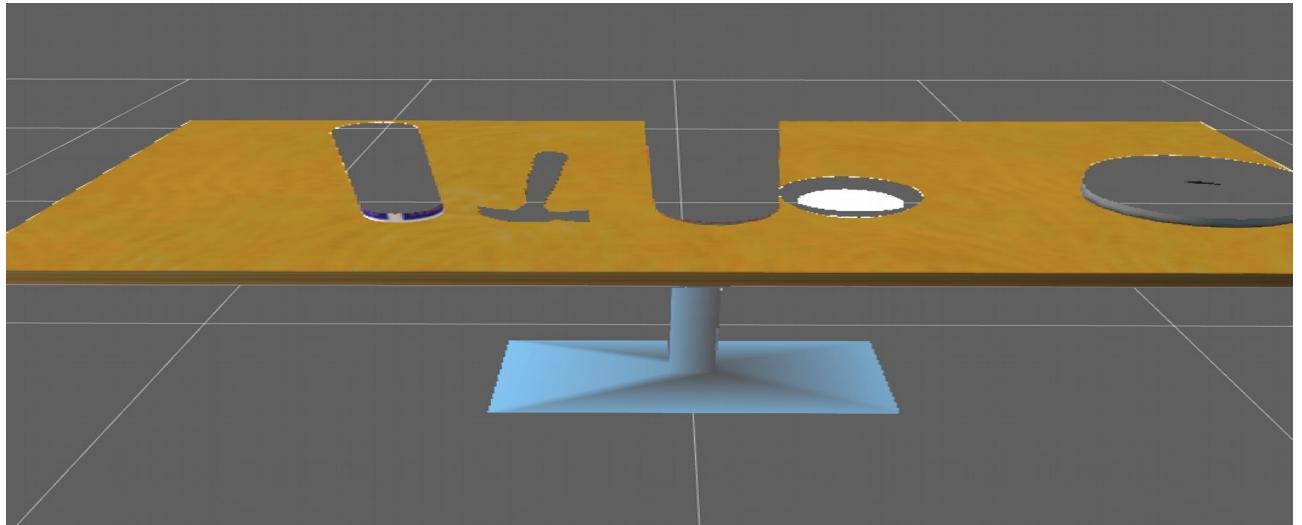


If you have some prior information about the location of your target in the scene, you can apply a Pass Through Filter to remove useless data from your point cloud.

The Pass Through Filter works **much like a cropping tool**, which allows you to crop any given 3D point cloud by specifying an axis with cut-off values along that axis. The region you allow to *pass through*, is often referred to as *region of interest*.

For instance, in our tabletop scene we know that the table is roughly in the center of our robot's field of view. Hence by using a Pass Through Filter we can select a region of interest to remove some of the excess data.

Applying a Pass Through filter along **z** axis (the height with respect to the ground) to our tabletop scene in the range 0.1 to 0.8 gives the following result:



### **Oops! We've retained the table but filtered out all of the objects!**

If we correctly apply the cut off ranges on the Pass Through Filter to the point cloud of the tabletop scene we can retain only the tabletop and the objects sitting on the table.



```
# PassThrough filter
```

```

# Create a PassThrough filter object.
passthrough = cloud_filtered.make_passthrough_filter()

# Assign axis and range to the passthrough filter object.
filter_axis = 'z'
passthrough.set_filter_field_name(filter_axis)
axis_min = 0.6
axis_max = 1.1
passthrough.set_filter_limits(axis_min, axis_max)

# Finally use the filter function to obtain the resultant point cloud.
cloud_filtered = passthrough.filter()

```

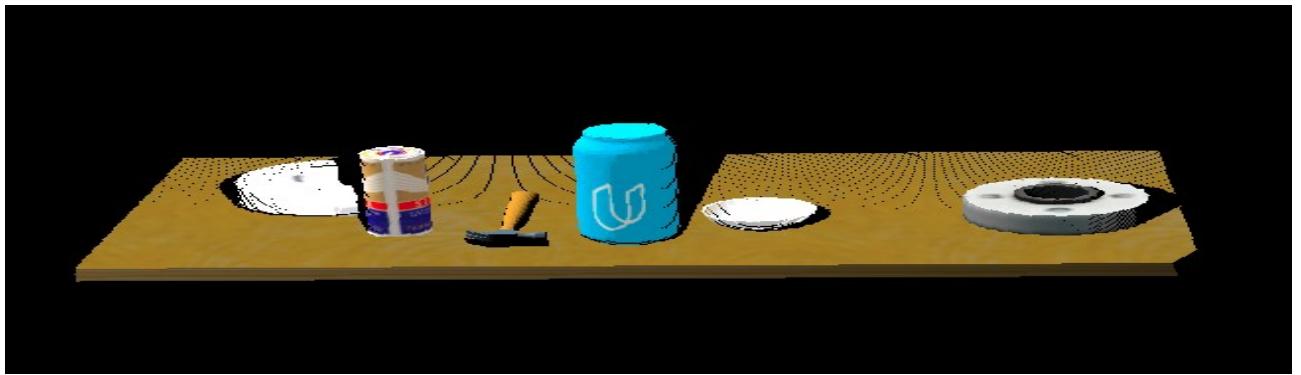
### 7.3.2 Segmentation in Perception - RANSAC

So now we applied filters to our point cloud to remove extraneous data and noise. As we discussed earlier point clouds are often very large and contain useless information. Filtering takes care of some of that, but not all. For example, in the tabletop scenario, there are still objects laying on the table along with our target object and **all these objects are part of the same point cloud**. To make further progress we need to **divide this point cloud** into smaller subsets **based on some common property**. This process is known as segmentation.

Segmentation allows us to divide the data into meaningful pieces, some of which can be discarded based on certain properties like shape, color, size or neighborhood.

In our tabletop example, we can use segmentation to separate out the table itself from the scene, followed by the objects which attributes much different from our target, leaving behind the best target object candidates.

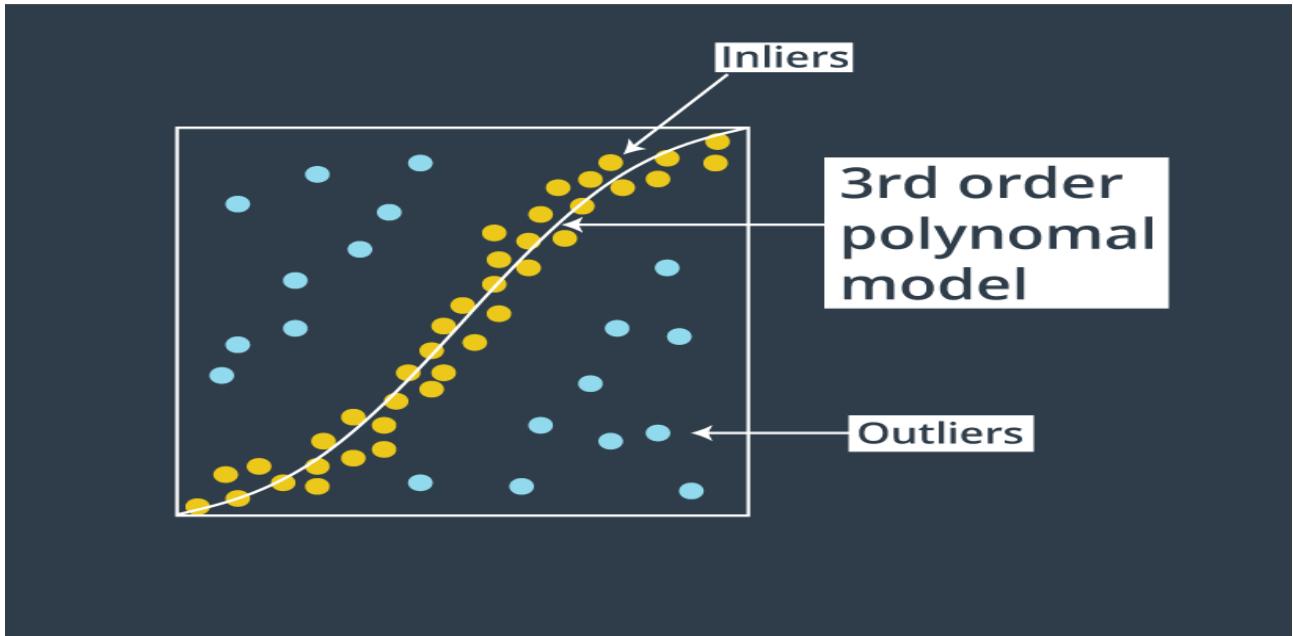
#### RANSAC Overview



In our tabletop scenario, combining some prior knowledge about the scene with a few point cloud filters, you were able to reduce the point cloud down to just the table and objects on top of it.

Next in your perception pipeline, we need to remove the **table itself** from the scene. To do this you will use a popular technique known as [Random Sample Consensus](#) or "RANSAC". RANSAC is an algorithm, that you can use to identify points in your dataset that belong to a particular model. In the case of the 3D scene you're working with here, the model you choose could be a plane, a cylinder, a box, or any other common shape.

The RANSAC algorithm assumes that all of the data in a dataset is composed of both inliers and outliers, where inliers can be defined by a particular model with a specific set of parameters, while outliers do not fit that model and hence can be discarded. Like in the example below, we can extract the outliers that are not good fits for the model.



If you have a prior knowledge of a certain shape being present in a given data set, you can use RANSAC to estimate what pieces of the point cloud set belong to that shape by assuming a particular model.

By modeling the **table as a plane**, you can remove it from the point cloud to obtain the following result:



On the other hand, a disadvantage of RANSAC is that there is no upper limit on the time it can take to compute the model parameters. This is somewhat alleviated by choosing a fixed number of iterations but that has its own demerits.

If you choose a lower number of iterations, the solution obtained may not be optimal. In this way RANSAC offers a trade-off between compute time versus model detection accuracy.

Since the top of the table in the scene is the single most prominent plane, **after ground removal**, you can effectively use RANSAC to identify points that belong to the table and discard/filter out those points using a relatively low number of iterations.

Another popular use case of such plane segmentation appears in mobile robot autonomous navigation. For collision avoidance with objects and to determine **traversable terrain**, ground plane segmentation is an important part of a mobile robot's perception toolkit.

## RANSAC Plane Fitting



The RANSAC algorithm mainly involves performing two iteratively repeated steps on a given data set: Hypothesis and Verification. First, a hypothetical shape of the desired model is generated by randomly selecting a minimal subset of n-points and estimating the corresponding shape-model parameters.

A minimal subset contains the smallest number of points required to uniquely estimate a model. For example, 2 points are needed to determine a line, while 3 non-collinear points are needed to determine a plane as described below.

Let's let  $p_1$ ,  $p_2$ , and  $p_3$  be the three randomly selected non-collinear points such that:

$$p_1 = (x_1, y_1, z_1)$$

$$p_2 = (x_2, y_2, z_2)$$

$$p_3 = (x_3, y_3, z_3)$$

A plane can then be described by an equation of the form:

$$ax + by + cz + d = 0.$$

The coefficients  $a, b, c, d$  can then be obtained by solving the following system of equations:

$$ax_1 + by_1 + cz_1 + d = 0$$

$$ax_2 + by_2 + cz_2 + d = 0$$

$$ax_3 + by_3 + cz_3 + d = 0$$

This system can be solved using [Cramer's rule](#) and some basic matrix manipulations in the following way:

$$\mathbf{a} = \frac{-d}{D} \begin{bmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \end{bmatrix}$$

If  $D$  is non-zero (so for planes not through the origin) the values for  $a$ ,  $b$ , and  $c$  can be calculated as follows:

$$\mathbf{b} = \frac{-d}{D} \begin{bmatrix} x_1 & 1 & z_1 \\ x_2 & 1 & z_2 \\ x_3 & 1 & z_3 \end{bmatrix}$$

$$\mathbf{c} = \frac{-d}{D} \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix}$$

Once a model is established, the remaining points in the point cloud are tested against the resulting candidate shape to determine how many of the points are well approximated by the model.

After a certain number of iterations, the shape that possesses the largest percentage of inliers is extracted and the algorithm continues to process the remaining data.

```
# Create the segmentation object
seg = cloud_filtered.make_segmenter()

# Set the model you wish to fit
seg.set_model_type(pcl.SACMODEL_PLANE)
seg.set_method_type(pcl.SAC_RANSAC)

# Max distance for a point to be considered fitting the model
# Experiment with different values for max_distance
# for segmenting the table
max_distance = 0.01
seg.set_distance_threshold(max_distance)

# Call the segment function to obtain set of inlier indices and model
coefficients
inliers, coefficients = seg.segment()
```

Hopefully PCL library support the RANSAC algorithm see appendix!

### 7.3.3 Extracting Indices

With RANSAC we identified which indices in our point cloud correspond to the table. If we applied the Pass Through Filter correctly, then the indices not corresponding to the table are those representing the objects on the table!

As the name suggests, the **ExtractIndices Filter** allows you to extract points from a point cloud by **providing a list of indices**. With the RANSAC fitting you just performed, the **output inliers corresponds to the point cloud indices that were within max\_distance of the best fit model**.

While this filter **does not perform any advanced filtering action**, it is **frequently used** along with **other techniques to obtain a subset of points from an input point cloud**. **Most object recognition algorithms return a set of indices associated with the points that form the identified target object**.

As a result, it's convenient to use the ExtractIndices Filter to extract the point cloud associated with the identified object.

```
# Extract inliers
extracted_inliers = cloud_filtered.extract(inliers, negative=False)
filename = 'extracted_inliers.pcd'
pcl.save(extracted_inliers, filename)
```

## Object Extraction

While it's exciting to have successfully fit the plane of the table using RANSAC and extracted the subset of points corresponding to the table itself using the ExtractIndices Filter, we actually just want to get rid of the table and focus on the objects on top of the table.

It's easy to use what you've done already to extract all the objects of interest from the point cloud by simply changing the `negative` flag on the `extract` method to `True`.

```
extracted_outliers = cloud_filtered.extract(inliers, negative=True)
filename = 'extracted_outliers.pcd'
pcl.save(extracted_outliers, filename)
```

### 7.3.4 Outlier Removal Filter

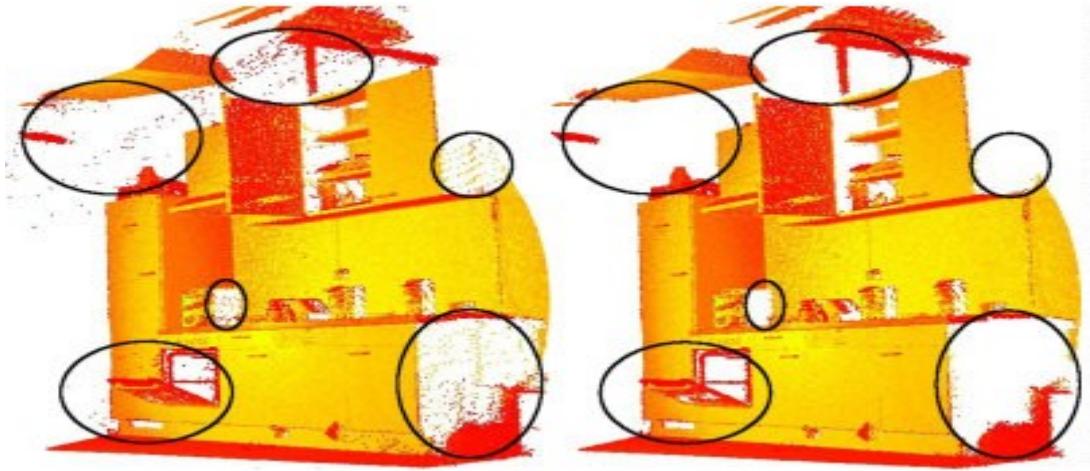
While calibration takes care of distortion , noise due to external factors like dust in the environment, humidity in the air, or presence of various light sources lead to sparse outliers which corrupt the results even more.

Such outliers lead to complications in the estimation of point cloud characteristics like curvature, gradients, etc. leading to erroneous values, which in turn might cause failures at various stages in our perception pipeline.

One of the filtering techniques used to remove such outliers is to perform a statistical analysis in the neighborhood of each point, and remove those points which do not meet a certain criteria. PCL's StatisticalOutlierRemoval filter is an example of one such filtering technique. For each point in the point cloud, it computes the distance to all of its neighbors, and then calculates a mean distance.

By assuming a Gaussian distribution, all points whose mean distances are outside of an interval defined by the global distances mean+standard deviation are considered to be outliers and removed from the point cloud.

The following graphic shows the result of applying the StatisticalOutlierRemoval Filter to noisy point cloud data:



```
# Much like the previous filters, we start by creating a filter object:  
outlier_filter = cloud_filtered.make_statistical_outlier_filter()  
  
# Set the number of neighboring points to analyze for any given point  
outlier_filter.set_mean_k(50)  
  
# Set threshold scale factor  
x = 1.0  
  
# Any point with a mean distance larger than global (mean distance+x*std_dev)  
# will be considered outlier  
outlier_filter.set_std_dev_mul_thresh(x)  
  
# Finally call the filter function for magic  
cloud_filtered = outlier_filter.filter()
```

### 7.3.5 Summary

We've covered a lot of ground so far in this introduction to 3D-imaging, we've explored camera calibration (not really but you should), filtering and segmentation and we now have some powerful tools in our kit to tackle 3-D point cloud data.

Next, we'll be building on this tools to perform more complex segmentation tasks using clustering analysis, combined with what we learned previously we'll be prepared to do 3-D object segmentation in complex environments.

# 7.4 Object Segmentation

## 7.4.1 Introduction

So now we have filtered our point cloud data and wrote a RANSAC plane filtering algorithm to remove the table from the scene. However, thus far, we have been relying solely on object shapes to perform segmentation.

Even though we've been dealing with point clouds, our dataset still contains feature rich color information as well that can be combined with our shape information to perform more complex segmentation tasks.

In this chapter we'll be introducing a technique called **Clustering** that will allow us to segment objects in our point cloud without having to assume a model shape. In essence, this technique will allow us to find points in our dataset that are clustered based on particular features.

These clusters could be based on color, position, texture or a combination of many features. Performing segmentation using clustering, gives you the freedom to discover an arbitrary number of objects of any shape in your data.

## 7.4.2 Downside of Model Fitting

So far we've filtered our point data and we wrote a RANSAC plane-fitting algorithm to remove the table from the scene. Next we'll learn how to further segment the remaining data into meaningful pieces. From a technical perspective, RANSAC can be used to fit a variety of shaped models like cylinders, cubes or spheres.

But in the scenario we're working with we have a cluttered environment with little to no prior information about how many objects there may be in the scene. It's very likely that many of the tabletop objects may fit the same shape model as your target object. For example, if your target object was a can of soda, other cylindrical objects like a cup, a bottle or a pen holder may induce false positives if you were to rely solely on the RANSAC cylinder-fitting algorithm.

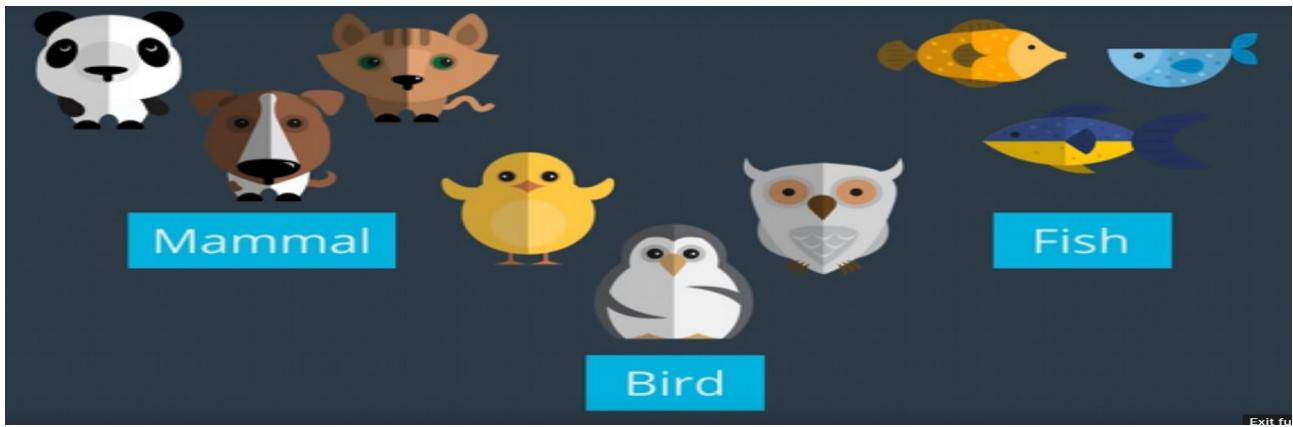
Not only that, but our scene has objects of various shapes, if we were to model fit many shapes for removal, we'd be processing the entire point cloud of data many times, once for each model, which is not at all optimal.

Instead we will now focus on other properties of our scene. Unlike a Lidar or a Time-of-Flight camera an RGB-D camera provides us with data which is rich in features like colors and texture. In this chapter we'll use these features to recognize the object we're looking for.

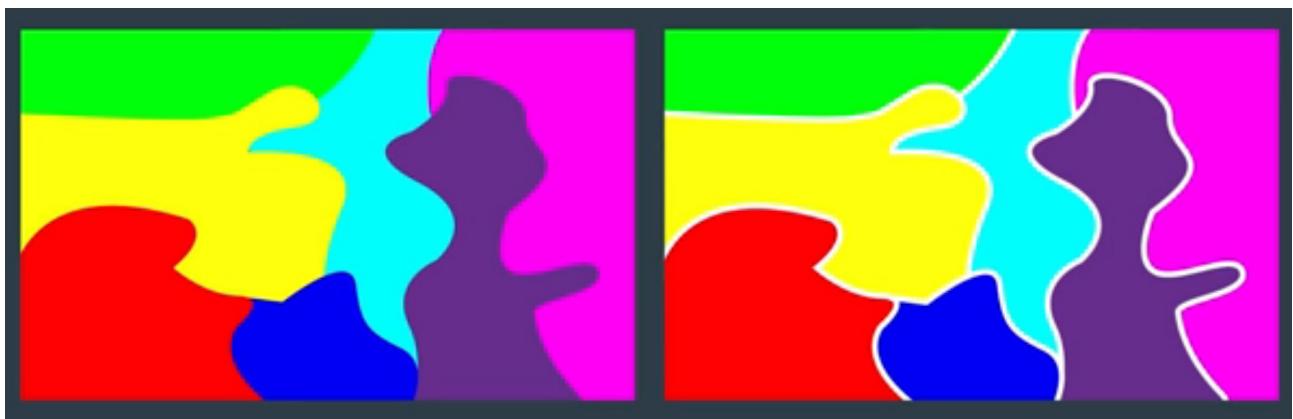
## 7.4.3 Clustering

While segmentation is the process of dividing our point cloud data into meaningful subsets, using some common property, clustering is the process of finding similarities among individual points, so that they may be segmented.

In essence, clustering tries to answer the simple question, which components of a dataset naturally belong together?



For instance, if I were to show you the below image and ask you to identify individual objects from the scene, we'll be able to quickly come up with something like the image on the right.

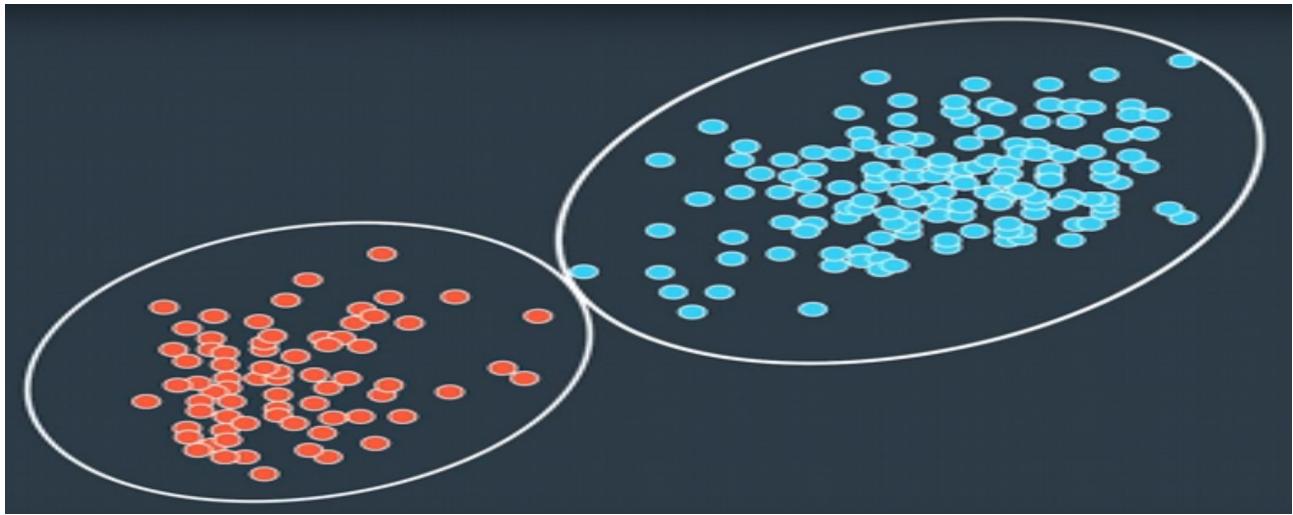


Our brain in this case, actually performed image segmentation via clustering, using the property of color. Even though the shapes of the objects in this picture are not regular or symmetric, our brain was able to cluster pixels together, based on the color property and hence, performing segmentation.

Now let's try something different, how about segmenting this data.



If I were to ask you to draw lines around different sets in this data, it is very likely that your answer will be close to this.

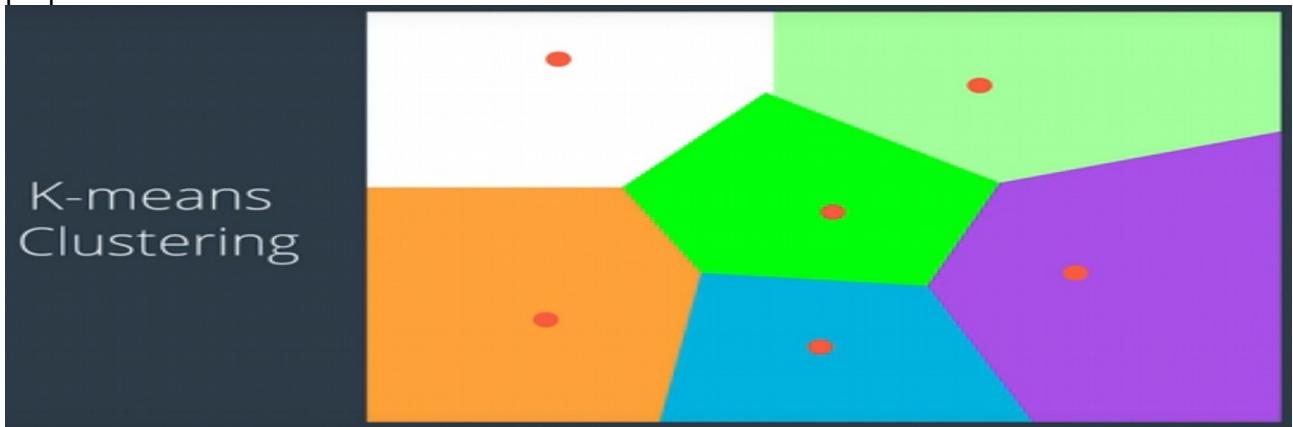


This time, your brain performed clustering based on spatial neighborhood, meaning points that are closer in the 2D space formed a cluster.

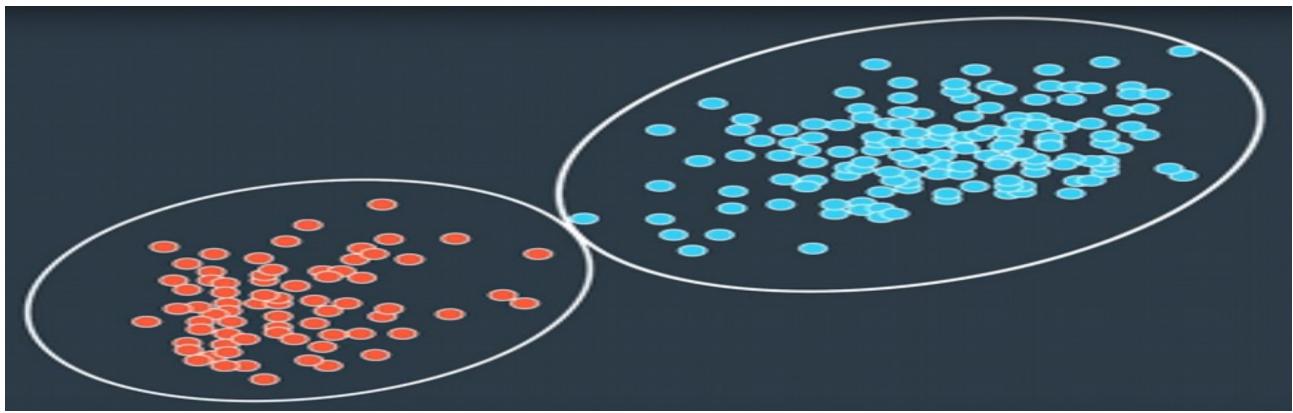
We intend to leverage such properties of our data, in order to achieve the best segmentation results via clustering, to simplify our point cloud.

Upcoming we will tackle the clustering problem using two different approaches.

The first technique is called K-means Clustering which is an iterative, unsupervised learning algorithm that divides the input data into a key number of clusters, based on one or more features or properties.



The second approach is Euclidean clustering, a simple data clustering approach in a Euclidean sense, where points that are closer to each other, are clustered together by making use of a 3D grid subdivision of the space, much like before.



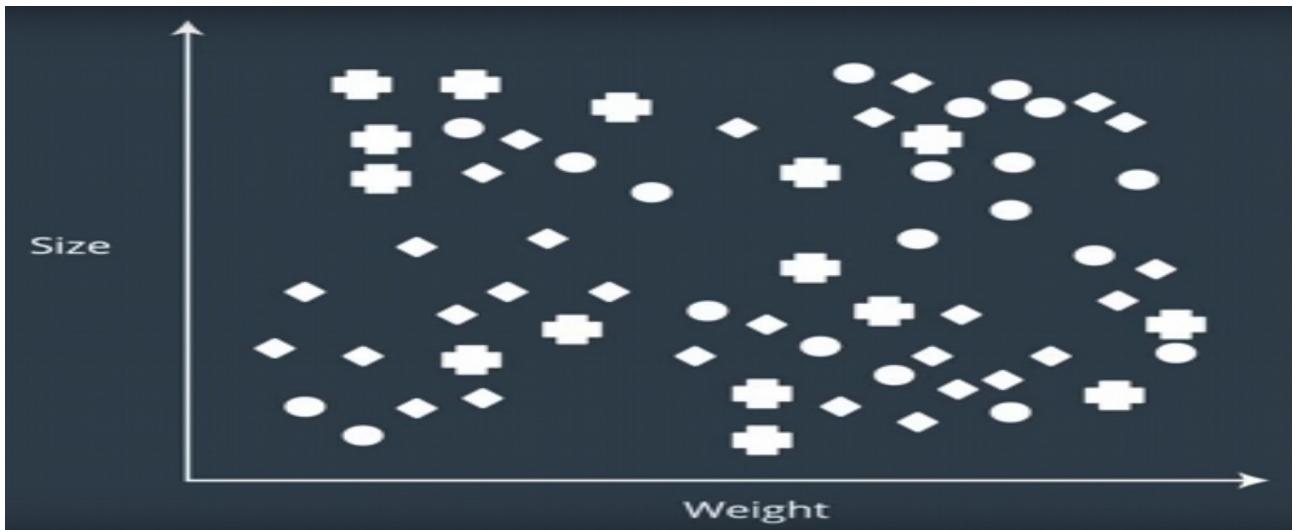
#### 7.4.4 K-means Clustering

K-means clustering is one of the most simple and widely used clustering techniques in computer vision, machine learning and data mining. To better understand this algorithm let's jump directly into a visual example.

Consider the following problem, let's say a company like Amazon wants to standardize the sizes of their shipping boxes. Since their shipping boxes must fit a variety of items, Amazon first needs to decide the size of these new boxes based on the items they will ship.



To do this some Amazon employee pulls out a master catalog, containing the size and weight of each and every product they sell. If we plot these set of data with Size and Weight as x and y axes respectively, we'll get something like this.



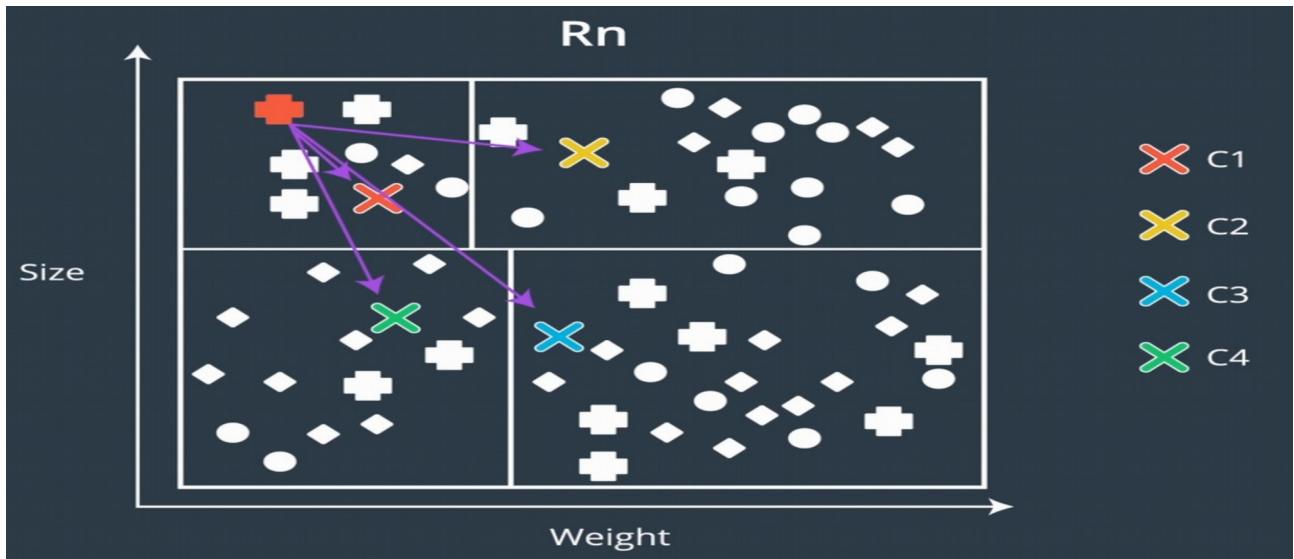
Now, it would be impractical to have a different size shipping box for each item since it meets the purpose of cost saving by mass production. So our Amazon employee decides to create four different sized boxes that will fit all the items.



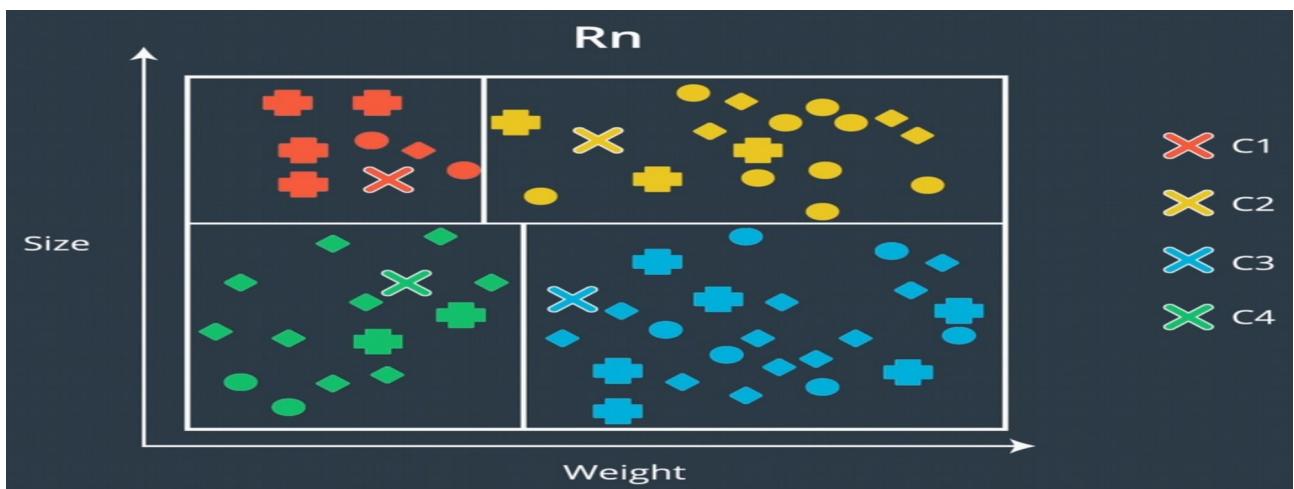
Now, in order to estimate the size of these boxes and to decide what group of objects will be packed in each of these four boxes, our Amazon employee will use the K-means Clustering algorithms to create clusters of objects and their corresponding shipping box size.



Let's do now a quick walkthrough of the algorithm itself. Consider  $R_n$  to be our set of items from the Amazon catalog, we begin by randomly selecting four initial means or centroids,  $C_1, C_2, C_3$  and  $C_4$ . Next, we create four clusters by associating each data point with the nearest centroid. To do this, we calculate the distance from each point to each centroid and then label the point with the centroid color closest to it.



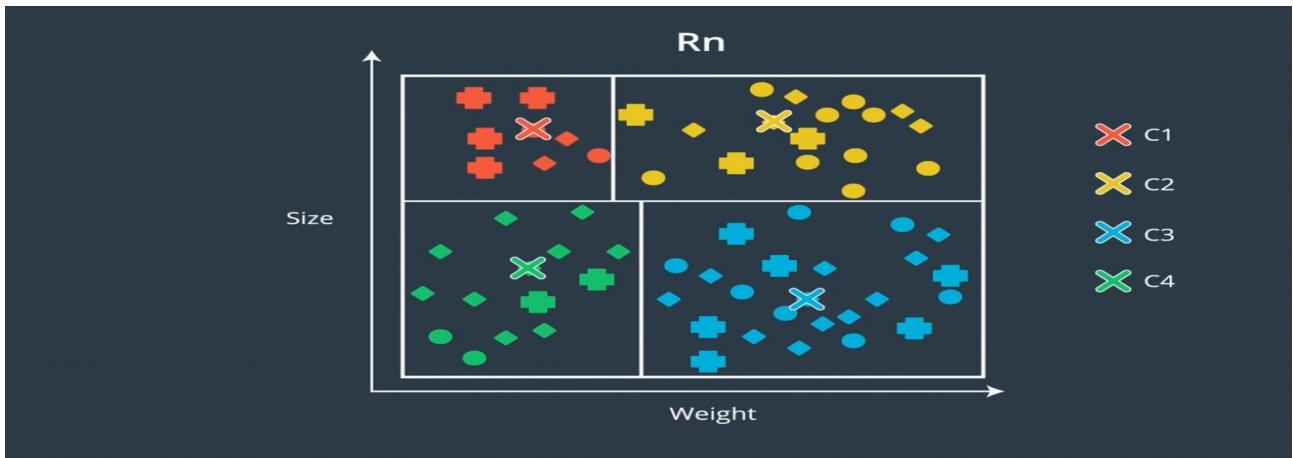
The resulting partition represents a Voronoi diagram generated by the centroids. For visualization purposes, let us assign a different color to each cluster.



New centroid is calculated for each cluster by taking the mean or average of all data points with the same label meaning the centroid C1, C2, C3 and C4 shift to the newly calculated locations. **Finally, steps two and three are repeated until one of the following criteria are met.**

- 1. The number of iterations reaches a specified limit
- 2. Convergence is achieved by reaching a specified tolerance between two consecutive centroid locations.

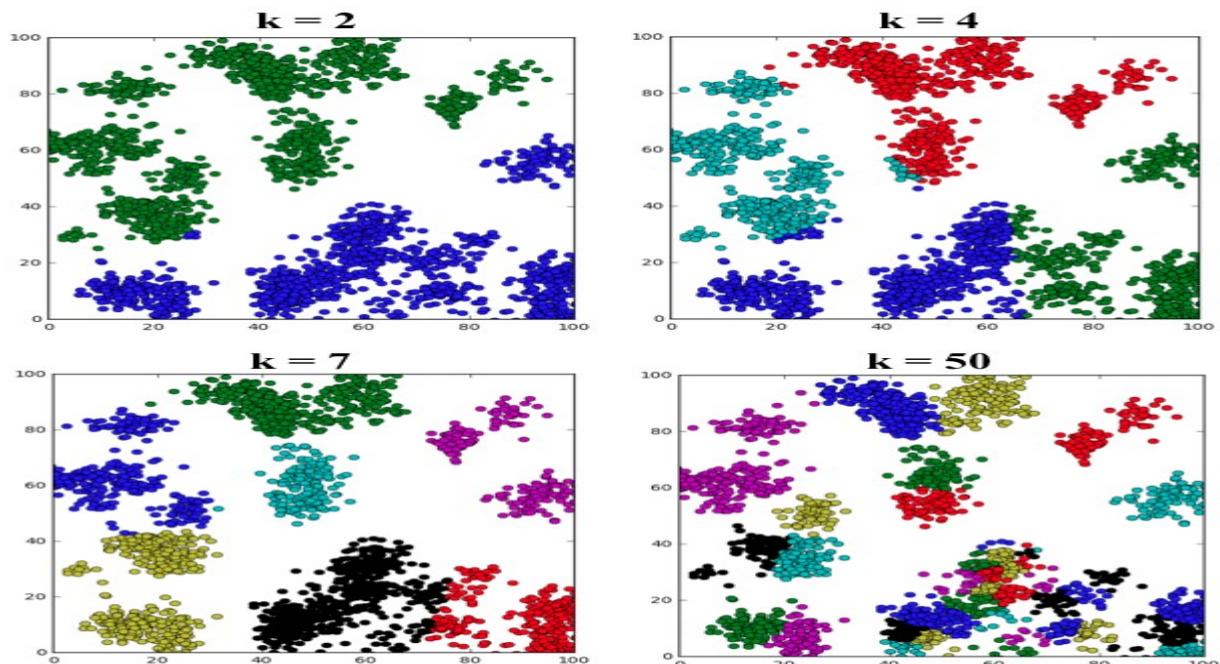
**The final result will look something like this.**



The process of clustering has resulted into four different item groups from the catalog clustered together based on their weight and size. The size of the four shipping boxes can now be determined by referring to a single object in each cluster with the highest weight and highest size attributes.

**After you have gotten idea of how k-means works for clustering data. Take some time to visit [here](#) and explore it in action!**

To check how to code the K-means check on the appendix!



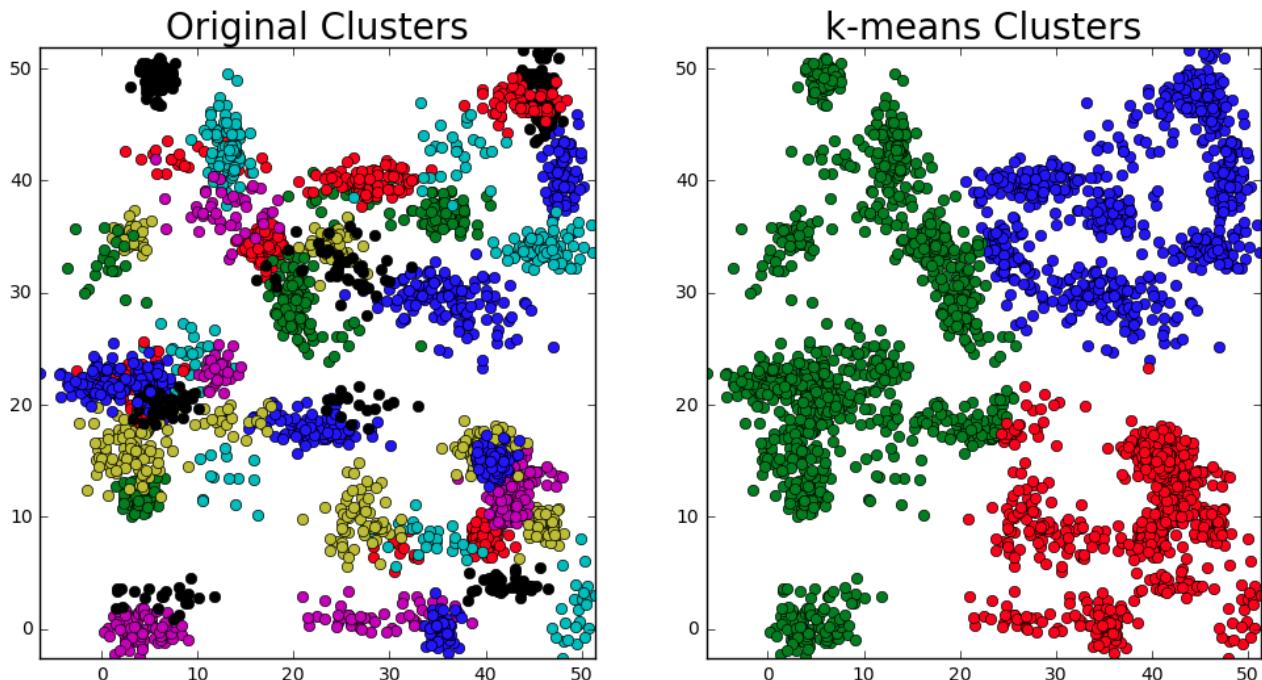
## The k-means Algorithm

The k-means clustering algorithm works like this:

Suppose you have a set of n data points:  $p_1, p_2, \dots, p_n$  and you intend to divide the data into k clusters.

- Start by selecting k individual points  $c_1, c_2, \dots, c_k$  from the dataset as the initial cluster centroids.
- Define convergence / termination criteria (stability of solution and max number of iterations)

- while convergence / termination criteria are not met do:
  - for i=1 to n:
    - Calculate distance from  $p_i$  to each cluster centroid
    - Assign  $p_i$  to its closest centroid and label it accordingly
  - endfor
  - For j=1 to k:
    - Recompute the centroid of cluster j based on the average of all data point that belong to the cluster
  - endfor
- endwhile



**Regarding the image above, the following are true?**

- **k = 3, but the number of original clusters is much higher.**
- **With k = 3, such as in the image above, k-means will always divide the data up into exactly 3 clusters.**

**Under which of the conditions might the k-means algorithm fail?**

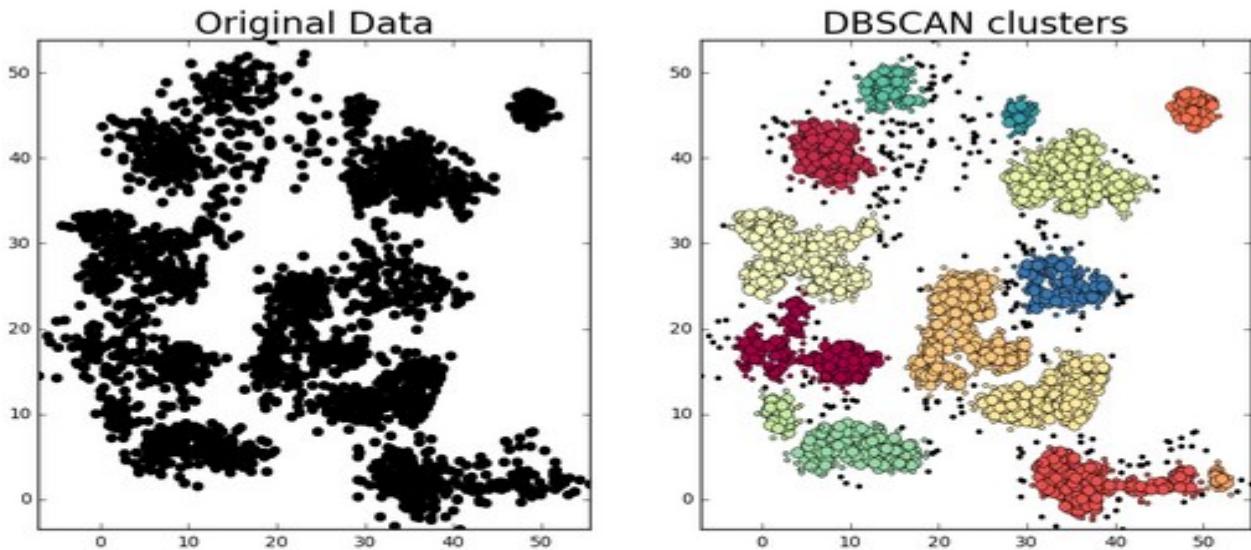
- **Clusters are highly overlapping**
- **Clusters are complex shapes**
- **Cluster assignment is highly sensitive to random cluster center initialization.**

**If you have no idea how many clusters to expect in your data, what should you do?**

- **Try a different clustering algorithm other than k-means.**

#### 7.4.5 DBSCAN Algorithm and comparing it to K-means

## DBSCAN Algorithm



Original data on the left and clusters identified by the DBSCAN algorithm on the right. For DBSCAN clusters, large colored points represent core cluster members, small colored points represent cluster edge members, and small black points represent outliers.

## The DBSCAN Algorithm

DBSCAN stands for Density-Based Spatial Clustering of Applications with Noise. This algorithm is a nice alternative to k-means **when you don't know how many clusters to expect in your data, but you do know something about how the points should be clustered in terms of density** (distance between points in a cluster).

The DBSCAN algorithm creates clusters by grouping data points that are within some threshold distance  $d_{th}$  from the nearest other point in the data.

The algorithm is sometimes also called “Euclidean Clustering”, because the decision of whether to place a point in a particular cluster is based upon the “Euclidean distance” between that point and other cluster members.

You can think of Euclidean distance the length of a line connecting two points, but the coordinates defining the positions of points in your data **need not be spatial coordinates**. They could be defined in color space, for example, or in any other feature space of the data.

The Euclidean distance between points  $p$  and  $q$  in an  $n$ -dimensional dataset, where the position of  $p$  is defined by coordinates  $(p_1, p_2, \dots, p_n)$  and the position of  $q$  is defined by  $(q_1, q_2, \dots, q_n)$  then the distance between the two points is just:

$$D = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \dots + (p_n - q_n)^2}$$

## DBSCAN Clustering Steps:

Suppose you have a set P of n data points p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>n</sub>:

- Set constraints for the minimum number of points that comprise a cluster (`min_samples`)
- Set distance threshold or maximum distance between cluster points (`max_dist`)
- For every point p<sub>i</sub> in P, do:
  - if p<sub>i</sub> has at least one neighbor within `max_dist`:
    - if p<sub>i</sub>'s neighbor is part of a cluster:
      - add p<sub>i</sub> to that cluster
    - if p<sub>i</sub> has at least `min_samples`-1 neighbors within `max_dist`:
      - p<sub>i</sub> becomes a "core member" of the cluster
    - else:
      - p<sub>i</sub> becomes an "edge member" of the cluster
  - else:
    - p<sub>i</sub> is defined as an outlier

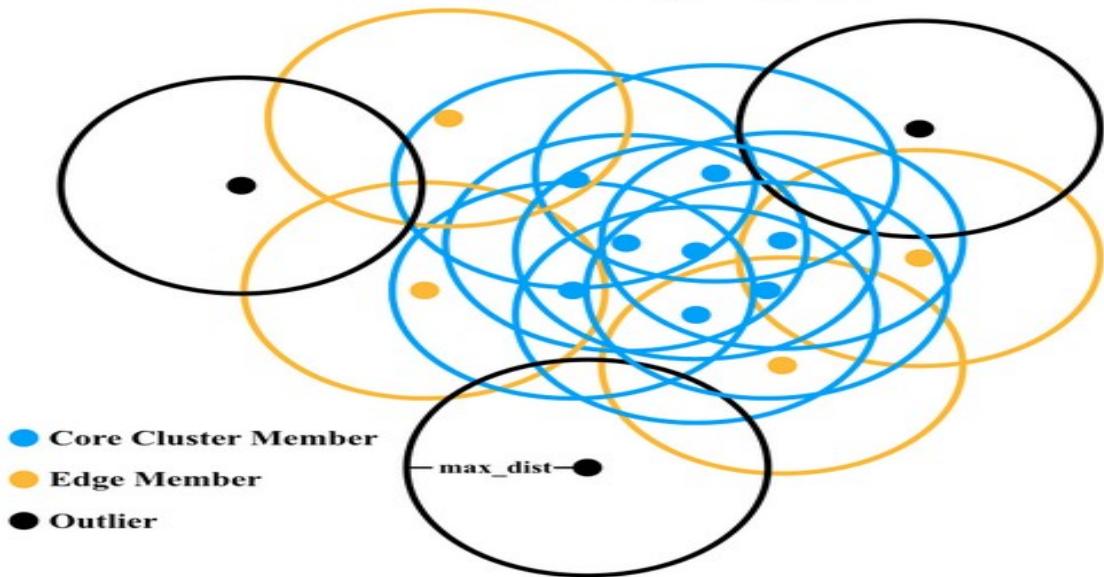
The above code is a bit simplified, to aid in a beginner's understanding of the algorithm. If you visited every data point at random, you would end up creating significantly more clusters than needed and would have to merge them somehow. We can avoid this by controlling the order in which we visit data points.

Upon creating a new cluster with our first qualifying data point, we will add all of its neighbors to the cluster. We will then add its neighbor's neighbors, and their neighbors, until we have visited every data point that belongs to this cluster. Only after doing so, do we move on to choosing another data point at random. This way, we can guarantee that a cluster is complete, and there are no more points in the data set that belong to the cluster.

The full pseudo-code implementation of the DBSCAN algorithm can be found on the [DBSCAN Wikipedia page](#).

Here's how the outcome looks visually for an example where `min_samples` = 4:

### Example: `min_samples = 4`



Blue points are core cluster members having at least three neighbors within `max_dist` (satisfying number of neighbors plus self `>= min_samples`). Yellow points are edge members having neighbors, but fewer than three. Black points are outliers having no neighbors.

## Comparing DBSCAN and k-means Clustering

### ■ DBSCAN

- There are an unknown number of clusters present in your data but you know something about the characteristics you expect clusters to have (i.e. their density in parameter space).
- The data contain outliers that you want to exclude from clusters.
- No need to define a termination / convergence criteria for this algorithm.

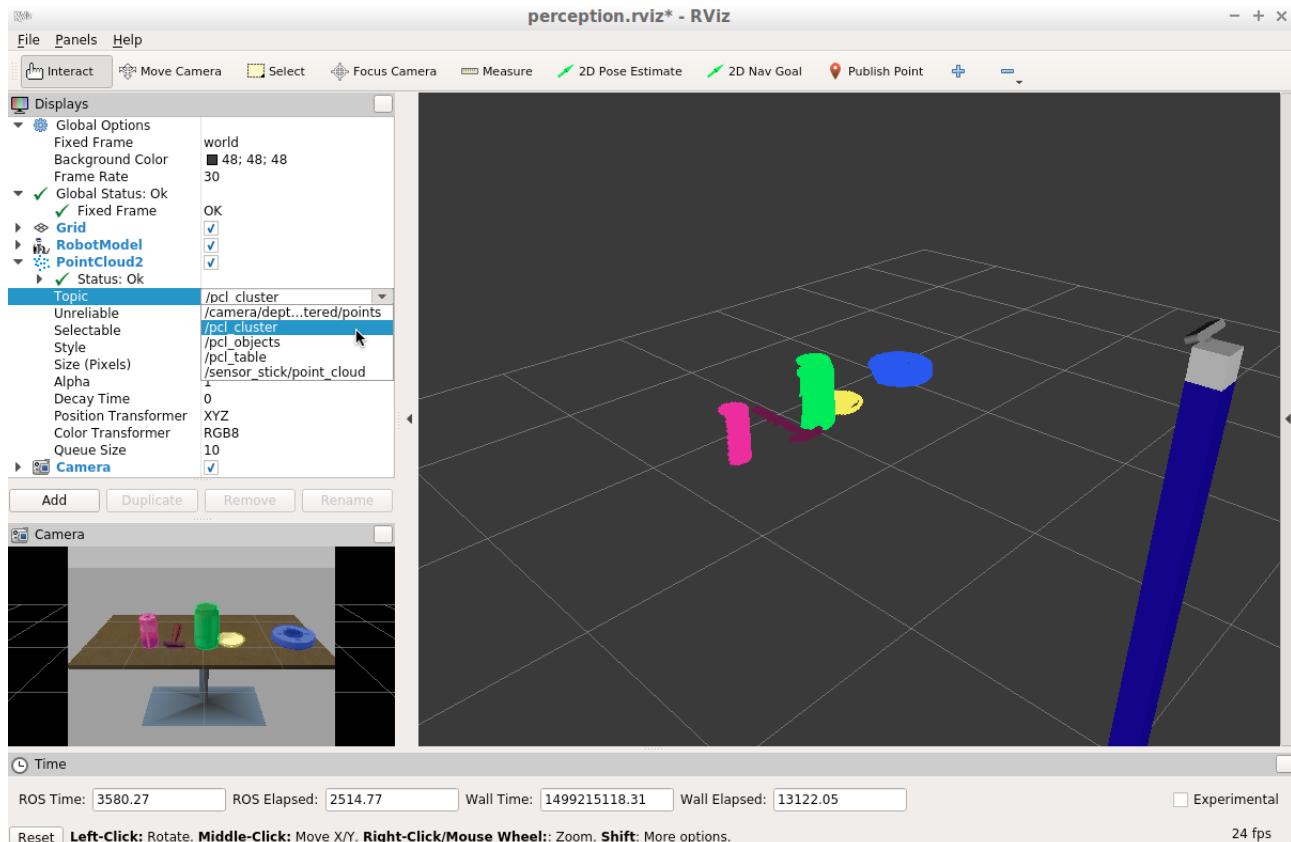
### ■ K-Means

- You would like to cluster your data into a fixed number of clusters
- Performance may vary based on number of iterations

## 7.4.6 Summary

With the techniques shown so far we have everything we need to perform segmentation tasks. So what happens next? Well at this point we're all setup to take our raw RGB-D camera data, convert it to a point cloud, filter out the junk and noise and then segment the data into individual objects.

Next, our task will be to perform object recognition analysis to identify which of the segmented objects is the one we're looking for.



**See the appendix for the experimental results!!**

# 7.5 Object Recognition

## 7.5.1 Intro to Object Recognition

Object recognition is a central theme in computer vision and perception for robotics, when you and I view a scene with our eyes, we are constantly performing the task of object recognition for the world we see before us.

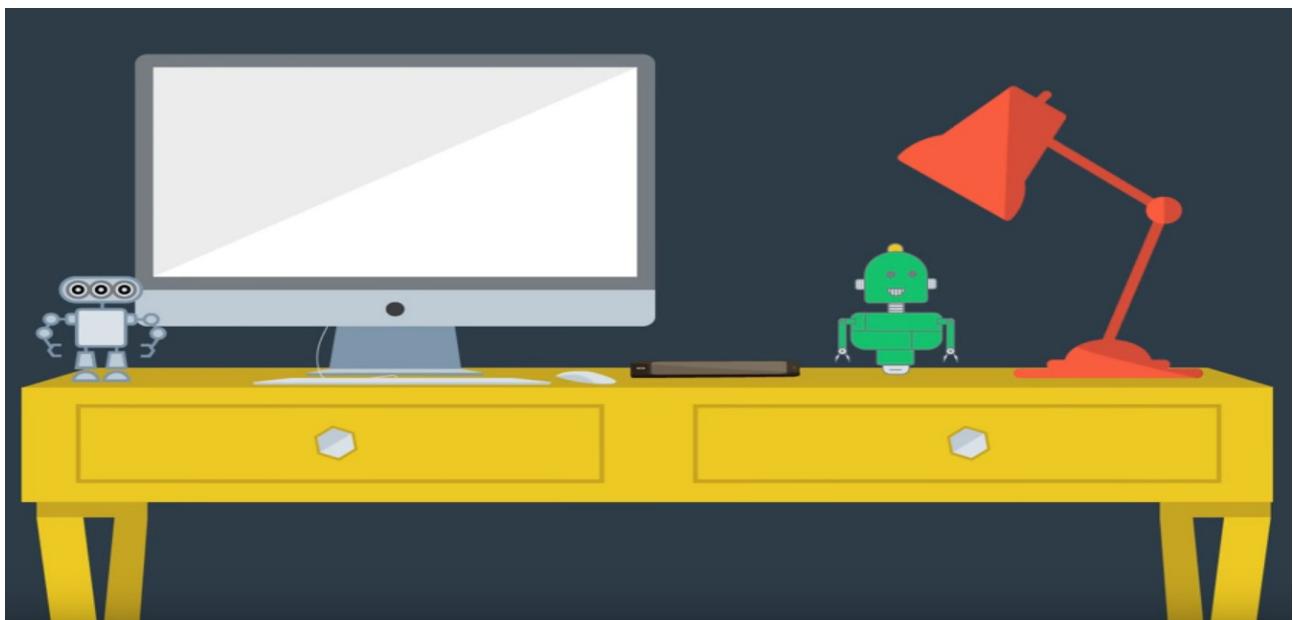
With sensors, a robot can perceive the world around it in terms of depth, color, and so on, But ultimately, we would like a robot to be able to recognize objects in its surroundings, just like we do.

In any given image or 3D point cloud, you might find a variety of objects of different shapes and sizes, in many robotics applications, there will be a particular object that we're looking for in the scene. This object of interest might be at any distance in any orientation and it might even be obscured by other objects in the scene.

So the question then becomes, how can we reliably locate what we're looking for in our field of view, regardless of its position or orientation?

For decades people have been working out better and better ways to look at objects in their data sets. But no matter the method it really boils down to identifying the features that best describe the object we're looking for.

It's just like if you were to tell me I'm looking for a small green robot.



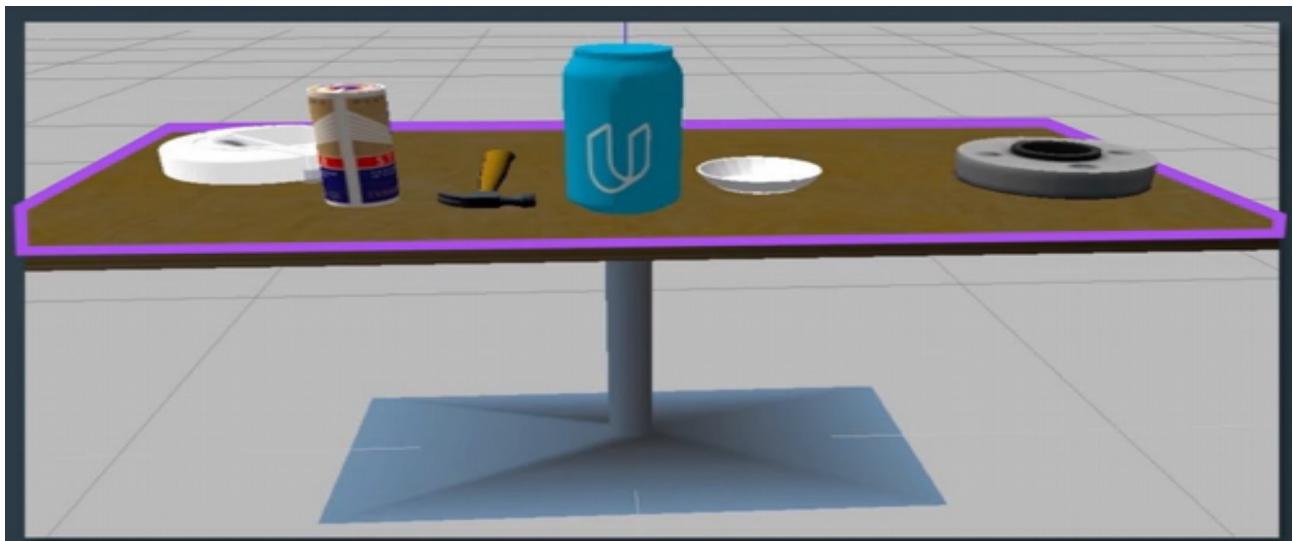
Once you're telling me what you are looking for, I can immediately identify where it is. And the same goes for computer vision algorithms, the better your description of the object you're looking for, the more likely the algorithm is to find it. Which is to say the better we can characterize the features that uniquely differentiate our target from other objects in the scene, the more robust our object recognition algorithm will be.

## 7.5.2 Features

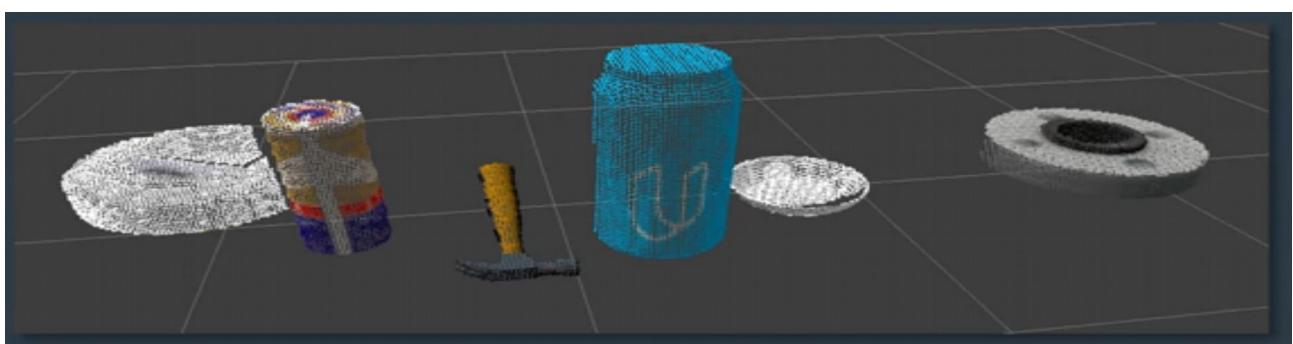
Here we will continue to work with point cloud data and focus on identifying a set of features that we can use to explicitly describe, not only what we're looking for in our data but also what we're not looking for.

With this feature set in hand, we can then train a classifier to recognize the object we're searching for in our point cloud. But first, let's back up a step and think what makes a good set of features.

With the previous filtering and segmentation exercises, we saw that having prior knowledge of things, like where we expect to find our object of interest, can help us zero in on the areas of the point cloud containing our object.



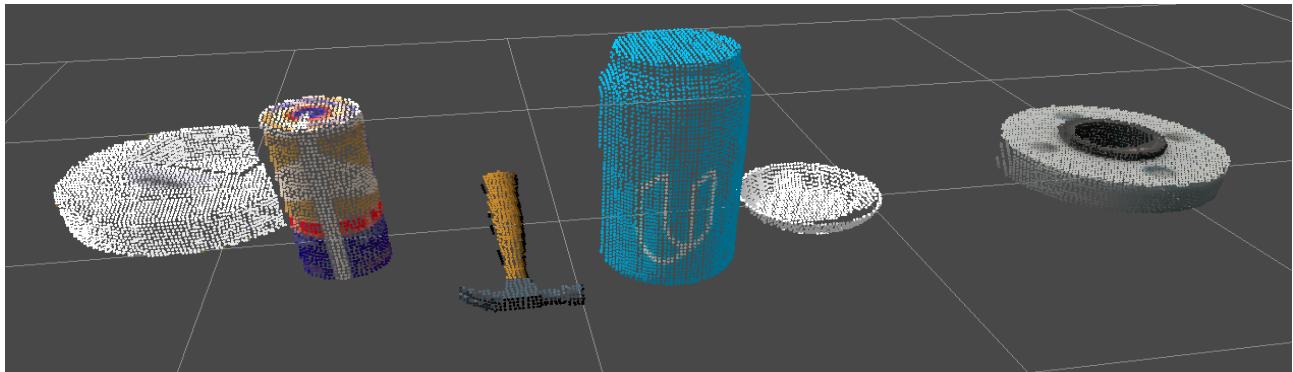
After segmentation, we have a point cloud that we've broken down into individual objects and for each point in the cloud, we have RGB color information as well as spatial information in three dimensions.



Therefore, it makes sense to explore feature sets that include some combination of this color and shape information in a way that differentiates our object of interest from other objects in the environment.

Upcoming, we will explore various aspects of how color and shape information can be best used to generate a set of features for object recognition.

## Feature Intuition



Given the segmented point cloud shown above, what features might be useful in identifying a particular object in the scene?

- Color
- Position within the scene
- Shape
- Size

Features describe the characteristics of an object. With RGB-D point clouds, you have three color channels and three spatial coordinates.

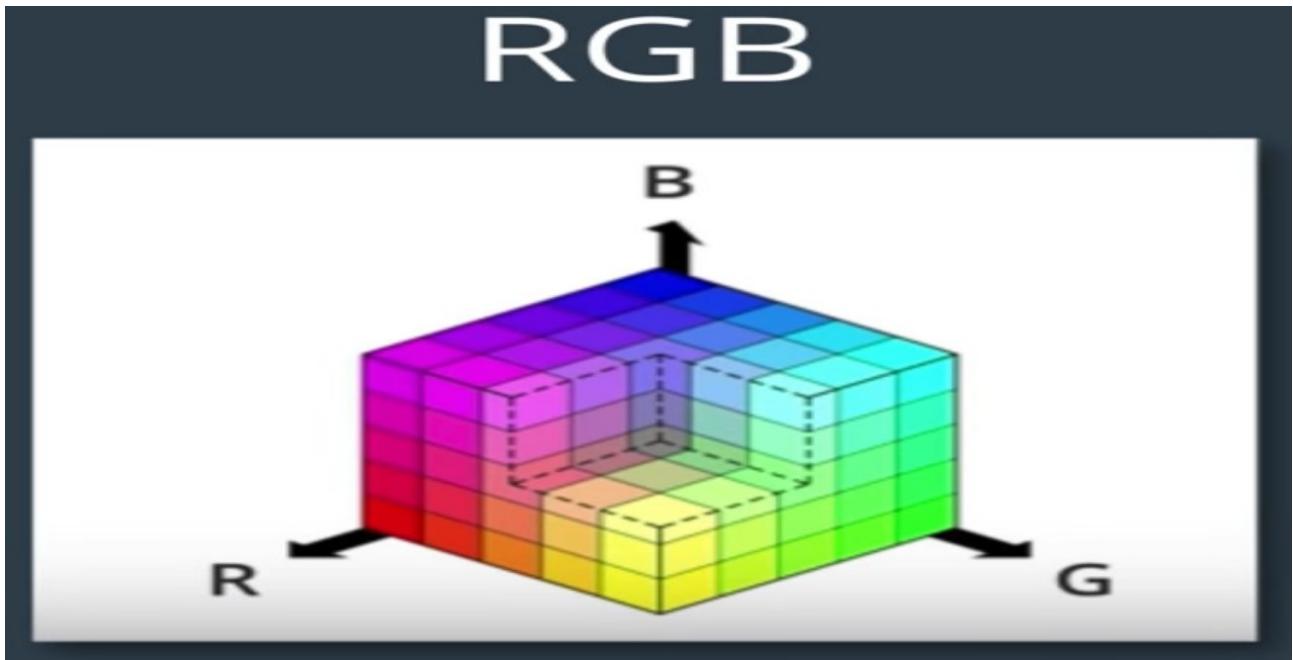
Match up the different features with the characteristic(s) that they capture about an object in a point cloud.

FEATURES	CHARACTERISTICS
The color distribution of points in 2D (e.g., after a projection along the viewing axis).	Object color and shape
The distribution in color space (e.g., a histogram of each color channel).	Detailed color information
3D spatial distribution of points.	Detailed shape information

### 7.5.3 Color Spaces

For the data we've been working with so far, we know that just like each pixel and image, each point in our point cloud has an associated set of red, green and blue or RGB color values.

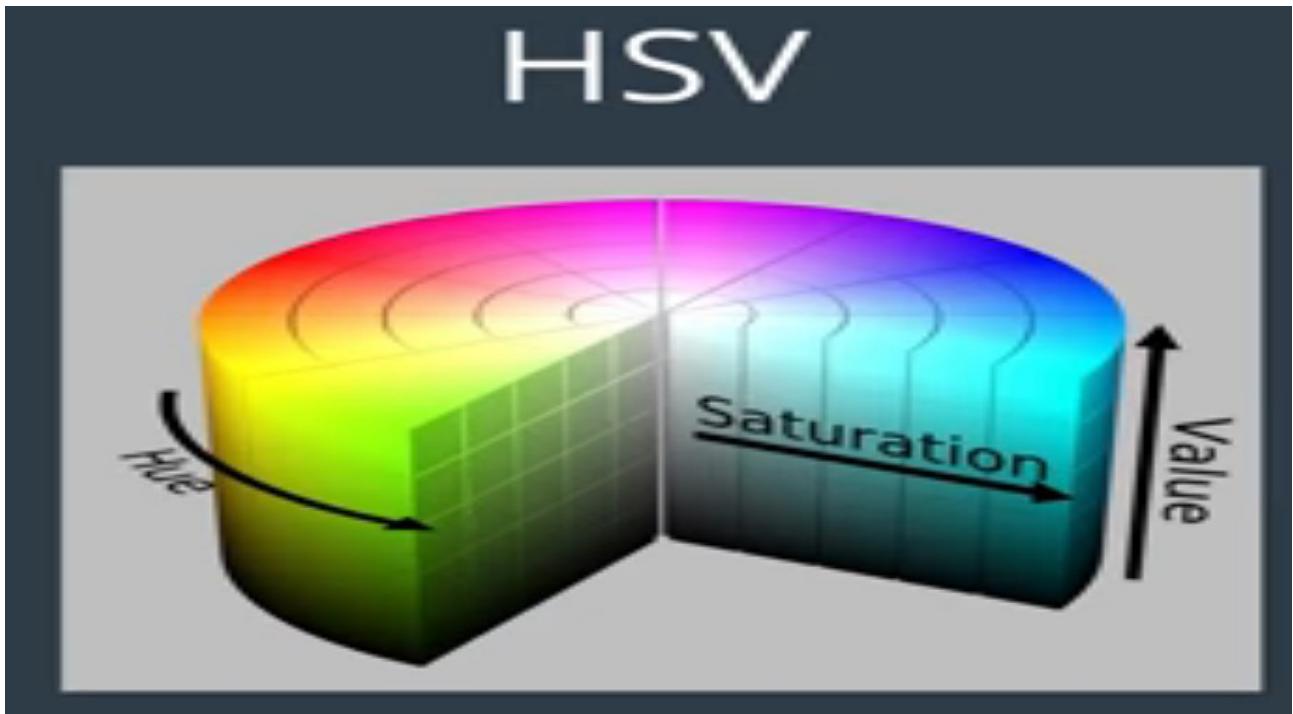
We can think of RGB values as filling a color grid like this.



Where your position along each of the axes defines how much red, green and blue we have in a point. An RGB representation of color does a nice job of reproducing what we see with our own eyes, but **it's not the most robust color representation for perception tasks in robotics**, because objects can appear to have quite a different color under different lighting conditions.

Fortunately, it's easy to convert your data to other color representations in order to make our thresholding or color selection operations less sensitive to changes in lighting.

Different color representations are known as color spaces, and one such color space that is particularly robust to lighting changes is HSV.



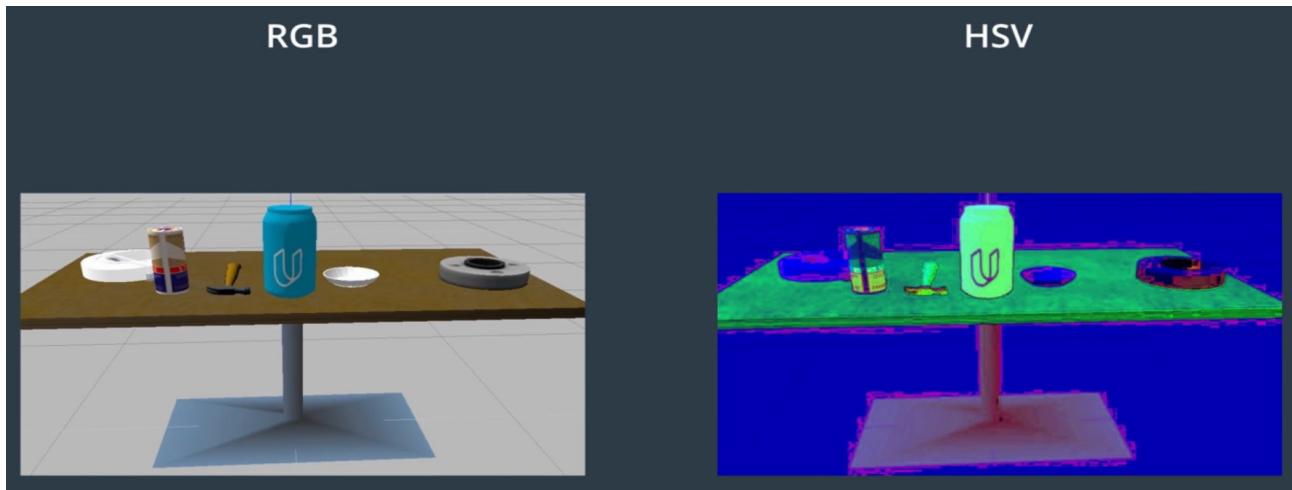
Which stands for hue, saturation and value. In the HSV space, color is represented by a cylinder like above.

We can think of the hue which is represented as angular position around the cylinder as describing what color is in a pixel, the saturation which is measured as radial distance from the center axes as being the intensity of that color, and value or aural brightness along the vertical axes.

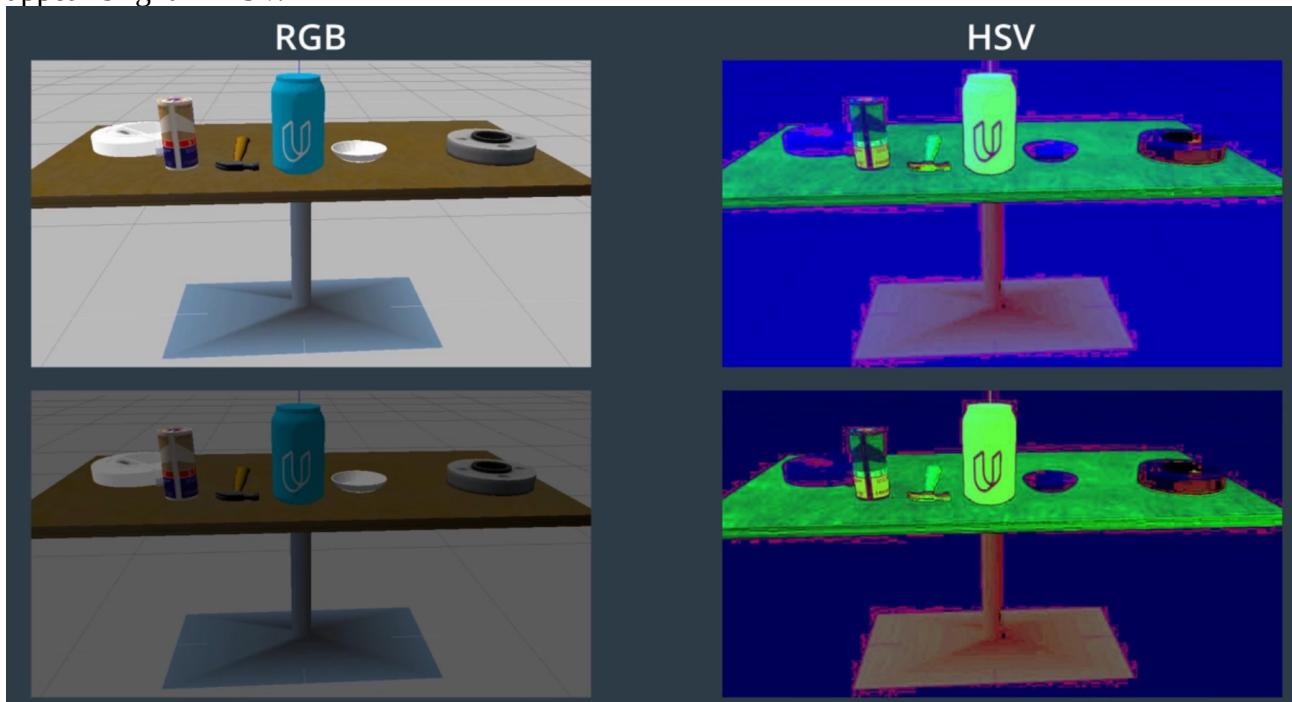
To convert RGB image to HSV color space using openCV, we can use the cv2 to convert color function like this:

```
$ hsv_image = cv2.cvtColor(rgb_image, cv2.COLOR_RGB2HSV)
```

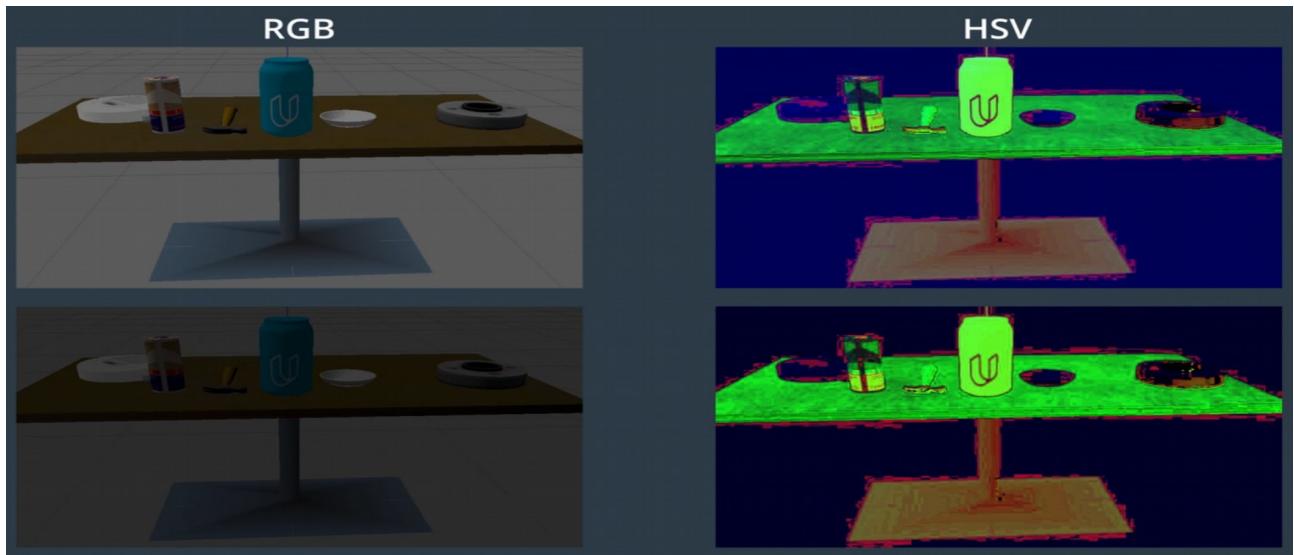
So here is a familiar image in RGB color space on the left and on the right what it looks like in HSV color space.



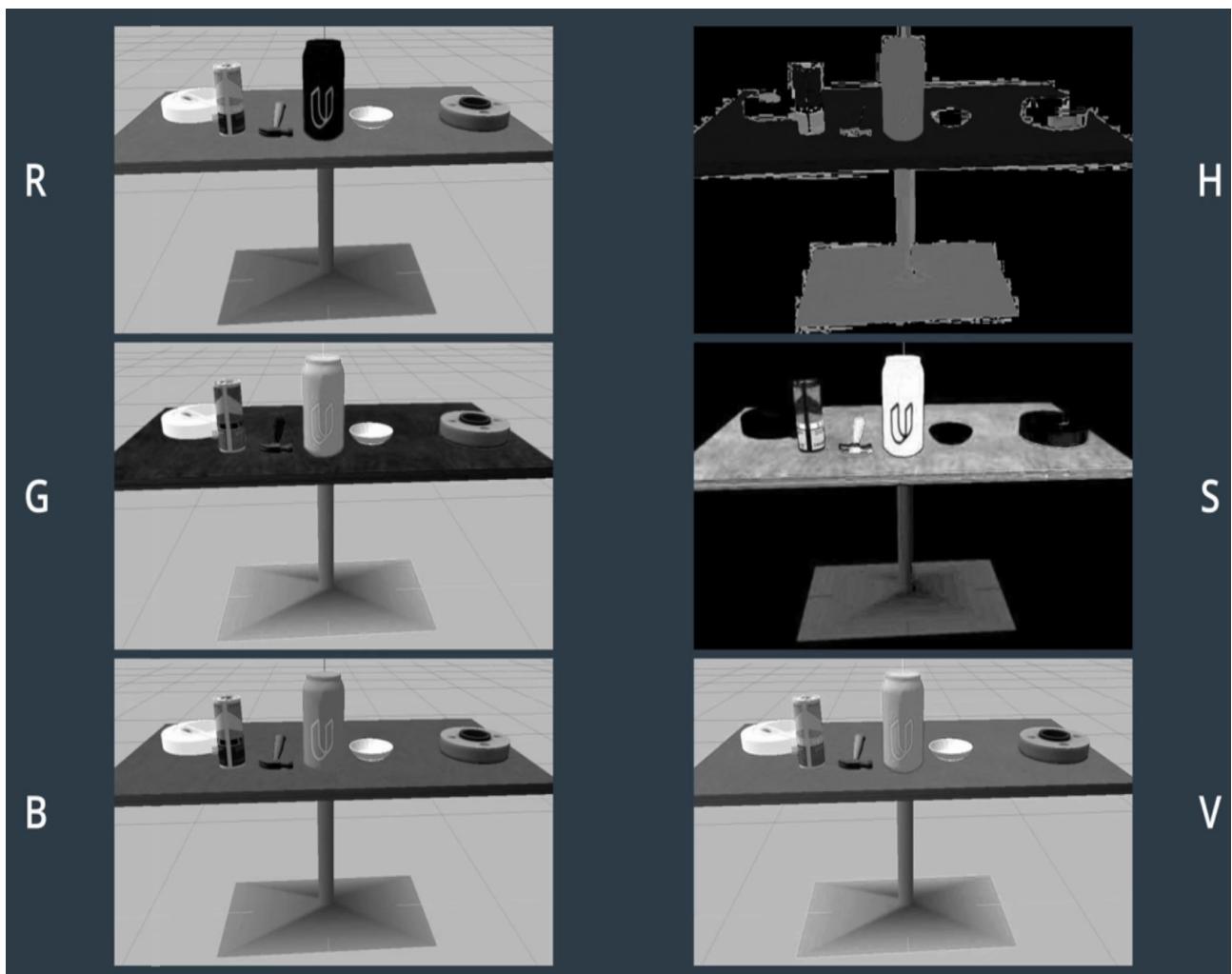
If we turn the lights down in the scene the rgb image looks like this, but the colorful objects still appear bright in HSV.



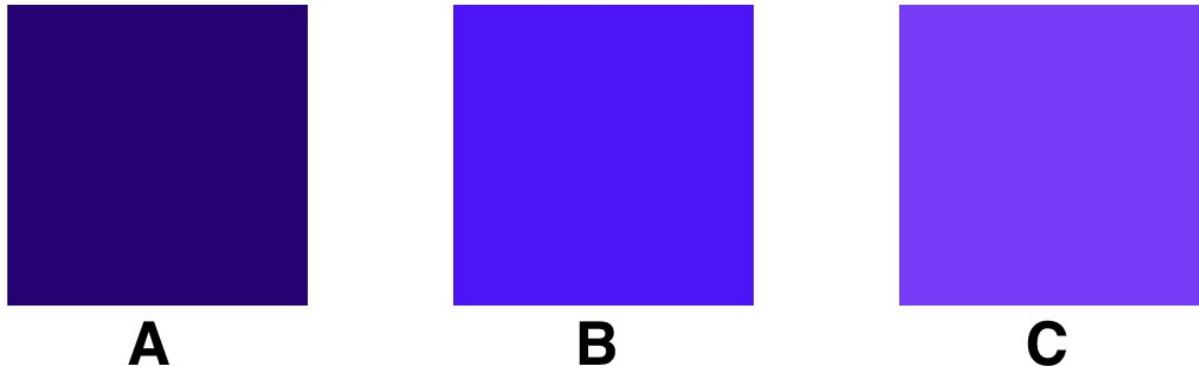
We can darken the image even further but the objects in the HSV image remain bright



Now let's have a look at the individual color channels.



Here we can really see the difference between RGB and HSV color spaces. While the individual red, green and blue channels all look relatively similar, the hue, saturation and value channels look dramatically different.



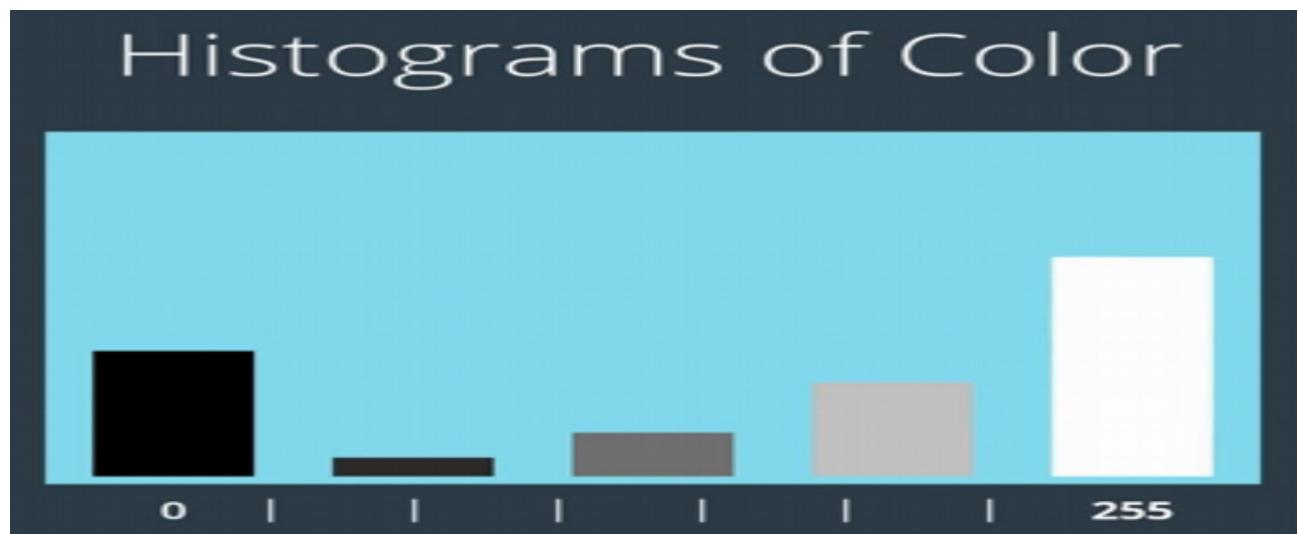
**Considering the colors pictured above, in HSV color space, which of these options will have the lowest V value?**

**Answer : A**

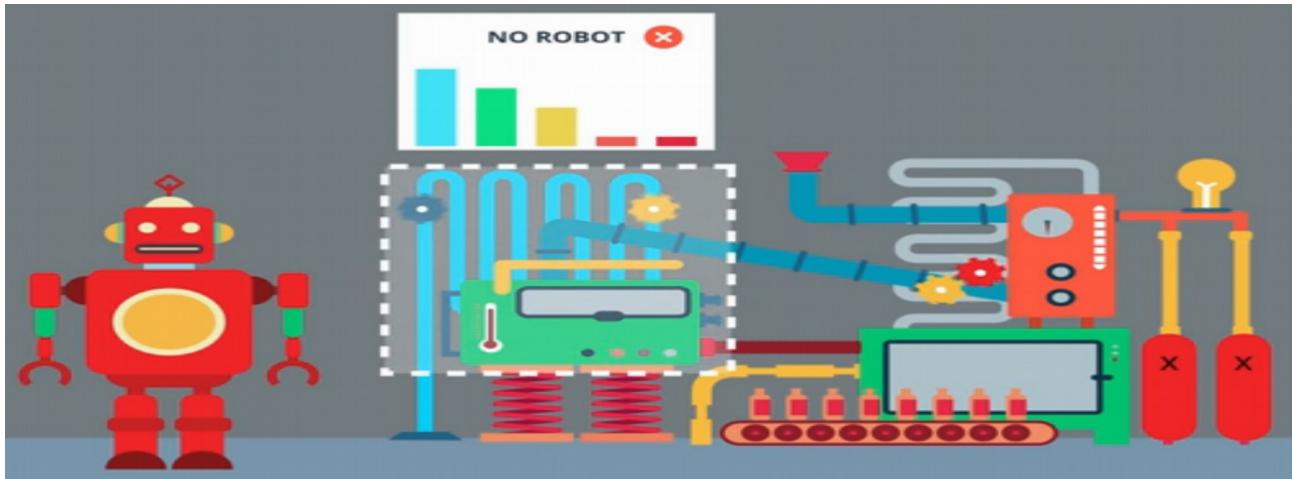
#### 7.5.4 Color Histograms

One way to convert color information into features that we can use for classification is by building up our color values into a histogram.

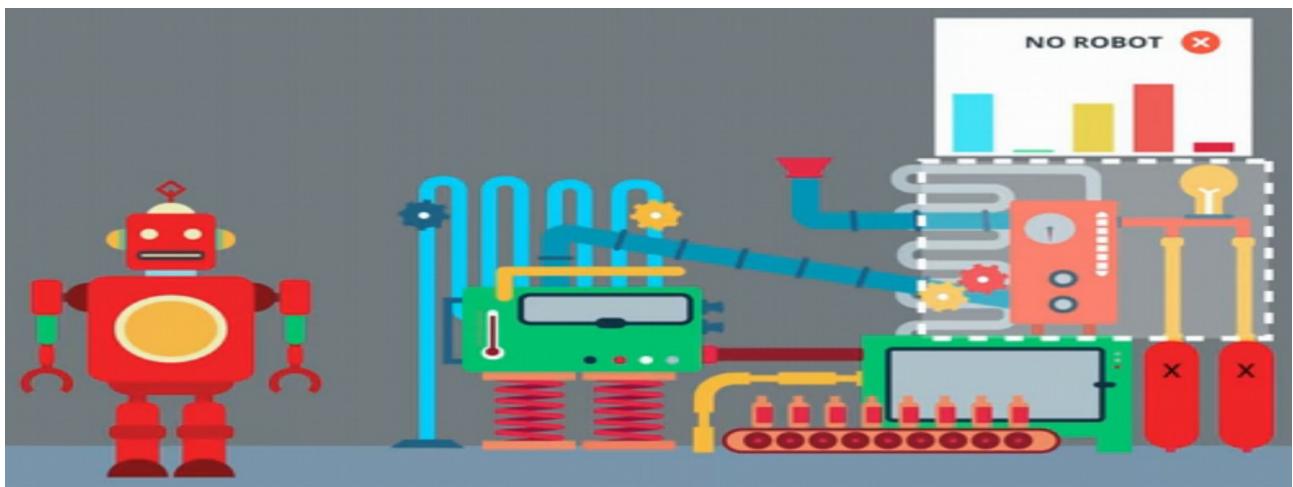
To construct a histogram we simple need to divide up the range of our data values 0-255 in this case into discrete bins. Then count up how many of the values fall into each bin.



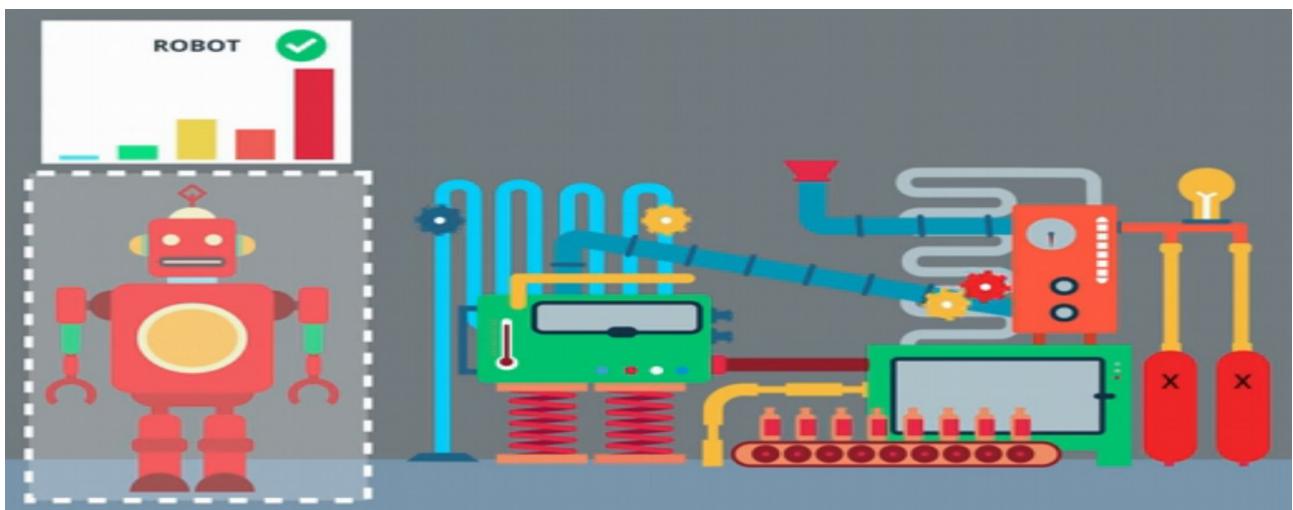
When we compare the colored histogram of a known object image with regions of a test image, location with similar color distributions will reveal a close match.



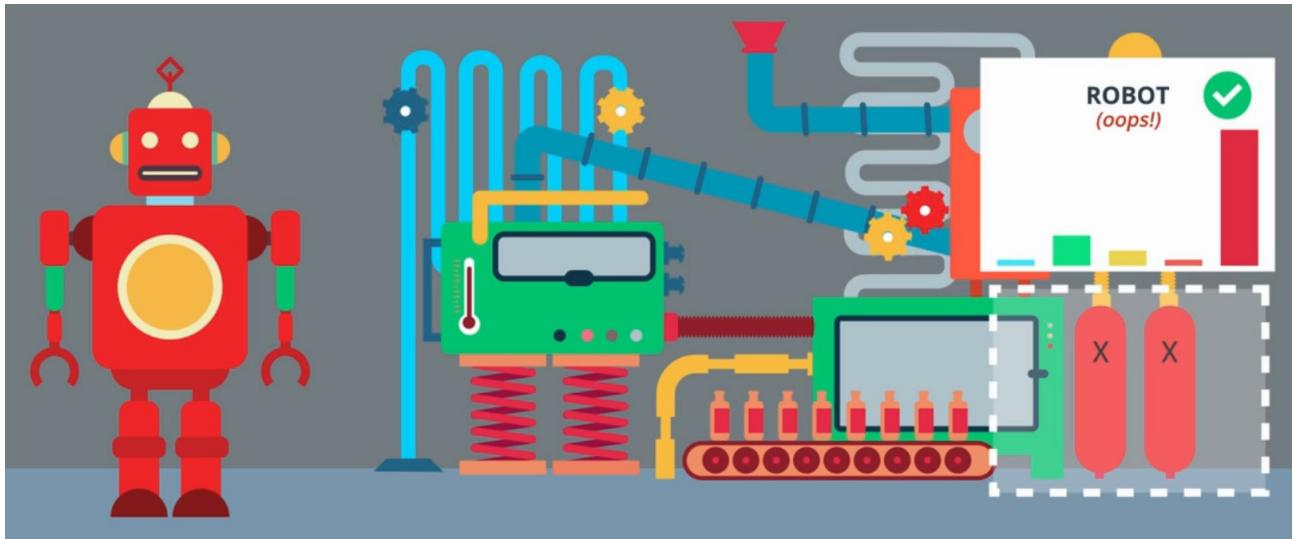
With this method we have removed any dependency on spatial structure, that is we are no longer sensitive to a perfect arrangement of points.



Therefore objects that appear in slightly different poses and orientation will still be matched. Variations in image size can also be accommodated by normalizing the histograms.



However, note that we are now solely relying on the distribution of color values which might match some unwanted regions resulting in false positives.



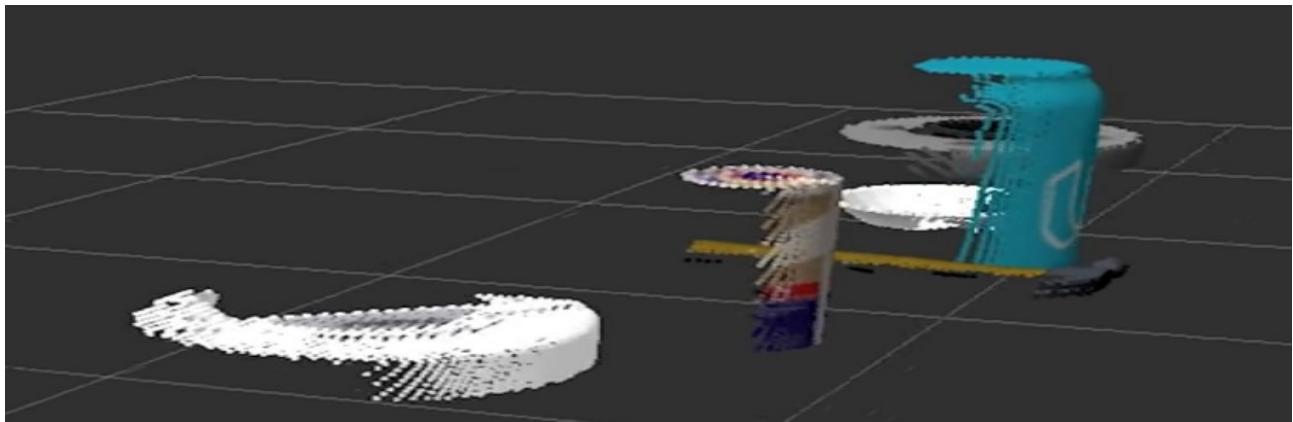
### 7.5.5 Surface Normals

So far, we have been talking about using color for object recognition, but another powerful way to find what we're looking for in our data is by searching for particular shapes vector Images, we can search for a given template shape or simply take the gradient of the image and explore the distribution of lights and edges that emerge.



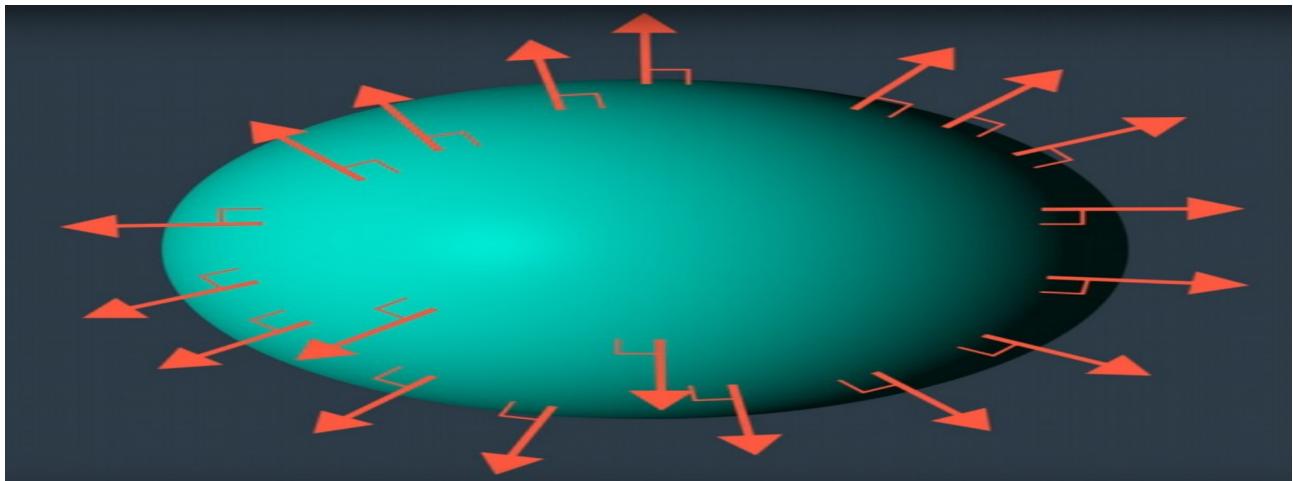
In this chapter, however we're working with 3D point clouds so we have an extra dimension of shape information to investigate.

In our point cloud, we have partial information on the 3D shapes of the object, which is to say we have the view of the object surface from just one perspective.

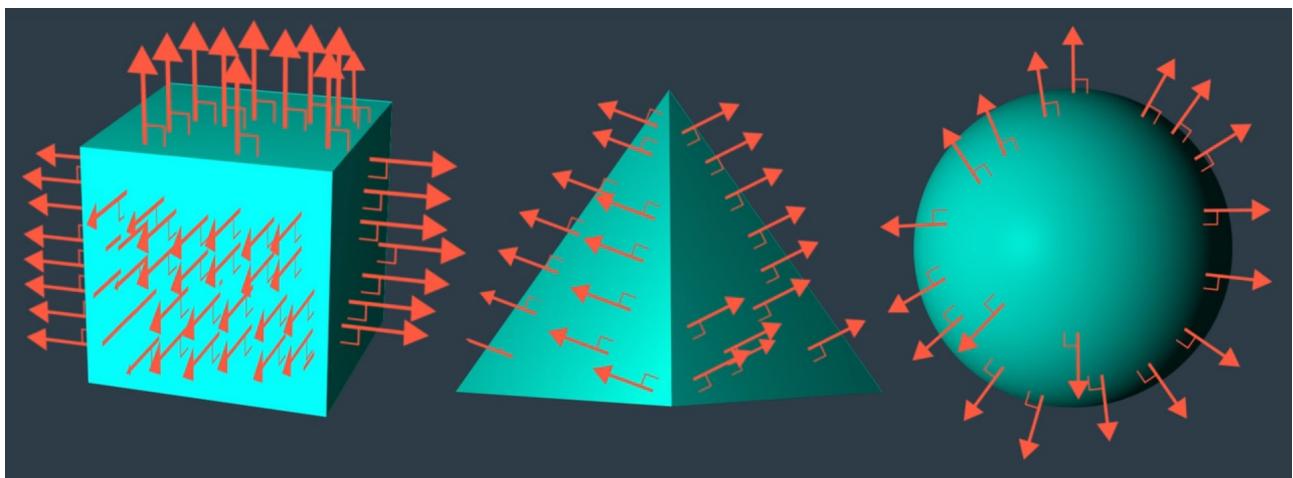


What we would like to do is to compare the distribution of points with a ground truth or reference distribution in order to decide whether or not we have found what we're looking for.

To do this we need a metric that captures shape and one such metric is the distribution of surface normals.



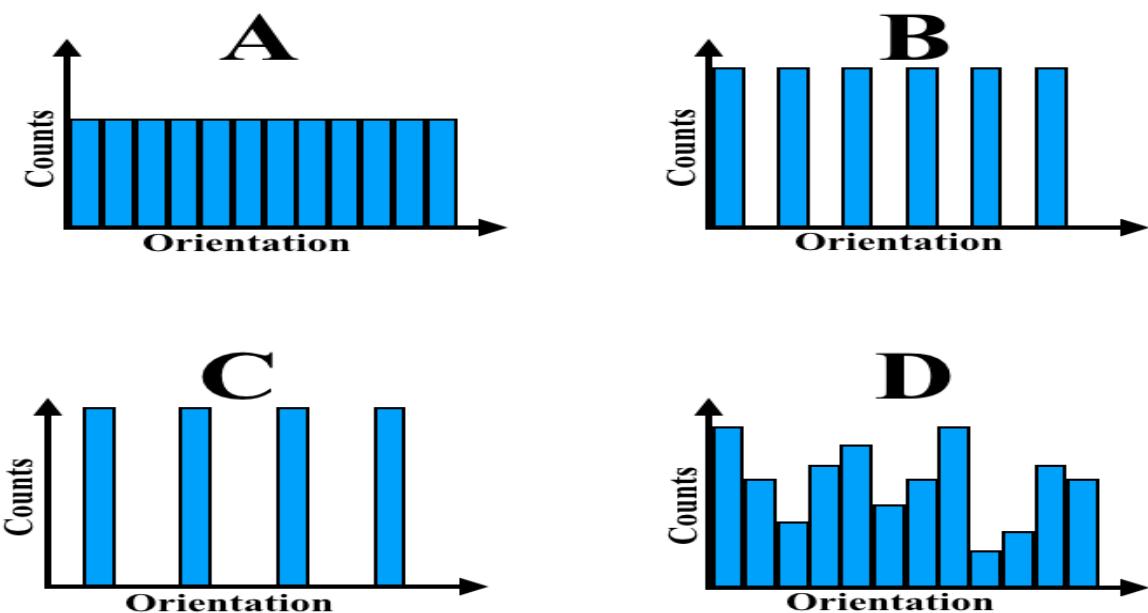
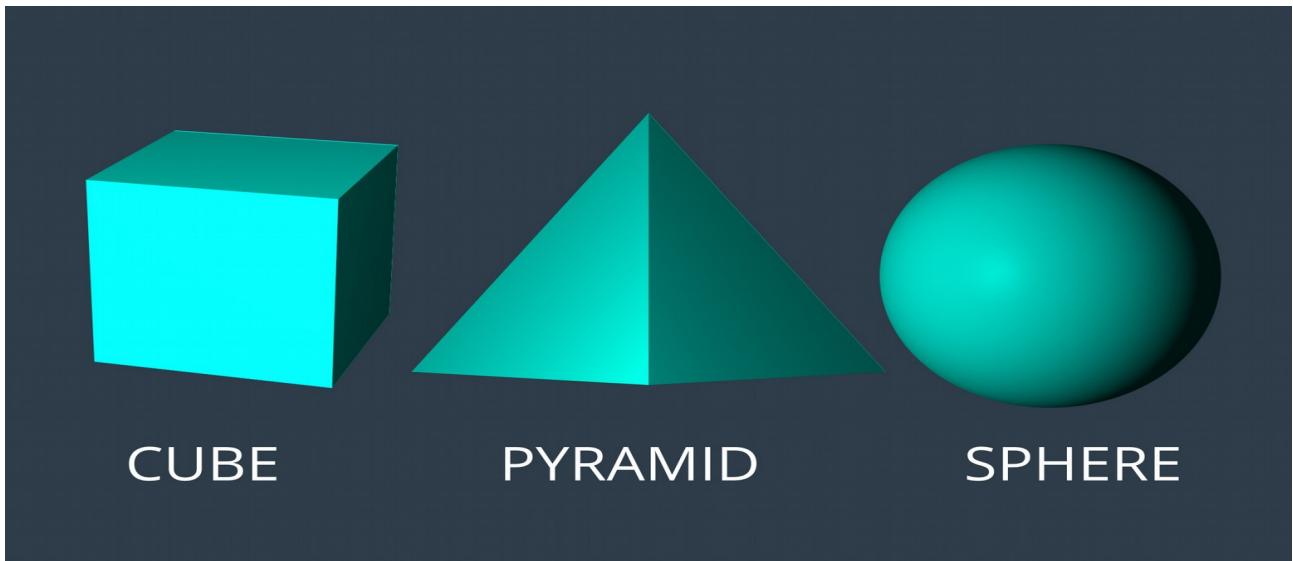
The normal, to any surface, is just a unit vector that is perpendicular to that surface. The normals at different points, along the changing surface, will point in different directions and the distribution of surface normals taken as a whole can be used to describe the shape of the object.



We can create this distribution just as we did with color by building up the individual surface normals into a histogram.

Next, we will take a look at the surface normals distributions for different objects so that we can get an intuition for how this metric characterizes shapes.

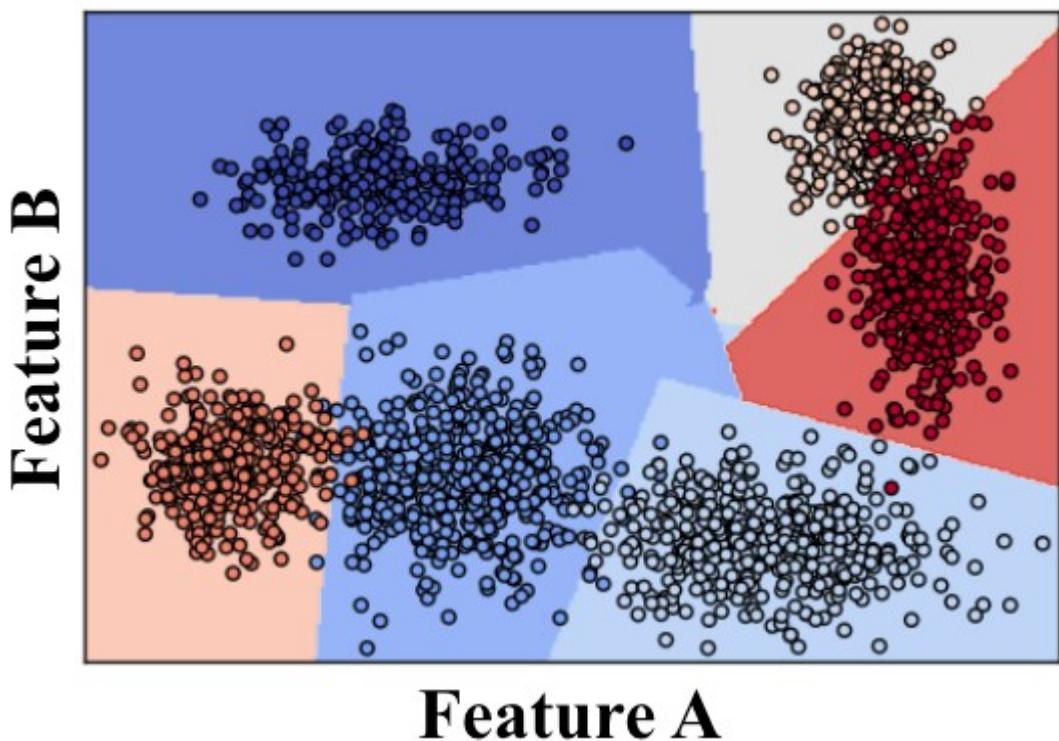
## Normals Intuition



Consider the shapes above (cube, sphere and pyramid) and the surface normal histograms labeled A, B, C and D. Assume that "orientation" is some continuous variable that describes which way the surface normals are pointing. Match each shape with the correct surface normal histogram.

- Cube : B
- Sphere : A
- Pyramid : C

## 7.5.6 Support Vector Machine



## Feature A

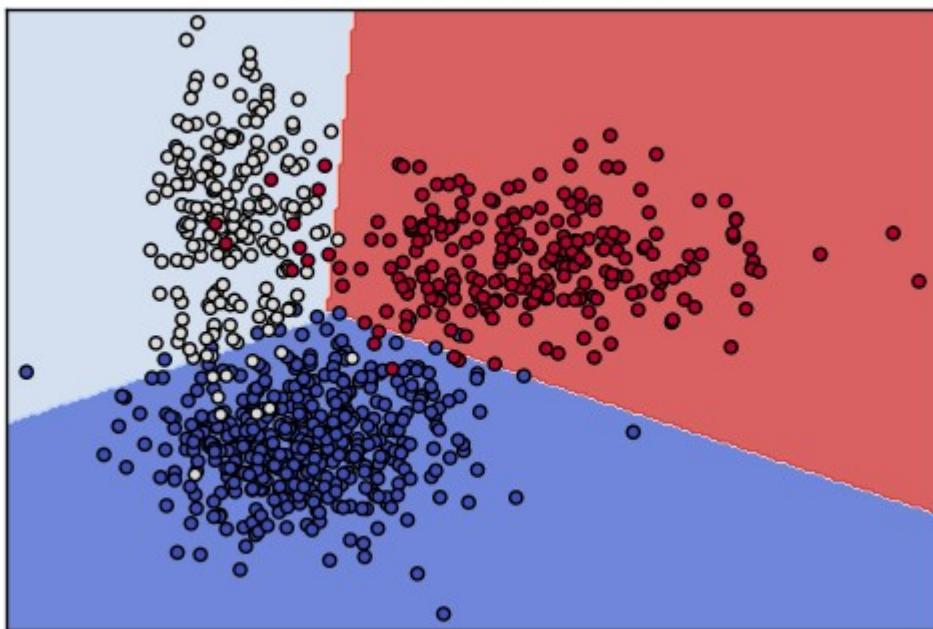
Support Vector Machine or "SVM" is just a funny name for a particular supervised machine learning algorithm that allows you to characterize the parameter space of your dataset into discrete classes.

SVMs work by applying an iterative method to a training dataset, where each item in the training set is characterized by a feature vector and a label. In the image above, each point is characterized by just two features, A and B. The color of each point corresponds to its label, or which class of object it represents in the dataset.

Applying an SVM to this training set allows you to characterize the entire parameter space into discrete classes. The divisions between classes in parameter space are known as "decision boundaries", shown here by the colored polygons overlaid on the data. Having created decision boundaries means that when you're considering a new object for which you have features but no label, you can immediately assign it to a specific class. In other words, **once you have trained your SVM, you can use it for the task of object recognition!**

## SVM Intuitions

SVC with linear kernel



**What happens when clusters are slightly overlapping?**

**SVC finds a decision boundary that best separates the clusters.**

**What happens when clusters are completely overlapping?**

**SVC does a poor job of separating them or fails completely.**

**What happens when you switch to an "rbf" kernel?**

**SVC takes longer to train but the decision boundaries are much more complex.**

**Besides linear and rbf, What other kernels are available with `sklearn.svm.SVC`?**

**poly, sigmoid, precomputed and callable.**

## 7.5.7 Summary

In this chapter, we've explored some powerful techniques for object recognition and pose estimation. These techniques are used in real robotics applications across the wide variety of industries.

Through this chapter we've been working with point cloud data using techniques like filtering and RANSAC fitting to isolate regions of interest in our point cloud.

After that we used clustering techniques to segment the data.

Finally, we applied object recognition algorithms to find exactly what we're looking for.

Sounds like a good time to bring all these skills and build a powerful perception pipeline

**So now we have the tools needed to put all this theory in action with ROS- PCL and sklearn**

**We will filter the point cloud data and with RANSAC model-fitting we will remove the ground or table from our point cloud, then we will segment our point cloud and divide into many smaller individual pointclouds.**

**After that we will use a simple Gazebo World with a kinect camera and we will spawn different objects in different orientations to extract features (a.k.a histograms of color spaces and surface normals)**

**With that features we will train an SVM to identify the different objects, so we know which is object is which and where, we will also put labels in RVIZ.**

**After that we can say to our Mobile Manipulator to identify a particular object and grap it!!**

**check out the appendix on how to do this in more detail and experimental results!!**

# 8. Kinematics of Serial Manipulators

## 8.1 Intro to Kinematics

### 8.1.1 Overview

Kinematics is a branch of classical mechanics that studies how things move **without considering the forces required to produce the motion**, and it's a subject area that is fundamental not only to robotics but also to many areas of engineering. More generally, kinematics is used to help answer all sorts of interesting questions, like how to optimize human movements, design complex machines, or even predict the motion of celestial bodies. We will see in the appendix how to solve a common robotic task picking up a object and placing it somewhere else, but to be successful, we'll first need to learn several important concepts including **reference frames, generalized coordinates, degrees of freedom and homogeneous transforms**.

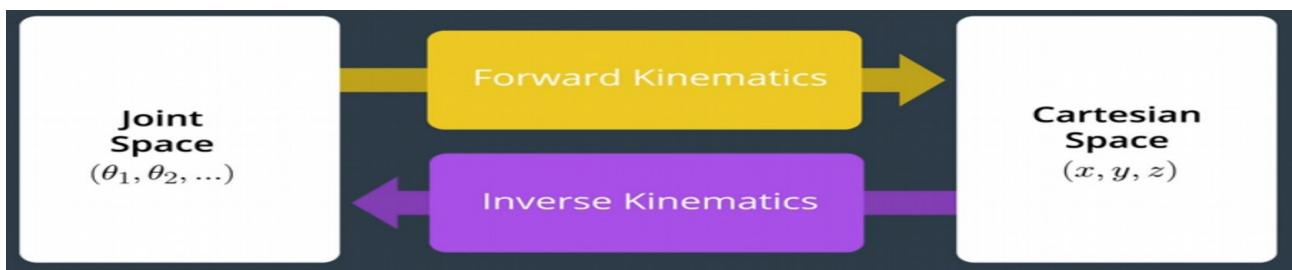
Reference frames are fundamental to kinematics, vectors such as position, velocity and acceleration are always described in the context of a reference frame.

Most robots like serial manipulators have multiple degrees of freedom which means that to understand the configuration of the robot at any given time, we need to know things like the angles and positions of various components of the robot simultaneously.

Using multiple reference frames to describe the configuration of your robot can greatly simplify the analysis, because vectors can always be described in the most convenient frame, but multiple reference frames create a challenge. How we can relate vectors expressed in one frame to those expressed in another?

The answer to this question is the homogeneous transform, which is a  $4 \times 4$  matrix that captures the relative rotation and translation between two frames. Next we will see how to chain multiple homogeneous transforms together so that for example, we can describe the position and orientation of the robot's end effector to that of the base.

By creating this mathematical expressions that describe the relationship between each of the reference frames in a system, we'll have the tools needed to solve the two basic kinematic problems associated with serial manipulators, known **as the forward and inverse kinematics problems**.



In forward kinematics, the joint variables are known, and the goal is to calculate the position of the end effector in Cartesian coordinates relative to the base frame. The aptly named inverse kinematics problem is the opposite scenario, here the pose or position and orientation of the end effector is known, and the joint angles that would achieve this pose must be calculated. **As we will see solving the inverse kinematics problem is much more challenging than the forward kinematics.**

**In this chapter we will explore the following concepts:**

- **Degrees of freedom (DoF)**
- **Reference frames**

- **Generalized coordinates**
- **Joint types**
- **Principal types of serial manipulators**
- **Serial manipulator applications**

### 8.1.2 Degrees of Freedom

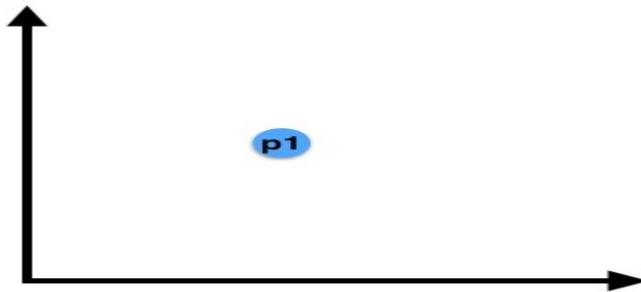
The phrase "Degrees of Freedom" (DoF) refers to the minimum number of variables that are required to define the position or configuration of a mechanism in space. Check out the following quizzes to get an intuitive understanding of what DoF represents and how it relates to a robot.

For a point (p1 above) that can move along a line, how many coordinates do you need to specify to describe its position?



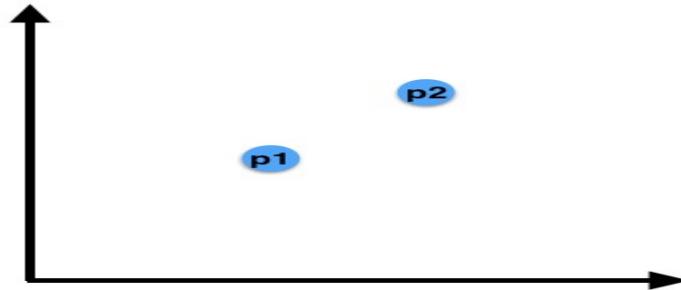
Answer : 1

What about a point that can move about on a plane? How many coordinates are required to specify its position?



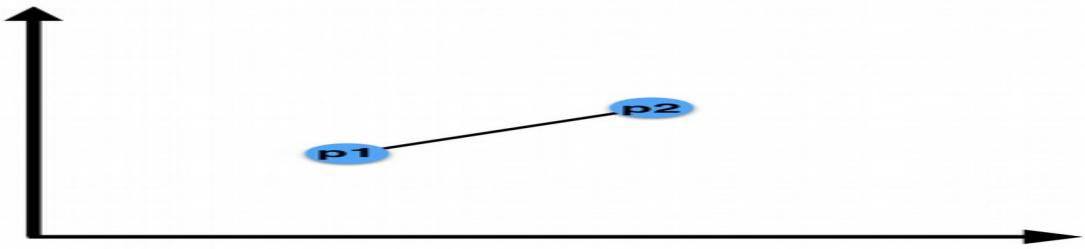
Answer : 2

How about two points on a plane? How many coordinates are required?



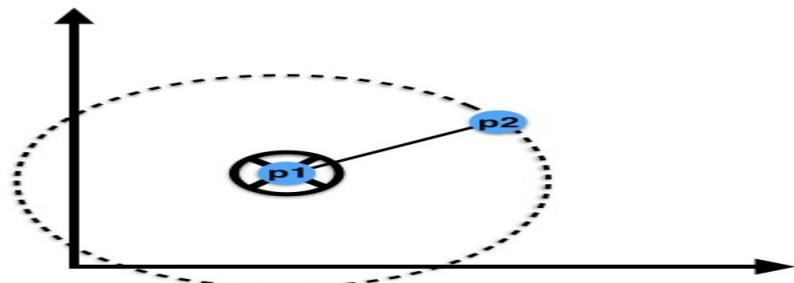
Answer: 4

Now what if we connect those two points with a rigid rod? How many coordinates do you need to describe this new system of two connected points on a plane?



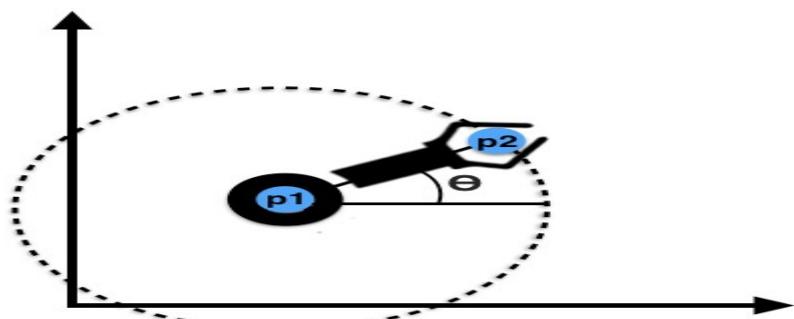
Answer: 3

Now what if we anchor the system on an axis at one of the two points ( $p_1$  in the image above), so now  $p_1$  is constrained to a fixed position and connected to  $p_2$  with a fixed length rod. How many degrees of freedom does this system have?



Answer: 1

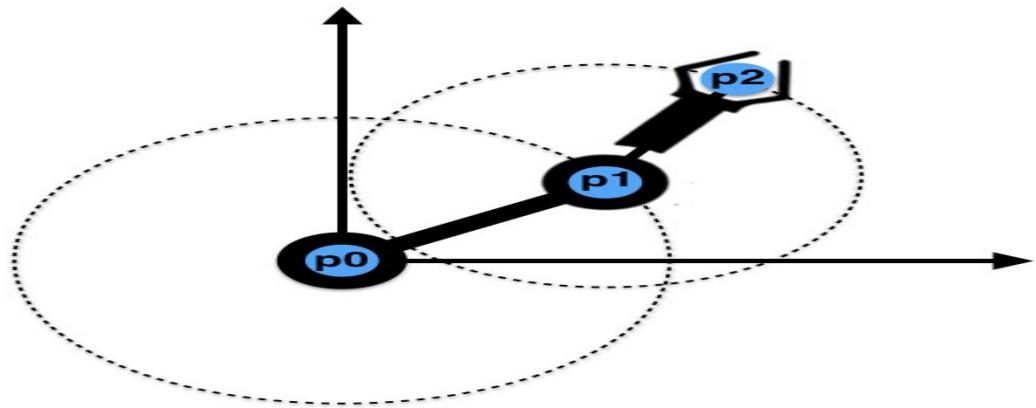
Is this system starting to remind you of something?



### 8.1.3 Two DoF Arm

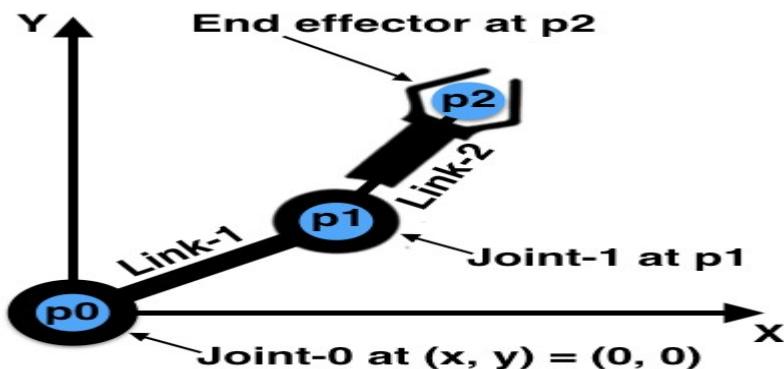
#### **Building a two-degree-of-freedom serial manipulator**

You've now seen how a system of two points can be constrained down to just one degree of freedom (DoF) by connecting the points with a rod and anchoring the system at one end. Now, let's add a third point and a second DoF to build a robotic arm! Our system now looks like the image below, where  $p_0$  is the base of the two-link arm.



Which quantities need to be specified as constraints to make this arm a 2-DoF system?

The position of p0 and the length of the links connecting p0 to p1 and p1 to p2

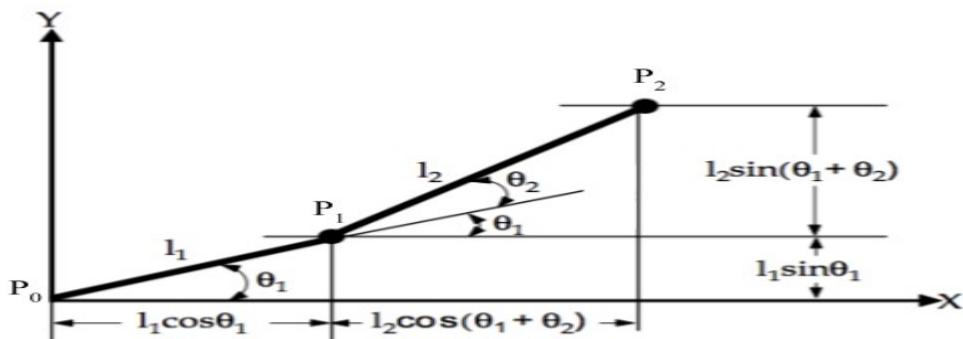


Here is our 2-DoF manipulator with all the parts labeled. Let's assume the base (Joint-0) is fixed at the origin (0,0) of our coordinate system.

With the constraints in place, which two additional parameters could be given to completely describe the configuration of the system at any time?

The angle of Link-1 away from the x-axis and the angle of Link-2 from the Link-1 axis

To solve the Forward Kinematics of this two DoF arm we also need the lengths of link1 and link2.



At the local frame 1, solving the position of  $P_1$  with respect to the global frame at  $P_0$ :

$$x_1 = l_1 \cos(\Theta_1)$$

$$y_1 = l_1 \sin(\Theta_1)$$

At the local frame 2, solving the position of  $P_2$  with respect to the reference frame at  $P_1$ :

$$x_2 = l_2 \cos(\Theta_1 + \Theta_2)$$

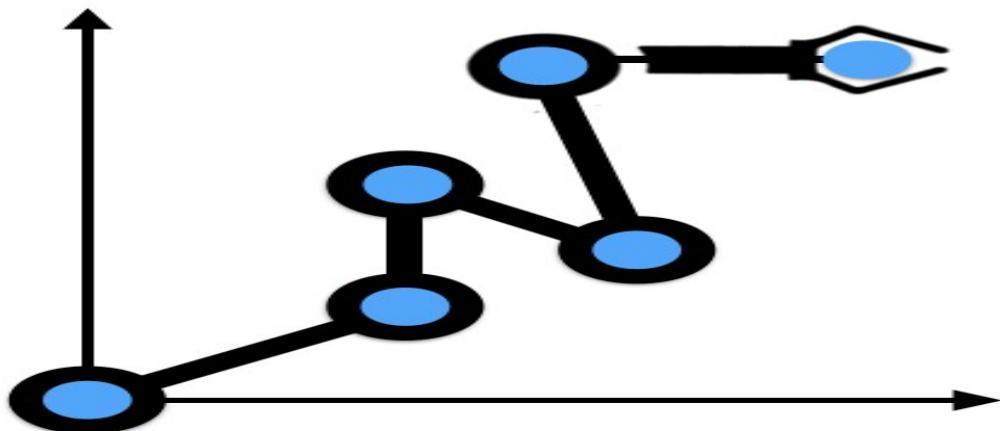
$$y_2 = l_2 \sin(\Theta_1 + \Theta_2)$$

Finally, putting it all together and solving the position of the end effector  $P_2$  with respect to the global frame at  $P_0$ :

$$x_2 = l_1 \cos(\Theta_1) + l_2 \cos(\Theta_1 + \Theta_2)$$

$$y_2 = l_1 \sin(\Theta_1) + l_2 \sin(\Theta_1 + \Theta_2)$$

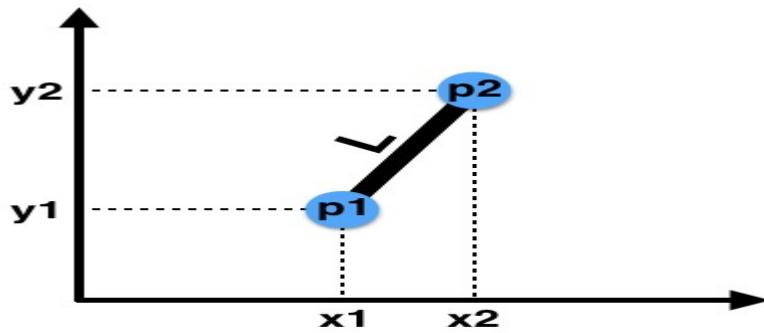
### 8.1.4 Generalized Coordinates



The same logic you applied in the last example to calculate the position of the end effector for a 2-DoF manipulator could be applied to any similar n-DoF system of joints and links constrained to a plane. With each new joint/link pair, you add a degree of freedom, and one additional coordinate is needed to fully describe the configuration of the system.

The coordinates used to describe the instantaneous configuration (snapshot in time) of a system are often called [generalized coordinates](#). The term "generalized" refers to the notion that for some arbitrary system these could be angles, x and y coordinates, or even some other quantity that has no geometrical significance.

In the previous 2-DoF arm example, the set of generalized coordinates you used consisted of the two joint angles. You could, however, have just as well used other sets of generalized coordinates, like for example, the angle of the first link and the x and y positions of the end effector, or the x and y positions of both the joint and end effector. However, in those cases you would have had 3 and 4 coordinates, respectively. This is because the x and y positions of the joint and end effector are not independent. They are related to one another through the constraint of the fixed link length ( $L$ ).



Solving for L using the [Pythagorean theorem](#):

$$L^2 = (x_1 - x_2)^2 + (y_1 - y_2)^2$$

Simplifying:

$$L = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

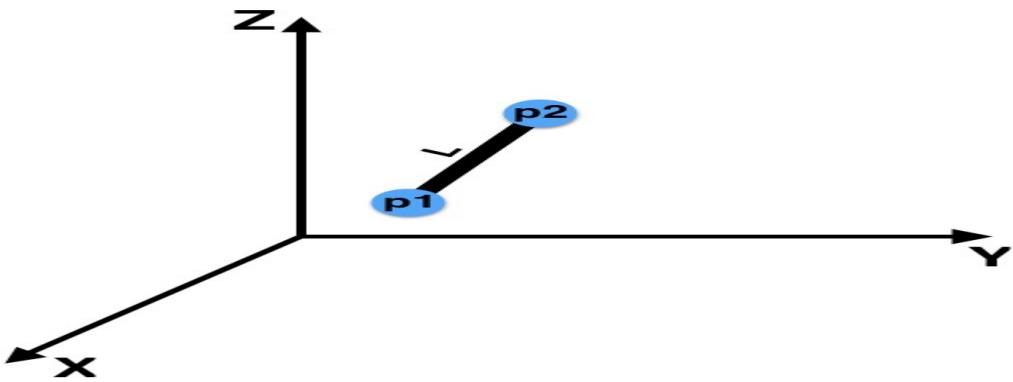
The x and y coordinates are related through this "constraint equation". The constraint that p1 and p2 are always separated by a fixed distance removes one degree of freedom the system.

### **DoF = number of independent generalized coordinates**

The number of independent generalized coordinates that are required to describe the configuration of a system is equal to the number of degrees of freedom. In the 2-DoF manipulator exercise we chose two particular joint angles with respect to a particular reference frame, but we could have just as easily chosen other angles with respect to some other reference frame. The number of possible choices for the generalized coordinates that describe a system is, in fact, infinite, but the smart choice of which set of coordinates to use is the one that most simplifies the problem you're trying to solve.

In the robotics of serial manipulators, you'll often run into the term **configuration space** or "joint space", which refers to the set of all possible configurations a manipulator may have. As you'll see later in this chapter, understanding of the configuration space is important for path planning and obstacle avoidance.

### **8.1.5 Rigid Bodies in Free Space**

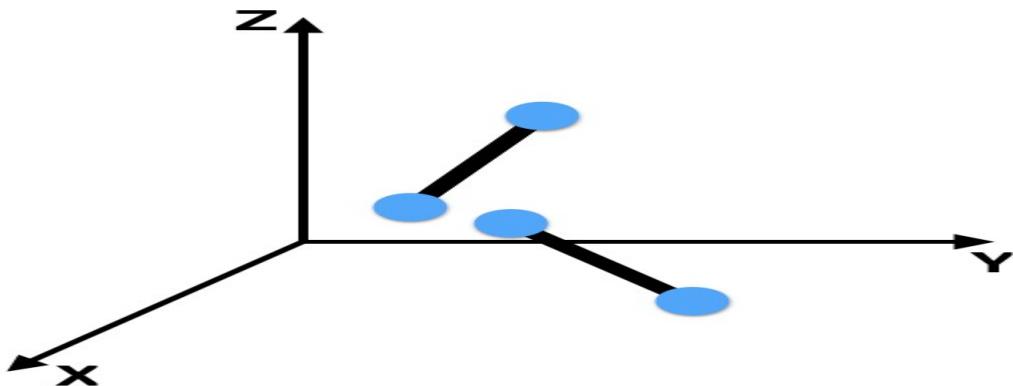


In the previous examples, you saw that two particles connected by rod and confined to a planar surface did not have four degrees of freedom, only three. This is, in fact, true of **any** rigid body moving on a plane. If you locate one point on the body and define its orientation with respect to some fixed axis, you have completely specified its configuration. Or, in other words, you could immediately determine the location of any other point on the rigid body with the knowledge of these three variables. What about a rigid body in “free space”, i.e., an unrestricted three-dimensional world?

How many coordinates do you need to specify in order to fully describe the configuration of a rigid body in 3-dimensional free space? Or in other words, how many degrees of freedom does it have?

**Answer: 6**

Individual rigid bodies are not particularly interesting in and of themselves, but in the world of serial manipulators, you can break most problems down in terms of rigid bodies connected by joints. Just as the addition of links between points in space reduces the number of degrees of freedom of a system (as seen in previous examples), the addition of joints to a system of otherwise disconnected bodies also reduces the number of degrees of freedom. So for starters, how many degrees of freedom do two independent rigid bodies in free space have?

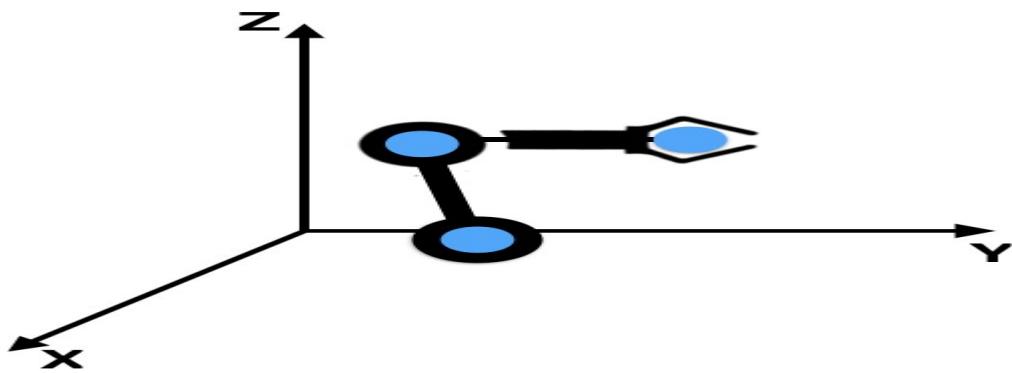


How many degrees of freedom do two rigid bodies in free space have? Or in other words, how many coordinates are required to fully describe the configuration of both simultaneously?

**Answer: 12**

### **Adding a Joint**

What if we now connect the two free floating rigid bodies with a revolute joint, such that the whole system is still free floating, but the two bodies are attached to one another?



Here we have our serial manipulator back, but now it is not constrained to a plane and the base is not fixed. How many degrees of freedom does this system have now?

**Answer:** 7

### Things to keep in mind

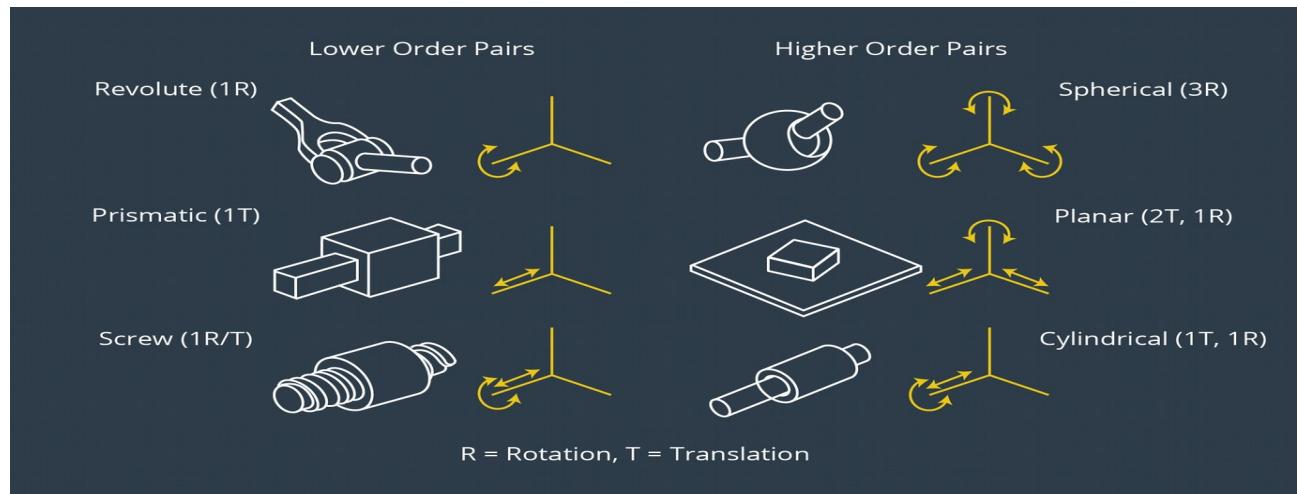
A **rigid body** has **6** degrees of freedom in space:

- Position(x, y, z)
- Orientation( $\alpha, \beta, \gamma$ )

A **point** has **3** degrees of freedom in space:

- Position(x, y, z)

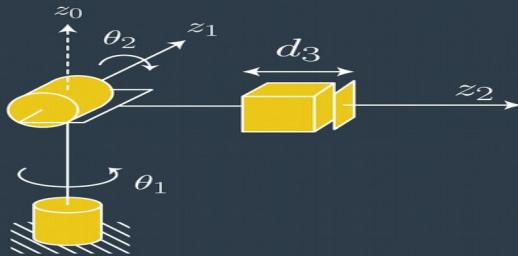
### 8.1.6 Joint Types



As shown in the above figure, joints can be grouped into lower order pairs having one DoF and higher order pairs offering two or three DoF. For serial manipulators (a.k.a., robot arm or kinematic chain), the most common joint types are revolute and prismatic allowing one rotational and one translational DoF, respectively.

As seen in the previous exercise, two rigid bodies in free space connected by a one DoF revolute joint would only have a total of  $2*6 - 5 = 7$  DoF. For modeling purposes, higher order pairs can always be replaced by a collection of lower order pairs, **so without loss of generality, we can restrict our attention to only the revolute and prismatic 1 DoF joint types.**

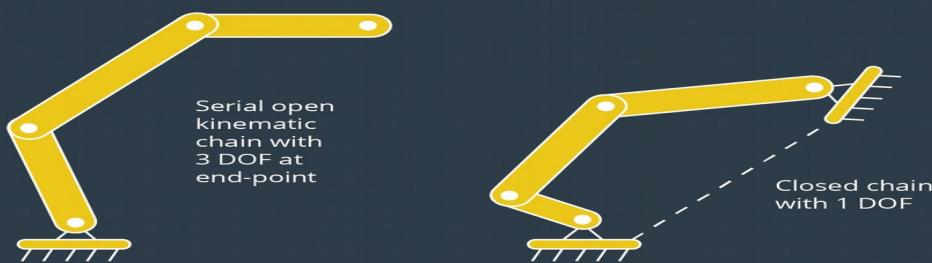
Let's look at another example to make this concept more explicit. Let's define the number of 1-DoF joints to be equal to  $n$ .



The serial manipulator shown here has  $n=3$  joints: 2 revolute (cylinders) and 1 prismatic (square). Each joint connects two links so the total number of links is  $n+1 = 4$ . Notice that the revolute described by  $\theta_1$  connects the ground (i.e., fixed "base link") to the link between joints 1 and 2. Thus the total number of DoF for any serial manipulator with three 1-DoF joints is:

$$\begin{aligned} \text{DoF} &= 6 \text{ (number of moveable links)} - 5 \text{ (number of 1-DoF joints)} \\ &= 6(3) - 5(3) \\ &= (18 - 15) \\ &= 3 \end{aligned}$$

The takeaway is this: For serial manipulators with only revolute and/or prismatic joints, the number of degrees of freedom is always equal to the number of joints. The exception to this rule is when both ends of the manipulator are fixed (closed chain linkage), as shown in the image below.



If a manipulator has more DoF than is required for its given task, it is said to be **kinematically redundant** or **Overconstrained**. For example, if a three DoF arm is used to locate a rigid body on a plane, it would have one degree of redundancy since only two generalized coordinates are needed to locate a point on a plane. However, to control both the position and orientation of a rigid body on a plane, a manipulator with at least three DoF is required.

Kinematically redundant manipulators have a number of advantages. The extra DoF means that they are more dexterous (End effector can reach more points with an arbitrary orientation) and better at avoiding obstacles. Because they have more flexibility in terms of path planning they can also be more energetically efficient. **However, the redundancy does come at a cost, in that they are more difficult to control.**

## SCARA

The selective compliance assembly robot arm known by SCARA is one of the famous serial robotic manipulators designed and built in 1981. The SCARA is composed of two revolute joints and a prismatic joint at its tip.



**Identify the number of degrees of freedom of a SCARA manipulator with its based link fixed on a table:**

**Answer: 3**

### 8.1.7 Principal Types of Serial Manipulators

Most serial manipulators used in industry have between 4 and 6 DoF.



Denoting the fixed base as Link-0 and the first movable link as Link-1, then links number 1, 2 and 3 are usually referred to the arm. **Any other jointed segments after the arm make up the wrist.** Quite often the wrist is composed of three revolutes whose axes of rotation all intersect at a **common point**, this type of wrist is known as the **spherical wrist** because it provides three rotational DoF just like a spherical joint. The toy robot shown above has a spherical wrist. As we will see later a spherical wrist has many nice properties in terms of the Jacobian, Inverse Kinematics and motion planning. **These properties are due to the fact that the first 3 joints**

**control the location of the wrist center while the last three only control the orientation of the end effector.**

One of the ways in which manipulators are classified is in terms of their kinematic configuration of their arm, that is the sequence of joint types for the first three movable links.

**With only revolute and prismatic joints, there are 8 possible permutations of arm types.  
However, only 4 types are commonly used in industry.**

		P = PRISMATIC R = REVOLUTE
1.	CARTESIAN	(PPP)
2.	CYLINDRICAL	(RPP)
3.	ANTHROPOMORPHIC	(RRR)
4.	SCARA	(RRP)
	SPHERICAL	(RRP)

The last two both have two revolutes and one prismatic joint, but they're joints axes are aligned differently, in the case of the SCARA robot all three joint axes are parallel.

### Spherical Wrist

A spherical wrist is composed of three revolute joints whose axes of rotation all intersect at a common point. This wrist has many properties which we will focus on them on the next chapter!

## 8.1.8 Serial Manipulators Applications

Here we take a slightly more in depth look at the principal types of manipulators by considering their pros, cons, and typical applications. We also discuss an important property known as the workspace (sometimes called work volume or work envelope).

### Workspace

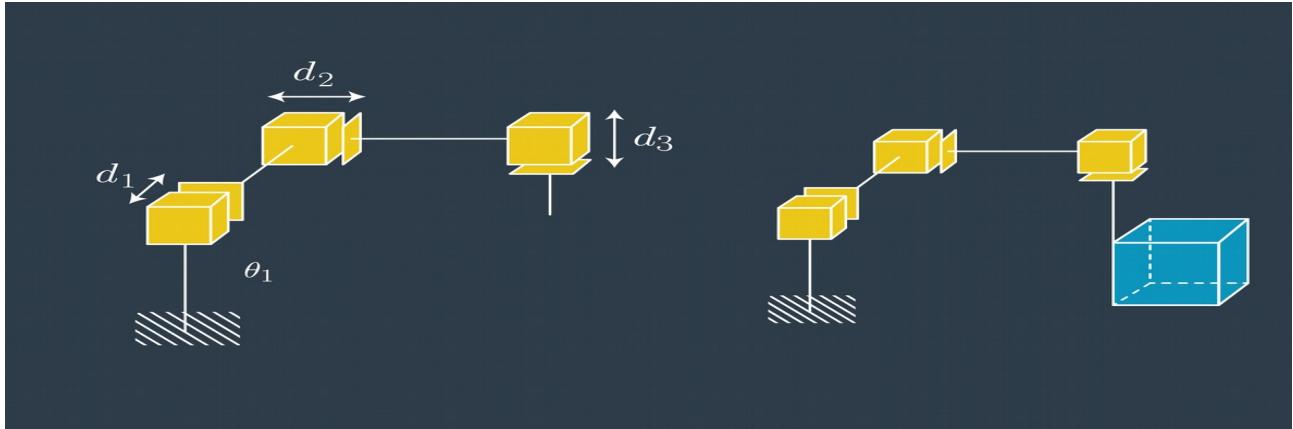
The workspace is the set of all points reachable by the end effector and is a primary design constraint when selecting a manipulator for a task. The workspace can be divided into two regions: the **reachable workspace**, i.e., what is implied by the simpler term workspace, and the **dextrous workspace**. The dextrous workspace is the set of all points reachable by the end effector with an *arbitrary* orientation. The dextrous workspace is a subset of the reachable workspace.

In many cases, e.g., machining or painting, the tool tip must interact with the environment in a particular configuration in order to have the desired result so ensuring the task lies completely

within the manipulator's dexterous workspace is essential. Unfortunately, it can be quite difficult to precisely define the boundary of the dexterous workspace.

**Note: in the descriptions that follow, the joint types of each manipulator are indicated in parentheses after the name, where "P" indicates a prismatic joint, and "R" indicates a revolute.**

## Cartesian Manipulator (PPP)



The first three joints of a Cartesian manipulator are prismatic joints with mutually orthogonal axes of translation.

### Pros

- Can have very high positional accuracy
- Large payloads (gantry)
- Simplest control strategy since there are no rotational movements
- Very stiff structure

### Cons:

- All the fixtures and associated equipment must lie within its workspace
- Requires large operating volume

### Typical Applications:

- Palletizing
- Heavy assembly operations (e.g., cars and airplane fuselage)

## Cylindrical Manipulator (RPP)



**As the name suggests, the joints of a cylindrical manipulator are the cylindrical coordinates of the wrist center relative to the base.**

#### Pros:

- Large, easy to visualize working envelope
- Relatively inexpensive for their size and payload

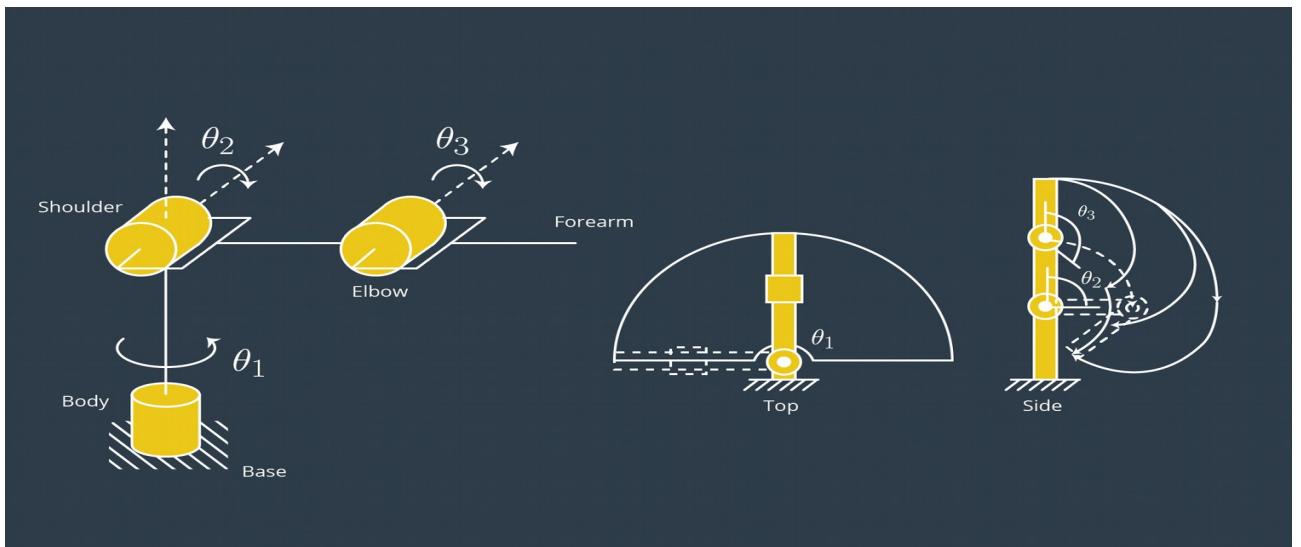
#### Cons:

- Low average speed
- Less repeatable than SCARA

#### Typical Applications:

- Depends on the size, small versions used for precision assembly, larger ones for material handling, machine loading/unloading

### Anthropomorphic Manipulator (RRR)



Anthropomorphic (sometimes called articulated) manipulators provide a relatively large workspace with a compact design. The first revolute joint has a vertical axis of rotation and can be thought of as mimicking a human's ability to rotate at the waist. The other two revolute joints have axes of rotation that are perpendicular to the "waist" and mimic a one DoF "shoulder" and a one DoF "elbow".

### Pros:

- Large workspace
- Compact design

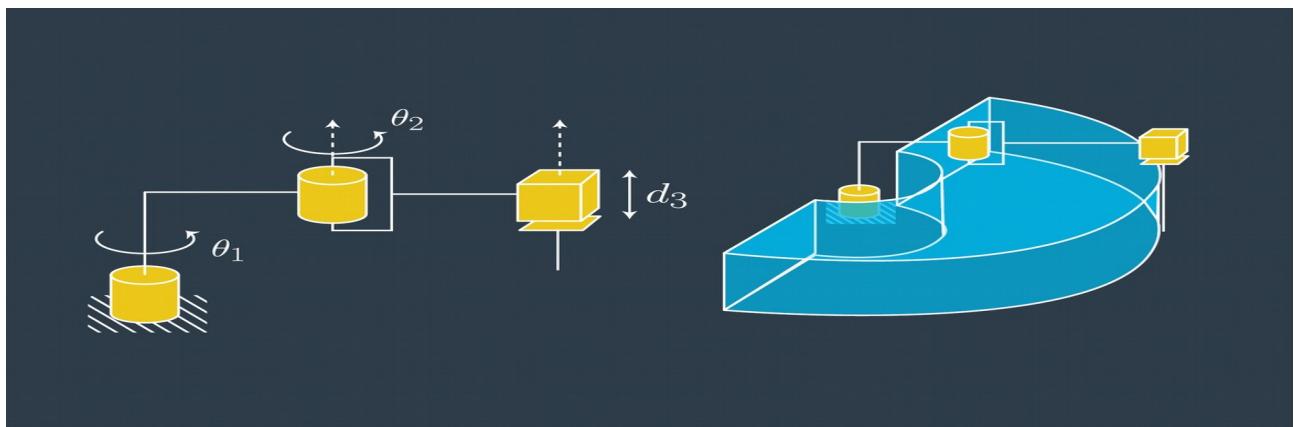
### Cons:

- Positional accuracy and repeatability is not as good as some other designs

### Typical Applications:

- Welding, spray painting, deburring, material handling

## SCARA (RRP)



The SCARA, or Selectively Compliant Assembly Robot Arm, was invented by Professor Hiroshi Makino of Yamanashi University (Japan) in the early 1980s. SCARA robots typically employ a single revolute wrist with the axis of rotation parallel to the other two revolute joints. Since the base link typically houses the actuators for the first two joints, the actuators can be very large and the moving links relatively light. Thus, very high angular speeds are obtainable with this design. The arm is very stiff in the vertical (z-axis), but relatively compliant in the x-y plane, which makes it ideal for tasks such as inserting pegs or other fasteners into holes.

### Pros:

- Fast
- Compact structure

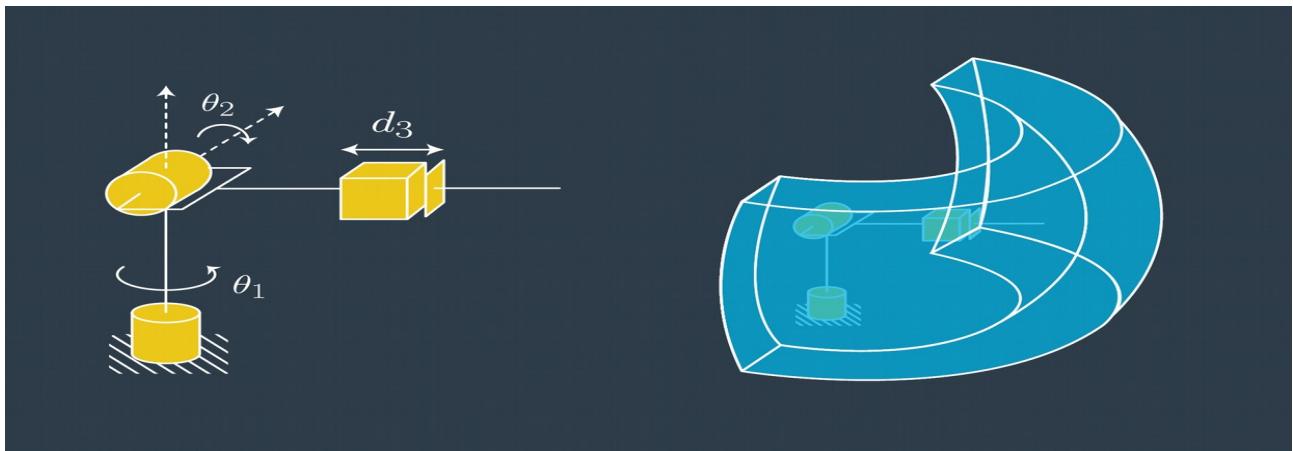
### Cons:

- Operations requiring large vertical motions

### Typical Applications:

- Precision, high-speed, light assembly within a planar environment

## Spherical (RRP)



Like the cylindrical manipulator, the spherical manipulator's wrist center can also be described as a well-known coordinate system. Probably the best known version of this kinematic type is Stanford's Scheinman arm, invented by [Victor Scheinman](#) in 1969.

It was adapted by manufacturers to become the leading robot in assembling and spot-welding products, ranging from fuel pumps and windshield wipers for automobiles to inkjet cartridges for printers.

#### **Pros:**

- Large working envelope

#### **Cons:**

- Complex coordinates more difficult to visualize, control, and program
- Low accuracy
- Relatively slow

#### **Typical Applications:**

- Material handling
- Spot welding

One other type that we do not cover in this chapter but you should at least be aware of is called a *parallel manipulator*. [Parallel manipulators](#) have many variants but they are characterized by all having at least one closed kinematic chain. The dynamics and control strategies can be quite a bit more complex than serial manipulators, but in general they have more precise movements due to their structural rigidity.

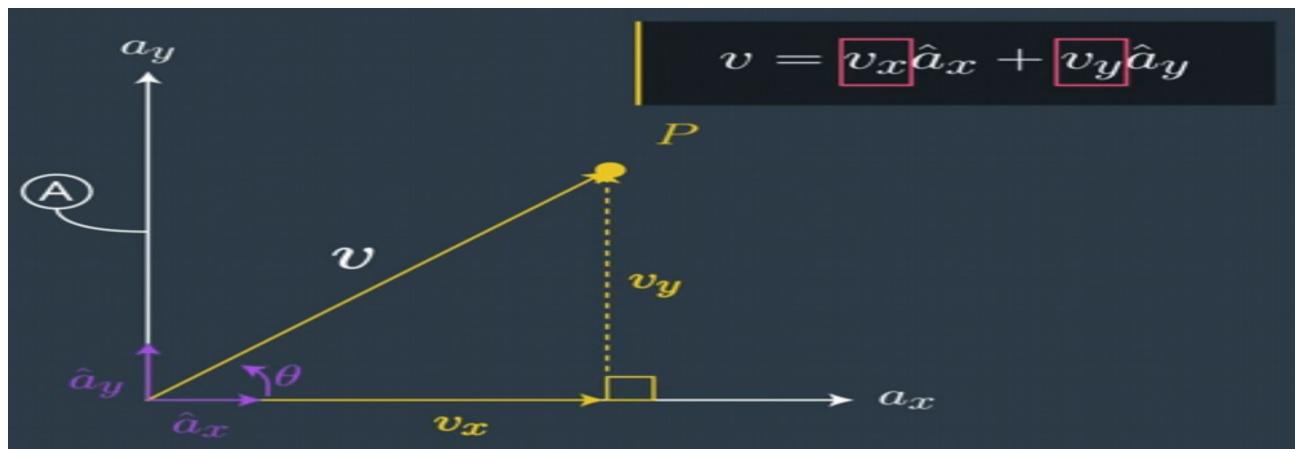
## 8.2 Forward and Inverse Kinematics

### 8.2.1 Setting up the Problem

Hopefully you've enjoyed the tour of manipulators so far. Now, it's time to begin exploring the mathematics needed to explain how they move. In order for a serial manipulator to be useful, it's essential that we know how the end-effector moves in a 3D environment. Perhaps, we would like to have the manipulator pick up an object from the assembly line and place it in a shipping container. Although it's very easy to think solely about the translation of the object from point A to point B in some xyz coordinate system, manipulators have actuators that control revolute and prismatic joints, and the motion of these links can be highly nonlinear. Converting the manipulator's generalized coordinates into the position of the end-effector is known as solving the forward kinematics problem. The first step to solving the forward kinematics problem is to see how to express vectors in different reference frames.

### 8.2.2 Coordinate Frames and Vectors

A vector is a mathematical quantity that has both magnitude and direction, these concepts only make sense in the context of a coordinate or reference frame.



Here, we have a two-dimensional reference frame A that has orthogonal coordinate axes  $a_x$  and  $a_y$ , a vector  $v$  in this reference frame extends from the origin to a point P. This vector could represent velocity, force acceleration or other things but for this example let's assume that it represents the position of point P. Graphically v can be expressed in terms of its components, that is, its basis vectors as follow or as an equation. The coefficients  $v_x$  and  $v_y$  are called measure numbers because they measure how much of  $v$  is pointing in the direction defined by the unit vector  $a_x\text{-hat}$  and  $a_y\text{-hat}$ . In other words,  $v_x$  and  $v_y$  are the magnitudes of  $v$  acting in the  $a_X$  and  $a_y$  directions.

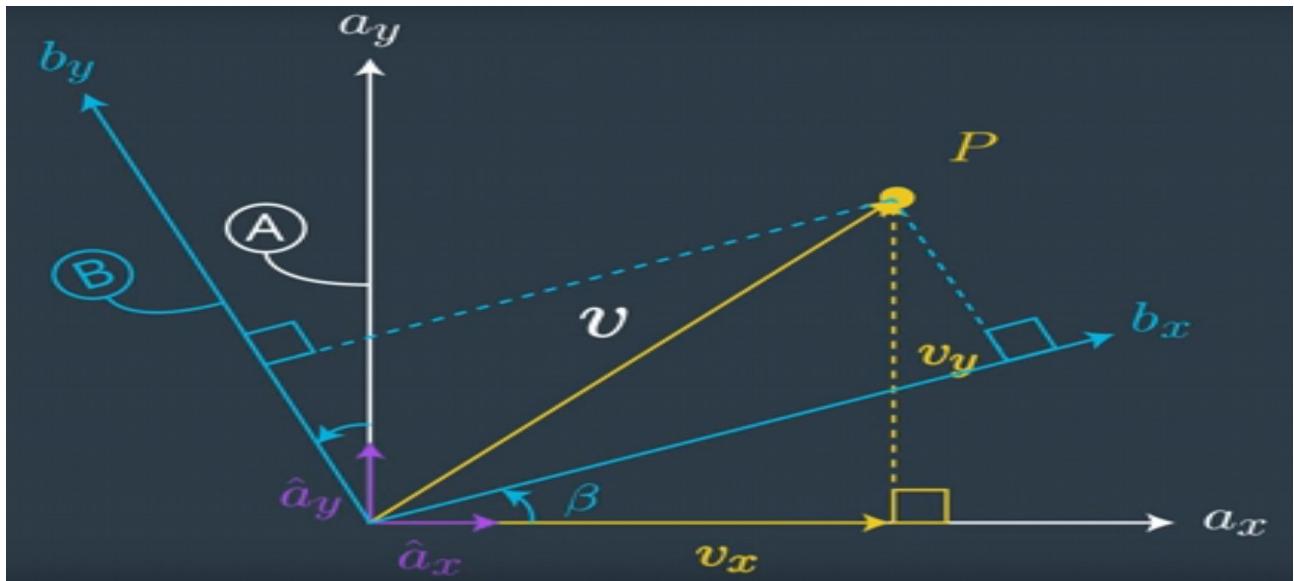
So the question is, how do we determine numerical values for the measure numbers?

Well, we project  $v$  onto  $a_x\text{-hat}$  and  $a_y\text{-hat}$  using the dot product operator. The dot product of  $v$  and  $a_x\text{-hat}$  is given by this expression and since cosine is defined as the adjacent side over the hypotenuse, you can see that the measure number  $v_x$  is equal to  $v$  times cosine theta.

$$v \cdot \hat{a}_x = \|v\| \|1\| \cos(\theta)$$

$$\cos(\theta) = \frac{v_x}{v} \Rightarrow v_x = v \cos(\theta)$$

What you should recognize, however, is this: there is nothing special about the A frame shown here.



Its orientation is undoubtedly the one you're most accustomed to seeing, but it is not unique, as we can see with the B coordinate frame, on which the vector v is the same with the exact same magnitude. Clearly the measure numbers of  $b_x$ -hat and  $b_y$ -hat are different than the measure numbers in the A frame. We can see this more easily by comparing the y values and noticing that the blue frame has a smaller y value. If you understand this then the next topic rotation matrices is just a slight extension of these. With rotation matrices all we are doing is projecting vectors expressed in one frame onto some other frame.

#### Additional Material

If you'd like to review some of the concepts related to vectors, the following resource may be useful

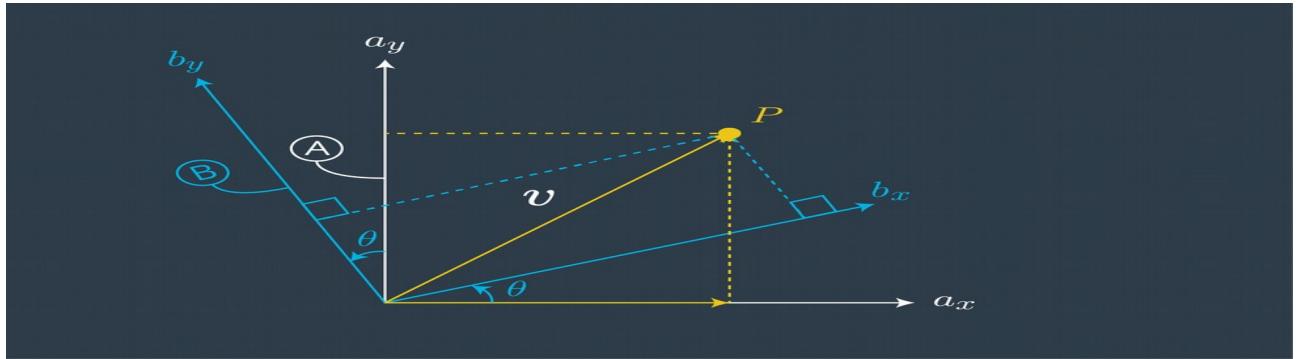
- [Khan Academy - Vectors and Spaces](#)

### 8.2.3 Rotation Matrices

Rotation matrices have two conceptual, but mathematically equivalent, interpretations. They can be viewed as a means of "expressing" a vector in one coordinate frame in terms of some other coordinate frame. This interpretation is known as a "mapping" between frames.

Alternatively, a rotation matrix can be seen as an "operator" that actually moves a vector within a single coordinate frame. It is important to be aware of this conceptual distinction because the

particular application will dictate which description is used, but do not let it confuse you: it's the same math!



Let's again consider a vector,  $\mathbf{v}$ , with respect to reference frames  $_A$  and  $_B$  and take a more formal look at how to express  $\mathbf{v}$ , whose measure numbers we may know in frame  $_A$ , in terms of frame  $_B$ .

$$\mathbf{v} = \mathbf{v}_x \hat{\mathbf{a}}_x + \mathbf{v}_y \hat{\mathbf{a}}_y = \mathbf{u}_x \hat{\mathbf{b}}_x + \mathbf{u}_y \hat{\mathbf{b}}_y \quad (1)$$

If we dot equation (1) with  $a^\wedge x$  we get:

$$\mathbf{v} \cdot \hat{\mathbf{a}}_x = \mathbf{v}_x \hat{\mathbf{a}}_x \cdot \hat{\mathbf{a}}_x + \mathbf{v}_y \hat{\mathbf{a}}_x \cdot \hat{\mathbf{a}}_y = \mathbf{u}_x \hat{\mathbf{a}}_x \cdot \hat{\mathbf{b}}_x + \mathbf{u}_y \hat{\mathbf{a}}_x \cdot \hat{\mathbf{b}}_y \quad (2)$$

Since  $a^\wedge x$  and  $a^\wedge y$  are orthogonal unit vectors so  $a^\wedge x \cdot a^\wedge x = 1$  and  $a^\wedge x \cdot a^\wedge y = 0$  so equation (2) simplifies to:

$$\mathbf{v}_x = \mathbf{u}_x \hat{\mathbf{a}}_x \cdot \hat{\mathbf{b}}_x + \mathbf{u}_y \hat{\mathbf{a}}_x \cdot \hat{\mathbf{b}}_y \quad (3)$$

Similarly, if we dot equation (1) with  $a^\wedge y$  and simplify, we get

$$\mathbf{v}_y = \mathbf{u}_x \hat{\mathbf{a}}_y \cdot \hat{\mathbf{b}}_x + \mathbf{u}_y \hat{\mathbf{a}}_y \cdot \hat{\mathbf{b}}_y \quad (4)$$

We can arrange equations (3) and (4) into a more compact matrix form,

$$\begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{a}}_x \cdot \hat{\mathbf{b}}_x & \hat{\mathbf{a}}_x \cdot \hat{\mathbf{b}}_y \\ \hat{\mathbf{a}}_y \cdot \hat{\mathbf{b}}_x & \hat{\mathbf{a}}_y \cdot \hat{\mathbf{b}}_y \end{bmatrix} \begin{bmatrix} \mathbf{u}_x \\ \mathbf{u}_y \end{bmatrix} \quad (5)$$

And after simplifying equation (5) by solving the dot products.

The first term on the right-hand side of the

$$\begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} \mathbf{u}_x \\ \mathbf{u}_y \end{bmatrix}$$

equality in equation (5) is known as a rotation matrix and is generally written in a more compact form,  $\mathbf{a}\text{-b:R}$ .

Examining the columns of the rotation matrix, you can see that they are in fact the basis vectors of frame  $\mathbf{_B}$  expressed in terms of the frame  $\mathbf{_A}$ . That is,

$$\mathbf{\hat{A}^B R} = [\mathbf{\hat{A}^B b_x \quad \hat{A}^B b_y}] = \begin{bmatrix} \hat{\mathbf{a}}_x \cdot \hat{\mathbf{b}}_x & \hat{\mathbf{a}}_x \cdot \hat{\mathbf{b}}_y \\ \hat{\mathbf{a}}_y \cdot \hat{\mathbf{b}}_x & \hat{\mathbf{a}}_y \cdot \hat{\mathbf{b}}_y \end{bmatrix} \quad (7)$$

Looking carefully, you may also notice something else: the *rows* of  $\mathbf{a}\text{-b:R}$  are the projection of the  $\mathbf{_A}$ -frame onto  $\mathbf{_B}$ ,

$$\mathbf{\hat{A}^B R} = [\mathbf{\hat{A}^B b_x \quad \hat{A}^B b_y}] = \begin{bmatrix} \hat{\mathbf{a}}_x \cdot \hat{\mathbf{b}}_x & \hat{\mathbf{a}}_x \cdot \hat{\mathbf{b}}_y \\ \hat{\mathbf{a}}_y \cdot \hat{\mathbf{b}}_x & \hat{\mathbf{a}}_y \cdot \hat{\mathbf{b}}_y \end{bmatrix} = \begin{bmatrix} \mathbf{B} \hat{\mathbf{a}}_x^T \\ \mathbf{B} \hat{\mathbf{a}}_y^T \end{bmatrix} \quad (8)$$

This relationship is interesting because it implies that *the rotation from A to B is equal to the transpose of the rotation of B to A*. In fact, because rotation matrices are *orthonormal* matrices (composed of orthogonal unit vectors), they have several useful properties that, for the sake of brevity, we summarize without proof.

Important properties of the rotation matrix, BAR:

1. The transpose is equal to its inverse.
2. The determinant is equal to +1 (assuming a right-handed coordinate system).
3. Columns (and rows) are mutually orthogonal unit vectors, therefore, the magnitude of any column (or row) is equal to one and the dot product of any two columns (or rows) is equal to zero
4. The columns define the basis vectors (i.e., x, y, z axes) of the rotated frame relative to the base frame

## Adding one more dimension

So far, we've only considered rotations in two dimensions but the same properties apply to three-dimensional rotations! In three dimensions, the rotation matrix looks like this:

$$\mathbf{\hat{A}^B R} = [\mathbf{\hat{A}^B b_x \quad \hat{A}^B b_y \quad \hat{A}^B b_z}] = \begin{bmatrix} \hat{\mathbf{a}}_x \cdot \hat{\mathbf{b}}_x & \hat{\mathbf{a}}_x \cdot \hat{\mathbf{b}}_y & \hat{\mathbf{a}}_x \cdot \hat{\mathbf{b}}_z \\ \hat{\mathbf{a}}_y \cdot \hat{\mathbf{b}}_x & \hat{\mathbf{a}}_y \cdot \hat{\mathbf{b}}_y & \hat{\mathbf{a}}_y \cdot \hat{\mathbf{b}}_z \\ \hat{\mathbf{a}}_z \cdot \hat{\mathbf{b}}_x & \hat{\mathbf{a}}_z \cdot \hat{\mathbf{b}}_y & \hat{\mathbf{a}}_z \cdot \hat{\mathbf{b}}_z \end{bmatrix}$$

Given the properties of rotation matrices listed above, the following matrices are valid.

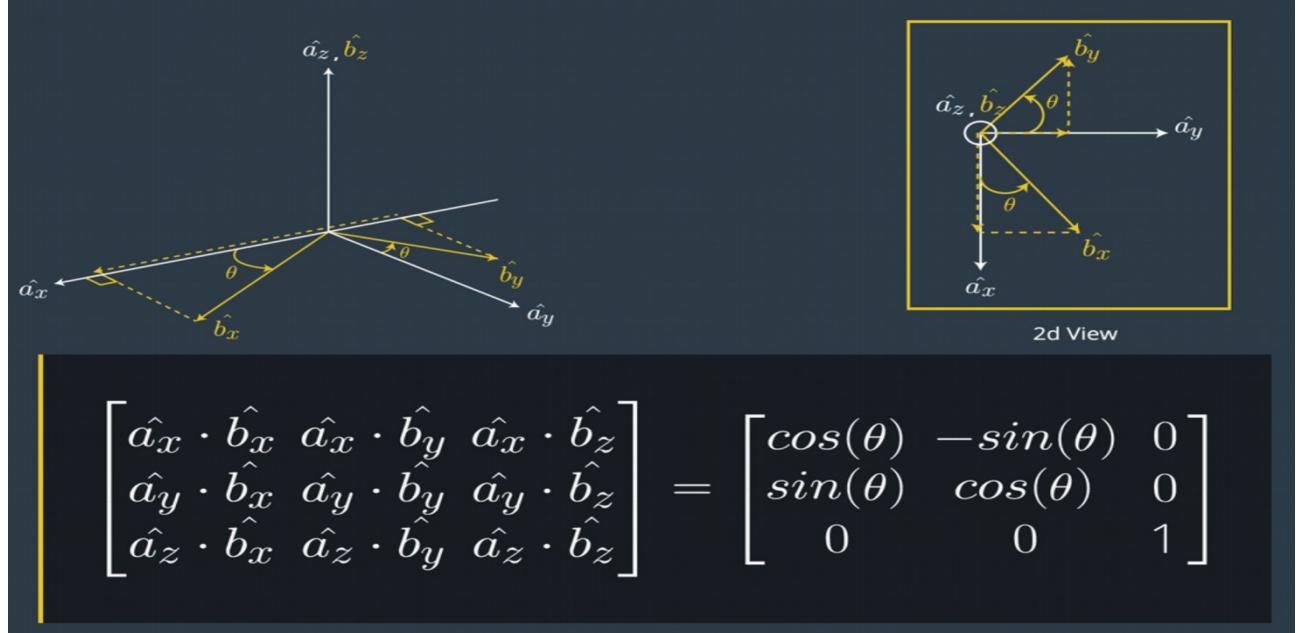
$$\begin{bmatrix} \frac{\sqrt{3}}{2} & 0 & -\frac{1}{2} \\ 0 & 1 & 0 \\ \frac{1}{2} & 0 & \frac{\sqrt{3}}{2} \end{bmatrix}$$

**Rotation Matrices in 3D example**

$$\begin{bmatrix} \frac{\sqrt{1}}{2} & \frac{\sqrt{3}}{2} & 0 \\ -\frac{\sqrt{3}}{2} & \frac{\sqrt{1}}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotations in 3D although more difficult to draw are fundamentally no different than the 2D case. The process still involves projecting the basis vectors of one frame onto those of another. In general, rotations can be performed about any arbitrary vector. However, when a rotation is performed about an axis of the coordinate frame, as it's usually the case, such a rotation is often referred to as elementary rotation.

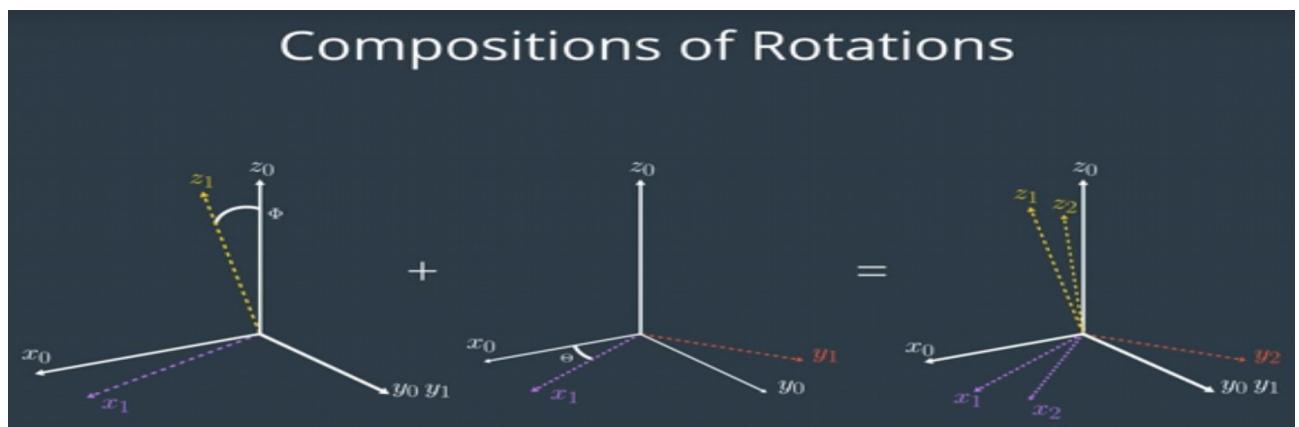
Let's consider two frames A and B that are initially aligned, frame B is then rotated about the  $\hat{a}_z$  axis by an angle theta, as we did on the 2D example to derive the relationship of the B frame with respect to the A frame, we project the basis vectors of frame B onto frame A as a reminder the equation between A and B look like the image below. The vectors below are unit vectors.



Now it's up to you derive the rotation matrices relative from another axes.

### 8.2.4 Composition of Rotations

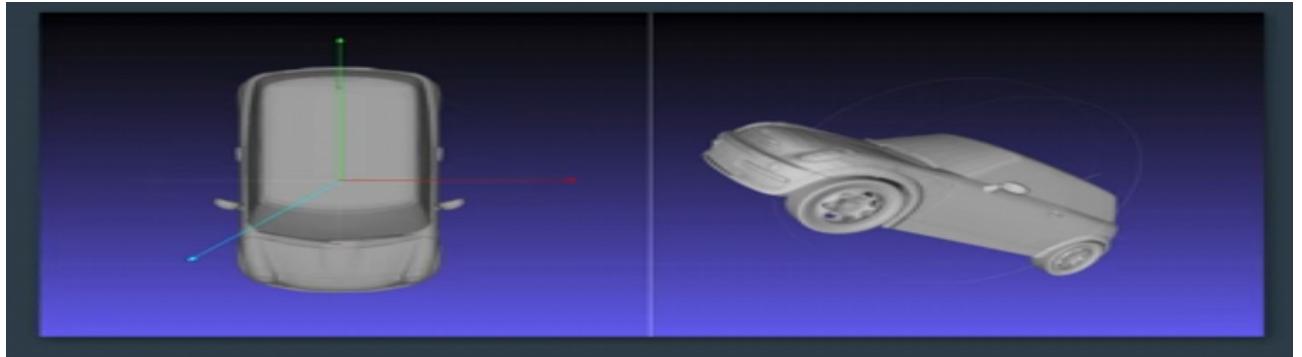
So far, we've seen single elementary rotation matrices in 2D and 3D. A logical extension then is to describe sequences of rotations a topic referred to as the composition of rotations.



The image above shows a sequence of two extrinsic rotations in 3D space. First is a rotation about the y-axis and then about the original z-axis, the resulting image on the right is the composition of these two rotations.

One system to describe a sequence of rotations is called Euler angles, named after the Swiss mathematician Leonard Euler. According to Euler's rotation theorem, the orientation of any rigid body with respect to some fixed reference frame can always be described by three elementary rotations in a given sequence.

For example to achieve the orientation of the image on the right,



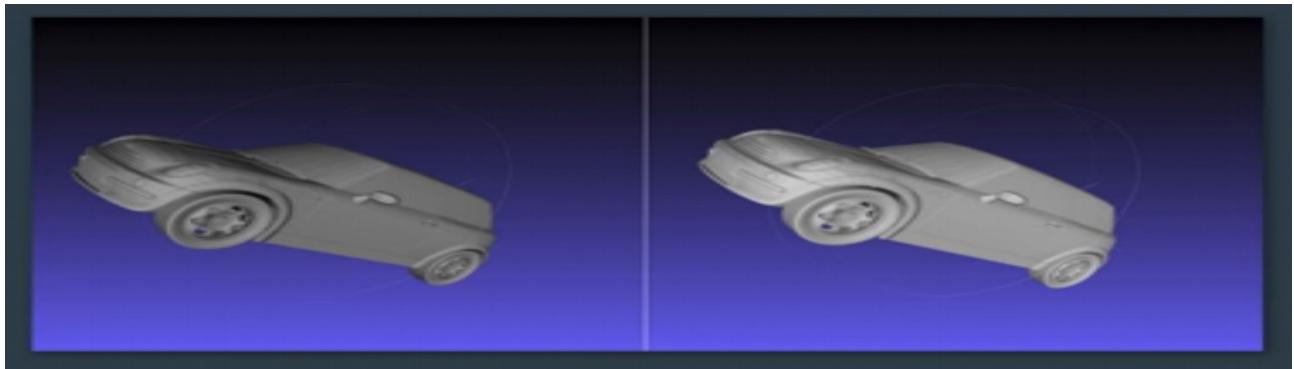
the object on the left can be rotated about a fixed x-axis



and then about the fixed z axis



and, finally, about the fixed y-axis



Before we dive into the mathematics behind compositional rotations, it's important to talk about several properties.

There are, in fact, numerous conventions regarding Euler angles, so you must be sure to identify which convention is being used. To completely identify a particular convention the following properties must be specified.

## Euler Angles

1. Tait-Bryan vs. Classic
2. Rotation Order
3. Intrinsic (body fixed) rotations vs. Extrinsic (fixed axis) rotations

Let's start off with Tait-Bryan versus classic convention.

## Euler Angles

1. Tait-Bryan vs. Classic

### Tait-Bryan

$x-y-z$	$y-x-z$	$z-y-x$
$x-z-y$	$y-z-x$	$z-x-y$

### Classic

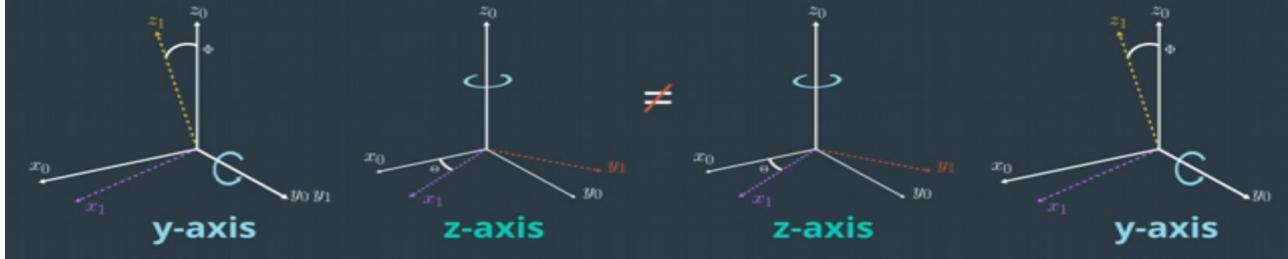
$x-y-x$	$y-x-y$	$z-y-z$
$x-z-x$	$y-z-y$	$z-x-z$

In the Tait-Bryan convention each elementary rotation is performed about a different Cartesian axis.

The **second** property that we must specify is rotation order, rotations are a **noncommutative** operation, thus the order in which rotations are performed matters. A rotation about the y-axis followed by a rotation about the z-axis will not generate the same result as rotating about the z-axis first followed by the y-axis. For this reason the order of rotations applied must be specified.

# Euler Angles

## 2. Rotation Order



Lastly, it's important to specify whether the rotation is extrinsic or intrinsic rotation.

# Euler Angles

## 3. Extrinsic vs. Intrinsic Rotations

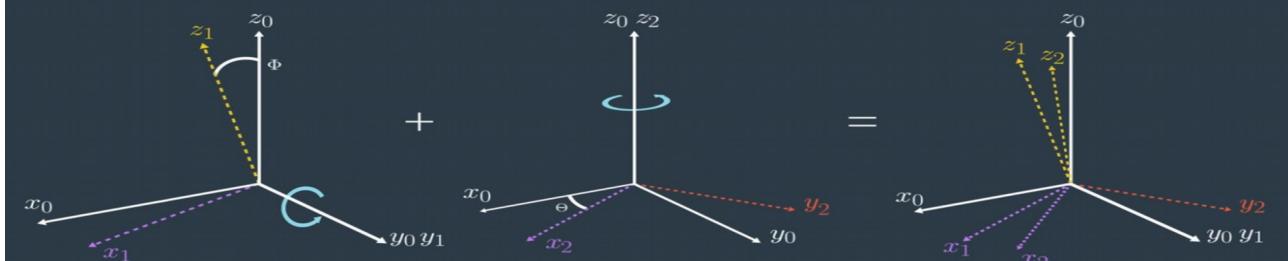
### Extrinsic

Extrinsic rotations are performed about the fixed world reference frame

### Intrinsic

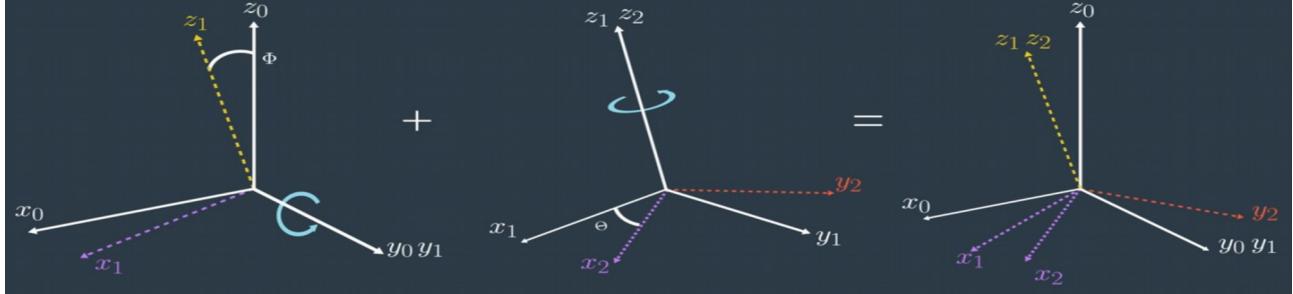
Intrinsic rotations are performed about the coordinate system as rotated by the previous operation

## Extrinsic Rotation



In this example of an extrinsic rotation, we can see the first rotation occurs about the  $y$ -axis, followed by a rotation about the  $z$ -axis of the fixed reference frame, not the new  $z$  axis.

# Intrinsic Rotation



In this example of an intrinsic rotation, the first rotation is about the y-axis, followed by a second rotation about the new z-axis.

Since Euler angles are a sequence of three rotations, let's consider the composition rule for three intrinsic rotations.

## Intrinsic Rotation

$$\frac{A}{B} R_{ZYX} = \frac{A}{B'} R \frac{B'}{B''} R \frac{B''}{B} R = R_Z(\alpha) R_Y(\beta) R_X(\gamma)$$

$$\frac{A}{B} R_{ZYX} = \begin{bmatrix} c\alpha & -s\alpha & 0 \\ s\alpha & c\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c\beta & 0 & s\beta \\ 0 & 1 & 0 \\ -s\beta & 0 & c\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\gamma & -s\gamma \\ 0 & s\gamma & c\gamma \end{bmatrix}$$

$$c\alpha = \cos(\alpha), s\alpha = \sin(\alpha)$$

The first rotation is about the z-axis by an angle alpha, then about the y-axis by an angle beta and finally about the x-axis by angle gamma. Mathematically we can express this in a very compact form with this notation which we will read it us ZYX rotation from frame A to B. Since this is an **intrinsic sequence** of rotations the elementary rotations are **post-multiplied**. Multiplying this out yields a somewhat complex-looking 3x3 matrix which we will see below.

## Extrinsic Rotation

$$\frac{A}{B} R_{ZYX} = R_X(\alpha) R_Y(\beta) R_Z(\gamma)$$

$$\frac{A}{B} R_{ZYX} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\gamma & -s\gamma \\ 0 & s\gamma & c\gamma \end{bmatrix} \begin{bmatrix} c\beta & 0 & s\beta \\ 0 & 1 & 0 \\ -s\beta & 0 & c\beta \end{bmatrix} \begin{bmatrix} c\alpha & -s\alpha & 0 \\ s\alpha & c\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

In the case of the same ZYX Euler angles set of extrinsic rotations, subsequent elementary rotations are **pre-multiplied**. Once again multiplying the matrices we get a very complex looking result.

# Intrinsic Rotations

$$\begin{matrix} A \\ B \end{matrix} R_{ZYX} = \begin{bmatrix} c\alpha c\beta & c\alpha s\beta s\gamma - s\alpha c\gamma & c\alpha s\beta c\gamma + s\alpha s\gamma \\ s\alpha c\beta & s\alpha s\beta s\gamma + c\alpha c\gamma & s\alpha s\beta c\gamma - c\alpha s\gamma \\ -s\beta & c\beta s\gamma & c\beta c\gamma \end{bmatrix}$$

# Extrinsic Rotations

$$\begin{matrix} A \\ B \end{matrix} R_{ZYX} = \begin{bmatrix} c\alpha c\beta & -s\alpha c\beta & s\beta \\ s\alpha c\gamma + s\beta s\gamma c\alpha & -s\alpha s\beta s\gamma + c\alpha c\gamma & -s\gamma c\beta \\ s\alpha s\gamma - s\beta c\alpha c\gamma & s\alpha s\beta c\gamma + s\gamma c\alpha & c\beta c\gamma \end{bmatrix}$$

Comparing the resultant rotation matrices for **intrinsic and extrinsic rotations**, we can see that the two are **not equal to each other**.

## Intrinsic Rotations

$$\begin{matrix} A \\ B \end{matrix} R_{ZYX} = R_Z(\gamma)R_Y(\beta)R_X(\alpha)$$

## Extrinsic Rotations

$$\begin{matrix} A \\ B \end{matrix} R_{ZYX} = R_X(\alpha)R_Y(\beta)R_Z(\gamma)$$

In fact this is a point worth repeating. **Rotations do not obey the commutative law of multiplication. That is, the order of matrix multiplication matters.**

The interested reader can explore the derivation of the composition rules [here](#).

Although Euler angles and rotation matrices are relatively intuitive, they do suffer from two significant drawbacks. First, are issues related to numerical performance. Defining the orientation of a rigid body in a three-dimensional environment only requires three generalized coordinates; however, rotation matrices contain nine elements. Clearly not all nine elements are independent and more memory is required to capture the orientation information than is required using a minimal set.

Rotation matrices are also not very numerically stable. Rounding errors associated with the repeated multiplication of rotation matrices also creates *numerical drift*. This means that over time the orthonormality conditions can be violated and the matrix will no longer be a valid rotation matrix. Also, it is a non-trivial task to interpolate smoothly between two rotation matrices (this is

particularly important for computer animations when trying to generate smooth motions between key frames).

The second significant drawback is *singularities of representation* (not to be confused with kinematic singularities). Singularities of representation occur when the second rotation in the sequence is such that the first and third coordinate frames become aligned causing a loss of a degree of freedom. This condition is often called "gimbal lock". In the case of the Z-Y-X rotation sequence used in the below equation, observe what happens to the overall rotation matrix when  $\beta=\pi/2$ .

$$\begin{aligned} {}^A_B R_{ZYX} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_\gamma & -s_\gamma \\ 0 & s_\gamma & c_\gamma \end{bmatrix} \begin{bmatrix} c_{\pi/2} & 0 & s_{\pi/2} \\ 0 & 1 & 0 \\ -s_{\pi/2} & 0 & c_{\pi/2} \end{bmatrix} \begin{bmatrix} c_\alpha & -s_\alpha & 0 \\ s_\alpha & c_\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ {}^A_B R_{ZYX} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_\gamma & -s_\gamma \\ 0 & s_\gamma & c_\gamma \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} c_\alpha & -s_\alpha & 0 \\ s_\alpha & c_\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ {}^A_B R_{ZYX} &= \begin{bmatrix} 0 & 0 & 1 \\ s_\gamma & c_\gamma & 0 \\ -c_\gamma & s_\gamma & 0 \end{bmatrix} \begin{bmatrix} c_\alpha & -s_\alpha & 0 \\ s_\alpha & c_\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ {}^A_B R_{ZYX} &= \begin{bmatrix} 0 & 0 & 1 \\ s_{\alpha+\gamma} & s_{\alpha+\gamma} & 0 \\ -c_{\alpha+\gamma} & s_{\alpha+\gamma} & 0 \end{bmatrix} \end{aligned}$$

Notice that no matter what the angles alpha and gamma may be, they have no effect on the X component. It is important to note that there is nothing wrong or physically inhibiting rotational motion, it is rather a mathematical deficiency in the choice of representing rotational motions. Further, ALL conventions using only three parameters to describe orientation suffer from singularities of representation. Therefore, a key consideration to using Euler angles in practice is to ensure that the range of motion of the object of interest does not come close to a singularity.

Just for fun:

- Video link explaining gimbal lock: [video link](#)
- Gyroscopes and Gimbal Lock [video link](#)
- Probably the most famous example of gimbal lock is from the [Apollo 11 Mission](#).

## 8.2.5 Euler Angles from a Rotation Matrix

In many cases, particularly involving inverse kinematics, we are given a composite rotation matrix and it is necessary to find a set of Euler Angles that would produce this rotation. The specific solution depends on the choice of Euler Angles but the basic procedure is the same. We illustrate the general procedure with an example.

Consider the *extrinsic* (i.e., fixed axis) X-Y-Z rotation sequence. The composite rotation matrix is,

$${}^A_B R_{XYZ} = R_Z(\alpha)R_Y(\beta)R_X(\gamma)$$

$${}^A_B R_{XYZ} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = \begin{bmatrix} c_\alpha c_\beta & c_\alpha s_\beta s_\gamma - s_\alpha c_\gamma & c_\alpha s_\beta c_\gamma + s_\alpha s_\gamma \\ s_\alpha c_\beta & s_\alpha s_\beta s_\gamma + c_\alpha c_\gamma & s_\alpha s_\beta c_\gamma - c_\alpha s_\gamma \\ -s_\beta & c_\beta s_\gamma & c_\beta c_\gamma \end{bmatrix}$$

The goal is to find the angles alpha, beta, and gamma, given that numerical values for rij are known. The solution is to use various combinations of rij so that each angle can be individually isolated and solved explicitly. Using the simplest terms possible seems to be a prudent strategy. Although beta appears in isolation in element r31, it is not a good idea to solve for angles using the inverse of the sine or cosine functions. The reason is the ambiguity in sign: if  $-\sin(\beta) = 0.5$ , in which quadrant is the angle? This type of ambiguity is avoided by using the [atan2](#) function. The syntax depends on the language or library used, but is often: atan2(y, x).

Thus, it is possible to find beta, by recognizing,

$$\beta = \text{atan2}(y, x) = \text{atan2}(-r_{31}, \sqrt{r_{11} * r_{11} + r_{21} * r_{21}})$$

A similar trick is used to find gamma,

$$\gamma = \text{atan2}(r_{32}, r_{33})$$

and also alpha,

$$\alpha = \text{atan2}(r_{21}, r_{11})$$

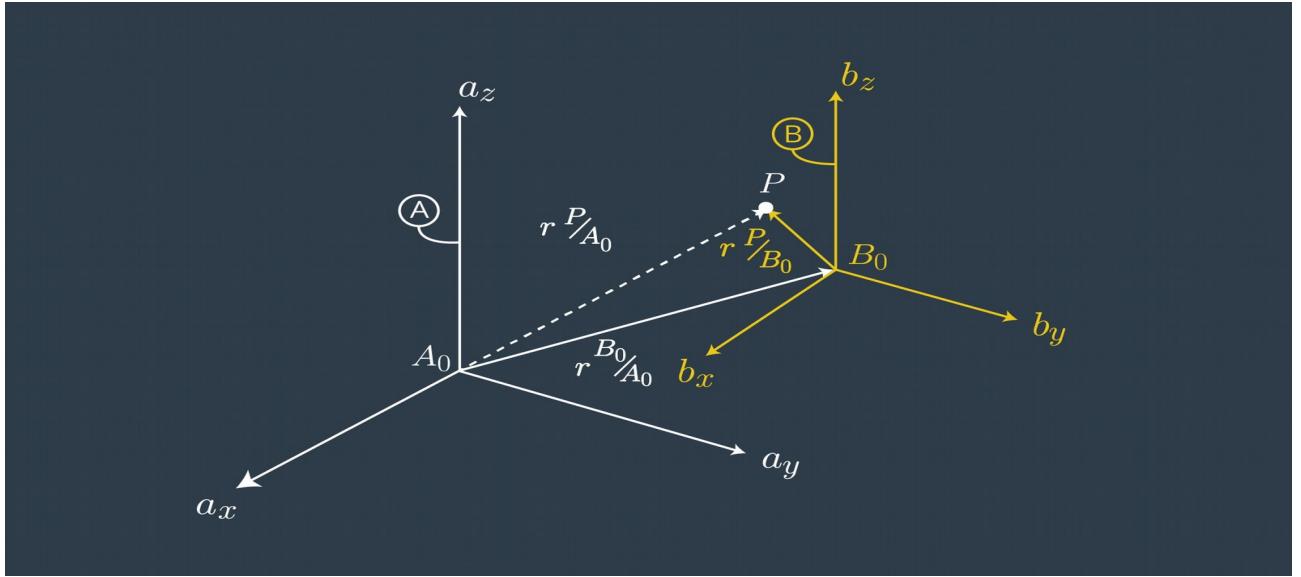
What happens then when  $\cos(\beta) = 0$ , that is, when  $\beta = +/-90$  degrees? At this point, atan2 is undefined and, as we saw with Euler Angles, the system exhibits a *singularity of representation*.

## 8.2.6 Translations

In comparison to rotating reference frames, translations are much simpler. Here we consider two reference frames, \_A\_ and \_B\_, that have the same orientation, but their origins, Ao and Bo, are no longer coincident. The position of point, \_P\_, relative to Bo is denoted by the vector BrP/Bo. The leading superscript is to denote that this vector is expressed in the \_B\_ frame. In other words,

$${}^B r_{P/B_0} = r_{B_x} \hat{b}_x + r_{B_y} \hat{b}_y + r_{B_z} \hat{b}_z$$

The goal then is to describe \_P\_ relative to Ao, ArP/Ao



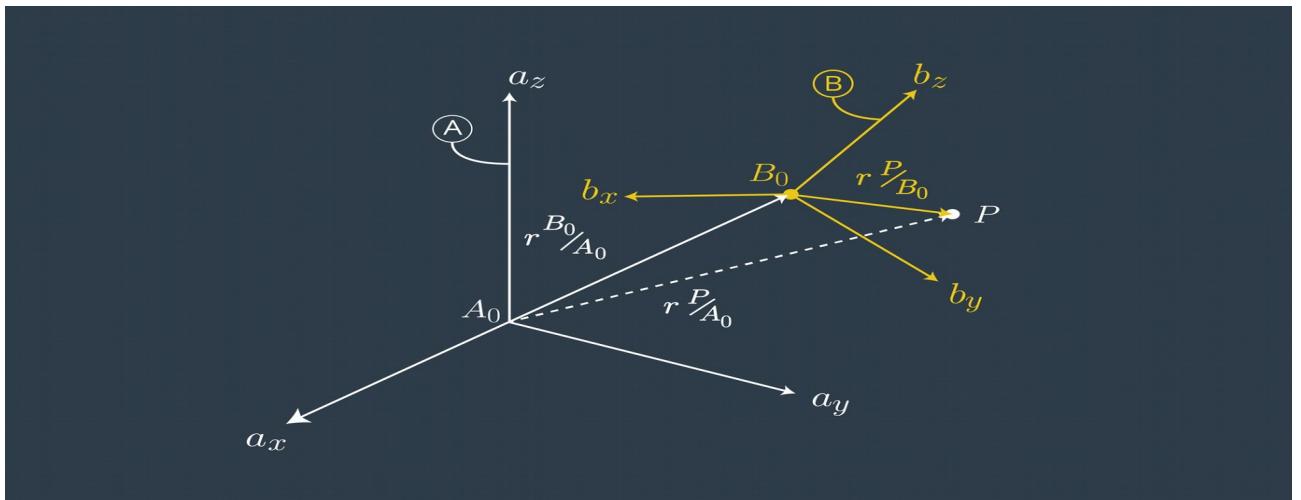
**Because both frames have the same relative orientation**, describing  $\underline{P}$  relative to Ao only requires simple vector addition:

$${}^A \mathbf{r}_{P/A_0} = {}^A \mathbf{r}_{B_0/A_0} + {}^B \mathbf{r}_{P/B_0}$$

where  $A_B/A_0$  is the origin of the  $B$  frame relative to the origin of the  $A$  frame. It is important to note that the location of point  $P$  has not changed in any way, we have simply described its position relative to a different frame of reference.

### 8.2.7 Homogeneous Transforms and their Inverse

Now we consider a more general case in which two reference frames are both simultaneously rotated and translated with respect to each other. Again we see a point,  $P$ , whose position is known relative to  $B_0$  and we wish to find  $A_P/A_0$ . In the case where the  $A$  and  $B$  frames both had the same orientation, the solution was easily achieved with vector addition. If only we knew how to rotate one frame relative to another we could solve this problem....



The solution is to first express BrP/Bo in terms of the  $A$  frame by applying a rotation matrix between  $B$  and  $A$  and then adding the offset of Bo relative to Ao. In equation form, this is:

$${}^A \mathbf{r}_{P/Ao} = {}_B^A R {}^B \mathbf{r}_{P/Bo} + {}^A \mathbf{r}_{Bo/Ao}$$

(1)

While the above equation is compact and easily read by humans, it doesn't lend itself to manipulation by computers. It is possible however to cast equation (1) in matrix form using homogeneous coordinates and then make use of powerful linear algebra libraries that are available in many programming languages.

$$\begin{bmatrix} {}^A \mathbf{r}_{P/Ao} \\ 1 \end{bmatrix} = \begin{bmatrix} {}_B^A R & {}^A \mathbf{r}_{Bo/Ao} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} {}^B \mathbf{r}_{P/Bo} \\ 1 \end{bmatrix}$$

Equation (2) is shorthand notation for equation (3) below.

The left hand side is a 4x1 vector, the first three elements of which are the x,y,z coordinate of point P expressed in the  $A$  frame.

The first term on the right hand side of the equality is a 4x4 matrix called a **homogeneous transform**; it is composed of four subparts.  $B \rightarrow A.R$  is the 3x3 rotation matrix.  $ArBo/Ao$  is a 3x1 vector and represents the origin of the  $B$  frame relative to the  $A$  frame, expressed in terms of the  $A$  frame.

Because of the restrictions on dimensions for matrix multiplication, the last term on the right hand side must also be a 4x1 vector where  $BrP/Bo$  is the location of  $P$  relative to  $Bo$  and expressed in terms of the  $B$  frame.

The notation for homogeneous transforms is similar to that of rotation matrices, a capital "T" (for transform) is used with leading super and subscripts. For example,  $B \rightarrow A.T$  is the homogeneous transform between  $A$  and  $B$ .

$$\begin{bmatrix} r_{Ax} \\ r_{Ay} \\ r_{Az} \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{BAx} \\ r_{21} & r_{22} & r_{23} & r_{BAy} \\ r_{31} & r_{32} & r_{33} & r_{BAz} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

(3)

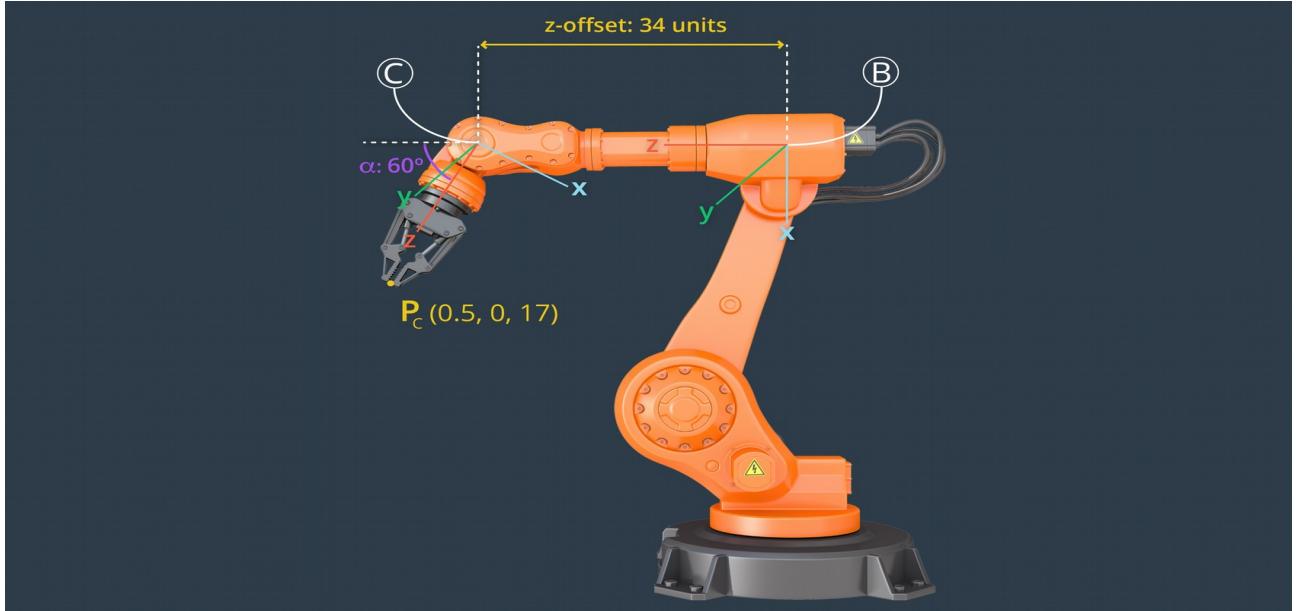
Upon multiplying the two matrices on the right, it becomes clear how the matrix form relates to our original equation (1).

$$\begin{bmatrix} r_{Ax} \\ r_{Ay} \\ r_{Az} \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11}x + r_{12}y + r_{13}z + r_{BAx} \\ r_{21}x + r_{22}y + r_{23}z + r_{BAy} \\ r_{31}x + r_{32}y + r_{33}z + r_{BAz} \\ 1 \end{bmatrix}$$

(4)

We can apply our newfound knowledge of homogeneous transformations to calculate the position of a robot arm's end effector. Given the initial position of point P with respect to reference frame C, below. What is the position of the point with respect to reference frame B, given the rotations and the translations in the image?

Recall that the offset for a translation is calculated from the perspective of the new frame, in our case reference frame B. Thus the translation is 34 units in the positive z-direction.



The first step is to setup our homogeneous transform matrix from to include a 60 degree rotation about the y-axis, and a 34 unit translation along the z-axis. Next, we multiply the homogeneous transform by a matrix that contains the position of point P (0.5, 0, 17).

$$\begin{bmatrix} r_{Ax} \\ r_{Ay} \\ r_{Az} \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(60) & 0 & \sin(60) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(60) & 0 & \cos(60) & 34 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0 \\ 17 \\ 1 \end{bmatrix}$$

Multiplying out the right-hand side of the equation, we get the following result.

$$\begin{bmatrix} r_{Ax} \\ r_{Ay} \\ r_{Az} \\ 1 \end{bmatrix} = \begin{bmatrix} 0.5\cos(60) + 17\sin(60) \\ 0 \\ -0.5\sin(60) + 17\cos(60) + 34 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} r_{Ax} \\ r_{Ay} \\ r_{Az} \\ 1 \end{bmatrix} = \begin{bmatrix} 15 \\ 0 \\ 42 \\ 1 \end{bmatrix}$$

The position of point P with respect to reference frame B, is then coarsely rounded to (15, 0, 42).

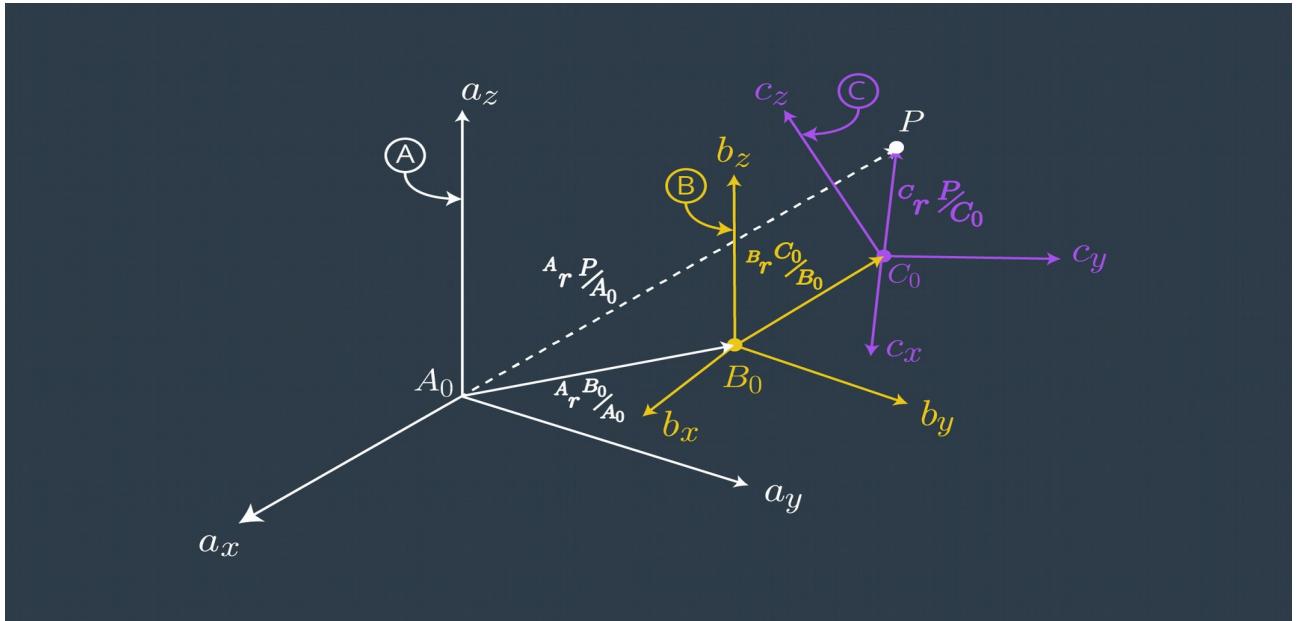
Now that we have seen how to compute a transform from an initial frame  $\{A\}$  to another frame  $\{B\}$ , let's consider the reverse operation: finding the transform from  $\{B\}$  to  $\{A\}$ . The most basic method for inverting the transform is simply to invert the  $4 \times 4$  matrix. However, given the orthonormal properties of rotation matrices it is possible to perform the inverse in a simpler, more computationally efficient form. We do not show the proof here, but the inverse transform can be written as:

One final note, which should hopefully come as no surprise,

$$\begin{matrix} {}^A_B T^{-1} = \left[ \begin{array}{ccc|c} {}^A_B R^T & -{}^A_B R^T {}^A_B r_{B_o/A_o} & & \\ \hline 0 & 0 & 0 & 1 \end{array} \right] \\ {}^A_B T {}^A_B T^{-1} = I_{4 \times 4} \end{matrix}$$

### 8.2.8 Composition of Homogeneous Transforms

The composition of homogeneous transforms follows much the same logic as composition of rotations.



Assume the transform from frame  $\{C\}$  relative to frame  $\{B\}$  is known, as is the transform from  $\{B\}$  to  $\{A\}$ . It is possible then to express CrP/Co in terms of frame  $\{A\}$  by first transforming it to the frame  $\{B\}$ ,

$$\overset{B}{r}P_{B_0} = \overset{B}{C} T \overset{C}{r}P_{C_0}$$

(1)

and then by transforming to the  $\{A\}$  frame,

$${}^A \mathbf{r}_{P/Ao} = {}^A_B T {}^B \mathbf{r}_{P/Bo}$$

(2)

Perhaps not surprisingly, equations (1) and (2) can be combined as,

$${}^A \mathbf{r}_{P/Ao} = {}^A_B T {}^B_C T {}^C \mathbf{r}_{P/Co} = {}^A_C T {}^C \mathbf{r}_{P/Co}$$

For completeness, let's expand the transform between  $_A$  and  $_C$  in terms it's known components,

$$\begin{matrix} {}^A_C T = \end{matrix} \left[ \begin{array}{ccc|c} {}^A_B R & {}^A \mathbf{r}_{B_o/A_o} & & \\ \hline 0 & 0 & 0 & 1 \end{array} \right] \left[ \begin{array}{ccc|c} {}^B_C R & {}^B \mathbf{r}_{C_o/B_o} & & \\ \hline 0 & 0 & 0 & 1 \end{array} \right]$$

(3)

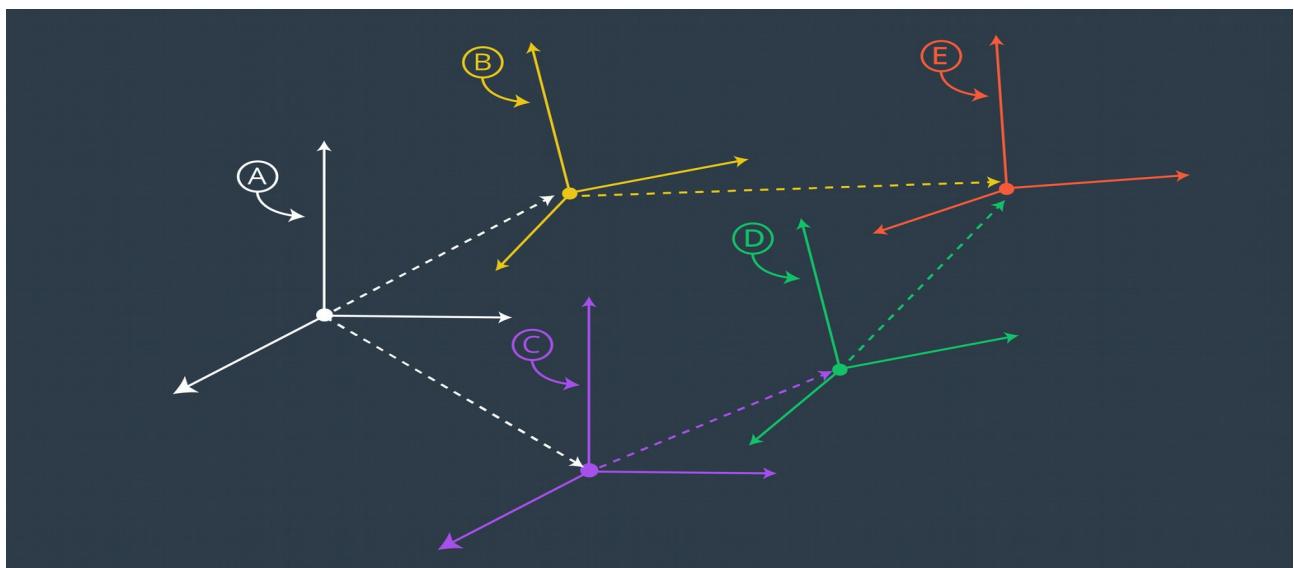
$$\begin{matrix} {}^A_C T = \end{matrix} \left[ \begin{array}{ccc|c} {}^A_B R {}^B_C R & {}^A_B R {}^B \mathbf{r}_{C_o/B_o} + {}^A \mathbf{r}_{B_o/A_o} & & \\ \hline 0 & 0 & 0 & 1 \end{array} \right]$$

(4)

The upper left 3x3 submatrix in equation (4) should look familiar; it is a composition of rotations between  $_A$  and  $_C$ .

$B-A.R$  BrCo/Bo is the vector from Bo to Co expressed in the  $_A$  frame. The final term, ArBo/Ao, is the vector from Ao to Bo expressed in  $_A$ .

To help solidify the concepts of transforms, inverse transforms, and composition of transforms, consider one more example. Here we have frames  $_A$  through  $_E$ .



Let's say we wish to create the transform between frames  $_A$  and  $_E$ ; clearly, there are two paths. One route has transforms,

$$\begin{matrix} {}^A_E T = {}^A_B T {}^B_E T \end{matrix}$$

(5)

while the second is,

$$\begin{matrix} {}^A_E T = {}^A_C T {}^C_D T {}^D_E T \end{matrix}$$

(6)

Equating (5) and (6) yields,

$$\begin{matrix} {}^A_B T {}^B_E T = {}^A_C T {}^C_D T {}^D_E T \end{matrix}$$

(7)

The following steps are taken to obtain coordinate frame E from frame A.

From Frame A to B to E:

- Frame A: Located at [0, 0, 0]
- Frame B: Rotate Frame A about  $a_y$  by -90 degrees. Translate A by [-2, 2, 4]
- Frame E: Rotate Frame B about  $b_x$  by 90 degrees. Translate B by [0, 2, 0]

From Frame A to C to D to E:

- Frame C: Translate A by [4, 4, 0]
- Frame D: Rotate Frame C about  $c_x$  by 90 degrees. Translate C by [-3, 3, 2]
- Frame E: Rotate Frame D about  $d_z$  by 90 degrees. Translate D by [-3, 2, 3]

Now what if we needed to explicitly solve for the transform  $C \rightarrow DT$ ? All that is required is to make use of inverse transforms, applied in the proper order. First, premultiply both sides by  $A \rightarrow C.T^{-1} = C \rightarrow A.T$  and then post-multiply both sides by  $D \rightarrow E.T^{-1} = E \rightarrow D.T$ .

## 8.2.9 Denavit-Hartenberg Parameters

Previously, the concepts of rotations, translations, and homogenous transforms were introduced. All of these concepts are essential to understanding the *forward kinematics* problem of manipulators, that is, given the joint variables, calculate the location of the end effector. The solution procedure involves attaching a reference frame to each link of the manipulator and writing the homogeneous transforms from the fixed base link to link 1, link 1 to link 2, and so forth, all the way to the end effector.

$${}^N_T = {}^0_T {}^1_T {}^2_T \dots {}^{N-1}_N T$$

(1)

In general, *each* transform would require **six independent parameters** to describe frame  $i$  relative to  $i-1$ , three for position and three for orientation.

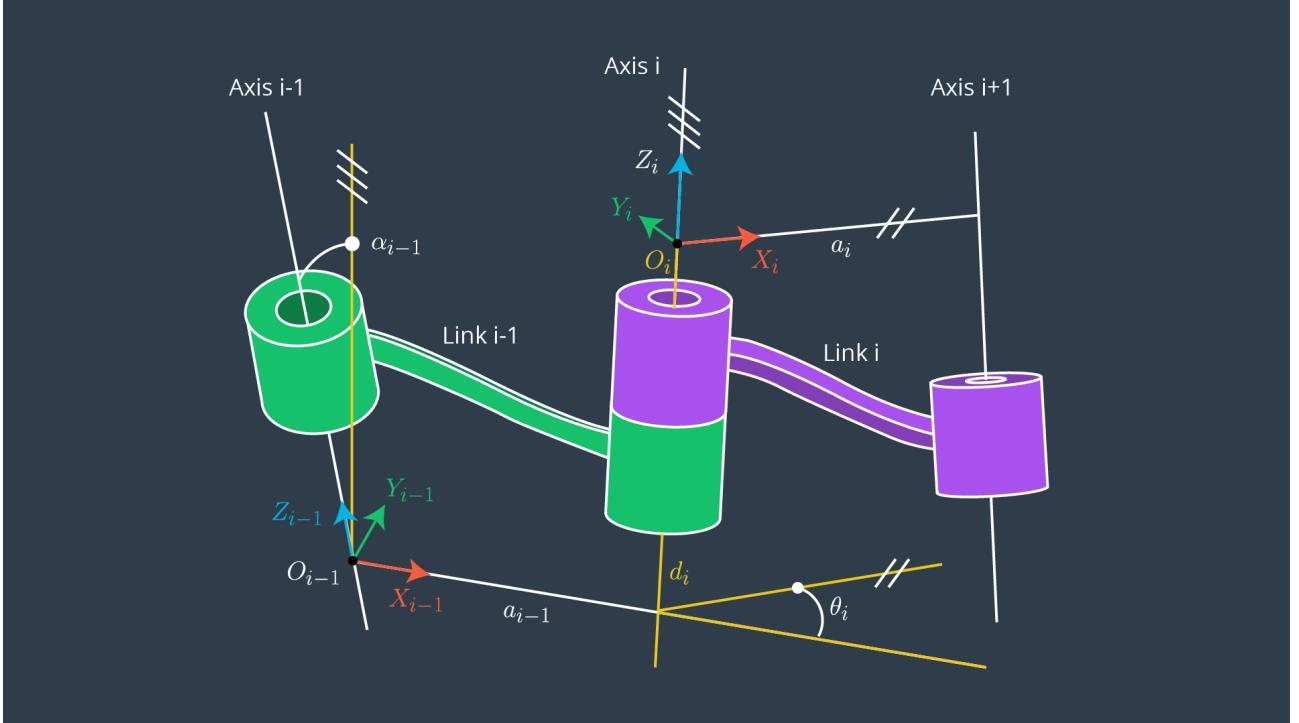
In 1955, Jacques Denavit and Richard Hartenberg proposed a systematic method of attaching reference frames to the links of a manipulator that simplified the homogeneous transforms. Their method only requires **four parameters** to describe the position and orientation of neighboring reference frames. In the early days of robotics, the more compact forward kinematics equations provided substantial benefit as calculations were typically done by hand or on computers with limited processing power. As a result, the Denavit-Hartenberg (DH) method for describing manipulator kinematics is now ubiquitous.

Since its original description, several modifications have been made to the DH method, mostly regarding the numbering and location of each reference frame's origin, so care must be taken to identify which convention is being used when comparing work from different sources. The five most common sources are listed below,

- Waldron, KJ. A study of overconstrained linkage geometry by solution of closure equations, Part I: A method of study (1973). **Mech. Mach. Theory** 8(1):95-104.
- Paul, R. (1982). Robot Manipulators: Mathematics, Programming and Control (MIT Press, Cambridge, MA)
- Craig, JJ. (2005). Introduction to Robotics: Mechanics and Control, 3rd Ed (Pearson Education, Inc., NJ)
- Khalil, W and Dombre, E. (2002). Modeling, Identification and Control of Robots (Taylor Francis, NY)
- M. Spong and M. Vidyasagar, Robot Modeling and Control, Wiley, 2005

Here, we will be using the convention described in John J Craig's book.

These differences in convention can make the comparison of results more difficult. It is important to always check how the sequence of homogeneous transforms is performed to relate neighboring links. The convention described below is consistent with Craig, (2005). The parameters involved,  **$\alpha$ ,  $a$ ,  $d$ , and  $\theta$** , are best understood with the aid of a figure.

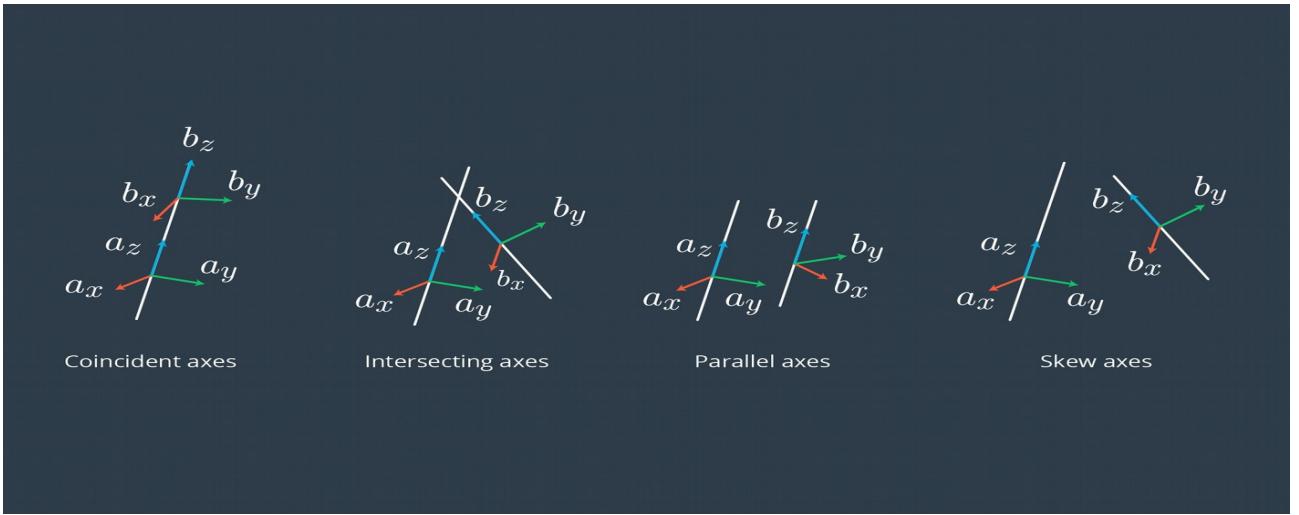


The parameter names and definitions are summarized as follows:

- $\alpha_{i-1}$  (twist angle) = angle between  $Z^{i-1}$  and  $Z^i$  measured about  $X^{i-1}$  in a right-hand sense.
- $a_{i-1}$  (link length) = distance from  $Z^{i-1}$  to  $Z^i$  measured along  $X^{i-1}$  where  $X^{i-1}$  is perpendicular to both  $Z^{i-1}$  to  $Z^i$
- $d_i$  (link offset) = signed distance from  $X^{i-1}$  to  $X^i$  measured along  $Z^i$ . Note that this quantity will be a variable in the case of prismatic joints.
- $\theta_i$  (joint angle) = angle between  $X^{i-1}$  to  $X^i$  measured about  $Z^i$  in a right-hand sense.  
Note that this quantity will be a variable in the case of a revolute joint.

For an  $n$ -degree of freedom manipulator, there will be  $n$ -joints  $\{1, 2, \dots, n\}$  and, because every joint connects two links, there are  $n+1$  links  $\{0, 1, \dots, n\}$ . By convention, the fixed base link is link 0 and the index,  $i$ , increases sequentially towards the end effector. Notice, as in the figure above, there is no requirement for the origin of the frame to be physically on the link, only that it move rigidly with the link. The Z-axis of the reference frame is, in the case of a revolute joint, aligned with joint's axis of rotation. Therefore, link  $i$ , rotates about  $Z^i$  and relative to link  $i-1$ . For prismatic joints, the Z-axis is aligned with its direction of motion.

$X^i$  represents a unit vector that is mutually perpendicular (or “normal”) to both  $Z^{i-1}$  to  $Z^i$ . The origin of a frame  $i$  is identified as the intersection of the common normal with  $Z^i$ . The parameter,  $a_{i-1}$ , measures the distance between the two Z-axes (not technically the “link length” as its name implies). In regards to the orientation of  $Z^i$  relative to  $Z^{i-1}$ , there are four cases to consider.



A unit vector that is mutually perpendicular to two vectors is defined as,

$$X_{i-1} = \pm \frac{Z_{i-1} \times Z_i}{\|Z_{i-1} \times Z_i\|}$$

(2)

i.e., the [cross product](#) divided by the magnitude of the cross product. Only in the case of skew axes is  $X^{i-1}$  unique. For intersecting axes, the choice of sign is arbitrary. The conventional wisdom is to choose the direction of  $X^{i-1}$  such that a positive rotation of link  $i$  is "intuitively satisfying". For parallel axes, there are infinitely many possibilities. In this case, you should look opportunities to place the origin of the reference frame where it can make some other parameters equal to zero. Finally, the Y-axis is chosen to complete a right-handed coordinate system.

The homogeneous transform from frame  $i-1$  to frame  $i$  is constructed as a sequence of four basic transformations, two rotations and two translations as follows:

$${}^{i-1}_iT = R_X(\alpha_{i-1}) D_X(a_{i-1}) R_Z(\theta_i) D_Z(d_i)$$

(3)

$${}^{i-1}_iT = \begin{bmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ s\theta_i c\alpha_{i-1} & c\theta_i c\alpha_{i-1} & -s\alpha_{i-1} & -s\alpha_{i-1} d_i \\ s\theta_i s\alpha_{i-1} & c\theta_i s\alpha_{i-1} & c\alpha_{i-1} & c\alpha_{i-1} d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

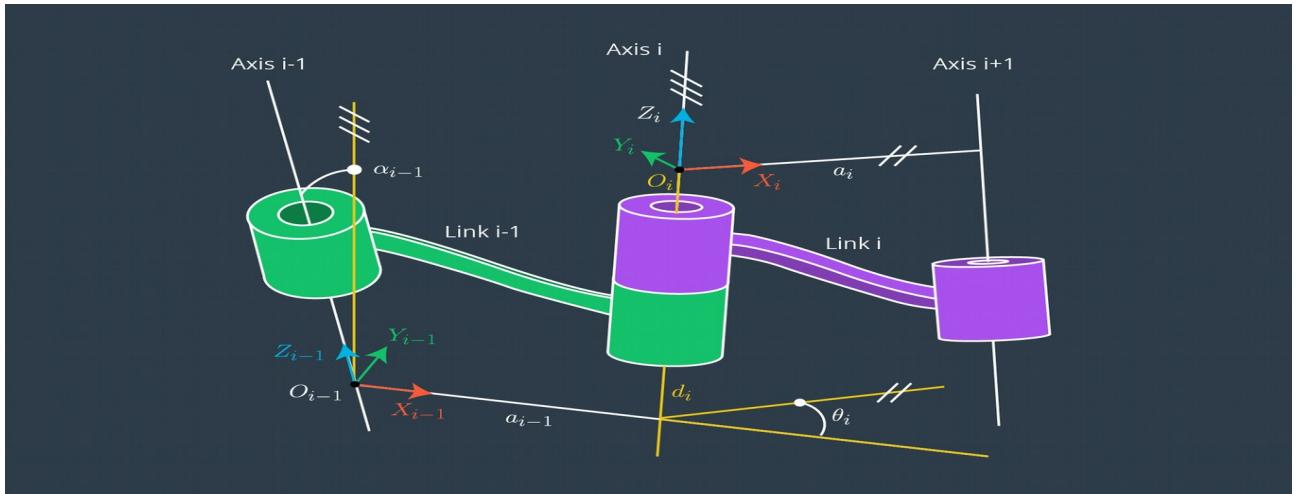
(4)

Equations (1) and (3) are very important to understand. When learning to write the DH parameters for a new manipulator, newcomers often struggle with where to place reference frames, how to orient them, and keeping track of the indices. A question we should always ask ourselves is: "Does the

transform move the reference frame in link  $i-1$  to be exactly coincident with the reference frame in link  $i$ ?" If the answer is yes, that is a good sign! It is also important to note that in equation (3), there is only one parameter that is variable (either  $\theta_i$  or  $d_i$ ), the link length and twist angle are constants. Thus, for the entire transform from base link to end effector,  $0 \rightarrow n.T$ , there are only  $n$ -variables.

### 8.2.10 DH Parameter Assignment Algorithm

The last section attempted to explain the theory and motivation behind DH parameters. Now the focus is on how to algorithmically assign reference frames to the manipulator's links. Keep in mind that frame  $i$  is rigidly attached to link  $i$ . For convenience, the supporting figure is repeated here.



The parameter assignment process for open kinematic chains with  $n$  degrees of freedom (i.e., joints) is summarized as:

1. Label all joints from  $\{1, 2, \dots, n\}$ .
2. Label all links from  $\{0, 1, \dots, n\}$  starting with the fixed base link as 0.
3. Draw lines through all joints, defining the joint axes.
4. Assign the Z-axis of each frame to point along its joint axis.
5. Identify the common normal between each frame  $Z^{\wedge}i-1$  and frame  $Z^{\wedge}i$ .
6. The endpoints of "intermediate links" (i.e., not the base link or the end effector) are associated with two joint axes,  $\{i\}$  and  $\{i+1\}$ . For  $i$  from 1 to  $n-1$ , assign the  $X^{\wedge}i$  to be ...
  - For skew axes, along the normal between  $Z^{\wedge}i$  and  $Z^{\wedge}i+1$  and pointing from  $\{i\}$  to  $\{i+1\}$ .
  - For intersecting axes, normal to the plane containing  $Z^{\wedge}i$  and  $Z^{\wedge}i+1$ .
  - For parallel or coincident axes, the assignment is arbitrary; look for ways to make other DH parameters equal to zero.
7. For the base link, always choose frame  $\{0\}$  to be coincident with frame  $\{1\}$  when the first joint variable ( $\theta_1$  or  $d_1$ ) is equal to zero. This will guarantee that  $\alpha_0 = a_0 = 0$ , and, if joint 1 is a revolute,  $d_1 = 0$ . If joint 1 is prismatic, then  $\theta_1 = 0$ .

- For the end effector frame, if joint  $n$  is revolute, choose  $X_n$  to be in the direction of  $X_{n-1}$  when  $\theta_n = 0$  and the origin of frame  $\{n\}$  such that  $d_n = 0$ .

Once the frame assignments are made, the DH parameters are typically presented in tabular form (below). Each row in the table corresponds to the transform from frame  $\{i\}$  to frame  $\{i+1\}$ .

$i$	$\alpha_{i-1}$	$a_{i-1}$	$d_i$	$\theta_i$
$T_1^0$				
$\vdots$				
$T_i^{i-1}$				
$\vdots$				
$T_n^{n-1}$				

### DH Parameters table

Here, we summarize some of the DH parameter simplifications to look for when choosing frame assignments.

- Special cases involving the  $Z^{i-1}$  to  $Z^i$  axes:
  - collinear lines:  $\alpha = 0$  and  $a = 0$
  - parallel lines:  $\alpha = 0$  and  $a \neq 0$
  - intersecting lines:  $\alpha \neq 0$  and  $a = 0$
  - If the common normal intersects  $Z^i$  at the origin of frame  $i$ , then  $d_i$  is zero

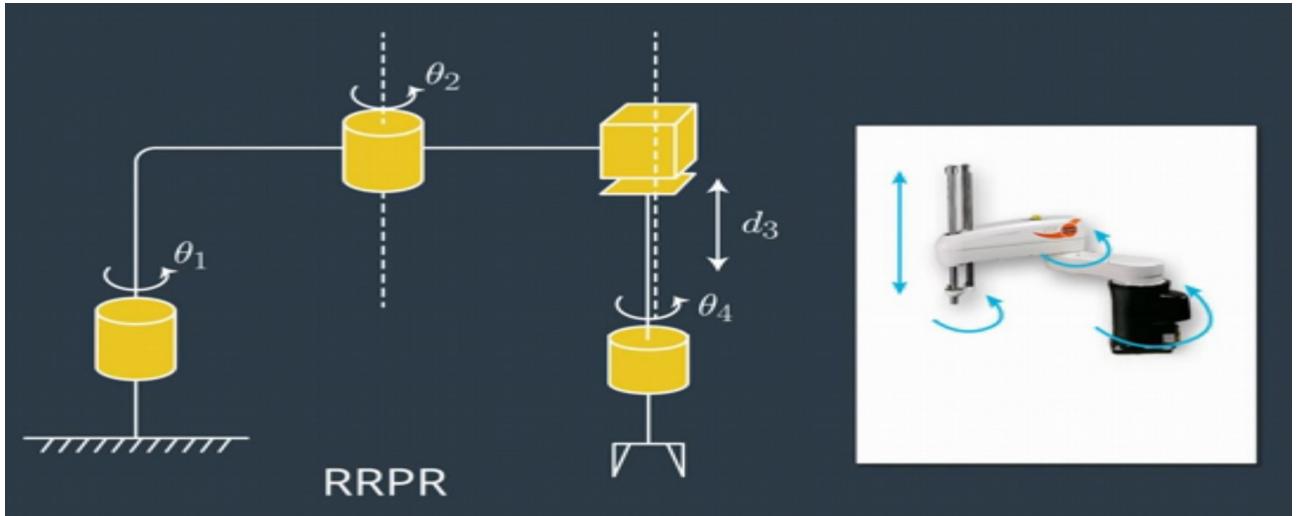
We mentioned previously that various authors have proposed multiple conventions for assigning DH parameters. Even if two analysts use the same convention, there is no guarantee that it will result in an identical assignment of frames to links. However, if the same base frame and same point is chosen as the origin of frame  $n$ , the overall transform from 0 to  $n$  would be the same regardless of what convention was used.

## 8.2.11 DH Steps and Parameter Table

DH parameters are a topic that most find confusing at first and although there is an algorithm to assign frames, we will find there is still a fair amount of free choices to be made. It takes a bit of practice to master.

Here we're gonna show how to derive the DH parameters and table for one common industrial robot called the SCARA.

Here is a schematic of the KUKA KR10 manipulator. SCARA's are characterized by having a revolute\_revolute\_prismatic\_revolute or RRPR structure, with all joint axes being parallel.

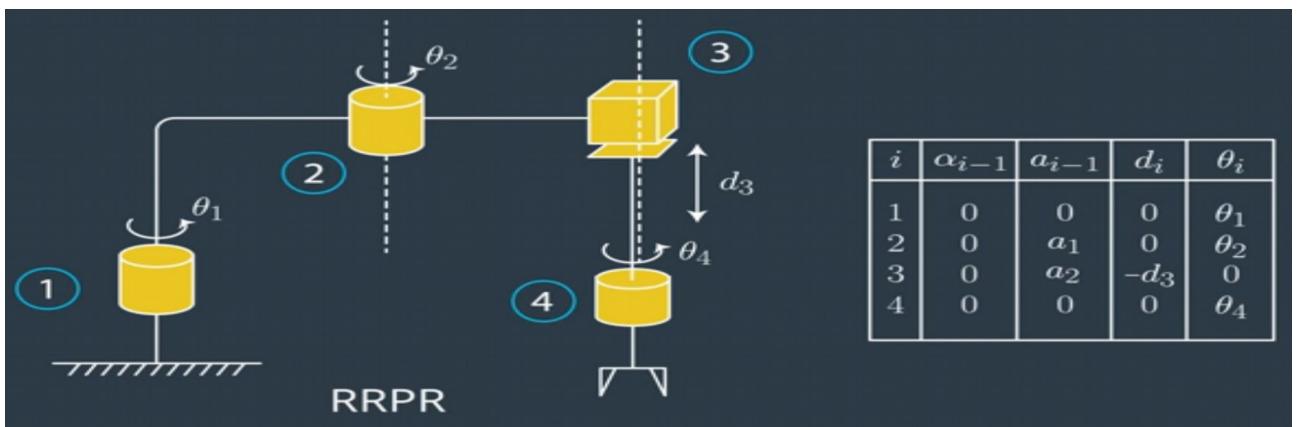


These manipulators are ideally suited for pick and place and fine assembly type tasks.

One thing that will make the process much easier is to always draw the schematic in a configuration such that all the revolute joints have zero angle and all prismatic joints have zero displacement.

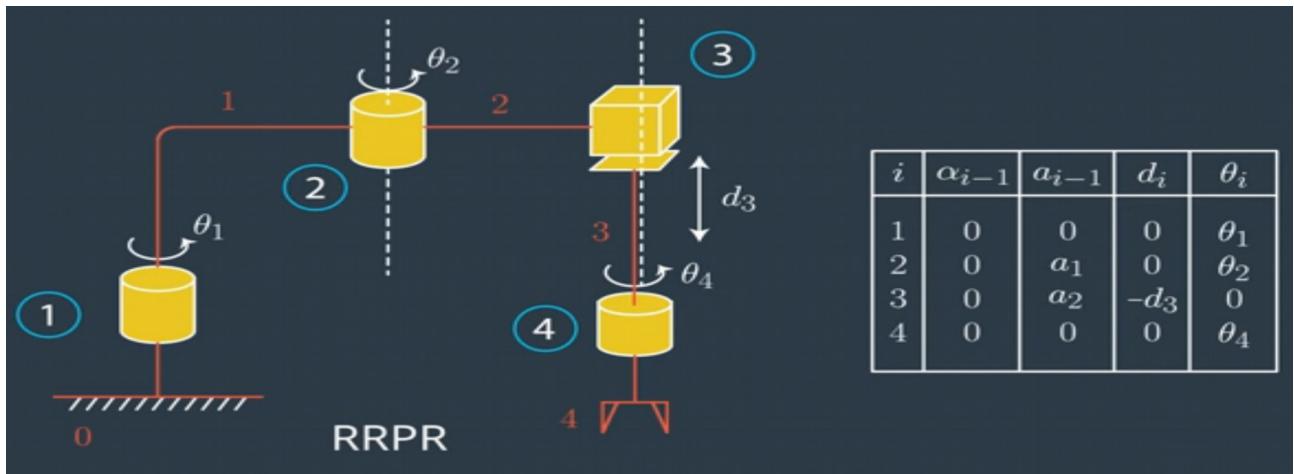
### Step 1:

Is to label all the joints from 1 to n. Here we go!



### Step 2:

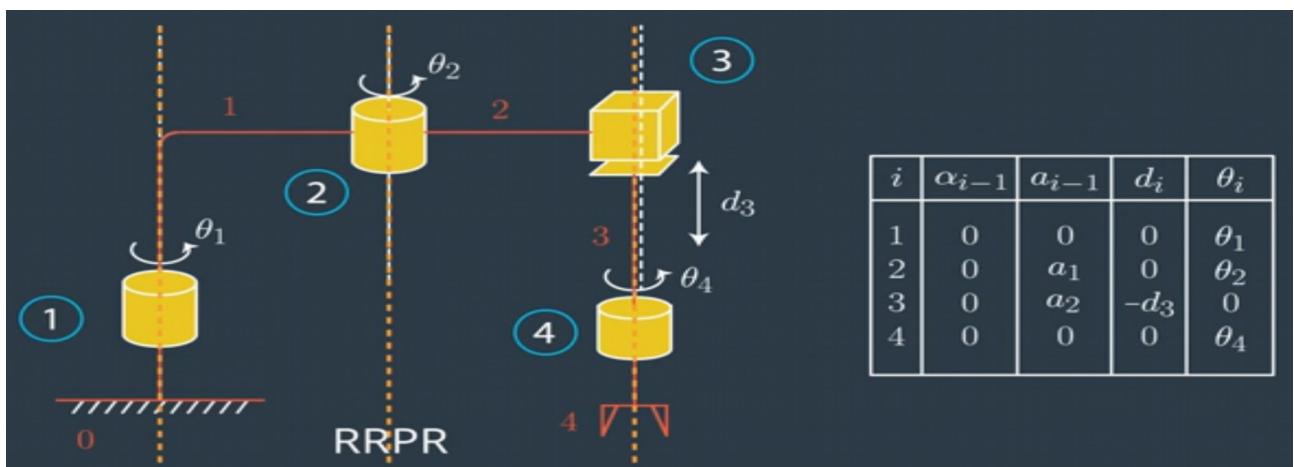
Is to label all the links 0 to n



The fixed base link is always link 0. Notice that links do not have to be straight line segments, they can be curved or angled rigid bodies like link 1. End Effector is link 4.

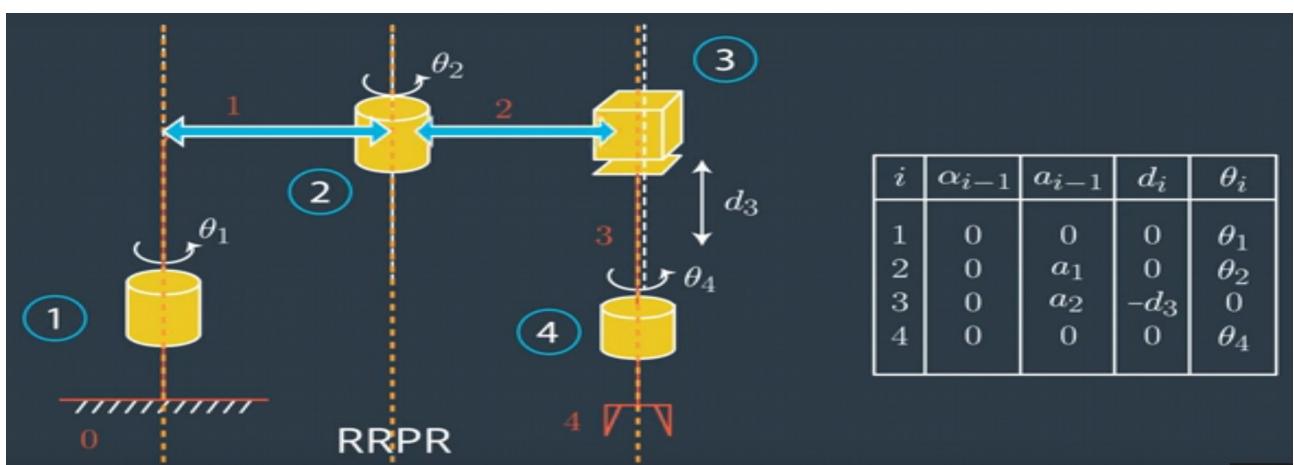
### Step 3:

Is to draw lines defining all joint axes.



The dotted lines represent the joint axes and to be clear the axes of joint 3 and 4 are coincident.

### Step 4:

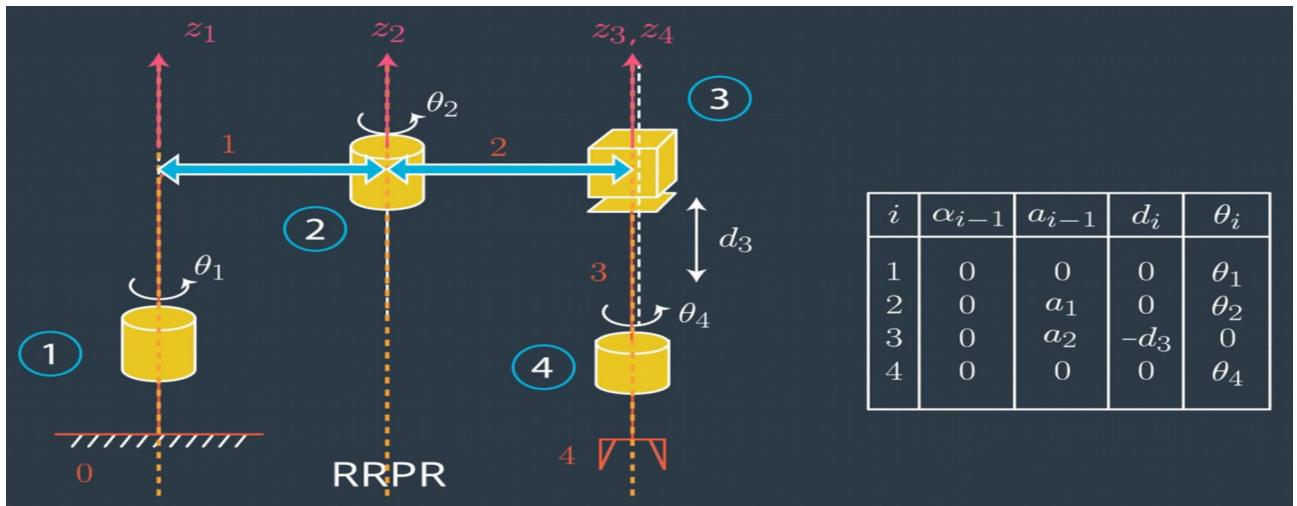


Is to define the common normal's between joint axes, we can see that all the joint axes are in fact parallel and joints 3 and 4 are coincident. So there are in fact, infinitely many possibilities between each joint pair. As always whenever there is freedom to choose, look for ways to minimize the number of non-zero DH parameters. Above is our choice for the common normal between joints 1 and 2 and for joints 2 and 3.

So what about links 0 and 1 and links 3 and 4? Both of these are special cases which we will discuss shortly, for now let's move on to step 5.

### Step 5:

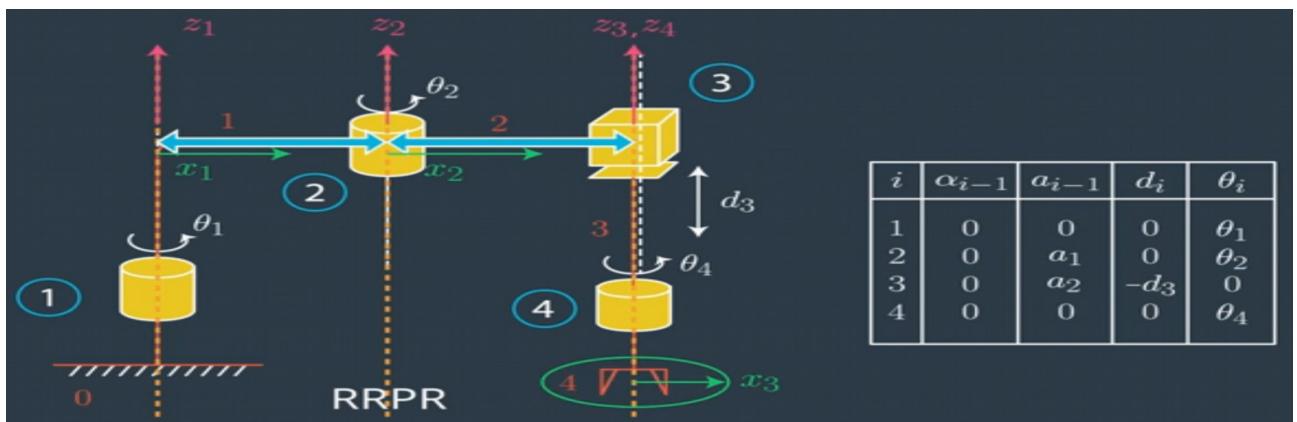
Assigning the Z-axis of frame i to point along the ith joint axis, the line of action for each joint has already been defined, but now we should decide whether the positive z-axis should point up or down, we should choose what it is more intuitive. In many applications, it's conventional to define positive rotations to be counter-clockwise when viewed from above.



So to stick with this convention we choose Z1 to be up, the same logic applies to Z2. Joint 3 is prismatic it feels natural to think of positive displacement as up so we choose Z3 up, and if we assign Z4 as up as we will see the twist angle between Z3 and Z4 will be 0. This is an example of practice and experience being helpful.

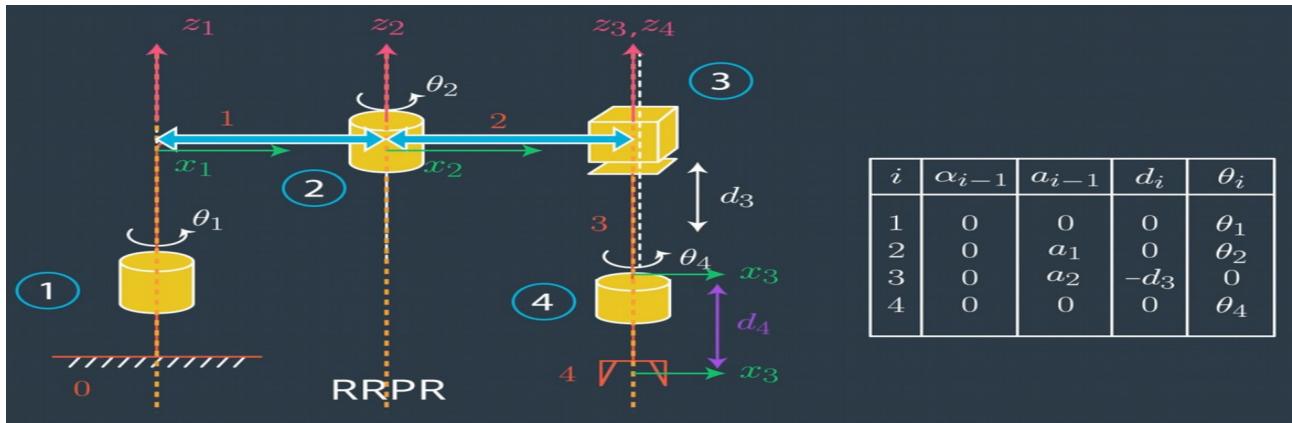
### Step 6:

Is to define the positive x axis for all the intermediate lengths, that is excluding the base and end effector. For z axis that are parallel, the positive x axis should point along the common normal from  $Z_{i-1}$  to  $Z_i$ .



So here is X1, X2 and X3. Technically, X3 can be oriented in any direction that is perpendicular to both Z3 and Z4, we could think of this as a circular ring around the Z axes but again, experience tells us that if I can make things parallel in this case, X3 parallel to X2, it usually makes more DH parameters 0.

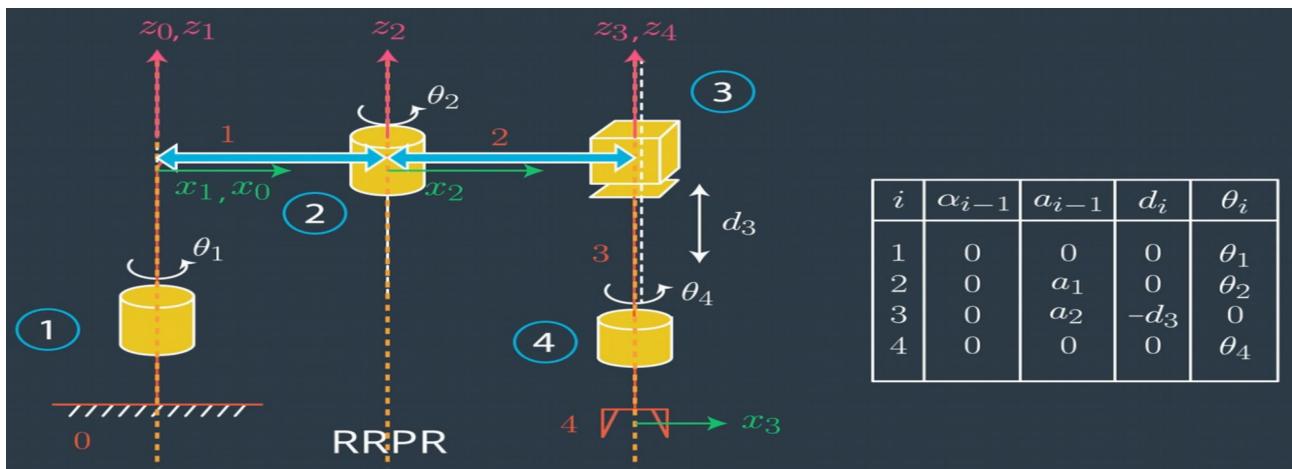
But why did we put X3 down there? Well, we need to capture the translation of the prismatic joint but any point along the Z3 axis would suffice. We're actually thinking one step ahead, what we care about is the location of the gripper. If instead we were to place X3 here,



then for the transform between frames 3 and 4, we would need an additional **constant term d4 and we don't want that**.

### Step 7:

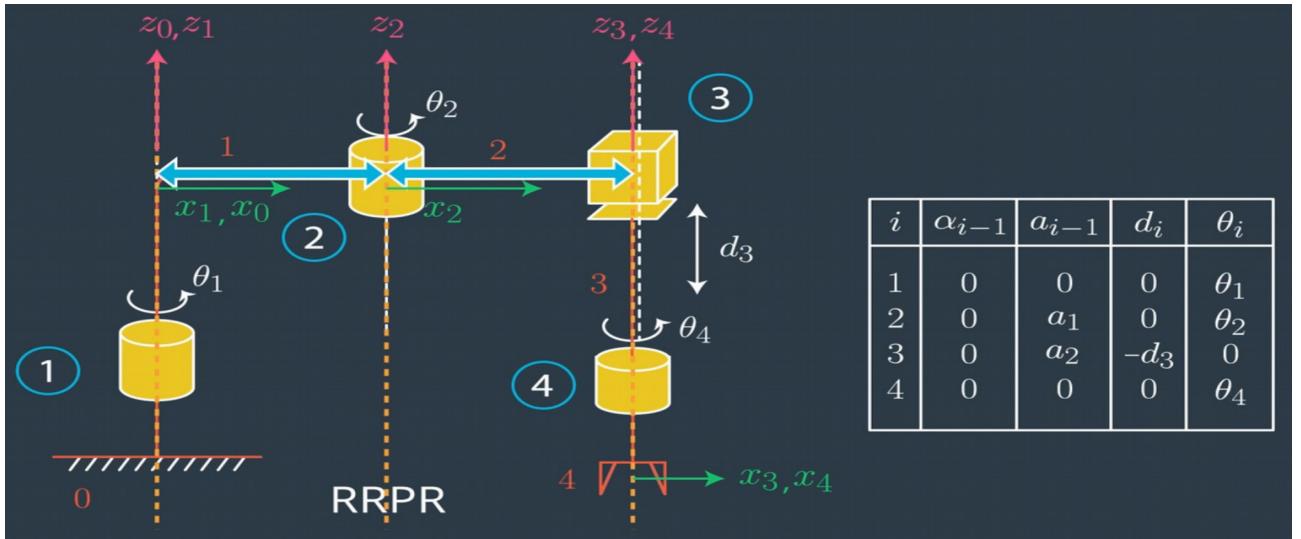
Is to assign the X axis of length 0.



The number of non-zero DH parameters can be minimized by **always choosing** X0 coincident with X1, when the first joint variable here  $\Theta_1$  is 0 and **always choosing** Z0 to be coincident with Z1.

### Step 8:

And finally, assigning the x axis for link n, the last link.

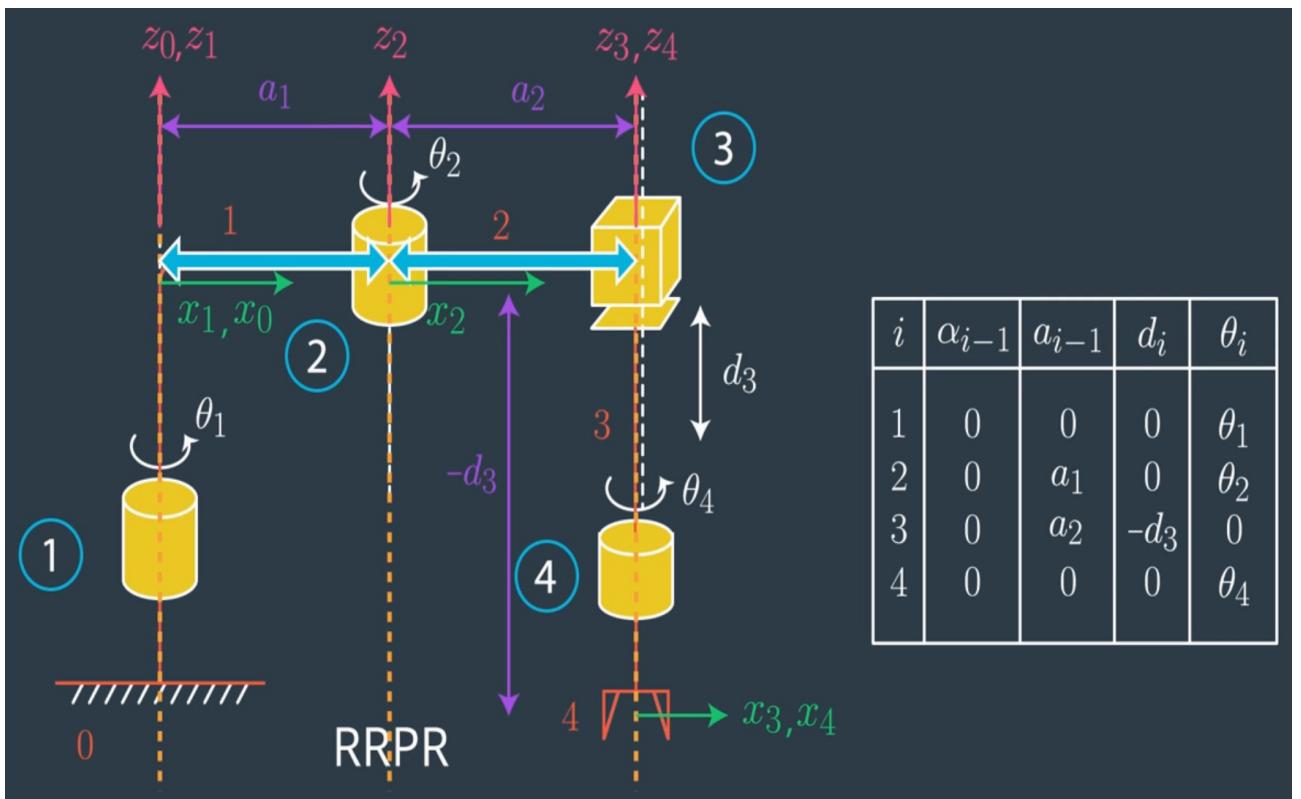


Here we always choose  $X_n$  to be in the same direction as  $X_{n-1}$ , when the last joint has a zero angle or zero displacement depending on what type of joint it is. Thus,  $X_4$  is parallel to  $X_3$ .

One last comment before we move on the parameter values. The total homogeneous transform we will construct relates the origin of link 0 to the origin of link 4. Notice that the origin of frame 0 is not physically connected to link 0. So if we were really interested in knowing the transform relative to some point on the actual fixed base, then we could move  $X_0$  down to that point, at the cost of introducing one or more non-zero DH parameters.

## DH Parameter Table

Now, it's time to fill in the table. But first let's see it completed!



Remember, each row is the transform from link<sub>i-1</sub> to link<sub>i</sub>. If your DH parameters are correct, then a basis vector of link<sub>i-1</sub> should be coincident with the corresponding basis vector and link<sub>i</sub> after the homogeneous transform is applied. For example X<sub>0</sub> should map to X<sub>1</sub> and Z<sub>0</sub> should map to Z<sub>1</sub>.

For **i=1**, we are considering the transform between frames 0 and 1.

- Alpha\_0 is the twist angle between Z<sub>0</sub> and Z<sub>1</sub>, measured about X<sub>0</sub> in a right hand sense.
- A\_0 is the distance from Z<sub>0</sub> to Z<sub>1</sub> measured along X<sub>0</sub>, well Z<sub>0</sub> and Z<sub>1</sub> are collinear which is very convenient **because for colinear axis, alpha and a are always 0**.
- The link offset d<sub>1</sub> is the signed distance from X<sub>0</sub> to X<sub>1</sub>, measured along Z<sub>1</sub>. Again, we have collinear axis which makes d<sub>1</sub> zero.
- Joint 1 is a revolute. So  **$\Theta_1$  is the only non-constant term** in row one. It measures the angle between X<sub>0</sub> and X<sub>1</sub>, about the Z<sub>1</sub> axis in a right hand sense. Since X<sub>0</sub> and X<sub>1</sub> are parallel, while  $\Theta_1$  equals 0 this term is just  $\Theta_1$ . Here's an important point, we may run into cases where the two X axes are not parallel when theta is zero. If so, the theta term would be theta **plus whatever the non-zero but constant offset happens to be**. So, be aware of that.

Let's move on to **i=2**. That is the transform between frames 1 and 2.

- Z<sub>1</sub> is parallel to Z<sub>2</sub>, so Alpha\_1 is again zero.
- A\_1 is the distance from Z<sub>1</sub> to Z<sub>2</sub>, measured along X<sub>1</sub>. Here it is a<sub>1</sub>. **To find the actual numerical value for a<sub>1</sub>, we would have to consult the manufacturer's specs or the appropriate joint origin tag in the manipulator's URDF file.**
- d<sub>2</sub> is the signed distance from X<sub>1</sub> to X<sub>2</sub> measured along Z<sub>2</sub>. Since X<sub>1</sub> is collinear to X<sub>2</sub>, d<sub>2</sub> is zero.
- Joint 2 is another revolute. And in fact, the same logic applies as we saw in Joint 1. Therefore,  $\Theta_2$  is just  $\Theta_2$ .

**i=3:**

- Z<sub>2</sub> is parallel to Z<sub>3</sub>, this makes the twist angle Alpha\_2 equal to zero.
- A\_2 is the distance between Z<sub>2</sub> and Z<sub>3</sub> measured along X<sub>2</sub>, here a<sub>2</sub>
- d<sub>2</sub> is the signed distance from X<sub>2</sub> to X<sub>3</sub>, measured along Z<sub>3</sub>. Notice that to get from X<sub>2</sub> to X<sub>3</sub>, we have to move in the negative Z<sub>3</sub> direction. (**d<sub>3</sub> is the variable here**)
- Joint 3 is prismatic, which means d<sub>3</sub> is the only variable in row three. And thus,  $\Theta_3$  is zero.

Finally **i=4**:

- This is similar to the transform between links zero and one, because we again define frame four to be coincident with frame three, when  $\Theta_4$  equals zero. As a result Alpha\_3, A\_3 and d<sub>4</sub> are all zero. And  $\Theta_4$  is just  $\Theta_4$ .

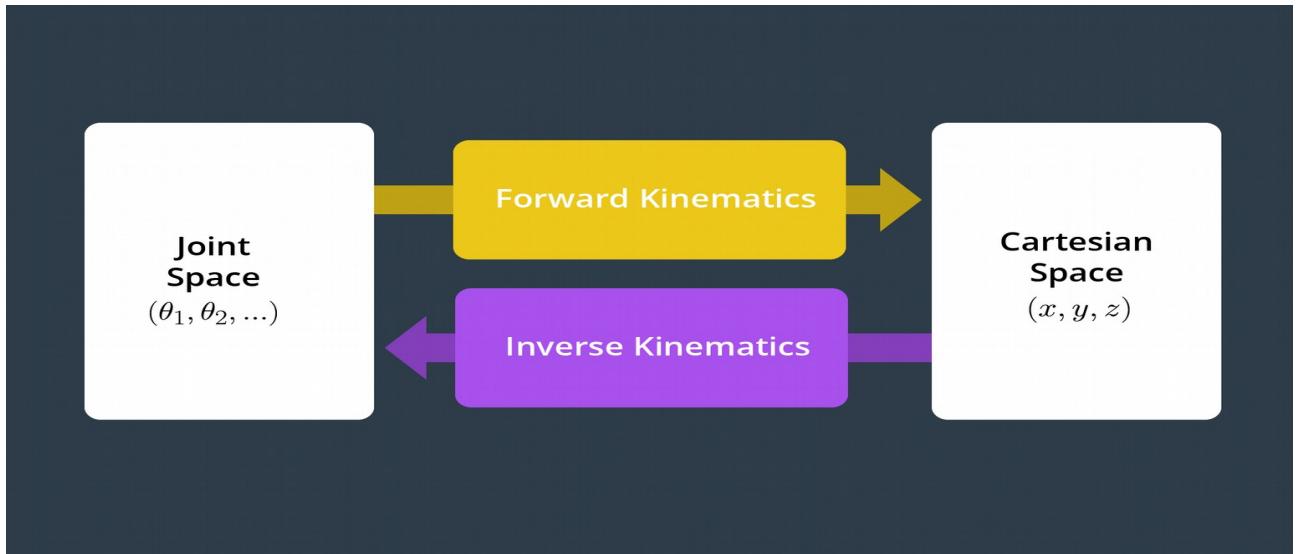
## 8.2.12 Forward and Inverse Kinematics

## Forward Kinematics

We have almost made it to the end of this chapter!!!!

Best of all, we have already seen everything we need to know to solve the forward kinematics (FK) problem for a serial manipulator. Putting it all together is going to be a piece of cake.

Recall that, in the FK problem, we know all the joint variables, that is the generalized coordinates associated with the revolute and prismatic joints, and we wish to calculate the pose of the end effector in a 3D world. **Next we will consider the more challenging inverse kinematics problem.** In the IK problem the position and orientation of the end effector is known and the objective is to solve for the joint variables.



The **FK problem boils down to the composition of homogeneous transforms.** We start with the base link and move link by link to the end effector. And **we use the DH parameters to build each individual transform.**

$${}^N_T = {}^0_1 T {}^1_2 T {}^2_3 T \dots {}^{N-1}_N T$$

Recall that the total transform between adjacent links,

$${}_{i-1}^i T = \begin{bmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ s\theta_i c\alpha_{i-1} & c\theta_i c\alpha_{i-1} & -s\alpha_{i-1} & -s\alpha_{i-1} d_i \\ s\theta_i s\alpha_{i-1} & c\theta_i s\alpha_{i-1} & c\alpha_{i-1} & c\alpha_{i-1} d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**for each link is actually composed of four individual transforms, 2 rotations and 2 translations, performed in the order shown here**

$${}_{i-1}^i T = R_X(\alpha_{i-1}) D_X(a_{i-1}) R_Z(\theta_i) D_Z(d_i)$$

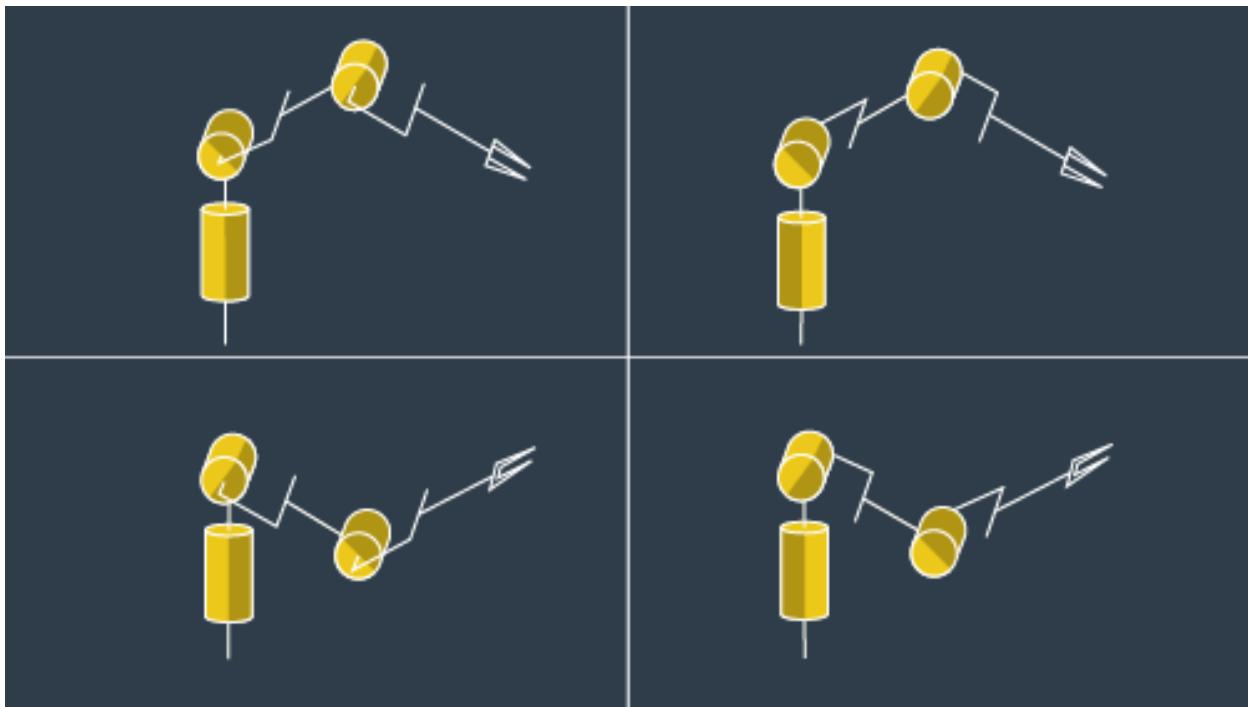
## Inverse Kinematics

Inverse kinematics (IK) is essentially the opposite idea of forwards kinematics. In this case, the pose (i.e., position and orientation) of the end effector is known and the goal is to calculate the joint angles of the manipulator. However, the IK problem is significantly more challenging. As we saw in the DH parameter section, the homogeneous transformation between neighboring links is,

$${}_{i-1}^i T = \begin{bmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ s\theta_i c\alpha_{i-1} & c\theta_i c\alpha_{i-1} & -s\alpha_{i-1} & -s\alpha_{i-1} d_i \\ s\theta_i s\alpha_{i-1} & c\theta_i s\alpha_{i-1} & c\alpha_{i-1} & c\alpha_{i-1} d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Thus, for a manipulator with  $n$ -joints the overall transformation between the base and end effector involves  $n$ -multiples of Equation (1). Clearly this can result in complicated, highly non-linear equations that, in general, can have zero or multiple solutions. Further, these mathematical solutions may in fact violate real-world joint limits so care is needed when choosing among the mathematically possible solution(s). Let's pause for a moment and think about what these different solutions physically mean.

Consider the case of an "anthropomorphic" (RRR) manipulator. The name comes from the fact that joint 1 is analogous to the human hip joint (imagine twisting at the waist about a vertical axis), joint 2 is the "shoulder", and joint 3 is the "elbow". If all we care about is the *position* of the end effector, there are four possible ways to reach a 3D point in the manipulator's workspace. Notice the shading on each revolute joint. The top row corresponds to the "elbow up" solutions. Column 1 corresponds to solutions with hip joint  $= \theta$  versus column 2 with hip joint  $= \theta + \pi$ .



If there are, in general, multiple solutions for every pose within a manipulator's workspace, what do you think the case of no solution corresponds to? If you guessed that the pose is outside of the manipulator's workspace, you guessed right!

There are two distinct solution methods to solving the IK problem. The first is a purely numerical approach. Essentially, this method is guess and iterate until the error is sufficiently small or it takes so long that you give up. The Newton-Raphson algorithm is a common choice because it is conceptually simple and has a quadratic rate of convergence if the initial guess is "sufficiently close" to the solution. However, there is no guarantee that the algorithm will converge or do so quickly enough to meet application requirements and it only returns one solution. To generate solutions for the various possible poses, different initial conditions must be used. The advantage of the numerical approach is that the same algorithm applies to all serial manipulators.

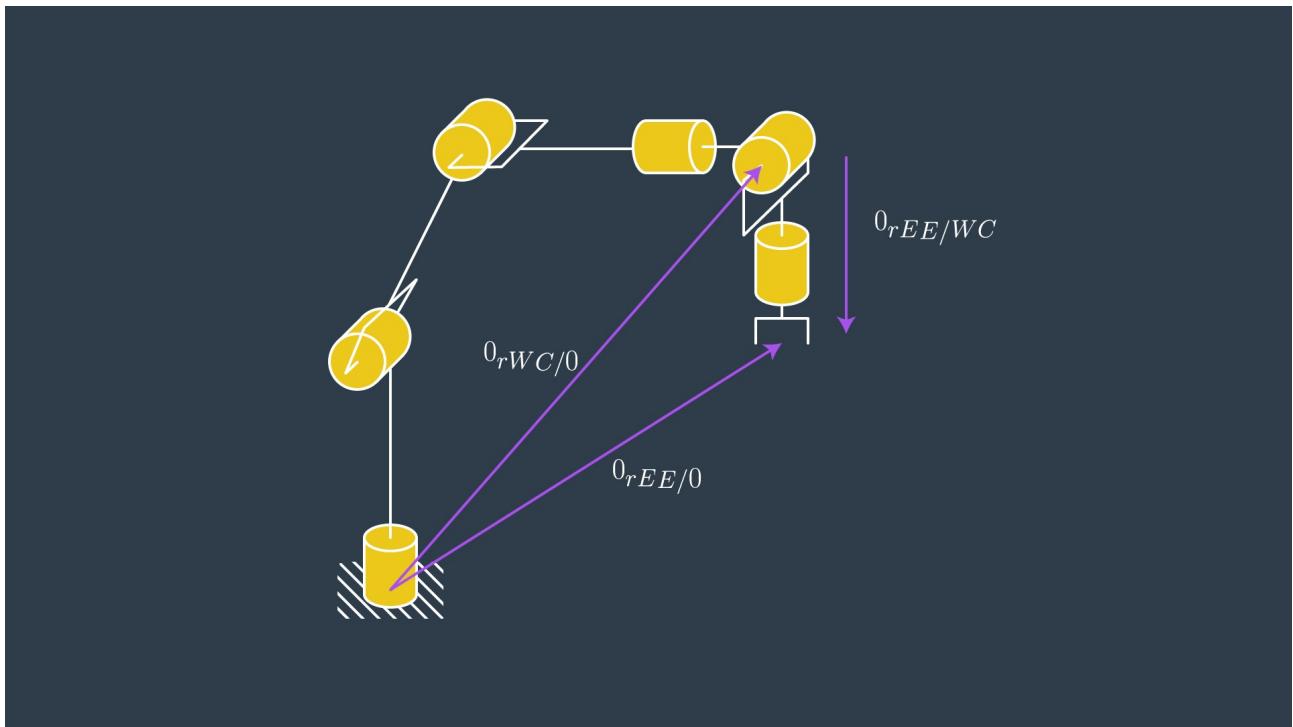
The other, and much preferred, solution method is known as an "analytical" or "closed-form" solution. Closed-form solutions are specific algebraic equation(s) that do not require iteration to solve and have two main advantages: generally they are much faster to solve than numerical approaches and it is easier to develop rules for which of the possible solutions is the appropriate one. However, only certain types of manipulators are solvable in closed-form. The obvious question is, so what types of manipulators have a closed-form solution? Research has shown that if either of the following two conditions are satisfied, then the serial manipulator is solvable in closed-form.

1. Three neighboring joint axes intersect at a single point, or
2. Three neighboring joint axes are parallel (which is technically a special case of 1, since parallel lines intersect at infinity)

Fortunately, the majority of six DoF serial manipulators currently used in industry will satisfy one of the above conditions.

Often the last three joints in a manipulator are revolute joints that satisfy condition 1, such a design is called a **spherical wrist** and the common point of intersection is called the **wrist center**. The advantage of such a design is that it **kinematically decouples** the position and orientation of the end effector. Mathematically, this means that instead of solving *twelve* nonlinear equations simultaneously (one equation for each term in the first three rows of the overall homogeneous transform matrix), it is now possible to independently solve two simpler problems: first, the Cartesian coordinates of the wrist center, and then the composition of rotations to orient the end effector. Physically speaking, a six degree of freedom serial manipulator with a spherical wrist would use the first three joints to control the *position* of the wrist center while the last three joints would orient the end effector as needed.

We will now formalize the solution procedure for serial manipulators with a spherical wrist. Consider the six degree of freedom manipulator shown here with joints 4, 5, and 6 comprising the spherical wrist. The location of the wrist center (WC) and end effector (EE) relative to the base frame "0" is given by,  ${}^0r_{WC/0}$  and  ${}^0r_{EE/0}$ , respectively. The location of the EE relative to the WC is given by,  ${}^0r_{EE/WC}$ . Note that all three vectors are expressed in terms of the base frame as is indicated by the leading superscript, "0".



**Step 1:** is to complete the DH parameter table for the manipulator. Hint: place the origin of frames 4, 5, and 6 coincident with the WC.

**Step 2:** is to find the location of the WC relative to the base frame. Recall that the overall homogeneous transform between the base and end effector has the form,

$${}_{EE}^0 T = \begin{bmatrix} {}_6^0 R & {}^0 r_{EE/0} \\ \begin{matrix} 0 & 0 & 0 \end{matrix} & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If, for example, you choose  $z_4$  parallel to  $z_6$  and pointing from the WC to the EE, then this displacement is a simple translation along  $z_6$ . The magnitude of this displacement, let's call it  $d$ , would depend on the dimensions of the manipulator and are defined in the URDF file. Further, since  $r_{13}$ ,  $r_{23}$ , and  $r_{33}$  define the Z-axis of the EE relative to the base frame, the Cartesian coordinates of the WC is,

$${}^0 r_{WC/0} = {}^0 r_{EE/0} - d \cdot {}_6^0 R \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} - d \cdot {}_6^0 R \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

**Step 3:** find joint variables,  $q_1$ ,  $q_2$  and  $q_3$ , such that the WC has coordinates equal to equation (3).

This is the hard step. One way to attack the problem is by repeatedly projecting links onto planes and using trigonometry to solve for joint angles. Unfortunately, there is no generic recipe that works for all manipulators so you will have to experiment. The example in the next section will give you some useful guidance.

**Step 4:** once the first three joint variables are known, calculate 03R via application of homogeneous transforms up to the WC.

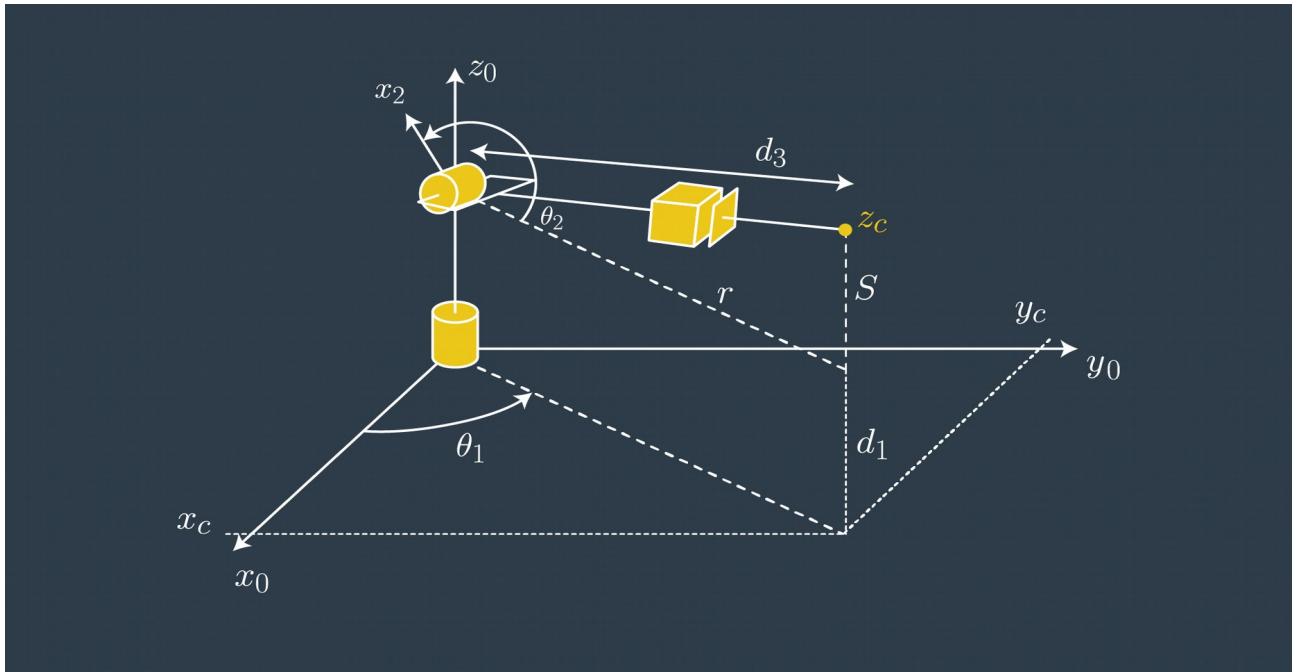
**Step 5:** find a set of Euler angles corresponding to the rotation matrix,

$${}^3_6 R = \left({}^0_6 R\right)^{-1} {}^0_0 R = \left({}^0_3 R\right)^T {}^0_6 R$$

**Step 6:** choose the correct solution among the set of possible solutions

## Inverse Kinematics Example

To give you more insight into the solution process, particularly for the first three joint variables, we present an example of an RRP manipulator.



The point  $z_c$  could be considered to be the wrist center of a spherical wrist. We will assume that the Cartesian coordinates of  $z_c$  have already been calculated. To find  $\theta_1$ , we need to project  $z_c$  onto the ground plane - a trivial task since it only requires setting the z-coordinate = 0! Thus,

$$\theta_1 = \text{atan2}(y_c, x_c)$$

To solve for  $\theta_2$ , imagine  $\theta_1 = 0$  and project links 2 and 3 onto the x-z plane. Again, we use the  $\text{atan2}$  function,

$$\theta_2 = \text{atan2}(s, r)$$

where,

$$r^2 = x_c^2 + y_c^2$$

and,

$$s = z_c - d_1$$

The final joint variable,  $d_3$ , represents the length of the prismatic joint. From the figure, it is clear that  $d_3$  is the hypotenuse of a right triangle with sides “ $r$ ” and “ $s$ ”. Solving for  $d_3$ ,

$$d_3 = \sqrt{r^2 + s^2} = \sqrt{x_c^2 + y_c^2 + (z_c - d_1)^2}$$

And that's it! The first three joint variables are now described explicitly in closed-form.

## 8.3 Summary

Well that's it, we now have all the theoretical background we need in order to complete Pick and Place Tasks, so let's recap what we covered.

First, we saw that reference frames are really important, they allow you to express vectors in whichever frame is most convenient or intuitive to use.

But the use of multiple frames motivated the need for establishing systematic relationships between the frames that, in the most general sense, differ by a translation and arbitrary rotation. The homogeneous transform provided a compact and computer-friendly way of converting between frames.

Further, the Denavit-Hartenberg convention provided a systematic method for assigning a reference frame to each link.

All of this was done to enable us to solve the forward and inverse kinematics of serial manipulators.

**If you want see how we construct the DH table for a more complicated 6 Degrees of Freedom industrial robot specifically the kuka KR210, then making an IK\_server on ROS to solve the inverse kinematics so that the MoveIt framework can use it to command the robot you can see the appendix!!**

**Furthermore you will see the hardware interface of a simple hobbyist grade robotic arm, and how to connect with ros control and MoveIt**

# APPENDIX EXPERIMENTAL RESULTS

## 9. Setting up the ROS Prerequisites

### 9.1 URDF of the robot

**Under the following link we can see the main xacro file of our robot that calls all the necessarily macros to build our robot description.**

[https://github.com/panagelak/Open\\_Mobile\\_Manipulator/blob/master/ommp\\_description/urdf/robots/lynxbot/lynx\\_rover.urdf.xacro](https://github.com/panagelak/Open_Mobile_Manipulator/blob/master/ommp_description/urdf/robots/lynxbot/lynx_rover.urdf.xacro)

**here is the xacro**

```
<?xml version='1.0'?>

<robot name="lynx_rover" xmlns:xacro="http://www.ros.org/wiki/xacro">

<!-- Common design Properties-->
<xacro:include filename="$(find
ommp_description)/urdf/robots/lynxbot/common_properties.urdf.xacro" />

<!-- Include Sensors macros -->

<!--ydlidar -->
<xacro:include filename="$(find
ommp_description)/urdf/sensors/ydlidar/ydlidar.urdf.xacro" />
<!--imu -->
<xacro:include filename="$(find
ommp_description)/urdf/sensors imu/imu.urdf.xacro" />
<!-- kinect -->
<xacro:include filename="$(find
ommp_description)/urdf/sensors/kinect/kinect.urdf.xacro" />

<!-- Include Bases-->
<xacro:include filename="$(find
ommp_description)/urdf/bases/lynxbot_base/lynxbot_bases.urdf.xacro" />

<!-- Wheel Macro-->
<xacro:include filename="$(find
ommp_description)/urdf/wheels/costum_wheel/costum_wheel.urdf.xacro" />

<!-- Incude Inertia macros -->
<xacro:include filename="$(find
ommp_description)/urdf/macros/inertia_macros.urdf.xacro" />

<!-- Include costum Arm -->
<xacro:include filename="$(find
ommp_description)/urdf/arms/costum_arm/costum_arm.urdf.xacro" />
```

```

<!-- Include costum Gripper -->
<xacro:include filename="$(find ommp_description)/urdf/grippers/costum_gripper/
costum_gripper.urdf.xacro" />

<xacro:if value="$(arg sim)" >

  <!-- General -->
  <xacro:include filename="$(find
ommp_description)/urdf/robots/lynxbot/lynx_rover.gazebo.xacro" />

</xacro:if>

<!-- All the bases of the robot -->
<xacro:lynxbot_bases />

<!-- Instanciate the 4 wheels of the robot -->
<xacro:costum_wheel prefix="front_left" parent="chassis" gz="true">
  <origin xyz="${wheel_offset_x} ${wheel_offset_y} ${wheel_offset_z}" rpy="0 0
0"/>
</xacro:costum_wheel>

<xacro:costum_wheel prefix="front_right" parent="chassis" gz="true">
  <origin xyz="${wheel_offset_x} -${wheel_offset_y} ${wheel_offset_z}" rpy="0 0
0"/>
</xacro:costum_wheel>

<xacro:costum_wheel prefix="back_left" parent="chassis" gz="true">
  <origin xyz="-${wheel_offset_x} ${wheel_offset_y} ${wheel_offset_z}" rpy="0 0
0"/>
</xacro:costum_wheel>

<xacro:costum_wheel prefix="back_right" parent="chassis" gz="true">
  <origin xyz="-${wheel_offset_x} -${wheel_offset_y} ${wheel_offset_z}" rpy="0 0
0"/>
</xacro:costum_wheel>

<!-- Instanciate the sensor macros -->
<!--(The gazebo plugin macros are instanciated on their corresponding xacro
files) -->

<xacro:sensor_kinect parent="${kinect_parent}" />

<xacro:sensor_laser parent="chassis" />

<xacro:imu_sensor parent="chassis" />

<!-- connect arm -->
<xacro:costum_arm parent="${arm_parent}" />
<!-- connect gripper-->
<xacro:costum_ee parent="${ee_parent}" />

</robot>

```

I tried to make the xacro as easy to modify as possible, in order to quickly set up your costum mobile manipulators.

On the same directory you can find the common\_properties.xacro where are some global variables that are available for all the macros, for example position of the lidar, kinect, arm etc size of the robot base etc.

On the bases macros you can find the robot\_footprint alongside upper level, kinect bases etc

On the common\_properties there is arg sim, this defines whether we include the gazebo.xacro that are needed on the simulation but not on the real robot, for example ros\_control\_plugin and gazebo\_sensor\_macros. Furthermore when we include for example the kinect xacro file this argument get's passed there and this file will decide if it will include the kinect.gazebo xacro-plugin and whether it will instantiate this gazebo macro.

common\_properties for robot building

```
<?xml version="1.0" ?>

<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

<!-- Include or Not Gazebo Files-->

<xacro:arg name="sim" default="true"/>
<xacro:arg name="gaz" default="true"/>

<!-- PI Value -->
<xacro:property name="M_PI" value="3.141592653" />
<xacro:property name="PI" value="3.141592653" />

<!--chassis-->
<xacro:property name="base_size">
<box size = "0.25 0.2 0.06"/>
</xacro:property>

<!-- wheels-->
<xacro:property name="wheel_size">
<cylinder length = "0.065" radius="0.055" />
</xacro:property>

<xacro:property name="wheel_offset_x" value="0.0975" />
<xacro:property name="wheel_offset_y" value="0.14" />
<xacro:property name="wheel_offset_z" value="-0.015" />
```

```

<!--Robot bases-->

<xacro:property name="upper_base_size">
  <box size = "0.3 0.15 0.02"/>
</xacro:property>

<xacro:property name="upper_base_joint_origin">
  <origin xyz="0.05 0.0 0.1275" rpy="0 0 0" />
</xacro:property>

<xacro:property name="kinect_base1_size">
  <box size = "0.01 0.03 0.38"/>
</xacro:property>

<xacro:property name="kinect_base2_size">
  <box size = "0.07 0.07 0.02"/>
</xacro:property>

<xacro:property name="kinect_base1_joint_origin">
  <origin xyz="-0.14 0.0 0.2" rpy="0 0 0" />
</xacro:property>

<xacro:property name="kinect_base2_joint_origin">
  <origin xyz="0.035 0.0 0.19" rpy="0 0 0" />
</xacro:property>

<!-- imu-->

<xacro:property name="imu_parent" value="chassis" />

<xacro:property name="imu_size">
  <box size = "0.01 0.01 0.01"/>
</xacro:property>

<xacro:property name="imu_joint_origin">
  <origin xyz="-0.1 0.0 0.0575" rpy="0 0 0" />
</xacro:property>
```

```

<!-- lidar base-->

<xacro:property name="lidar_parent" value="chassis" />

<xacro:property name="lidar_joint_origin">
  <origin xyz="0.0 0.0 0.0575" rpy="0 0 0" />
</xacro:property>

<!-- kinect base-->

<xacro:property name="kinect_parent" value="kinect_base2" />
<xacro:property name="kinect_joint_origin">
  <origin xyz="0.03 0.0 0.1" rpy="0 0.5 0" />
</xacro:property>

<!-- Arm -->

<xacro:property name="arm_joint_origin">
  <origin xyz="0.079 0.0 0.02" rpy="0 0 0" />
</xacro:property>

<xacro:property name="arm_parent" value="upper_base" />
<xacro:property name="ee_parent" value="link_5" />
</robot>

```

### example of the ydlidar xacro macro

```

<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="lynx_rover">

<xacro:if value="$(arg sim)">
  <!-- Include Ydlidar Plugin -->
  <xacro:include filename="$(find
    ommp_description)/urdf/sensors/ydlidar/gazebo.xacro" />

  <xacro:laser_gazebo_sensor link_name="laser_link" laser_name="ydlidar"
    frame_name="laser_link"/>
</xacro:if>

```

```

<!-- ydlidar Link -->
<xacro:macro name="sensor_laser" params="parent">
<link name="laser_link">

<inertial>
<mass value="0.1"/>
<origin xyz="0 0 0.0" rpy="0 0 0"/>
<inertia
  ixx="1e-6" ixy="0" ixz="0"
  iyy="1e-6" iyz="0"
  izz="1e-6"
/>
</inertial>
<visual>
<geometry>

<mesh
filename="package://ommp_description/meshes/sensors/ydlidar/ydlidar.dae"/>
</geometry>
<origin xyz="0 0 0" rpy="0 0 0"/>
</visual>
<collision>
<origin xyz="0.0 0.0 0.0" rpy="0 0 0"/>
<geometry>
<box size="0.0 0.0 0.0"/>
</geometry>
</collision>
</link>

<joint type="fixed" name="ydlidar_joint">
<xacro:insert_block name="lidar_joint_origin" />
<child link="laser_link"/>
<parent link="${parent}"/>
</joint>
</xacro:macro>

</robot>

```

## example of the gazebo sensor plugin for the ydlidar

```
<?xml version="1.0"?>

<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

<xacro:macro name="laser_gazebo_sensor" params="link_name laser_name frame_name">
<!-- hokuyo -->
<gazebo reference="${link_name}">
<sensor type="gpu_ray" name="${laser_name}">
<pose>0 0 0 0 0 0</pose>
<visualize>false</visualize>
<update_rate>40</update_rate>
<ray>
<scan>
<horizontal>
<samples>720</samples>
<resolution>1</resolution>
<min_angle>-3.14</min_angle>
<max_angle>3.14</max_angle>
</horizontal>
</scan>
<range>
<min>0.3</min>
<max>50.0</max>
<resolution>0.01</resolution>
</range>
<noise>
<type>gaussian</type>
<!-- Noise parameters based on published spec for Hokuyo laser
achieving "+-30mm" accuracy at range < 10m. A mean of 0.0m and
stddev of 0.01m will put 99.7% of samples within 0.03m of the true
reading. -->
<mean>0.0</mean>
<stddev>0.0</stddev>
</noise>
</ray>
<plugin name="gazebo_ros_head_ydlidar_controller"
filename="libgazebo_ros_gpu_laser.so">
```

```

<topicName>/scan</topicName>
<frameName>${frame_name}</frameName>
</plugin>
</sensor>
</gazebo>
</xacro:macro>

</robot>

```

## 9.2 Spawning the robot in simulation

**To Spawn the robot description in the parameter server – the robot state publisher – to load the controllers configuration on the parameter server – to launch the controller manager - and finally to spawn the robot in the gazebo environment.**

You can use the following main launch file.

`sim_bringup.launch`

```

<?xml version="1.0" encoding="UTF-8"?>

<launch>

<!-- parameters -->
<arg name="sim" default="true"/>
<arg name="world" default="simple_world"/>
<arg name="paused" default="false"/>
<arg name="use_sim_time" default="true"/>
<arg name="gui" default="true"/>
<arg name="headless" default="false"/>
<arg name="debug" default="false"/>

<!-- send urdf to param server -->
<param name="robot_description" command="$(find xacro)/xacro '$(find
ommp_description)/urdf/robots/lynxbot/lynx_rover.urdf.xacro' sim:=$(arg sim)" />
<!-- Send robot states to tf -->
<node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher" respawn="false" output="screen">
  <param name="publish_frequency" type="double" value="10.0" />
</node>
<!-- load the controller configuration on the param server -->

```

```

<rosparam file="$(find ommp_bringup)/config/control/my_robot_control.yaml"
command="load"/>

<rosparam file="$(find ommp_bringup)/config/control/joint_state_controller.yaml"
command="load"/>

<rosparam file="$(find ommp_bringup)/config/control/diff_control.yaml"
command="load"/>

<!-- launch the controller manager for ros control --&gt;
&lt;node name="controller_spawner" pkg="controller_manager" type="spawner"
respawn="false"

  output="screen" args="joint_state_controller diff_velocity_controller
arm_controller gripper_controller kinect_controller"/&gt;

<!-- launch the gazebo world --&gt;
&lt;include file="$(find gazebo_ros)/launch/empty_world.launch"&gt;

  &lt;arg name="world_name" value="$(find ommp_simulation)/worlds/${arg
world}.world"/&gt;

  &lt;arg name="paused" value="${arg paused}"/&gt;
  &lt;arg name="use_sim_time" value="${arg use_sim_time}"/&gt;
  &lt;arg name="gui" value="${arg gui}"/&gt;
  &lt;arg name="headless" value="${arg headless}"/&gt;
  &lt;arg name="debug" value="${arg debug}"/&gt;
&lt;/include&gt;

<!--spawn the robot in gazebo world--&gt;
&lt;node name="urdf_spawner_lynx_rover" pkg="gazebo_ros" type="spawn_model"
respawn="false"

  output="screen" args="-urdf -param robot_description -model lynx_rover -z 0.0"/&gt;

<!-- robot localization sensor fusion wheel encoders + imu --&gt;
&lt;node pkg="robot_localization" type="ekf_localization_node"
name="ekf_localization"&gt;

  &lt;rosparam file="$(find ommp_bringup)/config/navigation/ekf_localization.yaml"
command="load" /&gt;

  &lt;remap from="odometry/filtered" to="odom_combined"/&gt;
&lt;/node&gt;

&lt;/launch&gt;</pre>

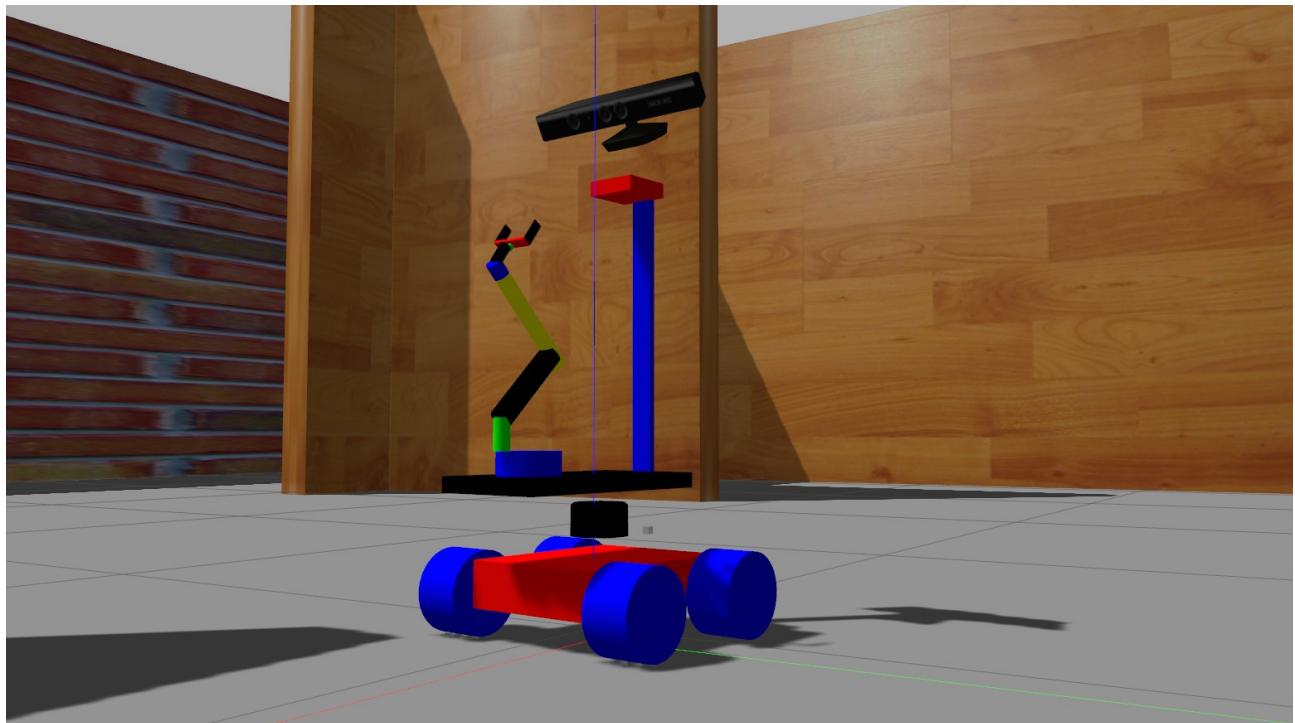
```

**After that in order to control the arm you can launch moveit and command the arm to go in it's start location with the following commands.**

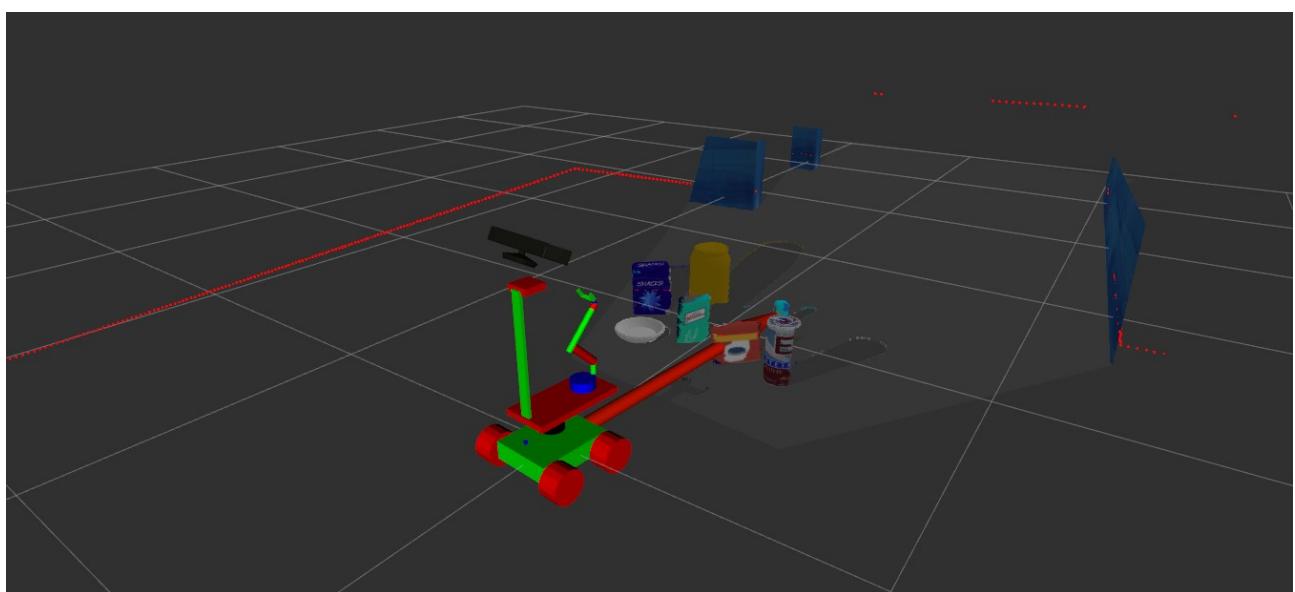
```
$ roslaunch ommp_bringup moveit.launch
```

```
$ rosrun ommp_bringup set_start_pos.py
```

**Then your robot is ready for use, able to take commands for the arm with moveit and ros control, to read the sensor values from the kinect and lidar and to command the base with cmd\_vel topic using the diff\_velocity ros controller.**



**Lastly you can launch rviz to visualize the inner representation of the robot.**

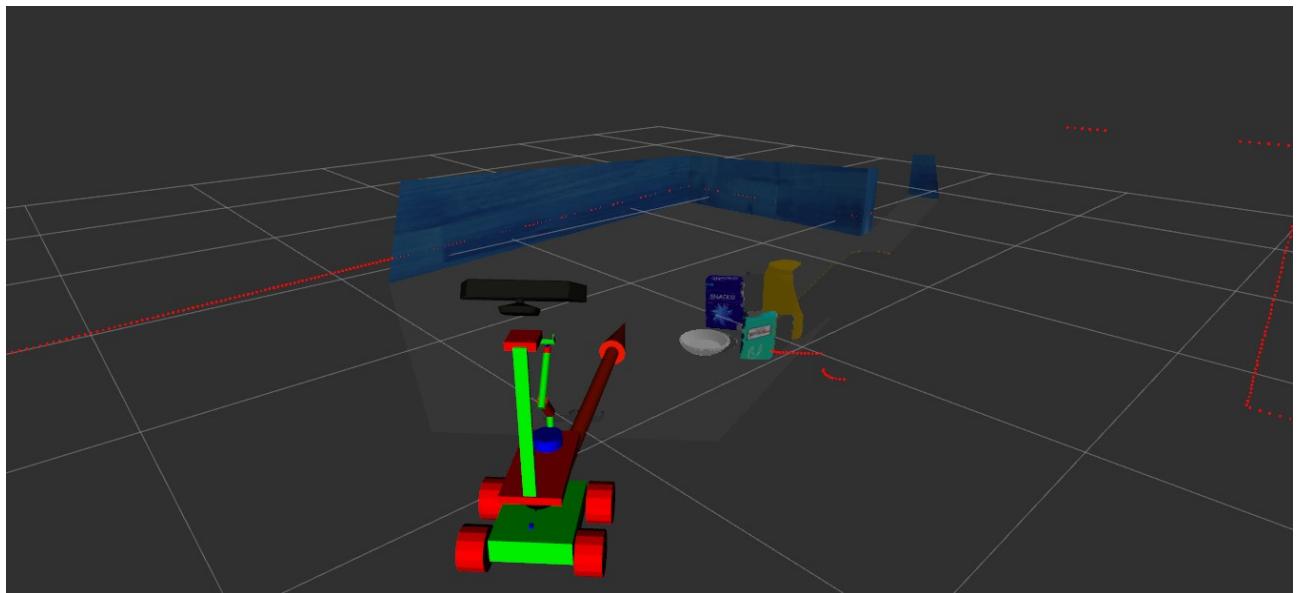


Above we can see, what the robot thinks it looks like, the pointcloud from the kinect RGB-D sensor, the laser scans from the 2D lidar and the odometry of the robot represented by the red arrow.

Next if we move the robot by publishing a cmd\_vel topic from teleop\_keyboard node.

```
$ rosrun ommp_bringup teleop.launch
```

The robot has moved accordingly



Note that for the real robot we will need to change the gazebo plugin to “real” sensor drivers. Furthermore to control the base of our robot we use an arduino due with rosserial communication. The due will accept velocity setpoint for the wheels and from the encoders will use pid controllers to keep the velocity of the wheels stable, furthermore it will publish the encoders ticks for a node on the jetson side to calculate the odometry. Lastly the due will be responsible to actuate the servos of the arm with the help of an I2C servo drivers.

For more details on this and general for the robot you can always have a look at my github repository.

[https://github.com/panagelak/Open\\_Mobile\\_Manipulator](https://github.com/panagelak/Open_Mobile_Manipulator)

# 10. Localization results

## 10.1 Kalman Filters

### 10.1.1 Programming Multidimensional KF in C++

```
#include <iostream>
#include <math.h>
#include <tuple>
#include "Core" // Eigen Library
#include "LU" // Eigen Library

using namespace std;
using namespace Eigen;

float measurements[3] = { 1, 2, 3 };

tuple<MatrixXf, MatrixXf> kalman_filter(MatrixXf x, MatrixXf P, MatrixXf u, MatrixXf F, MatrixXf H, MatrixXf R, MatrixXf I)
{
    for (int n = 0; n < sizeof(measurements) / sizeof(measurements[0]); n++) {

        // Measurement Update
        MatrixXf Z(1, 1);
        Z << measurements[n];

        MatrixXf y(1, 1);
        y << Z - (H * x);

        MatrixXf S(1, 1);
        S << H * P * H.transpose() + R;

        MatrixXf K(2, 1);
        K << P * H.transpose() * S.inverse();

        x << x + (K * y);

        P << (I - (K * H)) * P;

        // Prediction
        x << (F * x) + u;
        P << F * P * F.transpose();
    }

    return make_tuple(x, P);
}

int main()
{
    MatrixXf x(2, 1); // Initial state (location and velocity)
    x << 0,
        0;
    MatrixXf P(2, 2); // Initial Uncertainty
    P << 100, 0,
```

```

    0, 100;
MatrixXf u(2, 1); // External Motion
u << 0,
    0;
MatrixXf F(2, 2); // Next State Function
F << 1, 1,
    0, 1;
MatrixXf H(1, 2); // Measurement Function
H << 1,
    0;
MatrixXf R(1, 1); // Measurement Uncertainty
R << 1;
MatrixXf I(2, 2); // Identity Matrix
I << 1, 0,
    0, 1;

tie(x, P) = kalman_filter(x, P, u, F, H, R, I);
cout << "x= " << x << endl;
cout << "P= " << P << endl;

return 0;
}

```

## 10.1.2 Implement EKF package in ROS – Sensor Fusion

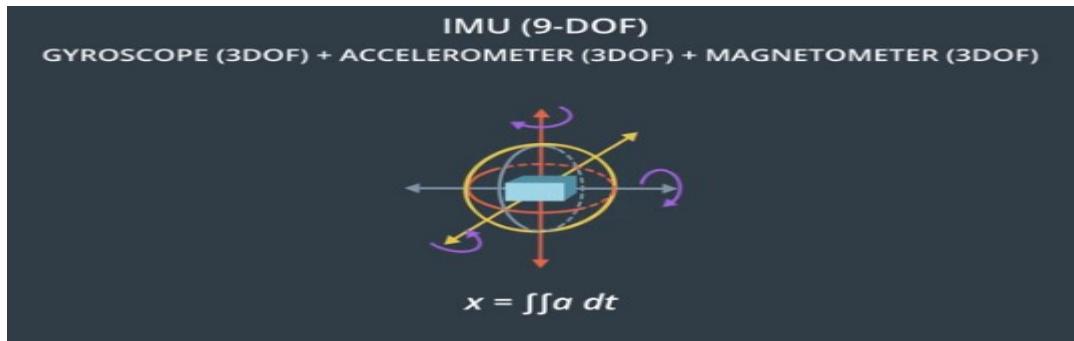
It's time to use the power of ROS and implement an EKF package mainly the **robot\_localization package**, we will collect sensitive information from the robots inertial measurement unit and the rotary encoders (optional visual odometry from visual sensors). The EKF ROS package will compare data generated from the robot's onboard sensors and apply sensor fusion to estimate the robot's pose as it moves around.

- First we need to launch our robot in the gazebo environment alongside the controllers and sensors (diff\_velocity\_controller will provide us with rotary encoders), and an inertial measurement unit plugin.
- Robot\_localization package will estimate the position and orientation of the robot.
- Odom\_to\_trajectory package which will append the odometry values over time into a trajectory path.
- Teleop package which will let us to drive the robot using keyboard commands.
- Rviz package which will let us visualize the estimated position and orientation of the robot.
- Rqt\_Mulltiplot that will let us to visualize the filtered and unfiltered trajectories X pos vs Y pos in a graph. (rqt\_plot can only do graphs with respect to time so we need to use the Multiplot package)

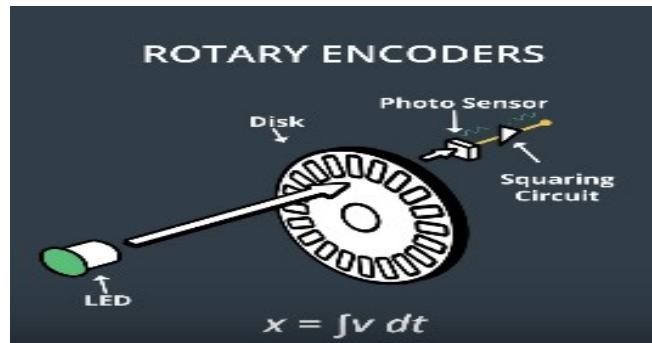
Let's see how the sensor fusion process works, Inside an environment a robot will localize itself by collecting data from its different on-board sensors.

Each of these sensors has limitations which manifest as noise and error. Nowadays, the three most common types of mobile robot sensors are **inertial measurement units, rotary encoders and vision sensors**.

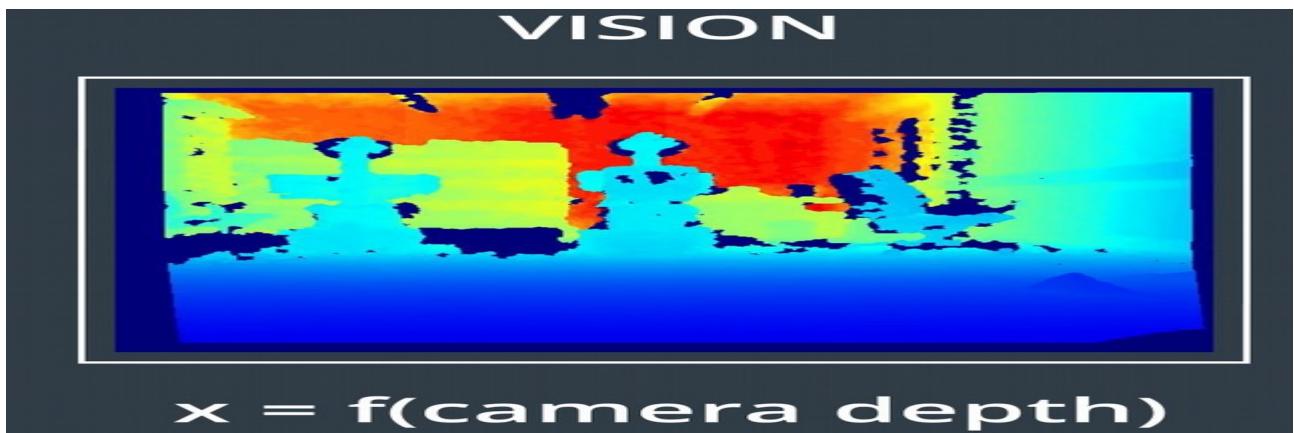
- The inertial measurement unit can measure both the linear acceleration and the angular velocity. To estimate the robot's position, a double integral is calculated. However, the IMU is noisy and so a double integration accumulates even more error over time. If you want to use an IMU be sure to check the drift for error parameters.



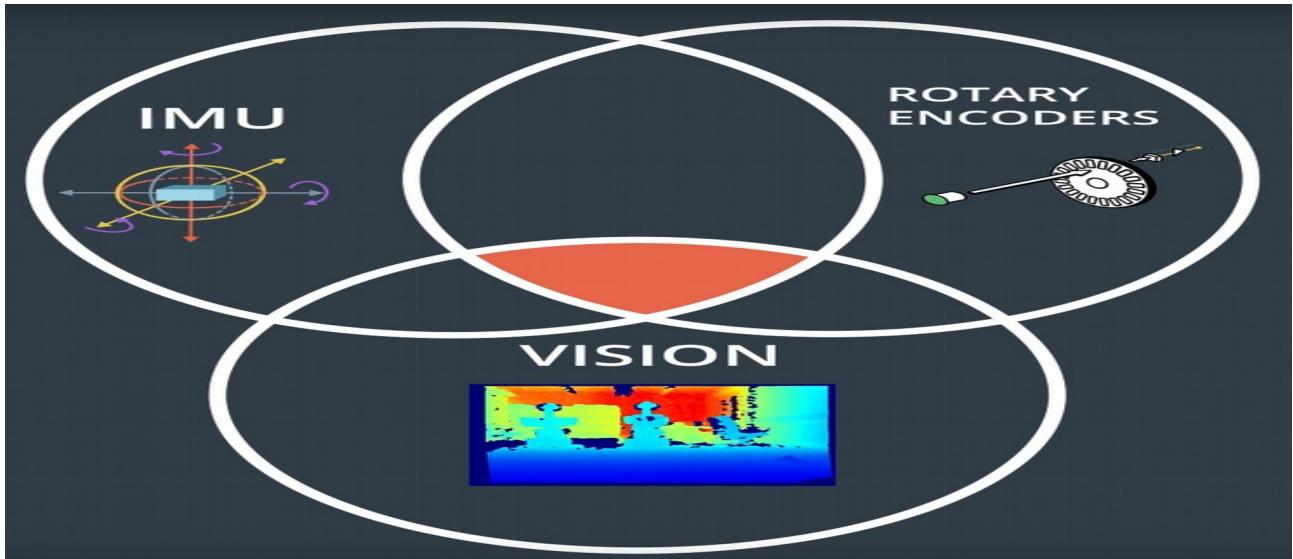
- The rotary encoder sensors are attached to the robot's actuated wheels and measures the velocity and position of the wheels. To estimate the robot's position, the integral of the velocity is calculated, but the robot wheel might be stuck between obstacles, or even worse, it might be driven on slippery floors which would lead to inaccurate and noisy measurements. So make sure to check its resolution, encoders with low resolution are usually highly sensitive to slippage



- Finally the vision sensor is often an RGB-D camera, using an RGB-D camera the robot can capture images and sense the depth towards the obstacle which can be translated to a position. But the image quality is usually affected by the light and the camera is often unable to measure the depths at small distances, so check its smallest range of operation.

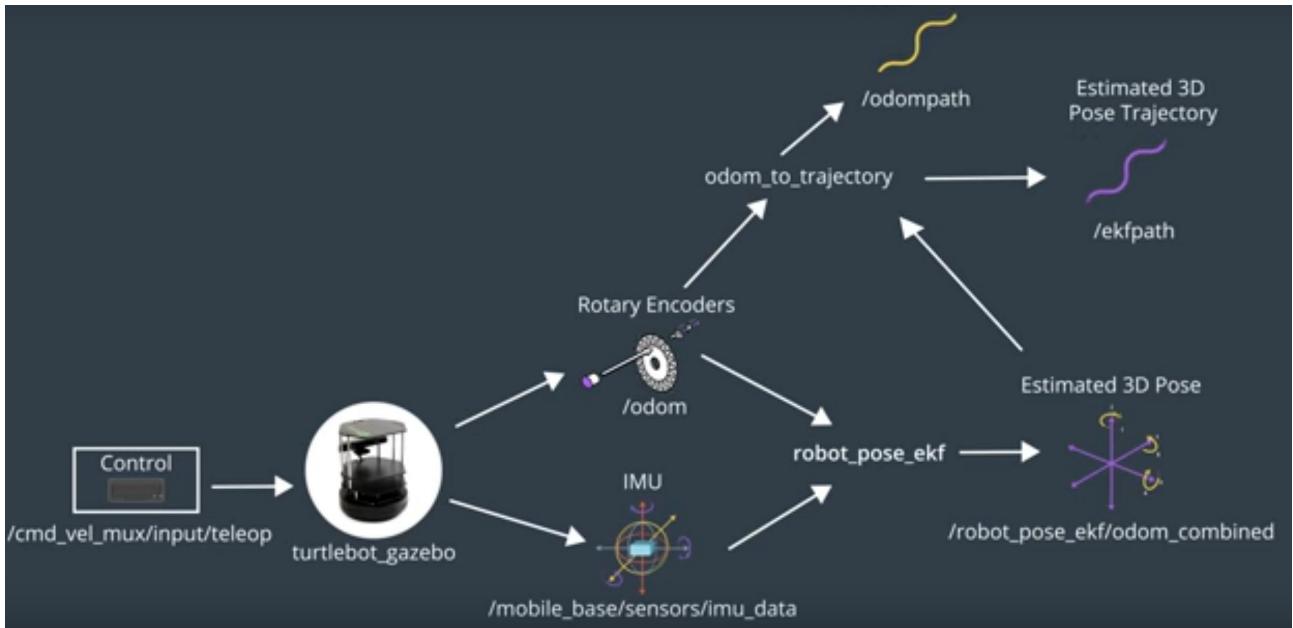


We just saw that **each and every sensor has its own limitation**, the IMU and encoders are prone to drift, whereas the camera might not always operate as expected. So we are unable to accurately determine the pose of the robot using just one of these sensors. **A sensor fusion of at least two of them is usually required.**

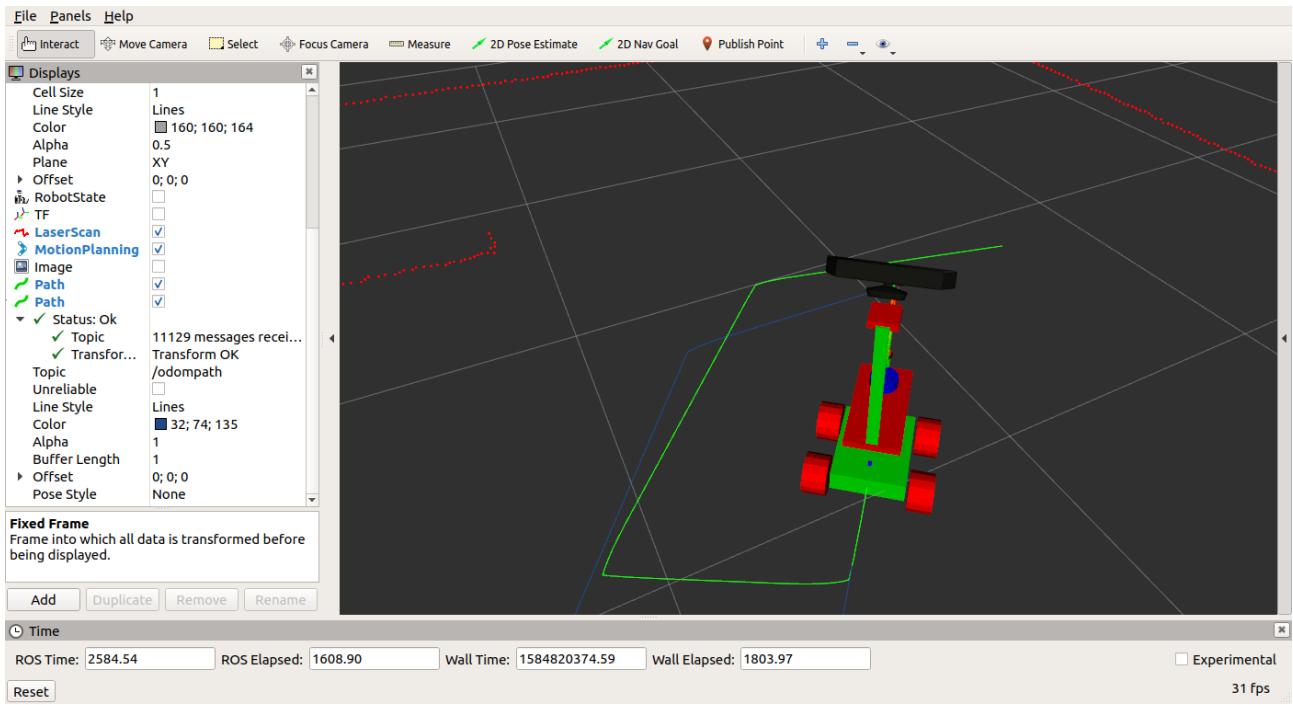


**The Extended Kalman Filter will take all the noisy measurements, compare them, filter the noise, remove the uncertainties, and provide a good estimate of the robot's pose.**

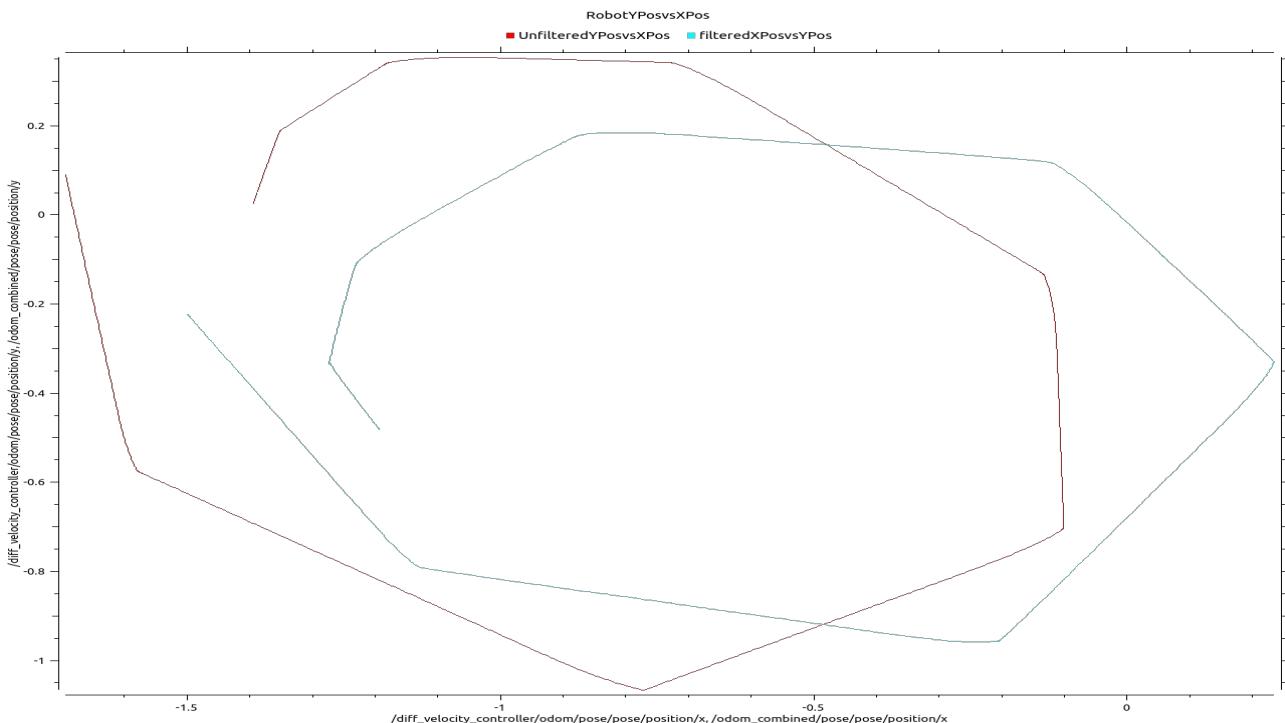
**Below is an overview of the different nodes and topics used to control the robot, read the sensors, perform EKF and compute the filtered and unfiltered trajectories.**



**Also we can visualize the filtered and unfiltered trajectories in RVIZ by moving the robot around with the teleop node.**



Finally we can plot these two trajectories with RQT\_Multiplot X Pos vs Y Pos



I think there is an error with the path of the unfiltered (encoder only trajectory), since it is from the simulation these two trajectories should be close to identical, regardless we can see that the filtered trajectory (green path), closely resembles the path of the robot!!!

## 10.2 Monte Carlo Localization

### 10.2.1 Programming Monte Carlo Localization in C++

we will program the MCL algorithm in a simple cyclic world, with 9 landmarks which are used to take sensor measurements from. The robot is a point seen as blue while the particles are seen as green and yellow.

```
//Compile with: g++ solution.cpp -o app -std=c++11 -I/usr/include/python2.7 -lpython2.7
```

```
#include "src/matplotlibcpp.h" //Graph Library
#include <iostream>
#include <string>
#include <math.h>
#include <stdexcept> // throw errors
#include <random> //C++ 11 Random Numbers
#include <vector>

namespace plt = matplotlibcpp;
using namespace std;

// Landmarks
double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },
    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },
    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters
double world_size=100.0;

// Random Generators
random_device rd;
mt19937 gen(rd());

// Global Functions
double mod(double first_term, double second_term);
double gen_real_random();

class Robot {
public:
    Robot()
    {
        // Constructor
        x=gen_real_random() * world_size; // robot's x coordinate
        y=gen_real_random() * world_size; // robot's y coordinate
        orient=gen_real_random() * 2.0 * M_PI; // robot's orientation

        forward_noise=0.0; //noise of the forward movement
        turn_noise=0.0; //noise of the turn
        sense_noise=0.0; //noise of the sensing
    }

    void set(double new_x, double new_y, double new_orient)
    {
        // Set robot new position and orientation
        if (new_x<0 || new_x>=world_size)
            throw std::invalid_argument("X coordinate out of bound");
    }
};
```

```

if (new_y < 0 || new_y >= world_size)
    throw std::invalid_argument("Y coordinate out of bound");
if (new_orient < 0 || new_orient >= 2 * M_PI)
    throw std::invalid_argument("Orientation must be in [0..2pi]");

x = new_x;
y = new_y;
orient = new_orient;
}

void set_noise(double new_forward_noise, double new_turn_noise, double new_sense_noise)
{
    // Simulate noise, often useful in particle filters
    forward_noise = new_forward_noise;
    turn_noise = new_turn_noise;
    sense_noise = new_sense_noise;
}

vector<double> sense()
{
    // Measure the distances from the robot toward the landmarks
    vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));
    double dist;

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {
        dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));
        dist += gen_gauss_random(0.0, sense_noise);
        z[i] = dist;
    }
    return z;
}

Robot move (double turn, double forward)
{
    if (forward < 0)
        throw std::invalid_argument("Robot cannot move backward");

    // turn, and add randomness to the turning command
    orient = orient + turn + gen_gauss_random(0.0, turn_noise);
    orient = mod(orient, 2 * M_PI);

    // move, and add randomness to the motion command
    double dist = forward + gen_gauss_random(0.0, forward_noise);
    x = x + (cos(orient) * dist);
    y = y + (sin(orient) * dist);

    // cyclic truncate
    x = mod(x, world_size);
    y = mod(y, world_size);

    // set particle
    Robot res;
    res.set(x, y, orient);
    res.set_noise(forward_noise, turn_noise, sense_noise);

    return res;
}

string show_pose()
{
    // Returns the robot current position and orientation in a string format
    return "[x=" + to_string(x) + " y=" + to_string(y) + " orient=" + to_string(orient) + "]";
}

string read_sensors()
{
    // Returns all the distances from the robot toward the landmarks
}

```

```

vector<double> z = sense();
string readings = "[";
for (int i = 0; i < z.size(); i++) {
    readings += to_string(z[i]) + " ";
}
readings[readings.size() - 1] = ']';

return readings;
}

double measurement_prob(vector<double> measurement)
{
    // Calculates how likely a measurement should be
    double prob = 1.0;
    double dist;

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {
        dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));
        prob *= gaussian(dist, sense_noise, measurement[i]);
    }

    return prob;
}

double x, y, orient; //robot poses
double forward_noise, turn_noise, sense_noise; //robot noises

private:
    double gen_gauss_random(double mean, double variance)
    {
        // Gaussian random
        normal_distribution<double> gauss_dist(mean, variance);
        return gauss_dist(gen);
    }

    double gaussian(double mu, double sigma, double x)
    {
        // Probability of x for 1-dim Gaussian with mean mu and var. sigma
        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 * M_PI * (pow(sigma,
2)));
    }
};

// Functions
double gen_real_random()
{
    // Generate real random between 0 and 1
    uniform_real_distribution<double> real_dist(0.0, 1.0); //Real
    return real_dist(gen);
}

double mod(double first_term, double second_term)
{
    // Compute the modulus
    return first_term - (second_term)*floor(first_term / (second_term));
}

double evaluation(Robot r, Robot p[], int n)
{
    //Calculate the mean error of the system
    double sum=0.0;
    for (int i = 0; i < n; i++) {
        //the second part is because of world's cyclicity
        double dx = mod((p[i].x - r.x + (world_size / 2.0)), world_size) - (world_size / 2.0);
        double dy = mod((p[i].y - r.y + (world_size / 2.0)), world_size) - (world_size / 2.0);
        double err = sqrt(pow(dx, 2) + pow(dy, 2));
        sum += err;
    }
}

```

```

    }
    return sum / n;
}
double max(double arr[], int n)
{
    // Identify the max element in an array
    double max= 0;
    for (int i = 0; i < n; i++) {
        if (arr[i] > max)
            max = arr[i];
    }
    return max;
}

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])
{
    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format
    plt::title("MCL, step " + to_string(step));
    plt::xlim(0, 100);
    plt::ylim(0, 100);

    //Draw particles in green
    for (int i = 0; i < n; i++) {
        plt::plot({ p[i].x }, { p[i].y }, "go");
    }

    //Draw resampled particles in yellow
    for (int i = 0; i < n; i++) {
        plt::plot({ pr[i].x }, { pr[i].y }, "yo");
    }

    //Draw landmarks in red
    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {
        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");
    }

    //Draw robot position in blue
    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot
    plt::save("./Images/Step" + to_string(step) + ".png");
    plt::clf();
}

int main()
{
    //Practice Interfacing with Robot Class
    Robot myrobot;
    myrobot.set_noise(5.0, 0.1, 5.0);
    myrobot.set(30.0, 50.0, M_PI / 2.0);
    myrobot.move(-M_PI / 2.0, 15.0);
    //cout << myrobot.read_sensors() << endl;
    myrobot.move(-M_PI / 2.0, 10.0);
    //cout << myrobot.read_sensors() << endl;

    // Create a set of particles
    int n= 1000;
    Robot p[n];

    for (int i = 0; i < n; i++) {
        p[i].set_noise(0.05, 0.05, 5.0);
        //cout << p[i].show_pose() << endl;
    }

    //Re-initialize myrobot object and Initialize a measurment vector
}

```

```

myrobot = Robot();
vector<double> z;

//Iterating 50 times over the set of particles
int steps = 50;
for (int t = 0; t < steps; t++) {

    //Move the robot and sense the environment afterwards
    myrobot = myrobot.move(0.1, 5.0);
    z = myrobot.sense();

    // Simulate a robot motion for each of these particles
    Robot p2[n];
    for (int i = 0; i < n; i++) {
        p2[i] = p[i].move(0.1, 5.0);
        p[i] = p2[i];
    }

    //Generate particle weights depending on robot's measurement
    double w[n];
    for (int i = 0; i < n; i++) {
        w[i] = p[i].measurement_prob(z);
        //cout << w[i] << endl;
    }

    //Resample the particles with a sample probability proportional to the importance
    weight
    Robot p3[n];
    int index = gen_real_random() * n;
    //cout << index << endl;
    double beta = 0.0;
    double mw = max(w, n);
    //cout << mw;
    for (int i = 0; i < n; i++) {
        beta += gen_real_random() * 2.0 * mw;
        while (beta > w[index]) {
            beta -= w[index];
            index = mod((index + 1), n);
        }
        p3[i] = p[index];
    }
    for (int k = 0; k < n; k++) {
        p[k] = p3[k];
        //cout << p[k].show_pose() << endl;
    }

    //Evaluate the Error
    cout << "Step = " << t << ", Evaluation = " << evaluation(myrobot, p, n) << endl;

    //#### DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

    //Graph the position of the robot and the particles at each step
    visualization(n, myrobot, t, p2, p3);

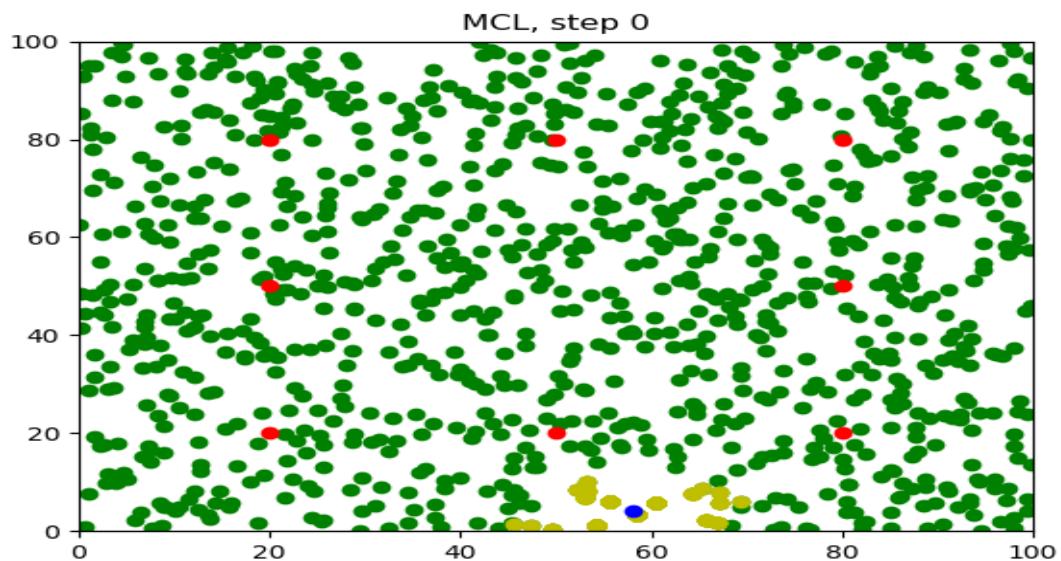
} //End of Steps loop

return 0;
}

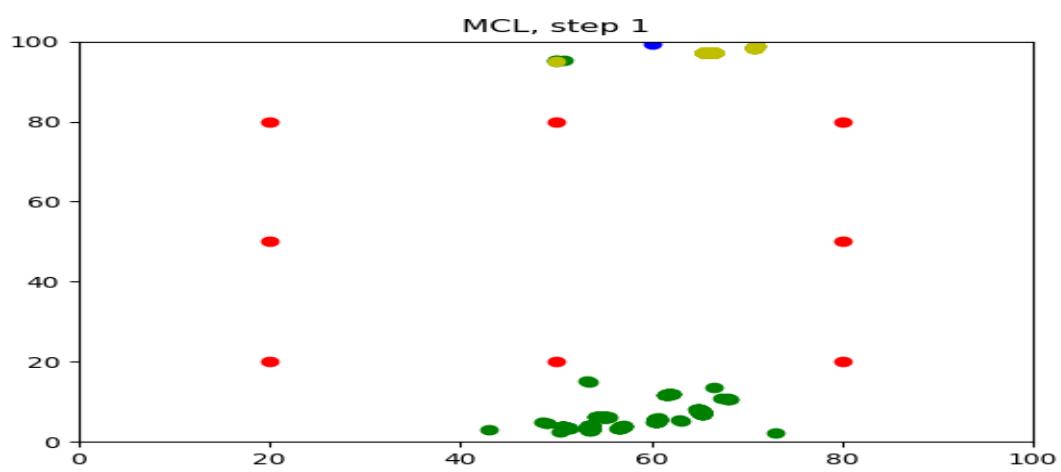
```

after running this code which you can find on my github repo [MCL Lab](#)  
It will output 50 images each representing a step of the algorithm

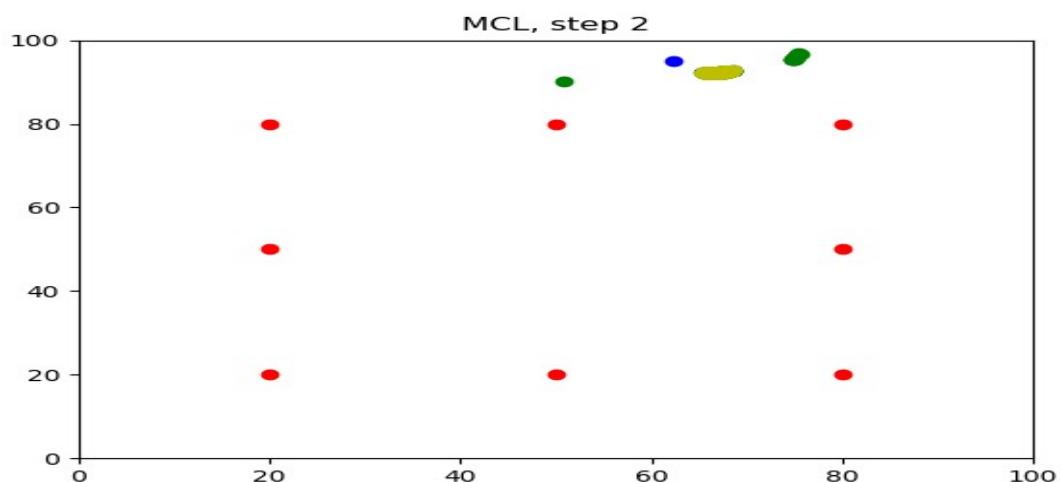
## Step 0



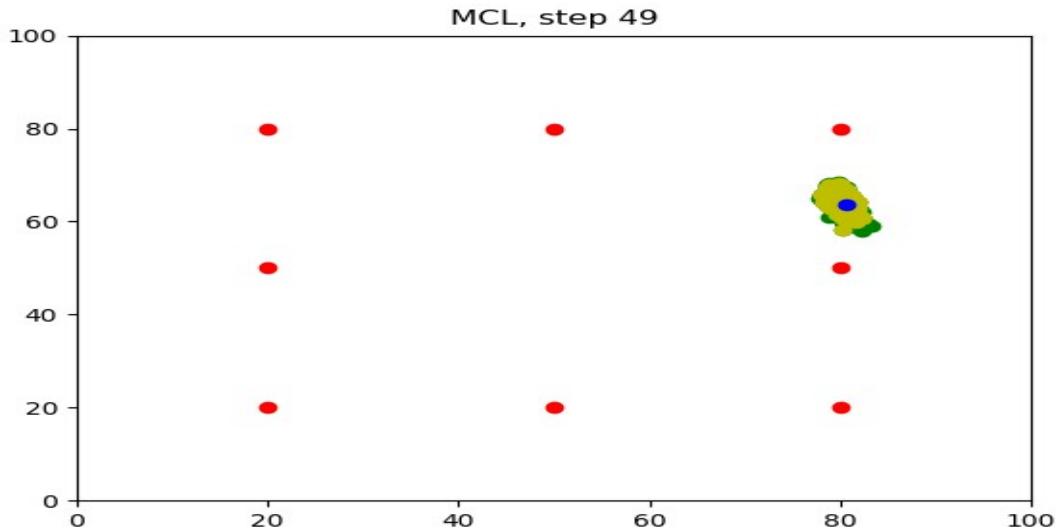
Step 1



Step 2



step 49



here we can see that the algorithm has converges (fairly quickly at step 3-4) and the particles closely follow the robot throughout it's motion.

## 10.2.2 Using the Adaptive Monte Carlo Localization with ROS

### Odometry check

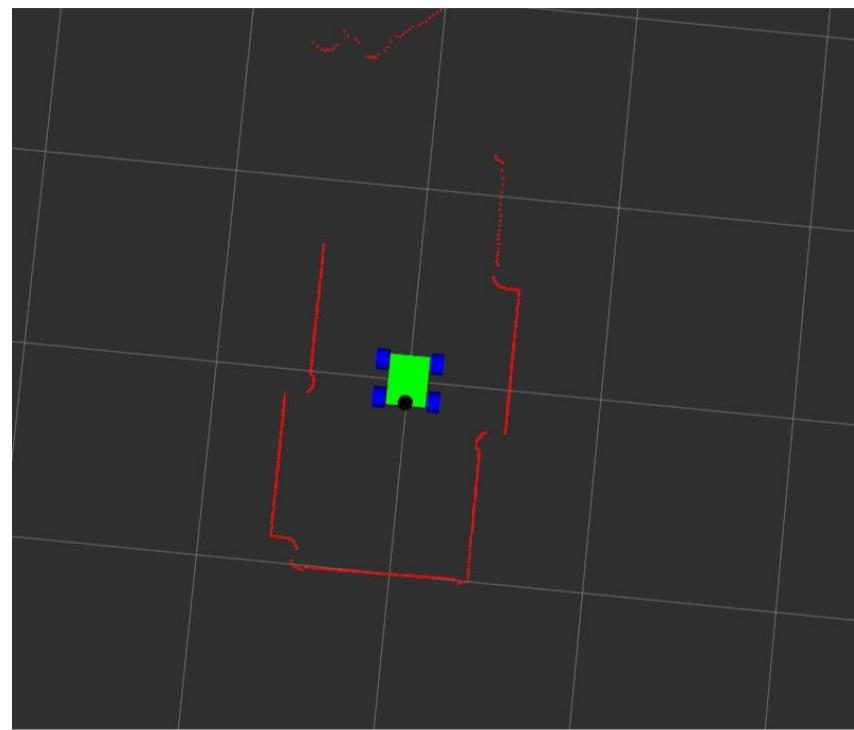
#### Correct laser Transform Frame

Before testing the odometry accuracy of the real robot, it is important to make sure that **the position of 2-D lidar** (ydlidar X4) is **accurately defined** through the **Transform Frames (TF)** relative to the robot center (also the wheels relative to the robot center), otherwise we will have severely suboptimal behaviour. For example the lidar will take measurements but we need to know from which point of the robot it does so.

Since the robot base i made is not factory made but rather a self made effort with **double-sided tape**, i didn't have blueprints for the position of the lidar and i had to measure it with a lurer and by aligning the robot with a wall starting the laser and seeing if the wall appeared parallel.

For that purpose i found a location like this, and tried to have the robot parallel to the walls. Then after fiddling with the xacro (the most important is the relative rotation or pitch of the lidar), i got a scan that look like the image below. (We need to add a LaserScan display in Rviz)

**Satisfied that my Lidar transform relative to the center of the robot was correct we can move on to the odometry test.**



## ***Odometry Source***

The most usual odometry source is with quadrature encoders on the wheels of the robot, it is important to have encoders on the DC motors in order to calculate how far the robot has moved forward or turned (a.k.a blind odometry), be careful with slippage of the wheels and with the accumulative error that every blind odometry suffers from.

Another use of the encoders is accurate control of the speed of the wheels through a PID control loop that the arduino handles.

As we discussed encoder provide a blind odometry that is accurate for small increments of distance.

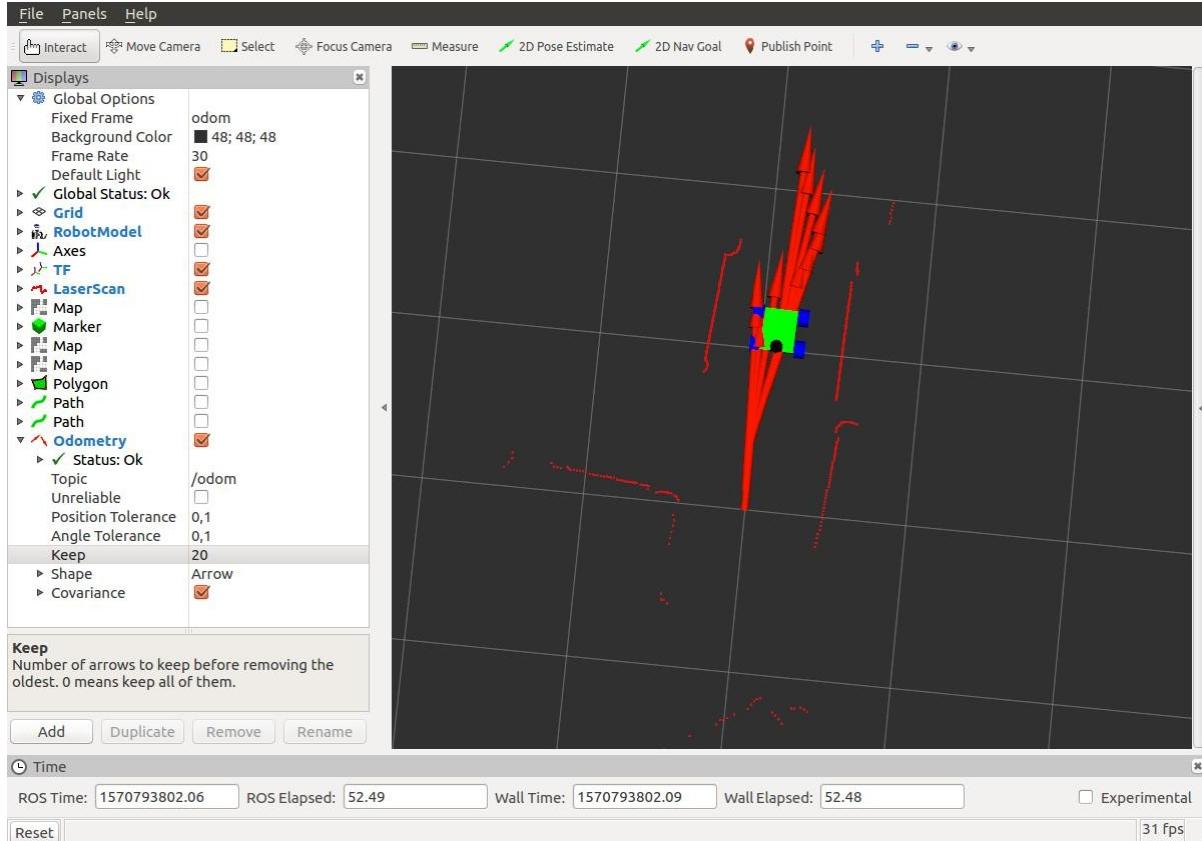
**Another option is an external source of odometry that the rf2o\_laser odometry package can provide.**

Estimation of 2D odometry based on planar laser scans. Useful for mobile robots with inaccurate base odometry.

RF2O is a fast and precise method to estimate the planar motion of a lidar from consecutive range scans. For every scanned point we formulate the range flow constraint equation in terms of the sensor velocity, and minimize a robust function of the resulting geometric constraints to obtain the motion estimate. Conversely to traditional approaches, this method does not search for correspondences but performs dense scan alignment based on the scan gradients, in the fashion of dense 3D visual odometry. The minimization problem is solved in a coarse-to-fine scheme to cope with large displacements, and a smooth filter based on the co-variance of the estimate is employed to handle uncertainty in unconstrained scenarios (e.g. corridors).

**P.S if you remember from the EKF chapter – Sensor Fusion we could fuse these two sources of odometry but i found it to be inaccurate usually we fuse odometry and imu sources, so i ended up using just the encoders for odometry source.**

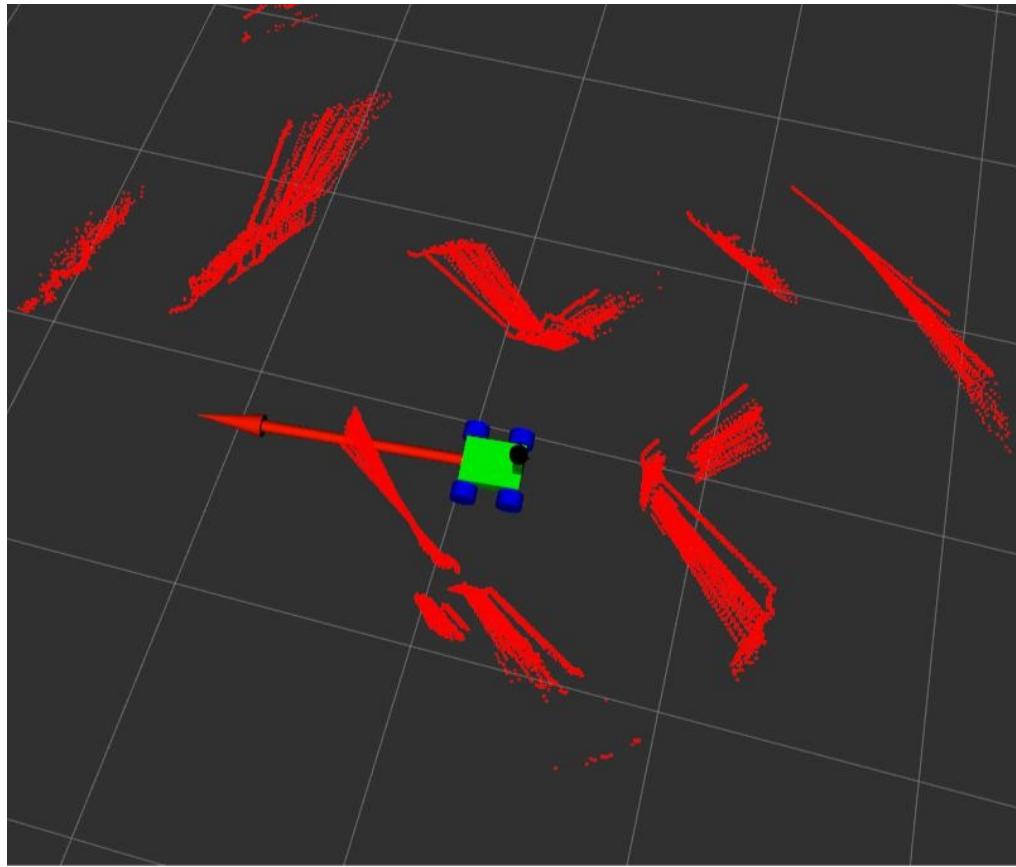
Now By setting in the RVIZ under Global options -> Fixed frame the odom frame and adding the Odometry display setting /odom topic with keep variable, by moving the robot a little with the teleop package we can visualize the odometry represented by the arrows.



### Sanity check :

A quite useful test in order to make sure your odometry is accurate is to set the decay time on the scan topic to something high like 20 seconds, this will keep displaying the scans of the previous 20 seconds. Now if we rotate the robot **the scans should ideally align with one another** or differ by a few degrees. **The more accurate odometry you have the easier will be for the AMCL to converge. Many times bad behavior is traced back to bad odometry.**

This picture depicts an average odometry, usually four wheel differential platform suffer from this, ofcourse AMCL can fix small errors like this in the odometry.



**Now we have launched all the prerequisites for launching the Adaptive Monte Carlo Package.**

## Launching the Adaptive Monte Carlo Localization package

```
<?xml version="1.0"?>
<launch>

    <!-- Scan topic -->
    <arg name="scan_topic" default="scan" />

    <!-- Map server -->
    <arg name="map_file" default="$(find lynxbot_bringup)/map/mymap.yaml"/>
    <node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" />

    <!-- Localization-->
    <node pkg="amcl" type="amcl" name="amcl">
        <rosparam file="$(find lynxbot_bringup)/config/my_amcl_params.yaml" command="load" /> <remap
            from="scan" to="$(arg scan_topic)"/>
    </node>

</launch>
```

In this launch file we first launch the **map\_server node** which offers map data as a ROS Service. It also provides the `map_saver` command-line utility, which allows dynamically generated maps to be saved to file.

### Map format

Maps are usually stored in a pair of files. The YAML file describes the map meta-data and names the image file. The image file encodes the occupancy data.

## Image format

The image describes the occupancy state of each cell of the world in the color of the corresponding pixel. In the standard configuration, whiter pixels are free, blacker pixels are occupied, and pixels in between are unknown. Color images are accepted, but the color values are averaged to a gray value.

## YAML format

The YAML format is best explained with a simple, complete example:

### home\_gluf.yaml

```
image: home_gluf.pgm
resolution: 0.05000
origin: [-15.0, -15.0, 0.0]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

## Required fields:

- **image** : Path to the image file containing the occupancy data; can be absolute, or relative to the location of the YAML file
- **resolution** : Resolution of the map, meters / pixel
- **origin** : The 2-D pose of the lower-left pixel in the map, as (x, y, yaw), with yaw as counterclockwise rotation (yaw=0 means no rotation). Many parts of the system currently ignore yaw.
- **occupied\_thresh** : Pixels with occupancy probability greater than this threshold are considered completely occupied.
- **free\_thresh** : Pixels with occupancy probability less than this threshold are considered completely free.
- **negate** : Whether the white/black free/occupied semantics should be reversed (interpretation of thresholds is unaffected)

### home\_gluf.pgm



**Next we launch the amcl node of the amcl package, with the topic /scan remapped to our scan\_topic argument, which in our case is also /scan.**  
**The adaptive refers to the capability of the algorithm to dynamically adjust the number of particles as the algorithm proceeds.**

Amcl is a probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach, which uses a particle filter to track the pose of a robot against a known map.

Algorithms used by the package: **sample\_motion\_model\_odometry**, **beam\_range\_finder\_model**, **likelihood\_field\_range\_finder\_model**, **Augmented\_MCL**, and **KLD\_Sampling\_MCL**.

### **Amcl node**

Amcl takes in a laser-based map, laser scans, and transform messages, and outputs pose estimates. On startup, amcl initializes its particle filter according to the parameters provided. Note that, because of the defaults, if no parameters are set, the initial filter state will be a moderately sized particle cloud centered about (0,0,0).

#### **Subscribed topics:**

scan (sensor msgs/LaserScan)

laser scans

tf (tf/tfMessage)

Transforms

initialpose (geometry msgs/PoseWithCovarianceStamped)

Mean and covariance with which to (re-)initialize the particle filter.

map (nav msgs/OccupancyGrid)

When the use\_map\_topic parameter is set, AMCL subscribes to this topic to retrieve the map used for laser-based localization.

#### **Published Topics:**

amcl\_pose (geometry msgs/PoseWithCovarianceStamped)

- Robot's estimated pose in the map, with

covariance. particlecloud (geometry msgs/PoseArray)

- The set of pose estimates being maintained by the filter.

tf (tf/tfMessage)

- Publishes the transform from odom (which can be remapped via the ~odom\_frame\_id parameter) to map.

#### **Services**

global\_localization (std\_srvs/Empty)

- Initiate global localization, wherein all particles are dispersed randomly through the free space in the map.

request\_nomotion\_update (std\_srvs/Empty)

- Service to manually perform update and publish updated particles.

set\_map (nav\_msgs/SetMap)

- Service to manually set a new map and pose.

## Services called

static\_map (nav\_msgs/GetMap)

- amcl calls this service to retrieve the map that is used for laser-based localization; startup blocks on getting the map from this service.

## Parameters

**my\_amcl\_params.yaml**

```

use_map_topic: true
odom_model_type: diff-corrected
odom_frame_id: odom
global_frame_id: map
base_frame_id: robot_footprint
tf_broadcast: true

min_particles: 350 # minimum allowed number of particles
max_particles: 5000 # maximum allowed number of particles
kld_err: 0.1 #0.01 : Maximum error between the true distribution and the estimated distribution.
kdl_z: 0.99 #0.99
update_min_d: 0.0 3 # Translational movement required before performing a filter update.
update_min_a: 0.03 # Rotational movement required before performing a filter update.
resample_interval: 1 # Number of filter updates required before resampling.
transform_tolerance: 0.4 #

laser_max_beams: 80 # How many evenly-spaced beams in each scan to be used when updating the filter.
laser_max_range: 9.0 # Maximum scan range to be considered; -1.0 will cause the laser's reported maximum range to be used.
laser_z_hit: 0.9 #0.5 default : Mixture weight for the z_hit part of the model.
laser_z_short: 0.05 #
laser_z_max: 0.05 #0.05
laser_z_rand: 0.5 #0.1
laser_likelihood_max_dist: 2.0
laser_sigma_hit: 0.1 # 0.2 default :Standard deviation for Gaussian model used in z_hit part of the model.
laser_lambda_short: 0.1
laser_model_type: likelihood_field

odom_alpha1: 0.3 # expected noise of rotation estimate from rotation movement
odom_alpha2: 0.1 # expected noise of rotation estimate from translation movement
odom_alpha3: 0.1 # expected noise of translation estimate from translation movement
odom_alpha4: 0.1 # expected noise of translation estimate from rotation movement
odom_alpha5: 0.1
initial_pose_x: 0

```

```

initial_pose_y: 0
initial_pose_a: 0

recovery_alpha_slow: 0.0
recovery_alpha_fast: 0.0

```

There are parameters listed in the amcl package about tuning the laser scanner model (measurement) and odometry model (motion).

To improve the localization of our robot, we increased **laser\_z\_hit** and **laser\_sigma\_hit** to incorporate higher measurement noise.

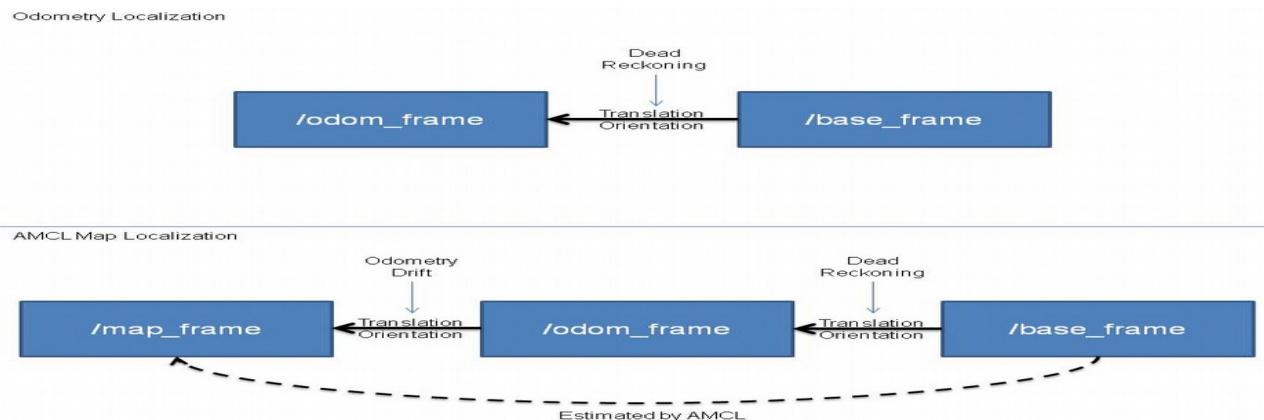
For the odometry model, moderate values were used except when the robot rotates specified by **odom\_alpha1: 0.3**, this will make the particles to be **spread more in rotation** when the robot rotates so **some of them will be correct and survive, convergence however is slower this way**.

We also increase the kdl\_err for similar reasons.

**Update\_min\_a** and **update\_min\_d** were kept small (3cm) so the robot as soon as it moves will perform a filter update, this is to make the localization process more robust.

Remember you need an accurate transform between the lidar and the center of the robot.

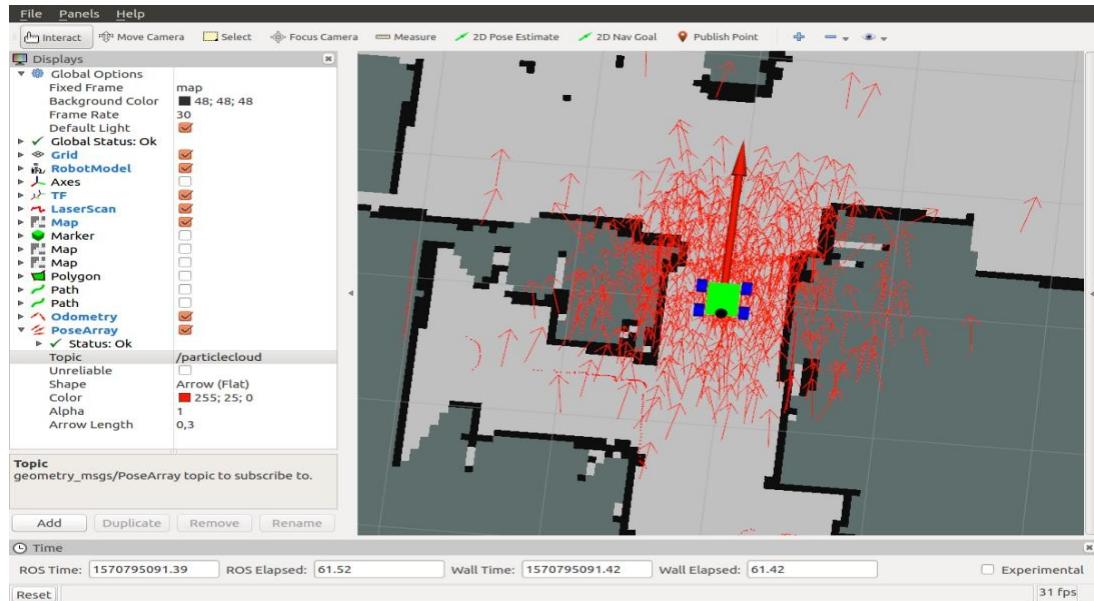
**Most of the parameters however were left as default, for a complete list please refer to the [Amcl-ROS wiki page](#).**



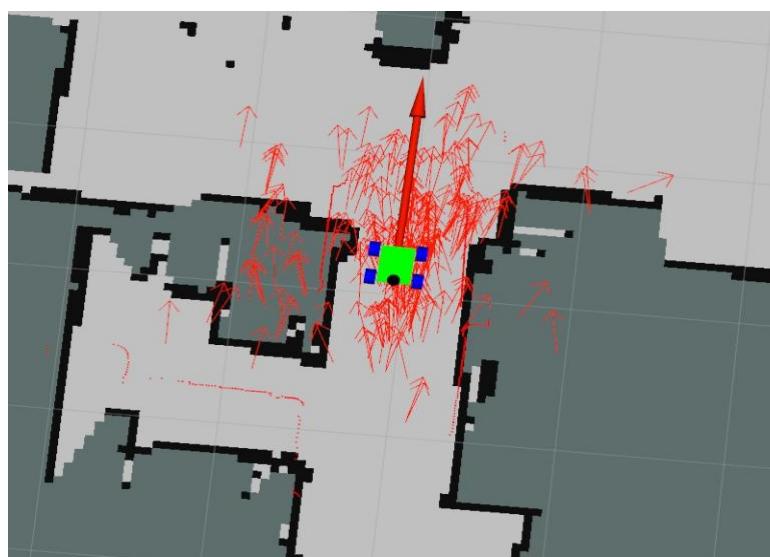
**Now let's see the results on the real robot,**

After launching the main launch file, we launch the amcl node  
We set the Fixed frame to map, and add a PoseArray display and set it to the /particlecloud topic

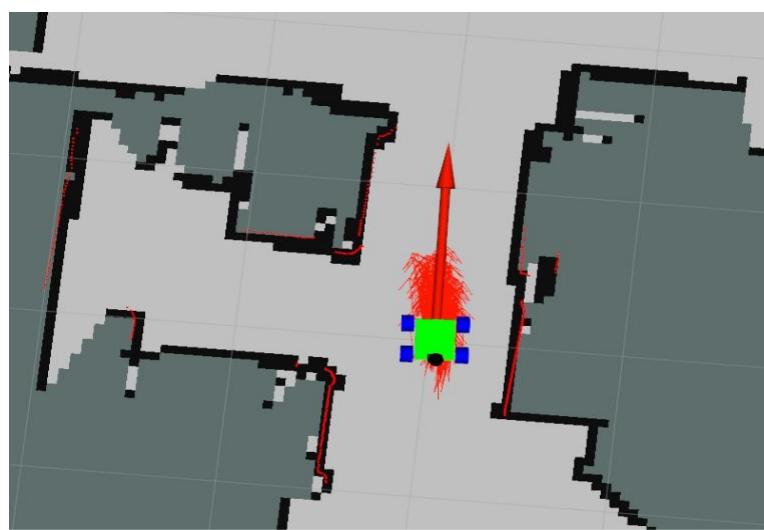
**Before the robot moves, the particles are spread out quite far around the robot's starting location.**



Next by launching the teleop node and moving the robot a little, the particles start to converge on the position of the robot.



And after some more time moving the robot back and forth, the particles have pretty much converged on the position of the robot.



**Great now we have a robot that can localize it's self inside a previous known map, and we are in a position to make it move autonomous inside the map while also avoiding dynamic obstacles, using the move\_base node that we will see in a different chapter.**

# 11. SLAM – gmapping – RTAB-Map – Autonomous SLAM

## 11.1 Programming Occupancy Grid Mapping algorithm in C++

Go to my repo [https://github.com/panagelak/RoboND-Occupancy\\_Grid\\_Mapping-](https://github.com/panagelak/RoboND-Occupancy_Grid_Mapping-) to see instructions on how to code the Occupancy Grid Mapping Algorithm in C++ with the result being outputted in the form of an image, the **measurements and poses** of the ‘robot’ are in the Data folder in .txt files. The robot is equipped with 8 range finder sensors which measurements are in the measurements.txt file.

Here is the code presented!!

```
#include <iostream>
#include <math.h>
#include <vector>
#include "src/matplotlibcpp.h" //Graph Library

using namespace std;
namespace plt = matplotlibcpp;

// Sensor characteristic: Min and Max ranges of the beams
double Zmax = 5000, Zmin = 170;
// Defining free cells(lfree), occupied cells(locc), unknown cells(10) log odds values
double l0 = 0, locc = 0.4, lfree = -0.4;
// Grid dimensions
double gridWidth = 100, gridHeight = 100;
// Map dimensions
double mapWidth = 30000, mapHeight = 15000;
// Robot size with respect to the map
double robotXOffset = mapWidth / 5, robotYOffset = mapHeight / 3;
// Defining an l vector to store the log odds values of each cell
vector<vector<double>> l((mapWidth/gridWidth,
vector<double>(mapHeight/gridHeight));

double inverseSensorModel(double x, double y, double theta, double xi, double yi,
double sensorData[])
{
    // Defining Sensor Characteristics
    double Zk, thetaK, sensorTheta;
    double minDelta = -1;
    double alpha = 200, beta = 20;

    //*****Compute r and phi*****
    double r = sqrt(pow(xi - x, 2) + pow(yi - y, 2));
    double phi = atan2(yi - y, xi - x) - theta;

    //Scaling Measurement to [-90 -37.5 -22.5 -7.5 7.5 22.5 37.5 90]
    for (int i = 0; i < 8; i++) {
        if (i == 0) {
            sensorTheta = -90 * (M_PI / 180);
        }
        else if (i == 1) {
            sensorTheta = -37.5 * (M_PI / 180);
        }
        else if (i == 2) {
            sensorTheta = -22.5 * (M_PI / 180);
        }
        else if (i == 3) {
            sensorTheta = -7.5 * (M_PI / 180);
        }
        else if (i == 4) {
            sensorTheta = 7.5 * (M_PI / 180);
        }
        else if (i == 5) {
            sensorTheta = 22.5 * (M_PI / 180);
        }
        else if (i == 6) {
            sensorTheta = 37.5 * (M_PI / 180);
        }
        else if (i == 7) {
            sensorTheta = 90 * (M_PI / 180);
        }
    }
}
```

```

        else if (i == 6) {
            sensorTheta = 37.5 * (M_PI / 180);
        }
        else if (i == 7) {
            sensorTheta = 90 * (M_PI / 180);
        }
        else {
            sensorTheta = (-37.5 + (i - 1) * 15) * (M_PI / 180);
        }

        if (fabs(phi - sensorTheta) < minDelta || minDelta == -1) {
            Zk = sensorData[i];
            thetaK = sensorTheta;
            minDelta = fabs(phi - sensorTheta);
        }
    }

//*****Evaluate the three cases*****
if (r > min((double)Zmax, Zk + alpha / 2) || fabs(phi - thetaK) > beta / 2 || Zk > Zmax || Zk < Zmin) {
    return 10;
}
else if (Zk < Zmax && fabs(r - Zk) < alpha / 2) {
    return locc;
}
else if (r <= Zk) {
    return lfree;
}
}

void occupancyGridMapping(double Robotx, double Roboty, double Robottheta, double sensorData[])
{
    //*****Code the Occupancy Grid Mapping Algorithm*****
    for (int x = 0; x < mapWidth / gridWidth; x++) {
        for (int y = 0; y < mapHeight / gridHeight; y++) {
            double xi = x * gridWidth + gridWidth / 2 - robotXOffset;
            double yi = -(y * gridHeight + gridHeight / 2) + robotYOffset;
            if (sqrt(pow(xi - Robotx, 2) + pow(yi - Roboty, 2)) <= Zmax) {
                l[x][y] = l[x][y] + inverseSensorModel(Robotx, Roboty, Robottheta, xi, yi, sensorData) - 10;
            }
        }
    }
}

void visualization()
{
    //Graph Format
    plt::title("Map");
    plt::xlim(0, (int)(mapWidth / gridWidth));
    plt::ylim(0, (int)(mapHeight / gridHeight));

    // Draw every grid of the map:
    for (double x = 0; x < mapWidth / gridWidth; x++) {
        cout << "Remaining Rows= " << mapWidth / gridWidth - x << endl;
        for (double y = 0; y < mapHeight / gridHeight; y++) {
            if (l[x][y] == 0) { //Green unkown state
                plt::plot({x}, {y}, "g.");
            }
        }
    }
}

```

```

    }
    else if (l[x][y] > 0) { //Black occupied state
        plt::plot({x}, {y}, "k.");
    }
    else { //Red free state
        plt::plot({x}, {y}, "r.");
    }
}
}

//Save the image and close the plot
plt::save("./Images/Map.png");
plt::clf();
}

int main()
{
    double timeStamp;
    double measurementData[8];
    double robotX, robotY, robotTheta;

    FILE* posesFile = fopen("Data/poses.txt", "r");
    FILE* measurementFile = fopen("Data/measurement.txt", "r");

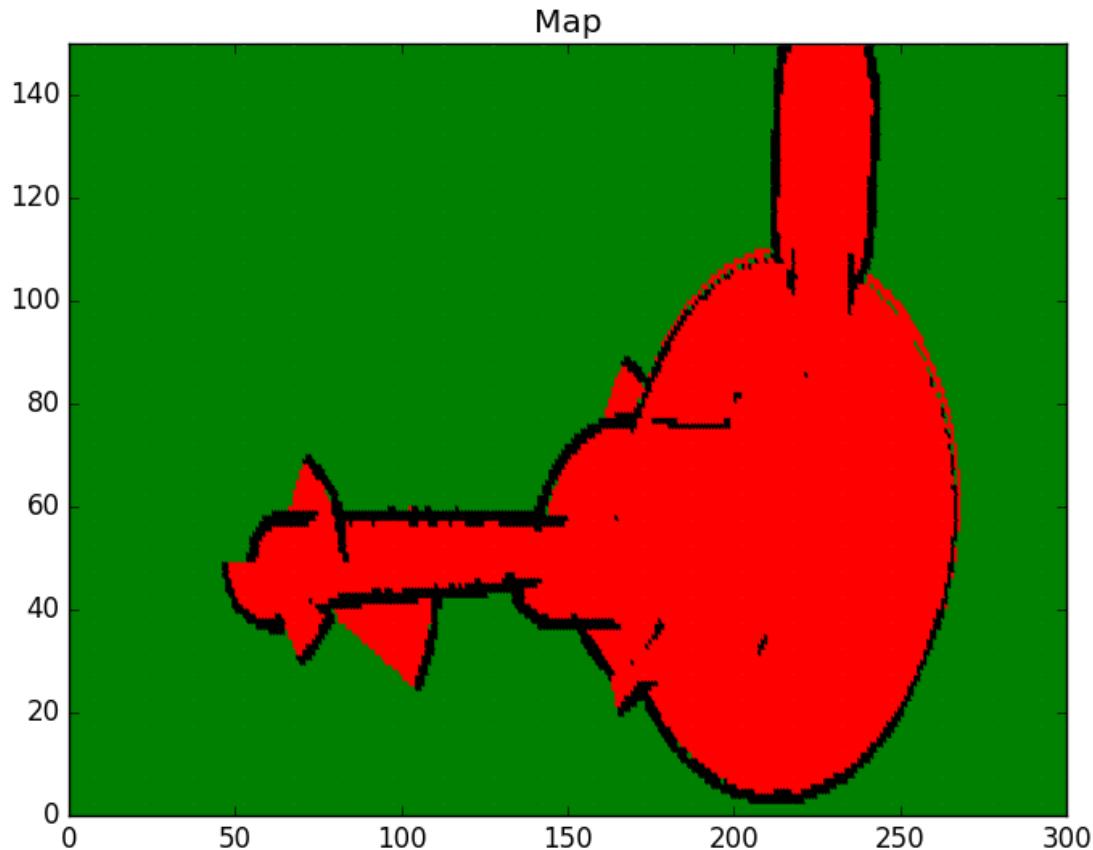
    // Scanning the files and retrieving measurement and poses at each timestamp
    while (fscanf(posesFile, "%lf %lf %lf %lf", &timeStamp, &robotX, &robotY,
&robotTheta) != EOF) {
        fscanf(measurementFile, "%lf", &timeStamp);
        for (int i = 0; i < 8; i++) {
            fscanf(measurementFile, "%lf", &measurementData[i]);
        }
        occupancyGridMapping(robotX, robotY, (robotTheta / 10) * (M_PI / 180),
measurementData);
    }

    // Visualize the map at the final step
    cout << "Wait for the image to generate" << endl;
    visualization();
    cout << "Done!" << endl;

    return 0;
}

```

After you run this code, assuming known poses from the pose.txt file and the measurements from the measurements.txt file the code will output the below image-map.



## 11.2 Grid-based FastSLAM – gmapping ROS package results

### gmapping package documentation

Access this [link](#) and go over the documentation of the gmapping ROS package

**gmapping** uses the grid-based fast SLAM algorithm, now let's take a closer look at the properties of this package. Gmapping contains a single **node, the slam\_gmapping one.**

This node subscribes to the transforms tf, relating the different frames of the laser, robot base and odometry as well as the robot laser scans.

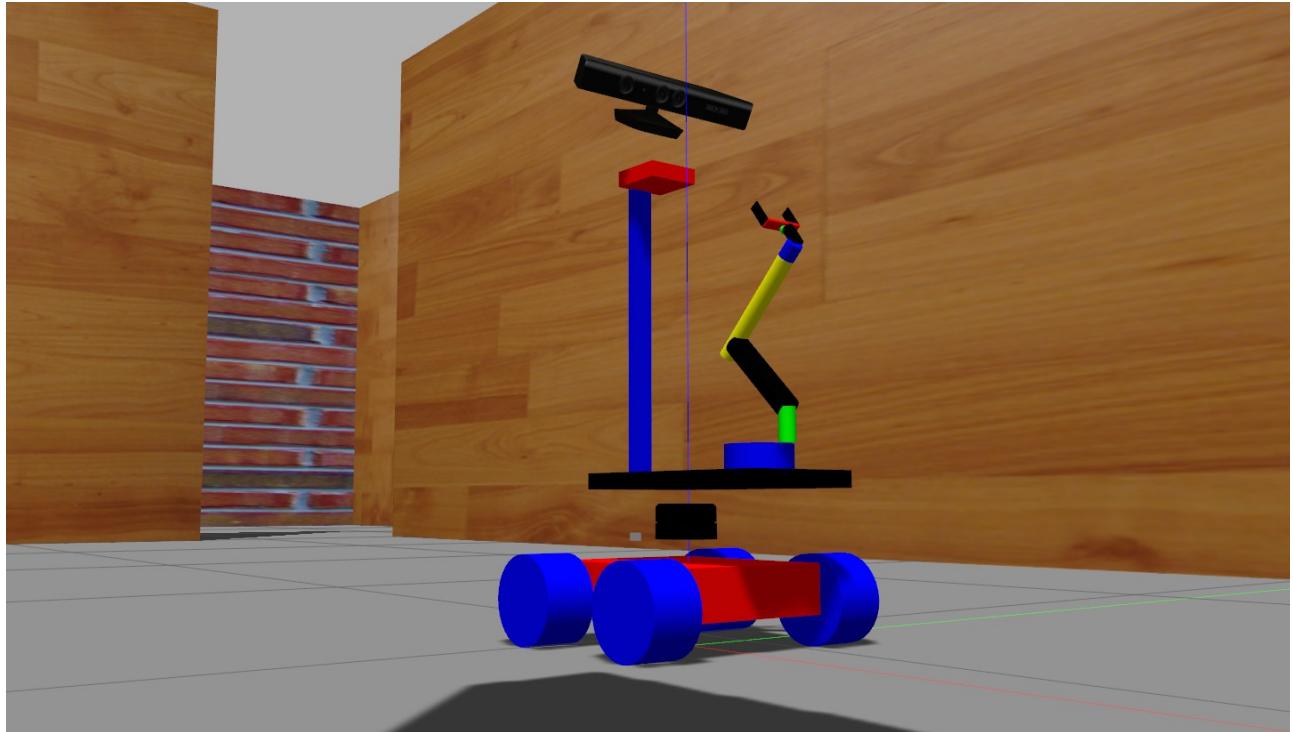
The **slam\_gmapping** node will publish the occupancy grid map characteristic, the 2D occupancy grid map and the entropy of distribution over robot pose. You can focus from the documentation on the package different topics, services and parameters.

To experiment with the gmapping package, we need to interface it with a real or simulated robot deployed in an environment. For that we'll deploy our mobile manipulator robot in a Gazebo environment-simulation and map that environment by **tele-operating** the robot using keyboard commands.

It is also possible to avoid manually teleoperating the robot manually to map the environment to use an exploratory node. This node prerequisite is that the robot is set up with the Navigation Stack so it

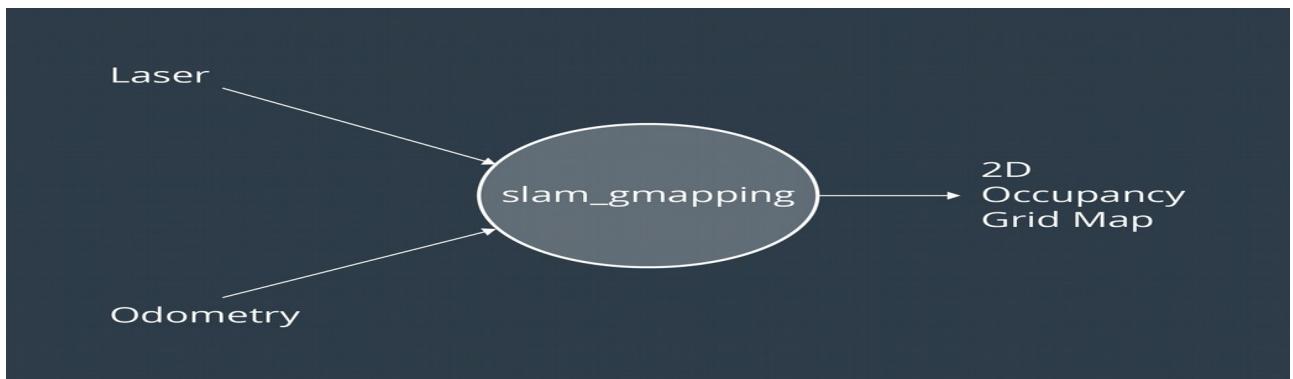
can move around manually (see path planning). Then this exploratory node can give navigation commands to greedily explore unexplored areas of the environment this is called **Autonomous SLAM**. You can find instructions for this on my github project, but the concept is the same.

Here we can see the simulated robot on the Gazebo simulation



The mobile manipulator comes equipped with a 2D laser scanner that gmapping requires to function (subscribes to laser scan measurements).

But what if your robot doesn't have a 2D laser scanner but only an RGB-D camera like Kinect? Well, all hope is not lost! You can use a node (`depthimage_to_laserscan`) to **generate laser measurements based on the depth images captured from the RGB-D camera**. However this laser measurements would be only in front of the robot and not all the way around like a proper 2D LIDAR and might be of lower accuracy. Since my robot comes equip with a 2D LIDAR we will simple use the `/scan` topic provided by it.



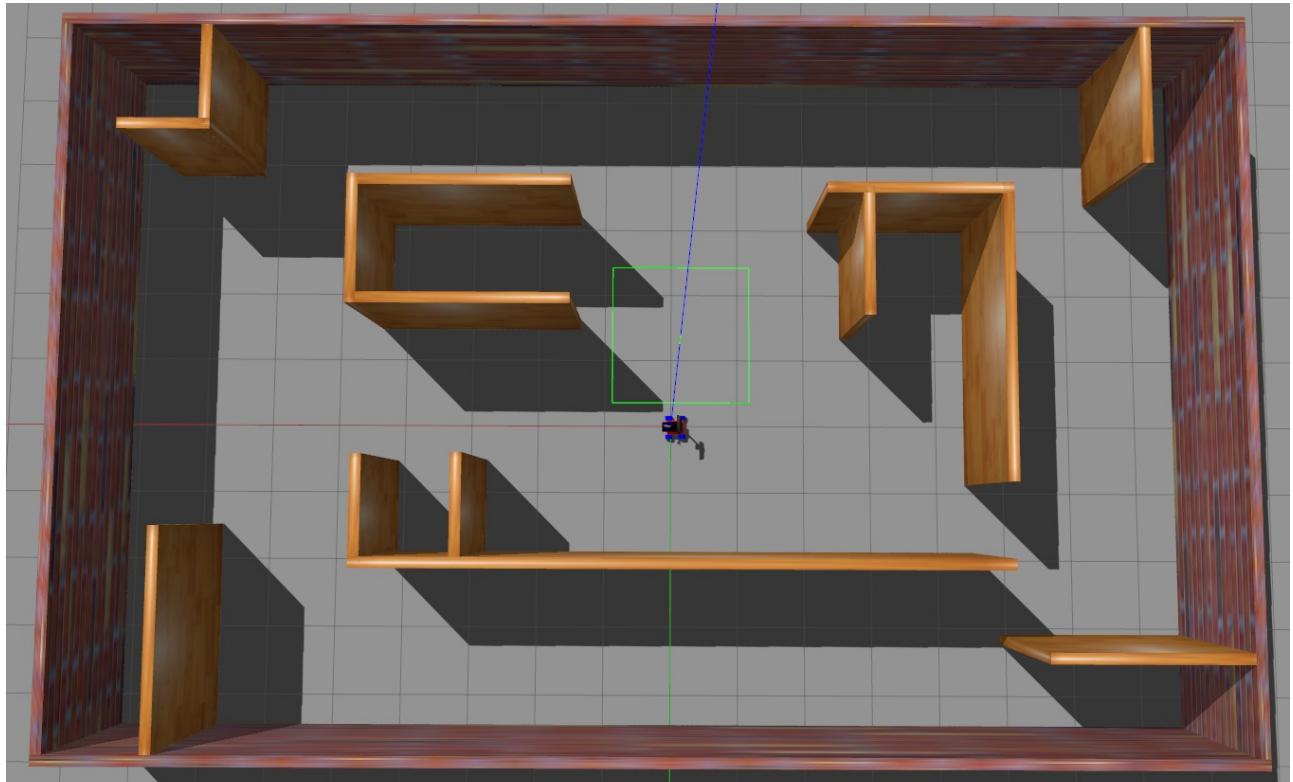
Now let's **launch the gmapping node** which provides laser based SLAM, meaning that you can feed this node with the robot laser measurements and odometry values and expect it to provide you with a 2D occupancy grid map of the environment. The map will be updated as the robot moves and collect sensory information using its laser range finder sensor.

Next we need to launch the **teleop node** to move around the robot with keyboard commands.

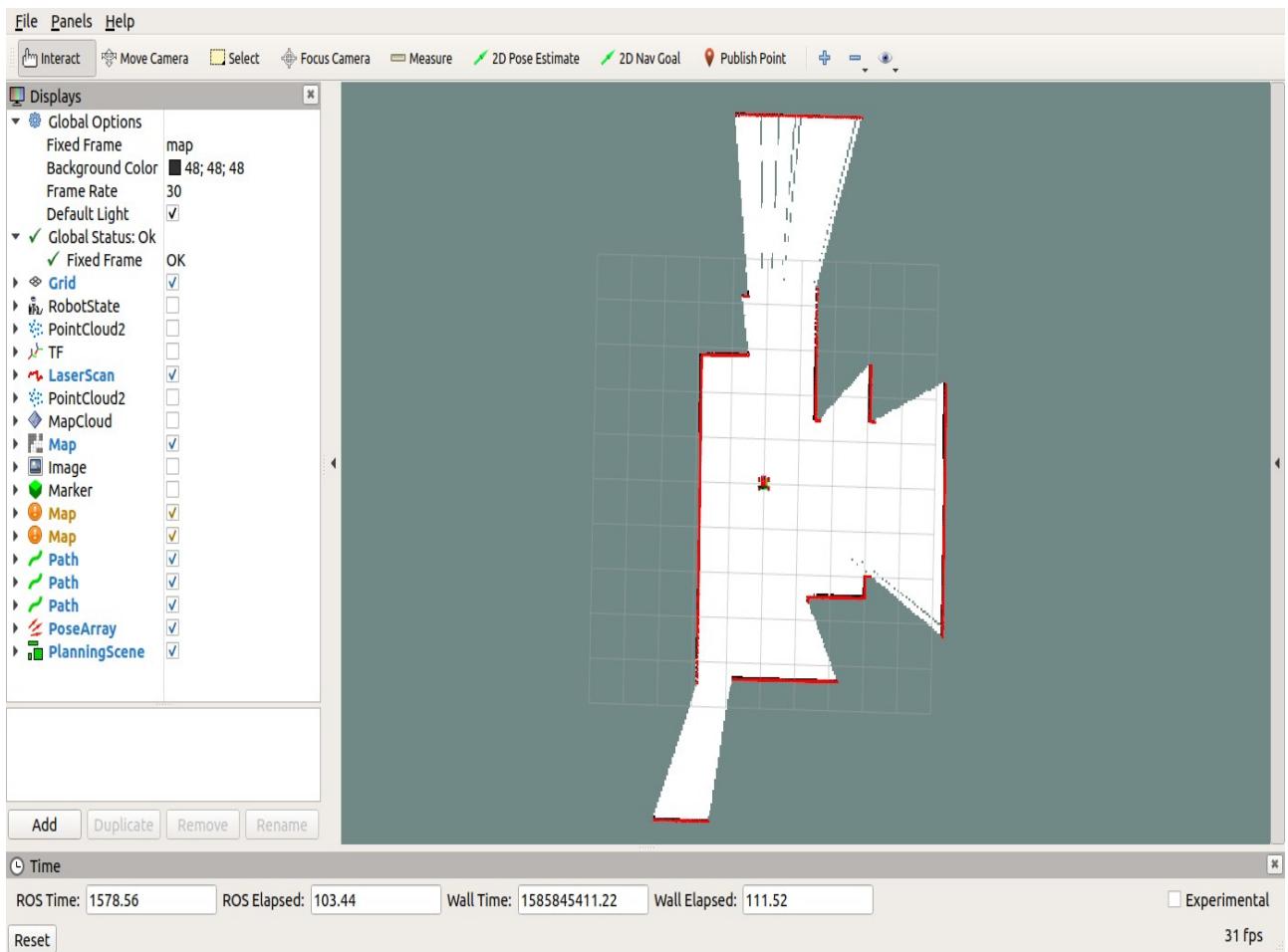
And finally **RVIZ** to visualize the map while it's being updated.

## Results

First, let's visualize the whole Gazebo world.



When RVIZ starts the robot will map first the areas that the laser 'sees' as shown in the below image.



And after teleoperating the robot around the environment either with the teleop node, or the exploratory node (requires move\_base) the completed map of the environment can be seen below.



This map then can be saved for later use with the AMCL node to localize the robot as we have seen or autonomous operate the robot as we will see in path planning.

With the [map\\_server](#) you can load and save maps. Running `map_server` will generate the **map.pgm** and the **map.yaml** files:

To save the map

```
$ rosrun map_server map_saver -f mymap
```

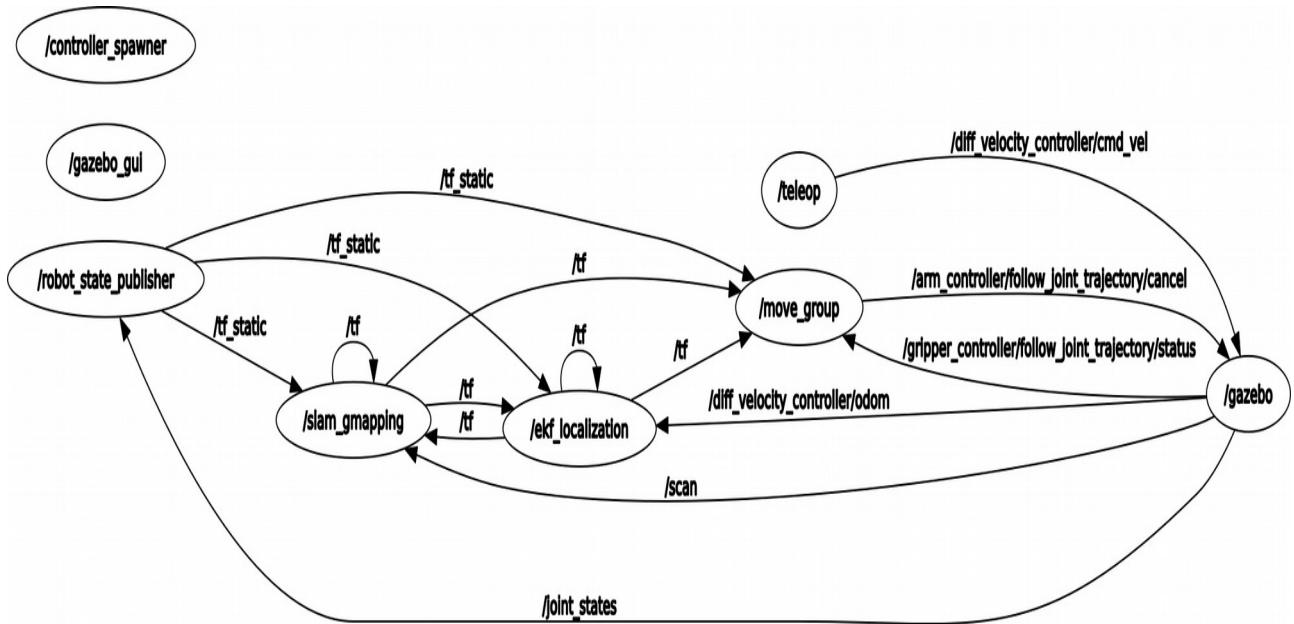
### **1- map.pgm: Picture of the map in occupancy grid representation**

- *White pixels*: Free cells
- *Black pixels*: Occupied cells
- *Gray pixels*: Unknown state

### **2- map.yaml: The map metadata**

- *image*: Map name
- *resolution*: Resolution of the map (meters/pixel)
- *origin*: Pose of the lower-left pixel in the map ( $x, y, \Theta$ )
- *Occupied\_thresh*: Cell is considered occupied if its probability is greater than this threshold.
- *free\_thresh*: Cell is considered unoccupied or free if its probability is less than this threshold.
- *negate*: This value will check whether the notation of black colored cell=occupied and white colored cell = free should be preserved

The nodes currently running in the system, this robot requires `move_group` node, also it uses the `ekf_localization` node so the graph is a little more complex is



## gmapping parameters

The parameters of this node were mostly left on their default values since they provide good results but in general, it's essential to tune them in order to get a 100% accurate map. These parameters are all listed under the gmapping documentation, where you can look at them yourself.

base\_frame: robot\_foot

odom\_frame: odom\_combined

map\_update\_interval: 5.0

maxUrangle: 16.0

sigma: 0.05

kernelSize: 1

lstep: 0.05

1step: 0.05  
astep: 0.05

astep: 0.05  
iterations: 5

Iterations.

Isigma: 0  
again: 3

ogain: 3.  
lskip: 0

Iskip: 0  
Gmax: 0.1

srr: 0.1

srt: 0.2

str: 0.1

stt: 0.2

linearUpdate: 0.5

angularUpdat

temporalUpda

resampleThr

## particle

xmin: -5

ymin: -50.0

xmax: 50.0

xmax: 50.0

ymax: 50.0  
delta: 0.05

llsamplerange: 0.01

llsamplerange: 0.0  
llsamplestep: 0.01

lasamplestep: 0.01  
lasamplerange: 0.005

lasampleRange: 0.00  
lasampleStep: 0.005

For example, you might try, reducing the angularUpdate and linearUpdate values so the map gets updated for smaller ranges of movements, reducing the x and y limits, which represent the initial map size, increasing the number of particles. You can try tweaking these parameters and/or any other parameter you think should be changed.

### 11.3 Real-Time Appearance-based Mapping, RTAB-Map package – experimental results

As we have seen in the theory RTAB-Map is a **3D SLAM** technique, for that we will use a **different gazebo world with more 3d features / objects** in order for RTAB-Map to be able to map the environment **find loop closures** and be **more interesting**.

Here are some pictures of the simulated world with 3D features.





RTAB-Map is one of the best solution for SLAM to develop robots that can map environments in 3D. These considerations come from RTAB-Map's speed and memory management, its custom developed tools for information analysis and, most importantly, the quality of the documentation. Being able to leverage RTAB-Map with your own robots will lead to a solid foundation for mapping and localization. Now we will be using the **rtabmap\_ros package**, which is a **ROS wrapper** (API) for interacting with rtabmap. Keep this in mind when looking at the relative documentation.

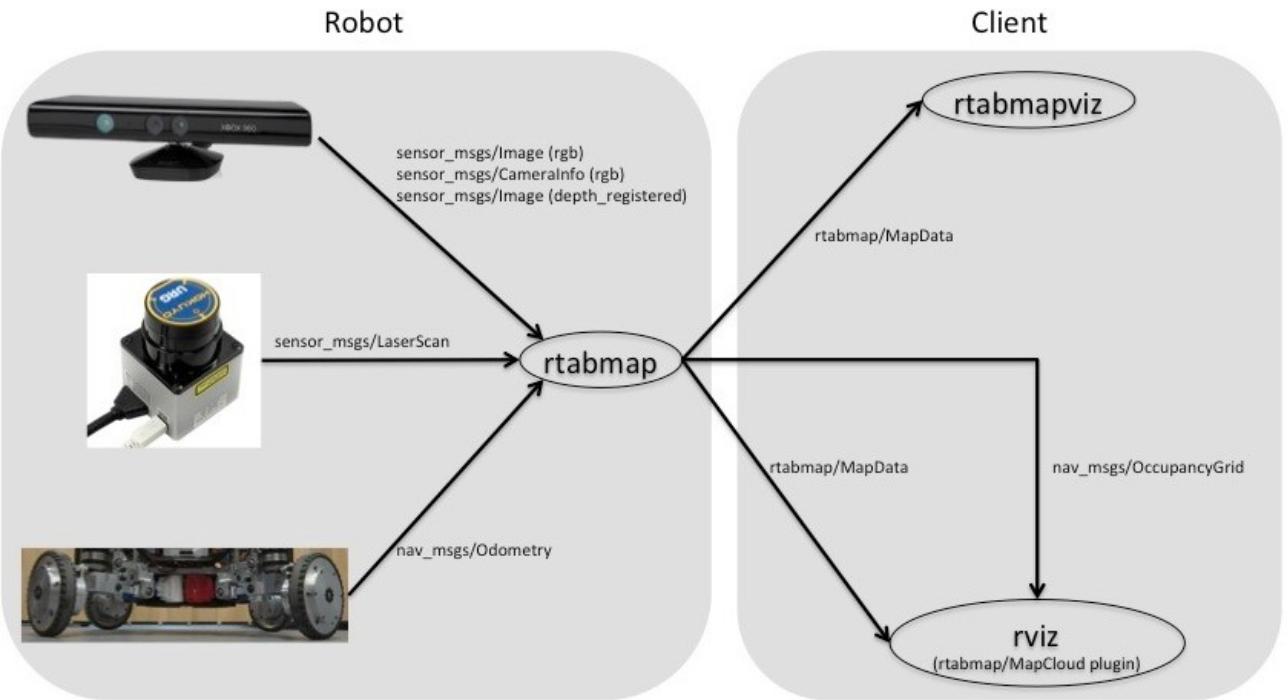
The recommended configuration for RTAB-Map is as follows:

- 2D laser which publishes LaserScan [sensor\\_msgs/LaserScan messages](#)
- Odometry (IMU, wheel encoders, ...) which publishes [nav\\_msgs/Odometry](#) messages
- A calibrated RGB-D sensor

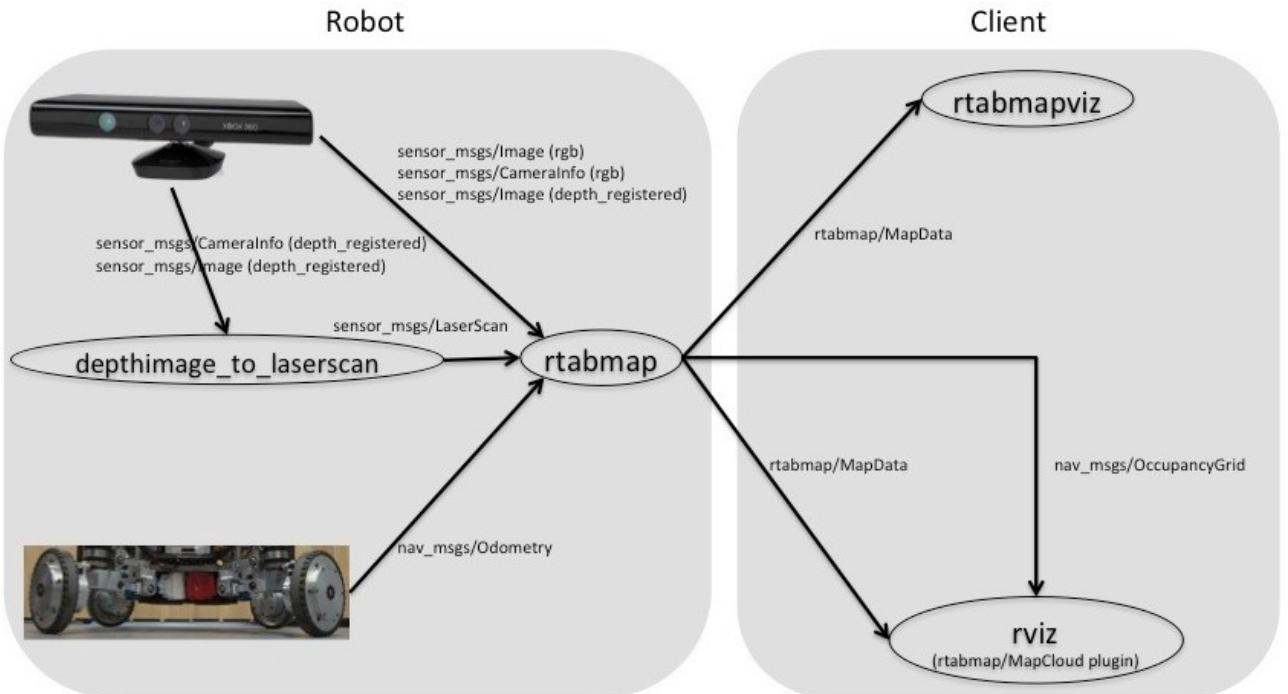
You may notice that a 2D laser scanner is listed and although it is recommended, it is not required. We will touch more on that shortly.

### RTAB-Map Quick Sensor Overview

When building our ROS package we will need to make sure that we are publishing the right information for RTAB-Map to successfully work. In the image below we can get a visual overview of the high level connections enabling both mapping and localization.



If we don't have a laser scanner we can use the [`depthimage\_to\_laserscan`](#) package to simulate a /scan topic from the kinect. This remapping can be viewed below.



This is a convenient solution for those without a laser scanner on their robot.

In order to launch RTAB-Map in ros we can use the following launch file.

```

<launch>
  <group ns="rtabmap">

    <!-- Use RGBD synchronization -->

    <node pkg="nodelet" type="nodelet" name="rgbd_sync" args="standalone
rtabmap_ros/rgbd_sync" output="screen">

      <remap from="rgb/image" to="/camera/rgb/image_raw"/>
      <remap from="depth/image" to="/camera/depth_registered/image_raw"/>
      <remap from="rgb/camera_info" to="/camera/rgb/camera_info"/>
      <remap from="rgbd_image" to="rgbd_image"/>

      <param name="approx_sync" value="true"/>

    </node>

    <!-- RTAB-Map node -->
    <node name="rtabmap" pkg="rtabmap_ros" type="rtabmap" output="screen" args="--delete_db_on_start">

      <remap from="odom" to="/diff_velocity_controller/odom"/>
      <remap from="scan" to="/scan"/>
      <remap from="rgbd_image" to="rgbd_image"/>

      <remap from="/rtabmap/grid_map" to="/map"/>

      <param name="frame_id" type="string" value="robot_footprint"/>
      <param name="subscribe_depth" type="bool" value="false"/>
      <param name="subscribe_rgbd" type="bool" value="true"/>
      <param name="subscribe_scan" type="bool" value="true"/>
      <param name="wait_for_transform" type="bool" value="true"/>
      <param name="queue_size" type="int" value="10"/>
      <param name="Reg/Force3DoF" type="string" value="true"/>
    <!-- RTAB-Map's parameters -->
      <param name="RGBD/NeighborLinkRefining" type="string" value="true"/>
      <param name="RGBD/ProximityBySpace" type="string" value="true"/>
      <param name="RGBD/AngularUpdate" type="string" value="0.1"/>
      <param name="RGBD/LinearUpdate" type="string" value="0.1"/>
      <param name="RGBD/OptimizeFromGraphEnd" type="string" value="false"/>
      <param name="Grid/FromDepth" type="string" value="false"/>
    <!-- occupancy grid from lidar -->
      <param name="Reg/Strategy" type="string" value="1"/>
    <!-- 1=ICP --><!-- ICP parameters -->
      <param name="Icp/VoxelSize" type="string" value="0.03"/>
      <param name="Icp/MaxCorrespondenceDistance" type="string" value="0.12"/>
    </node>
  </group>
</launch>

```

At this point, you could assemble your components to build your ROS package, but you may be met with difficulties in getting it to run. There may be a mis-named topic or an improper linkage. So what can we do? Well, refer to the debugging packages in order to find errors and correctly setting up your robot!

Further Resources:

[RTAB-Map Parameter Tutorial](#)

[List of RTAB-Map Parameters](#)

Most of the parameters well left per default with some minor changes!

After launching RTAB-Map we can teleoperate the robot around in order to map the environment.

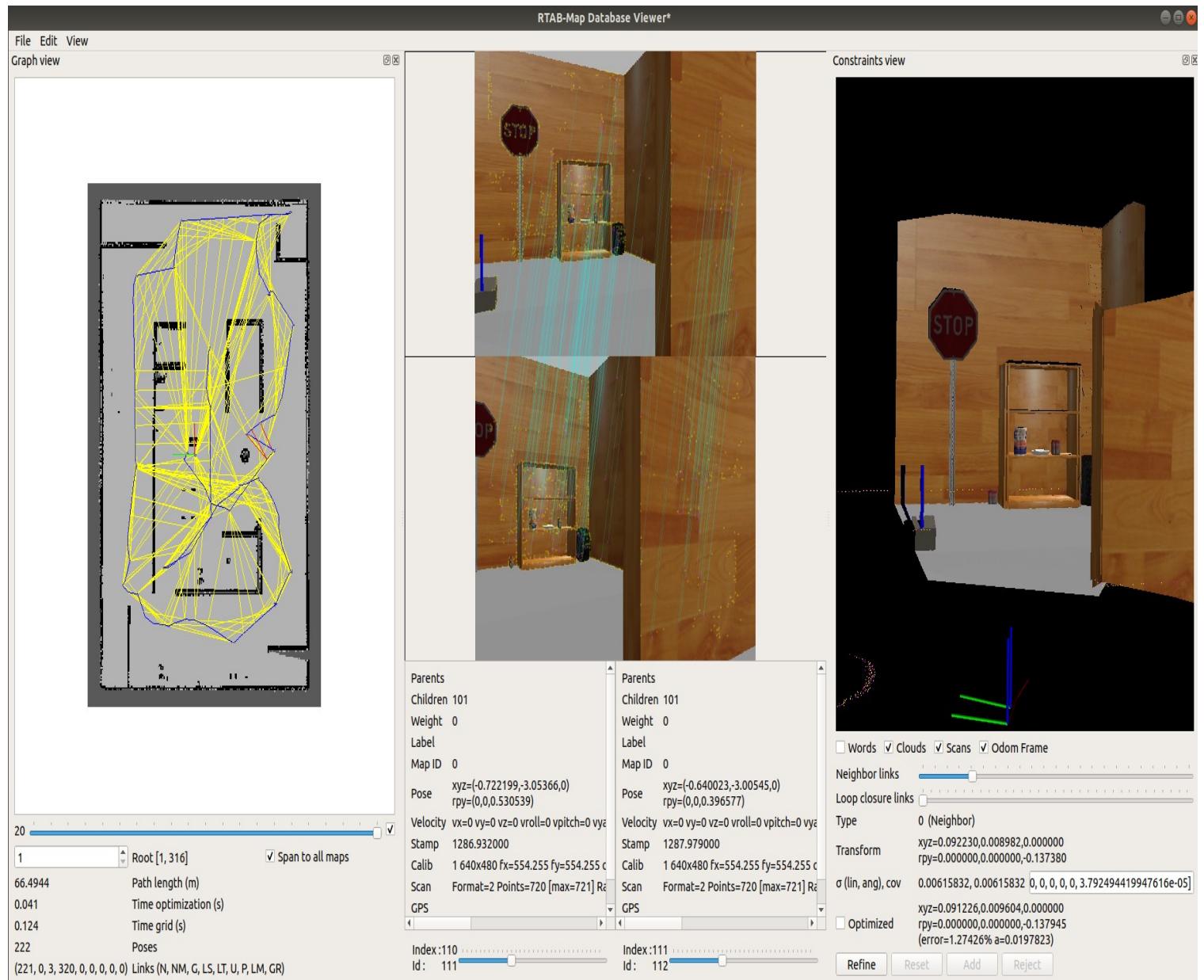
If we want to map the whole environment the robot needs to see **all perspectives** of the environment, also we can **revisit some places to detect loop closures and for better results**.

When we are finished mapping the environment RTAB-Map will **save the database in a rtabmap.db file**.

## Database Analysis

The **rtabmap-databaseViewer** is a great tool for exploring your database when you are done generating it. It is isolated from ROS and allows for complete analysis of your mapping session.

This is how you will check for loop closures, generate 3D maps for viewing, extract images, check feature mapping rich zones, and much more!



## Let's start by opening our mapping database:

We can do this like so:

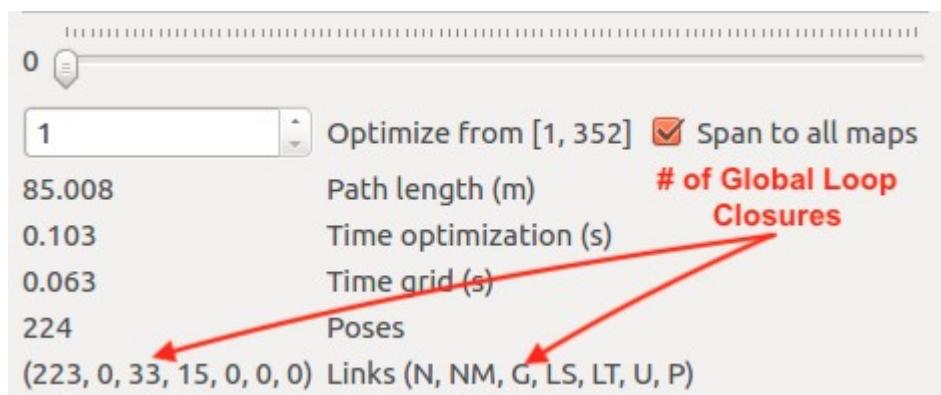
```
rtabmap-databaseViewer ~/ros/rtabmap.db
```

Once open, we will need to add some windows to get a better view of the relevant information, so:

- Say yes to using the database parameters
- View -> Constraint View
- View -> Graph View

Those options are enough to start, as there are many features built into the database viewer!

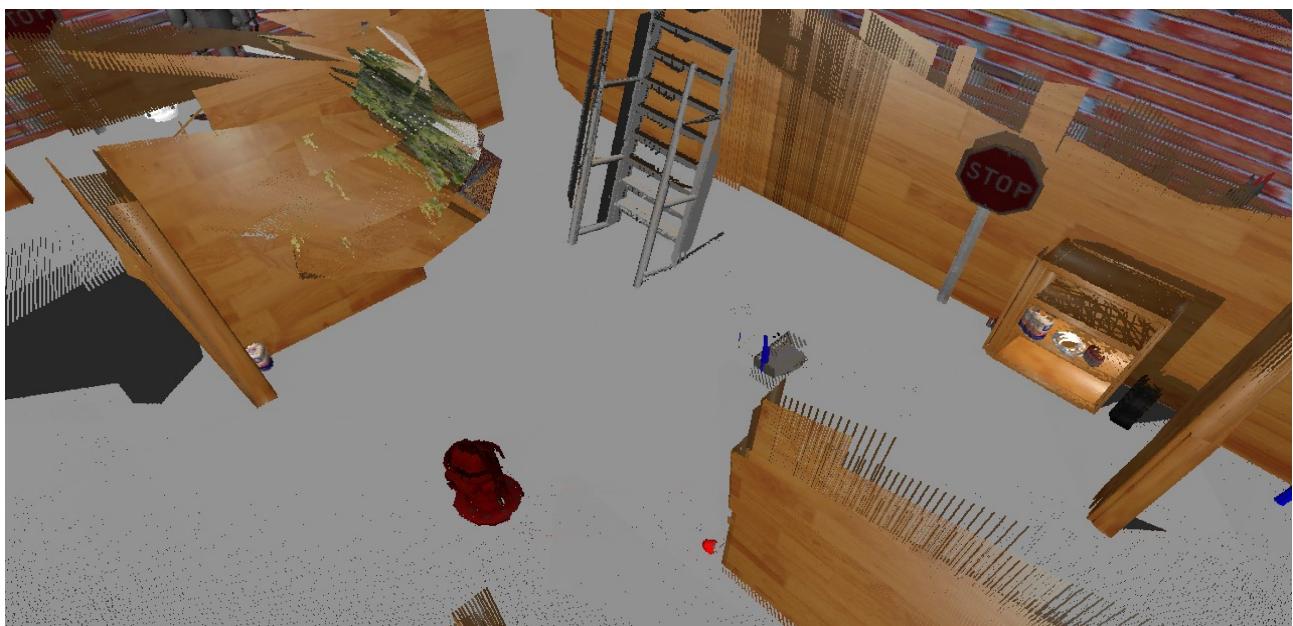
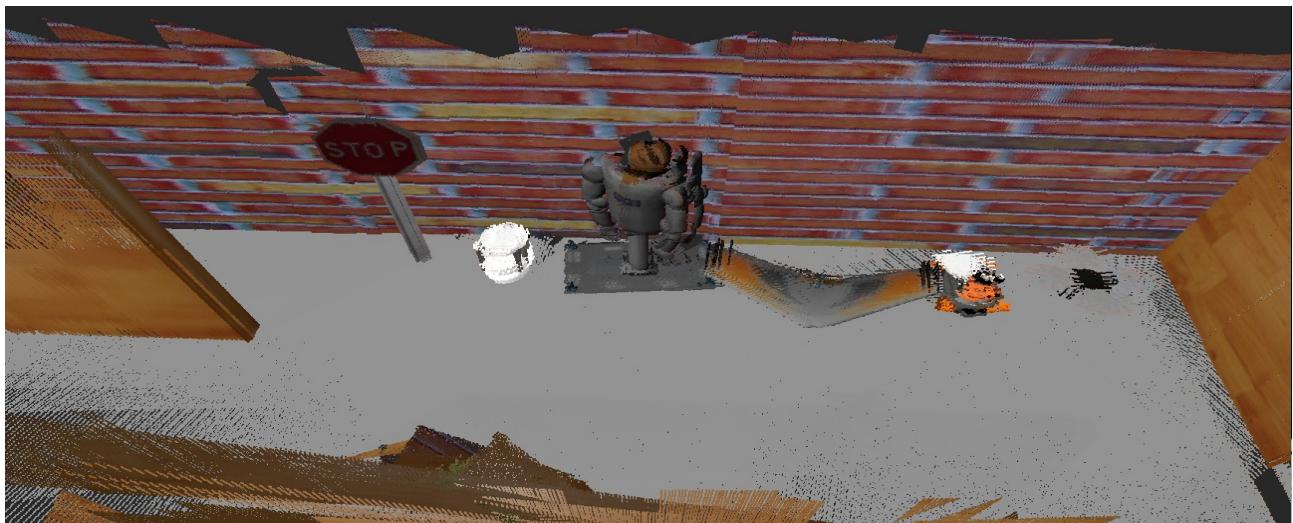
Let's talk about what we are seeing in the above image. On the left, we have our 2D grid map in all of its updated iterations and the path of our robot. In the middle we have different images from the mapping process. Here we can scrub through images to see all of the features from our detection algorithm. These features are in yellow. Then, what is the pink, you may ask? The pink indicates where two images have features in common and this information is being used to create neighboring links and loop closures! Finally, on the right we can see the constraint view. This is where we can identify where and how the neighboring links and loop closures were created.



We can see the number of loop closures in the bottom left. The codes stand for the following: Neighbor, Neighbor Merged, Global Loop closure, Local loop closure by space, Local loop closure by time, User loop closure, and Prior link.

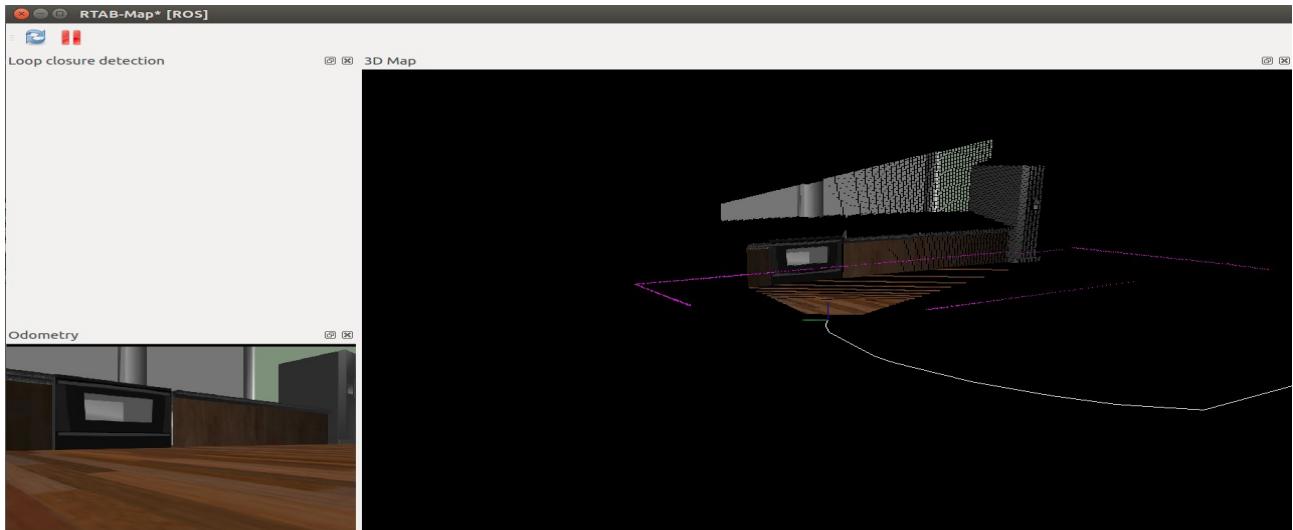
When it comes time to design our own environment, this tool can be a good resource for checking if the environment is feature-rich enough to make global loop closures. A good environment has many features that can be associated in order to achieve loop closures.

Then we can extract and visualize the whole point cloud here are some examples.



## Real time mapping visualization

Another tool that we can use is **rtabmapviz**, which is an additional node for real time visualization of feature mapping, loop closures, and more. It's not recommended to use this tool while mapping in simulation due to the computing overhead. **rtabmapviz** is great to deploy on a real robot during live mapping to ensure that we are getting the necessary features to complete loop closures.



To launch, add the following code to your **mapping.launch** file:

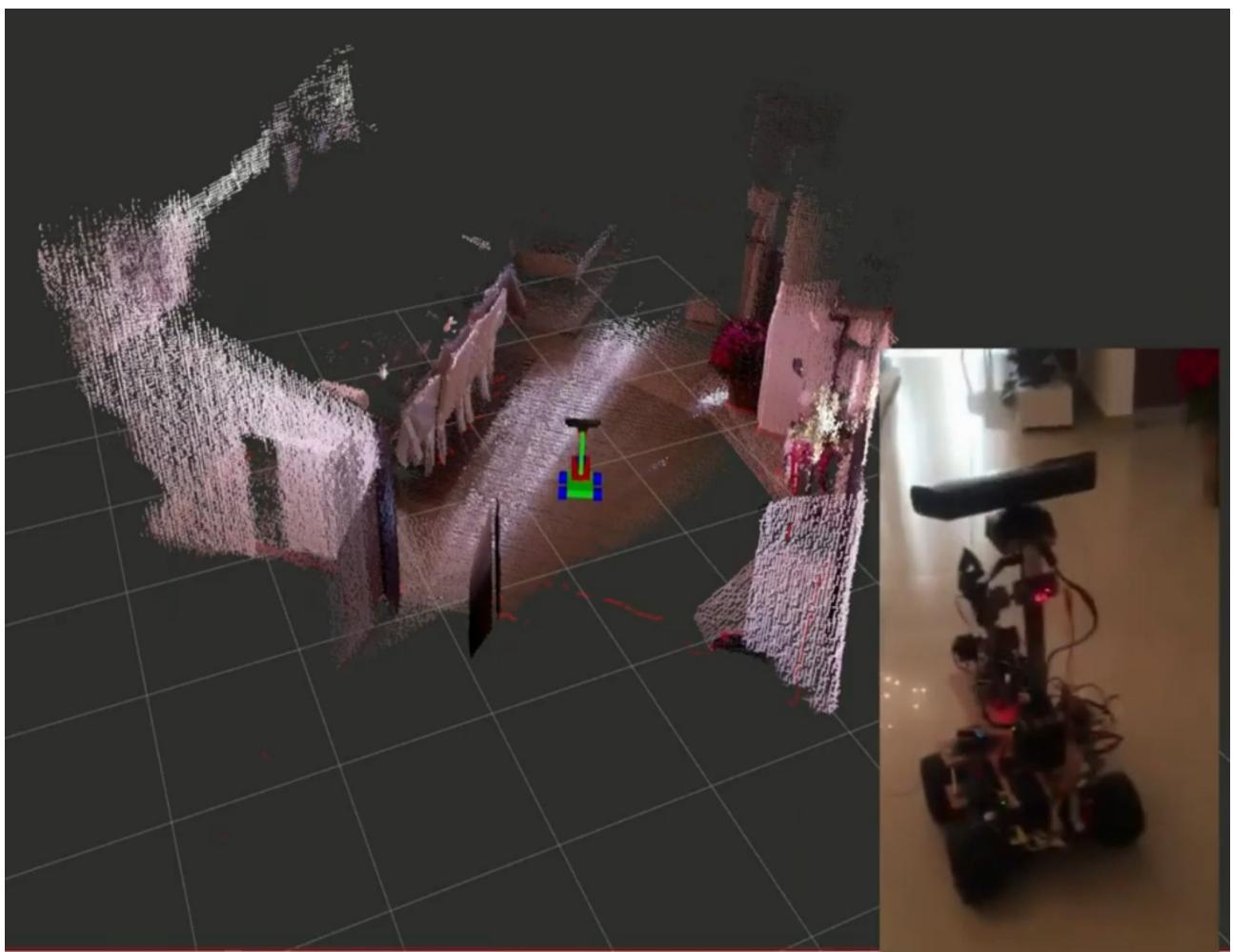
```
<!-- visualization with rtabmapviz -->
<node pkg="rtabmap_ros" type="rtabmapviz" name="rtabmapviz" args="-d $(find rtabmap_ros)/launch/config/rgbd_gui.ini" output="screen">
  <param name="subscribe_depth" type="bool" value="true"/>
  <param name="subscribe_scan" type="bool" value="true"/>
  <param name="frame_id" type="string" value="base_footprint"/>

  <remap from="rgb/image" to="$(arg rgb_topic)"/>
  <remap from="depth/image" to="$(arg depth_topic)"/>
  <remap from="rgb/camera_info" to="$(arg camera_info_topic)"/>
  <remap from="scan" to="/scan"/>
</node>
```

This will launch the **rtabmapviz GUI** and provide you with realtime feature detection, loop closures, and other relevant information to the mapping process.

Ofcourse we can use **RVIZ** for the same purpose with the MapCloud display but it is not as specialized as this tool.

**Below we can see our real robot mapping a real environment, specifically my home which you now know how it looks like!!!**



# RTAB-Map Localization - Robot Kidnap

## Localization

If you desire to perform localization, there are only a few changes you need to make. You can start by duplicating your `mapping.launch` file and calling it `localization.launch`.

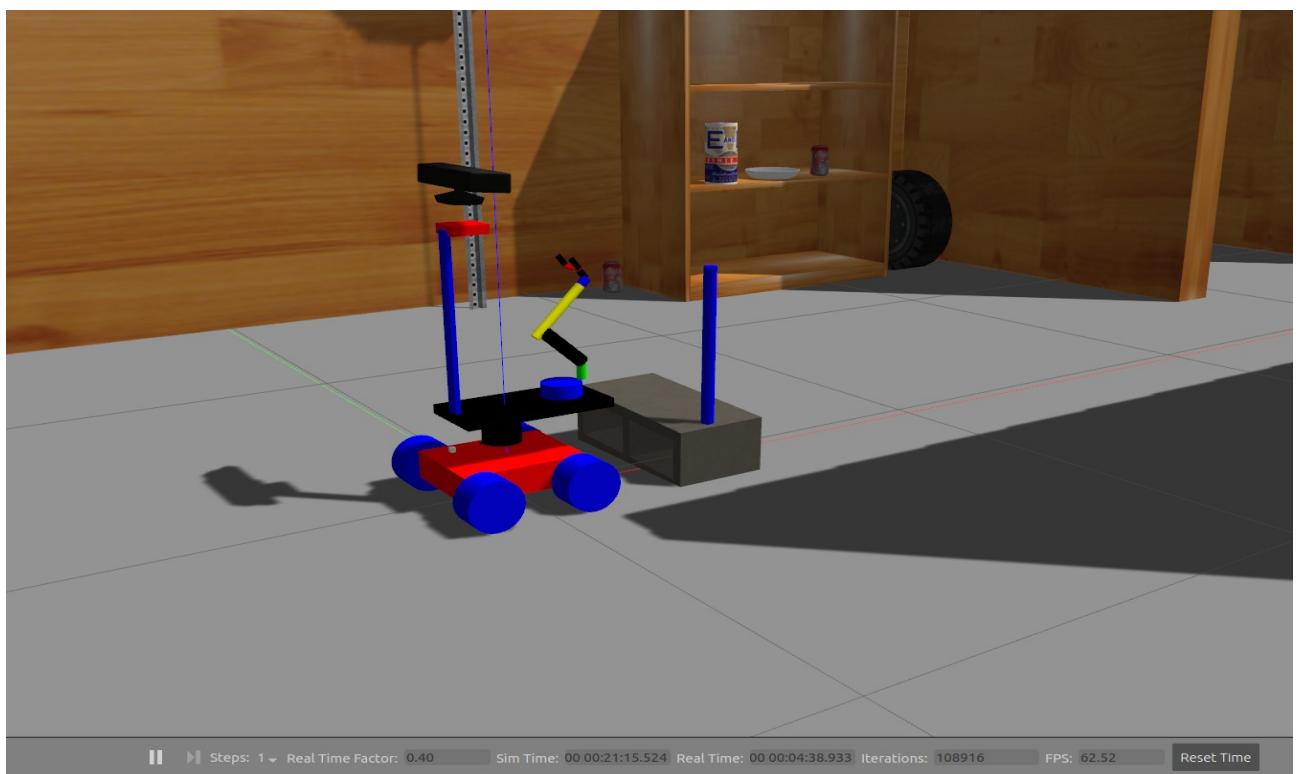
The following changes also need to be made to the `localization.launch` file:

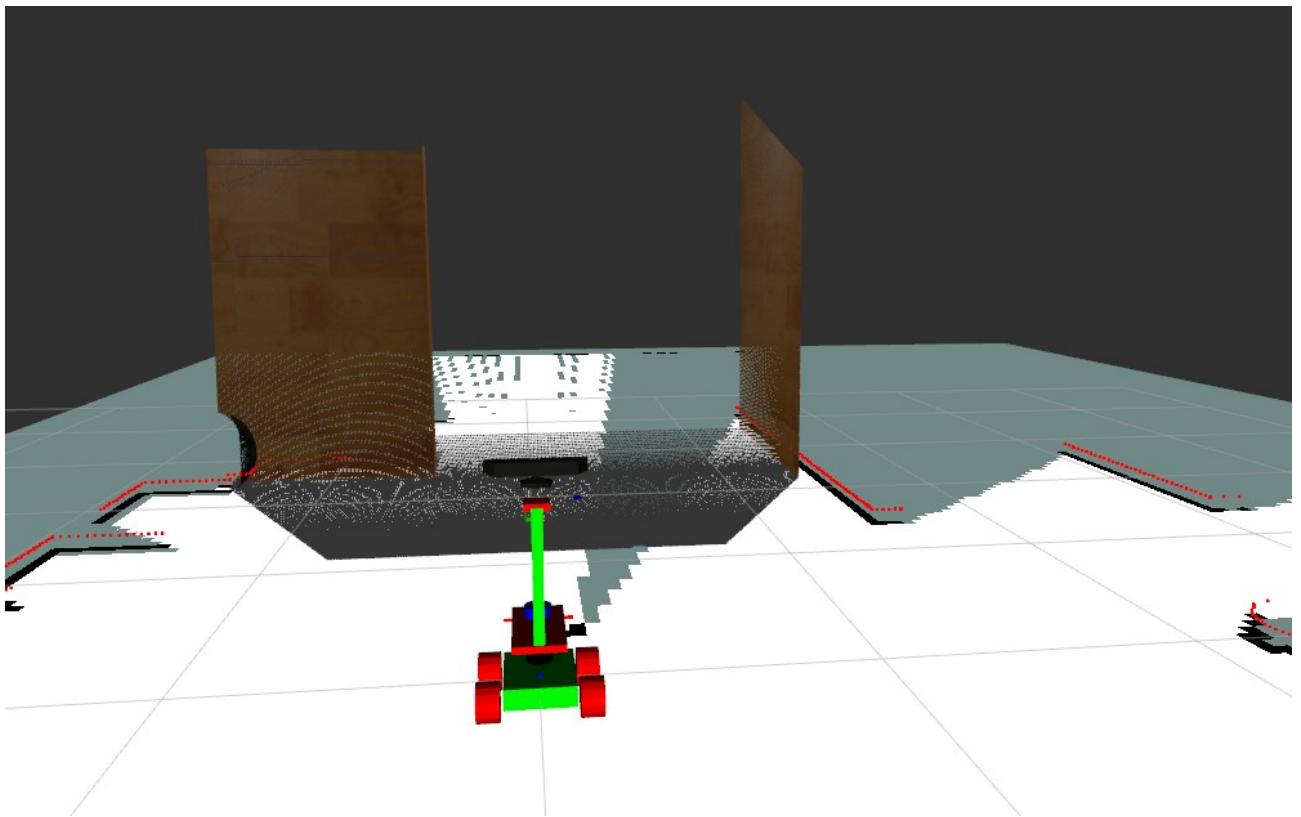
1. Remove the `args="--delete_db_on_start"` from your node launcher since you will need your database to localize too.
2. Remove the `Mem/NotLinkedNodesKept` parameter
3. Lastly, add the `Mem/IncrementalMemory` parameter of type string and set it to false to finalize the changes needed to put the robot into localization mode.

This is another method for localization you can keep in mind when working on your next robotics project!

## Let's see how to solve the Robot Kidnap problem in Action with RTAB-Map

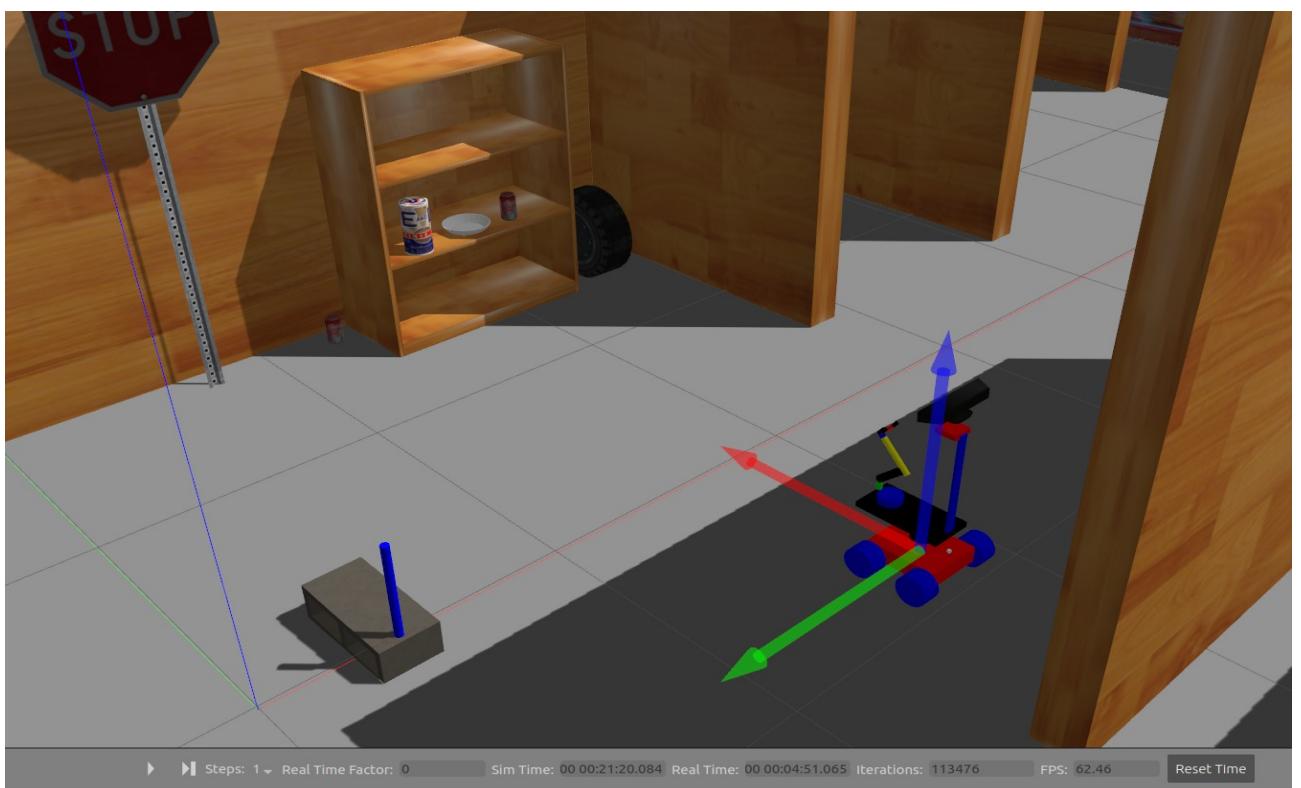
First we start the simulation, `rtabmap` in localization mode and `rviz`.



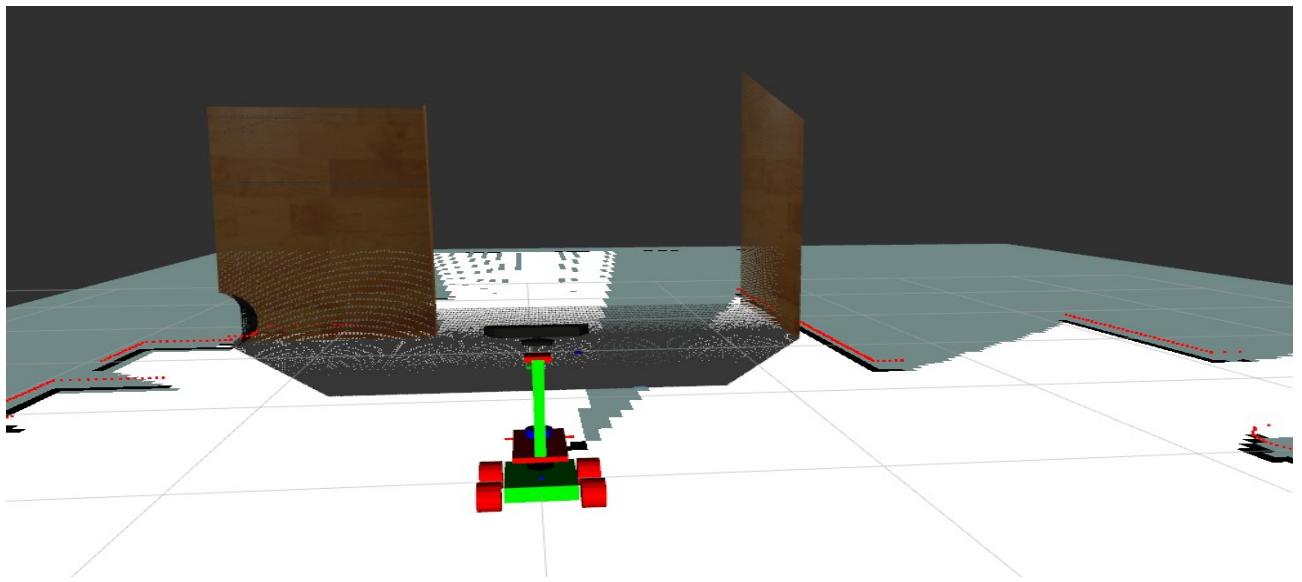


We can see ofcourse that the location of the robot in Gazebo and RVIZ matches since we just started the simulation.

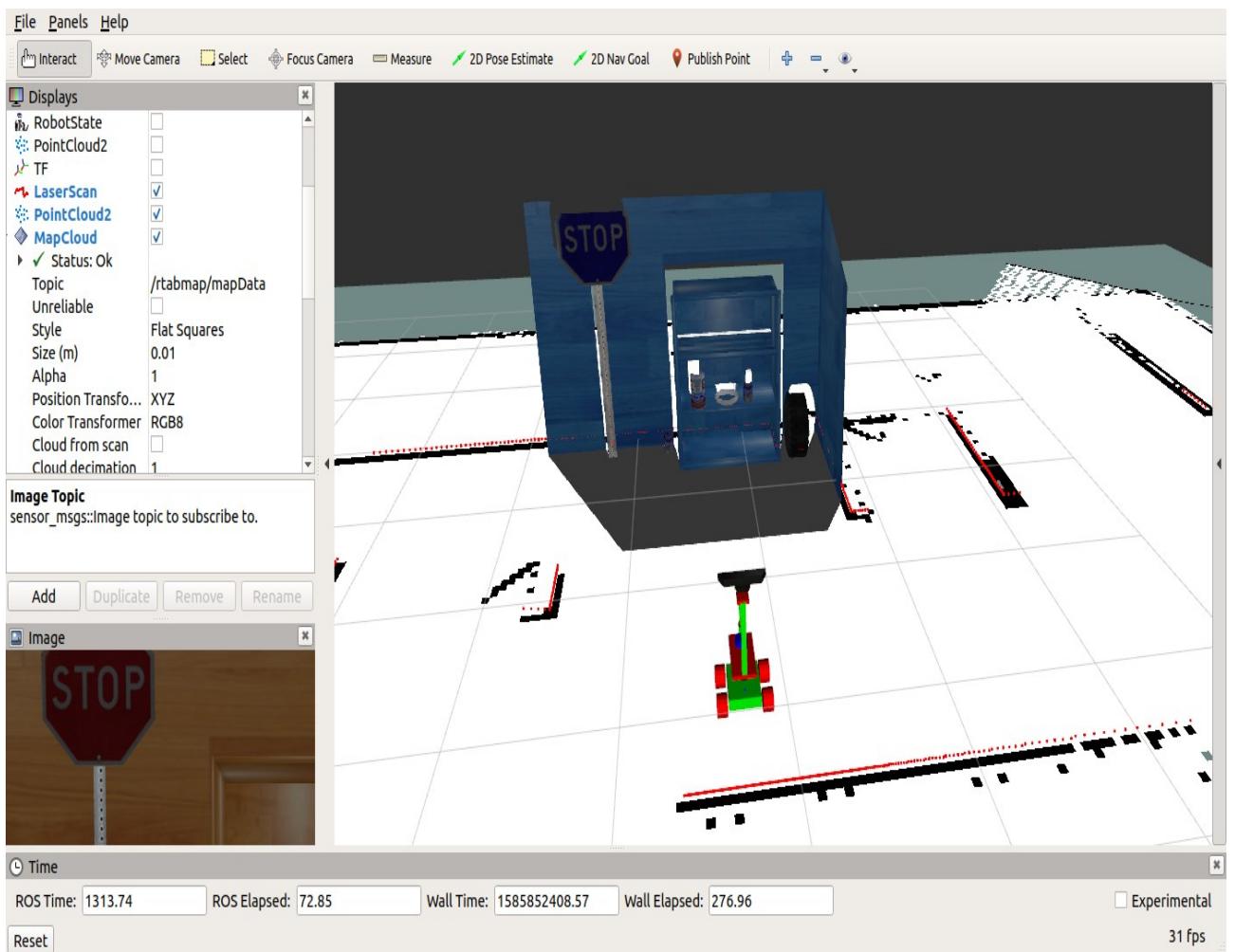
Next we pause the Gazebo simulation and Kidnap the robot by placing it in another location preferably were useful features can be detected for localization. Below we can see this!



The RVIZ at this point (a.k.a what the robot thinks is) is at the starting location since we haven't unparsed the simulation yet!



Next, by un-pausing the simulation new images will come and RTAB-Map should find the new location of the Robot solving the Robot Kidnap and providing us with a very robust Localization solution. Which we can see but the updated location seen in RVIZ.



## RTAB-Map extra features

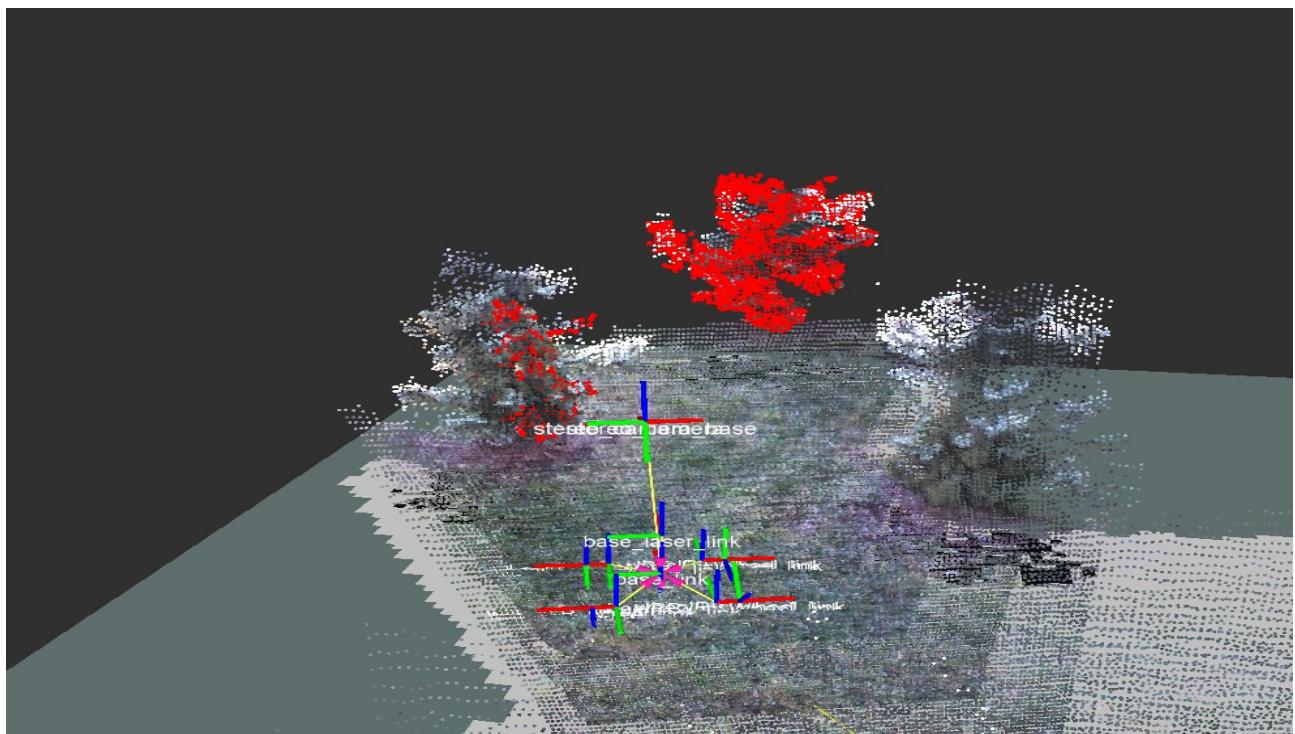
We have only touched on the main features of RTAB-Map, but let's take a few minutes to highlight some other fantastic features that one may like to incorporate into their robotics project!



## Visual Odometry

In situations where odometry data may not be available (ex. there are no wheel encoders), you can use visual odometry instead to gather the same information with less resolution. This might be more applicable to a drone with just an RGB-D camera.

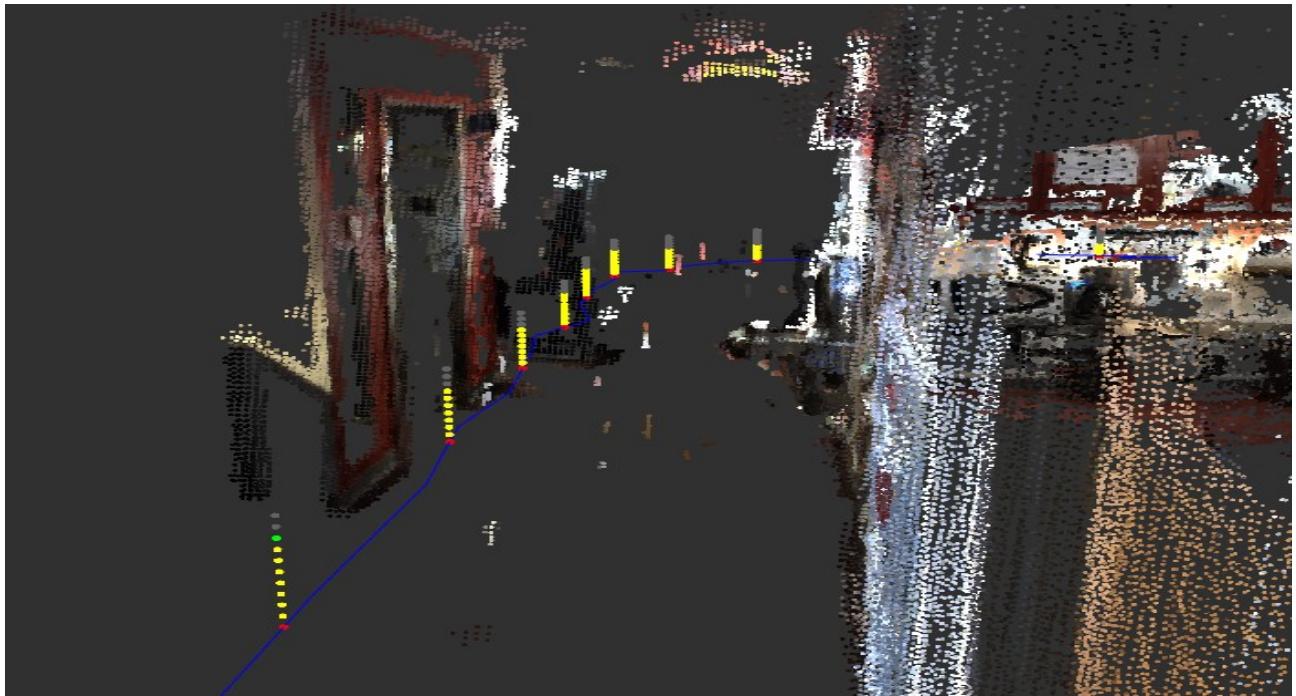
More information can be found [here](#).



## Obstacle Detection

This is a handy nodelet that extracts obstacles and the ground from your point cloud. With this information in hand, you can avoid these obstacles when executing a desired path.

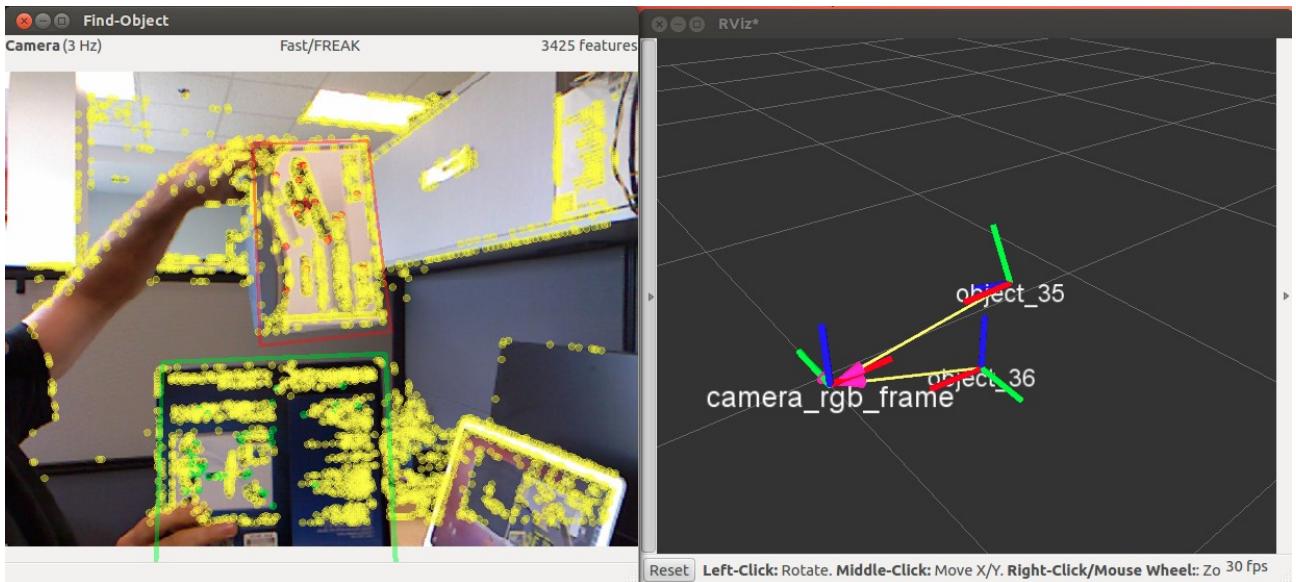
More information can be found [here](#).



## WiFi Signal Strength Mapping

This feature allows you to visualize the strength of your robot's WiFi connection. This can be important in order to determine where your robot may lose its signal, therefore dictating it to avoid certain areas. You could also try to find ways to remedy the situation with larger antennas.

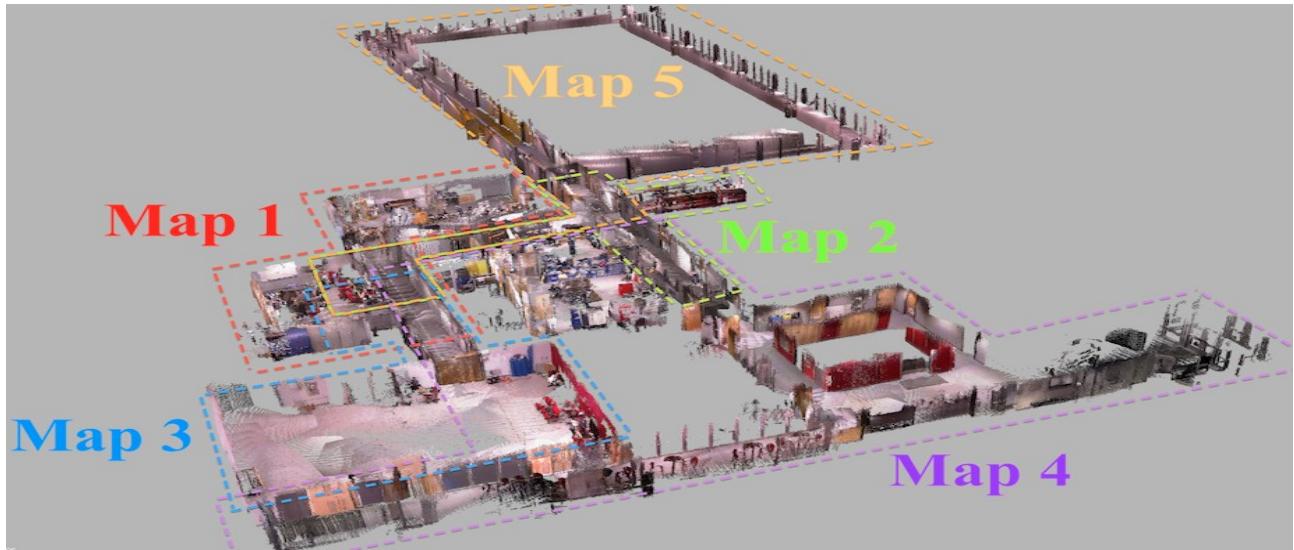
More information can be found [here](#).



## Finding Objects both in 2D and 3D

Object detection is a great addition to your robotic stack if you are searching for an object and would like to find it and its position when surveying the area.

More information can be found [here](#).



## Multisession Mapping

Multisession Mapping is a great way to map specific areas and then come back to do more mapping beyond your first mapped area and have it be added onto your previous mapping session.

More information can be found [here](#).

For any other information you can find it at the following links:

[rtabmap Documentation](#)

[rtabmap\\_ros Documentation](#)

## Finally

You can watch my youtube video

[RtabMap - 3D mapping - Localization Robot Kidnap](#)

to watch my robot using RTAB-Map to map my home in real time, and after the fact using the generated map-database to solve the Robot Kidnap problem.

# 12. Path Planning - The Navigation Stack and Move Base

## 12.1 Programming A\* in C++

Go to my repo <https://github.com/panagelak/RoboND-A--Algorithm> to see how you can program this code with exercises step by step by the Robotics Nanodegree program.

```
#include <iostream>
#include <math.h>
#include <vector>
#include <iterator>
#include <fstream>
#include "src/matplotlibcpp.h" //Graph Library

using namespace std;
namespace plt = matplotlibcpp;

// Map class
class Map {
public:
    const static int mapHeight = 300;
    const static int mapWidth = 150;
    vector<vector<double>> map = GetMap();
    vector<vector<int>> grid = Maptogrid();
    vector<vector<int>> heuristic = GenerateHeuristic();

private:
    // Read the file and get the map
    vector<vector<double>> GetMap()
    {
        vector<vector<double>> mymap (mapHeight, vector<double>(mapWidth));
        ifstream myReadFile;
        myReadFile.open("map.txt");

        while (!myReadFile.eof()) {
            for (int i = 0; i < mapHeight; i++) {
                for (int j = 0; j < mapWidth; j++) {
                    myReadFile >> mymap[i][j];
                }
            }
        }
        return mymap;
    }

    //Convert the map to 1's and 0's
    vector<vector<int>> Maptogrid()
    {
        vector<vector<int>> grid (mapHeight, vector<int>(mapWidth));
        for (int x = 0; x < mapHeight; x++) {
            for (int y = 0; y < mapWidth; y++) {
                if (map[x][y] == 0) //unkown state
                    grid[x][y] = 1;
                else if (map[x][y] > 0) //Occupied state
                    grid[x][y] = 0;
            }
        }
        return grid;
    }
}
```

```

        grid[x][y] = 1;

    else //Free state
        grid[x][y] = 0;
    }
}

return grid;
}

// Generate a Manhattan Heuristic Vector
vector<vector<int>> GenerateHeuristic()
{
    vector<vector<int>> heuristic(mapHeight, vector<int>(mapWidth));
    int goal[2] = { 60, 50 };
    for (int i = 0; i < heuristic.size(); i++) {
        for (int j = 0; j < heuristic[0].size(); j++) {
            int xd = goal[0] - i;
            int yd = goal[1] - j;
            // Manhattan Distance
            int d = abs(xd) + abs(yd);
            // Euclidian Distance
            // double d = sqrt(xd * xd + yd * yd);
            // Chebyshev distance
            // int d = max(abs(xd), abs(yd));
            heuristic[i][j] = d;
        }
    }
    return heuristic;
}
};

// Planner class
class Planner : Map {
public:
    int start[2] = { 230, 145 };
    int goal[2] = { 60, 50 };
    int cost = 1;

    string movements_arrows[4] = { "^", "<", "v", ">" };

    vector<vector<int>> movements{
        { -1, 0 },
        { 0, -1 },
        { 1, 0 },
        { 0, 1 }
    };

    vector<vector<int>> path;
};

// Printing vectors of any type
template <typename T>
void print2DVector(T Vec)
{
    for (int i = 0; i < Vec.size(); ++i) {
        for (int j = 0; j < Vec[0].size(); ++j) {
            cout << Vec[i][j] << ' ';
        }
        cout << endl;
    }
}

```

```

        }

    }

Planner search(Map map, Planner planner)
{
    // Create a closed 2 array filled with 0s and first element 1
    vector<vector<int>> closed(map.mapHeight, vector<int>(map.mapWidth));
    closed[planner.start[0]][planner.start[1]] = 1;

    // Create expand array filled with -1
    vector<vector<int>> expand(map.mapHeight, vector<int>(map.mapWidth, -1));

    // Create action array filled with -1
    vector<vector<int>> action(map.mapHeight, vector<int>(map.mapWidth, -1));

    // Defined the quadruplet values
    int x = planner.start[0];
    int y = planner.start[1];
    int g = 0;
    int f = g + map.heuristic[x][y];

    // Store the expansions
    vector<vector<int>> open;
    open.push_back({f, g, x, y});

    // Flags and Counts
    bool found = false;
    bool resign = false;
    int count = 0;

    int x2;
    int y2;

    // While I am still searching for the goal and the problem is solvable
    while (!found && !resign) {
        // Resign if no values in the open list and you can't expand anymore
        if (open.size() == 0) {
            resign = true;
            cout << "Failed to reach a goal" << endl;
        }
        // Keep expanding
        else {
            // Remove quadruplets from the open list
            sort(open.begin(), open.end());
            reverse(open.begin(), open.end());
            vector<int> next;
            // Stored the popped value into next
            next = open.back();
            open.pop_back();

            x = next[2];
            y = next[3];
            g = next[1];

            // Fill the expand vectors with count
            expand[x][y] = count;
            count += 1;

            // Check if we reached the goal:
            if (x == planner.goal[0] && y == planner.goal[1]) {

```

```

        found = true;
        //cout << "[" << g << ", " << x << ", " << y << "]" << endl;
    }

    //else expand new elements
    else {
        for (int i = 0; i < planner.movements.size(); i++) {
            x2 = x + planner.movements[i][0];
            y2 = y + planner.movements[i][1];
            if (x2 >= 0 && x2 < map.grid.size() && y2 >= 0 && y2 < map.grid[0].size()) {
                if (closed[x2][y2] == 0 and map.grid[x2][y2] == 0) {
                    int g2 = g + planner.cost;
                    f = g2 + map.heuristic[x2][y2];
                    open.push_back({f, g2, x2, y2});
                    closed[x2][y2] = 1;
                    action[x2][y2] = i;
                }
            }
        }
    }

    // Print the expansion List
    print2DVector(expand);

    // Find the path with robot orientation
    vector<vector<string> > policy(map.mapHeight, vector<string>(map.mapWidth,
    "-"));

    // Going backward
    x = planner.goal[0];
    y = planner.goal[1];
    policy[x][y] = '*';

    while (x != planner.start[0] or y != planner.start[1]) {
        x2 = x - planner.movements[action[x][y]][0];
        y2 = y - planner.movements[action[x][y]][1];
        // Store the Path in a vector
        planner.path.push_back({x2, y2});
        policy[x2][y2] = planner.movements_arrows[action[x][y]];
        x = x2;
        y = y2;
    }

    // Print the robot path
    cout << endl;
    print2DVector(policy);

    return planner;
}

void visualization(Map map, Planner planner)
{
    //Graph Format
    plt::title("Path");
    plt::xlim(0, map.mapHeight);
    plt::ylim(0, map.mapWidth);

    // Draw every grid of the map:
}

```

```

for (double x = 0; x < map.mapHeight; x++) {
    cout << "Remaining Rows= " << map.mapHeight - x << endl;
    for (double y = 0; y < map.mapWidth; y++) {
        if (map.map[x][y] == 0) { //Green unkown state
            plt::plot({ x }, { y }, "g.");
        }
        else if (map.map[x][y] > 0) { //Black occupied state
            plt::plot({ x }, { y }, "k.");
        }
        else { //Red free state
            plt::plot({ x }, { y }, "r.");
        }
    }
}

// Plot start and end states
plt::plot({ (double)planner.start[0] }, { (double)planner.start[1] }, "bo");
plt::plot({ (double)planner.goal[0] }, { (double)planner.goal[1] }, "b*");

// Plot the robot path
for (int i = 0; i < planner.path.size(); i++) {
    plt::plot({ (double)planner.path[i][0] }, { (double)planner.path[i][1] }, "b.");
}

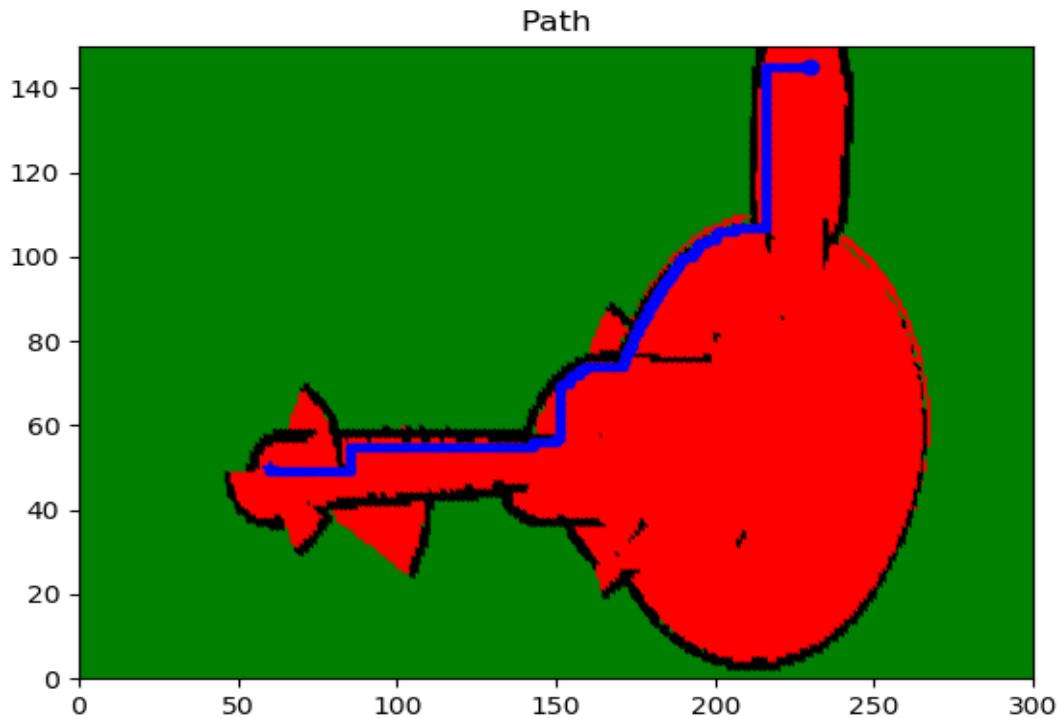
//Save the image and close the plot
plt::save("./Images/Path.png");
plt::clf();
}

int main()
{
    // Instantiate a planner and map objects
    Map map;
    Planner planner;
    // Generate the shortest Path using the Astar algorithm
    planner = search(map, planner);
    // Plot the Map and the path generated
    visualization(map, planner);

    return 0;
}

```

**The final result of this code would be to generate an image, with which you can visualize the output path.**



## 12.2 The Navigation Stack in ROS

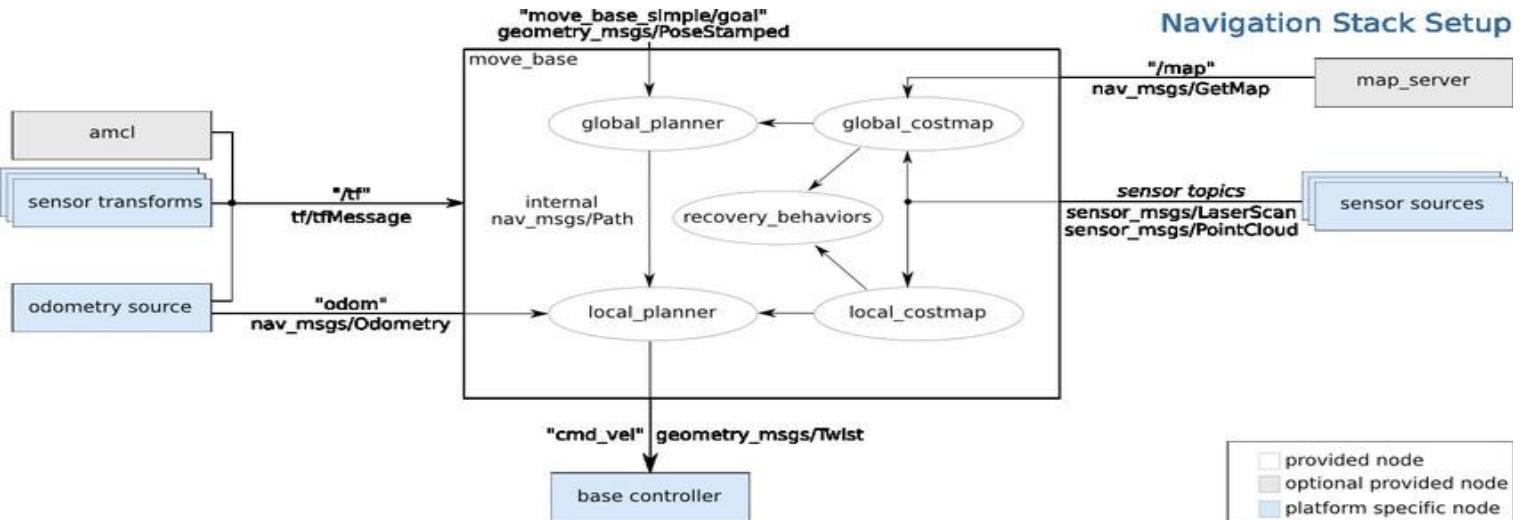
In order to understand the navigation stack, you should think of it as a set of algorithms that use the sensors of the robot and the odometry so that you can control the robot using a standard message. It can move your robot without any problems, such as crashing, getting stuck in a location, or getting lost to another position.

You would assume that this stack can be easily used with any robot. This is almost true, but it is necessary to tune some configuration files and write some nodes to use the stack.

The robot must satisfy some requirements before it uses the navigation stack:

- The navigation stack can only handle a differential drive and holonomic-wheeled robots. The shape requisites of the robot must either be a square or a rectangle. However, it can also do certain things with biped robots, such as robot localization, as long as the robot does not move sideways.
- It requires that the robot publishes information about the relationships between the positions of all the joints and sensors.
- The robot must send messages with linear and angular velocities.
- A planar laser must be on the robot to create the map and localization. Alternatively, you can generate something equivalent to several lasers or a sonar, or you can project the values to the ground if they are mounted at another place on the robot.

The following diagram shows you how the navigation stacks are organized. You can see three groups of boxes with colors (gray and white) and dotted lines. The plain white boxes indicate the stacks that are provided by ROS, and they have all the nodes to make your robot really autonomous:



We can see from the diagram that in order to use the Navigation Stack, we need the transforms of the robot, **especially the sensor transforms**. Furthermore we need the **sensor topics** in our case the **/scan topic**, alongside a **Map and odometry that can be provided by amcl**.

Up until now we have seen how to load and configure all of these prerequisites except the **move\_base** node which will enable our robot to move to a specific location in the map completely autonomous.

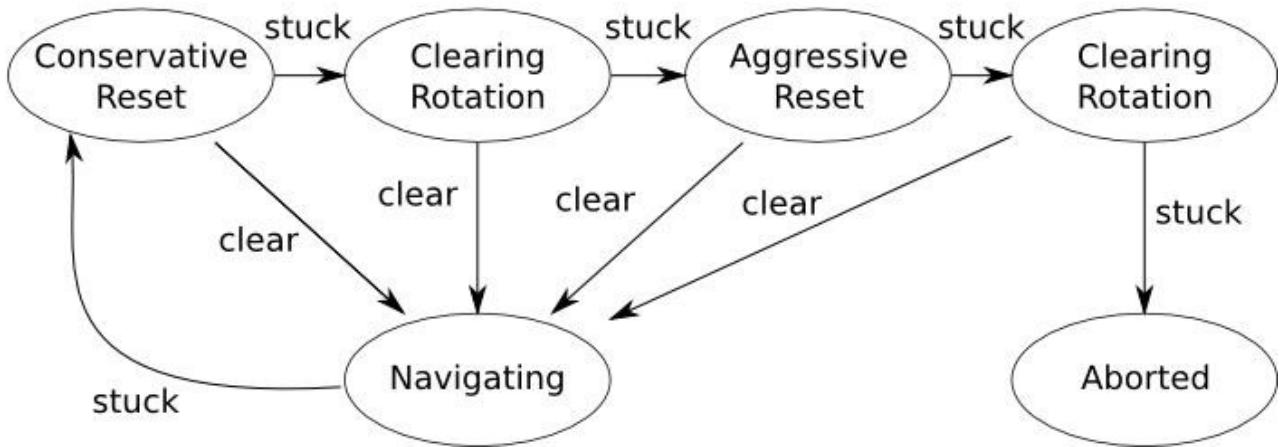
## 12.3 Move Base

The `move_base` package provides an implementation of an action (see the [actionlib](#) package) that, **given a goal in the world, will attempt to reach it with a mobile base**. The `move_base` node links together **a global and local planner to accomplish its global navigation task**. It supports any global planner adhering to the `nav_core::BaseGlobalPlanner` interface specified in the [nav\\_core](#) package and any local planner adhering to the `nav_core::BaseLocalPlanner` interface specified in the [nav\\_core](#) package. The `move_base` node **also maintains two costmaps, one for the global planner, and one for a local planner** (see the [costmap\\_2d](#) package) that are used to accomplish navigation tasks.

The `move_base` node provides a ROS interface for configuring, running, and interacting with the [navigation stack](#) on a robot as we saw in the previous graph.

## Expected Robot Behaviour

## move\_base Default Recovery Behaviors



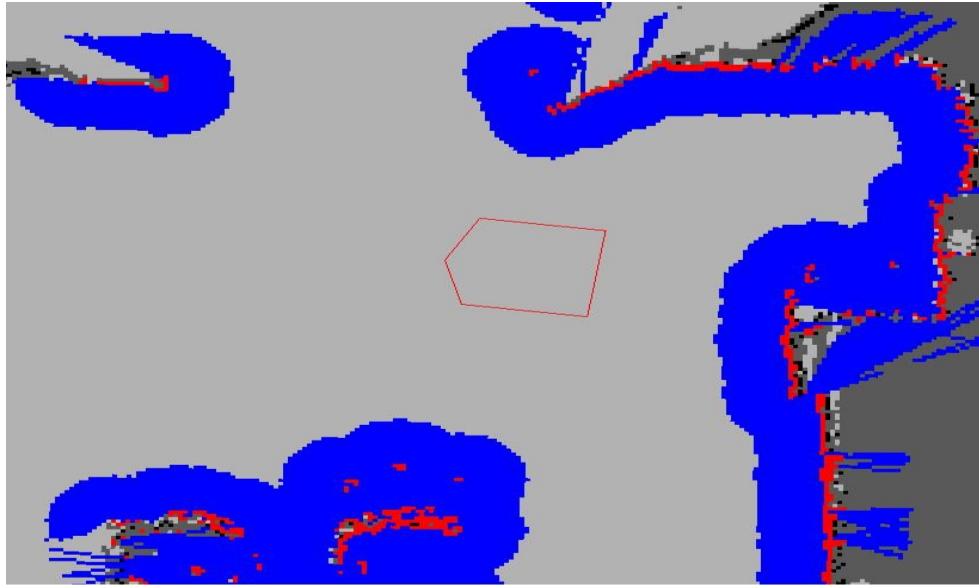
Running the `move_base` node on a robot that is properly configured (please see [navigation stack documentation](#) for more details) results in a robot that will attempt to achieve a goal pose with its base to within a user-specified tolerance. In the absence of dynamic obstacles, the `move_base` node will eventually get within this tolerance of its goal or signal failure to the user. The `move_base` node may **optionally perform recovery behaviors when the robot perceives itself as stuck**. By default, the `move_base` node will take the following actions to attempt to clear out space:

First, **obstacles outside of a user-specified region will be cleared from the robot's map**. Next, if possible, the robot will perform an **in-place rotation to clear out space**. If this too fails, the robot will more aggressively clear its map, removing all obstacles outside of the rectangular region in which it can rotate in place. This will be followed by another in-place rotation. If all this fails, the robot will consider its goal infeasible and notify the user that it has aborted. These recovery behaviors can be configured using the [recovery behaviors](#) parameter, and disabled using the [recovery behavior enabled](#) parameter.

## 12.4 Costmap\_2d

This package provides an implementation of a 2D costmap that takes in sensor data from the world, builds a 2D or 3D occupancy grid of the data (depending on whether a voxel based implementation is used), and inflates costs in a 2D costmap based on the occupancy grid and a user specified inflation radius. This package also provides support for `map_server` based initialization of a costmap, rolling window based costmaps, and parameter based subscription to and configuration of sensor topics.

### Overview



*Note: In the picture above, the **red cells** represent obstacles in the costmap, the **blue cells** represent obstacles inflated by the inscribed radius of the robot, and the **red polygon** represents the footprint of the robot. For the robot to avoid collision, the footprint of the robot should never intersect a red cell and the center point of the robot should never cross a blue cell.*

The costmap\_2d package provides a **configurable structure** that maintains information about where the robot should navigate in the form of an occupancy grid. The costmap uses sensor data and information from the static map to store and update information about obstacles in the world through the **costmap\_2d::Costmap2DROS** object. The costmap\_2d::Costmap2DROS object provides a purely **two dimensional interface** to its users, meaning that queries about obstacles can only be made in columns. For example, a table and a shoe in the same position in the XY plane, but with different Z positions would result in the corresponding cell in the costmap\_2d::Costmap2DROS object's costmap having an identical cost value. This is designed to help planning in planar spaces.

As of the Hydro release, the underlying methods used to write data to the costmap is fully configurable. **Each bit of functionality exists in a layer**. For instance, the **static map is one layer**, and the **obstacles are another layer**. By default, the obstacle layer maintains information three dimensionally (see [voxel\\_grid](#)). Maintaining 3D obstacle data allows the layer to deal with marking and clearing more intelligently.

The main interface is `costmap_2d::Costmap2DROS` which maintains much of the ROS related functionality. It contains a `costmap_2d::LayeredCostmap` which is used to keep track of each of the layers. Each layer is instantiated in the `Costmap2DROS` using [pluginlib](#) and is added to the `LayeredCostmap`. The layers themselves may be compiled individually, allowing arbitrary changes to the costmap to be made through the C++ interface. The `costmap_2d::Costmap2D` class implements the basic data structure for storing and accessing the two dimensional costmap.

The details about how the Costmap updates the occupancy grid are described below, along with links to separate pages describing how the individual layers work.

## Marking and Clearing

The costmap automatically subscribes to sensors topics over ROS and updates itself accordingly. Each sensor is used to either mark (insert obstacle information into the costmap), clear (remove

obstacle information from the costmap), or both. A marking operation is just an index into an array to change the cost of a cell. A clearing operation, however, consists of raytracing through a grid from the origin of the sensor outwards for each observation reported. If a three dimensional structure is used to store obstacle information, obstacle information from each column is projected down into two dimensions when put into the costmap.

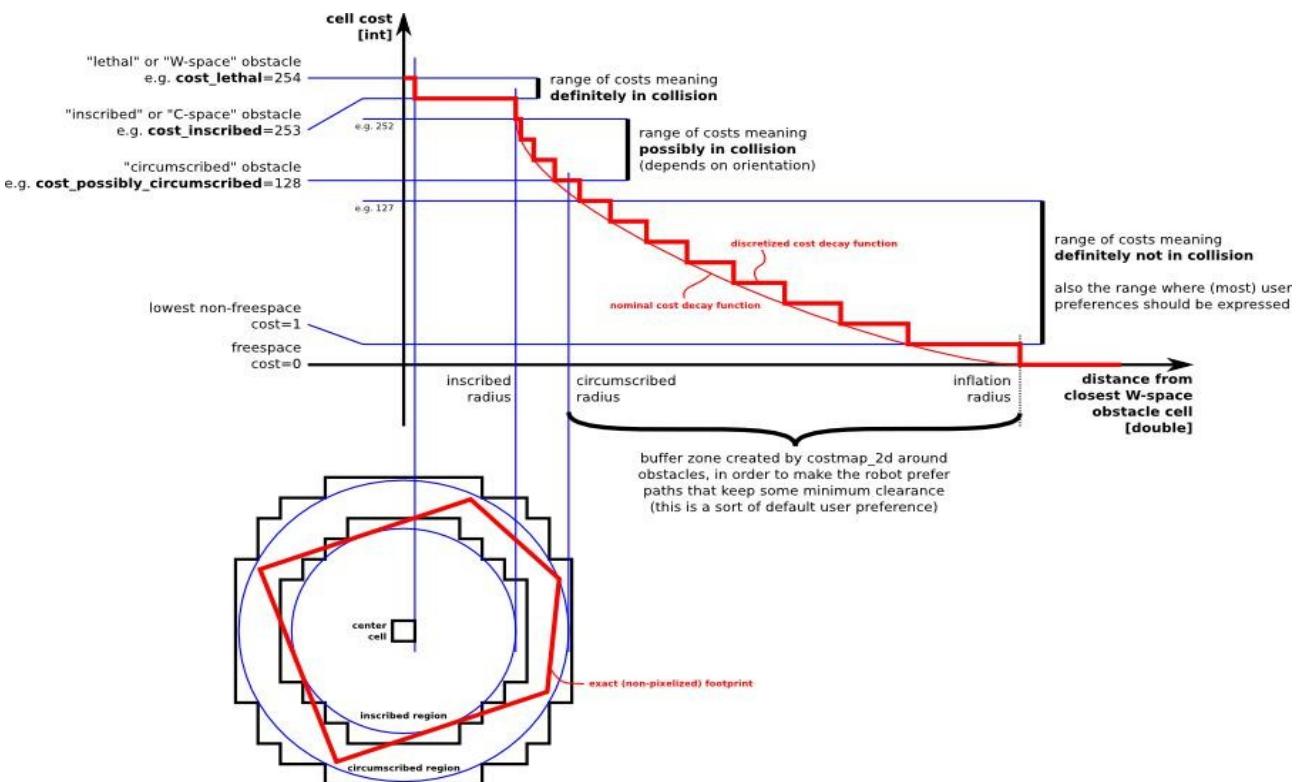
## Occupied, Free, and Unknown Space

While each cell in the costmap can have one of 255 different cost values (see the [inflation](#) section), the underlying structure that it uses is capable of representing only three. Specifically, each cell in this structure can be either free, occupied, or unknown. Each status has a special cost value assigned to it upon projection into the costmap. Columns that have a certain number of occupied cells (see [mark\\_threshold](#) parameter) are assigned a costmap\_2d::LETHAL\_OBSTACLE cost, columns that have a certain number of unknown cells (see [unknown\\_threshold](#) parameter) are assigned a costmap\_2d::NO\_INFORMATION cost, and other columns are assigned a costmap\_2d::FREE\_SPACE cost.

## Map Updates

The costmap performs map update cycles at the rate specified by the [update\\_frequency](#) parameter. Each cycle, sensor data comes in, marking and clearing operations are performed in the underlying occupancy structure of the costmap, and this structure is projected into the costmap where the appropriate cost values are assigned as described above. After this, each obstacle inflation is performed on each cell with a costmap\_2d::LETHAL\_OBSTACLE cost. This consists of propagating cost values outwards from each occupied cell out to a user-specified inflation radius. The details of this inflation process are outlined below.

### Inflation



**Inflation is the process of propagating cost values out from occupied cells that decrease with distance. For this purpose, we define 5 specific symbols for costmap values as they relate to a robot.**

- "Lethal" cost means that there is an actual (workspace) obstacle in a cell. So if the robot's center were in that cell, the robot would obviously be in collision.
- "Inscribed" cost means that a cell is less than the robot's inscribed radius away from an actual obstacle. So the robot is certainly in collision with some obstacle if the robot center is in a cell that is at or above the inscribed cost.
- "Possibly circumscribed" cost is similar to "inscribed", but using the robot's circumscribed radius as cutoff distance. Thus, if the robot center lies in a cell at or above this value, then it depends on the orientation of the robot whether it collides with an obstacle or not. We use the term "possibly" because it might be that it is not really an obstacle cell, but some user-preference, that put that particular cost value into the map. For example, if a user wants to express that a robot should attempt to avoid a particular area of a building, they may inset their own costs into the costmap for that region independent of any obstacles. Note, that although the value is 128 is used as an example in the diagram above, the true value is influenced by both the inscribed\_radius and inflation\_radius parameters as defined in the [code](#).
- "Freespace" cost is assumed to be zero, and it means that there is nothing that should keep the robot from going there.
- "Unknown" cost means there is no information about a given cell. The user of the costmap can interpret this as they see fit.
- All other costs are assigned a value between "Freespace" and "Possibly circumscribed" depending on their distance from a "Lethal" cell and the decay function provided by the user.

The rationale behind these definitions is that we leave it up to planner implementations to care or not about the exact footprint, yet give them enough information that they can incur the cost of tracing out the footprint only in situations where the orientation actually matters.

## Map Types

There are two main ways to initialize a `costmap_2d::Costmap2DROS` object. The first is to seed it with a user-generated static map (see the [map\\_server](#) package for documentation on building a map). In this case, the costmap is initialized to match the width, height, and obstacle information provided by the static map. This configuration is normally used in conjunction with a localization system, like [amcl](#), that allows the robot to register obstacles in the map frame and update its costmap from sensor data as it drives through its environment.

The second way to initialize a `costmap_2d::Costmap2DROS` object is to give it a width and height and to set the [rolling\\_window](#) parameter to be true. The [rolling\\_window](#) parameter keeps the robot in the center of the costmap as it moves throughout the world, dropping obstacle information from the map as the robot moves too far from a given area. This type of configuration is most often used in an odometric coordinate frame where the robot only cares about obstacles within a local area.

## 12.5 Configuring the costmaps – `global_costmap` and `local_costmap`

Okay, now we are going **to start configuring the navigation stack and all the necessary files to start it**. To start with the configuration, first we will learn what costmaps are and what they are used for. Our robot will move through the map using two types of navigation: **global and local** :

- The global navigation is used to create paths for a goal in the map or at a far-off distance

- The local navigation is used to create paths in the nearby distances and avoid obstacles, for example, a square window of 4 x 4 meters around the robot

These modules use costmaps to keep all the information of our map. The global costmap is used for global navigation and the local costmap for local navigation.

**The costmaps have parameters to configure the behaviors, and they have common parameters as well, which are configured in a shared file.**

The configuration basically consists of three files where we can set up different parameters.

The files are as follows:

- costmap\_common\_params.yaml
- global\_costmap\_params.yaml
- local\_costmap\_params.yaml

Just by reading the names of these configuration files, you can instantly guess what they are used for. Now that you have a basic idea about the usage of costmaps, we are going to create the configuration files and explain the parameters that are configured in them.

## Configuring the common paramaters

### costmap\_common\_params.yaml

```

footprint: [[-0.15, -0.15], [-0.15, 0.15], [0.15, 0.15], [0.15, -0.15]]
footprint_padding: 0.01
map_type: costmap

obstacle_range: 3
raytrace_range: 3.5

transform_tolerance: 0.3

min_obstacle_height: 0.0
max_obstacle_height: 0.4

#layer definitions
static:
  enable: true
  map_topic: /map
  subscribe_to_updates: true

obstacles_laser:
  enabled: true
  observation_sources: laser
  laser: {data_type: LaserScan, clearing: true, marking: true, topic: /scan, inf_is_valid: true}

inflation:
  enabled: true
  inflation_radius: 0.29
  cost_scaling_factor: 1.5

```

This file is used to configure common parameters. The parameters are used in local\_costmap and global\_costmap . Let's break the code and understand it.

The **obstacle\_range** and **raytrace\_range** attributes are used to indicate the maximum distance that the sensor will read and introduce new information in the costmaps. The first one is used for the obstacles. If the robot detects an obstacle closer than 3 meters in our case, it will put the obstacle in the costmap. The other one is used to clean/clear the costmap and update the free space in it when the robot moves. Note that we can only detect the echo of the laser or sonar with the obstacle; we cannot perceive the whole obstacle or object itself, but this simple approach will be enough to deal with these kinds of measurements, and we will be able to build a map and localize within it.

The transform\_tolerance parameter configures the maximum latency for the transforms, in our case 0.3 seconds

The footprint attribute is used to indicate the geometry of the robot to the navigation stack. It is used to keep the right distance between the obstacles and the robot, or to find out if the robot can go through a door. The inflation\_radius attribute is the value given to keep a minimal distance between the geometry of the robot and the obstacles.

The cost\_scaling\_factor attribute modifies the behavior of the robot around the obstacles. You can make a behavior aggressive or conservative by changing the parameter.

With the observation\_sources attribute, you can set the sensors used by the navigation stack to get the data from the real world and calculate the path.

The following line will configure the sensor's frame and the uses of the data:

```
scan: {sensor_frame: base_link, data_type: LaserScan, topic: /scan, marking: true, clearing: true}
```

The laser configured in the previous line is used to add and clear obstacles in the costmap. For example, you could add a sensor with a wide range to find obstacles and another sensor to navigate and clear the obstacles. The topic's name is configured in this line. It is important to configure it well, because the navigation stack could wait for another topic and all this while, the robot is moving and could crash into a wall or an obstacle.

## Configuring the global costmap

```
global_frame: map
robot_base_frame: robot_footprint
update_frequency: 5
publish_frequency: 1
width: 20.0
height: 20.0
resolution: 0.02
static_map: true
track_unknown_space: false
rolling_window: false
plugins:
  - {name: static, type: "costmap_2d::StaticLayer"}
  - {name: obstacles_laser, type: "costmap_2d::VoxelLayer"}
  - {name: inflation, type: "costmap_2d::InflationLayer"}
```

The **global\_frame** and the **robot\_base\_frame** attributes define the transformation between the map and the robot. This transformation is for the global costmap.

You can configure the frequency of updates for the costmap. In this case, it is 5 Hz. The static\_map attribute is used for the global costmap to see whether a map or the map server is used to initialize the costmap. If you aren't using a static map, set this parameter to false .

## Configuring the local costmap

```
global_frame: odom
robot_base_frame: robot_footprint
update_frequency: 15
publish_frequency: 3.0
width: 2.5
height: 2.5
resolution: 0.02
static_map: false
rolling_window: true
plugins:
  - {name: obstacles_laser, type: "costmap_2d::ObstacleLayer"}
  - {name: inflation, type: "costmap_2d::InflationLayer"}
```

Here the **global\_frame** needs to be set in the **odom** frame. The update frequency is larger than the global costmap alongside the publish frequency (for visualization).

The **rolling\_window** parameter is used to keep the costmap centered on the robot when it is moving around the world.

You can configure the dimensions and the resolution of the costmap with the **width** , **height** , and **resolution** parameters. The values are given in meters.

## Base local planner configuration

Once we have the costmaps configured, it is necessary to configure the base planner.

The base planner is used to generate the velocity commands to move our robot.

### base\_local\_planner.yaml

TrajectoryPlannerROS:

```
# Robot Configuration Parameters
```

```
acc_lim_x: 0.5 acc_lim_theta: 0.5
```

```
max_vel_x: 0.23
```

```
min_vel_x: 0.08
```

```
max_vel_theta: 0.12
```

```
min_vel_theta: -0.12
```

```
max_in_place_vel_theta: 0.13
```

```
min_in_place_vel_theta: 0.04
```

```
holonomic_robot: false
```

```
escape_vel: -0.21
```

```
# Goal Tolerance Parameters
```

```
yaw_goal_tolerance: 0.2
```

```
xy_goal_tolerance: 0.2
```

```
latch_xy_goal_tolerance: true
```

```

# Forward Simulation Parameters
sim_time: 1.3
sim_granularity: 0.01
angular_sim_granularity: 0.01
vx_samples: 20
vtheta_samples: 30
controller_frequency: 50.0

# Trajectory scoring parameters
meter_scoring: true # Whether the gdist_scale and pdist_scale parameters should assume that goal_distance and path_distance are expressed in units of meters or cells. Cells are assumed by default (false).
occdist_scale: 0.23 #The weighting for how much the controller should attempt to avoid obstacles.
default 0.01
pdist_scale: 0.8 # The weighting for how much the controller should stay close to the path it was given . default 0.6
gdist_scale: 1.1 # The weighting for how much the controller should attempt to reach its local goal, also controls speed default 0.8

heading_lookahead: 0.2 #How far to look ahead in meters when scoring different in-place-rotation trajectories
heading_scoring: false #Whether to score based on the robot's heading to the path or its distance from the path.
default false
heading_scoring_timestep: 0.4 #How far to look ahead in time in seconds along the simulated trajectory when using heading scoring
dwa: false #Whether to use the Dynamic Window Approach (DWA)_ or whether to use Trajectory Rollout
simple_attractor: false
publish_cost_grid_pc: true

# Oscillation Prevention Parameters
oscillation_reset_dist: 0.05 #How far the robot must travel in meters before oscillation flags are reset (double, default: 0.05)
escape_reset_dist: 0.21
escape_reset_theta: 0.13

```

The config file will set the maximum and minimum velocities for your robot.

The acceleration is also set.

The holonomic\_robot parameter is true if you are using a holonomic platform. In our case, our robot is based on a non-holonomic platform and the parameter is set to false . A holonomic vehicle is one that can move in all the configured space from any position. In other words, if the places where the robot can go are defined by any x and y values in the environment, this means that the robot can move there from any position. For example, if the robot can move forward, backward, and laterally, it is holonomic. A typical case of a non-holonomic vehicle is a car, as it cannot move laterally, and from a given position, there are many other positions (or poses) that are not reachable. Also, a differential platform is non-holonomic.

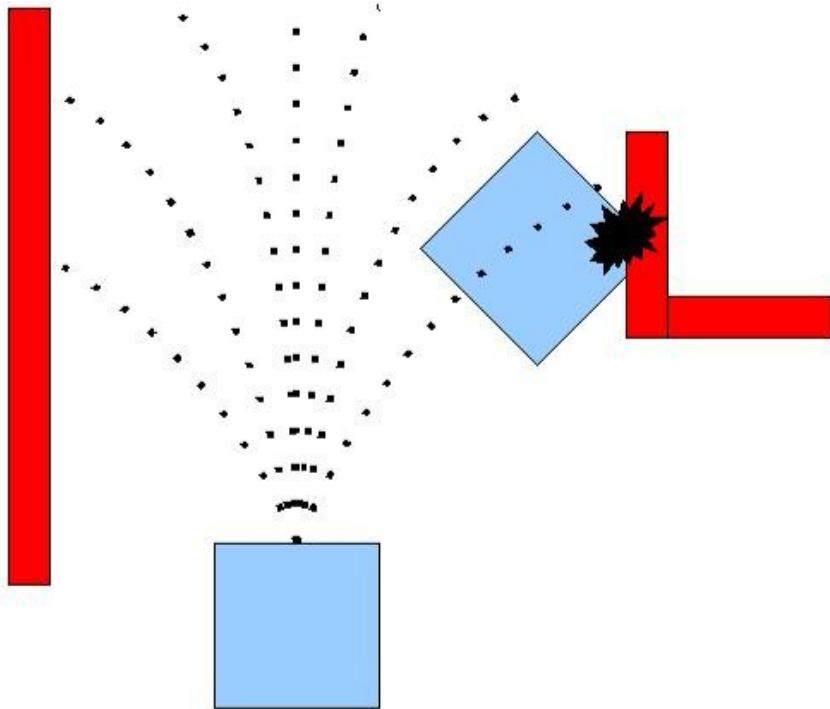
This is the most tricky config to set, some experimentation especially with the paramaters **occdist\_scale, pdist\_scale and gdist\_scale is required.**

**This local planner was especially beneficiary since when it's near an obstacle the robot has a recovery behaviour that moves it a little backwards, this in many cases will cause the robot to get unstuck.**

## Overview

The base\_local\_planner package provides a controller that drives a mobile base in the plane. This controller serves to connect the path planner to the robot. **Using a map, the planner creates a kinematic trajectory for the robot to get from a start to a goal location.** Along the way, the

planner creates, at least locally around the robot, a value function, represented as a grid map. This value function encodes the costs of traversing through the grid cells. The controller's job is to use this value function to determine dx,dy,dtheta velocities to send to the robot.



The basic idea of both the Trajectory Rollout and Dynamic Window Approach (DWA) algorithms is as follows:

1. Discretely sample in the robot's control space (dx,dy,dtheta)
2. For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time.
3. Evaluate (score) each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Discard illegal trajectories (those that collide with obstacles).
4. Pick the highest-scoring trajectory and send the associated velocity to the mobile base.
5. Rinse and repeat.

**DWA differs from Trajectory Rollout in how the robot's control space is sampled.** Trajectory Rollout samples from the set of achievable velocities over the entire forward simulation period given the acceleration limits of the robot, while DWA samples from the set of achievable velocities for just one simulation step given the acceleration limits of the robot. This means that DWA is a more **efficient** algorithm because it samples a smaller space, but may be **outperformed by Trajectory Rollout for robots with low acceleration limits because DWA does not forward simulate constant accelerations.**

In most cases we use the DWA for it's efficient gains **however in my robot this planner didn't achieve good results, while Trajectory planner worked best.**

Another local planner you can try is called **teb\_local\_planner** usually used for car-like robots.

### Base global planner configuration

```
recovery_behaviour_enabled: true
controller_frequency: 5.0
NavfnROS:
```

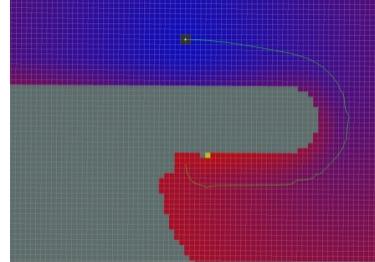
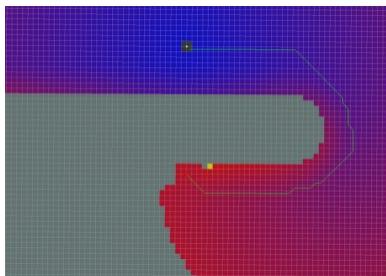
```

allow_unknown: false # Specifies whether or not to allow navfn to create plans that traverse unknown space.
default_tolerance: 0.15 # A tolerance on the goal point for the planner.
orientation_mode: 3

```

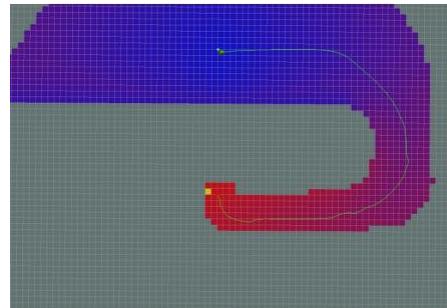
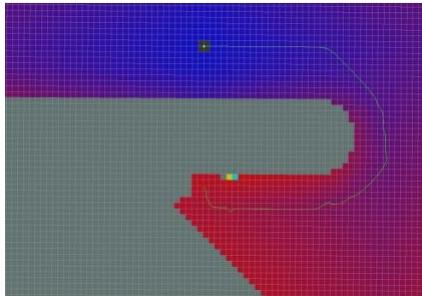
navfn uses Dijkstra's algorithm to find a global path with minimum cost between start point and end point. Global planner is built as a more flexible replacement of navfn with more options. These options include (1) support for A\*, (2) toggling quadratic approximation, (3) toggling grid path.

### Examples of different Parameterizations



*Illustration 1:  
use\_grid\_path = True*

*Illustration 2: all parameters  
default*



*Illustration 3: use\_dijkstra = False*

*Illustration 4: use\_quadratic = True*

#### Orientation filter

As a post-processing step, an orientation can be added to the points on the path. With use of the ~orientation\_mode parameter (dynamic reconfigure), the following orientation modes can be set:

- **None=0** (No orientations added except goal orientation)
- **Forward=1** (Positive x axis points along path, except for the goal orientation)
- **Interpolate=2** (Orientations are a linear blend of start and goal pose)
- **ForwardThenInterpolate=3** (Forward orientation until last straightaway, then a linear blend until the goal pose)
- **Backward=4** (Negative x axis points along the path, except for the goal orientation)
- **Leftward=5** (Positive y axis points along the path, except for the goal orientation)
- **Rightward=6** (Negative y axis points along the path, except for the goal orientation)

The orientation of point i is calculated using the positions of `i - orientation_window_size` and `i + orientation_window_size`. The window size can be altered to smoothen the orientation calculation.

**Orientation mode 3 seemed to work best for me.**

**Creating a launch file for the move\_base**

### **move\_base.launch**

```
<?xml version="1.0"?>

<launch>
  <!-- Scan topic -->
  <arg name="scan_topic" default="scan" />

  <node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen">
    <param name="base_global_planner" value="navfn/NavfnROS"/>
    <param name="base_local_planner" value="base_local_planner/TrajectoryPlannerROS"/> <!--param
name="base_local_planner" value="teb_local_planner/TebLocalPlannerROS"/--> <!--param
name="base_local_planner" value="dwa_local_planner/DWAPlannerROS"/-->

    <rosparam      file="$(find    lynxbot_bringup)/config/base_global_planner.yaml"      command="load"/>
    <rosparam file="$(find lynxbot_bringup)/config/base_local_planner.yaml" command="load" />

    <rosparam file="$(find lynxbot_bringup)/config/costmap_common_params.yaml"
command="load" ns="global_costmap" />
    <rosparam file="$(find lynxbot_bringup)/config/costmap_common_params.yaml"
command="load" ns="local_costmap" />

    <rosparam file="$(find lynxbot_bringup)/config/local_costmap_params.yaml" command="load"
ns="local_costmap"/>
    <rosparam file="$(find lynxbot_bringup)/config/global_costmap_params.yaml"
command="load" ns="global_costmap"/>

    <remap from="cmd_vel" to="/cmd_vel"/>
    <remap from="odom" to="/odom"/>
    <remap from="scan" to="/scan"/>

  </node>

</launch>
```

**In this file we launch all the files discussed previous, with their appropriate namespaces.**

**Now after launching main.launch, amcl.launch and move\_base.launch we are ready to use the Navigation Stack**

## **12.6 Setting up rviz for the navigation stack**

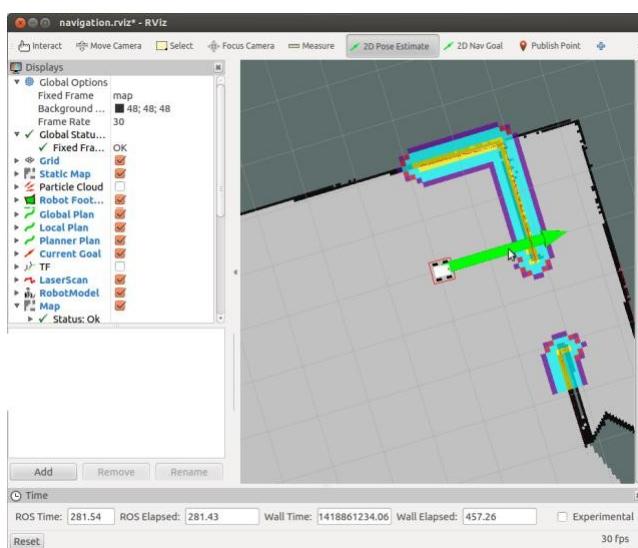
### **The 2D Pose estimated**

The 2D pose estimate (P shortcut) allows the user to initialize the localization system used by the navigation stack by setting the pose of the robot in the world.

The navigation stack waits for the new pose of a new topic with the name `initialpose`. This topic is sent using the rviz windows where we previously changed the name of the topic.

You can see in the following screenshot how you can use `initialpose`. Click on the 2D Pose Estimate button, and click on the map to indicate the initial position of your robot. If you don't do this at the beginning, the robot will start the auto-localization process and try to set an initial pose:

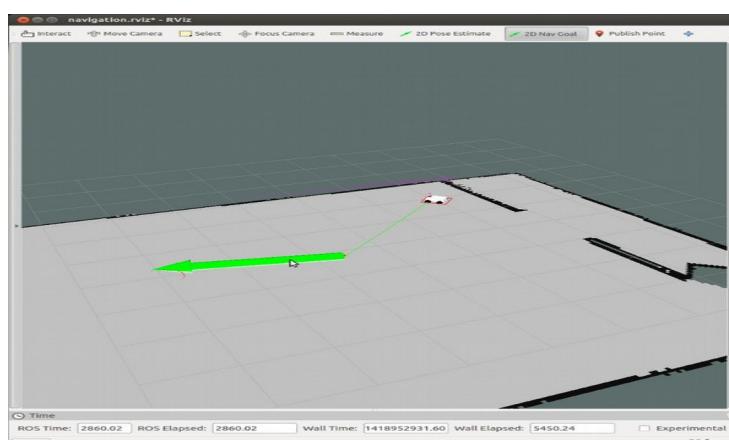
- Topic: `initialpose`
- Type: `geometry_msgs/PoseWithCovarianceStamped`



## The 2D nav goal

The 2D nav goal (G shortcut) allows the user **to send a goal to the navigation by setting a desired pose for the robot to achieve**. The navigation stack waits for a new goal with `/move_base_simple/goal` as the topic name; for this reason, you must change the topic's name in the rviz windows in Tool Properties in the 2D Nav Goal menu. The new name that you must put in this textbox is `/move_base_simple/goal`. In the next window, you can see how to use it. Click on the 2D Nav Goal button, and select the map and the goal for your robot. You can select the x and y position and the end orientation for the robot:

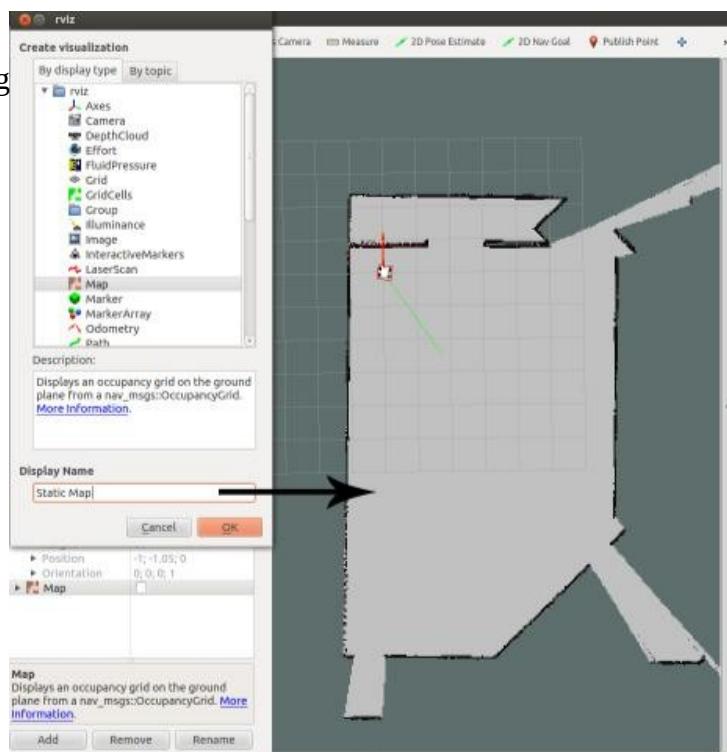
- Topic: `move_base_simple/goal`
- Type: `geometry_msgs/PoseStamped`



## The static map

This displays the static map that is being served by map\_server , if one exists. When you add this visualization, you will see the map that is pre-built.

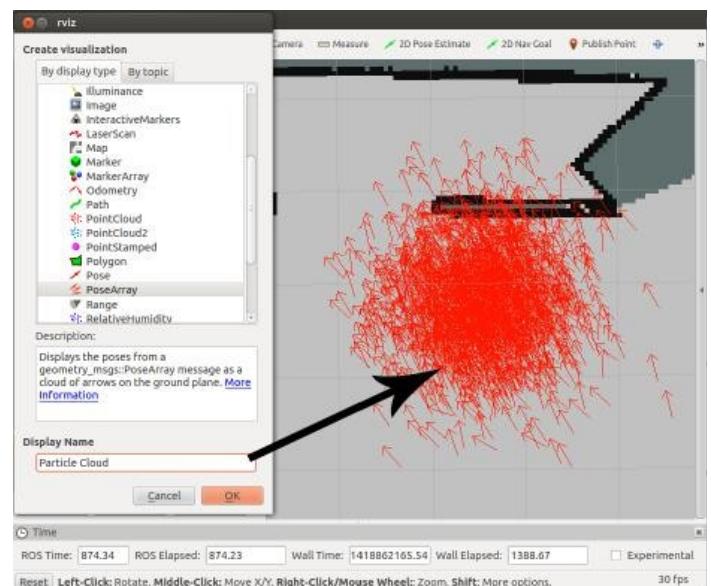
- **Topic:** map
- **Type:** nav\_msgs/GetMap



## The particle cloud

This displays the particle cloud used by the robot's localization system. The spread of the cloud represents the localization system's uncertainty about the robot's pose. A cloud that spreads out a lot reflects high uncertainty, while a condensed cloud represents low uncertainty.

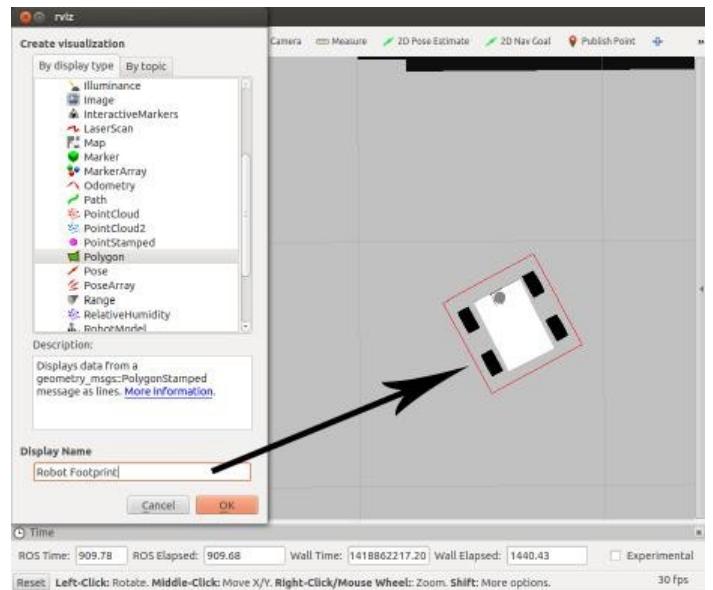
- **Topic:** particlecloud
- **Type:** geometry\_msgs/PoseArray



## The robot's footprint

This shows the footprint of the robot. Remember that this parameter is configured in the costmap\_common\_params file. This dimension is important because the navigation stack will move the robot in a safe mode using the values configured before:

- **Topic:** local\_costmap/robot\_footprint
- **Type:** geometry\_msgs/Polygon



## The local costmap

This shows the local costmap that the navigation stack uses for navigation. The yellow line is the detected obstacle. For the robot to avoid collision, the robot's footprint should never intersect with a cell that contains an obstacle. The blue zone is the inflated obstacle. To avoid collisions, the center point of the robot should never overlap with a cell that contains an inflated obstacle:

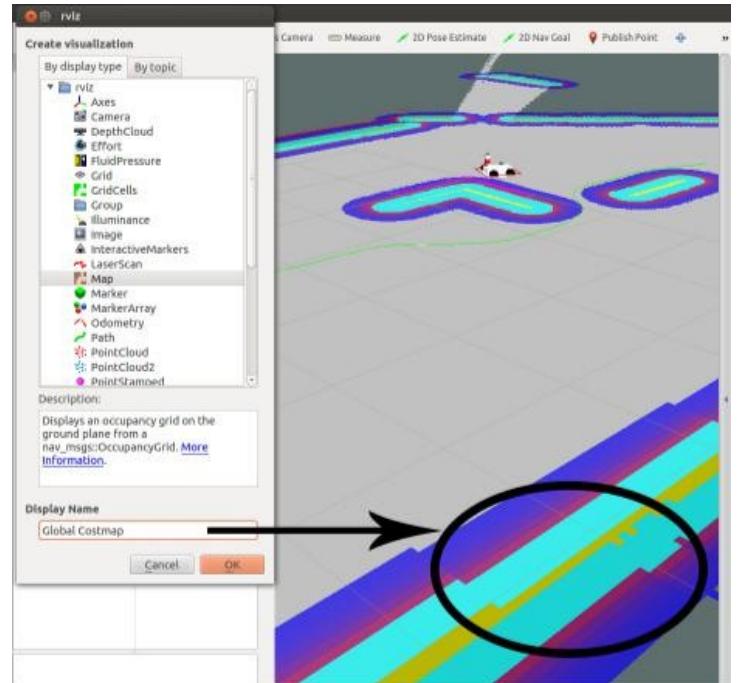
- **Topic:** move\_base/local\_costmap/costmap
- **Type:** nav\_msgs/OccupancyGrid



## The global costmap

This shows the global costmap that the navigation stack uses for navigation. The yellow line is the detected obstacle. For the robot to avoid collision, the robot's footprint should never intersect with a cell that contains an obstacle. The blue zone is the inflated obstacle. To avoid collisions, the center point of the robot should never overlap with a cell that contains an inflated obstacle:

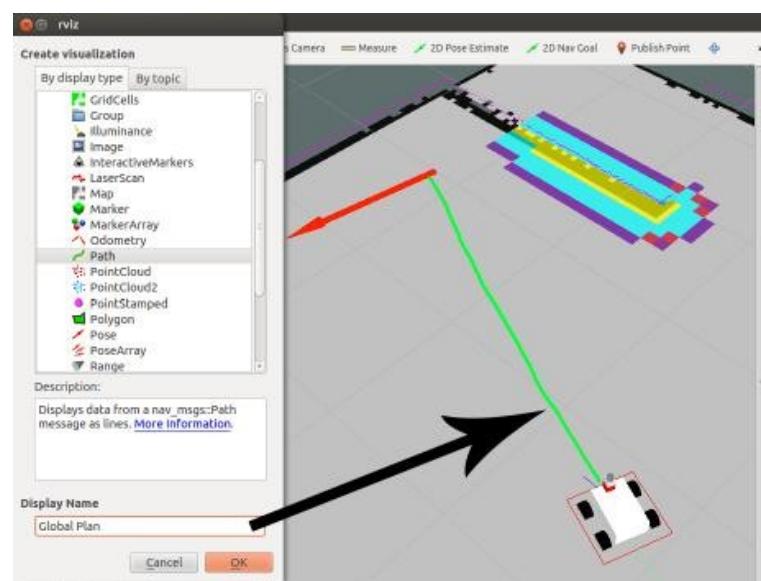
- Topic:  
/move\_base/global\_costmap/costmap
- Type: nav\_msgs/OccupancyGrid



## The global plan

This shows the portion of the global plan that the local planner is currently pursuing. You can see it in green in the next image. Perhaps the robot will find obstacles during its movement, and the navigation stack will recalculate a new path to avoid collisions and try to follow the global plan.

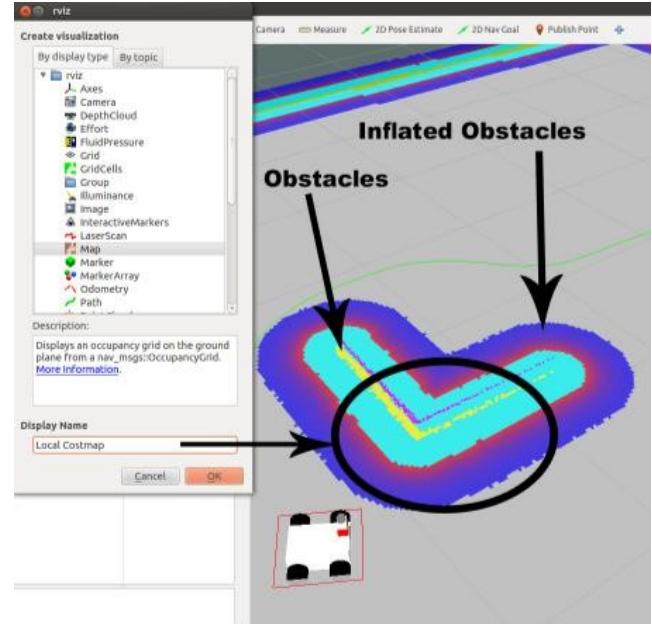
- Topic: TrajectoryPlannerROS/global\_plan
- Type: nav\_msgs/Path



## The local plan

This shows the trajectory associated with the velocity commands currently being commanded to the base by the local planner. You can see the trajectory in blue in front of the robot in the next image. You can use this display to see whether the robot is moving, and the approximate velocity from the length of the blue line:

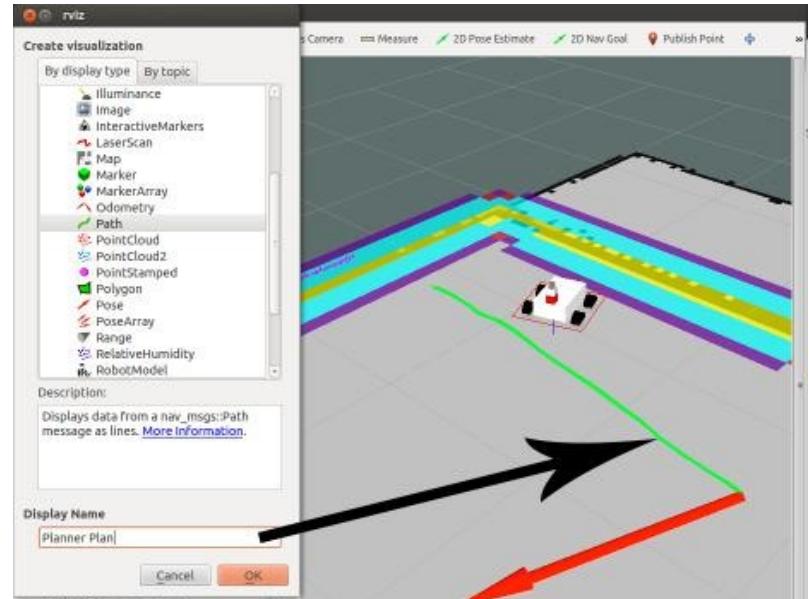
- Topic: TrajectoryPlannerROS/local\_plan
- Type: nav\_msgs/Path



## The planner plan

This displays the full plan for the robot computed by the global planner. You will see that it is similar to the global plan:

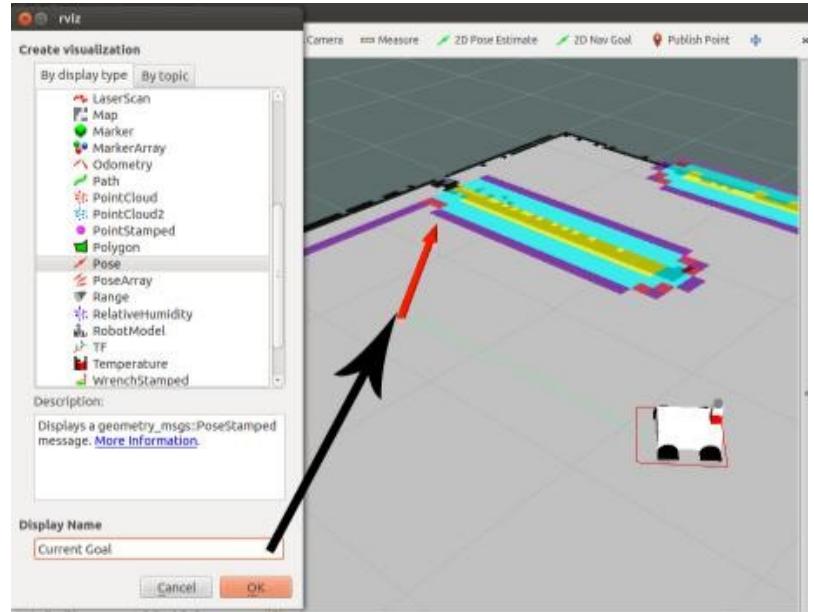
- Topic: NavfnROS/plan
- Type: nav\_msgs/Path



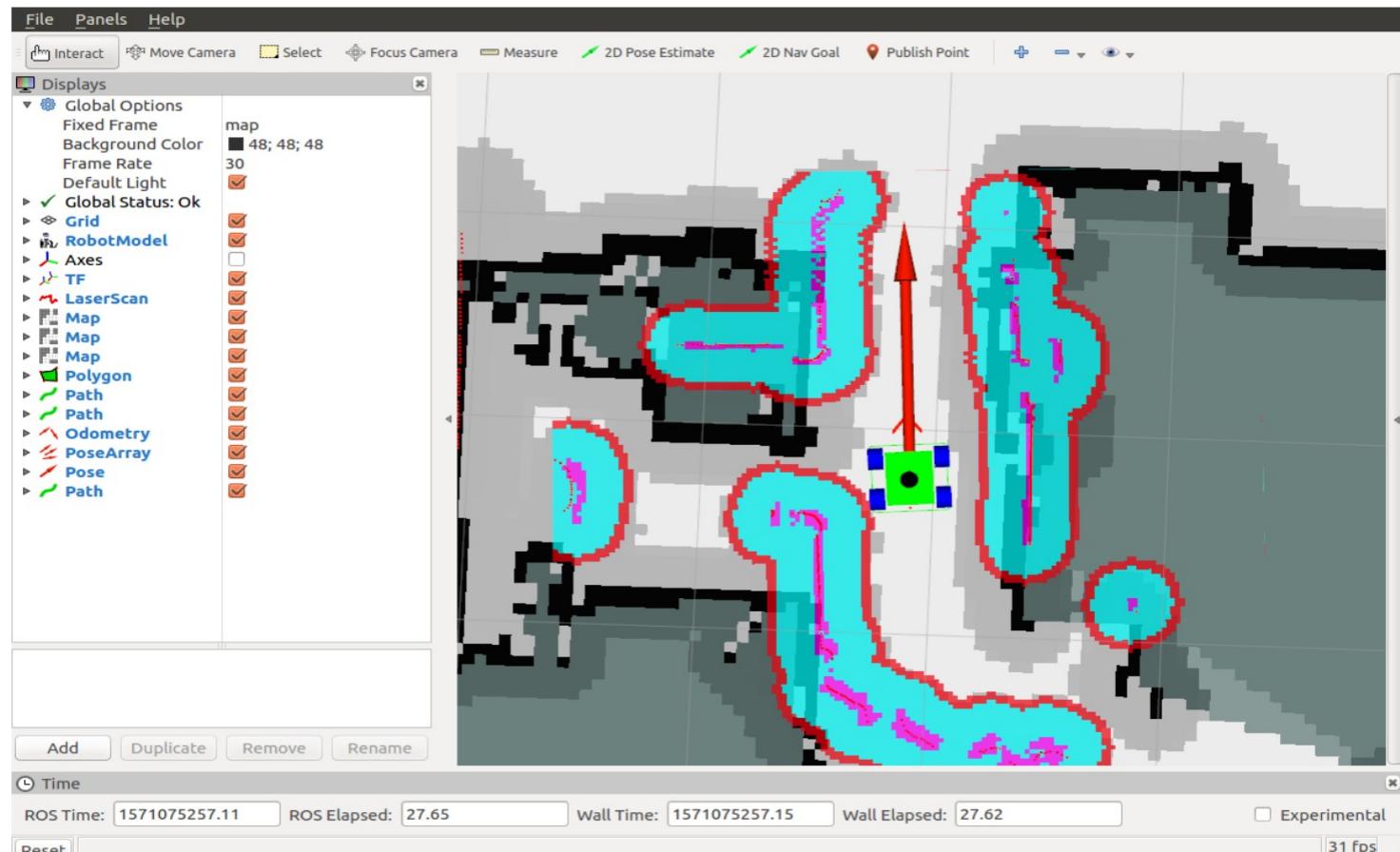
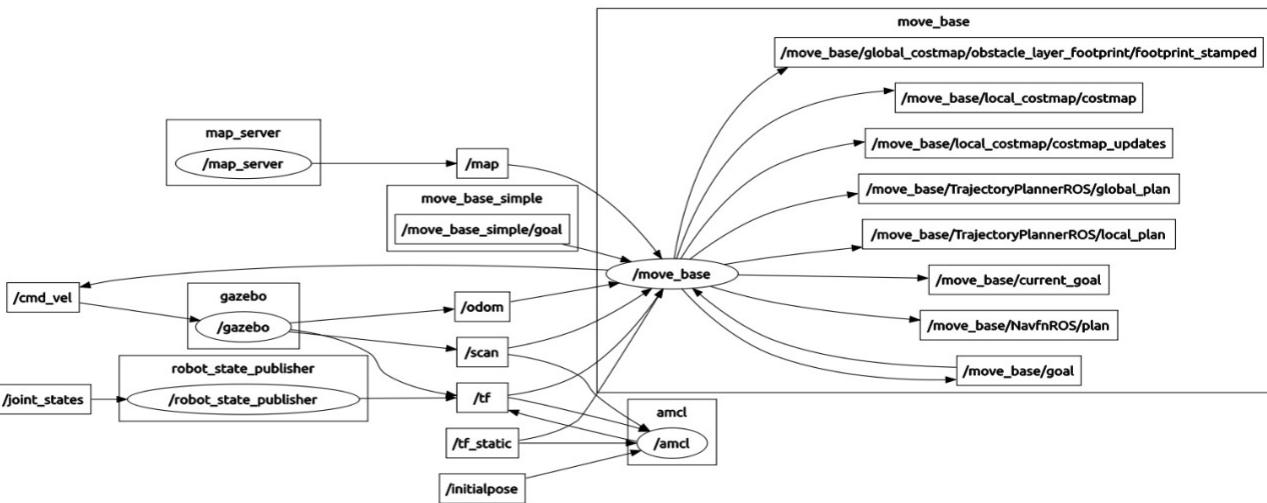
## The current goal

This shows the goal pose that the navigation stack is attempting to achieve. You can see it as a red arrow, and it is displayed after you put in a new 2D nav goal. It can be used to find out the final position of the robot:

- Topic: current\_goal
- Type: geometry\_msgs/PoseStamped



These visualizations are all you need to see the navigation stack in rviz . With this, you can notice whether the robot is doing something strange. Now we are going to see a general image of the system. Run `rqt_graph` to see whether all the nodes are running and to see the relations between them.



## 12.7 Modifying parameters with rqt\_reconfigure

A good option for understanding all the parameters configured in this chapter, is by using rqt\_reconfigure to change the values without restarting the nodes. To launch rqt\_reconfigure , use the following command:

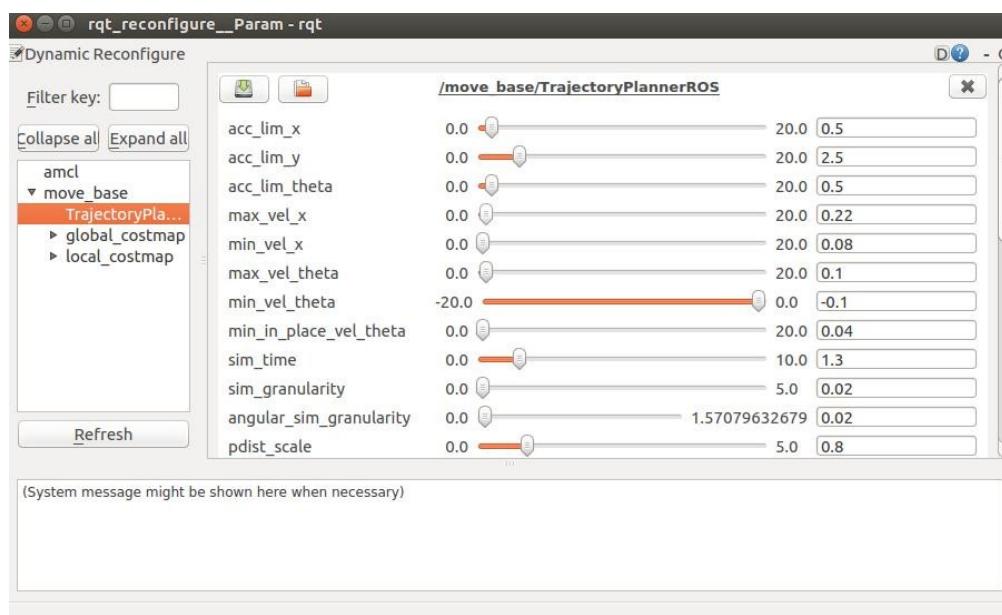
```
$ rosrun rqt_reconfigure rqt_reconfigure
```

You will see the screen as follows:



As an example, we are going to change the parameter max\_vel\_x configured in the file, base\_local\_planner.yaml . Click over the move\_base menu and expand it. Then select TrajectoryPlannerROS in the menu tree. You will see a list of parameters. As you can see, the max\_vel\_x parameter has the same value that we assigned in the configuration file.

You can see a brief description for the parameter by hovering the mouse over the name for a few seconds. This is very useful for understanding the function of each parameter.



## 12.8 Avoiding obstacles

A great functionality of the navigation stack is the recalculation of the path if it finds obstacles during the movement. You can easily see this feature by adding an object in front of the robot. In our case I added a wood board in front of the robot. The navigation stack detects the new obstacle, and automatically creates an alternative path.

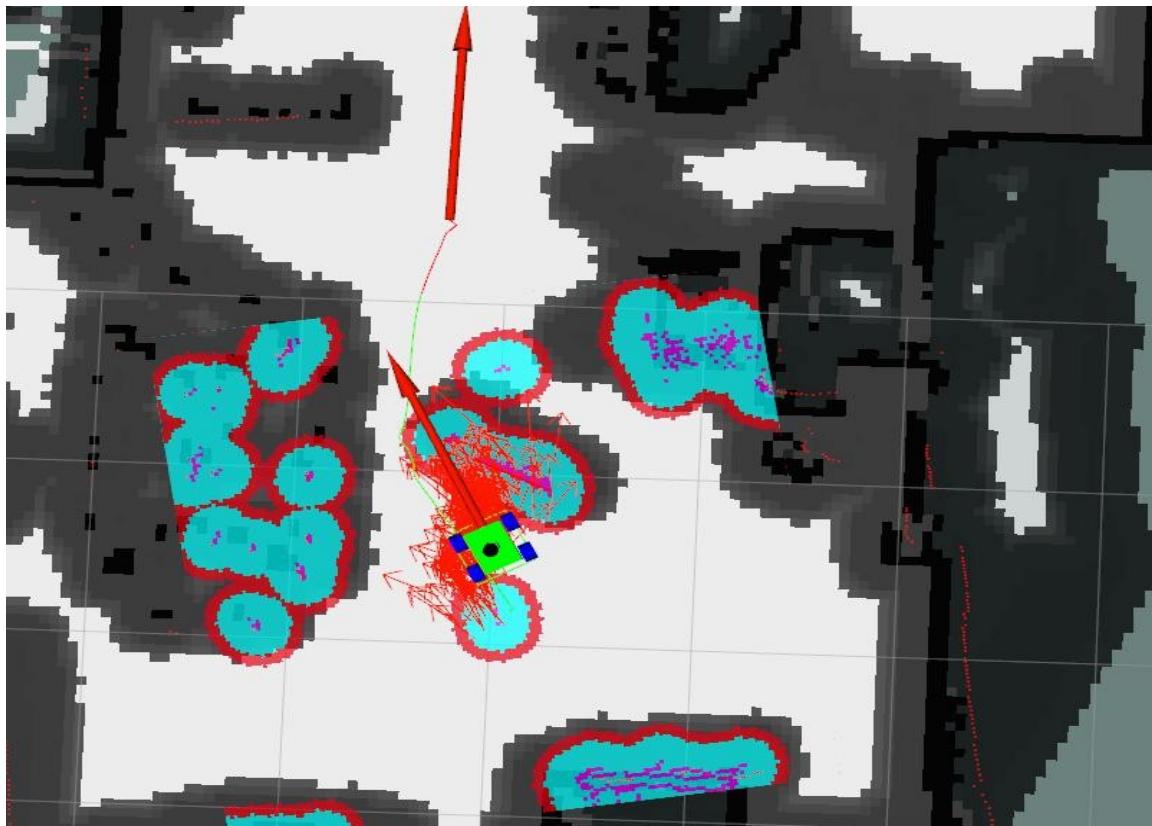
In the image we can see this newly added obstacle:

Now if we go to the rviz window, we will see a new global plan to avoid the obstacle. This feature is very useful when you use the robot in real environments with people walking around the robot. If the robot detects a possible collision, it will change the direction, and it will try to arrive at the goal.

Remember that the detection of such obstacles is reduced to the area covered by the local planner costmap (for example 2.5 x 2.5 meters around the robot)



We can see this feature in the next screenshot:



## 12.9 Sending goals programmatically

We are sure that you have been playing with the robot by moving it around the map a lot. This is funny but a little tedious, and it is not very functional. Perhaps you were thinking that it would be a great idea to program a list of movements and send the robot to different positions with only a button, even when we are not in front of a computer with rviz .

**Okay, now we are going to learn how to do it using actionlib and make our robot patrol an area.**

The actionlib package provides a standardized interface for interfacing with tasks. For example, you can use it to send goals for the robot to detect something at a place, make scans with the laser, and so on. In our case, we will send a goal to the robot to move into a certain location, and we will wait for this task to end, then we will tell the robot to move into another point and repeat, thus patrolling an area.

It could look similar to services, but if you are doing a task that has a long duration, you might want the ability to cancel the request during the execution, or get periodic feedback about how the request is progressing. You cannot do this with services. Furthermore, actionlib creates messages (not services), and it also creates topics, so we can still send the goals through a topic without taking care of the feedback and the result, if we do not want to.

```
#!/usr/bin/env python
import rospy
import actionlib
import tf
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal

# the list of points to
patrol waypoints = [
    ['one', (3.14, -0.347, 0.1)],
    ['two', (5.5857, 0.05, 0.9995)]
]

class Patrol:

    def __init__(self):
        self.client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
        self.client.wait_for_server()

    def set_goal_to_point(self, point):

        goal = MoveBaseGoal()
        goal.target_pose.header.frame_id = "map"
        goal.target_pose.header.stamp = rospy.Time.now()
        goal.target_pose.pose.position.x = point[0]
        goal.target_pose.pose.position.y = point[1]
```

```

quaternion = tf.transformations.quaternion_from_euler(0.0, 0.0,
point[2]) goal.target_pose.pose.orientation.x = quaternion[0]
goal.target_pose.pose.orientation.y = quaternion[1]
goal.target_pose.pose.orientation.z = quaternion[2]
goal.target_pose.pose.orientation.w = quaternion[3]

self.client.send_goal(goal)
wait = self.client.wait_for_result()
if not wait:
    rospy.logerr("Action server not available!")
    rospy.signal_shutdown("Action server not available!")
else:
    return self.client.get_result()

if __name__ == '__main__':
    rospy.init_node('patrolling')
    try:
        p = Patrol()
        while not rospy.is_shutdown():
            for i, w in enumerate(waypoints):
                rospy.loginfo("Sending waypoint %d - %s", i, w[0])
                p.set_goal_to_point(w[1])
    except rospy.ROSInterruptException:
        rospy.logerr("Something went wrong when sending the waypoints")

```

**Now the robot should move into the waypoints sequentially**, you can get this points by moving the robot to the desired location and get the point on the map that it is, by the rviz transform display map-> base\_link

You can make a list of goals or waypoints, and create a route for the robot. This way you can program missions, guardian robots, or collect things from other rooms with your robot.

## 12.10 Summary

Now you should be able to have a robot-simulated or real-moving autonomously through the map (which models the environment), using the navigation stack. You can program the control and the localization of the robot by following the ROS philosophy of code re-usability, so that you can have the robot completely configured without much effort. The most difficult part is to understand all the parameters and learn how to use each one of them appropriately. The correct use of them will determine whether your robot works fine or not; for this reason, you must practice changing the parameters and look for the reaction of the robot.

# 13. 3D Perception – experimental results

## 13.1 Point Cloud Pre-processing Filtering

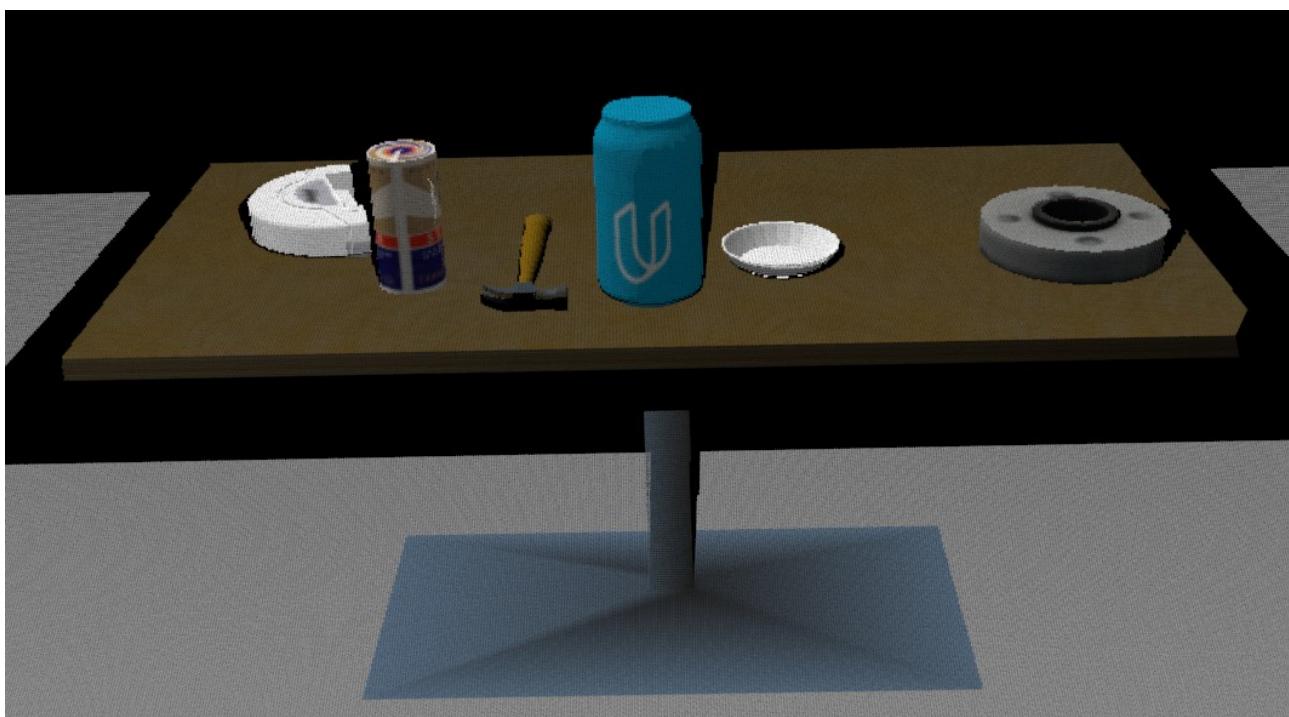
We will start by showing the Point Cloud that we will use for preprocessing, it consist of a table with a variety of different objects on top.

PCL library comes with a simplistic point cloud viewer usefull to visualize point cloud in a .pcd format.

If we run the command

```
$ pcl_viewer tabletop.pcd
```

We can visualize our point cloud



As we saw in the theory the step we will take are as follow

1. **Voxel Grid filter** - Downampling
2. **Passthrough filter** - Cropping
3. **RANSAC plane fitting** - To remove the table from the point cloud
4. **Extract indices** - To get the table and object pointcloud from the indices that RANSAC gives us
5. Optional Outlier removal, we will not use this filter since our Point cloud doesn't have Outliers

```

# Import PCL module
import pcl
# Load Point Cloud file
cloud=pcl.load_XYZRGB("tabletop.pcd")

# STEP 1 : Voxel Grid filter

# Create a VoxelGrid filter object for our input point cloud
vox=cloud.make_voxel_grid_filter()
# Choose a voxel (also known as leaf) size
# Note: this (0.01) is a decent choice of leaf size
LEAF_SIZE=0.01
# Set the voxel (or leaf) size
vox.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)
# Call the filter function to obtain the resultant downsampled point cloud
cloud_filtered=vox.filter()
# Save the Downsampled Point Cloud
filename = 'voxel_downsampled.pcd'
pcl.save(cloud_filtered, filename)

# STEP 2 : PassThrough filter

# Create a PassThrough filter object.
passthrough=cloud_filtered.make_passthrough_filter()
# Assign axis and range to the passthrough filter object.
filter_axis='z'
passthrough.set_filter_field_name(filter_axis)
axis_min=0.6
axis_max=1.1
passthrough.set_filter_limits(axis_min, axis_max)
# Finally use the filter function to obtain the resultant point cloud.
cloud_filtered=passthrough.filter()
# Save the Cropped Point Cloud
filename = 'pass_through_filtered.pcd'
pcl.save(cloud_filtered, filename)

# STEP 3 : RANSAC plane segmentation

# Create the segmentation object
seg=cloud_filtered.make_segmenter()
# Set the model you wish to fit
seg.set_model_type(pcl.SACMODEL_PLANE)
seg.set_method_type(pcl.SAC_RANSAC)

# Max distance for a point to be considered fitting the model
# Experiment with different values for max_distance
# for segmenting the table
max_distance=0.01
seg.set_distance_threshold(max_distance)
# Call the segment function to obtain set of inlier indices and model
coefficients
inliers, coefficients = seg.segment()

#STEP 4 Extract the indices for the table and objects into Point Clouds

# Extract inliers
extracted_inliers=cloud_filtered.extract(inliers, negative=False)
# Save pcd for table

```

```

filename = 'extracted_inliers_table.pcd'
pcl.save(extracted_inliers, filename)

# Extract outliers
extracted_outliers = cloud_filtered.extract(inliers, negative=True)
# Save pcd for tabletop objects
filename = 'extracted_outliers_objects.pcd'
pcl.save(extracted_outliers, filename)

'''

# STEP 5: Optional Outlier filter

# Much like the previous filters, we start by creating a filter object:
outlier_filter = cloud_filtered.make_statistical_outlier_filter()

# Set the number of neighboring points to analyze for any given point
outlier_filter.set_mean_k(50)

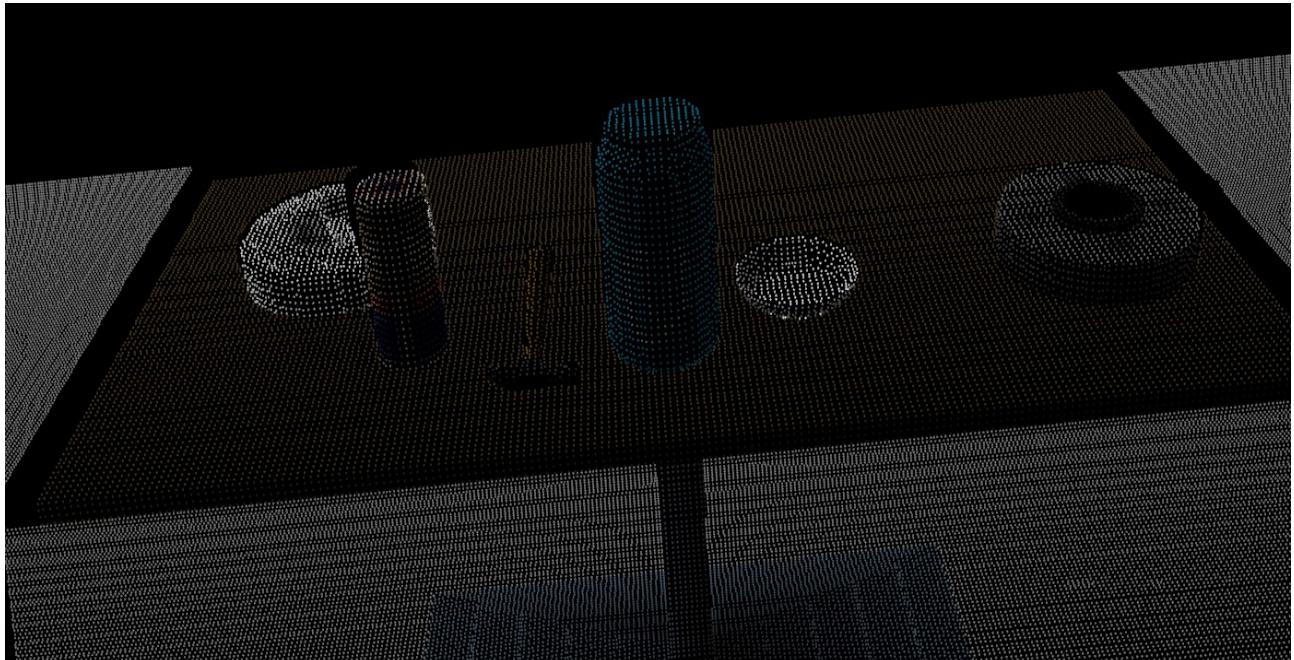
# Set threshold scale factor
x = 1.0

# Any point with a mean distance larger than global (mean distance+x*std_dev)
# will be considered outlier
outlier_filter.set_std_dev_mul_thresh(x)

# Finally call the filter function for magic
cloud_filtered = outlier_filter.filter()
'''
```

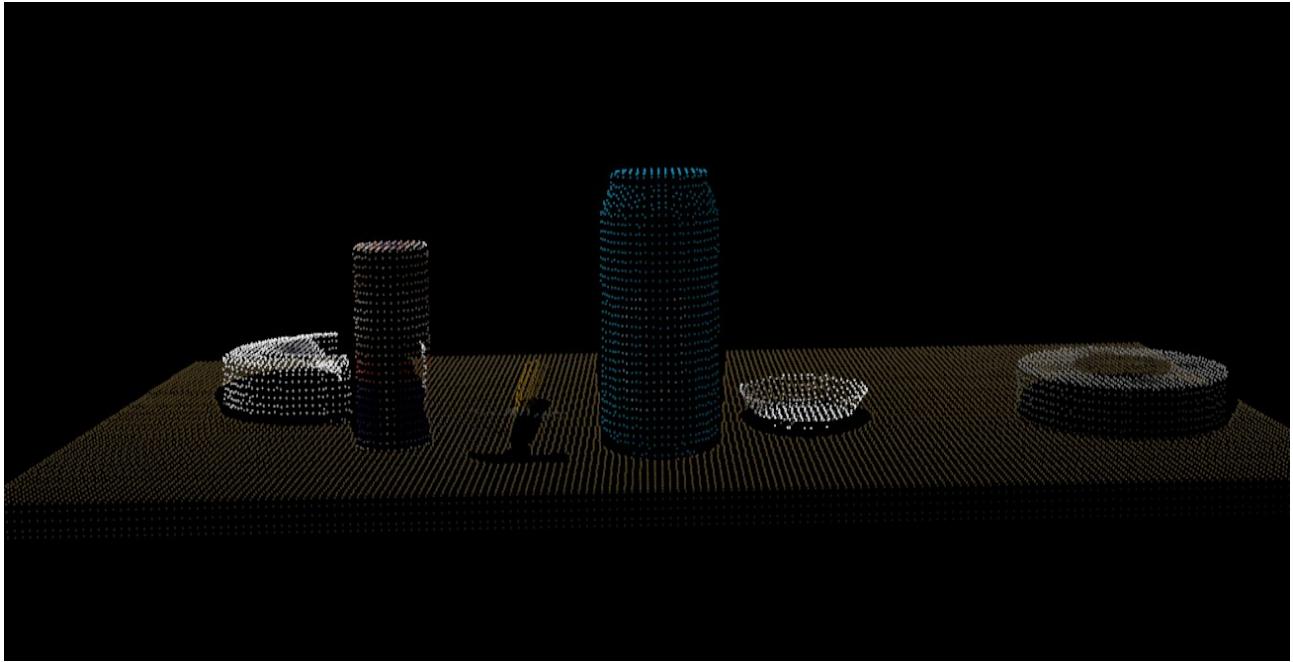
## Resulting Point Clouds

### Step 1 : Voxel Grid Filter – Downsampling



We can see that the point cloud is now more sparsed (1 point per cubic centimeter)

## **Step 2 : Passthrough Filter – Cropping**



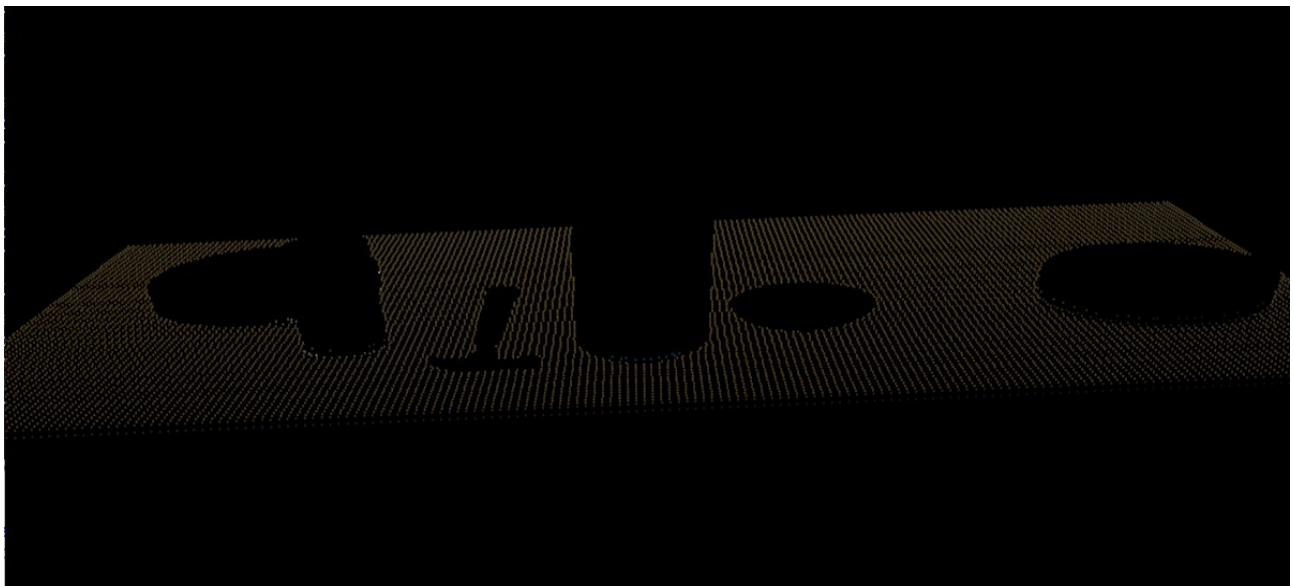
We can see that we "cropped" the point cloud removing the ground and the base of the table.

## **Step 3 : RANSAC plane segmentation**

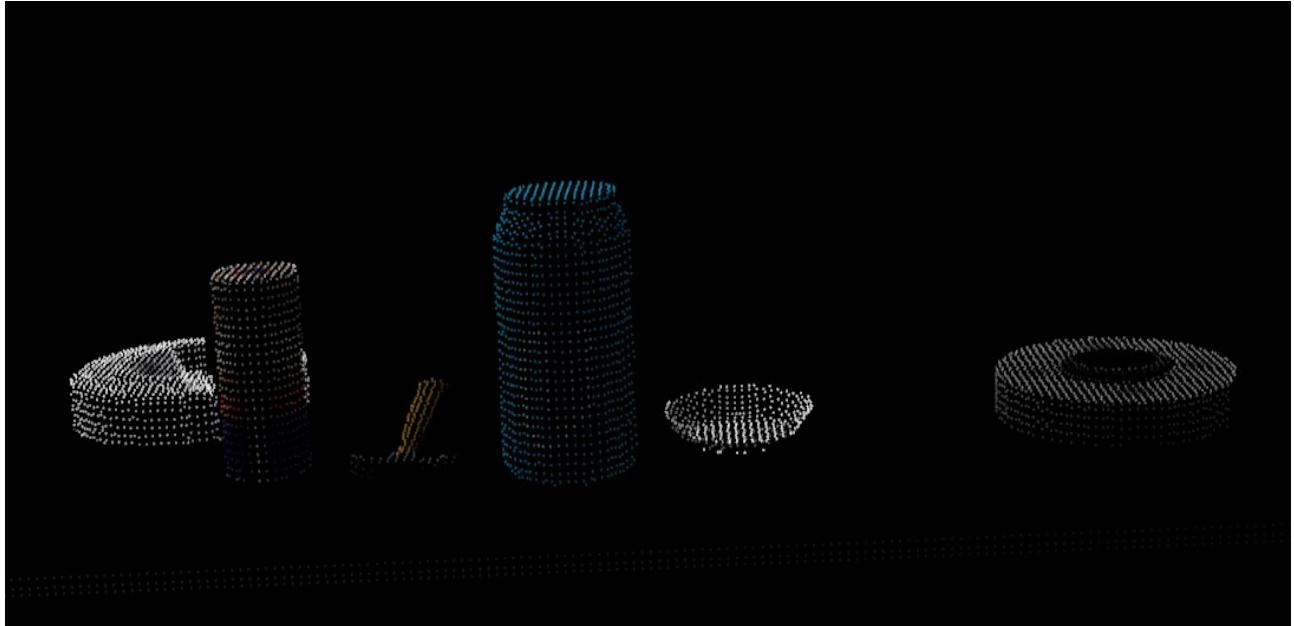
refer to the code it gives us the indices belonging to the table and the indices belonging to the objects

## **Step 4 : Extract the indices for the table and objects into Point Clouds**

### **Table point cloud**



## Objects point cloud



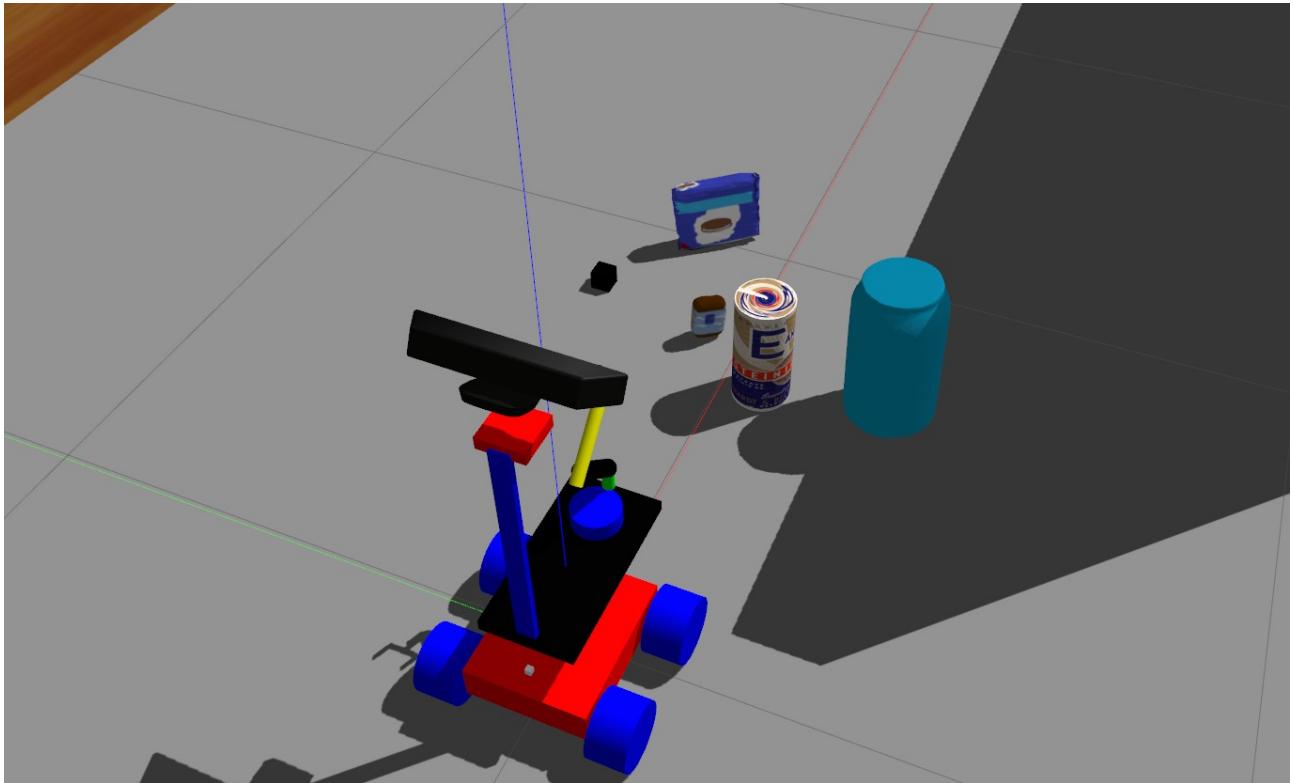
## 13.2 Point Cloud Segmentation

Great we have now pre – processed our point cloud and extracted the objects of interest from that, **but wait all the objects now belong to a single point cloud.**

Next up , we'll use a PCL library function called `EuclideanClusterExtraction()` to perform a DBSCAN cluster search on our 3D point cloud, in order to segment the objects into multiple point clouds, with each point cloud corresponding to a single object.

We will now go to ROS and Gazebo for this stage, meaning that we will launch our robot Mobile Manipulator to the Gazebo environment and we will get our Point Cloud from the Kinect camera on the `camera/depth_registered/points` topic.

Let's first see the Gazebo world we would experiment with



**You can see the entire code for the segmentation part under my github repository on the package `sensor_stick`**

**`sensor_stick/scripts/segmentation.py`**

also this node uses some `pcl_helper` functions located under

**`sensor_stick/src/sensor_stick/pcl_helper.py`**

The relevant code that we use for the Euclidean clustering after we have preprocessed the point cloud with a similar manner as before is the follow:

```
# Euclidean Clustering
white_cloud = XYZRGB_to_XYZ(cloud_objects) # Apply function to convert XYZRGB to XYZ
tree = white_cloud.make_kdtree()
# Create a cluster extraction object
ec = white_cloud.make_EuclideanClusterExtraction()
# Set tolerances for distance threshold
# as well as minimum and maximum cluster size (in points)
# NOTE: These are decent choices of clustering parameters
# Experiment and find values that work for segmenting objects.
ec.set_ClusterTolerance(0.02)
ec.set_MinClusterSize(10)
```

```

ec.set_MaxClusterSize(30000)
# Search the k-d tree for clusters
ec.set_SearchMethod(tree)
# Extract indices for each of the discovered clusters
cluster_indices = ec.Extract() # Create a cluster extraction object

# Create Cluster-Mask Point Cloud to visualize each cluster separately
# Assign a color corresponding to each segmented object in scene
cluster_color = get_color_list(len(cluster_indices))

color_cluster_point_list = []

for j, indices in enumerate(cluster_indices):
    for i, indice in enumerate(indices):
        color_cluster_point_list.append([white_cloud[indice][0],
                                         white_cloud[indice][1],
                                         white_cloud[indice][2],
                                         rgb_to_float(cluster_color[j])])

#Create new cloud containing all clusters, each with unique color
cluster_cloud = pcl.PointCloud_PointXYZRGB()
cluster_cloud.from_list(color_cluster_point_list)

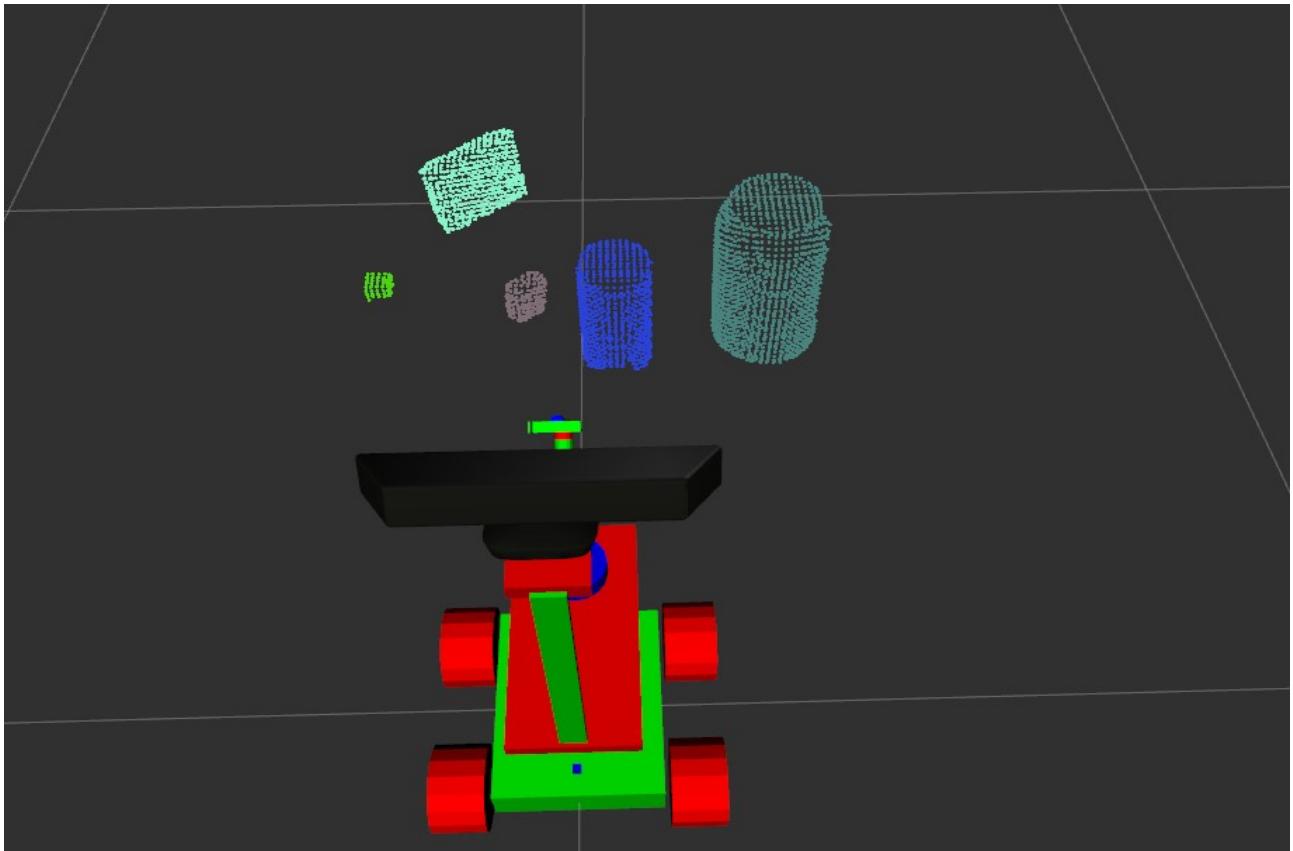
# Convert PCL data to ROS messages
ros_cloud_objects = pcl_to_ros(cloud_objects)
ros_cloud_table = pcl_to_ros(cloud_table)
ros_cluster_cloud = pcl_to_ros(cluster_cloud)
# Publish ROS messages
pcl_objects_pub.publish(ros_cloud_objects)
pcl_table_pub.publish(ros_cloud_table)
pcl_cluster_cloud.publish(ros_cluster_cloud)

```

Where we segment the point cloud containing the objects based on distance metric into many different smaller point clouds each containing a single object, we also colorize each point cloud with a random color for better visualization.

Then we convert the point cloud types into ROS messages and we publish them, in order to visualize them with RVIZ.

## **And here is the result**



Where we can see that have segmented the point cloud containing the objects, into many smaller point clouds each containing one object, while also colorizing them with random color in order to visualize them better.

### 13.3 Point Cloud Object Recognition

Great now we have segmented our point cloud and divide it into multiple pointclouds, each ideally representing a different object.

But we still don't know which object is which!!

What if for example we wanted to select and identify a particular object, let's say the soda can, we have idea still which object of our segmented object it is!

Up next we will see how we can train a classifier to recognize our objects of interest and get their 3D locations in space, which can be useful for later when we will want to grasp a particular object.

The object recognition process can be divided into 3 parts.

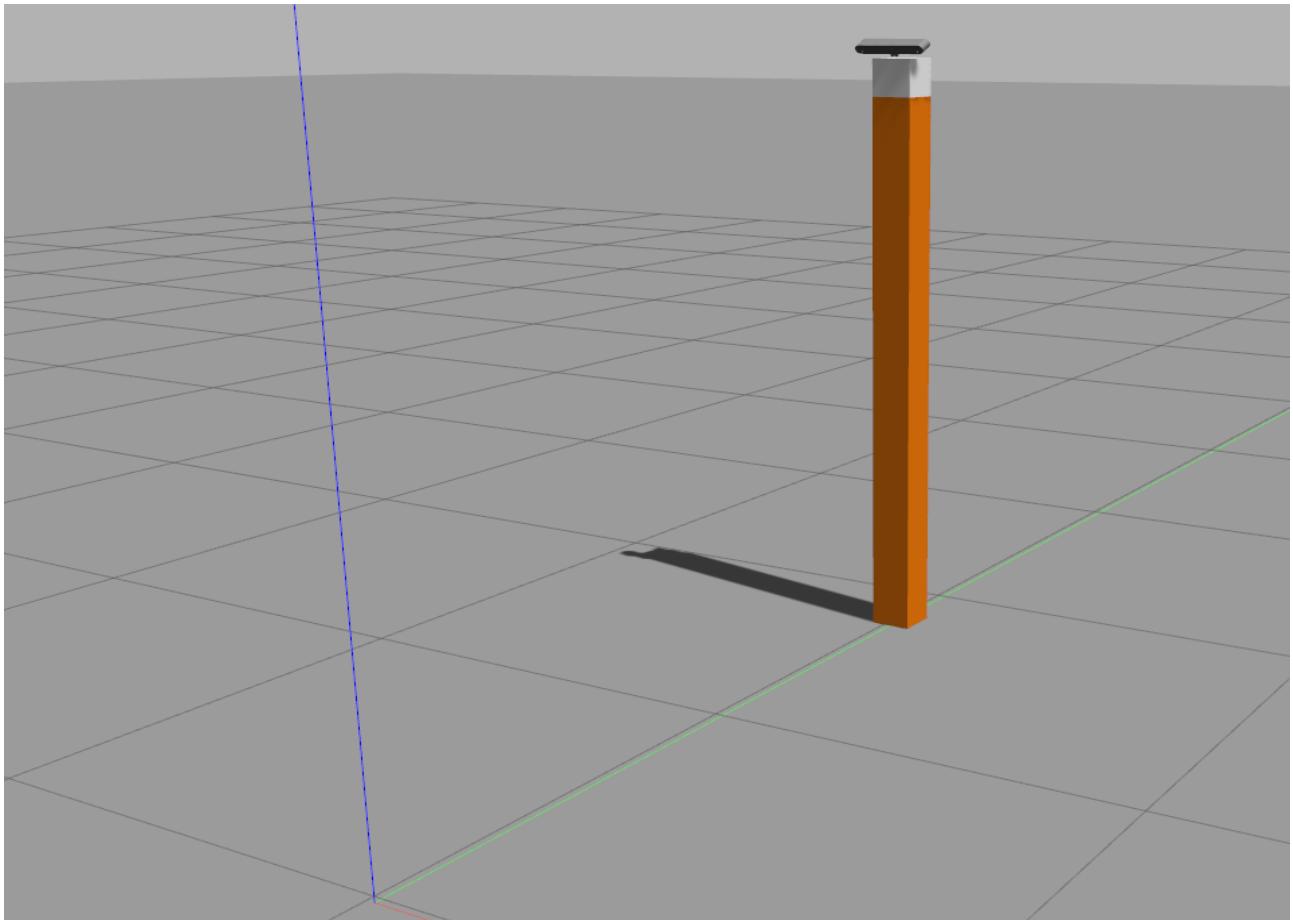
- 1) Generate features that uniquely describe the objects, in our case these features will be hsv histograms and surface normals histograms
- 2) Train a Support Vector Machine Classifier with our extracted features and our known labels meaning a training set and then test it with a test set.
- 3) Finally perform object recognition with our trained classifier.

**Let's now start with the first step and see how we can generate features**

with the command

```
$ rosrun sensor_stick training.launch
```

We launch a simple Gazebo world consisting of a sensor stick, and an RGB-D camera that can pan and tilt, we can see it below.



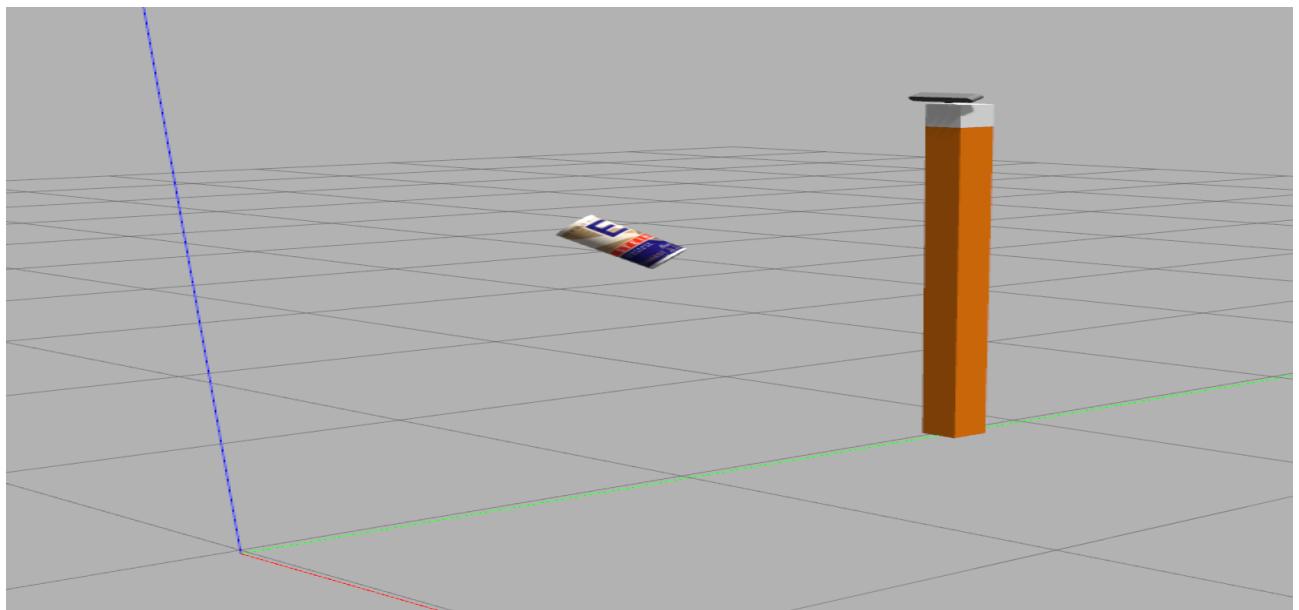
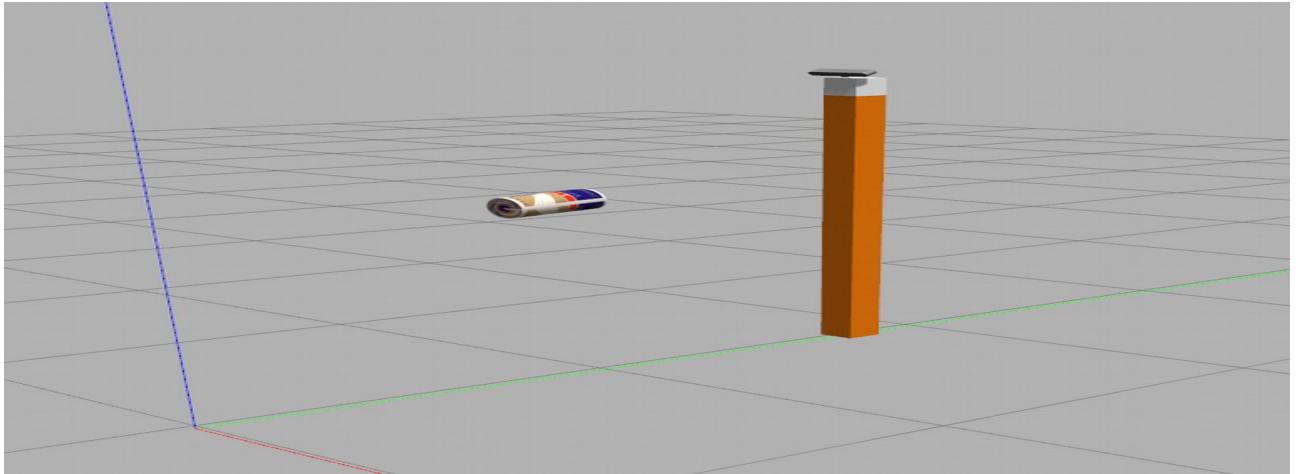
Then we can run a node that will spawn our selected objects in this world, and extract the features from them.

```
$ rosrun sensor_stick capture_features.py
```

The objects will be spawned in random orientations many times in order to capture features from multiple points of view of the object.

In my case I trained for 100 orientations for each object and for a total of 10 objects, which took about 90 minutes to complete, this was enough for moderate results as we will see next in the feature I will train for more iterations.

Now let's see how this looks like for the beer model in two orientations.



Here are the functions that this node uses to extract the HSV and Normal Features in a Histogram form.

```
def compute_color_histograms(cloud, using_hsv=False):

    # Compute histograms for the clusters
    point_colors_list = []

    # Step through each point in the point cloud
    for point in pc2.read_points(cloud, skip_nans=True):
        rgb_list = float_to_rgb(point[3])
        if using_hsv:
            point_colors_list.append(rgb_to_hsv(rgb_list) * 255)
        else:
            point_colors_list.append(rgb_list)

    # Populate lists with color values
    channel_1_vals = []
    channel_2_vals = []
    channel_3_vals = []
```

```

for color in point_colors_list:
    channel_1_vals.append(color[0])
    channel_2_vals.append(color[1])
    channel_3_vals.append(color[2])

# Compute histograms
nbins=32
bins_range=(0, 256)

hist_1 = np.histogram(channel_1_vals, bins=nbins, range=bins_range)
hist_2 = np.histogram(channel_2_vals, bins=nbins, range=bins_range)
hist_3 = np.histogram(channel_3_vals, bins=nbins, range=bins_range)
# Concatenate and normalize the histograms
hist_conc = np.concatenate((hist_1[0], hist_2[0], hist_3[0])).astype(np.float64)
normed_features = hist_conc / np.sum(hist_conc)
# Return normed_features
return normed_features


def compute_normal_histograms(normal_cloud):
    norm_x_vals = []
    norm_y_vals = []
    norm_z_vals = []

    for norm_component in pc2.read_points(normal_cloud,
                                           field_names = ('normal_x', 'normal_y', 'normal_z'),
                                           skip_nans=True):
        norm_x_vals.append(norm_component[0])
        norm_y_vals.append(norm_component[1])
        norm_z_vals.append(norm_component[2])

    # Compute histograms of normal values (just like with color)
    nbins=32
    bins_range=(-1, 1)
    norm_x_hist = np.histogram(norm_x_vals, bins=nbins, range=bins_range)
    norm_y_hist = np.histogram(norm_y_vals, bins=nbins, range=bins_range)
    norm_z_hist = np.histogram(norm_z_vals, bins=nbins, range=bins_range)
    # Concatenate and normalize the histograms
    hist_features = np.concatenate((norm_x_hist[0], norm_y_hist[0],
                                    norm_z_hist[0])).astype(np.float64)
    normed_features = hist_features / np.sum(hist_features)
    # Return normed_features

    return normed_features

```

After this node has finished executing and we have trained for all our selected objects in all orientations it **will output our features in a training\_set.sav file** that **our classifier will now use to train our object recognition pipeline.**

**Now that we have our features in the training\_Set.sav file it's to go to the next step.**

## TRAIN OUR SVM CLASSIFIER – LINEAR KERNEL

By running the next node with the command:

**\$ rosrun sensor\_stick train\_svm.py**

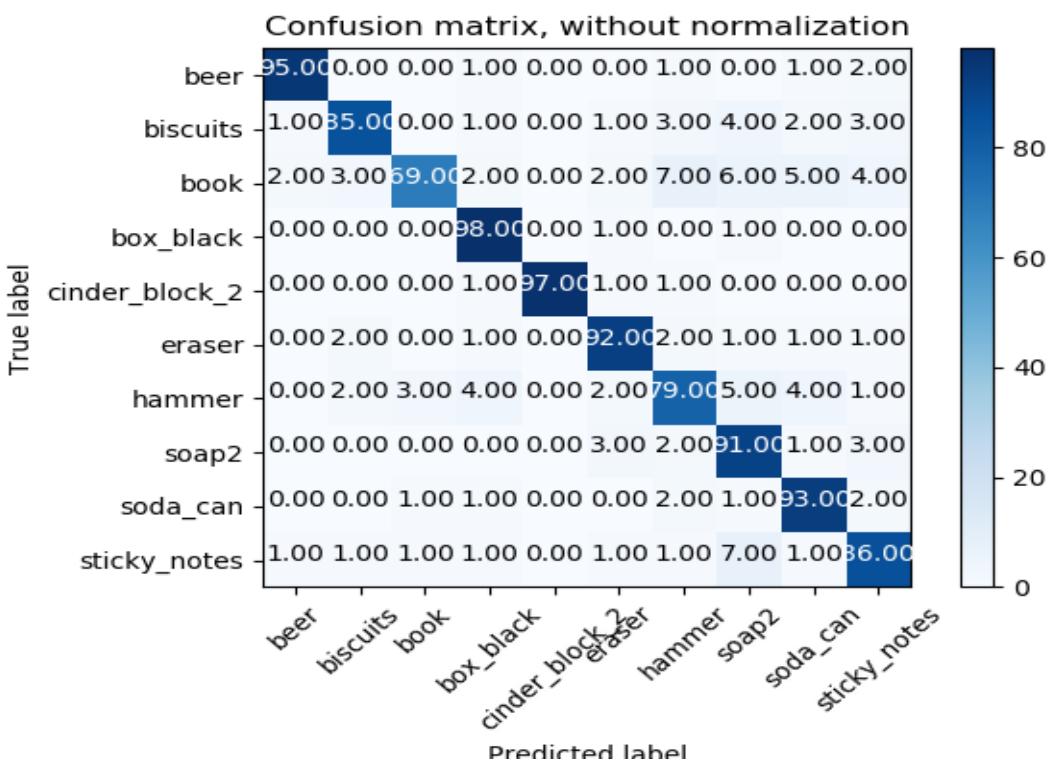
We will train an SVM classifier which uses a linear kernel (with the help of the sklearn library) (here you can experiment with different kernels for better results but in our case a linear case performed good), with our training features extracted before.

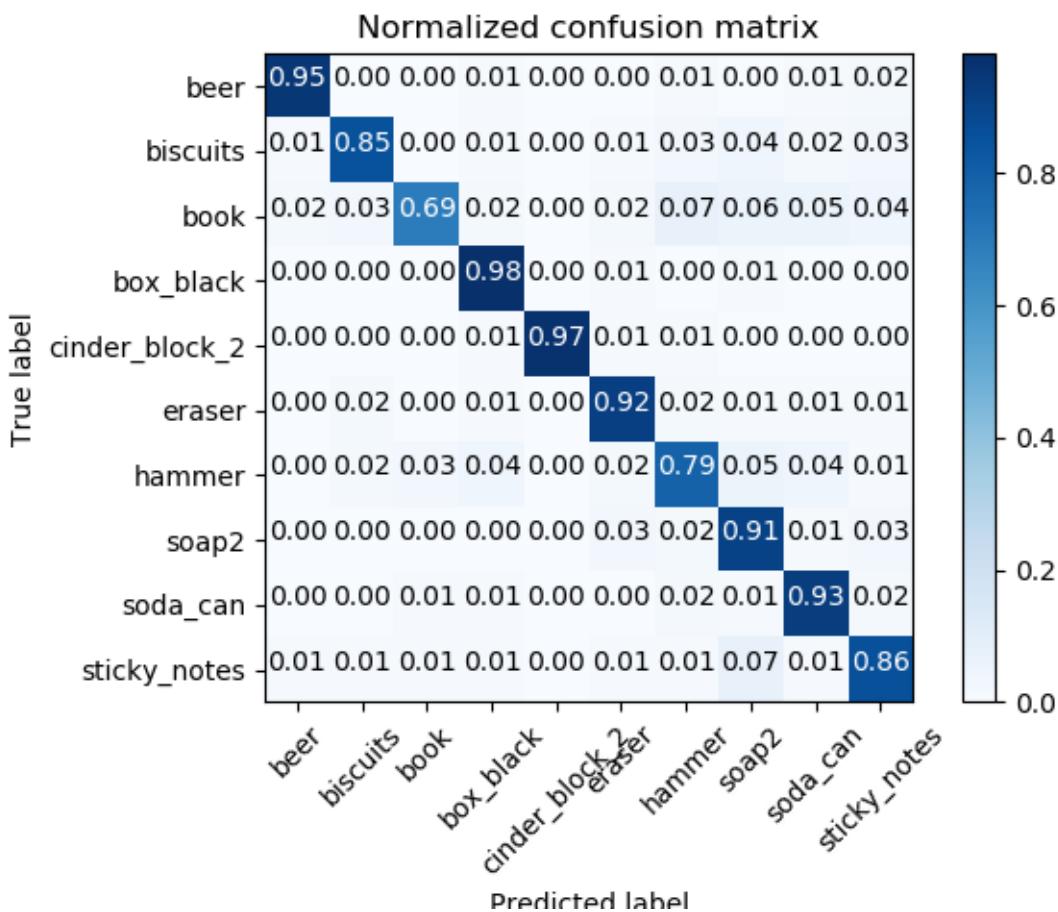
**After we train our classifier the node will output the resulting model classifier in a model.sav for our object recognition pipeline to use later.**

**Furthermore when this node runs it will output some text on the terminal regarding overall accuracy of our classifier.**

```
Terminal
File Edit View Search Terminal Help
makemelive ~ $ rosrun sensor_stick train_svm.py
Features in Training Set: 1000
Invalid Features in Training set: 0
Scores: [ 0.875  0.865  0.92   0.87   0.895]
Accuracy: 0.89 (+/- 0.04)
accuracy score: 0.885
```

**Lastly it will also output the following plots which show the relative accuracy of our classifier for the objects we trained on.**





These plots are showing us two different versions of the [confusion matrix](#) for our classifier. Above is the raw counts and below as a percentage of the total.

Here are some examples on how to read this confusion matrices

- Number of times the "hammer" was misclassified as a "biscuits" : Answer – 2
- Number of times the "beer" was misclassified as a "hammer" : Answer – 1
- Number of times the "beer" was misclassified as a "box\_black" : Answer – 1

Again this results can be improved if we train for more than just 100 orientations per object.

**NOW WE ARE READY FOR THE FINAL STEP OF OUR RECOGNITION PIPELINE**

**USE OUR TRAINED CLASSIFIER TO RECOGNIZE OBJECTS IN THE GAZEBO SIMULATION PUT LABELS IN RVIZ AND TF FOR 3D LOCATION.**

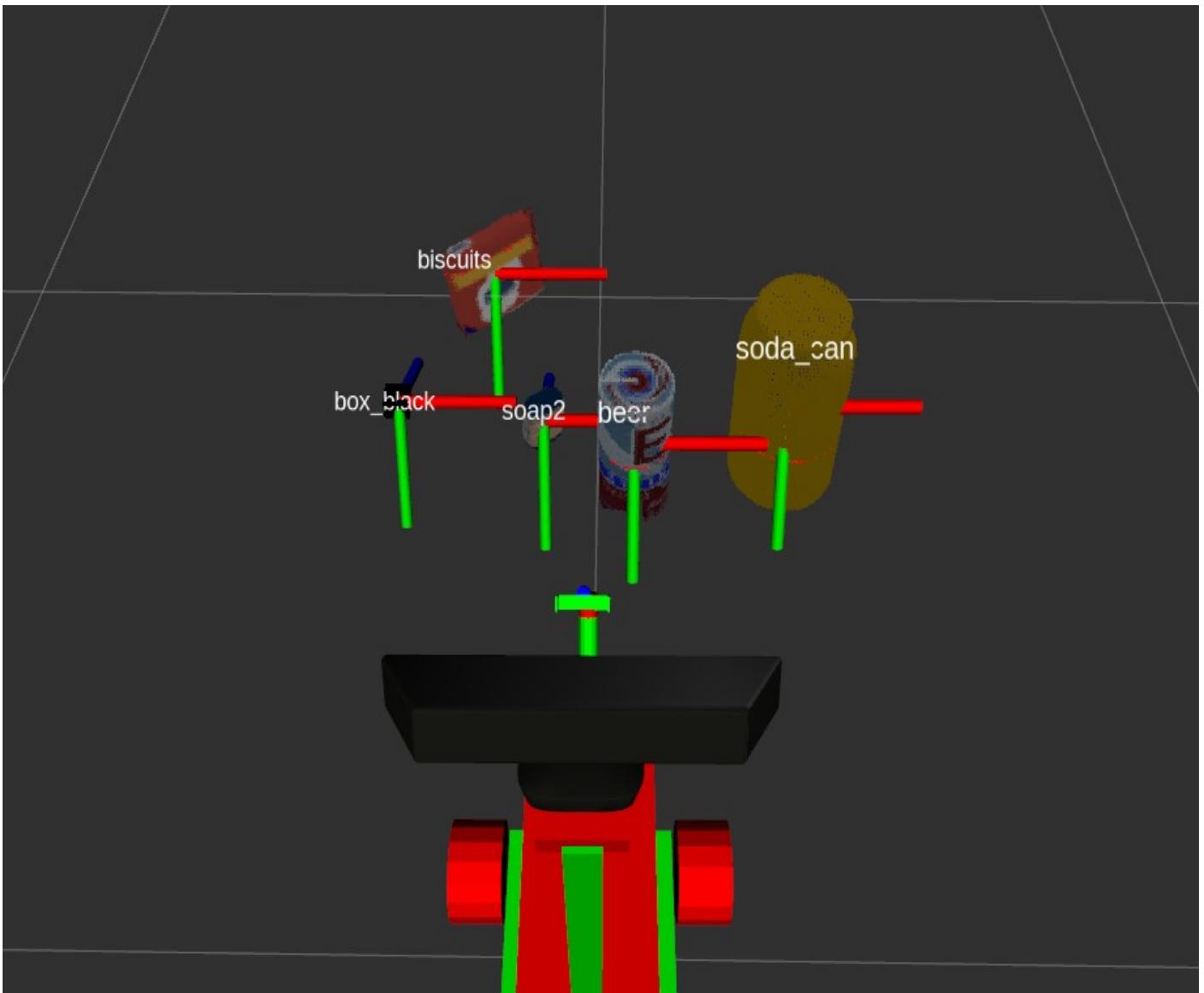
Below we can see the Mobile Manipulator robot in a Gazebo Environment with several objects in front of it.



**After we run the node located at my github repository under  
sensor\_stick/script/object\_recognition.py**

**\$ rosrun sensor\_stick object\_recognition.launch**

**We can see the result in RVIZ**



From the image above we can see that the objects are correctly identified, so we know which object is which.

Markers in RVIZ were added to display the corresponding text displaying the name of each object for our convenience.

Lastly we have calculated the centroid position for each object and published it's Transformation Frame or TF so later to be able to easily get the 3D location of each object, which can be useful for Pick and Place Tasks with our Arm.

## 14.1 Kinematics – experimental results

### 14.1 Creating an IK server for the KUKA KR210



### Forward Kinematic Analysis

Extracting joint positions and orientations from URDF file.

from the URDF file `kr210.urdf.xacro` we can extract the position xyz and orientation rpy of each joint from `origin` tag in each joint XML section:

for example, from the following fixed base joint XML section:

```
<!-- joints -->
<joint name="fixed_base_joint" type="fixed">
  <parent link="base_footprint"/>
  <child link="base_link"/>
  <origin xyz="0 0 0" rpy="0 0 0"/>
</joint>
```

In the **origin** tag:

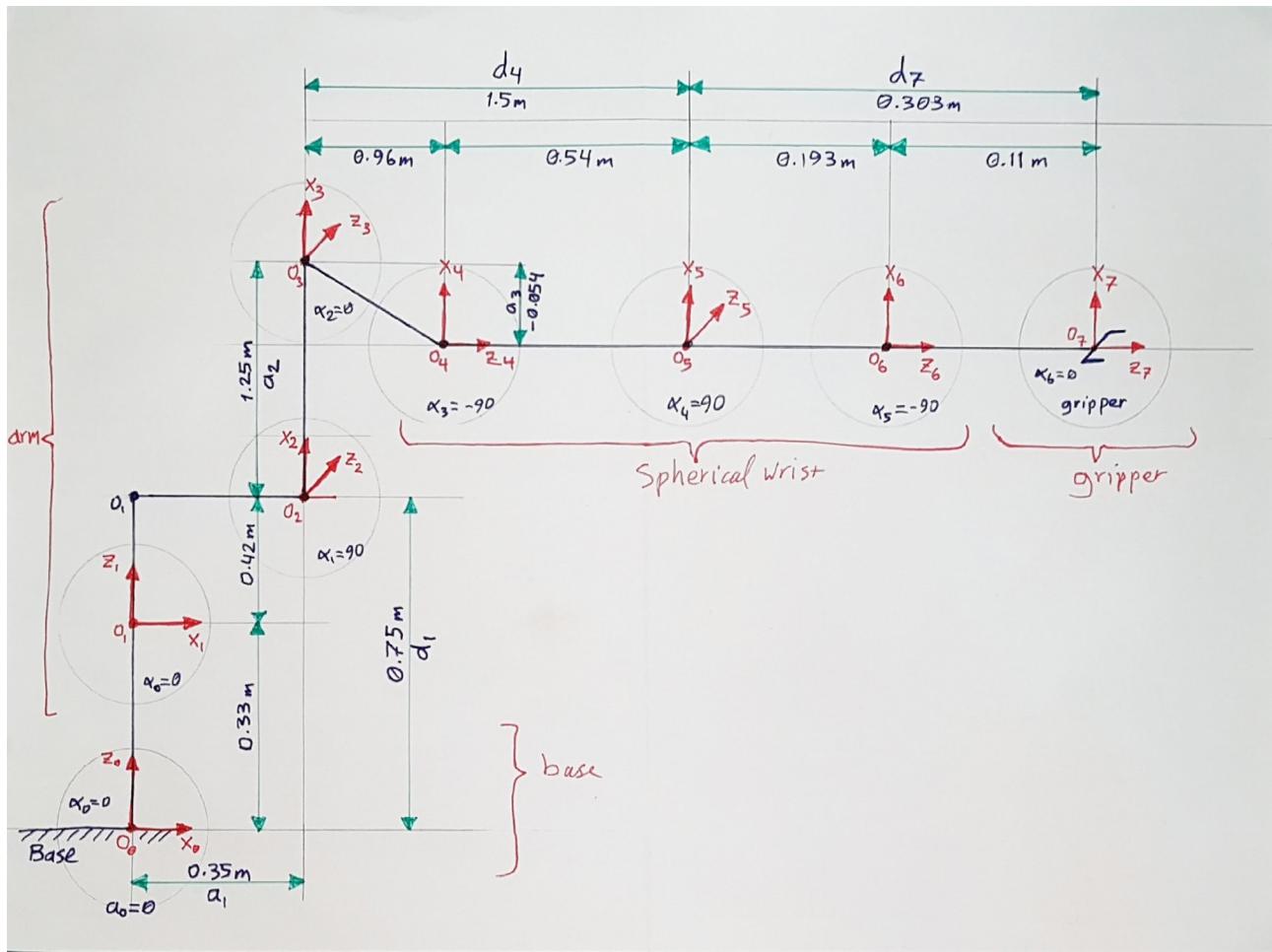
```
<origin xyz="0 0 0" rpy="0 0 0"/>
```

We can see that **xyz="0 0 0"** and **rpy="0 0 0"**.

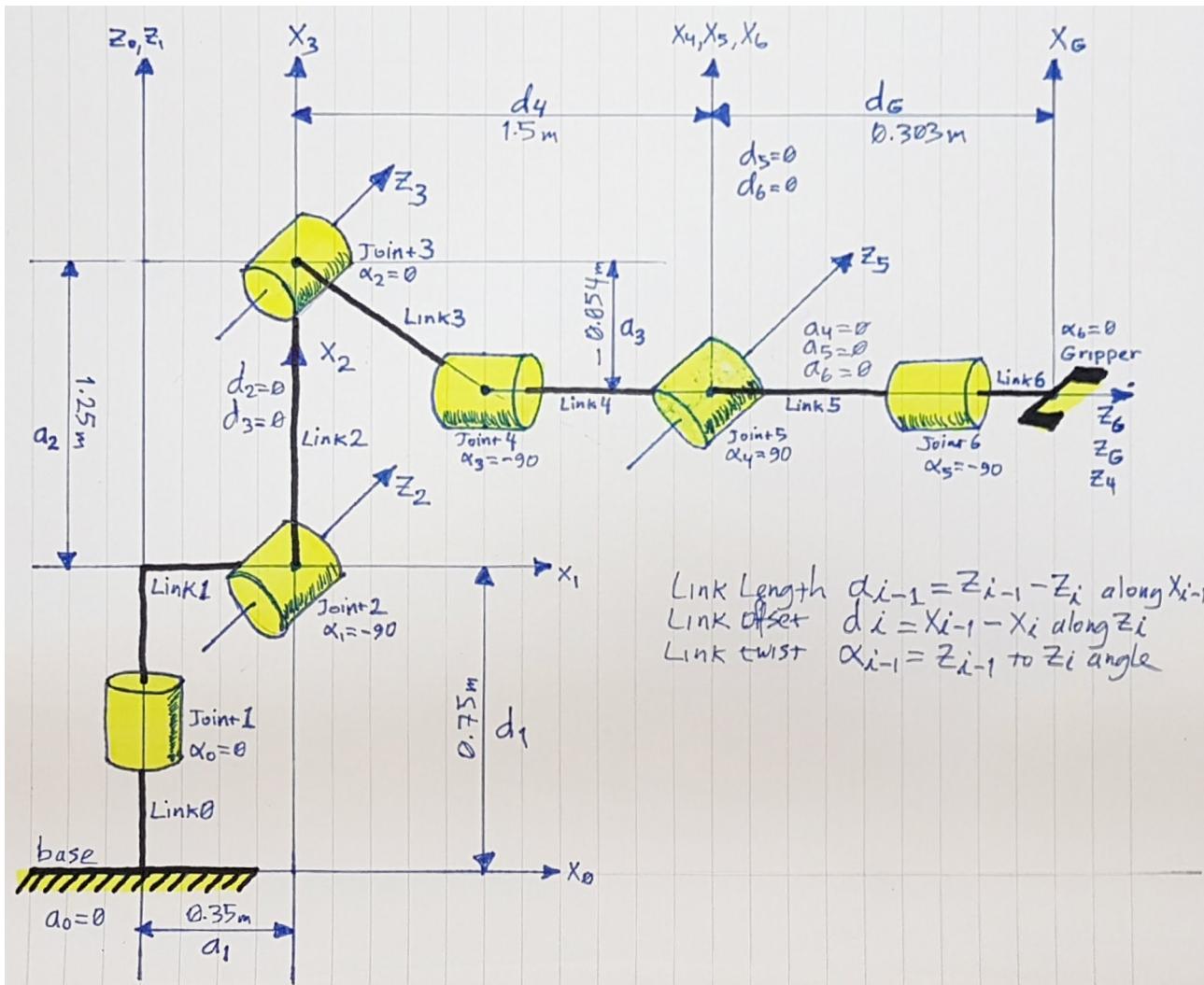
Following table is showing complete extracted list for all joints base to gripper:

O	joint	parent	child	x	y	z	r	p	y
0	fixed_base	base_footprint	base_link	0	0	0	0	0	0
1	joint_1	base_link	link_1	0	0	0.33	0	0	0
2	joint_2	link_1	link_2	0.35	0	0.42	0	0	0
3	joint_3	link_2	link_3	0	0	1.25	0	0	0
4	joint_4	link_3	link_4	0.96	0	-0.054	0	0	0
5	joint_5	link_4	link_5	0.54	0	0	0	0	0
6	joint_6	link_5	link_6	0.193	0	0	0	0	0
7	gripper	link_6	gripper_link	0.11	0	0	0	0	0
.	<b>Total (m)</b>			<b>2.153</b>	<b>0</b>	<b>1.946</b>	<b>0</b>	<b>0</b>	<b>0</b>

Now using the above table we can draw the different frames with x and z translations from one frame to another as shown in below figure.



Then we can further simplify by combining the last three joints (4,5, and 6) in joint\_5 since their axes in actual KR210 robot intersect at a single point which represent the center of the robot spherical wrist:



Note that:

**Origin O(i)** = intersection between  $X_i$  and  $Z_i$  axis

**Link Length:**  $a(i-1) = Z_{i-1} - Z_i$  along the  $X(i-1)$  axis

**Link Offset:**  $d(i) = X_{i-1} - X_i$  along  $Z(i)$  axis

**Link Twist:**  $\alpha(i-1) = \text{angle from } Z_{i-1} \text{ to } Z(i)$  measured about  $X_{i-1}$  using right hand rule

**Joint Angle:**  $\theta(i) = \text{angle from } X_{i-1} \text{ to } X_i$  measured about  $Z_i$  using right hand rule. all joint angles will be zero at initial Robot state in KR210 except joint 2 which has a -90 degree constant offset between  $X(1)$  and  $X(2)$ .

**Gripper frame:** is the end point that we care about. it is displaced from Frame 6 by a translation along  $Z(6)$ .

## KUKA KR210 robot DH parameters.

Using the above mentioned formulas, we can generate the DH parameters table as following:

Links	i	alpha(i-1)	a(i-1)	d(i)	theta(i)
0->1	1	0	0	0.75	q1
1->2	2	-90	0.35	0	-90+q2
2->3	3	0		1.25	q3
3->4	4	-90	-0.05	1.5	q4
4->5	5	90	0	0	q5
5->6	6	-90	0	0	q6
6->7	7	0	0	0.303	q7

in which q(i) is our input to joint angles (theta(i)).

I will be using python to code the forward kinematics:

To start with, we need the following imports:

```
import numpy as np
from numpy import array
from sympy import symbols, cos, sin, pi, simplify, sqrt, atan2, pprint
from sympy.matrices import Matrix
```

Python code to represent DH parameters table is:

```
# DH Table
s = {alpha0: 0, a0: 0, d1: 0.75, q1: q1,
      alpha1: -pi/2., a1: 0.35, d2: 0, q2: -pi/2.+q2,
      alpha2: 0, a2: 1.25, d3: 0, q3: q3,
      alpha3: -pi/2., a3: -0.054, d4: 1.50, q4: q4,
      alpha4: pi/2., a4: 0, d5: 0, q5: q5,
      alpha5: -pi/2., a5: 0, d6: 0, q6: q6,
      alpha6: 0, a6: 0, d7: 0.303, q7: 0}
```

## Creating the individual transformation matrices about each joint:

Using above DH parameter table, we can create individual transforms between various links. DH convention uses four individual transforms:

$${}^{i-1}_iT = R(x_{i-1}, \alpha_{i-1}) T(x_{i-1}, a_{i-1}) R(z_i, \theta_i) T(z_i, d_i)$$

Using the DH parameter table, we can transform from one frame to another using the following matrix:

$${}^{i-1}_iT = \begin{bmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ s\theta_i c\alpha_{i-1} & c\theta_i c\alpha_{i-1} & -s\alpha_{i-1} & -s\alpha_{i-1}d_i \\ s\theta_i s\alpha_{i-1} & c\theta_i s\alpha_{i-1} & c\alpha_{i-1} & c\alpha_{i-1}d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Python code for a function that will return the individual frame transformation matrix is as following:

```
# Function to return homogeneous transform matrix

def TF_Mat(alpha, a, d, q):
    TF = Matrix([[cos(q), -sin(q), 0,
a], [sin(q)*cos(alpha), cos(q)*cos(alpha), -sin(alpha), -sin(alpha)*d],
[sin(q)*sin(alpha), cos(q)*sin(alpha), cos(alpha), cos(alpha)*d],
[0, 0, 0, 1]])
    return TF
```

Then using the following code to substitute the DH parameters into the transformation matrix:

```
## Substitute DH_Table
T0_1 = TF_Mat(alpha0, a0, d1, q1).subs(dh)
T1_2 = TF_Mat(alpha1, a1, d2, q2).subs(dh)
T2_3 = TF_Mat(alpha2, a2, d3, q3).subs(dh)
T3_4 = TF_Mat(alpha3, a3, d4, q4).subs(dh)
T4_5 = TF_Mat(alpha4, a4, d5, q5).subs(dh)
T5_6 = TF_Mat(alpha5, a5, d6, q6).subs(dh)
T6_7 = TF_Mat(alpha6, a6, d7, q7).subs(dh)
```

To get the composition of all transforms from base to gripper we simply multiply the individual matrices using the following code:

```
# Composition of Homogeneous Transforms
# Transform from Base link to end effector (Gripper)
T0_2 = (T0_1 * T1_2) ## (Base) Link_0 to Link_2
T0_3 = (T0_2 * T2_3) ## (Base) Link_0 to Link_3
T0_4 = (T0_3 * T3_4) ## (Base) Link_0 to Link_4
T0_5 = (T0_4 * T4_5) ## (Base) Link_0 to Link_5
T0_6 = (T0_5 * T5_6) ## (Base) Link_0 to Link_6
T0_7 = (T0_6 * T6_7) ## (Base) Link_0 to Link_7 (End Effector)
```

In order to apply correction needed to account for Orientation Difference Between definition of Gripper Link\_7 in URDF versus DH Convention we need to rotate around y then around z axis:

```
R_y = Matrix([[cos(-np.pi/2), 0, sin(-np.pi/2), 0],
[0, 1, 0, 0],
[-sin(-np.pi/2), 0, cos(-np.pi/2), 0],
[0, 0, 0, 1]]))

R_z = Matrix([[cos(np.pi), -sin(np.pi), 0, 0],
[sin(np.pi), cos(np.pi), 0, 0],
[0, 0, 1, 0],
[0, 0, 0, 1]]))

R_corr = (R_z * R_y)

T_total= (T0_7 * R_corr)
```

To check results we can evaluate the individual results when all thetas are equal to zero and compare it to rviz simulator values. I have used pretty print (pprint) to show the resulting matrix as shown in below code.

```
### Numerically evaluate transforms (compare this to output of tf_echo/rviz)

print("\nT0_7 = \n")
pprint(T0_7.evalf(subs={q1: 0, q2: 0, q3: 0, q4: 0, q5: 0, q6: 0}))
```

Remember that the homogeneous transform consists of a rotation part and a translation part as follows:

$$T = \begin{bmatrix} R_T & \begin{matrix} P_x \\ P_y \\ P_z \end{matrix} \\ \begin{matrix} 0 & 0 & 0 \end{matrix} & 1 \end{bmatrix}$$

where Px, Py, Pz represent the position of end-effector w.r.t. base\_link and RT represent the rotation part using the Roll-Pitch-Yaw angles

Individual transform matrices about each joint using the DH table are as following:

$$T_{0\_1} = \begin{bmatrix} \cos(q_1) & -\sin(q_1) & 0 & 0 \\ \sin(q_1) & \cos(q_1) & 0 & 0 \\ 0 & 0 & 1 & 0.75 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{1\_2} = \begin{bmatrix} \cos(q_2 - 0.5 \cdot \pi) & -\sin(q_2 - 0.5 \cdot \pi) & 0 & 0.35 \\ 0 & 0 & 1 & 0 \\ -\sin(q_2 - 0.5 \cdot \pi) & -\cos(q_2 - 0.5 \cdot \pi) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{2\_3} = \begin{bmatrix} \cos(q_3) & -\sin(q_3) & 0 & 1.25 \\ \sin(q_3) & \cos(q_3) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{3\_4} = \begin{bmatrix} \cos(q_4) & -\sin(q_4) & 0 & -0.054 \\ 0 & 0 & 1 & 1.5 \\ -\sin(q_4) & -\cos(q_4) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{4\_5} = \begin{bmatrix} \cos(q_5) & -\sin(q_5) & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \sin(q_5) & \cos(q_5) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{5\_6} = \begin{bmatrix} \cos(q_6) & -\sin(q_6) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\sin(q_6) & -\cos(q_6) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{6\_7} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.303 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In order to compare the output of forward kinematics code with simulator I used the following ROS launch command to run simulator:

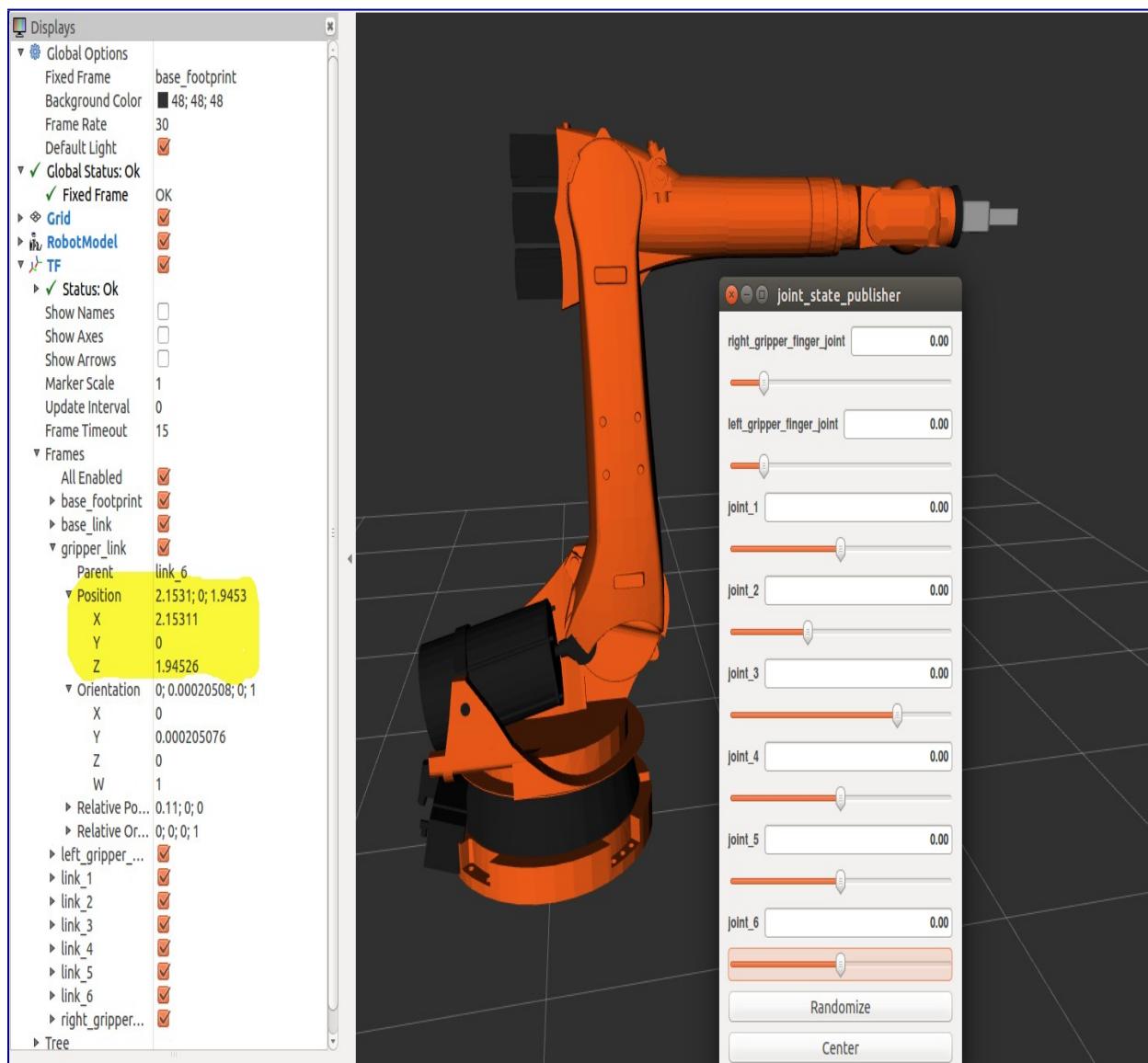
```
$ roslaunch kuka_arm forward_kinematics.launch
```

Then as shown below; I used RViz tf frames to check values of Px, Py, and Pz in compare to output of python code.

### Test Case 1: When all thetas = 0 which is the robot initial state.

```
T0_7.evalf(subs={q1: 0, q2: 0, q3: 0, q4: 0, q5: 0, q6: 0})
```

#### Rviz output:



**Python output:**

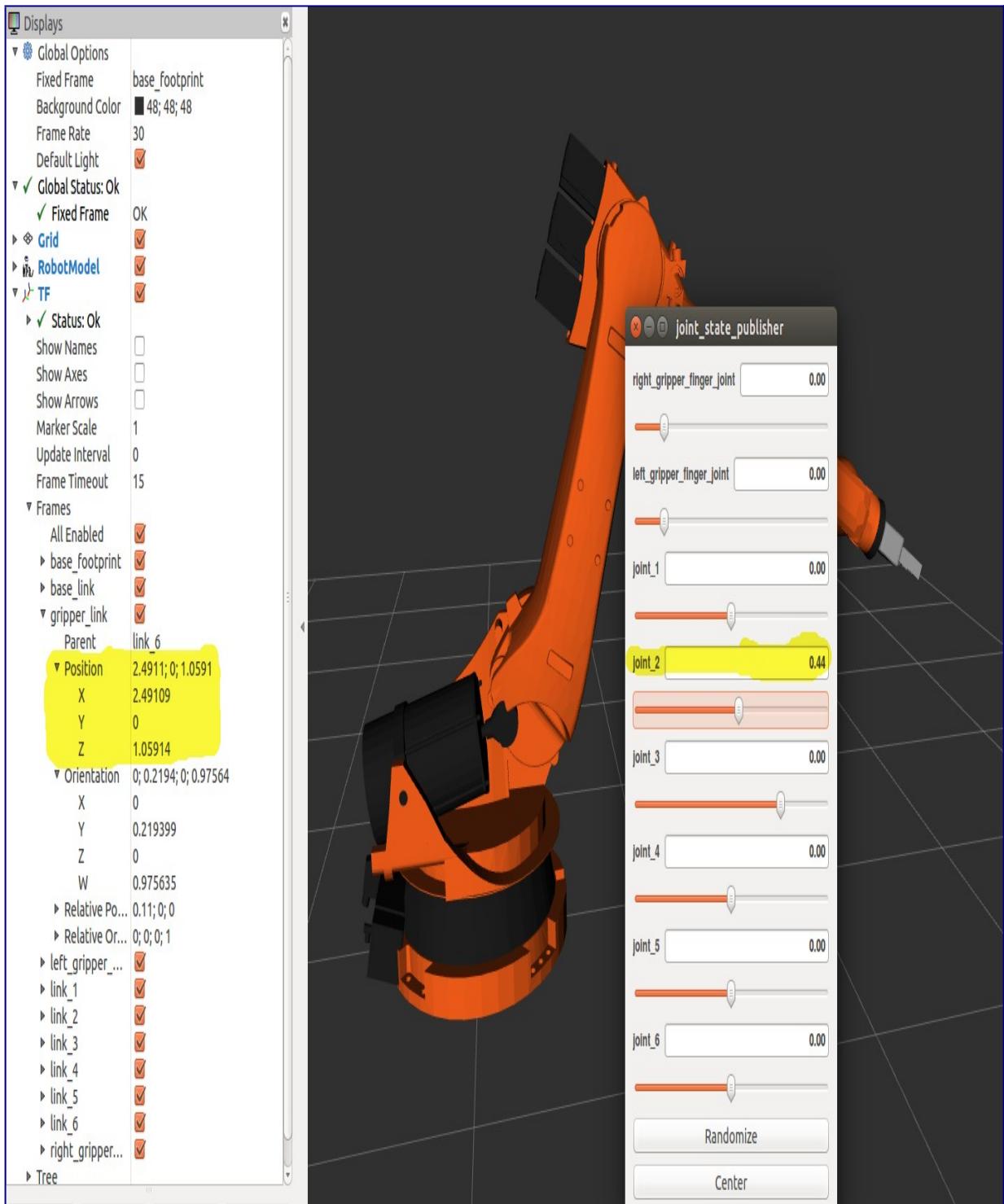
T0\_7 =

**Test Case 2:  
When  
theta2=0.44 and  
remaining  
thetas=0**

$$\begin{bmatrix} -0.e-1069 & 0 & 1.0 & 2.153 \\ 0 & -1.0 & 0 & 0 \\ 1.0 & 0 & 0.e-270 & 1.946 \\ 0 & 0 & 0 & 1.0 \end{bmatrix}$$

```
T0_7.evalf(subs={q1: 0, q2: 0.44, q3: 0, q4: 0, q5: 0, q6: 0})
```

## Rviz output:



**Python output:**

```
T0_7 =  
[ 0.425939465066 0 0.904751663219963 2.49069084900453 ]  
[ 0 0 -1.0 0 0 ]  
[ 0.904751663219963 0 -0.425939465066 1.06411413369708 ]  
[ 0 0 0 1.0 ]
```

## Inverse Kinematics Analysis

Since the last three joints in KUKA KR210 robot (Joint\_4, Joint\_5, and Joint\_6) are revolute and their joint axes intersect at a single point (Joint\_5), we have a case of spherical wrist with joint\_5 being the common intersection point; the wrist center (**WC**). This allows us to kinematically decouple the IK problem into **Inverse Position** and **Inverse Orientation** problems.

### Inverse Position

First step is to get the end-effector position(**Px**, **Py**, **Pz**) and orientation (**Roll**, **Pitch**, **Yaw**) from the test cases data class as shown in below code:

```
# Requested end-effector (EE) position  
px = req.poses[x].position.x  
py = req.poses[x].position.y  
pz = req.poses[x].position.z  
  
# store EE position in a matrix  
EE = Matrix([[px],  
            [py],  
            [pz]])  
  
# Requested end-effector (EE) orientation  
(roll,pitch,yaw) = tf.transformations.euler_from_quaternion(  
    [req.poses[x].orientation.x,  
     req.poses[x].orientation.y,  
     req.poses[x].orientation.z,  
     req.poses[x].orientation.w])
```

We will need rotation matrix for the end-effector:

$$\mathbf{R}_{rpy} = \text{Rot}(Z, \text{yaw}) * \text{Rot}(Y, \text{pitch}) * \text{Rot}(X, \text{roll})$$

and orientation difference correction matrix (**Rot\_corr**) as earlier discussed in FK section.

$$\mathbf{R}_{EE} = \mathbf{R}_{rpy} * \mathbf{R}_{corr}$$

We substitute the obtained roll, pitch and yaw in the final rotation matrix. Python Code is as following:

```
# Find EE rotation matrix RPY (Roll, Pitch, Yaw)
r,p,y = symbols('r p y')

# Roll
ROT_x = Matrix([[ 1, 0, 0],
                 [ 0, cos(r), -sin(r)],
                 [ 0, sin(r), cos(r)]])

# Pitch
ROT_y = Matrix([[ cos(p), 0, sin(p)],
                 [ 0, 1, 0],
                 [ -sin(p), 0, cos(p)]])

# Yaw
ROT_z = Matrix([[ cos(y), -sin(y), 0],
                 [ sin(y), cos(y), 0],
                 [ 0, 0, 1]])

ROT_EE = ROT_z * ROT_y * ROT_x

# Correction Needed to Account for Orientation Difference Between
# Definition of Gripper Link_G in URDF versus DH Convention

ROT_corr = ROT_z.subs(y, radians(180)) * ROT_y.subs(p, radians(-90))

ROT_EE = ROT_EE.subs({'r': roll, 'p': pitch, 'y': yaw})
```

The obtained matrix will be the rotation part of the full homogeneous transform matrix as yellow highlighted in the following:

$$\begin{bmatrix} l_x & m_x & n_x & p_x \\ l_y & m_y & n_y & p_y \\ l_z & m_z & n_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$${}_{EE}^0 T = \left[ \begin{array}{ccc|c} 0 & {}_6 R & {}^0 \mathbf{r}_{EE/0} & 1 \\ 0 & 0 & 0 & 1 \end{array} \right] = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where  $\mathbf{l}$ ,  $\mathbf{m}$  and  $\mathbf{n}$  are orthonormal vectors representing the end-effector orientation along X, Y, Z axes of the local coordinate frame.

Since  $\mathbf{n}$  is the vector along the **z-axis** of the **gripper\_link**, we can say the following:

$$\begin{aligned} X_{wc} &= P_x - (d_6 + d_7) \cdot n_x \\ Y_{wc} &= P_y - (d_6 + d_7) \cdot n_y \\ Z_{wc} &= P_z - (d_6 + d_7) \cdot n_z \end{aligned}$$

Where,

**Px, Py, Pz** = end-effector positions obtained from test case data

**Xwc, Ywc, Zwc** = wrist center positions that we are trying to find.

**d6** = link\_6 length obtained from DH table ( $d6=0$ )

**d7** = end-effector length obtained from DH table ( $d7=0.303$ )

The same equation in vectorized version (d is the displacement):

$${}^0\mathbf{r}_{WC/0} = {}^0\mathbf{r}_{EE/0} - d \cdot {}^0R \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} - d \cdot {}^0R \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

In Python code:

```
# Calculate Wrest Center
WC = EE - (0.303) * ROT_EE[:,2]
```

WC is now having position of wrist center (Wx, Wy, Wz).

To find  $\theta 1$ , we need to project Wz onto the ground plane Thus,

**Theta1=atan2(Wy,Wx)**

```
# Calculate theat1
theta1 = atan2(WC[1], WC[0])
```

Using trigonometry, we can calculate  $\theta 2$  and  $\theta 3$ .

We have a triangle (the green color in below figure) with two sides known to us ( $A = d4 = 1.5$ ) and ( $C = a2 = 1.25$ ), the 3rd side (**B**) can be calculated as following:

$$\begin{aligned} R &= \sqrt{X_{wc}^2 + Y_{wc}^2} - a_1 \\ S &= Z_{wc} - d_1 \\ B &= \sqrt{R^2 + S^2} \end{aligned}$$

Below is the same in Python code:

```
#SSS triangle for theta2 and theta3
```

```

A = 1.501
C = 1.25
B = sqrt(pow((sqrt(WC[0]*WC[0] + WC[1]*WC[1]) - 0.35), 2) + pow((WC[2] - 0.75), 2))

```

Now since we have all three sides of the triangle known to us we can calculate all of the three inner angles of the triangle from the known three sides Using trigonometry (specifically the **Cosine Laws SSS type**).

$$\begin{aligned}
\cos(a) &= (B^2 + C^2 - A^2)/(2*B*C) \\
\cos(b) &= (A^2 + C^2 - B^2)/(2*A*C) \\
\cos(c) &= (A^2 + B^2 - C^2)/(2*A*B) \\
\\
a &= \arccos((B^2 + C^2 - A^2)/(2*B*C)) \\
b &= \arccos((A^2 + C^2 - B^2)/(2*A*C)) \\
c &= \arccos((A^2 + B^2 - C^2)/(2*A*B))
\end{aligned}$$

The same in Python code:

```

a = arccos((B*B + C*C - A*A) / (2*B*C))
b = arccos((A*A + C*C - B*B) / (2*A*C))
c = arccos((A*A + B*B - C*C) / (2*A*B))

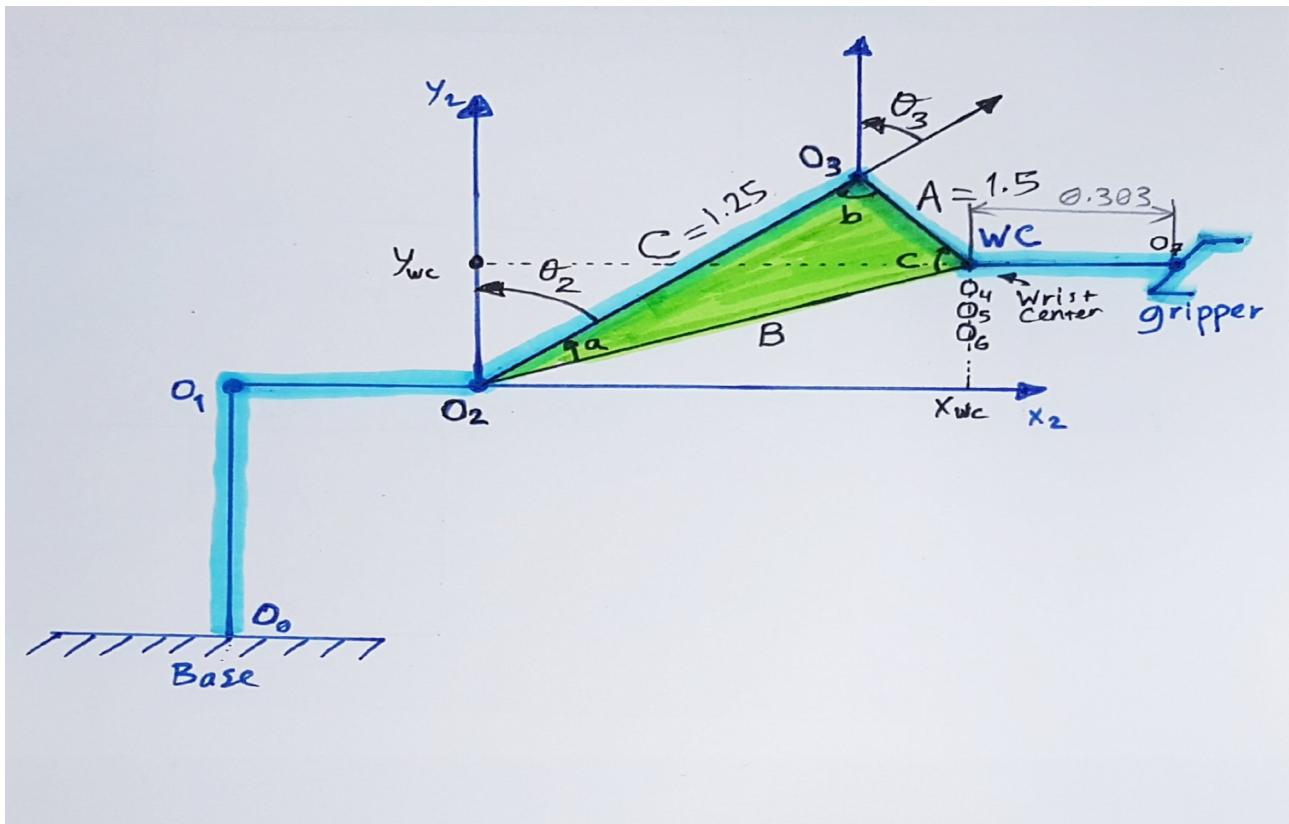
```

Finally we calculate  $\theta_2$  and  $\theta_3$

```

theta2 = pi/2 - a - atan2(WC[2]-0.75, sqrt(WC[0]*WC[0]+WC[1]*WC[1])-0.35)
theta3 = pi/2 - (b+0.036) # 0.036 accounts for sag in link4 of -0.054m

```



## Inverse Orientation

For the **Inverse Orientation** problem, we need to find values of the final three joint variables  **$\theta 4$ ,  $\theta 5$  and  $\theta 6$** .

Using the individual DH transforms we can obtain the resultant transform and hence resultant rotation by:

$$\mathbf{R}_{0\_6} = \mathbf{R}_{0\_1}\mathbf{R}_{1\_2}\mathbf{R}_{2\_3}\mathbf{R}_{3\_4}\mathbf{R}_{4\_5} * \mathbf{R}_{5\_6}$$

Since the overall RPY (Roll Pitch Yaw) rotation between base\_link and gripper\_link must be equal to the product of individual rotations between respective links, following holds true:

$$\mathbf{R}_{0\_6} = \mathbf{R}_{EE}$$

where,

**R\_EE** = Homogeneous RPY rotation between base\_link and gripper\_link as calculated above.

We can substitute the values we calculated for  **$\theta 1$ ,  $\theta 2$  and  $\theta 3$** . in their respective individual rotation matrices and pre-multiply both sides of the above equation by **inv(R0\_3)** which leads to:

$$\mathbf{R}_{3\_6} = \text{inv}(\mathbf{R}_{0\_3}) * \mathbf{R}_{EE}$$

$${}^3_6 R = \begin{pmatrix} {}^0_3 R \\ 0 \end{pmatrix}^{-1} {}^0_6 R = \begin{pmatrix} {}^0_3 R \\ 0 \end{pmatrix}^T {}^0_6 R$$

The resultant matrix on the RHS (Right Hand Side of the equation) does not have any variables after substituting the joint angle values, and hence comparing LHS (Left Hand Side of the equation) with RHS will result in equations for  $\theta 4$ ,  $\theta 5$  and  $\theta 6$ .

```
# Extract rotation matrix R0_3 from transformation matrix T0_3 the
substitute angles q1-3
R0_3 = T0_1[0:3,0:3] * T1_2[0:3,0:3] * T2_3[0:3,0:3]
R0_3 = R0_3.evalf(subs={q1: theta1, q2: theta2, q3:theta3})

# Get rotation matrix R3_6 from (inverse of R0_3 * R_EE)
R3_6 = R0_3.inv(method="LU") * ROT_EE
```

I have added if/else to select the best solution for  $\theta 4$ ,  $\theta 5$  and  $\theta 6$ .

```
# Euler angles from rotation matrix
    theta5 = atan2(sqrt(R3_6[0,2]*R3_6[0,2] +
R3_6[2,2]*R3_6[2,2]),R3_6[1,2])

    # select best solution based on theta5
    if (theta5 > pi) :
        theta4 = atan2(-R3_6[2,2], R3_6[0,2])
        theta6 = atan2(R3_6[1,1], -R3_6[1,0])
    else:
        theta4 = atan2(R3_6[2,2], -R3_6[0,2])
        theta6 = atan2(-R3_6[1,1], R3_6[1,0])
```

Also I have added to the forward kinematics code to help in checking for errors.

```
FK =
T0_7.evalf(subs={q1:theta1,q2:theta2,q3:theta3,q4:theta4,q5:theta5,q6:theta6})
```

The rest of the code will utilize wrist center position **WC** and the **thetas** to calculate the corresponding errors. Using these error values as a basis, We can gauge how well our current IK performs.

I have added one line of code to print out the test case number. rest of the code is as provided.

```
# Print test case number
print ("Using Test Case Number %d" %test_case_number)
```

The output of all 3 provided test cases are as following:

### Test Case 1 output:

```
Total run time to calculate joint angles from pose is 0.6331 seconds
Using Test Case Number 1
```

```
Wrist error for x position is: 0.00000046
Wrist error for y position is: 0.00000032
Wrist error for z position is: 0.00000545
```

Overall wrist offset is: 0.00000548 units

Theta 1 error is: 0.00093770  
Theta 2 error is: 0.00181024  
Theta 3 error is: 0.00205031  
Theta 4 error is: 0.00172067  
Theta 5 error is: 0.00197873  
Theta 6 error is: 0.00251871

\*\*These theta errors may not be a correct representation of your code, due to the fact that the arm can have multiple positions. It is best to add your forward kinematics to confirm whether your code is working or not\*\*

End effector error for x position is: 0.00002010  
End effector error for y position is: 0.00001531  
End effector error for z position is: 0.00002660  
Overall end effector offset is: 0.00003668 units

## **Test Case 2 output:**

Total run time to calculate joint angles from pose is 0.6635 seconds  
Using Test Case Number 2

Wrist error for x position is: 0.00002426  
Wrist error for y position is: 0.00000562  
Wrist error for z position is: 0.00006521  
Overall wrist offset is: 0.00006980 units

Theta 1 error is: 3.14309971  
Theta 2 error is: 0.27930449  
Theta 3 error is: 1.86835102  
Theta 4 error is: 3.08639294  
Theta 5 error is: 0.06340564  
Theta 6 error is: 6.13524247

\*\*These theta errors may not be a correct representation of your code, due to the fact that the arm can have multiple positions. It is best to add your forward kinematics to confirm whether your code is working or not\*\*

End effector error for x position is: 0.00002566  
End effector error for y position is: 0.00002581  
End effector error for z position is: 0.00000461  
Overall end effector offset is: 0.00003668 units

## **Test Case 3 output:**

Total run time to calculate joint angles from pose is 0.6569 seconds  
Using Test Case Number 3

Wrist error for x position is: 0.00000503  
Wrist error for y position is: 0.00000512  
Wrist error for z position is: 0.00000585  
Overall wrist offset is: 0.00000926 units

Theta 1 error is: 0.00136747

```
Theta 2 error is: 0.00325738
Theta 3 error is: 0.00339563
Theta 4 error is: 6.28212730
Theta 5 error is: 0.00284405
Theta 6 error is: 6.28223850
```

```
**These theta errors may not be a correct representation of your code, due to
the fact
that the arm can have multiple positions. It is best to add your forward
kinematics to
confirm whether your code is working or not**
```

```
End effector error for x position is: 0.00000069
End effector error for y position is: 0.00000011
End effector error for z position is: 0.00003668
Overall end effector offset is: 0.00003668 units
```

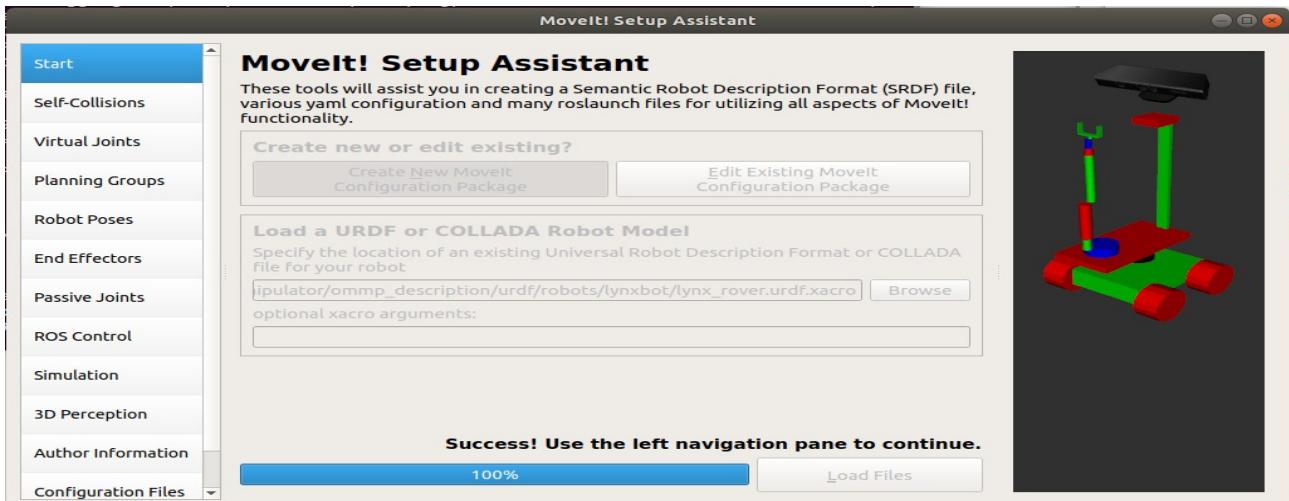
You can see the entire code on the following [link](#)

## 14.2 Controlling a real robotic arm with MoveIT

The first think we will need to is to create a MoveIt configuration package that will tell MoveIt how to move and operate our robot this time with pre-built generic IK solver alongside how to find our 3D sensors for obstacle avoidance when planning a trajectory, fortunately MoveIt provides us with a handy GUI to easily create this package, by running the following command.

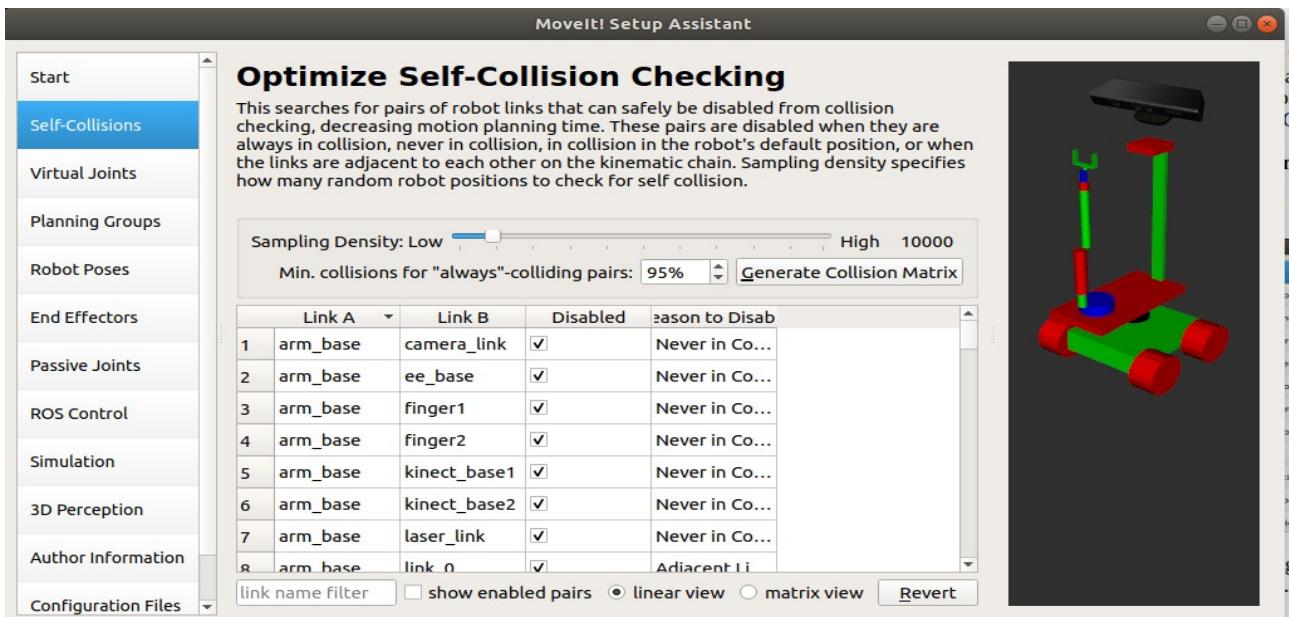
```
$ roslaunch moveit_setup_assistant setup_assistant.launch
```

After we tell it want to create a new moveit package and load our urdf we can see the following



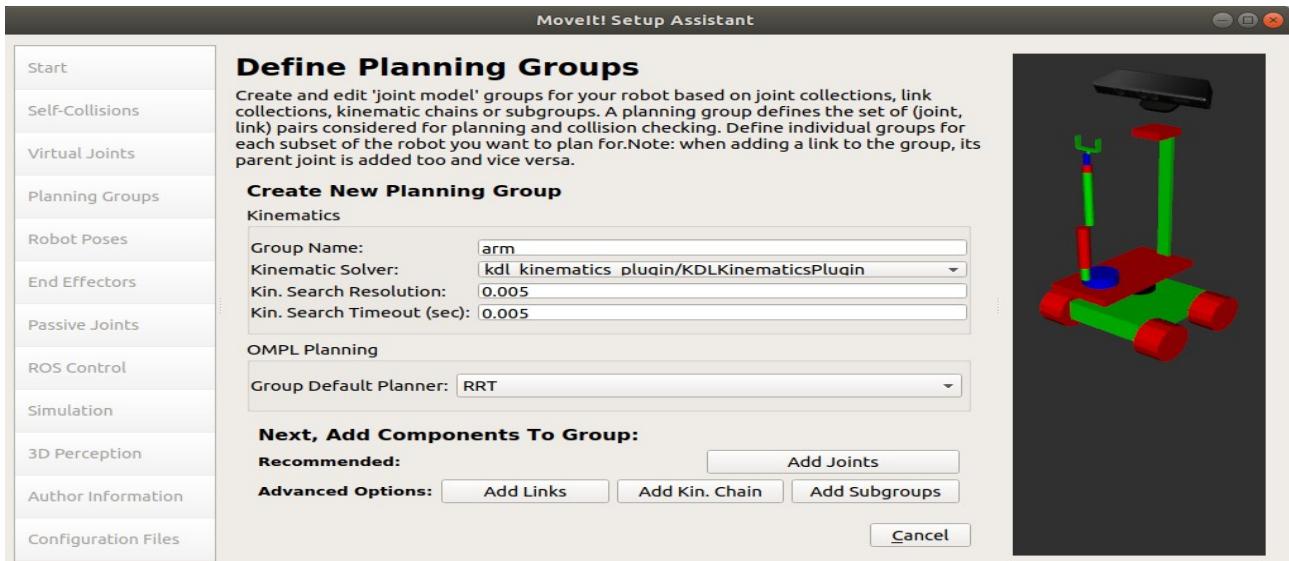
On the right we can see our mobile manipulator in an internal simulator that moveit uses for motion planning.

After that we can generate a Self-Collision matrix so that moveit knows which links are adjacent, or never in collision so it would not consider them on the motion planning.



Next is the Virual Joint that is usefull when we want to tell moveit that our robot is considered fixed from an external frame of reference with respect to the robot, we would not need that in this case.

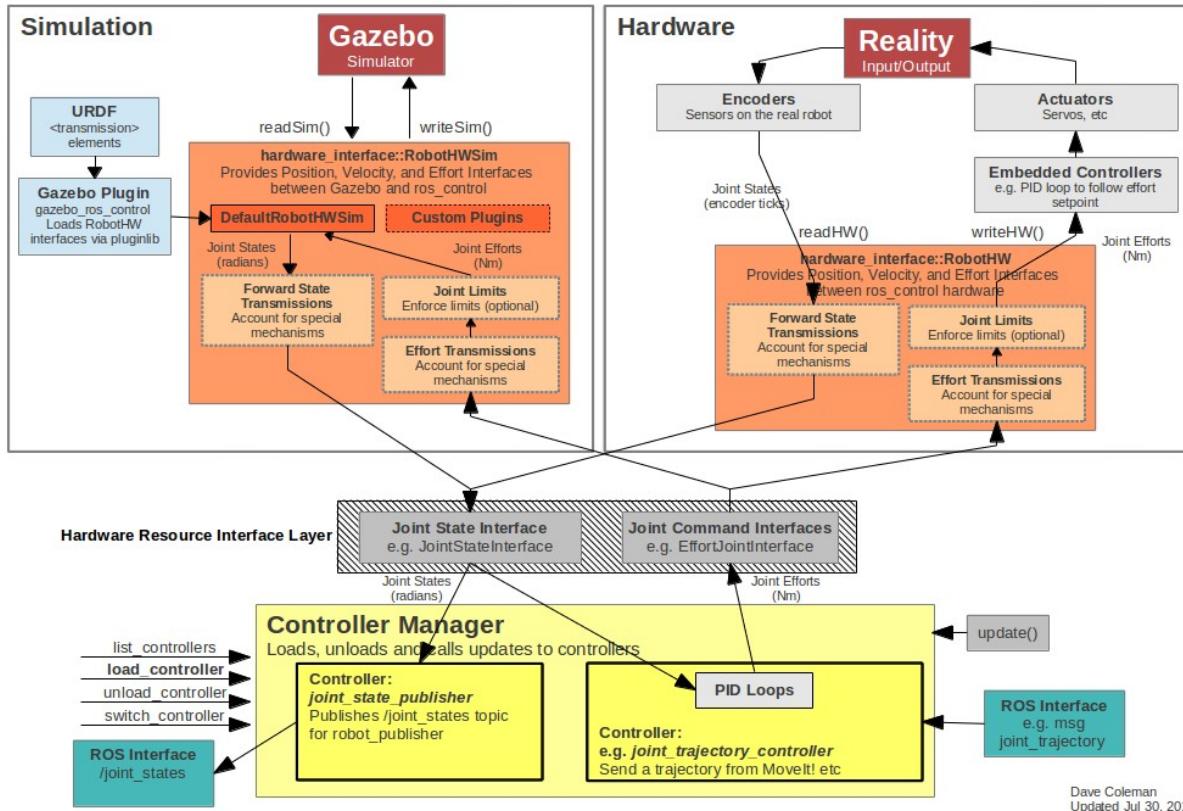
Next we define the planning groups that Moveit uses for IK calculation for the arm and gripper, alongside which joints correspond to each group, here is an example.



In the same manner we can move on with the next sections to configure our moveit package, the next steps are omitted here.

When we are finished with this, we should create the ROS controller in order to move the arm. Typically MoveIt in order to execute the motions on a real or simulated robot needs an action server, usually this action server is implemented through `ros_control` with a `follow_joint_trajectory_controller`.

Below you can see a reminder how `ros_control` works in a high level, note the controller manager and the different hardware interfaces for simulated and real robots.



Below is an example of the config file of an arm controller.

```
arm_controller:
  type: effort_controllers/JointTrajectoryController
  joints:
    - joint_0
    - joint_1
    - joint_2
    - joint_3
    - joint_4
    - joint_5
  gains:
    joint_0: {p: 100, i: 0.01, d: 0.1}
    joint_1: {p: 100, i: 0.01, d: 0.1}
    joint_2: {p: 100, i: 0.01, d: 0.1}
    joint_3: {p: 40, i: 0.004, d: 0.1}
    joint_4: {p: 40, i: 0.004, d: 0.1}
    joint_5: {p: 100, i: 0.02, d: 0.1}
  constraints:
    goal_time: 0.6
    joint_0: {trajectory: 1, goal: 0.1}
    joint_1: {trajectory: 1, goal: 0.1}
    joint_2: {trajectory: 1, goal: 0.1}
```

```

joint_3: {trajectory: 1, goal: 0.1}
joint_4: {trajectory: 1, goal: 0.1}
joint_5: {trajectory: 1, goal: 0.1}

```

Then we can launch the controller manager like so:

```

<!-- launch the controller manager for ros control -->
<node name="controller_spawner" pkg="controller_manager"
type="spawner" respawn="false"
output="screen" args="joint_state_controller
diff_velocity_controller arm_controller gripper_controller
kinect_controller"/>

```

## 14.3 Hardware Interface – ROS Control

### Gazebo

Finally, we need to implement the hardware interface since these controllers so far are robot agnostic. In simulation with Gazebo this is fairly easy since Gazebo provides a plugin that can automate this process for the most part, ofcourse if your implementation requires a specific hardware interface for the simulation you can implement your own.

```

<!-- ros_control plugin -->
<gazebo>
  <plugin name="gazebo_ros_control"
filename="libgazebo_ros_control.so">
    <!--legacyModeNS>true</legacyModeNS-->

<robotSimType>gazebo_ros_control/DefaultRobotHWSim</robotSimType>
</gazebo>

```

### Real Robot

But on the real robot, we do not have this luxury since we do not use Gazebo that can simulate a hardware interface, and we need to implement our own hardware interface in order to tell to ros\_control how to move the real robot and with what communication protocol a.k.a rosserial, ROS Industrial, Can-Bus etc.

If you look at the above sketch of the high level function of ros\_control, you will notice the read() and write() function. Fortunately ros\_control makes it easy to interface our real robot with ros controller, by providing us classes and libraries for the hardware\_interface that we can inherit. Then, our job is to write the **write()** and **read()** functions, which are specific to our robot and our communication protocol to the hardware. This means that the code you put in the write() function must be executed on the real robot and it is your job to make sure this happens, furthermore the read() function is to read the feedback of your real robot.

**To ease my process of development is used this template that i found on this github repo definitely check it out!! It helped me a lot to write the hardware interface of my real arm.**

[https://github.com/PickNikRobotics/ros\\_control\\_boilerplate](https://github.com/PickNikRobotics/ros_control_boilerplate)

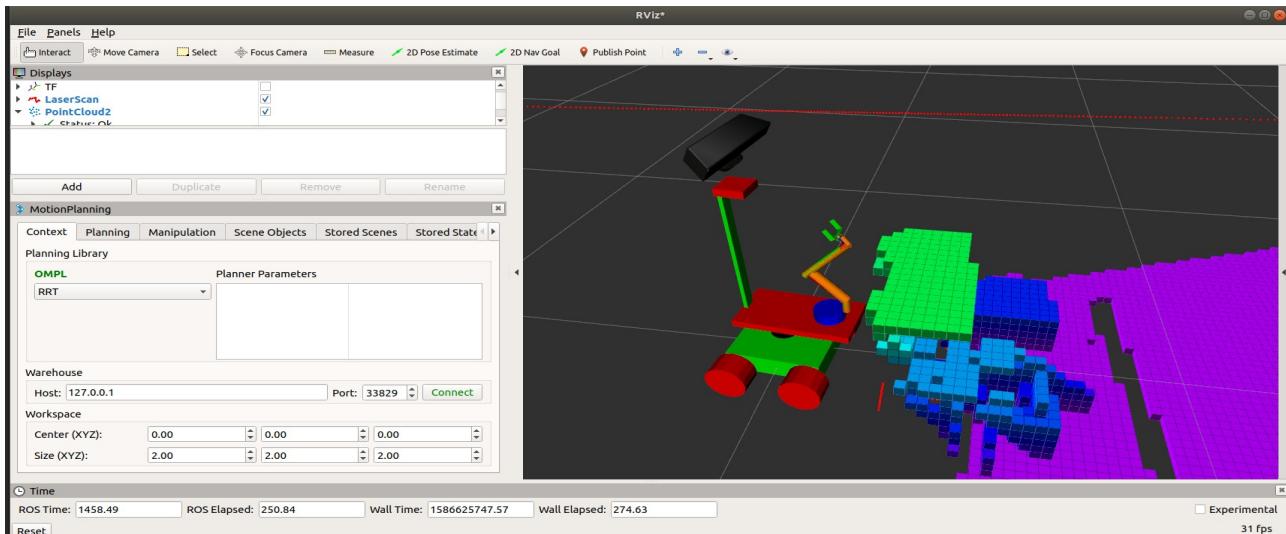
Furthermore, since my hobby-grade robotic arm **doesn't have feedback**, i assumed perfect execution this means that whatever command was coming to me from the joint\_position\_command interface (that theoritacally i should pass to the robot to be executed) instead i directly passed to the read function on the joint\_state\_interface or feedback (**a.k.a perfect execution**).

This has the effect that the robot or better it's inner representation of it RVIZ, **to think that the "real" arm is moving whether the real arm is connected or not and the joint\_states where moving accordingly.**

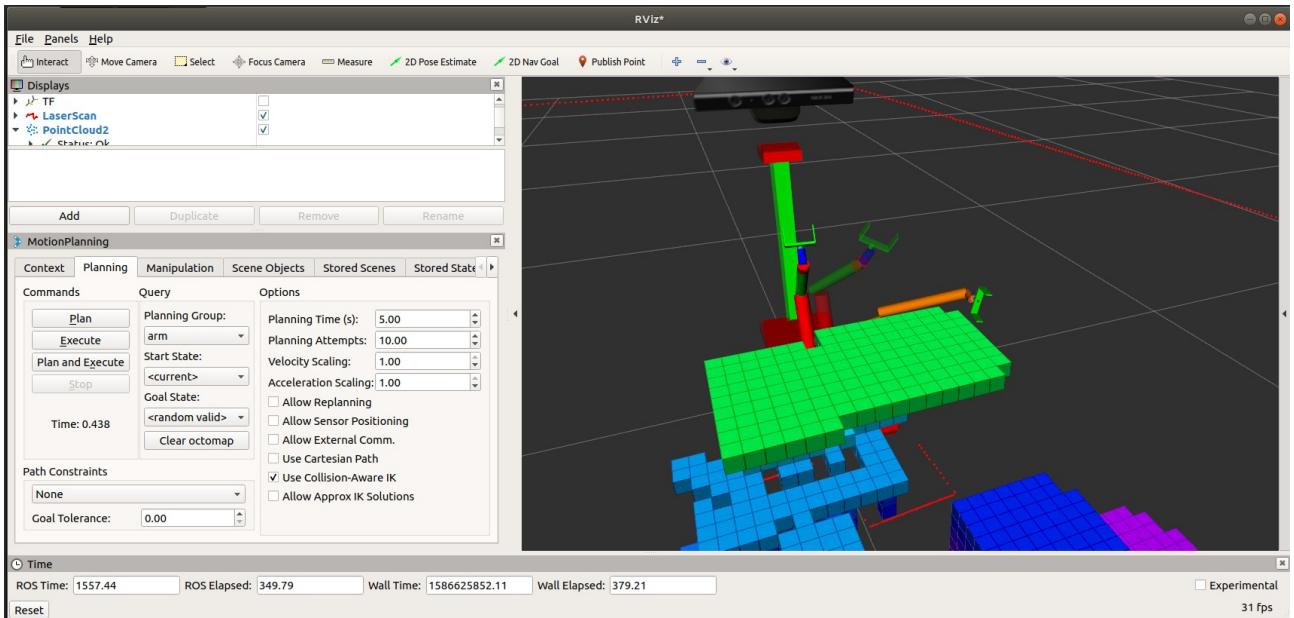
**Then with a node that subscribes to the joint\_States and transforms them to appropiate servo commands in order to be send them with rosserial to the real arm, we achieve the control of the real arm. This means that the arm is a clone of the joint\_states. To avoid sudden movements we first need to activate the servos that are in a position that closely resembles the joint\_states at that time. Do to that we can "pseudo move" the arm to this starting location with moveit before activating the servos of the arm.**

#### 14.4 Controlling the arm in simulation with MotionPlanning RVIZ display.

By adding the MotionPlanning display in rviz we can see control our simulated or real arm by giving it random valid goal position to go to, furthermore we can see an octomap provided to us by the kinect in order for moveit to consider these obstacles and avoid them on it's motion planning.



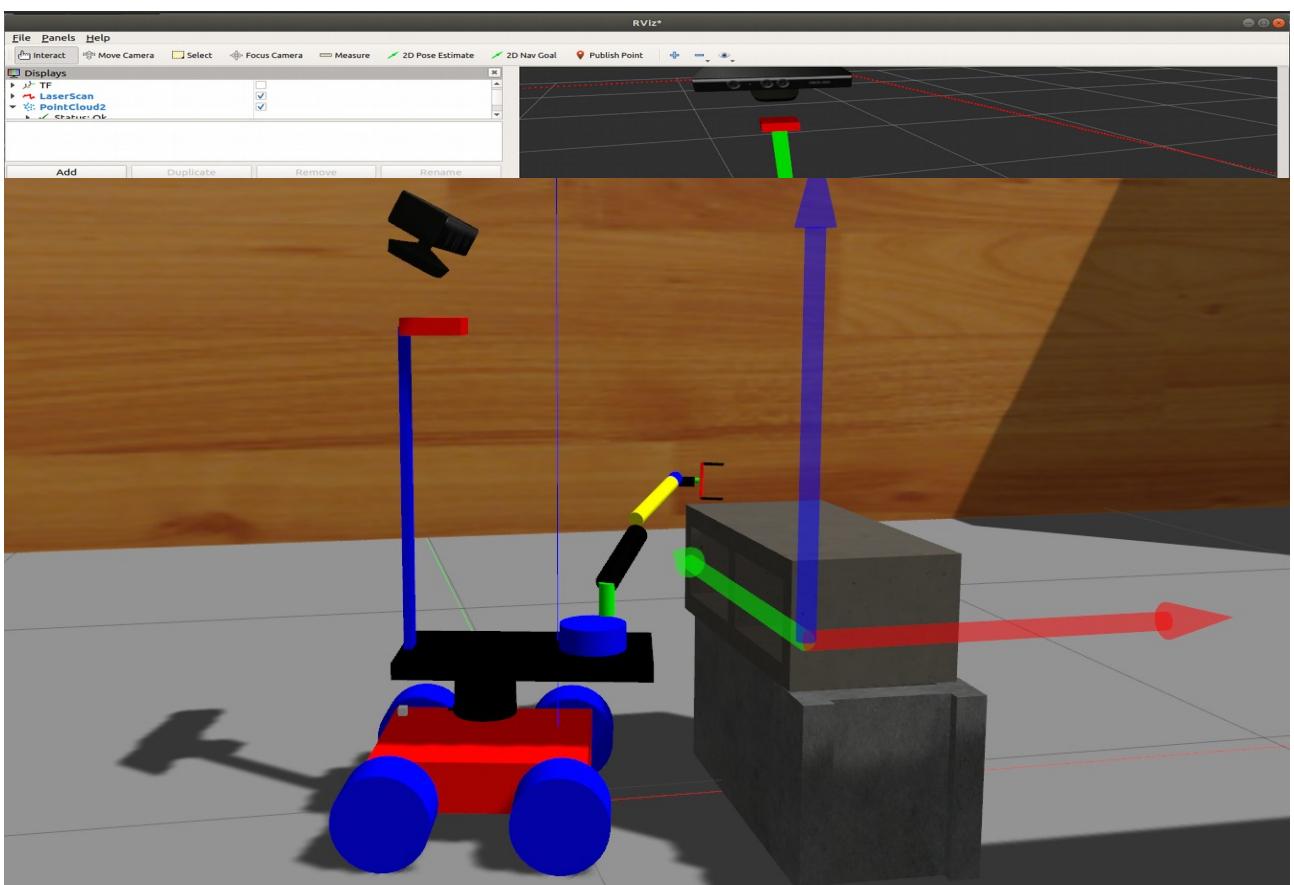
Then we can plan a random valid trajectory



We can see the final location that the arm wants to go in orange.

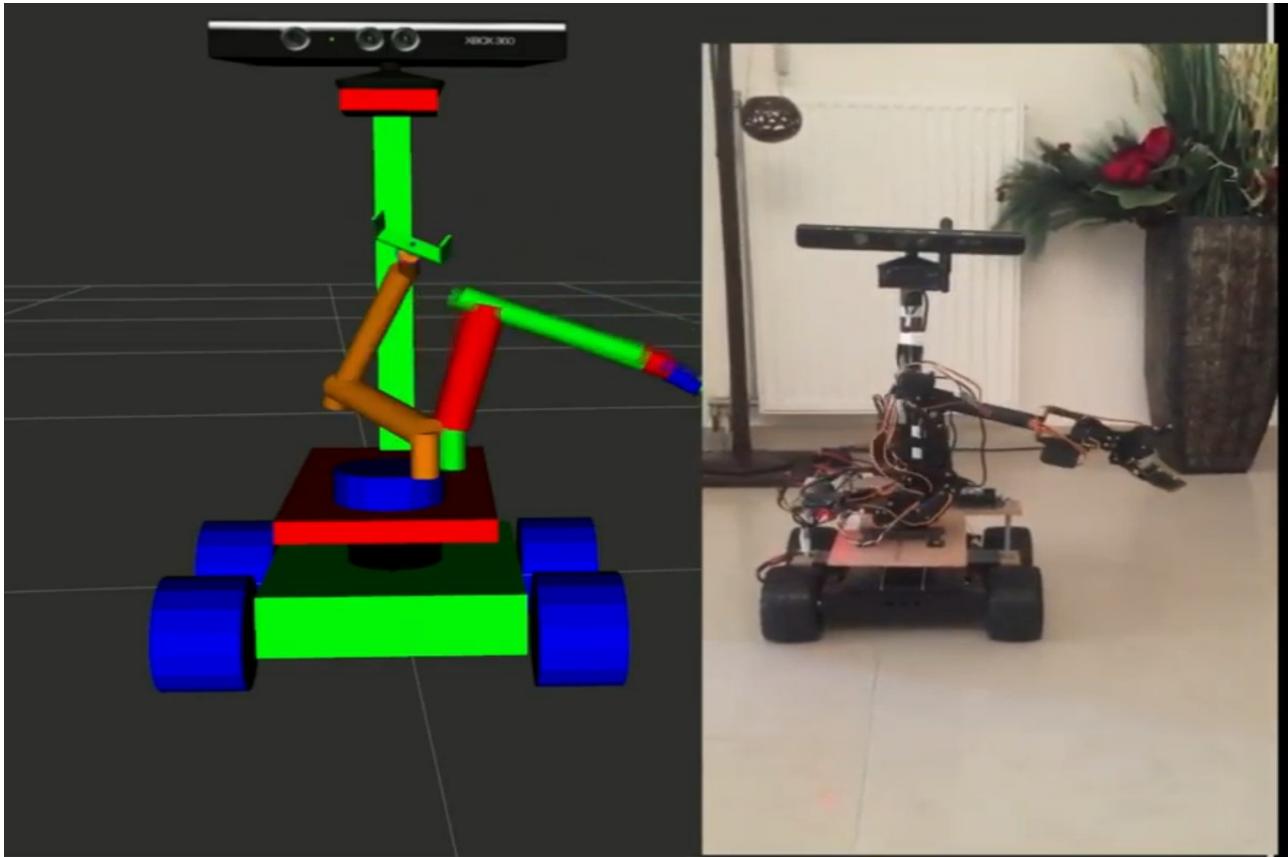
The location that the arm is, and the ghost arm that is semi-transparent that shows the trajectory that the arm will follow.

Then if we execute this trajectory



We can see that the arm has reached the desired location both in RVIZ and on the "real" Gazebo simulation.

**Here is the same and on the real robot!**



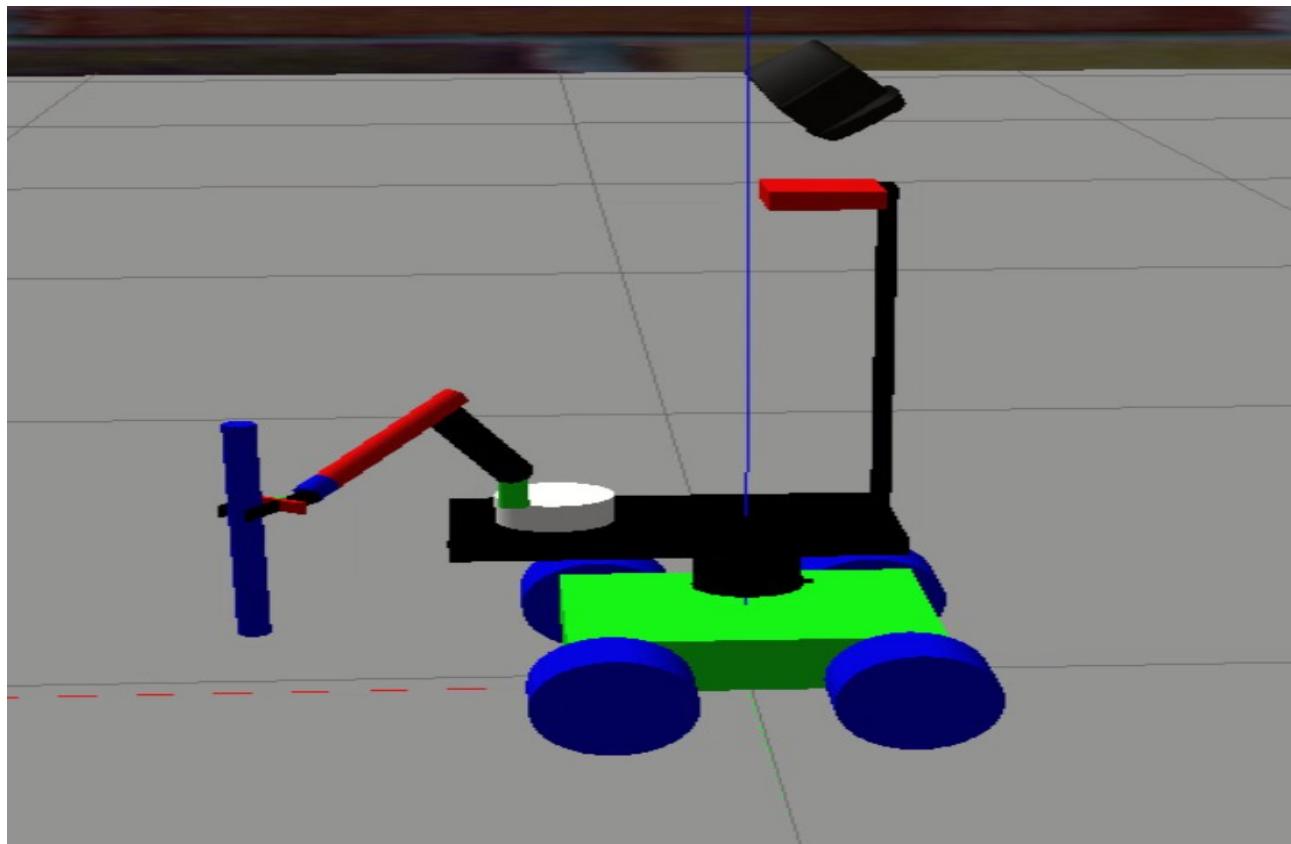
You can see the motion of the real arm on this youtube after 4:50 time mark.

[https://www.youtube.com/watch?v=-vyhhE-3uxY&feature=emb\\_logo](https://www.youtube.com/watch?v=-vyhhE-3uxY&feature=emb_logo)

Ofcourse we can send moveit goals programatically with the moveit commander, instead for random valid goal from the moveit display in rviz. The goals can be desired joint location or end effector location.

In the code which you can find in the following link you can see how to perform a sequence grasping motion, in order to grasp an object that we know it's location from 3D perception and TF.

[https://github.com/panagelak/Open\\_Mobile\\_Manipulator/blob/master/ommp\\_bringup/scripts/traj\\_exec.py](https://github.com/panagelak/Open_Mobile_Manipulator/blob/master/ommp_bringup/scripts/traj_exec.py)



## 15. Project Github Repository

In the project github repository

[https://github.com/panagelak/Open\\_Mobile\\_Manipulator](https://github.com/panagelak/Open_Mobile_Manipulator)

you can find the complete project, with youtube videos, gifs and instructions on how to run the simulation, some demos and lastly on how to build the real robot yourself.

## 16. Extra Projects

Furthermore on my github, you can find some extra project's that I did on the Udacity Robotic's Nanodegree program that aren't referenced here, some examples are.

- [Deep Reinforcement learning - Manipulator Arm - Gazebo API](#)
- [Follow Me Project - Fully Convolutional Neural Network - A drone is following a person In a crowded city](#)
- [Inference Project : Use NVIDIA DIGITS to quickly design and prototype two Neural Networks, one for data supplied to us \(75% <10ms\), and one for data collected by me.](#)
- [Search-and-Sample-Return : A rover in a simulated environment searches for yellow rocks and returns them](#)

## 17. REFERENCES

- [Udacity Robotics Nanodegree](#)
- [The Construct Robot Ignite Academy](#)
- [ROS Answers forum](#)
- [RTAB-map forum](#)
- [ROS Programming: Building Powerful Robots](#)
- [PickNikRobotics/ros\\_control\\_boilerplate](#)