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	Ο Διευθυντής του Τομέα
Ε. ΔΕΡΜΑΤΑΣ	Ν. Φακωτακής
Αναπληρωτής καθηγητής	Καθηγητής

Αριθμός Διπλωματικής Εργασίας:

Θέμα: «Αποφυγή εμποδιών με αυτοκινούμενο ρομποτικό οχήμα»

Φοιτητής: Επιβλέπων:

Αγγελάκης Παναγιώτης 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.

Επίσης 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.

Επίσης 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.

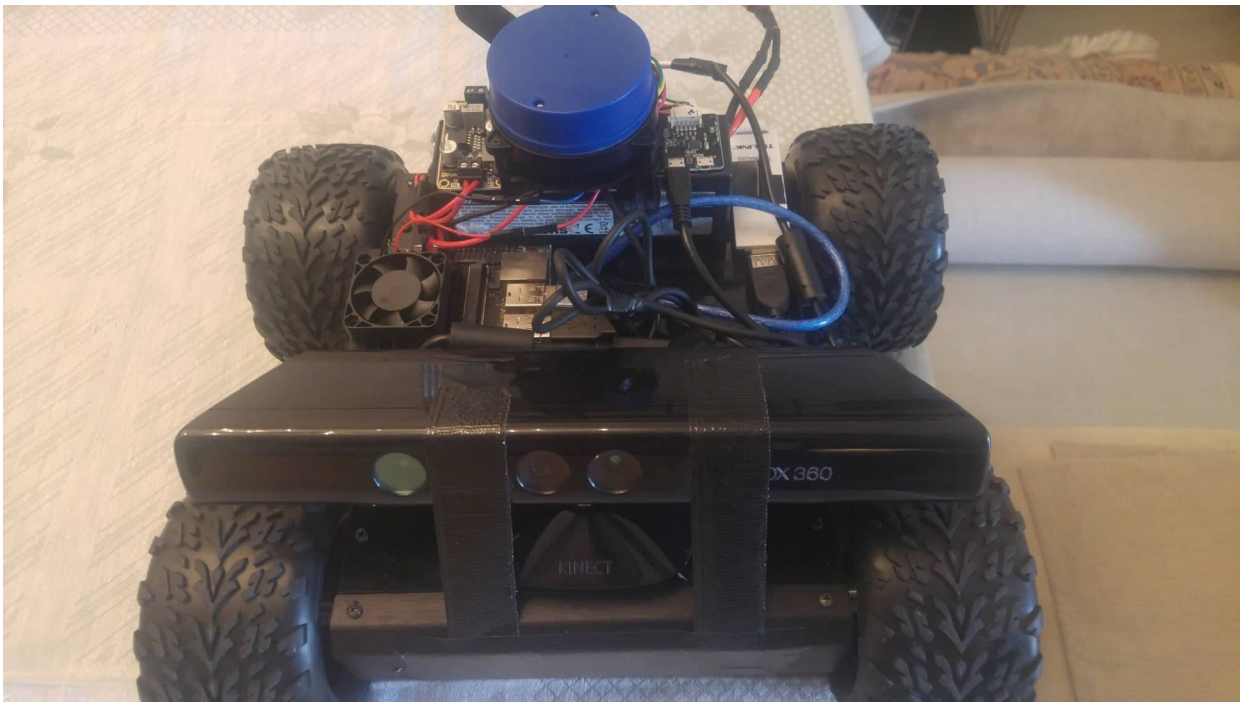
Διαθέτει ένα 2-d lidar για να μετράει την απόσταση παντού γύρω του σε ένα επίπεδο, καθώς και μια 3-d RGB-D κάμερα (kinect) για να φτιάξουμε τούς τρισδιάστατους χάρτες.

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

Ο εγκέφαλος είναι ο ενσωματωμένος υπολογιστής Jetson Nano που με 4 cores στα 1.3Ghz και 125 cores gpu μας παρέχει με την υπολογιστική ισχύ 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 it's 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 so that it can be re-orientated, 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 it;s 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.
Lastly we will attach to the robot, a robotic hand of 6 degrees of freedom so that it can interact with it's 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) capabillities that help us quickly program robots with packages designed by others, as well as a system to communicating 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

Abstract 4
1. Intoduction to Robot Operating System (ROS).....7
1.1 Ubuntu operating system.....7
1.2 What is a robot?.....7
1.3 Before ROS..... 8
1.4 What is ROS..... 8
1.5 Brief history of ROS.....8
1.6 Nodes and compute graphs.....9
1.7 Ros Master and Paramater Server.....10
1.8 Packages and Catkin workspaces.....11
1.9 Running the turtlesim example.....13
1.10 Launch and configuration files.....15
1.11 Topics and writing our first c++ node.....17
1.12 URDF and XACRO.....21
1.13 Robot state publisher and Joint state publisher..... 26
1.14 Gazebo.....28
1.15 Actions.....37
1.16 Rviz.....40
1.17 Moveit.....44
1.18 Debugging Tools.....48
2. Hardware and Sensors Used
2.1 DC motor.....51
2.2 Stepper motor.....54
2.3 Quadrature encoder.....56

2.4 L298N Dual H-Bridge driver.....58
2.5 A4988 driver.....60
2.5 Arduino.....62
2.6 Jetson nano embeded computer.....63
2.7 Lidar.....64
2.8 Kinect v1 Time of Flight camera.....67
2.9 Writing the hardware interface of our robot.....70

3. Monte Carlo localization

3.1 What is localization.....80
3.2 Localization challenges.....80
3.3 Monte carlo localization.....81
3.4 Using the Adaptive Monte Carlo Localization Package.....86
3.4.1 URDF of the robot.....86
3.4.2 Main Launch file.....91
3.4.3 Adaptive Monte Carlo Localization package.....96
3.4.4 Simulating the robot in Gazebo.....103

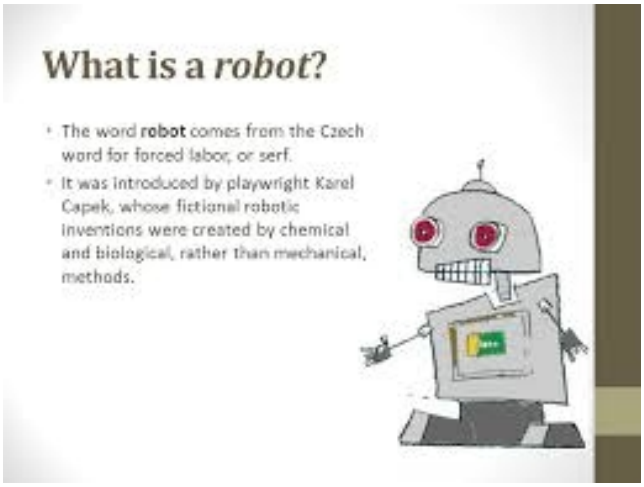
4. The Navigation Stack and Move Base
4.1 The Navigation stack in ROS.....106
4.2 Move Base.....107
4.3 Costmap_2d.....108
4.4 Congiguring the costmaps – Global and Local costmaps.....112
4.5 Setting up rviz for the navigation stack.....120
4.6 Modifying paramaters with rqt_reconfigure.....127
4.7 Avoiding Obstacles.....128
4.8 Sending goals programmatically.....129
4.9 Summary.....131

1. Intoduction to Robot Operating System (ROS)

1.1 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 it's hardware support and it's 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 run's a variant of Ubuntu 18.04 Bionic Beaver for arm processors.

1.2 What is a robot?



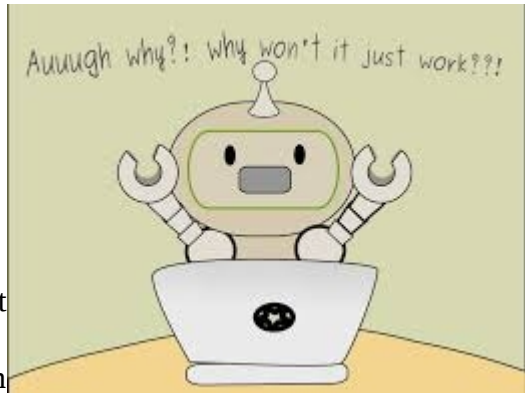
A robot is made to perform a task, this can be autonomous or semi-autonomous like the surgeon robot Da-vinci or even not physical like Alexa, because answering a question is considered an action. On the other hand a battlebot may not be considered a robot since it's completely manual operated and it has no means of perception or decision making.
So a robot must be able to percieve it's environment with sensors, perform a form of decision making and then perform a suitable action.

- ➔ **Perception:** a robot uses sensors to percieve it's 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 it's 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.

1.3 Before ROS



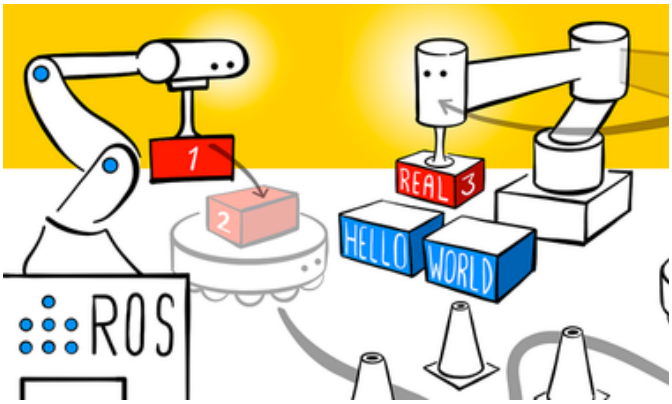
So want to build a robot? Got the hardware?
What's left to do?

Simply

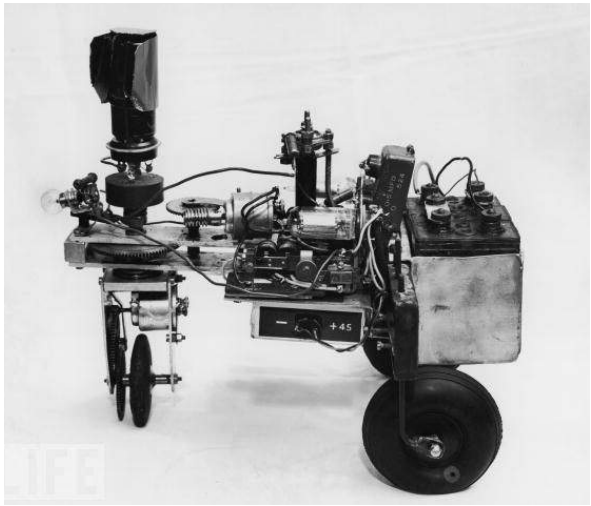
- Write device drivers for all our sensors and actuators.
- Develop a communication framework that can support different protocols.
- Write algorithms to do perception, navigation and motion planning.
- Implement mechanism to log our data.
- Write control algorithms.
- Error handling.

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

1.4 What is Robot Operating System (ROS)



ROS is not an operating sytem 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 powerfull software libraries.



[checkout this short documentary on ROS]

1.5 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 reinventig 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 costum solutions, so it was difficult for them to share code and ideas or compare results
- Ros aims to facilitate the development process by elimanting these problems
- Who uses ROS? Lot's of people and companies , there are drones, kinematic arms, wheeled robot's and even bi-pedal robots that use ROS

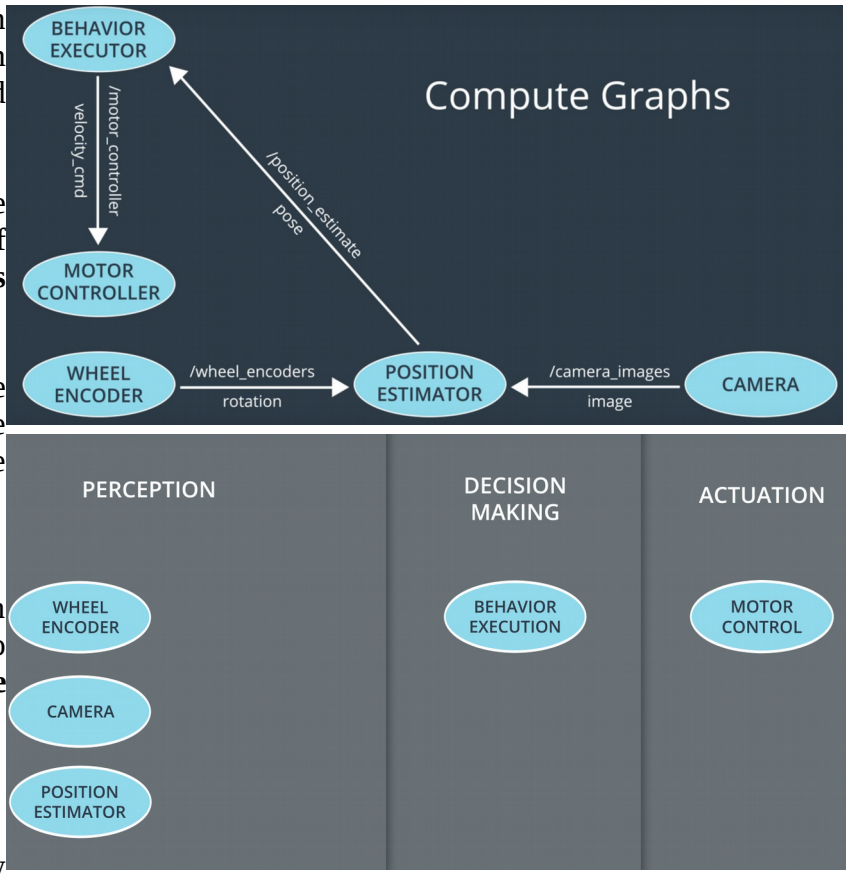
1.6 Nodes and compute graphs

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

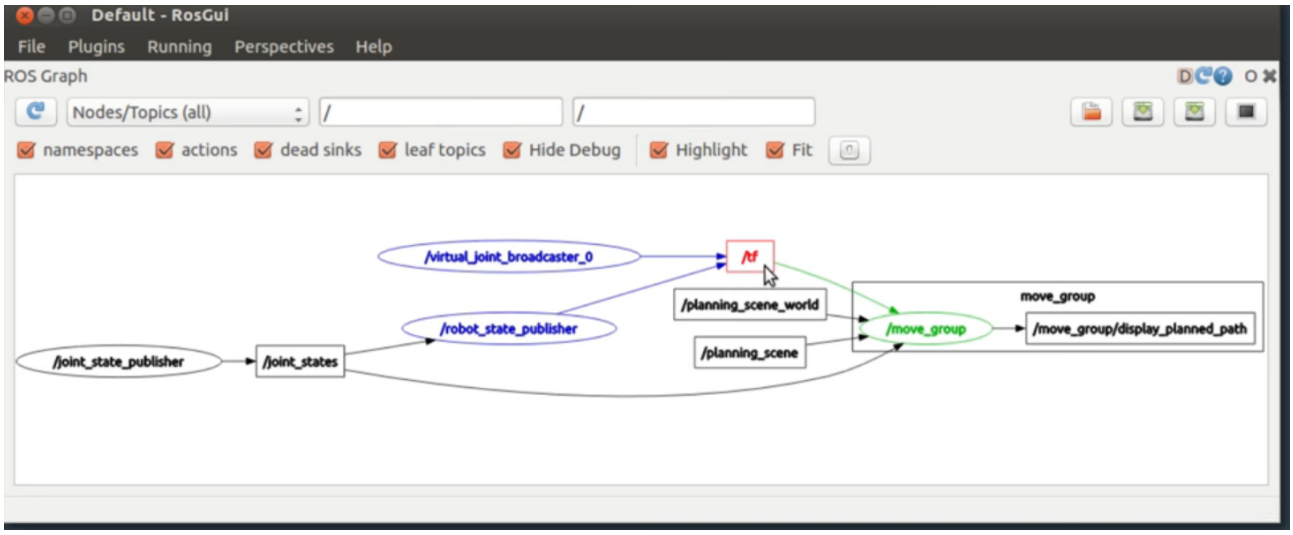
On the software side ROS manages these three complex steps **by breaking** each of them down into **many small unix processes called nodes.**

Typically each node is responsible for one small and relatively specific portion of the robot's overall functionallity for example capturing an image or moving the wheels.

ROS provides a powerfull 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 usefull 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.



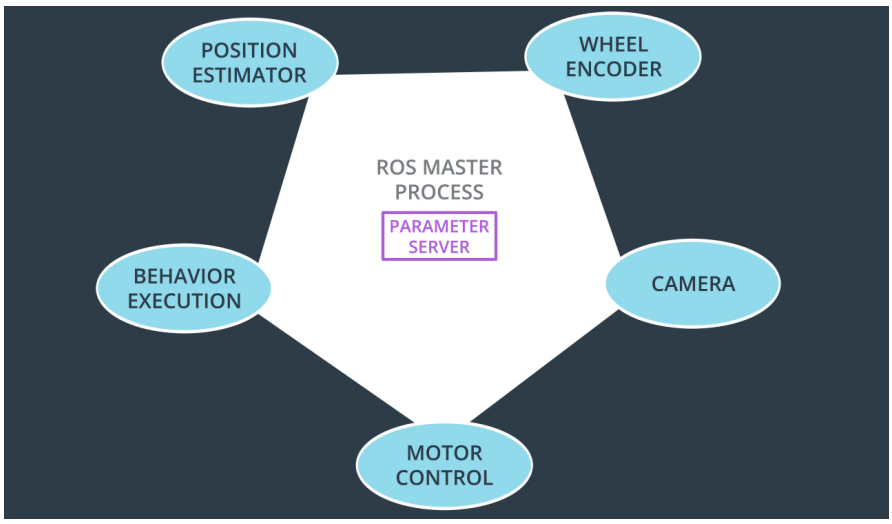
Topic: A topic is simpy a **named bus** which you can think of as a pipe bewteen nodes through which messages can flow. In order to **send a message** on a topic, we say that a node **must publish to it,** likewise to recieve a message on a topic, a node **must subscribe to it.** In the compute graph 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 predifined messages.** These can include messages for **Physical Qunatities** 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 bewteen Publishers and Subscribers **is usefull but it's not a one size fits all communication solution,** there are times when a **request-response pattern** is usefull, for these types of interactions ros provides what's called services.
Services like topics allow the passing of messages bewteen 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 paramaters 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 usefull when we tell the robot to go from A to B and we want feedback, or a kinematic arm grasping an object.

1.7 Ros Master and Paramater 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 **Paramater Server** which is a **central repository** typically used to store paramaters and configuration values that are shared amongst 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.

1.8 Packages and Catkin workspaces

- Ros provides a powerfull build and package management system called catkin, named after the flowers on the willow trees surounding their office.
- A **catkin workspace** is a directory where **catkin packages are built, modified and isntalled**, 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 recourses, which when considered together constitute some sort of usefull module

- Catkin packages may contain **source code for nodes, usefull 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 **costum** message/service/action messages.

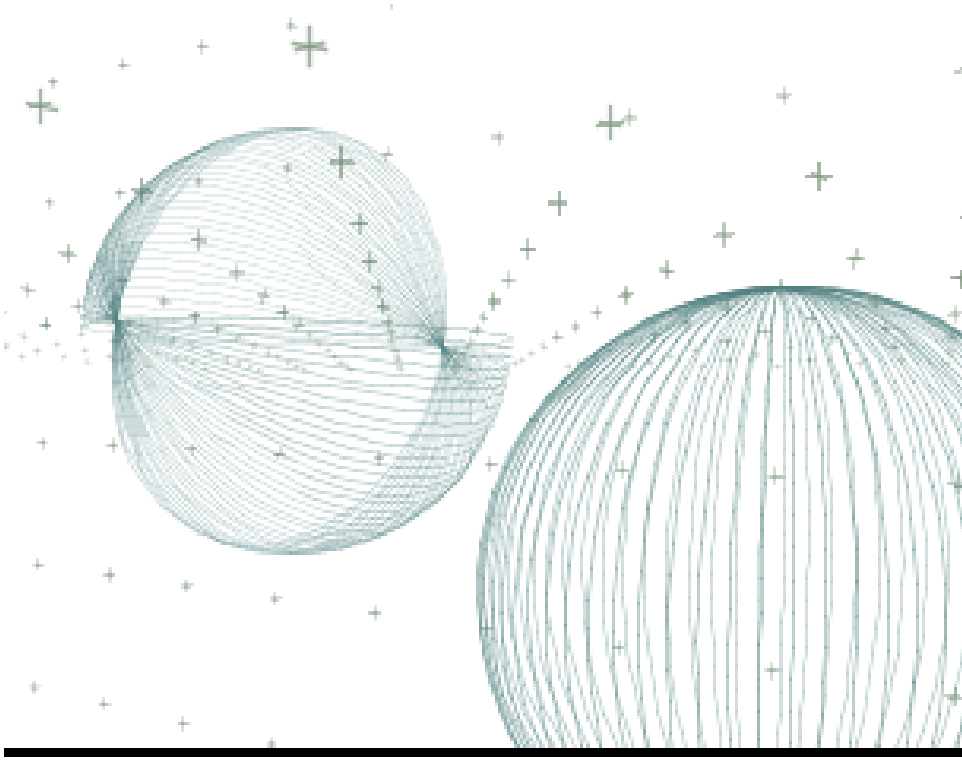
Now we have build our first ros workspace it doesn’t do much but it satisfies or the criteria of a workspace. If we **cd** to **catkin_ws** the workspace and **ls** we can see three files

- **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 souce 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 depdencies** usually other ros packages.

1.9 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 (**Recommended**)	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/

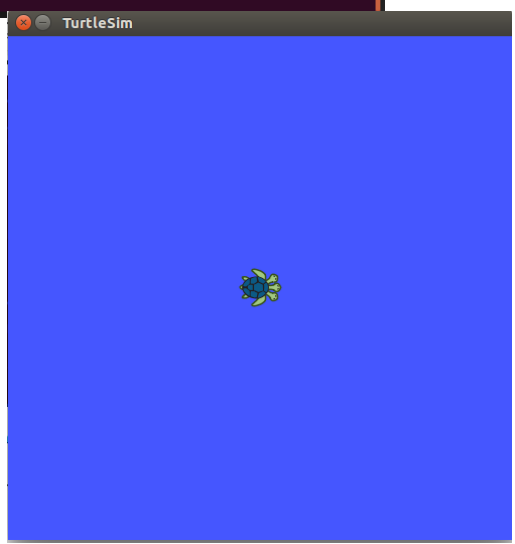
- **PYTHONPATH** ROS requires that your PYTHONPATH be updated, **even if you don't program in Python!** Many ROS infrastructure tools rely on Python and need access to the roslib package for bootstrapping.
  export PYTHONPATH=$PYTHONPATH:$ROS_ROOT/core/roslib/src
- and many more!!

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
- Fascillitating connections bewteen nodes

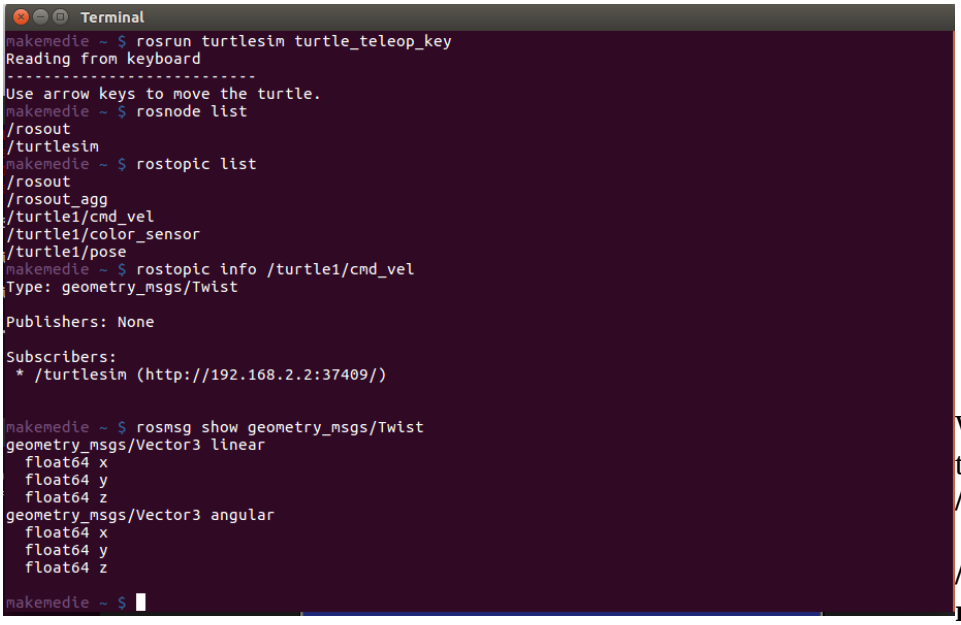
```
... logging to /home/makemedie/.ros/log/b0b144ac-e2ac-11e9-baac-7085c2697a96/roslaunch-ma
kemedie-desktop-6900.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

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 apear
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.



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.

1.10 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 usefull 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
$ roslaunch 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.

1.11 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 recieved (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 recieving any callbacks.** However we add it for good measure. Since if it don’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 giong 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 a write a node to recieve 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 depedencies

$ catkin_create_pkg pub_sub roscpp
Creates a pub_sub package with roscpp as depedency

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 costum 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 depedency

```
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
)
```

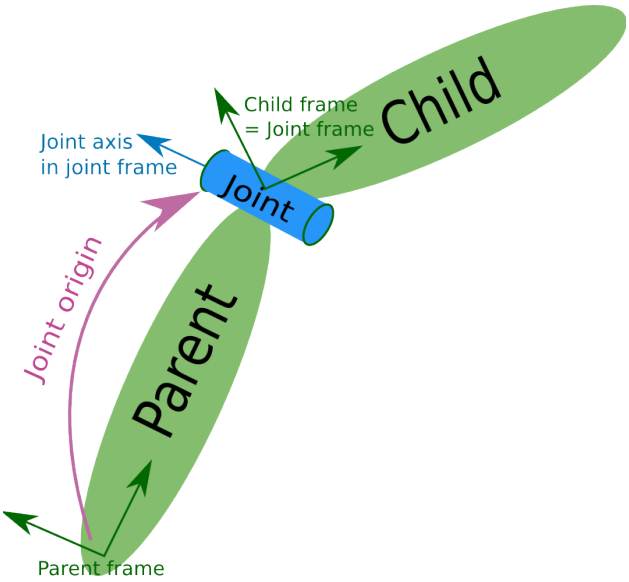
Also

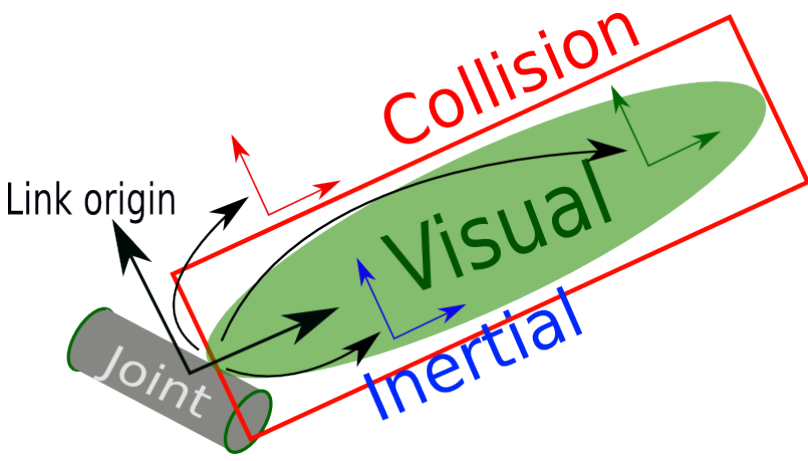
```
generate_messages(

  Depedencies

  std_msgs

)
```

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

rosmsg show package_name/Age

1.12 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 of

	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>
  <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 **weld** 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 **Revolute** joint but has no limits. It can rotate continuously about an axis.
Planar	A 2D **Prismatic** 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
<origin>	This is the transform from the parent link to the child link. The joint is located at the origin of the child link.
<parent>	Name of the Parent link for the respective joint.
<child>	Name of the child link for the respective joint.
<axis>	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 paramaterized 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" />
```

Paramaterized 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

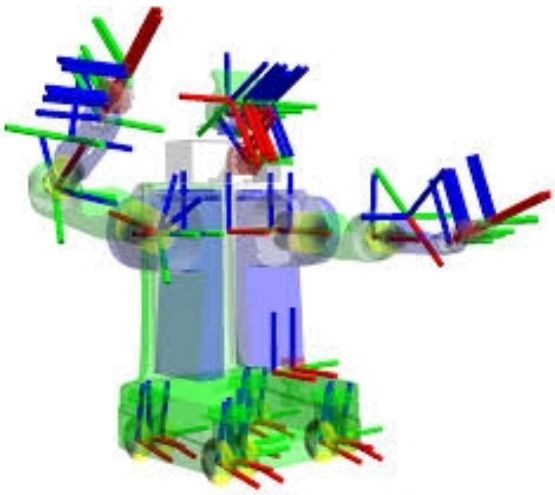
```
<xacro:include filename="$(find lynxbot_description)/urdf/common_properties.urdf.xacro" />
```

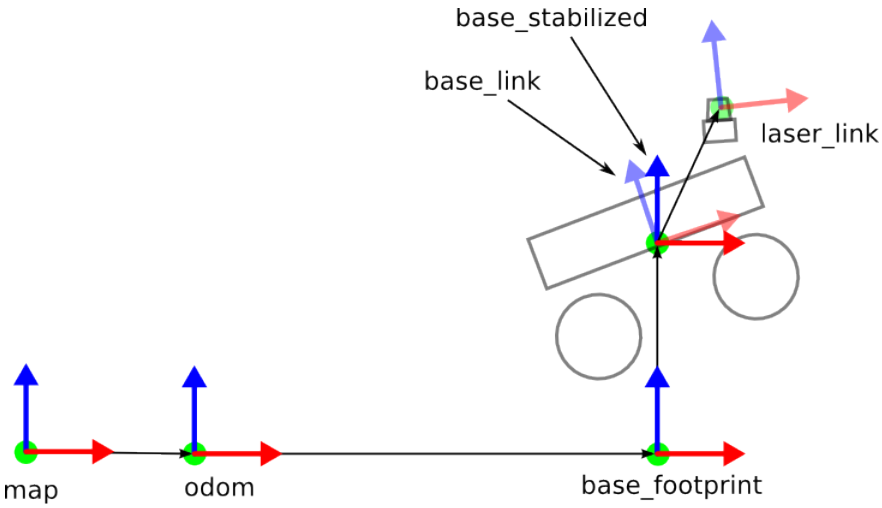
Common Trick 1: Use a name prefix to get two similarly named objects

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.

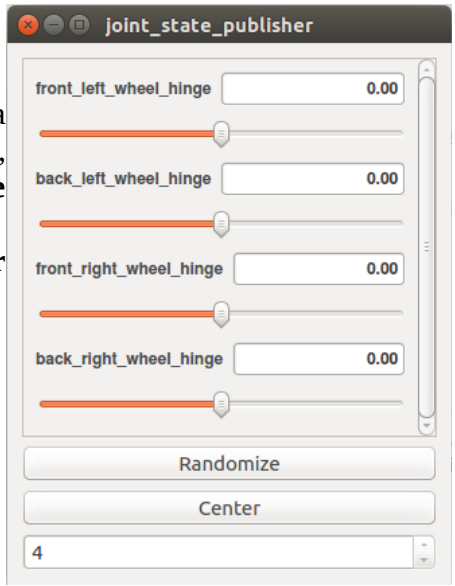
In the next chapter we will see how i created the xacro/urdf of my robotic vehicle, using one macro for the 4 wheels, alongside the lidar sensor and the kinect camera.

1.13 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 bewteen 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="$(find xacro)/xacro '$(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 -->
```

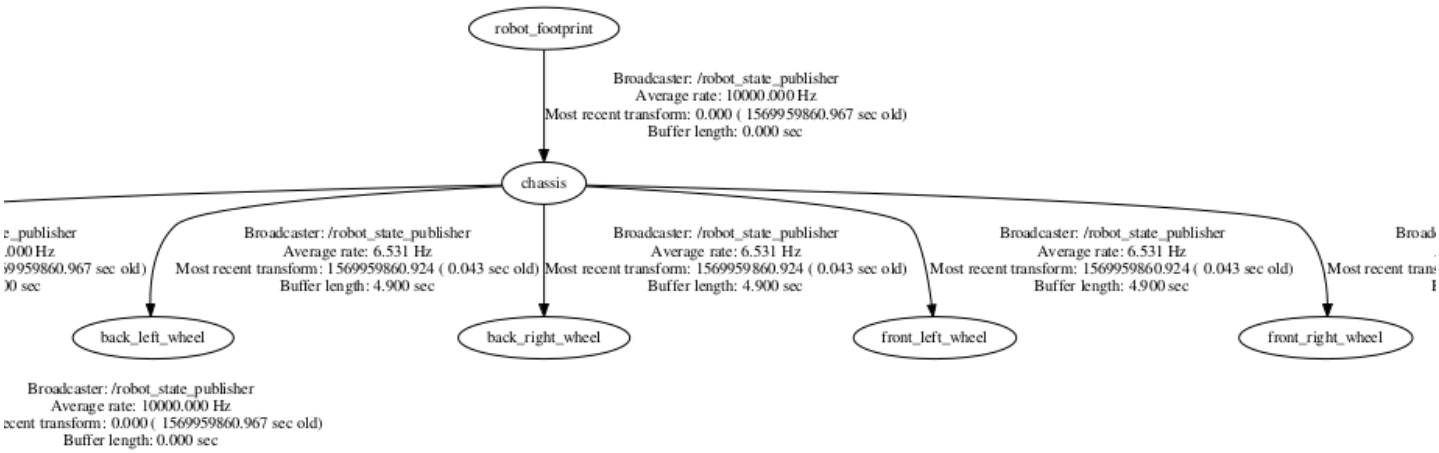
```
    <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher"
respawn="false" output="screen">

        <param name="publish_frequency" type="double" value="10.0" />

    </node>

</launch>
```

We first **load the URDF to the paramater 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 ($ **roslaunch 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**



1.14 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:

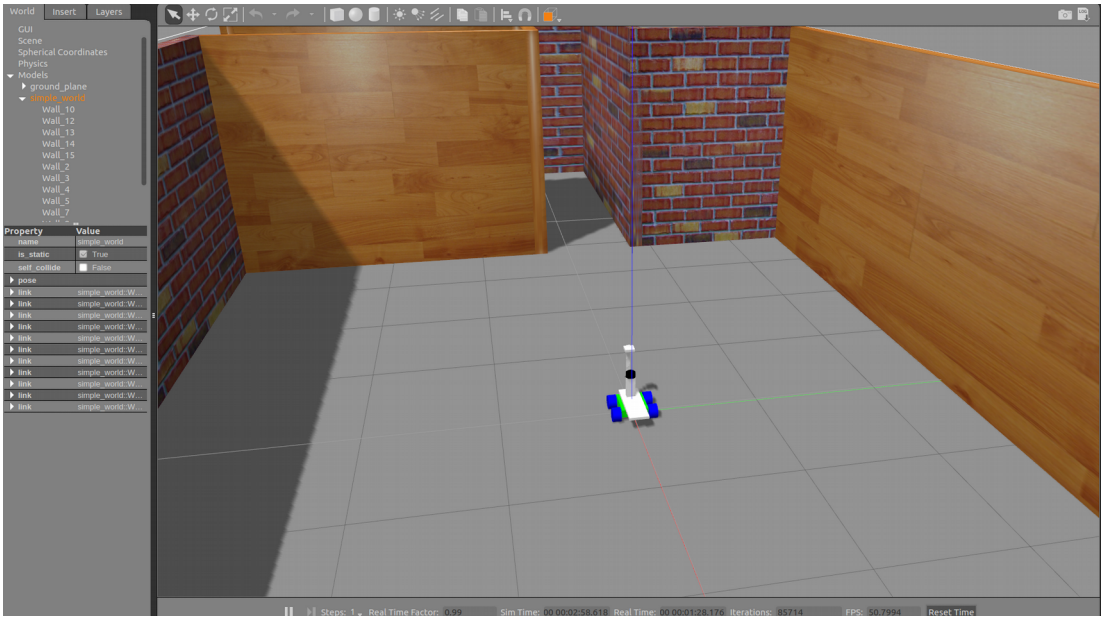
$ gzclient

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

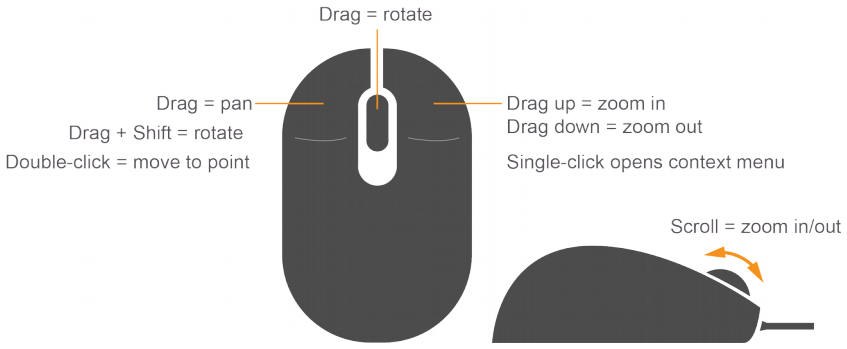


The Gazebo gui is divided into 4 major sections:

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 Ctlr+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

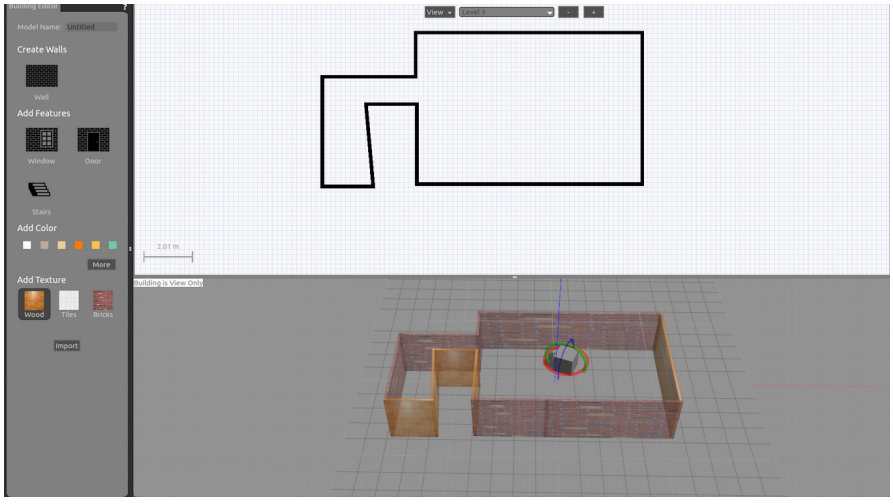
```
<?xml version="1.0" ?>

<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 descibed 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 paramater 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.

Writing a Gazebo service

First of all **services are Synchronous** meaning that the program can't continue until it recieves 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::ServiceClientdelete_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 usefull 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

1.15 Actions

Actions unlike Service are asynchronous and can provide feedback.

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

Action Client ----- **goal** -------> **Action Server**
----- **cancel** ----->
<---- **status** ------

<----- **result** -------
<----- **feedback** ---

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;
}
```

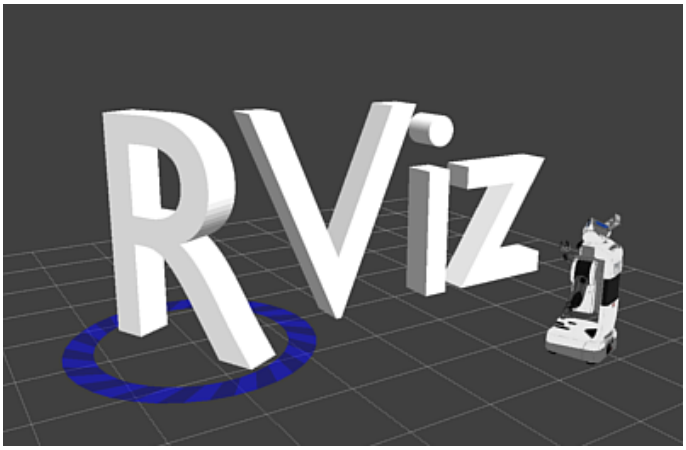
Writing a simple client

```
#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.

1.16 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.

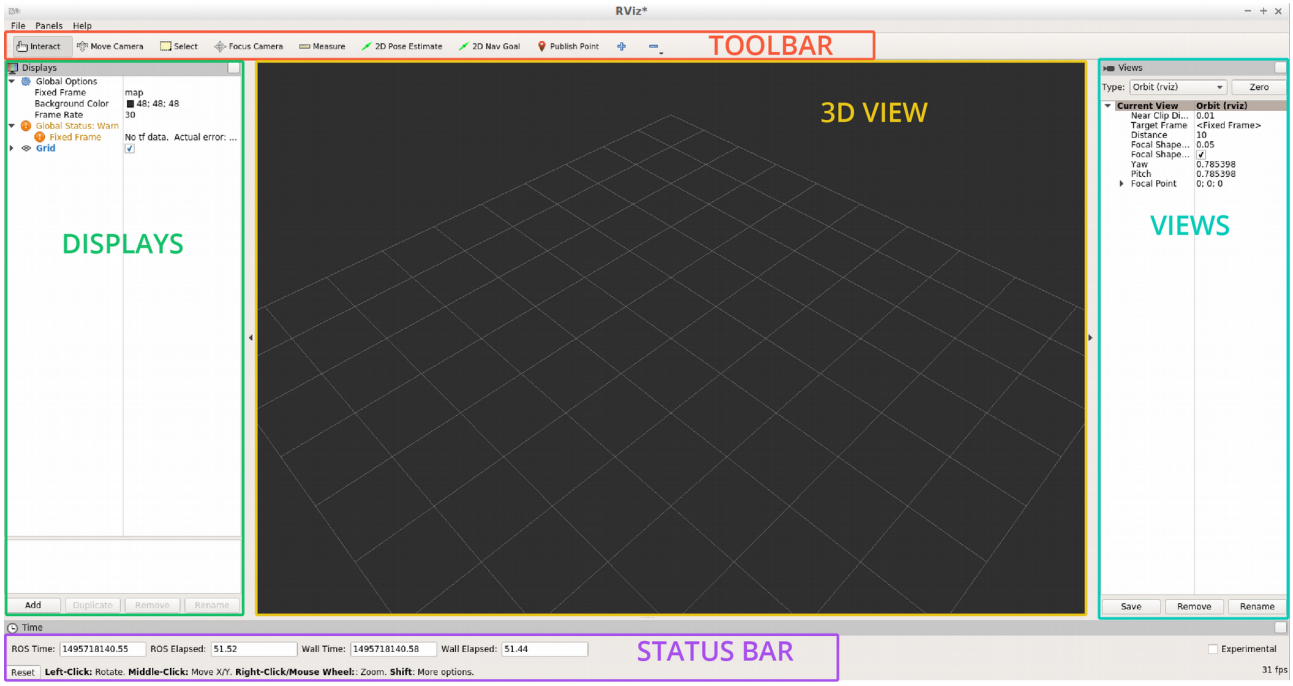
Since rviz is a ROS package, you need roscore running to launch rviz. In a terminal spin up roscore:

$ 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.**

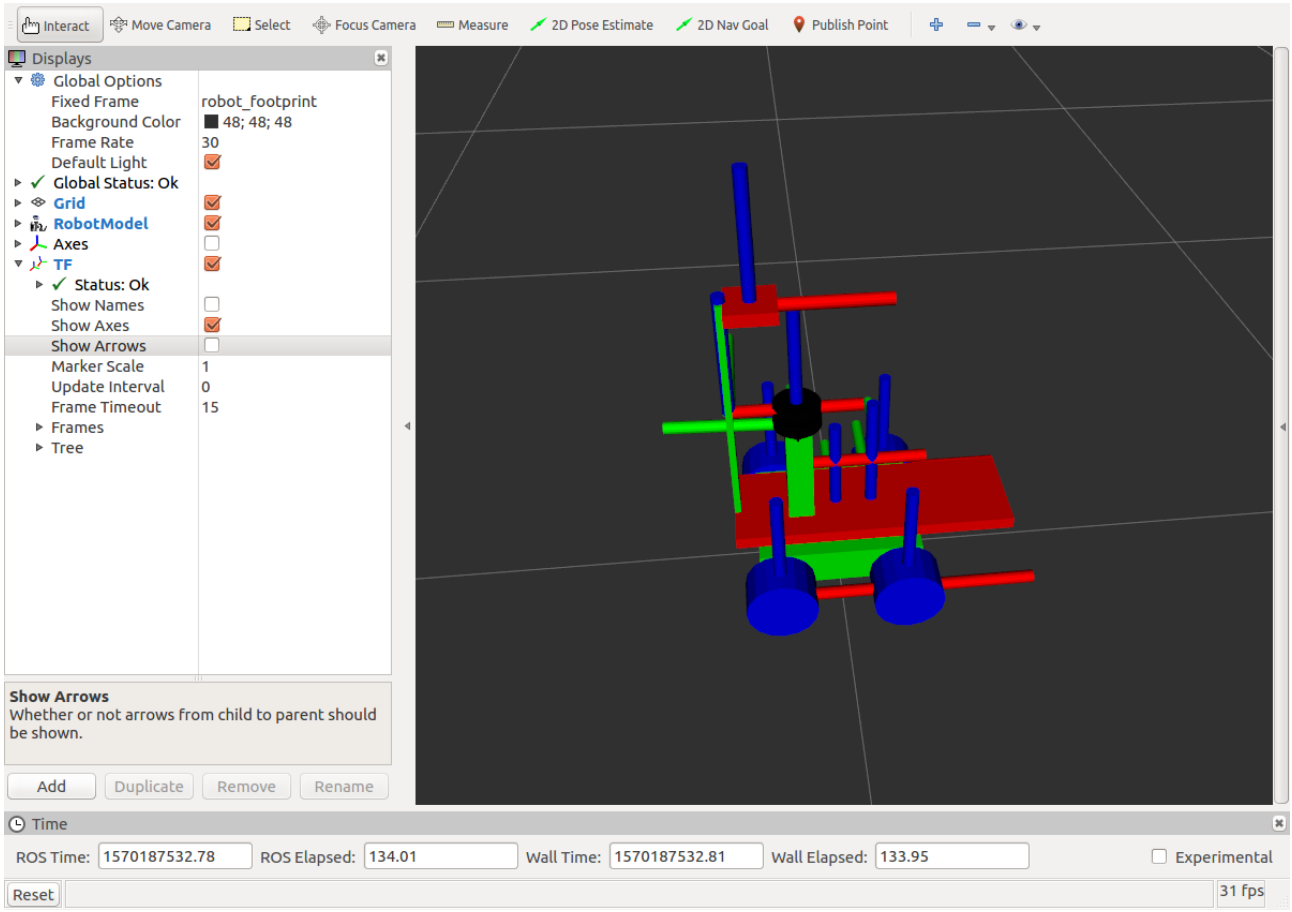
Rviz by default starts up with two fixed property fields that cannot be removed - **Global Options** and **Global Status**. While these are not displays, **one governs simple global settings,** while the other **detects and displays useful status notifications.**

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 a**nd 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.

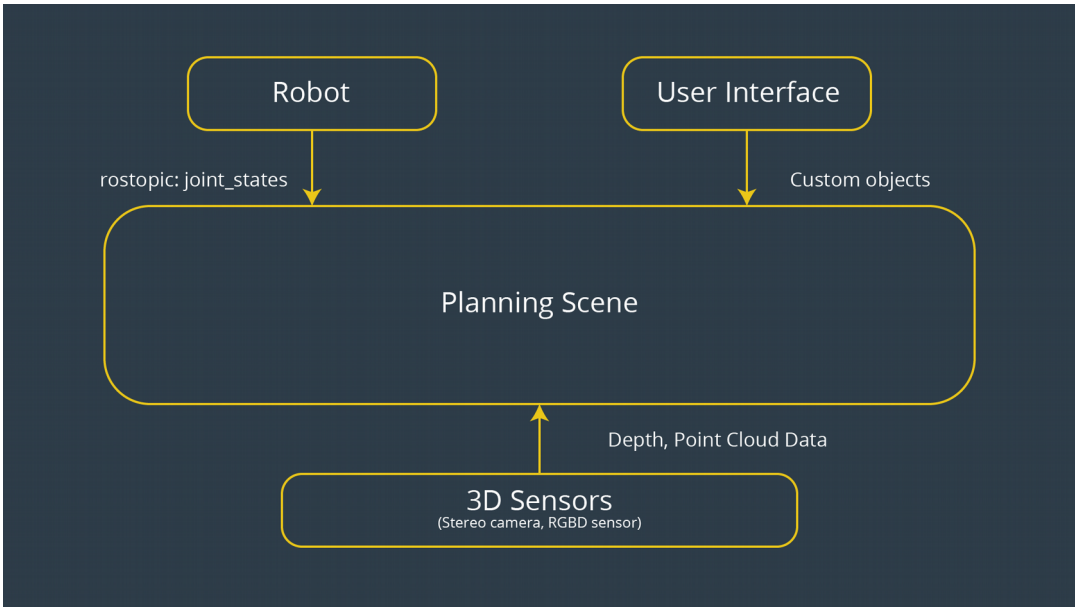
1.17 MOVEIT

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.



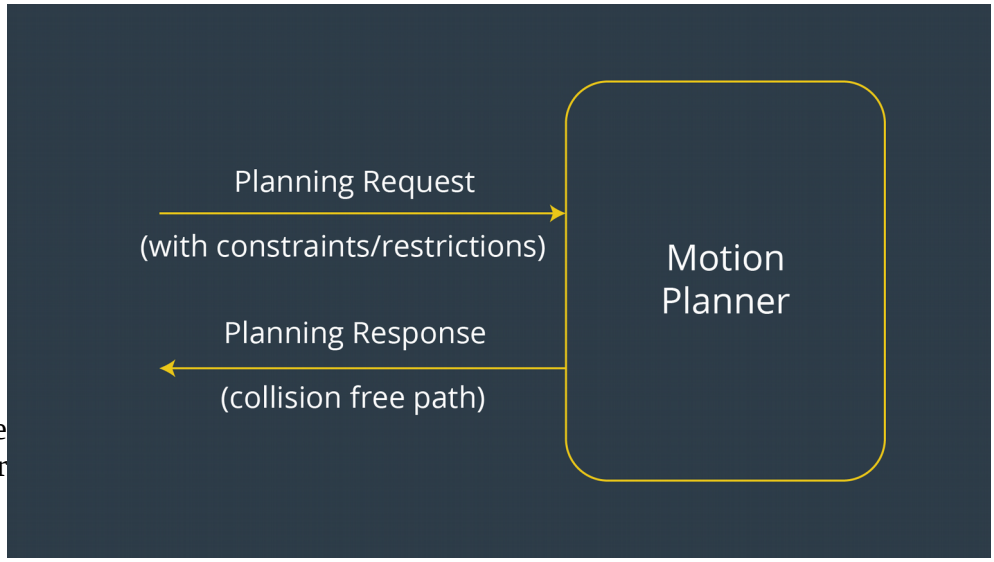
You may also insert abstract or virtual objects into this so called 3D map by publishing them to specific topics.

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.**

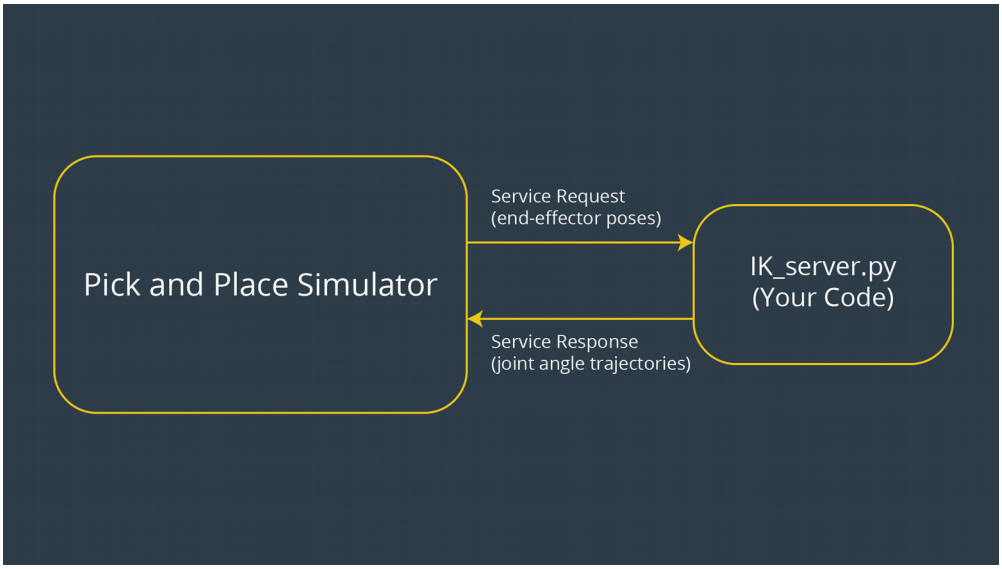


In response to our request message, Moveit! generates a desired collision free trajectory using the Planning Scene discussed above.
A point to be noted is that a motion planner only generates a collision free path from the start state of the robot to its end state, and then Moveit! converts this path 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

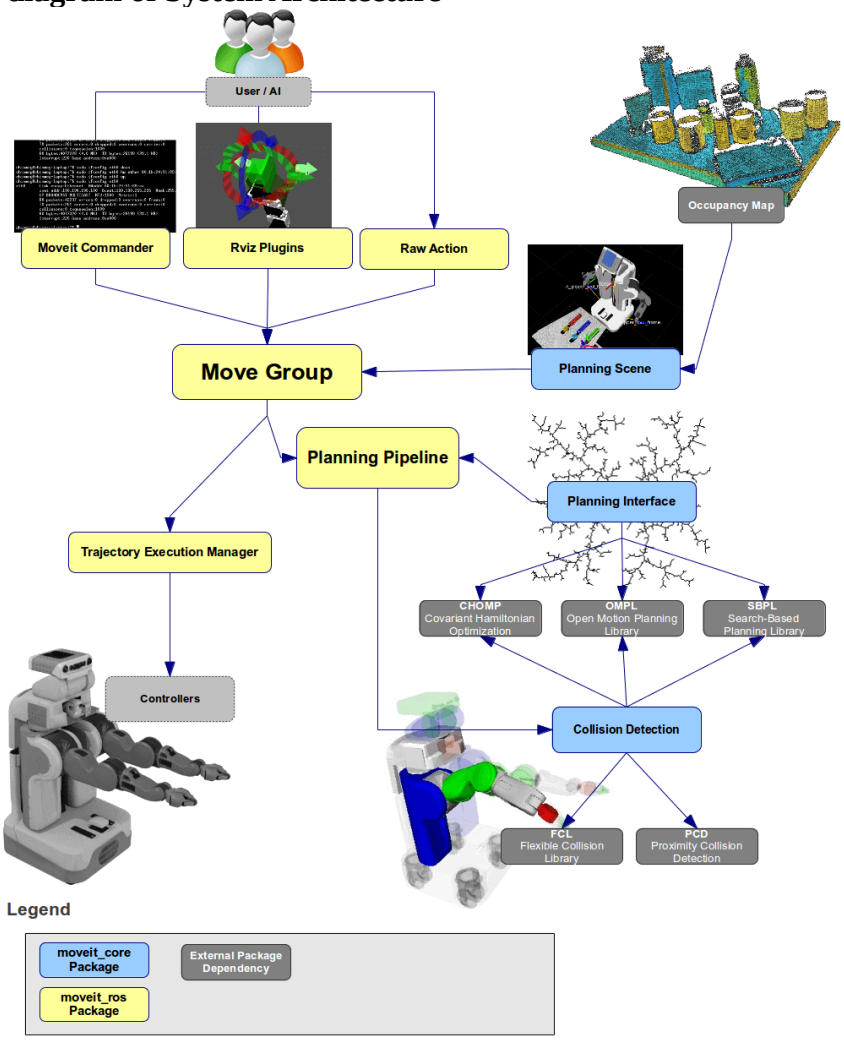
At various stages in the process of motion planning, Moveit! needs to convert the joint_state of the robot into the end-effector pose using Forward Kinematics, and vice-versa using Inverse Kinematics.
While **ROS's RobotState class provides access to Forward Kinematics functionality,** Moveit! Allows users to write **their own Inverse Kinematics implementations.**



While your 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:

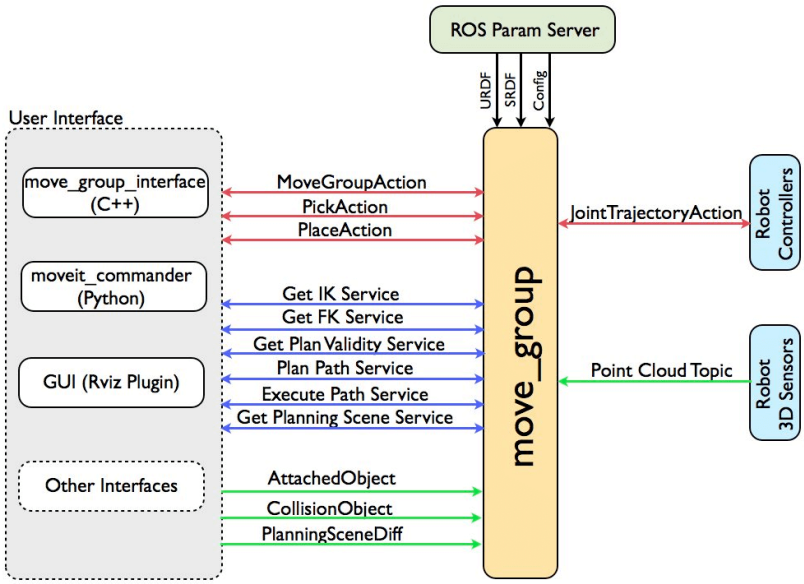
- The Kinematics and Dynamics Library (KDL) developed by orocos
- IKFast Solver by OpenRAVE

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!!

1.11 Debugging Tools

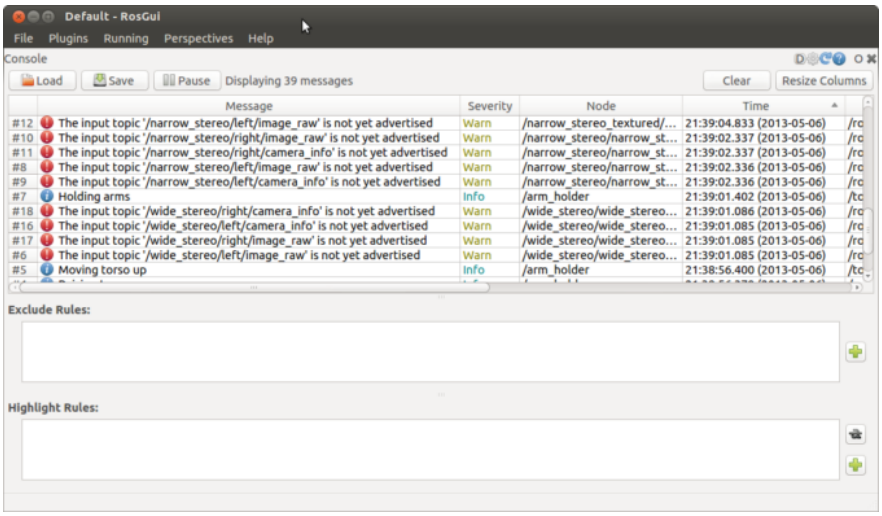
Over the years, developers have built an amazing suite of tools for debugging and introspecting the ROS system.

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

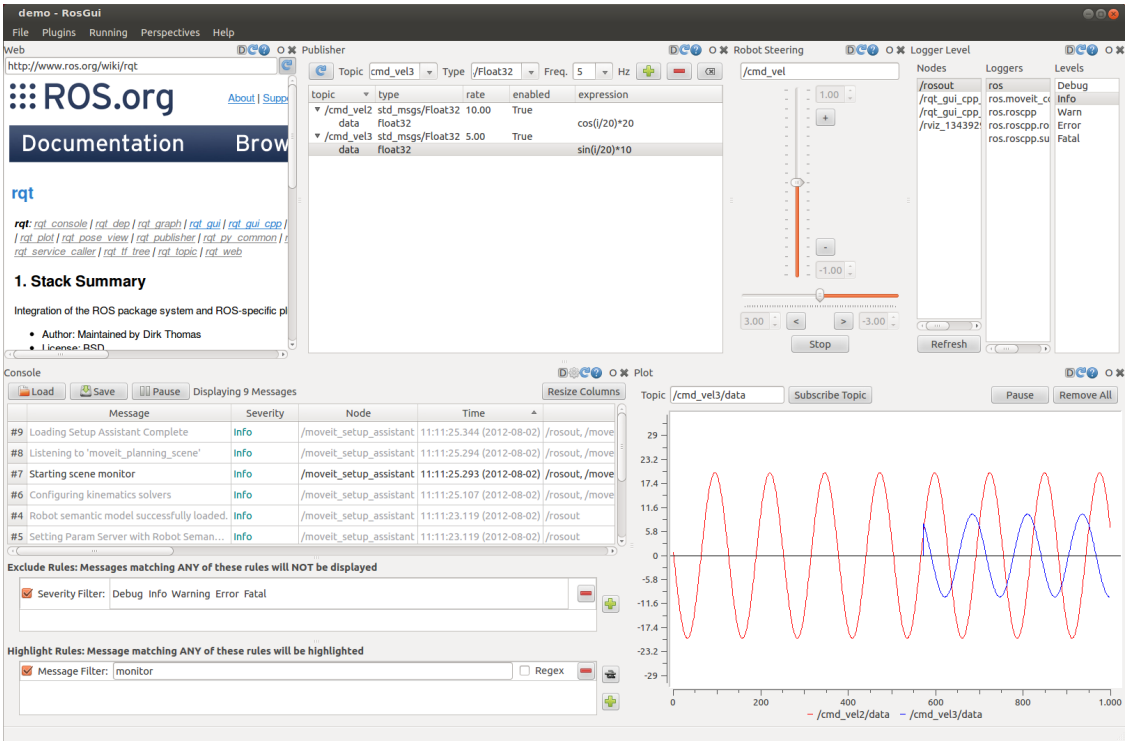
Launch files often end up nested three or four layers deep as you try to keep your code modular and clean. As you start calling launch files in MoveIt or the ROS navigation stack, the launch file include structure can get even more convoluted, making debugging a huge pain. RQT Launchtree is designed 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 paramaters 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.

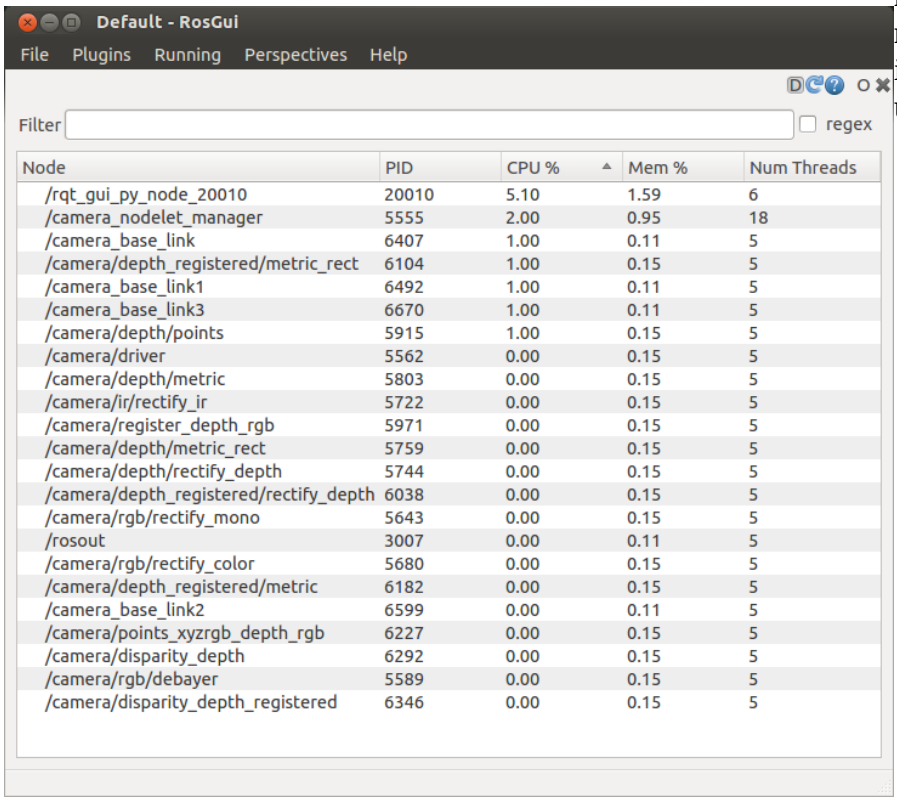


rosnode info

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.

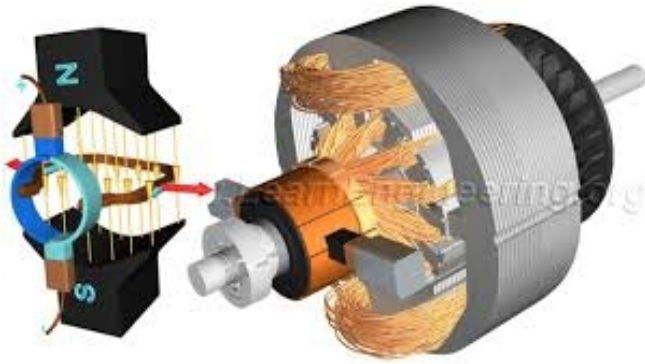


roswtf

Perhaps the 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.

2. Hardware and Sensors Used

2.1 DC Motor

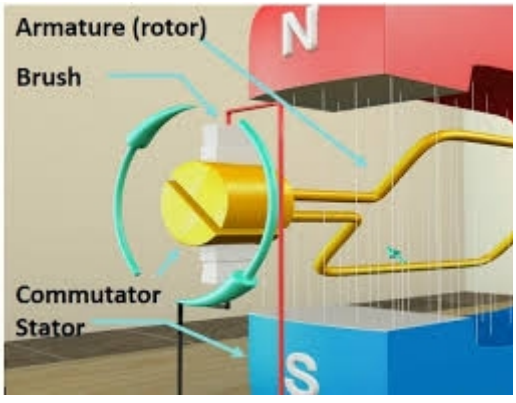


The Basic DC Motor

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 were 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 produce 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

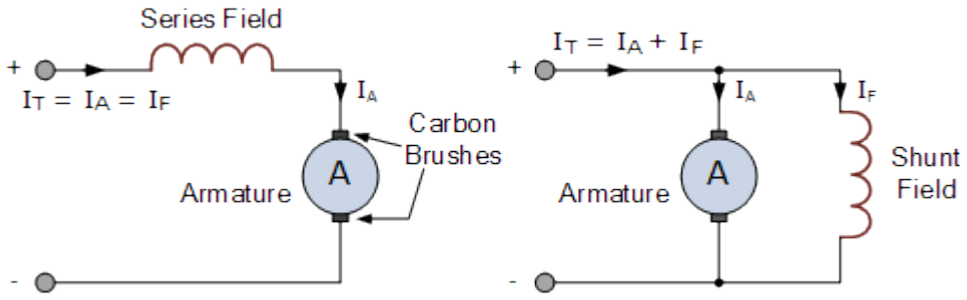


A conventional brushed DC Motor consist basically of two parts, the stationary body of the motor called the Stator and the inner part which rotates producing the movement called the Rotor or "Armature" for DC machines.

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.

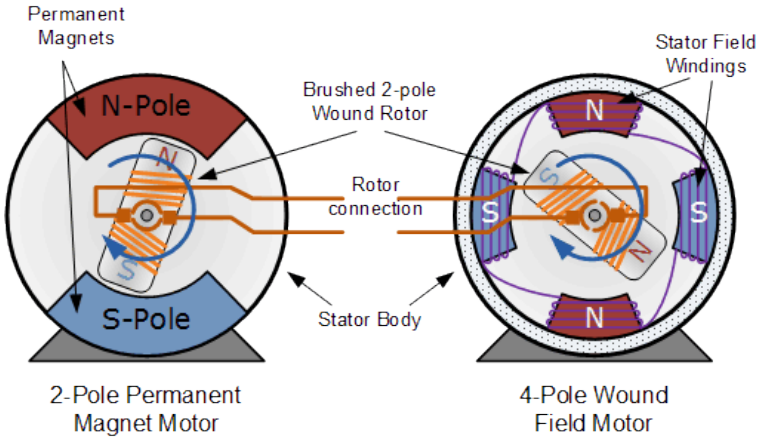
These electromagnetic coils which form the stator field can be electrically connected in series, parallel or both together (compound) with the motors armature. A series wound DC motor has its stator field windings connected in *series* with the armature. Likewise, a shunt wound DC motor has its stator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



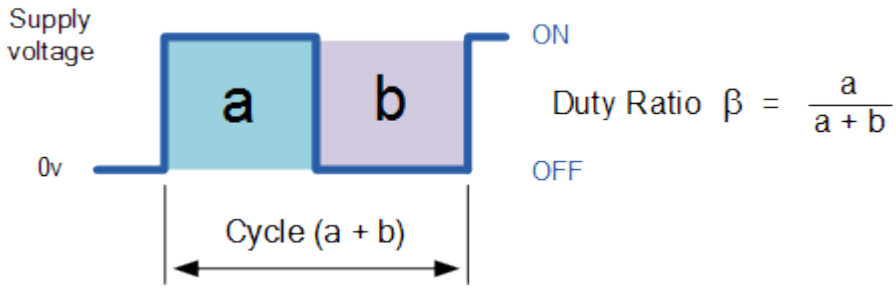
The Permanent magnet (PMDC) brushed DC motor is generally much smaller and cheaper than its equivalent wound stator type DC motor cousins as they have no field winding. In permanent magnet DC (PMDC) motors these field coils are replaced with strong rare earth (i.e. Samarium Cobolt, or Neodymium Iron 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 β is given as:

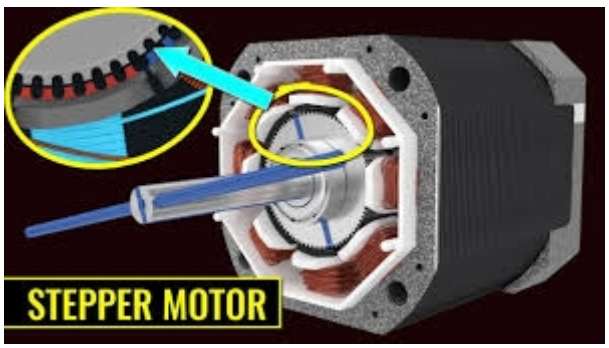


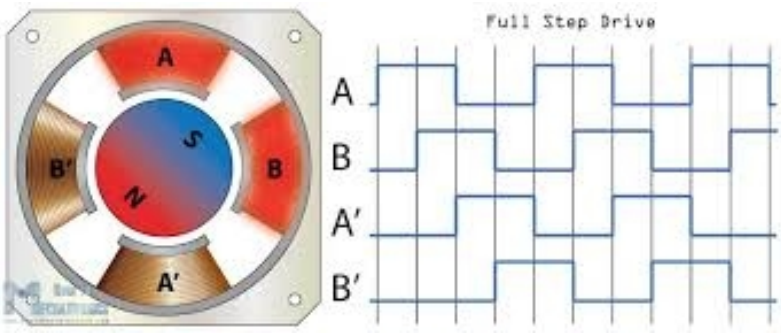
and the mean DC output voltage fed to the motor is given as: Vmean = β x Vsupply. Then by varying the width of pulse a, the motor voltage and hence the power applied to the motor can be controlled and this type of control is called Pulse Width Modulation or PWM.
Another way of controlling the rotational 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).

2.2 THE DC 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 it 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, lets assume that a stepper motor completes one full revolution (360^{o} in exactly 100 steps.

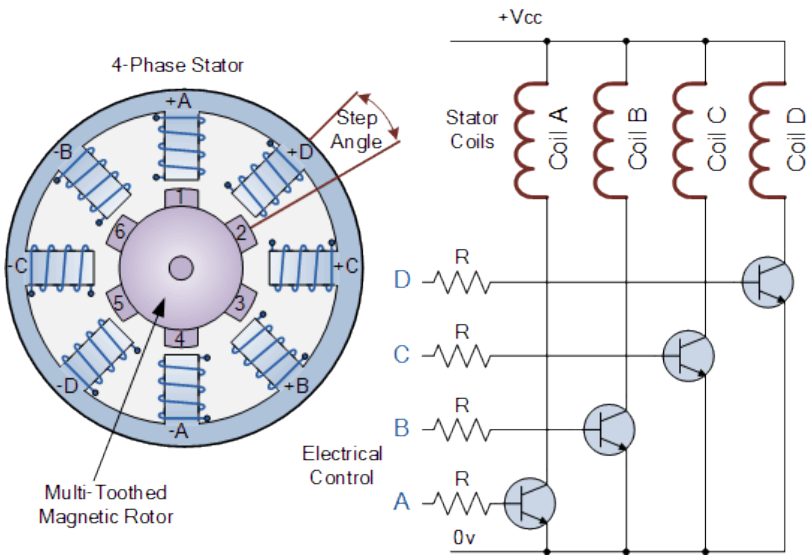
Then the step angle for the motor is given as 360 degrees/100 steps = 3.6 degrees per step. This value is commonly known as the stepper motors **Step Angle**.

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 require 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 stators 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 degs/(50×4)).

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 x 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 x 4 coils) possible positions or “steps” for the rotor to complete one full revolution. Therefore, the step angle above is given as: $360^{o}/24 = 15^{o}$.

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 degrees/(3.6 degs/step) = 80 pulses applied to the stator coils.

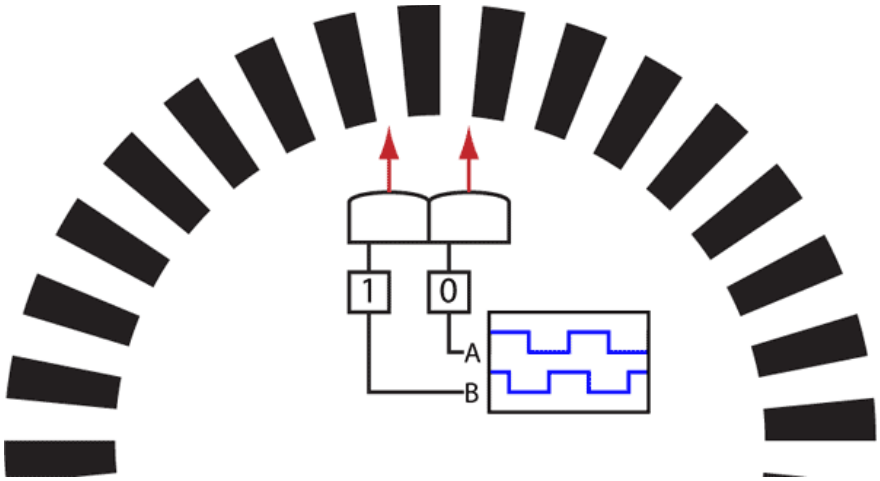
There are many stepper motor controller IC’s available which can control the step speed, speed of rotation and motors direction. One such controller IC is the SAA1027 which has all the necessary counter and code conversion built-in, and can automatically drive the 4 fully controlled bridge outputs to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

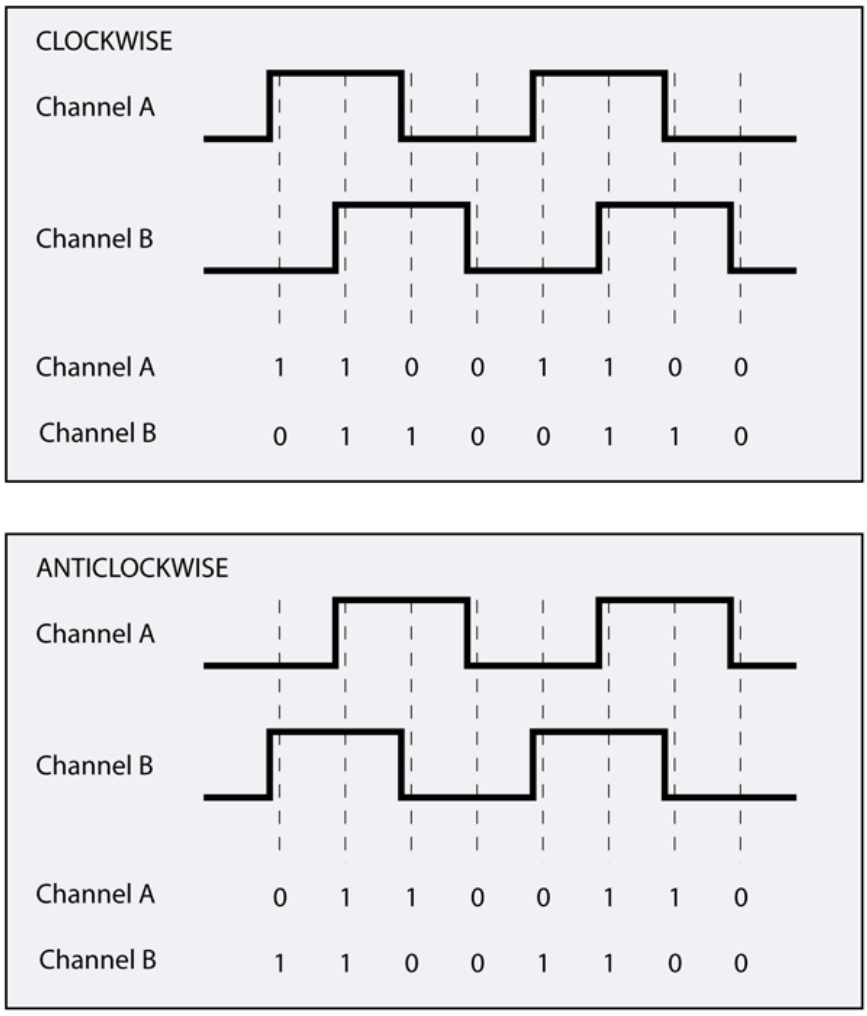
2.3 Quadrature encoder

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 -> high -> 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?
Lets 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

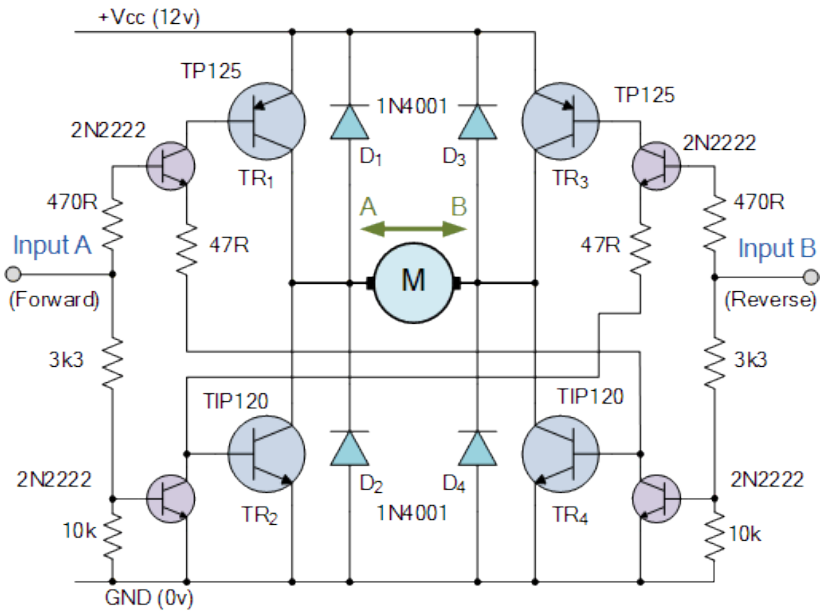
2 Channel B is HIGH

=>

Counter-Clockwise motion

2.4 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

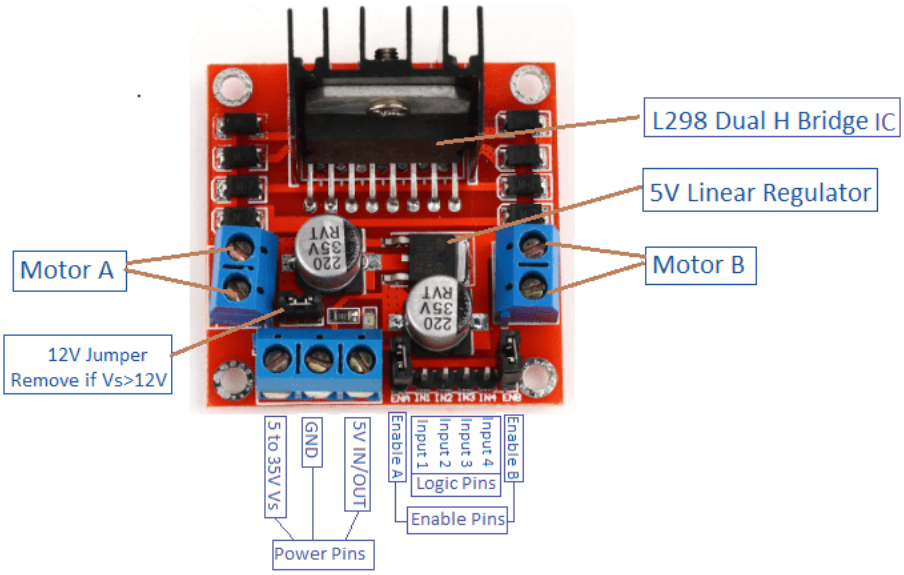
Input A	Input B	Motor Function
TR1 and TR4	TR2 and TR3	
0	0	Motor Stopped (OFF)
1	0	Motor Rotates Forward
0	1	Motor Rotates Reverse

1	1	NOT ALLOWED

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.

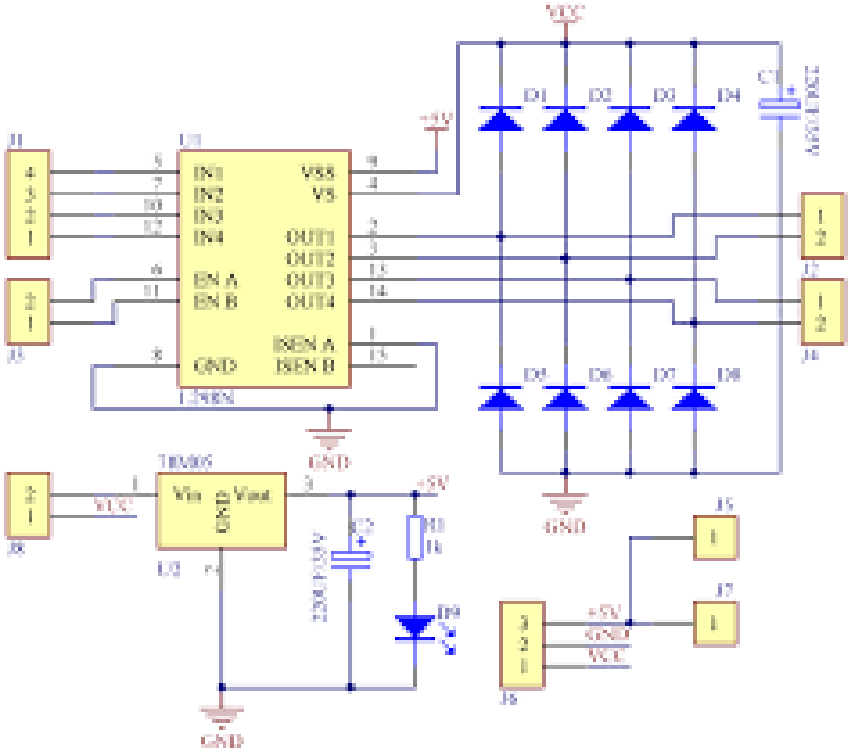
Commercial off the shelf decoder IC's **such as the SN754410 Quad Half H-Bridge IC or the L298N which has 2 H-Bridges are available with all the necessary control and safety logic built in are specially designed for H-bridge bi-directional motor control circuits.**

L298N

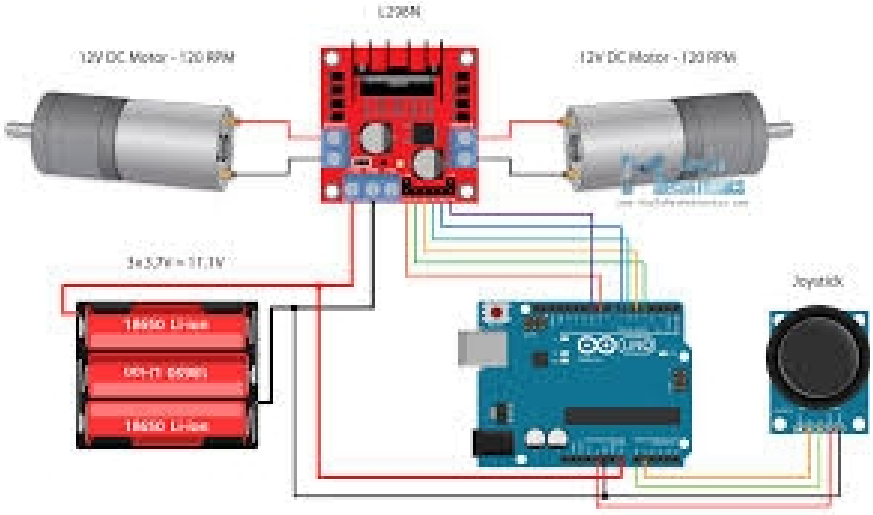


This dual bidirectional motor driver, is based on the very popular L298 Dual H-Bridge Motor Driver Integrated Circuit. The circuit will allow you to easily and independently control two motors of up to 2A each in both directions.It is ideal for robotic applications and well suited for connection to a 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. 5 A4988 driver

Overview

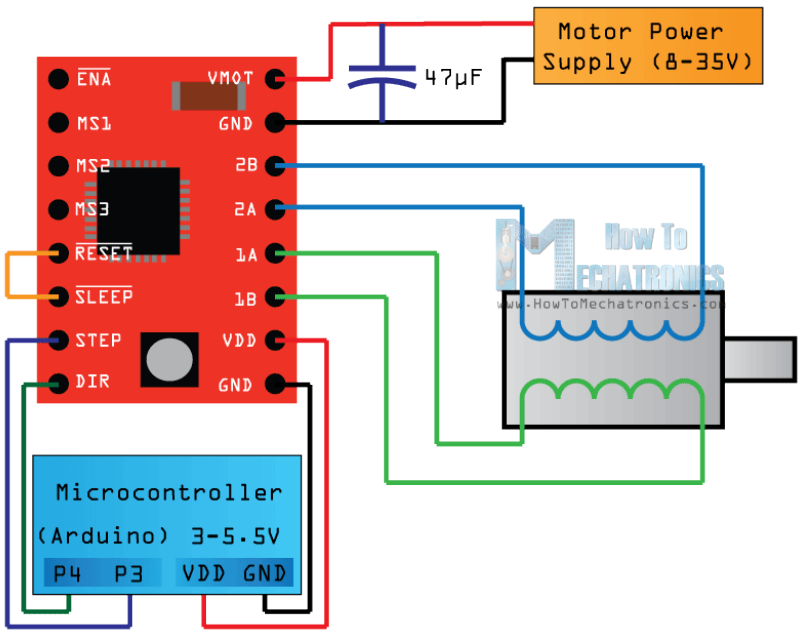
The A4988 is a microstepping driver for controlling bipolar stepper motors which has built-in translator for easy operation. This means that we can control the stepper motor with just 2 pins from our controller, or one for controlling the rotation direction and the other for controlling the steps.

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 µF 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	Halft 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.

2.6 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.

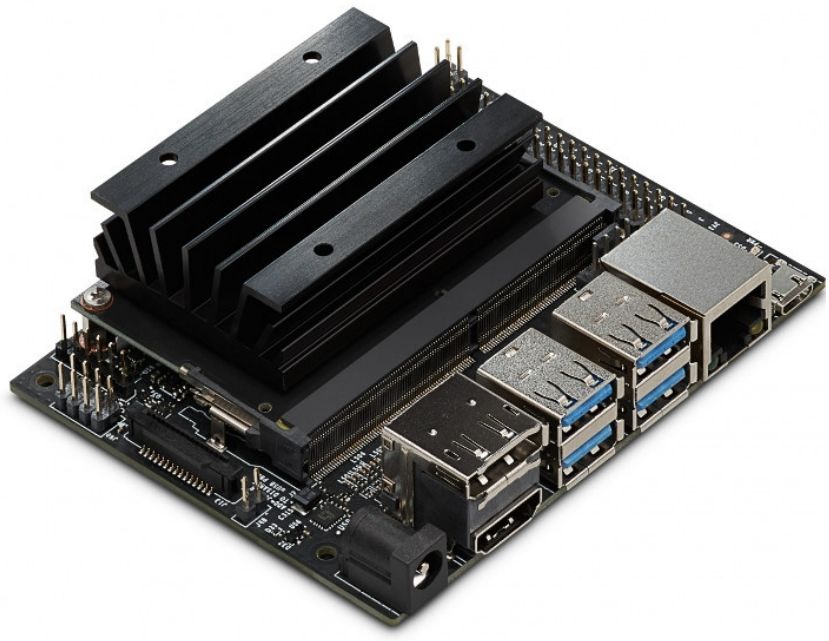
The Arduino project started in 2005 as a program for students at the Interaction Design Institute Ivrea in Ivrea, Italy, aiming to provide a low-cost and easy way for novices and professionals to create devices that interact with their environment using sensors and actuators. Common examples of such 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 chapet 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 uneccesary for our project, especially when we use stepper motors, however arduino provides as with hard pwm and an easy way to read the encoder signals reliably.

2.7 Jetson Nano



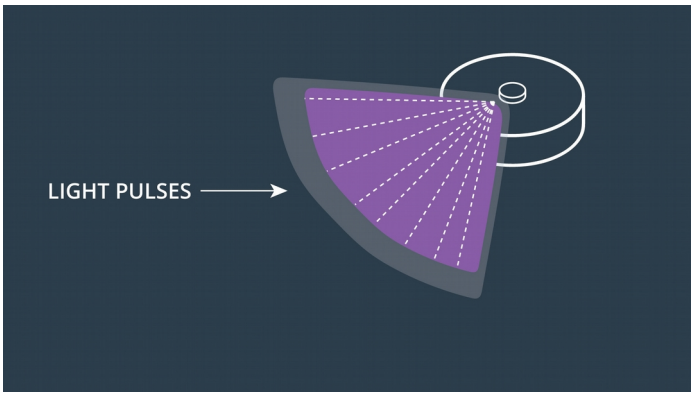
The brain of our robot is a small factor computer called Jetson Nano Developer Kit.
Jetson Nano runs full desktop Ubuntu out of the box, and is supported by the JetPack 4.2.
NVIDIA Jetson Nano **targets AI projects such as mobile robots and drones, digital assistants, automated appliances and more.**

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

2.8 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 horizonal 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	

2.9 Kinect v1 Time of Flight camera

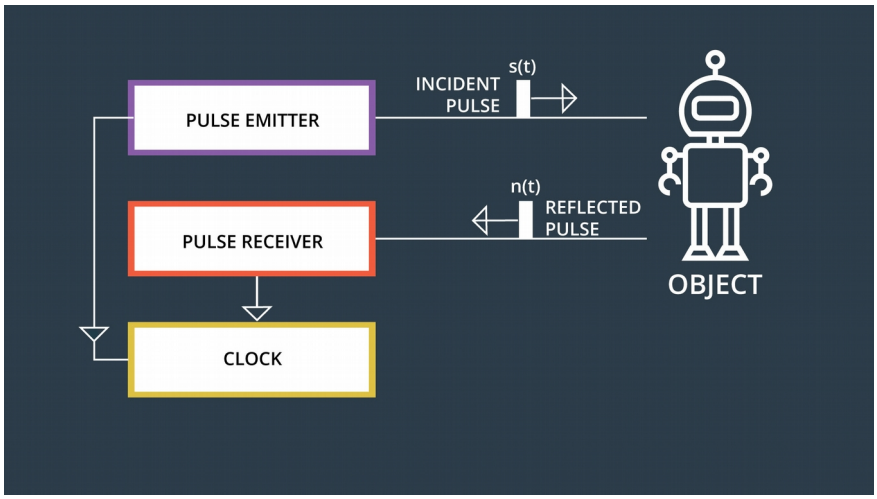
Time of Flight camera

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

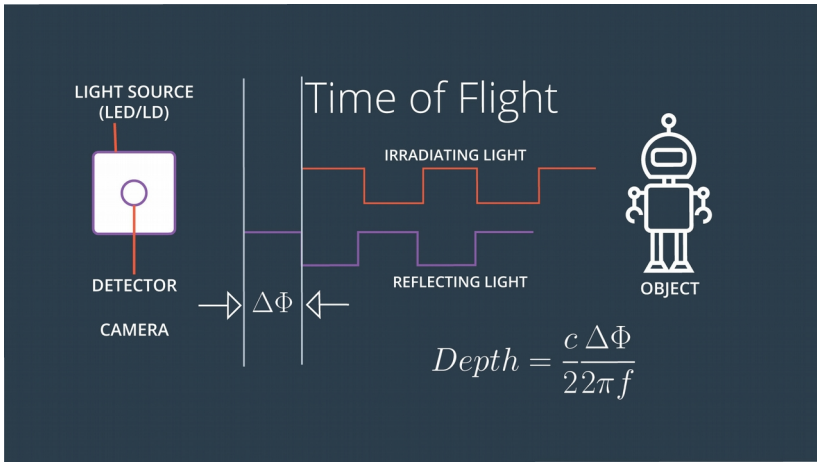
Pulse Runtime Sensor

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.

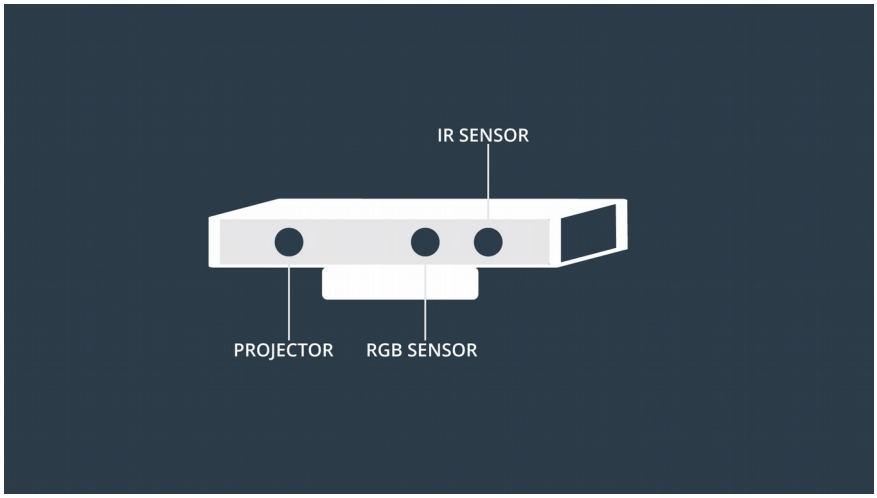


Pros	Cons
Compact size	Secondary reflections
Rapid data acqusition	Ambient light interference (do not work well outdoors)
Ideal for real-time applications	Multiple ToF camera may interfere with one another
	Cannot easily detect glass

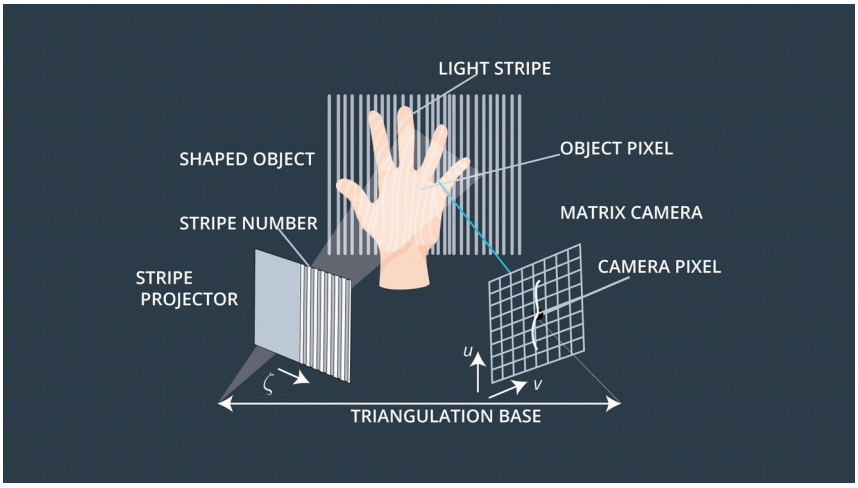
RGB-D Camera

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

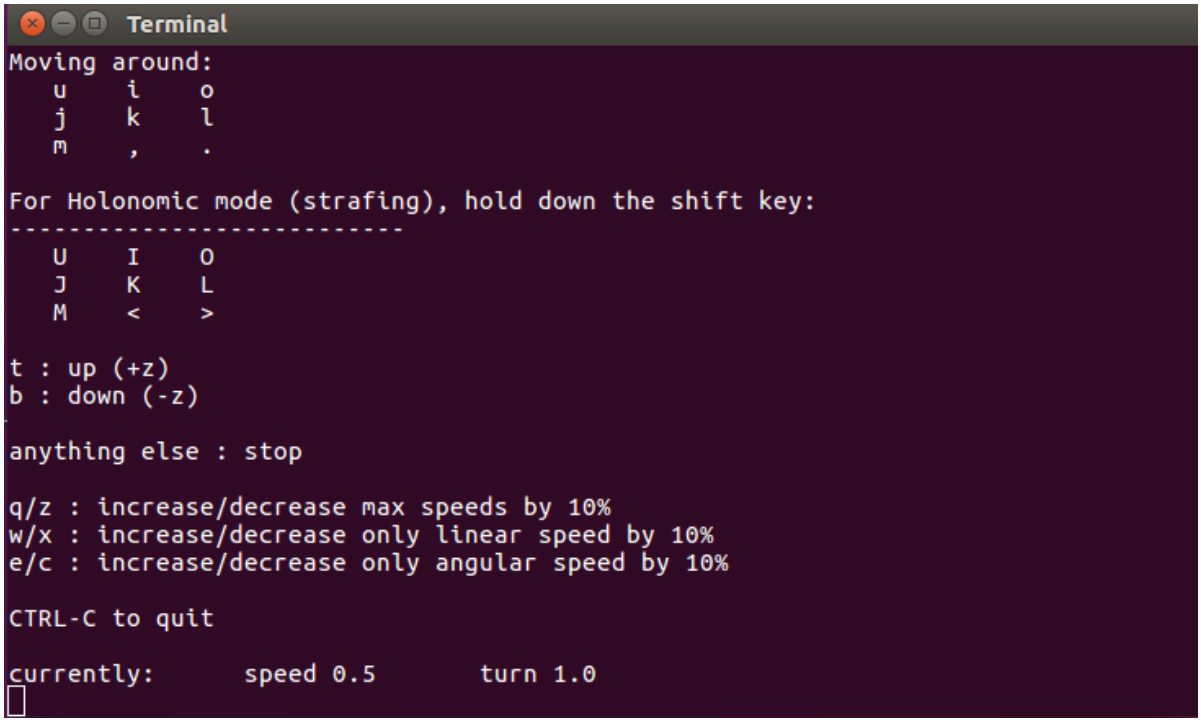
We will use the Kinect v1 Time of Flight camera to build 3D Maps of our environment. Although we can we will not use this camera for obstacle avoidance rather for object recognition and map building.

2.10 Writing the hardware interface of our robot

As we have seen in our **turtlesim example.** A mobile base is usually controlled under ROS via a **Twist message** publish over a topic often **called cmd_vel.**

We can run this node with the command:

$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py



Now by using the keyboard we can send Twist messages over the cmd_vel topic.
With the command
$ rosmsg show geometry_msgs/Twist

We see that the message consists of two geometry_msgs/Vector3 fields one used for linear velocity and the other for angular velocity. It's vector is represented by 3 float64 variables representing the x, y and z components of each velocity type.

Now let's see how we can run a ROS node on the arduino with rosserial and explain the code.
Depedencies arduino library: rosserial, quadrature encoder library, PID library.

```
//Necessary to use rosserial with arduino DUE
#define USE_USBCON

//  Some ros and library headers
#include <ros.h>
#include <std_msgs/Int32.h>
#include <std_msgs/Float32.h>
```

```
#include <quadrature.h>
#include <geometry_msgs/Twist.h>
#include <PID_v1.h>

// Initialize PID paramaters

double Setpoint_fl, Input_fl, Output_fl;
double Setpoint_fr, Input_fr, Output_fr;
double Setpoint_bl, Input_bl, Output_bl;
double Setpoint_br, Input_br, Output_br;

double aggKp=500, aggKi=2000, aggKd=0.25;

PID myPID_fl(&Input_fl, &Output_fl, &Setpoint_fl, aggKp, aggKi, aggKd, DIRECT);
PID myPID_fr(&Input_fr, &Output_fr, &Setpoint_fr, aggKp, aggKi, aggKd, DIRECT);
PID myPID_bl(&Input_bl, &Output_bl, &Setpoint_bl, aggKp, aggKi, aggKd, DIRECT);
PID myPID_br(&Input_br, &Output_br, &Setpoint_br, aggKp, aggKi, aggKd, DIRECT);

// Initialize quadrature encoder paramaters

Quadrature_encoder<47,46> encoder_fright(Board::due);
Quadrature_encoder<48,49> encoder_fleft(Board::due);
Quadrature_encoder<42,43> encoder_bright(Board::due);
Quadrature_encoder<44,45> encoder_bleft(Board::due);

// Initialize pin numbers for the L298N driver

const uint8_t RF_PWM = 11;
const uint8_t RF_BACK = 27;
const uint8_t RF_FORW = 26;
const uint8_t LF_BACK = 24;
const uint8_t LF_FORW = 25;
const uint8_t LF_PWM = 10;

const uint8_t RB_PWM = 13;
const uint8_t RB_BACK = 31;
const uint8_t RB_FORW = 30;
const uint8_t LB_BACK = 29;
const uint8_t LB_FORW = 28;
const uint8_t LB_PWM = 12;

// used to tell if the robot is stationary and remove integral windup
bool wtf;

// Initialize ROS paramaters

ros::NodeHandle nh;

std_msgs::Int32 lfcount;
std_msgs::Int32 rfcount;
std_msgs::Int32 lbcount;
std_msgs::Int32 rbcount;

ros::Publisher lfwheel("lfwheel", &lfcount);
ros::Publisher rfwheel("rfwheel", &rfcount);
ros::Publisher lbwheel("lbwheel", &lbcount);
ros::Publisher rbwheel("rbwheel", &rbcount);

std_msgs::Float32 lfvel;
std_msgs::Float32 rfvel;
std_msgs::Float32 lbvel;
```

```
std_msgs::Float32 rbvel;


ros::Publisher lfspeed("lfspeed", &lfvel);
ros::Publisher rfspeed("rfspeed", &rfvel);
ros::Publisher lbspeed("lbspeed", &lbvel);
ros::Publisher rbspeed("rbspeed", &rbvel);



//These callbacks are used to change the PID paramaters dynamically by publishing to topics (not the best way to do it but functional)


void onKp(const std_msgs::Int32 &msg)
{
   aggKp = msg.data;
   myPID_fl.SetTunings(aggKp, aggKi, aggKd);
   myPID_fr.SetTunings(aggKp, aggKi, aggKd);
   myPID_bl.SetTunings(aggKp, aggKi, aggKd);
   myPID_br.SetTunings(aggKp, aggKi, aggKd);
}
void onKi(const std_msgs::Int32 &msg)
{
   aggKi = msg.data;
   myPID_fl.SetTunings(aggKp, aggKi, aggKd);
   myPID_fr.SetTunings(aggKp, aggKi, aggKd);
   myPID_bl.SetTunings(aggKp, aggKi, aggKd);
   myPID_br.SetTunings(aggKp, aggKi, aggKd);
}
void onKd(const std_msgs::Int32 &msg)
{
   aggKd = msg.data;
   myPID_fl.SetTunings(aggKp, aggKi, aggKd);
   myPID_fr.SetTunings(aggKp, aggKi, aggKd);
   myPID_bl.SetTunings(aggKp, aggKi, aggKd);
   myPID_br.SetTunings(aggKp, aggKi, aggKd);
}

// Cmd_vel Callback
// Sets the setpoints of the pid for each wheel


void onTwist(const geometry_msgs::Twist &msg)
{
  nh.loginfo("Inside Callback");
  float x = msg.linear.x;
  float z = msg.angular.z;
  float w = 0.4;
  if( x > 0.2) {x = 0.2;}
  if( x < -0.2){x = -0.2;}
  if( z > 0.13) {z = 0.13;}
  if(z < -0.13){z = -0.13;}


  if(x==0 && z==0){wtf = true;}
  else {wtf=false;}


  Setpoint_fr = x + (z * w / 2.0)/0.1;
  Setpoint_fl = x - (z * w / 2.0)/0.1;
  Setpoint_br = x + (z * w / 2.0)/0.1;
  Setpoint_bl = x - (z * w / 2.0)/0.1;
}


ros::Subscriber<geometry_msgs::Twist> sub("cmd_vel", &onTwist);
ros::Subscriber<std_msgs::Int32> Kp_sub("Kp_set", &onKp);
ros::Subscriber<std_msgs::Int32> Ki_sub("Ki_set", &onKi);
ros::Subscriber<std_msgs::Int32> Kd_sub("Kd_set", &onKd);

// Move any motor function with speed_pwm value and pin numbers
```

```
void Move_motor(int speed_pwm,const uint8_t pwm,const uint8_t forw,const uint8_t back)
{
 if(speed_pwm >= 0)
 {
  digitalWrite(forw, HIGH);
  digitalWrite(back, LOW);
  analogWrite(pwm, abs(speed_pwm));
 }
 else if(speed_pwm < 0)
 {
  digitalWrite(forw, LOW);
  digitalWrite(back, HIGH);
  analogWrite(pwm, abs(speed_pwm));
 }
}
```

```
// Initialize pins for forward movement

void setpins()
{
 pinMode(LF_FORW,OUTPUT);
 pinMode(LF_BACK,OUTPUT);
 pinMode(RF_FORW,OUTPUT);
 pinMode(RF_BACK,OUTPUT);
 pinMode(LF_PWM,OUTPUT);
 pinMode(RF_PWM,OUTPUT);
 pinMode(LB_FORW,OUTPUT);
 pinMode(LB_BACK,OUTPUT);
 pinMode(RB_FORW,OUTPUT);
 pinMode(RB_BACK,OUTPUT);
 pinMode(LB_PWM,OUTPUT);
 pinMode(RB_PWM,OUTPUT);
 digitalWrite(LF_FORW, HIGH);
 digitalWrite(LF_BACK, LOW);
 digitalWrite(RF_FORW, HIGH);
 digitalWrite(RF_BACK, LOW);
 digitalWrite(LB_FORW, HIGH);
 digitalWrite(LB_BACK, LOW);
 digitalWrite(RB_FORW, HIGH);
 digitalWrite(RB_BACK, LOW);
}
```

```
// Encoders tend to reverse regarding of the pins??
// This way we move the robot forward a bit on startup
// And if an encoder has negative value we reverse it.

void fix_encoder_ori_on_start(){


 analogWrite(RF_PWM, 200);
 analogWrite(LF_PWM, 200);
 analogWrite(RB_PWM, 200);
 analogWrite(LB_PWM, 200);


 delay(350);


 analogWrite(RF_PWM, 0);
 analogWrite(LF_PWM, 0);
 analogWrite(RB_PWM, 0);
 analogWrite(LB_PWM, 0);


 int ct1 = encoder_fleft.count();
 int ct2 = encoder_fright.count();
 int ct3 = encoder_bleft.count();
 int ct4 = encoder_bright.count();
 if(ct1 < 0) {encoder_fleft.reverse();}
```

```
  if(ct2 < 0) {encoder_fright.reverse();}
  if(ct3 < 0) {encoder_bleft.reverse();}
  if(ct4 < 0) {encoder_bright.reverse();}


}

//void reset Integral error when we stop
void reset_pid_Ki()
{
  myPID_fl.SetMode(MANUAL);
  myPID_fr.SetMode(MANUAL);
  myPID_bl.SetMode(MANUAL);
  myPID_br.SetMode(MANUAL);
  Output_fl=0;
  Output_fr=0;
  Output_bl=0;
  Output_br=0;


  myPID_fl.SetMode(AUTOMATIC);
  myPID_fr.SetMode(AUTOMATIC);
  myPID_bl.SetMode(AUTOMATIC);
  myPID_br.SetMode(AUTOMATIC);
}



void setup() {
  // 115200 baud rate
  nh.getHardware()->setBaud(115200);


  // Pid setup


  myPID_fl.SetOutputLimits(-255, 255);
  myPID_fr.SetOutputLimits(-255, 255);
  myPID_bl.SetOutputLimits(-255, 255);
  myPID_br.SetOutputLimits(-255, 255);


  myPID_fl.SetMode(AUTOMATIC);
  myPID_fr.SetMode(AUTOMATIC);
  myPID_bl.SetMode(AUTOMATIC);
  myPID_br.SetMode(AUTOMATIC);


  myPID_fl.SetSampleTime(25);
  myPID_fr.SetSampleTime(25);
  myPID_bl.SetSampleTime(25);
  myPID_br.SetSampleTime(25);


  // Encoder setup


  encoder_fright.begin();
  encoder_fleft.begin();
  encoder_bright.begin();
  encoder_bleft.begin();


  // setup pins and fix encoders


  setpins();
  fix_encoder_ori_on_start();




  // ros node setup
```

```
  nh.initNode();
  nh.advertise(lfwheel);
  nh.advertise(rfwheel);
  nh.advertise(lbwheel);
  nh.advertise(rbwheel);
  nh.advertise(lfspeed);
  nh.advertise(rfspeed);
  nh.advertise(lbspeed);
  nh.advertise(rbspeed);


  nh.subscribe(sub);
  nh.subscribe(Kp_sub);
  nh.subscribe(Ki_sub);
  nh.subscribe(Kd_sub);


}

// Initialize starting loop paramaters for calculating velocity and time


unsigned long prev = 0;
int old_ct1=0;
int old_ct2=0;
int old_ct3=0;
int old_ct4=0;


float ticks_per_meter = 33000.1;


void loop() {


  // count encoder ticks
  int ct1 = encoder_fleft.count();
  int ct2 = encoder_fright.count();
  int ct3 = encoder_bleft.count();
  int ct4 = encoder_bright.count();
  // for some reason if i omit this it does not work properly
  if (ct1!=-1){
    lfcount.data = ct1;}
  if (ct2!=-1){
    rfcount.data = ct2;}
  if (ct3!=-1){
    lbcount.data = ct3;}
  if (ct4!=-1){
    rbcount.data = ct4;}
  // Publish encoder ticks to calculate odom on Jetson Nano side
  lfwheel.publish(&lfcount);
  rfwheel.publish(&rfcount);
  lbwheel.publish(&lbcount);
  rbwheel.publish(&rbcount);


 // calculate time and current velocity


  unsigned long now = millis();
  Input_fl = (float(ct1 - old_ct1) / ticks_per_meter) / ((now - prev) / 1000.0);
  Input_fr = (float(ct2 - old_ct2) / ticks_per_meter) / ((now - prev) / 1000.0);
  Input_bl = (float(ct3 - old_ct3) / ticks_per_meter) / ((now - prev) / 1000.0);
  Input_br = (float(ct4 - old_ct4) / ticks_per_meter) / ((now - prev) / 1000.0);


  lfvel.data = Input_fl;
  rfvel.data = Input_fr;
  lbvel.data = Input_bl;
  rbvel.data = Input_br;


  lfspeed.publish(&lfvel);
  rfspeed.publish(&rfvel);
  lbspeed.publish(&lbvel);
  rbspeed.publish(&rbvel);
```

```
 // Compute  Pid
 myPID_fl.Compute();
 myPID_fr.Compute();
 myPID_bl.Compute();
 myPID_br.Compute();



 if(wtf){
   reset_pid_Ki();
 }

 // Move the motors with the output of the pid


 Move_motor(Output_fl,LF_PWM,LF_FORW,LF_BACK);
 Move_motor(Output_fr,RF_PWM,RF_FORW,RF_BACK);
 Move_motor(Output_bl,LB_PWM,LB_FORW,LB_BACK);
 Move_motor(Output_br,RB_PWM,RB_FORW,RB_BACK);
 // spin the ros node
   nh.spinOnce();
 // take the old encoder ticks and time for calculating velocity
 old_ct1 = encoder_fleft.count();
 old_ct2 = encoder_fright.count();
 old_ct3 = encoder_bleft.count();
 old_ct4 = encoder_bright.count();
 prev = now;
 delay(25);
}
```

When we upload this code to the arduino our robot will accept geometry_msgs/Twist commands and will use the encoders to keep the speed as constant as possible.

We also publish the encoder ticks on topics in order for another node to subscribe to these and publish the odometry of the robot in the /odom_encoder topic.
Odometry is used to calculate how far and in what direction the robot has moved from a starting point.
Please Note that odometry based on encoders only is not optimal because through slippage an accumulative error will occur, that is why we also need and external odometry information. We will see how to get that with our lidar.
Lastly we publish the current velocity of each wheel in order to find more optimal pid paramaters by using rqt_plot.

Here is a node that subscribes to the encoder ticks and publish the odometry of the robot.

```
#!/usr/bin/env python

import rospy
import roslib
#roslib.load_manifest('differential_drive')
from math import sin, cos, pi

from geometry_msgs.msg import Quaternion
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from tf.broadcaster import TransformBroadcaster
from std_msgs.msg import Int32

##############################################################################
class DiffTf:
```

```
##############################################################################

    ##############################################################################
    def __init__(self):
    ##############################################################################
        rospy.init_node("diff_tf")
        self.nodename = rospy.get_name()
        rospy.loginfo("-I- %s started" % self.nodename)


        #### parameters #######
        self.rate = rospy.get_param('~rate',10.0)  # the rate at which to publish the transform
        self.ticks_meter = float(rospy.get_param('~ticks_meter', 36000))  # The number of wheel encoder ticks per meter of travel
        self.base_width = float(rospy.get_param('~base_width', 0.25)) # The wheel base width in meters
        self.publish_tf = int(rospy.get_param('~publish_tf', 0))
        self.base_frame_id = rospy.get_param('~base_frame_id','robot_footprint') # the name of the base frame of the robot
        self.odom_frame_id = rospy.get_param('~odom_frame_id', 'odom_encoder') # the name of the odometry reference frame


        self.encoder_min = rospy.get_param('encoder_min', -2147483648)
        self.encoder_max = rospy.get_param('encoder_max', 2147483647)
        self.encoder_low_wrap = rospy.get_param('wheel_low_wrap', (self.encoder_max - self.encoder_min) * 0.3 + self.encoder_min )
        self.encoder_high_wrap = rospy.get_param('wheel_high_wrap', (self.encoder_max - self.encoder_min) * 0.7 + self.encoder_min )


        self.t_delta = rospy.Duration(1.0/self.rate)
        self.t_next = rospy.Time.now() + self.t_delta


        # internal data
        self.enc_fleft = None        # wheel encoder readings
        self.enc_fright = None
        self.enc_bleft = None        # wheel encoder readings
        self.enc_bright = None
        self.fleft = 0               # actual values coming back from robot
        self.fright = 0
        self.bleft = 0
        self.bright = 0
        self.lfmult = 0
        self.rfmult = 0
        self.lbmult = 0
        self.rbmult = 0
        self.prev_lfencoder = 0
        self.prev_rfencoder = 0
        self.prev_lbencoder = 0
        self.prev_rbencoder = 0
        self.x = 0                   # position in xy plane
        self.y = 0
        self.th = 0
        self.dx = 0                  # speeds in x/rotation
        self.dr = 0
        self.then = rospy.Time.now()


        # subscriptions
        rospy.Subscriber("lfwheel", Int32, self.lfwheelCallback)
        rospy.Subscriber("rfwheel", Int32, self.rfwheelCallback)
        rospy.Subscriber("lbwheel", Int32, self.lbwheelCallback)
        rospy.Subscriber("rbwheel", Int32, self.rbwheelCallback)
        self.odomPub = rospy.Publisher("odom_encoder", Odometry, queue_size=10)
        self.odomBroadcaster = TransformBroadcaster()

    ##############################################################################
    def spin(self):
    ##############################################################################
        r = rospy.Rate(self.rate)
        while not rospy.is_shutdown():
            self.update()
            r.sleep()



    ##############################################################################
    def update(self):
    ##############################################################################
        now = rospy.Time.now()
```

```
    if now > self.t_next:
        elapsed = now - self.then
        self.then = now
        elapsed = elapsed.to_sec()


        # calculate odometry
        if self.enc_fleft == None:
            d_left = 0
            d_right = 0
        else:
            d_left = (int((self.fleft + self.bleft)*0.5) - int((self.enc_fleft + self.enc_bleft)*0.5)) / self.ticks_meter
            d_right = (int((self.fright + self.bright)*0.5) - int((self.enc_fright + self.enc_bright)*0.5)) / self.ticks_meter
        self.enc_fleft = self.fleft
        self.enc_fright = self.fright
        self.enc_bleft = self.bleft
        self.enc_bright = self.bright


        # distance traveled is the average of the two wheels
        d = ( d_left + d_right ) / 2
        # this approximation works (in radians) for small angles
        th = ( d_right - d_left ) / self.base_width
        # calculate velocities
        self.dx = d / elapsed
        self.dr = th / elapsed



        if (d != 0):
            # calculate distance traveled in x and y
            x = cos( th ) * d
            y = -sin( th ) * d
            # calculate the final position of the robot
            self.x = self.x + ( cos( self.th ) * x - sin( self.th ) * y )
            self.y = self.y + ( sin( self.th ) * x + cos( self.th ) * y )
        if( th != 0):
            self.th = self.th + th


        # publish the odom information
        quaternion = Quaternion()
        quaternion.x = 0.0
        quaternion.y = 0.0
        quaternion.z = sin( self.th / 2 )
        quaternion.w = cos( self.th / 2 )
        if (self.publish_tf == 1):
            self.odomBroadcaster.sendTransform(
                (self.x, self.y, 0),
                (quaternion.x, quaternion.y, quaternion.z, quaternion.w),
                rospy.Time.now(),
                self.base_frame_id,
                self.odom_frame_id
            )


        odom = Odometry()
        odom.header.stamp = now
        odom.header.frame_id = self.odom_frame_id
        odom.pose.pose.position.x = self.x
        odom.pose.pose.position.y = self.y
        odom.pose.pose.position.z = 0
        odom.pose.pose.orientation = quaternion
        odom.child_frame_id = self.base_frame_id
        odom.twist.twist.linear.x = self.dx
        odom.twist.twist.linear.y = 0
        odom.twist.twist.angular.z = self.dr
        self.odomPub.publish(odom)



##################################################################################
def lfwheelCallback(self, msg):
```

```
    ############################################################################
        enc = msg.data
        if (enc < self.encoder_low_wrap and self.prev_lfencoder > self.encoder_high_wrap):
            self.lfmult = self.lfmult + 1


        if (enc > self.encoder_high_wrap and self.prev_lfencoder < self.encoder_low_wrap):
            self.lfmult = self.lfmult - 1


        self.fleft = 1.0 * (enc + self.lfmult * (self.encoder_max - self.encoder_min))
        self.prev_lfencoder = enc


    ############################################################################
    def rfwheelCallback(self, msg):
    ############################################################################
        enc = msg.data
        if(enc < self.encoder_low_wrap and self.prev_rfencoder > self.encoder_high_wrap):
            self.rfmult = self.rfmult + 1


        if(enc > self.encoder_high_wrap and self.prev_rfencoder < self.encoder_low_wrap):
            self.rfmult = self.rfmult - 1


        self.fright = 1.0 * (enc + self.rfmult * (self.encoder_max - self.encoder_min))
        self.prev_rfencoder = enc


    ############################################################################
    def lbwheelCallback(self, msg):
    ############################################################################
        enc = msg.data
        if (enc < self.encoder_low_wrap and self.prev_lbencoder > self.encoder_high_wrap):
            self.lbmult = self.lbmult + 1


        if (enc > self.encoder_high_wrap and self.prev_lbencoder < self.encoder_low_wrap):
            self.lbmult = self.lbmult - 1


        self.bleft = 1.0 * (enc + self.lbmult * (self.encoder_max - self.encoder_min))
        self.prev_lbencoder = enc


    ############################################################################
    def rbwheelCallback(self, msg):
    ############################################################################
        enc = msg.data
        if(enc < self.encoder_low_wrap and self.prev_rbencoder > self.encoder_high_wrap):
            self.rbmult = self.rbmult + 1


        if(enc > self.encoder_high_wrap and self.prev_rbencoder < self.encoder_low_wrap):
            self.rbmult = self.rbmult - 1


        self.bright = 1.0 * (enc + self.rbmult * (self.encoder_max - self.encoder_min))
        self.prev_rbencoder = enc
################################################################################
################################################################################
if __name__ == '__main__':
    """ main """
    try:
        diffTf = DiffTf()
        diffTf.spin()
    except rospy.ROSInterruptException:
        pass
```

Now we have integrated our mobile base in the ROS framework and we are ready to explore packages for localization, mapping and path planning!!!!

3. Monte Carlo localization

3.1 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 probabillity 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.

3.2 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 it's 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 it's 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.

3.3 Monte Carlo Localization

As a roboticist, you will certainly be interested in mapping your home with the help of your robot. Then you'll want to operate a robot inside the mapped environment by keeping track of it's position and orientation.

To do so you will have a wide range of localization algorithms you could implement ranging from Extended Kalman Filter, to Markov, to Grid and finally Monte Carlo Localization (MCL).

The MCL is the most popular localization algorithm in robotics, and for that reason is the algorithm we will use to keep track of the robot;s pose.

Now our robot will be navigating inside the known map and collect sensory information using range finder sensors, in our a case a 2D Lidar.

Many scientists **refer to MCL as particle filter localization algorithm** since it uses particles to localize the robot, **you can think of particles as a virtual element that resemples the robot.** Each **particle has a position and orientation** and represents a guess as to where your robot might be located at.

Please note here that MCL is capable of solving only the Local and Global localization problem.

POWER OF MCL

With the EKF algorithm (that we don't explain here) you can estimate the pose of almost any robot with accurate on-board sensors. So why use another algorithm?

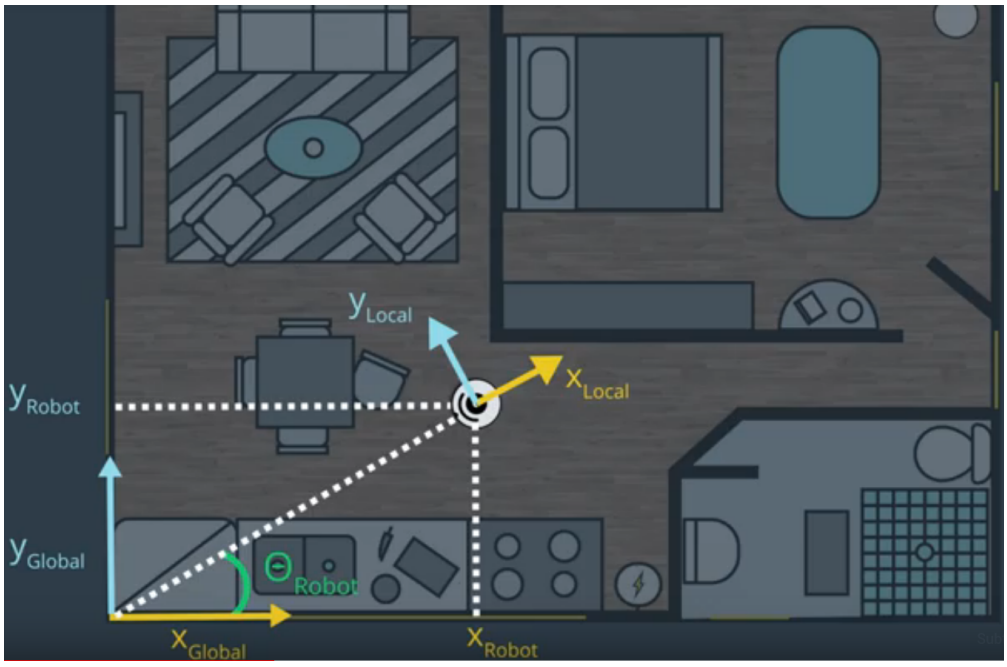
MCL advantages over EKF

As you know MCL uses particles to localize the robot's pose meaning that it can **approximate almost any state space distribution.**

Furthermore **MCL is easy to program** , it can represent non-Gaussian distributions and can approximate any other practical important distribution, this means that MCL is unrestricted by linear Gaussian states-based assumption as is the case with 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.
Lastly 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.**

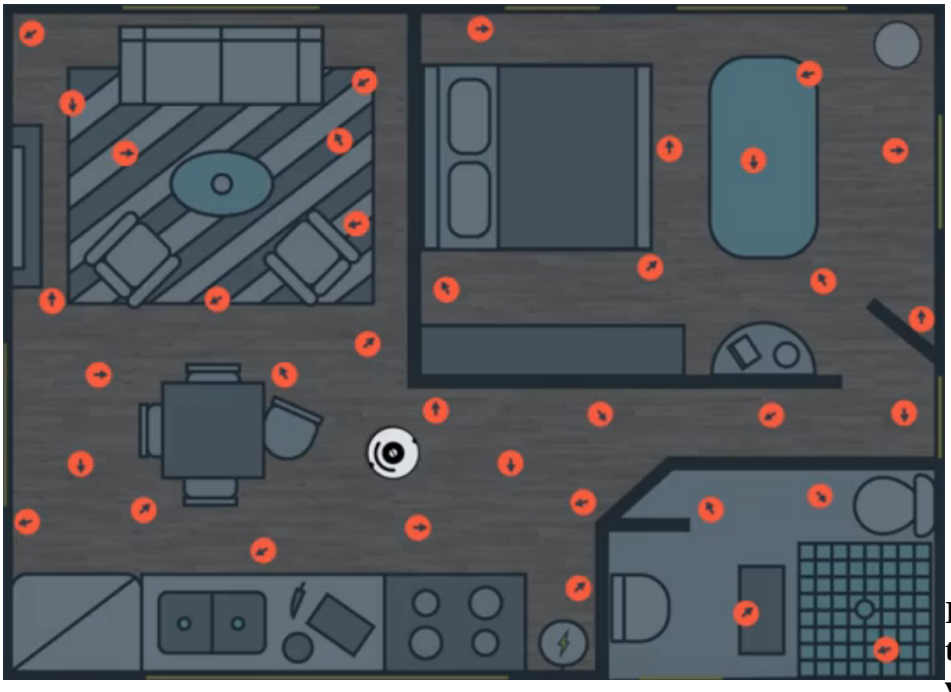
	MCL	EKF
Measurements	Raw Measurements	Landmarks
Measurement Noise	Any	Gaussian
Posterior	Particles	Gaussian
Efficiency(memory)	✓	✓✓
Efficiency(time)	✓	✓✓
Ease of Implementation	✓✓	✓
Resolution	✓	✓✓
Robustness	✓✓	x
Memory & Resolution Control	Yes	No
Global Localization	Yes	No
State Space	Multimodel Discrete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

Particle Filter

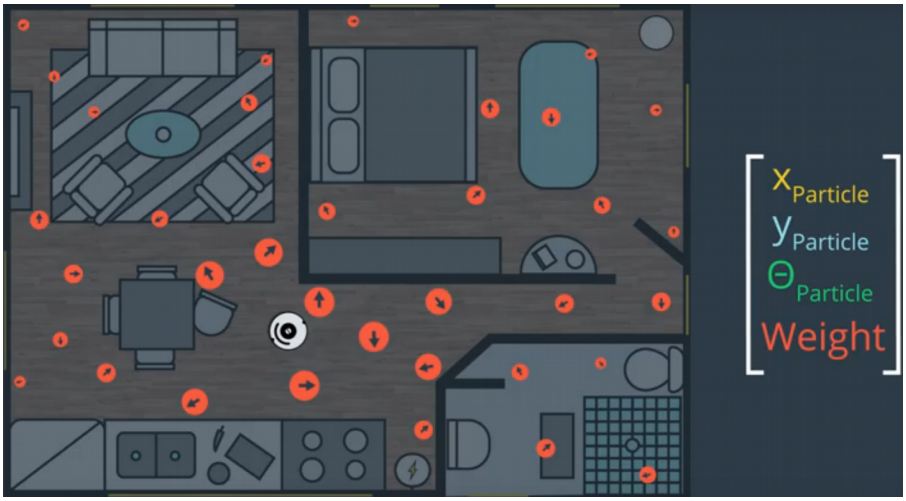


In a 2d map the robot has no idea where it's located at, since the initial state is unknown, the robot is trying to estimate its pose by solving the global localization problem. The robot has on-board range sensors which permits it to sense obstacles, walls and ultimately determine where it's located (current robot pose x,y and orientation θ).

With the MCL the particles are initially spread randomly and uniformly throughout the entire map. These particles do not physically exist and are just shown in simulation.



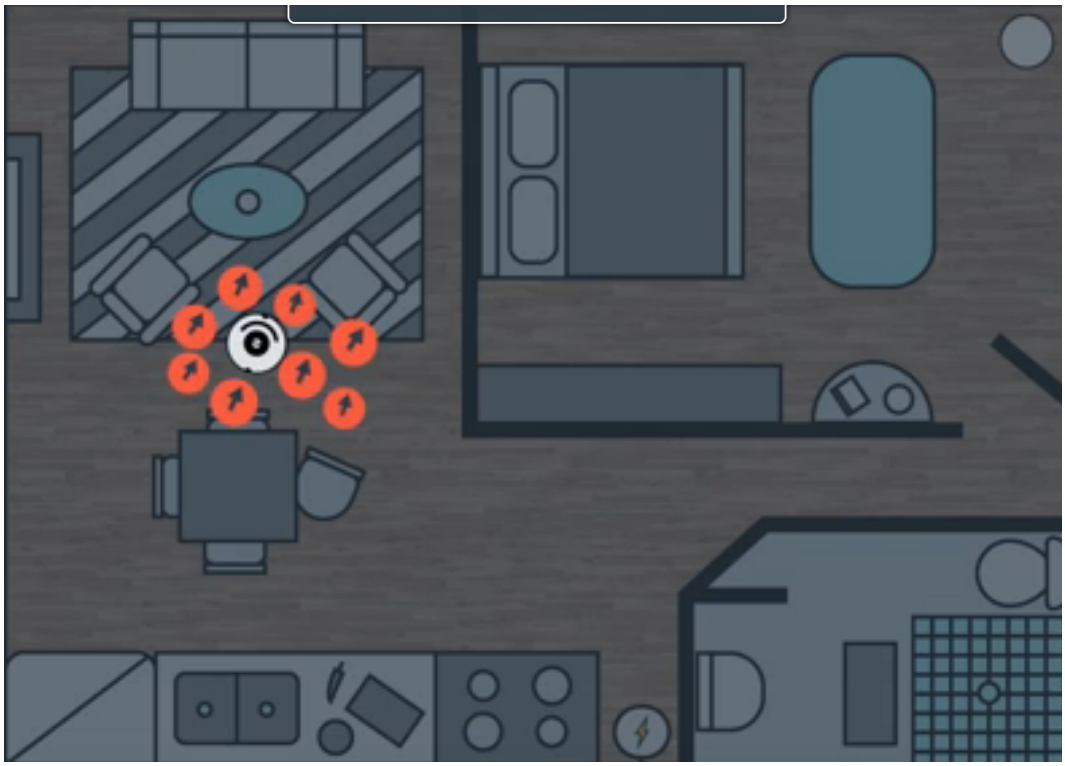
In addition to a 3-D vector particles are assigned a weight. The weight of a particle is the difference bewteen the robot's actual pose and the particle predicted pose. The importance of a particle depends on it's weight and the bigger the particle the more accurate it is.
Particles with large weights are more likely to survive during the resampling process.



For example P2 will be twice more likely to survive than P1 since it has larger weight and thus a larger probabillity of survival.

After the resampling process, particles with significant weights are more likely to survive whereas the others are more likely to die.

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



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(Xt)=P(Xt|Z1...t) Where,

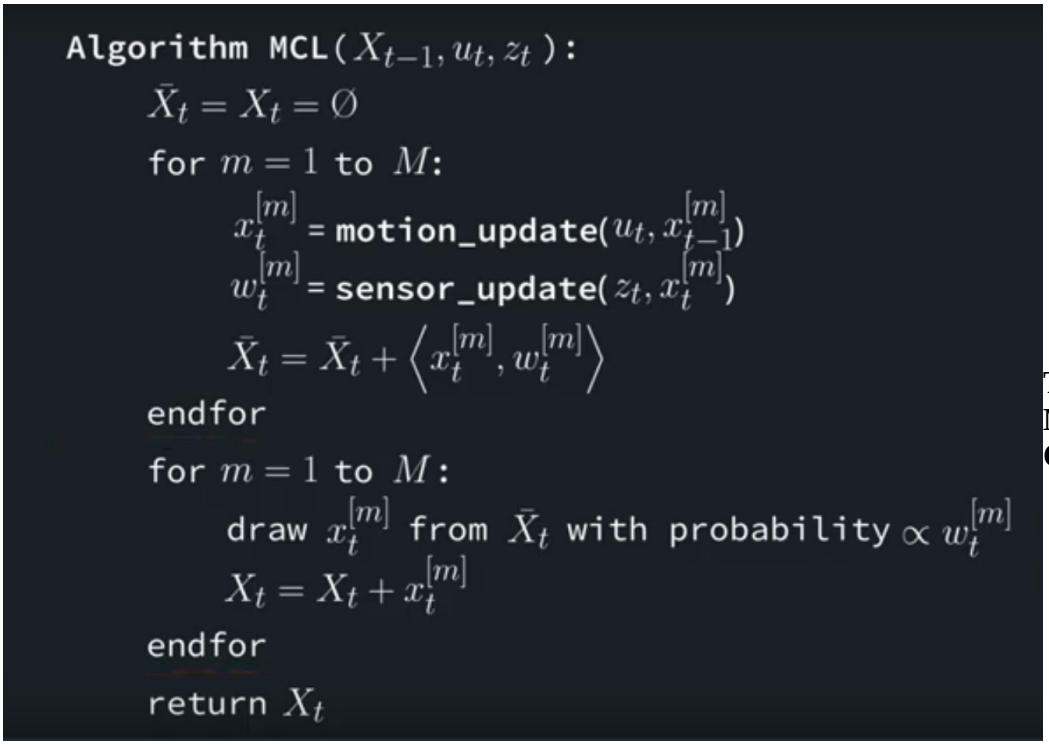
- $\mathbf{X_t}$: State at time t
- $\mathbf{Z_{1...t}}$: Measurements from time 1 up to time t

<u>Probabillity</u>

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

P(A|B)=P(B|A)∗P(A) / P(B)=P(B|A)∗P(A) / (P(A)∗P(B|A)+P(¬A)∗P(B|¬A))

MCL the algorithm



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 Xt-1** the **Actuation Command Ut** and the **Sensor Measurements Zt,** initially the belief is obtained by randomly generating #n particles, then in the first for loop, the hypothetical state is computed whenever the robot moves **(motion_update)** following the **particles weight is computed using the latest sensor measurement (sensor_update).** Now motion and measurement are both added to the previous state **Xt.**

Moving on to the **second section of the MCL where resampling process happens,** here the **particles with high probabillity survive and are re-drawn in the next iteration while the others die.**
Finally the algorithms **outputs the new belief and another cycle starts.**

3.4 Using the Adaptive Monte Carlo Localization Package

3.4.1 URDF of the robot

First of all we must create an accurate description of our robot using URDF.
The robot at this stage consists of a chassis, 4 wheels, a lidar base and the 2-D lidar.

Let's see the URDF

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

 <robot name="lynx_rover" xmlns:xacro="http://www.ros.org/wiki/xacro">


   <xacro:include filename="$(find lynxbot_description)/urdf/common_properties.urdf.xacro" />
   <xacro:include filename="$(find lynxbot_description)/urdf/sensors/ydlidar.urdf.xacro" />
   <xacro:include filename="$(find lynxbot_description)/urdf/wheels.urdf.xacro" />
   <xacro:include filename="$(find lynxbot_description)/urdf/sensor_bases.urdf.xacro" />

   <!--xacro:include filename="$(find lynxbot_description)/urdf/lynx_rover.urdf.gazebo" /-->

   <xacro:property name="M_PI" value="3.141592653" />
   <xacro:property name="Length_base" value="0.255" />
   <xacro:property name="Width_base" value="0.2" />
   <xacro:property name="Height_base" value="0.06" />


   <link name="base_link"></link>

   <joint name="base_link_joint" type="fixed">
     <origin xyz="0 0 0" rpy="0 0 0" />
     <parent link="base_link"/>
     <child link="chassis" />
   </joint>

   <link name='chassis'>
     <pose>0 0 0 0 0 0</pose>

     <inertial>
       <mass value="8.0"/>
       <origin xyz="0.0 0 0" rpy=" 0 0 0"/>
```

```
      <inertia
        ixx="0.1" ixy="0" ixz="0"
        iyy="0.1" iyz="0"
        izz="0.1"
      />
    </inertial>

    <collision name='collision'>
      <origin xyz="0 0 0" rpy=" 0 0 0"/>
      <geometry>
        <box size = "${Length_base} ${Width_base} ${Height_base}"/>
      </geometry>
    </collision>

    <visual name='chassis_visual'>
      <origin xyz="0 0 0" rpy=" 0 0 0"/>
      <geometry>
        <box size = "${Length_base} ${Width_base} ${Height_base}"/>
      </geometry>
      <material name="green">
        <color rgba="0 1 0 1"/>
      </material>
    </visual>

  </link>

  <xacro:property name="wheel_joint_origin_x" value="0.0975" />
  <xacro:property name="wheel_joint_origin_y" value="0.14" />
  <xacro:property name="wheel_joint_origin_z" value="-0.015" />

  <wheel_link prefix="front_left"
            origin_xyz="0 0 0"
            origin_rpy="${M_PI/2} 0 0" />
  <wheel_link prefix="back_left"
            origin_xyz="0 0 0"
            origin_rpy="${M_PI/2} 0 0" />
  <wheel_link prefix="front_right"
            origin_xyz="0 0 0"
            origin_rpy="${M_PI/2} 0 0" />
  <wheel_link prefix="back_right"
            origin_xyz="0 0 0"
            origin_rpy="${M_PI/2} 0 0" />

  <wheel_joint prefix="front_left"
             origin_xyz="${wheel_joint_origin_x} ${wheel_joint_origin_y} ${wheel_joint_origin_z}"
             origin_rpy="0 0 0" />

  <wheel_joint prefix="back_left"
             origin_xyz="${-wheel_joint_origin_x} ${wheel_joint_origin_y} ${wheel_joint_origin_z}"
             origin_rpy="0 0 0" />

  <wheel_joint prefix="front_right"
             origin_xyz="${wheel_joint_origin_x} ${-wheel_joint_origin_y} ${wheel_joint_origin_z}"
             origin_rpy="0 0 0" />
  <wheel_joint prefix="back_right"
             origin_xyz="${-wheel_joint_origin_x} ${-wheel_joint_origin_y} ${wheel_joint_origin_z}"
             origin_rpy="0 0 0" />

  <lidar_base />

</robot>
```

In this xacro we define the structure of our robot, for that we also include additional xacro files (**common_properties, ydlidar, wheels and sensor_bases)**

if we run a simulation it is necessary to also include the lynx_rover.urdf.gazebo file.

we use macros for the wheels link and joints with paramaters (prefix and origin_xyz and origin_rpy). Prefix is used to name the wheels.

Here we can see the definition of that macro.

Wheels.urdf.xacro

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

 <robot name="lynx_rover" xmlns:xacro="http://www.ros.org/wiki/xacro">

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

   <xacro:macro name="wheel_link" params="prefix origin_xyz origin_rpy">

     <link name="${prefix}_wheel">

       <inertial>
         <mass value="5"/>
         <origin xyz="${origin_xyz}" rpy="${origin_rpy}"/>
         <inertia
           ixx="0.1" ixy="0" ixz="0"
           iyy="0.1" iyz="0"
           izz="0.1"
           />
        </inertial>
       <visual>
         <geometry>
           <cylinder length="0.065" radius="0.055"/>
         </geometry>
         <origin xyz="${origin_xyz}" rpy="${origin_rpy}"/>
         <material name="blue">
           <color rgba="0 0 1 1"/>
         </material>
       </visual>
       <collision>
         <geometry>
           <cylinder length="0.065" radius="0.055"/>
         </geometry>
         <origin xyz="${origin_xyz}" rpy="${origin_rpy}"/>
       </collision>
     </link>
   </xacro:macro>

<xacro:macro name="wheel_joint" params="prefix origin_xyz origin_rpy">
     <joint type="continuous" name="${prefix}_wheel_hinge">
     <origin xyz="${origin_xyz}" rpy="${origin_rpy}"/>
     <child link="${prefix}_wheel"/>
     <parent link="chassis"/>
     <axis xyz="0 1 0" rpy="0 0 0"/>
     <limit effort="10000" velocity="1000"/>
```

```
    <joint_properties damping="1.0" friction="1.0"/>
  </joint>
  </xacro:macro>

 </robot>
```

ydlidar.urdf.xacro

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

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

  <!-- ydlidar Link -->

  <link name="laser_frame">

    <inertial>
      <mass value="0.1"/>
      <origin xyz="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://lynxbot_description/meshes/ydlidar.dae"/>
      </geometry>
      <origin xyz="0 0 0" rpy="0 0 0"/>
    </visual>
    <collision>
      <geometry>
        <mesh filename="package://lynxbot_description/meshes/ydlidar.dae"/>
      </geometry>
      <origin xyz="0 0 0" rpy="0 0 0"/>
    </collision>
  </link>

  <joint type="fixed" name="ydlidar_joint">
    <origin xyz="0 0 0.081" rpy="0 0 ${ydlidar_tilt}"/>
    <child link="laser_frame"/>
    <parent link="lidar_base"/>
  </joint>

 </robot>
```

and here is the macro for the ydlidar, we can see here that we include a mesh for the collision and visual elements.

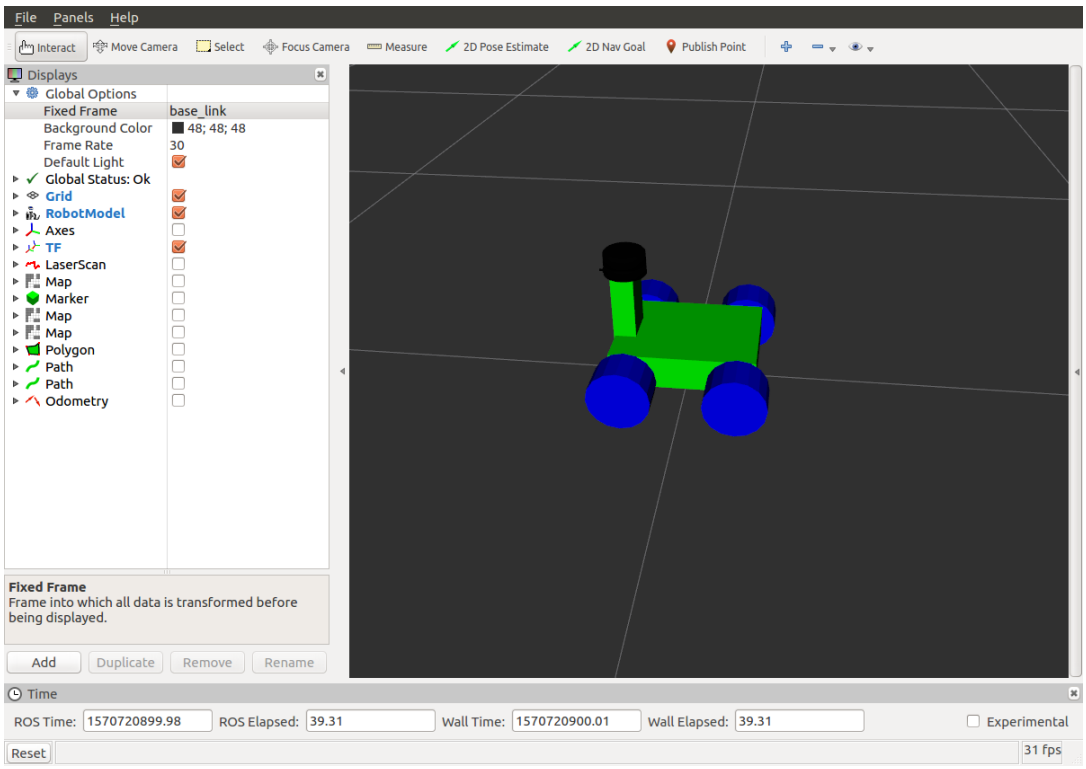
The most important thing here is to have accurate placement of the wheels and the lidar relative to the robot center, otherwise we will have suboptimal behaviour. For example the lidar will take measurements but we need to accurately transform these measurements relative to the center of the robot.

Now by launching the robot_description.launch launch file and rviz after adding a RobotModel display we can see our robot.

robot_description.launch

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

   <!-- send urdf to param server -->
   <param name="robot_description" command="$(find xacro)/xacro '$(find
lynxbot_description)/urdf/lynx_rover.urdf.xacro'" />


   <!-- Send fake 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="5.0" />
    </node>

 </launch>
```

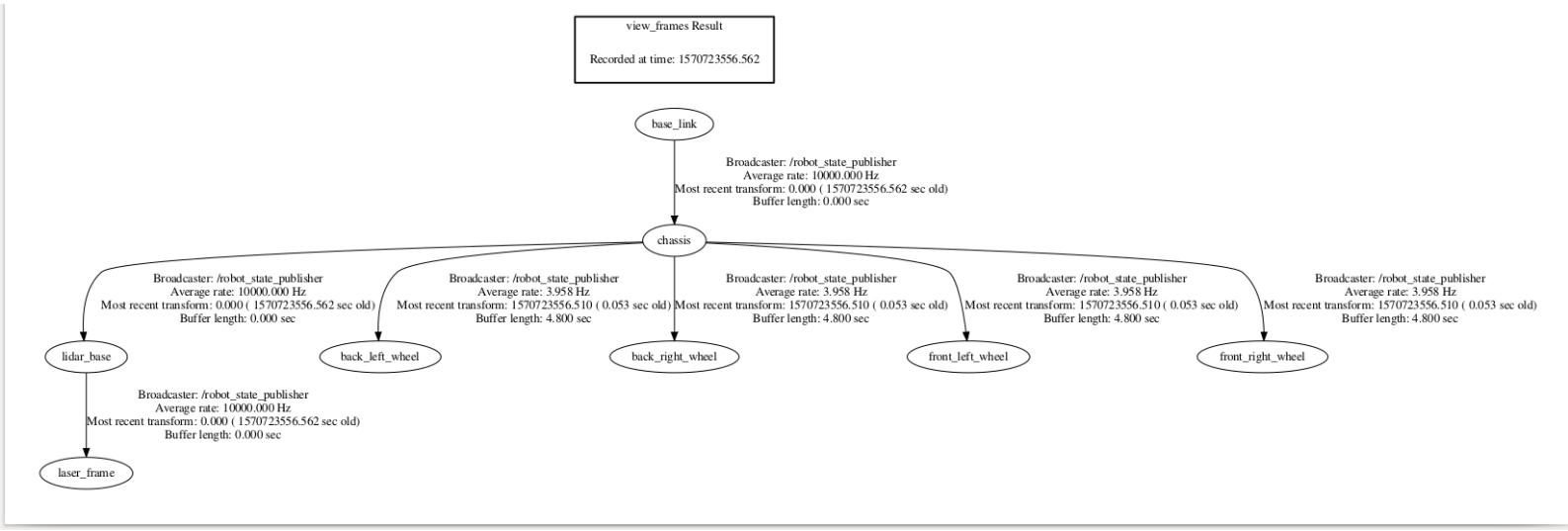
$rosrun rviz rviz



Now we have succesfully loaded the URDF in the paramater server, alongside with the Robot_state_publisher and Joint_state_publisher.

If we run the command

$rosrun tf view_frames



we get the structure of our robot

3.4.2 Main Launch file

```
<?xml version="1.0"?>
 <launch>
   <include file="$(find lynxbot_bringup)/launch/robot_description.launch"/>

   <node name="ydlidar_node"  pkg="ydlidar"  type="ydlidar_node" output="screen"
respawn="false" >
     <rosparam file="$(find lynxbot_bringup)/config/ydlidar_params.yaml" command="load" />
   </node>

   <node pkg="rf2o_laser_odometry" type="rf2o_laser_odometry_node"
name="rf2o_laser_odometry" output="screen">
     <rosparam file="$(find lynxbot_bringup)/config/rf2o_laser_odometry_params.yaml"
command="load" />
   </node>

   <node name="serial_node" pkg="rosserial_python" type="serial_node.py" output="screen">
      <rosparam file="$(find lynxbot_bringup)/config/serial_params.yaml" command="load" />
   </node>

 </launch>
```

In this main launch file

First we launch the previous robot_description.launch file

After that we launch the node responsible for the **2D -lidar (ydlidar_node)** that will start the lidar and provide us with the /scan topic.

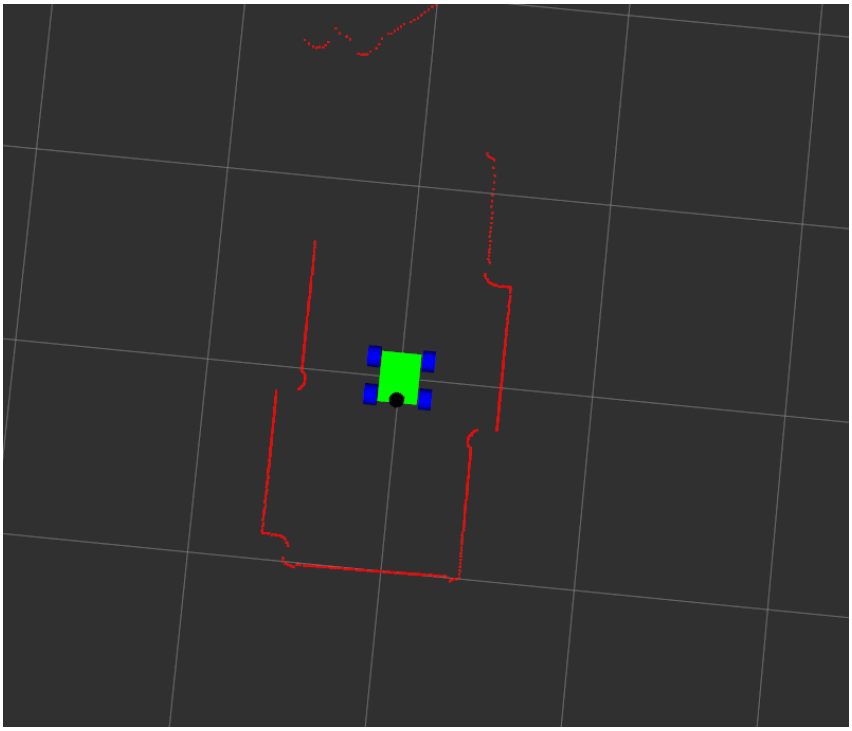
ydlidar_params.yaml

```
port: /dev/ydlidar
baudrate: 115200
frame_id: laser_frame  #the link that the scans are relative to.
low_exposure: false
resolution_fixed: true
auto_reconnect: true
reversion: false
angle_min: -180   # used to remove an area on the scan to avoid self collision
angle_max: 180
range_min: 0.1
range_max: 10.0
ignore_array: ""
samp_rate: 9
frequency: 6
```

Sanity check:

It is important to have an accurate position for the lidar. Since i double taped the lidar to the robot, it was hard for me to get an accurate xacro. Errors especially in the rotation of the lidar can have a big impact on the performance.
For that purpose i found a location like this, and tried to have the robot parrallel to the walls.

Then after fiddling with the xacro (pitch of the lidar) i get a scan that look like this (Need to add the LaserScan display in rviz and set the topic to /scan)



Next we launch the rf2o_laser_odometry node.

It is used to provide us with an accurate odometry, based on external measurements of the scan. I prefer to use this odometry versus the blind odometry of the encoders which can be quite error prone, especially for 4 wheeled vehicles and in the rotational movement. However it is much lower Hz (~6) limited by the lidar frequency. A sensor fusion with EKF using the robot_localize package is possible here in order to increase the Hz of the odometry.

rf20_laser_odometry package : (from ROS wiki)

Estimation of 2D odometry based on planar laser scans. Useful for mobile robots with innacurate 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 covariance of the estimate is employed to handle uncertainty in unconstraint scenarios (e.g. corridors).

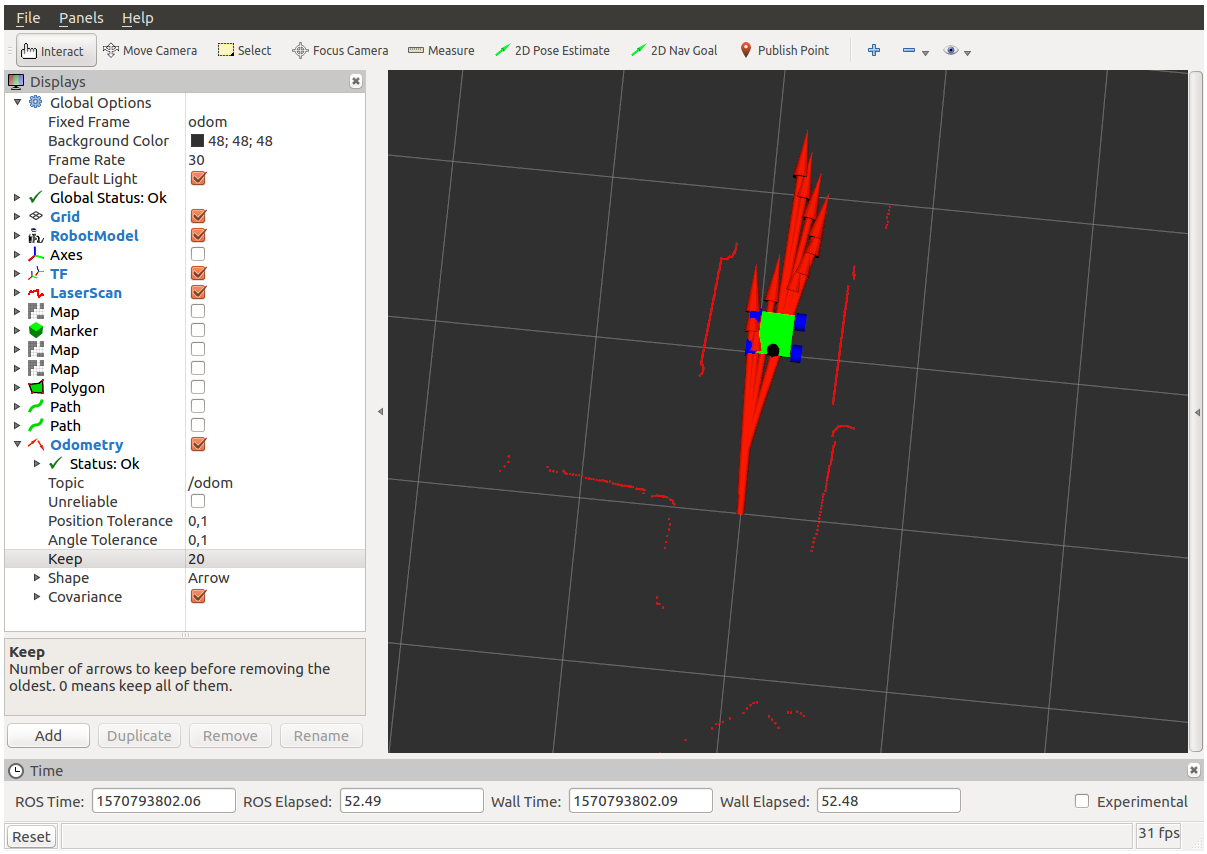
rf20_laser_odometry_params.yaml

```
laser_scan_topic: scan        # topic where the lidar scans are being published
odom_topic: /odom_rf2o            # topic where tu publish the odometry estimations
publish_tf: false                 # wheter or not to publish the tf::transform (base->odom)
```

```
base_frame_id: robot_footprint    # A tf transform from the laser_frame to the base_frame is mandatory
odom_frame_id: odom_rf2o          # frame_id (tf) to publish the odometry estimations
init_pose_from_topic: "" # (Odom topic) Leave empty to start at point (0,0)
freq: 7.5                 # Execution frequency.
verbose: false
```

By setting in the RVIZ under Global options -> Fixed frame the odom frame (previously we had the base_link frame) and adding the Odometry display setting /odom topic with keep variable 20.

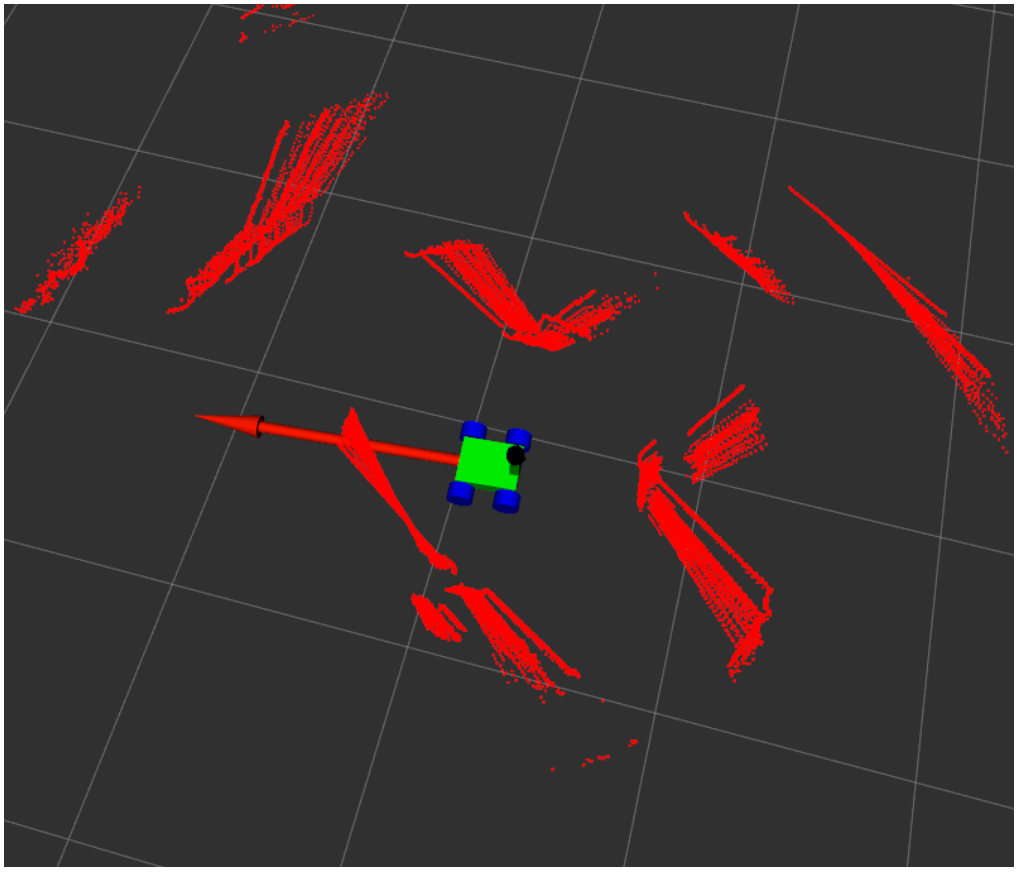
Now by moving the robot a little with the teleop package we can visualize the odometry representted by the arrows.



Sanity check :

A quite usefull 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 behaviour is traced back to bad odometry.**

This picture depicts an average odometry, i think my xacro here is not very accurate since i had modified hastily the robot.



serial_node

Last we start the serial_node of the rosserial_python package to start the pseudo node inside the arduino due. The code in the arduino acts as our hardware interface subscribing to the cmd_vel topic and publishing the encoder ticks for odometry, while moving the wheels with the speed specified by the cmd_vel topic using pid controller.

serial_params.yaml

port: /dev/ttyACM0
baud: 115200

Now we have launched all the prerequisites for launching the Adaptive Monte Carlo Package.

3.4.3 Adaptive Monte Carlo Localization package

amcl.launch

```
<?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)" />

  <!-- Place map frame at odometry frame -->
  <!--node pkg="tf" type="static_transform_publisher" name="map_odom_broadcaster"
     args="0 0 0 0 0 0 map odom 100"/-->

  <!-- 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 launh 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 capabillity 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.

Paramaters

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 # maximumm 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
```

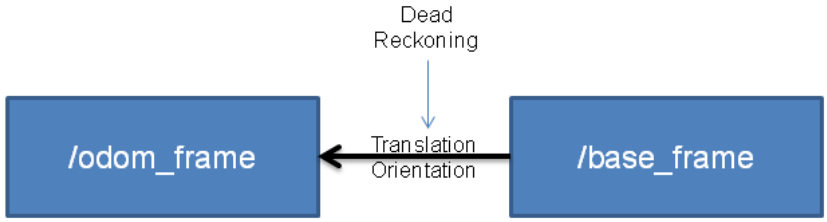
There are paramaters listed in the amcl package about tuning the laser scanner model (measurement) and odometry model (motion).

To improve the localization of our robot, we increased **laser_z_hit** and **laser_sigma_hit** to incorparate higher measurement noise.
For the odometry model, moderate values where 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 upodate_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.

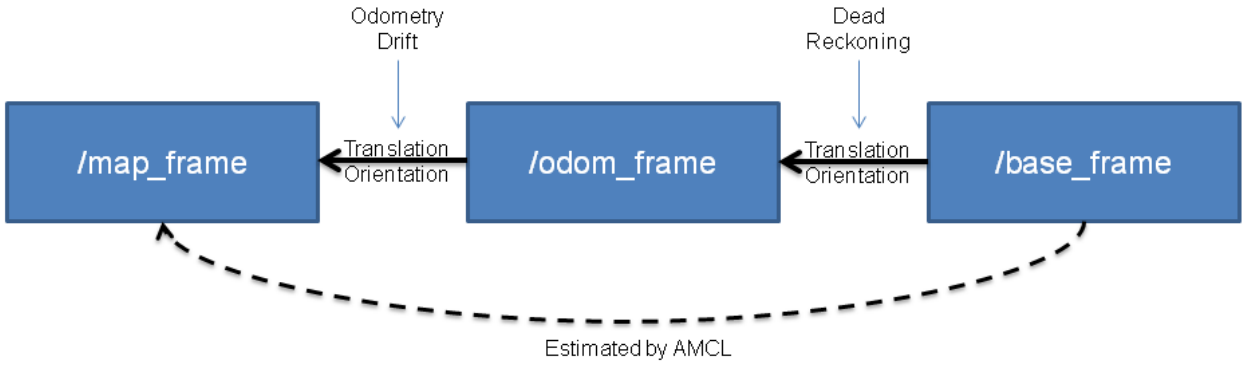
Remember you need an accurate transform bewteen the lidar and the center of the robot.

Most of the paramaters however were left as default, for a complete list please refer to the Amcl-ROS wiki page.

Odometry Localization



AMCL Map Localization

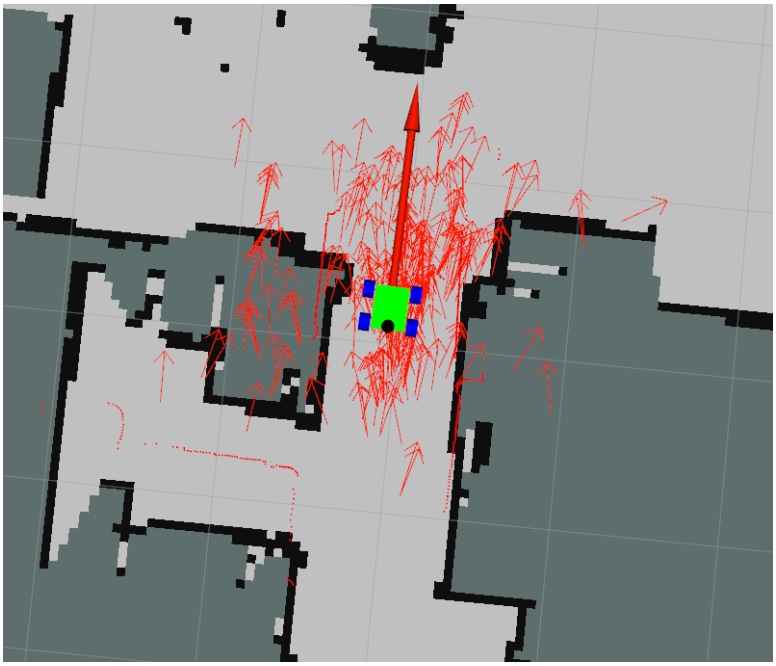


Now let’s see the results,

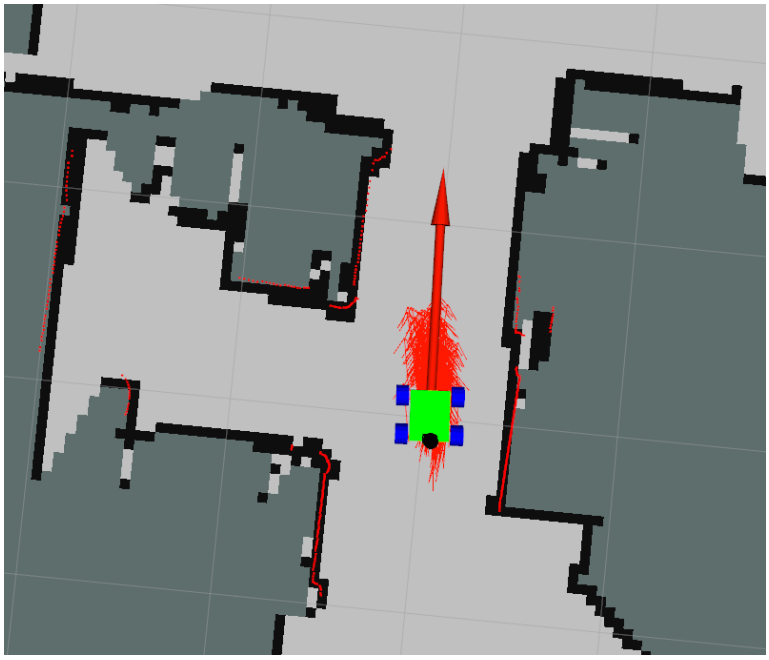
After launching the main launch file, we launch the amcl node
We set the Fixed frame to map, and add a PoseArray display and set it to the /particlecloud topic

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.

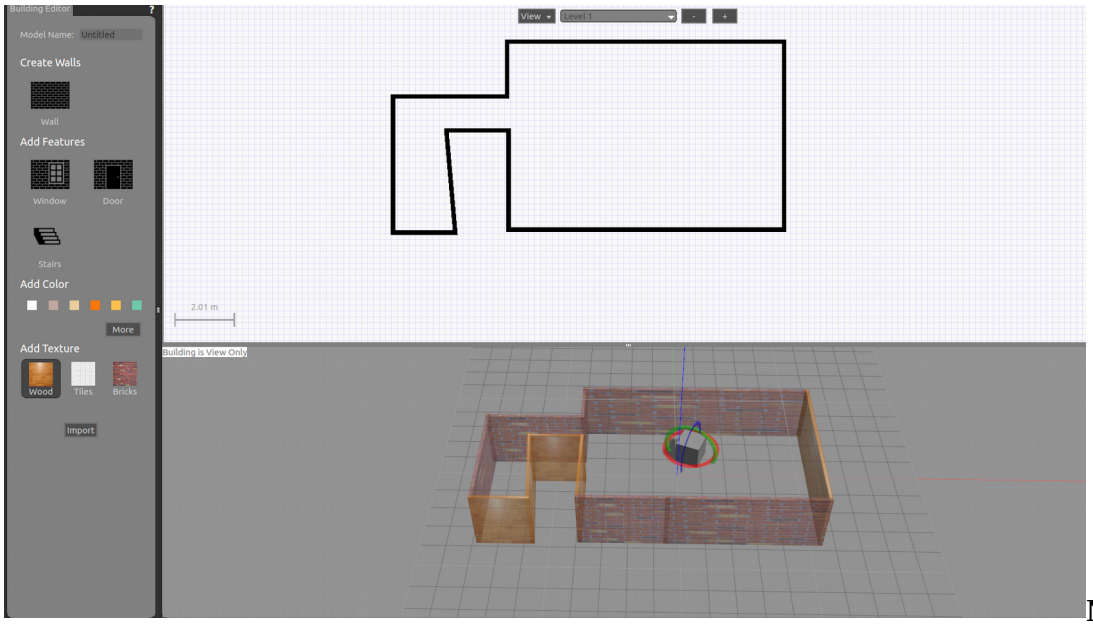


Great now we have a robot that can localize it's self inside a previous known map, and we are in a position to make it move autonomous inside the map while also avoiding dynamic obstacles, using the move_base node that we will see in the next chapter.

Before that let's see how we can simulate our robot in gazebo if you don't have a real robot.

3.4.4 Simulating the robot in Gazebo

First of all using the building editor that gazebo provides we can very easy build a simple world were our robot will reside.



Next we need to add to our xacro the lynx_rover.urdf.gazebo file where the plugins for driving the robot and the sensors are defined.

lynx_rover.urdf.gazebo

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

   <!-- ros_control plugin -->
   <gazebo>
     <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
     </plugin>
   </gazebo>



         <gazebo>
           <plugin name="skid_steer_drive_controller"
filename="libgazebo_ros_skid_steer_drive.so">
             <updateRate>100.0</updateRate>
```

```
            <robotNamespace>/</robotNamespace>
            <leftFrontJoint>front_left_wheel_hinge</leftFrontJoint>
            <rightFrontJoint>front_right_wheel_hinge</rightFrontJoint>
            <leftRearJoint>back_left_wheel_hinge</leftRearJoint>
            <rightRearJoint>back_right_wheel_hinge</rightRearJoint>
            <wheelSeparation>0.28</wheelSeparation>
            <wheelDiameter>0.11</wheelDiameter>
            <robotBaseFrame>robot_footprint</robotBaseFrame>
            <torque>10</torque>
            <topicName>cmd_vel</topicName>
            <broadcastTF>true</broadcastTF>
          </plugin>
        </gazebo>

<gazebo reference="laser_frame">
  <sensor type="gpu_ray" name="head_ydlidar_sensor">
    <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>0.30</min>
        <max>3.0</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>0.01</stddev>
      </noise>
    </ray>
    <plugin name="gazebo_ros_head_ydlidar_controller" filename="libgazebo_ros_gpu_laser.so">
      <topicName>scan</topicName>
      <frameName>laser_frame</frameName>
    </plugin>
  </sensor>
</gazebo>

</robot>
```

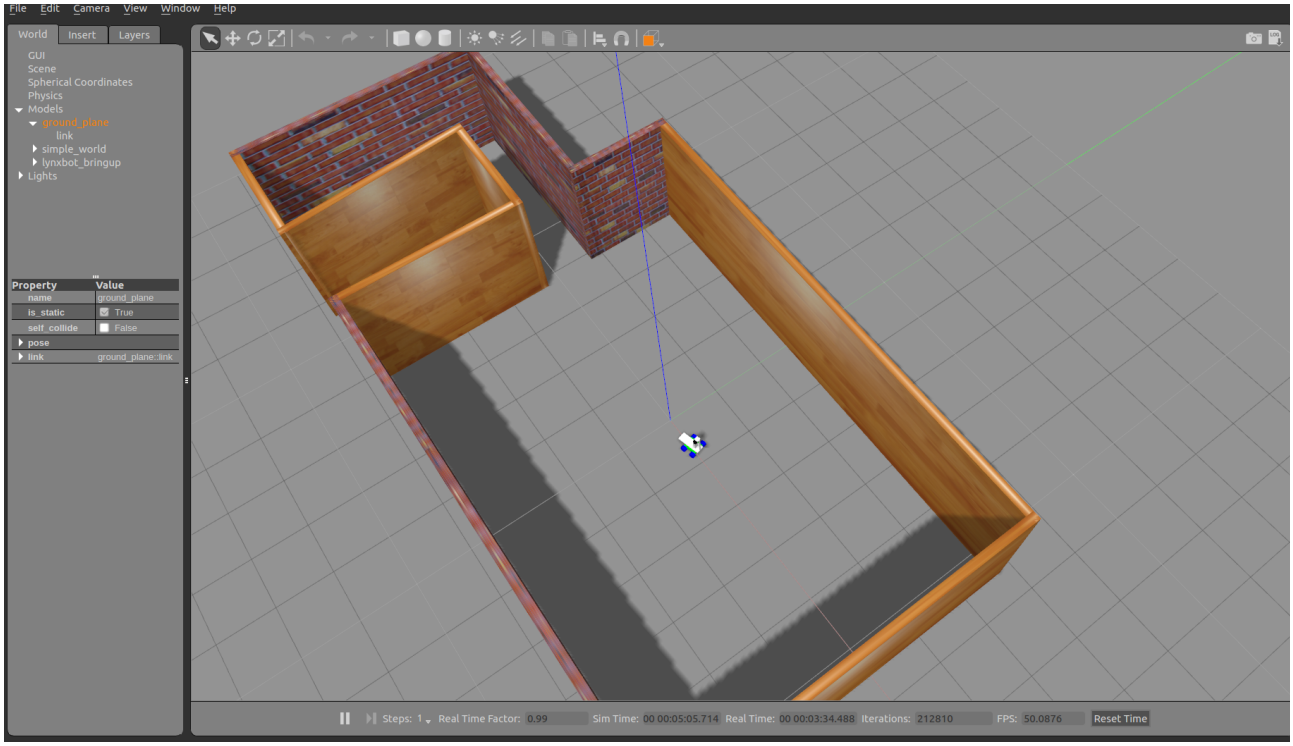
these plugins are used to simulate the laser scan, and a skid_steer_drive_controller for the simulated robot, now as with the real robot we can open up rviz and see the laser scan data and drive the robot with a cmd_vel topic.

In order to launch the robot in the simulated gazebo world we can use this launch file.

world.launch

```
<?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>
```

were as usual we launch the robot_description.launch file, then we load a gazebo instance with our created world and we spawn the urdf of our robot into gazebo using the urdf_spawner package.



4. The Navigation Stack and Move Base

4.1 The navigation stack in ROS

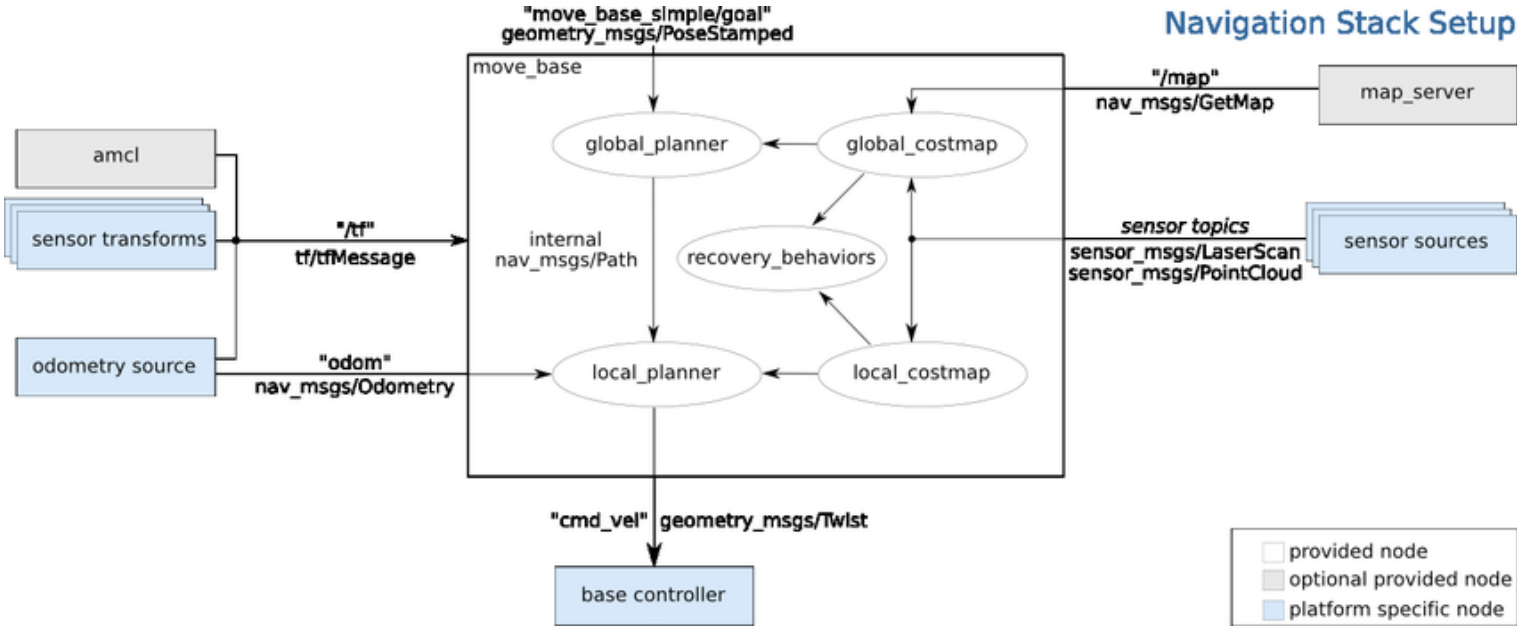
In order to understand the navigation stack, you should think of it as a set of algorithms that use the sensors of the robot and the odometry so that you can control the robot using a standard message. It can move your robot without any problems, such as crashing, getting stuck in a location, or getting 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.** Furthmore we need the **sensor topics** in our case the **/scan topic, alongside a Map and odometry that can be provided by amcl.**

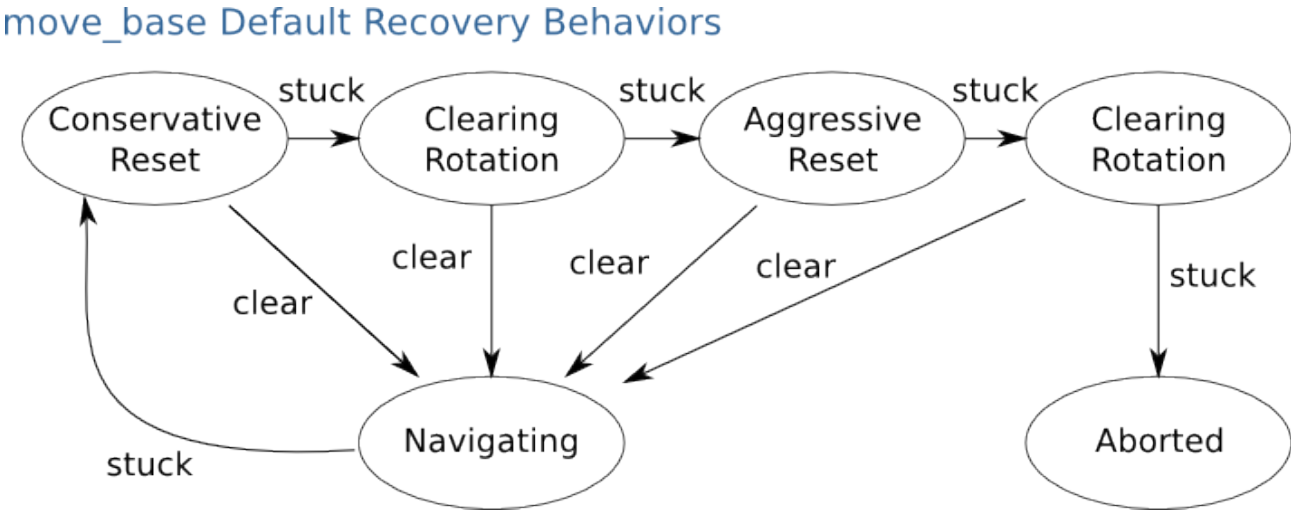
Up until now we have seen how to load and configure all of these prerequisities except the move_base node which will enable our robot to move to a specific location in the map completely autonomous.

4.2 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

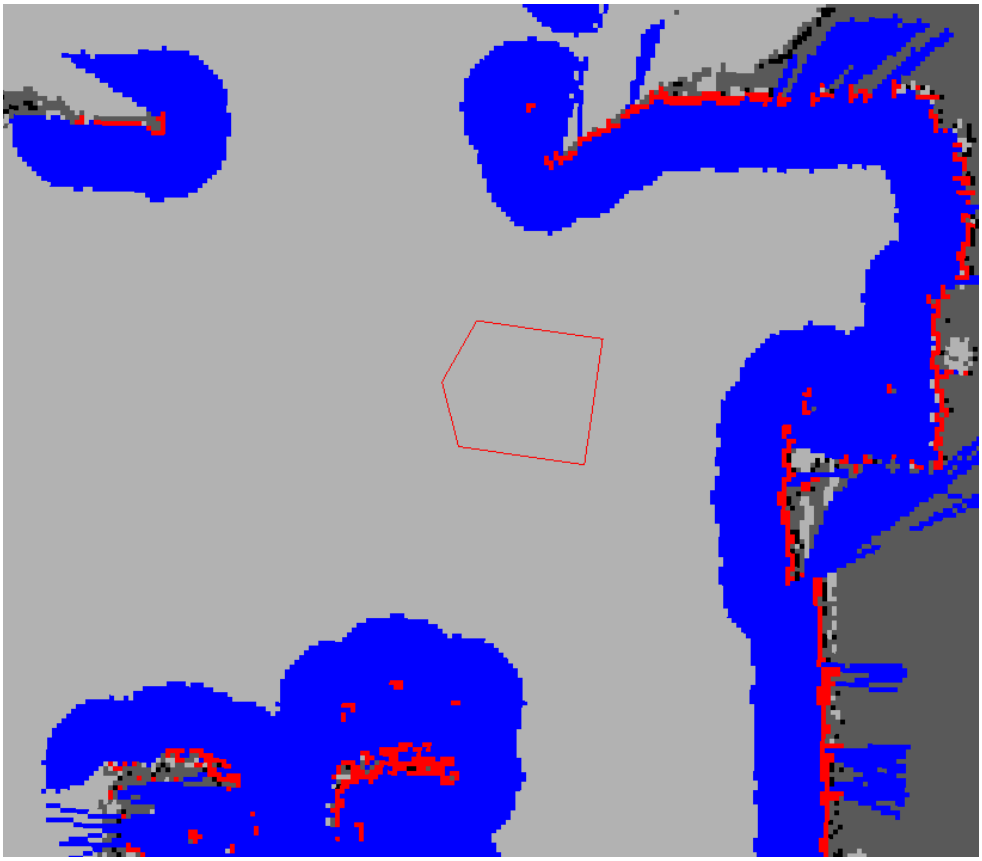


Running the move_base node on a robot that is properly configured (please see navigation stack documentation for more details) results in a robot that will attempt to achieve a goal pose with its base to within a user-specified tolerance. In the absence of dynamic obstacles, the move_base node will eventually 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.

4.3 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 o**bstacles 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.

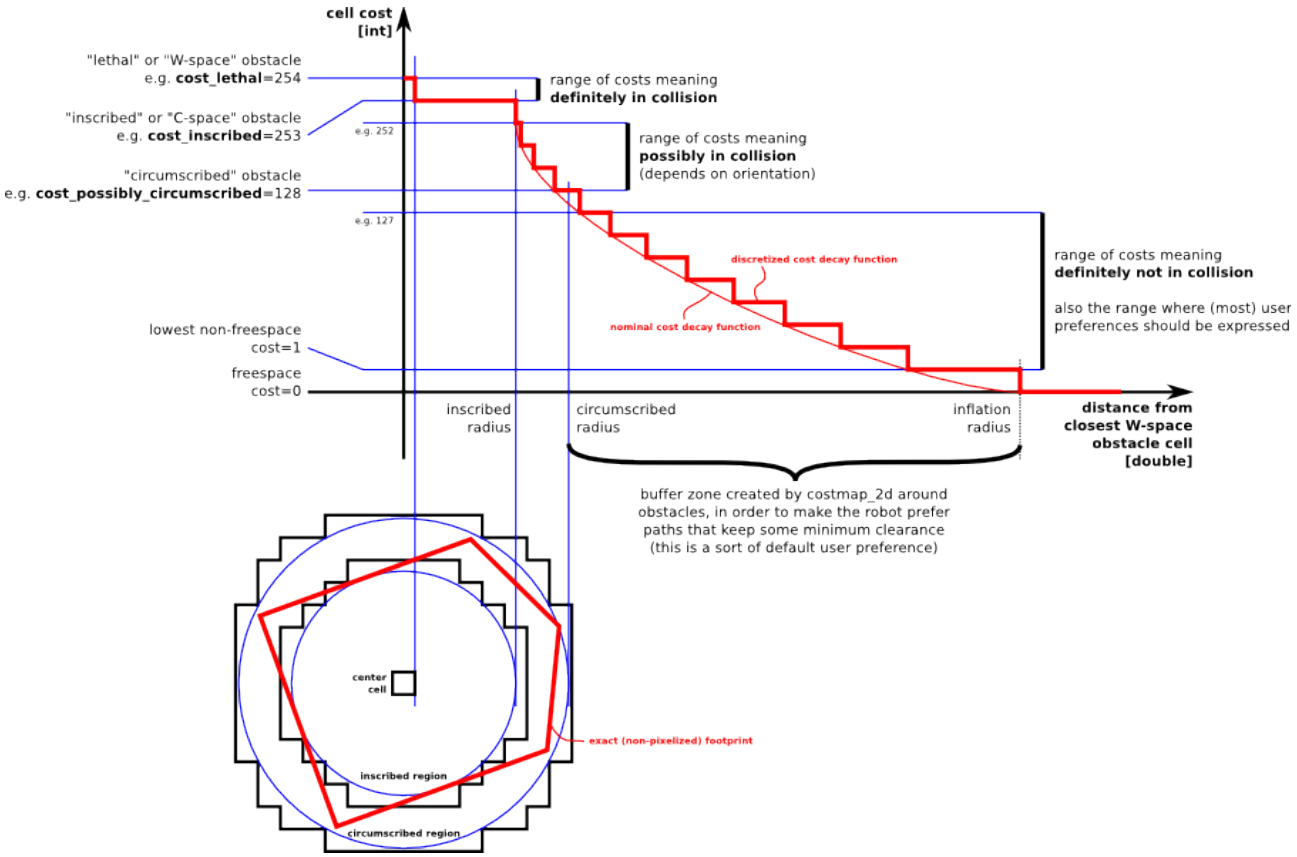
Occupied, Free, and Unknown Space

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 perfomed 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.

4.4 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
```

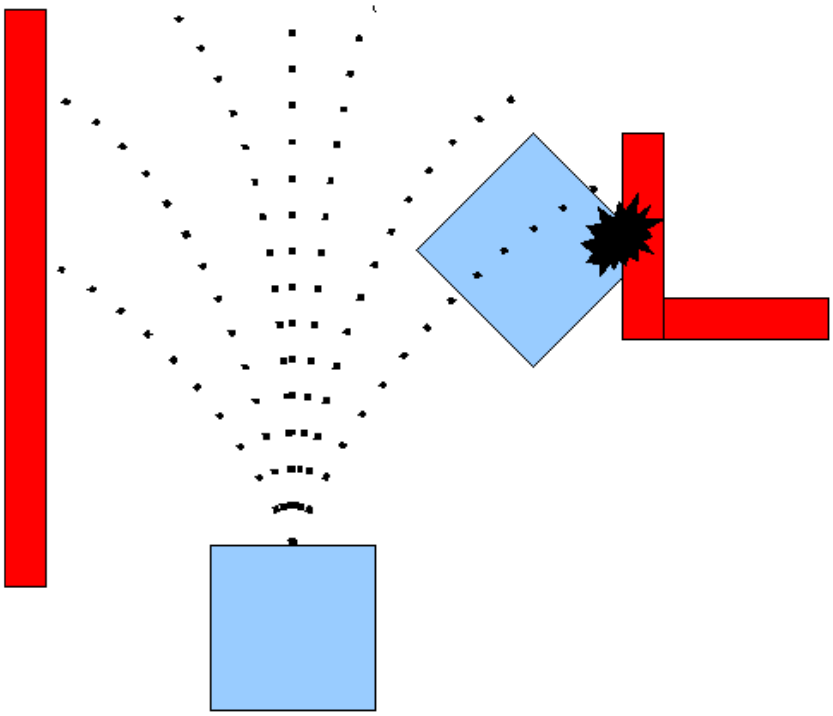
The config file will set the maximum and minimum velocities for your robot. The acceleration is also set.
The holonomic_robot parameter is true if you are using a holonomic platform. In our case, our robot is based on a non-holonomic platform and the parameter is set to false . A holonomic vehicle is 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) togglingquadratic approximation, (3) toggling grid path.

Examples of different Parameterizations

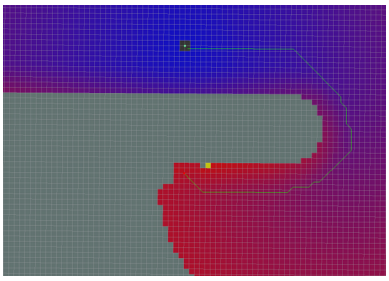


Illustration 2: all paramaters default

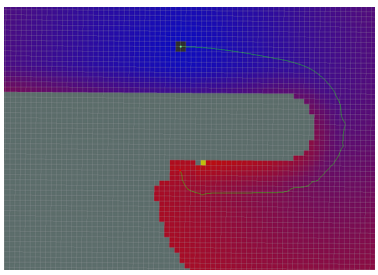


Illustration 1: use_grid_path = True

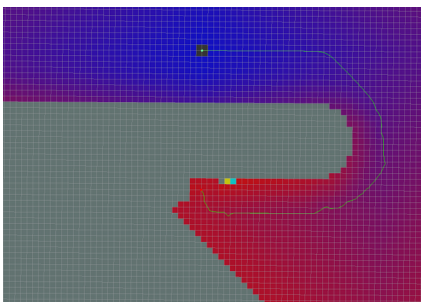


Illustration 4: use_quadratic = True

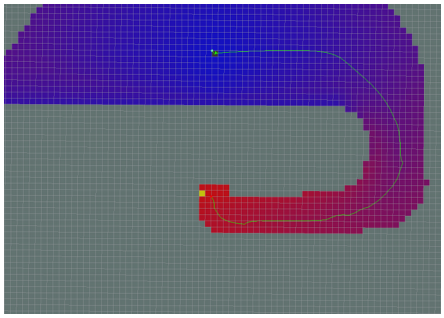


Illustration 3: use_dijkstra = False

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 appropiate namespaces.

Now after launching main.launch, amcl.launch and move_base.launch we are ready to use the Navigation Stack

4.5 Setting up rviz for the navigation stack

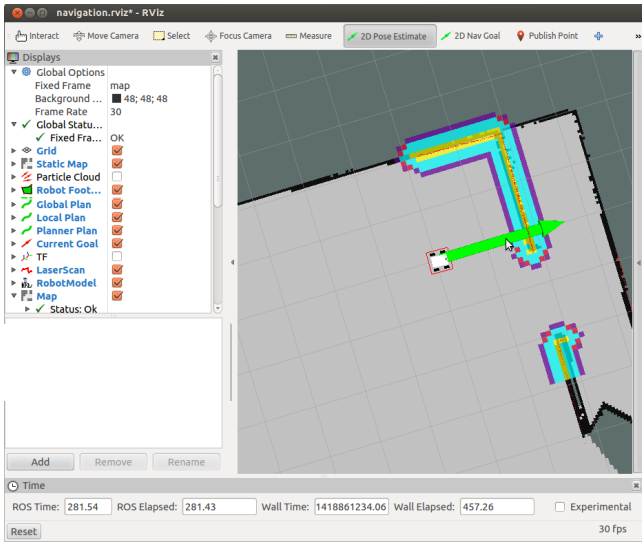
The 2D Pose estimated

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.

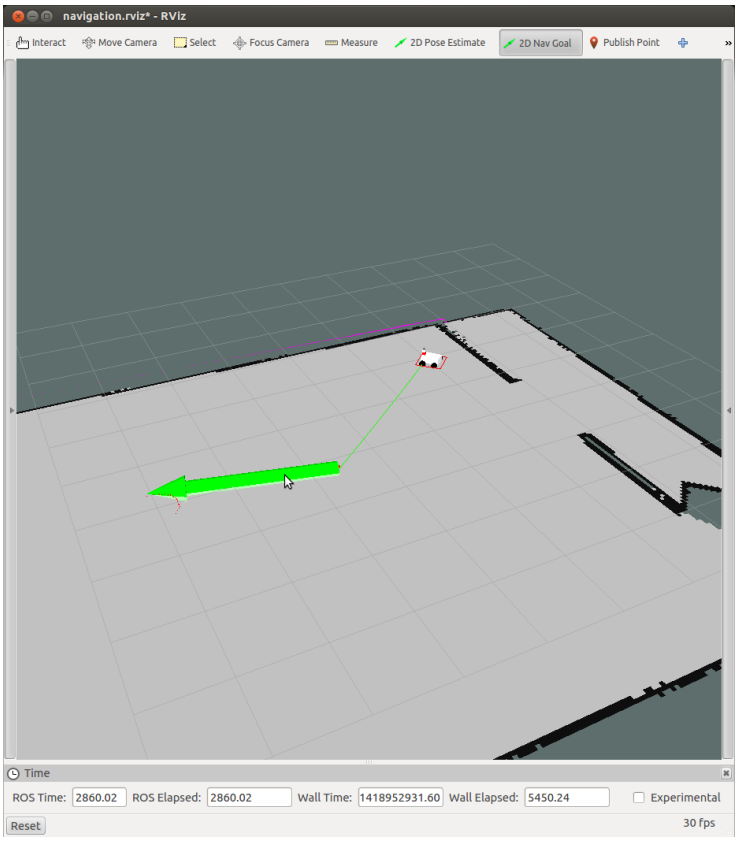
You can see in the following screenshot how you can use initialpose . Click on the 2D Pose Estimate button, and click on the map to indicate the initial position of your robot. If you don't do this at the beginning, the robot will start the auto-localization process and try to set an initial pose:

- Topic: initialpose
- Type: geometry_msgs/PoseWithCovarianceStamped



The 2D nav goal

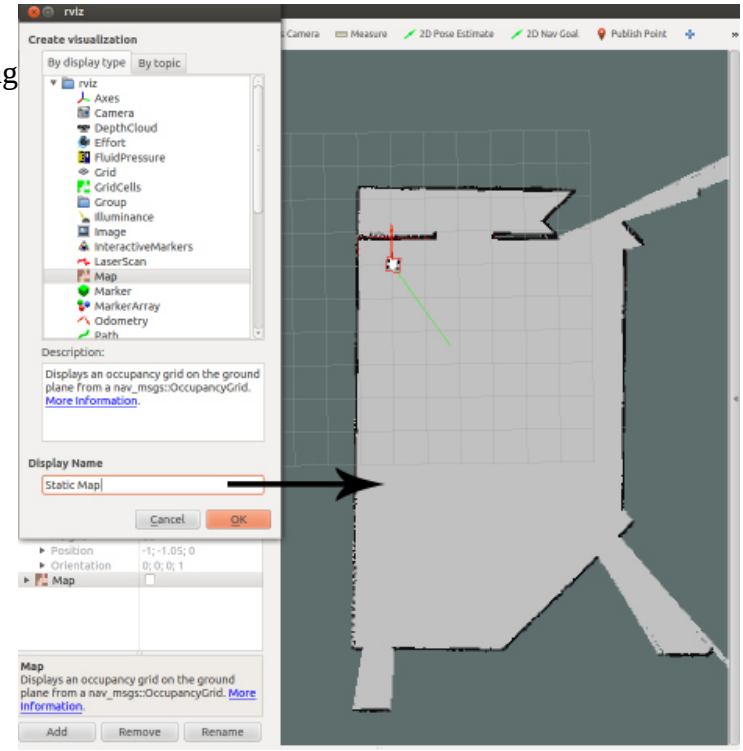
The 2D nav goal (G shortcut) allows the user **to send a goal to the navigation by setting a desired pose for the robot to achieve.** The navigation stack waits for a new goal with /move_base_simple/goal as the topic name; for this reason, you must change the topic's name in the rviz windows in Tool 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.

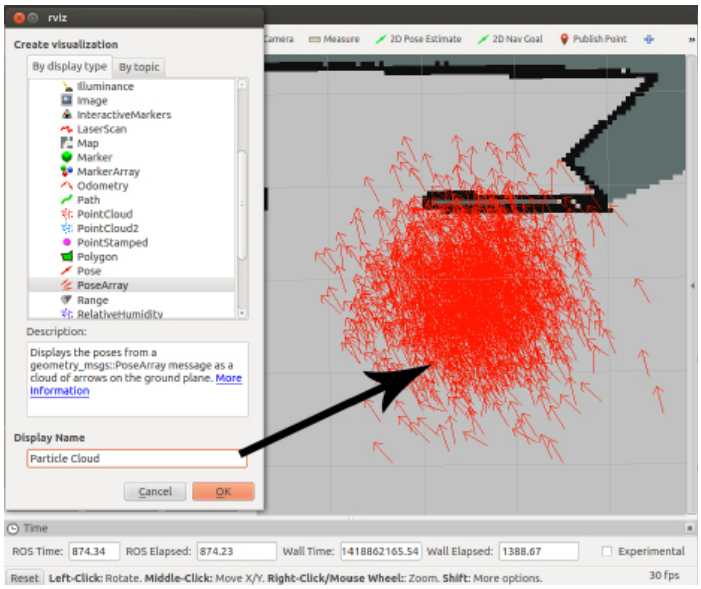


- **Topic:** map
- **Type:** nav_msgs/GetMap

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.

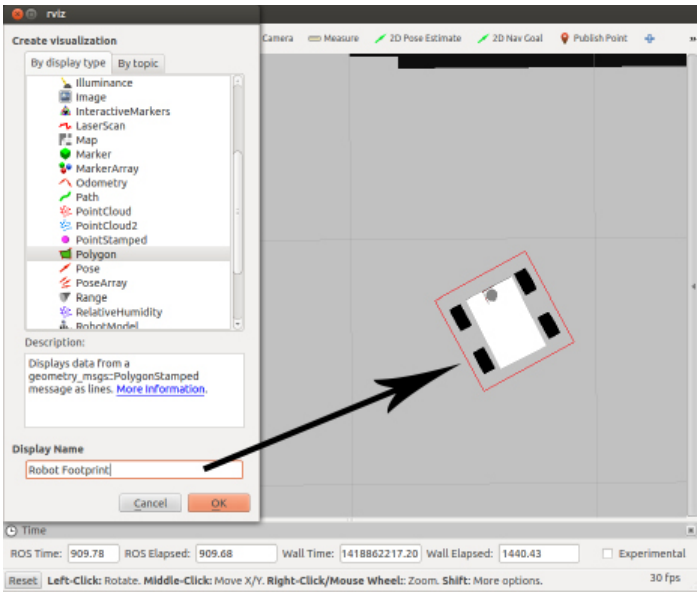
- **Topic:** particlecloud
- **Type:** geometry_msgs/PoseArray



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:

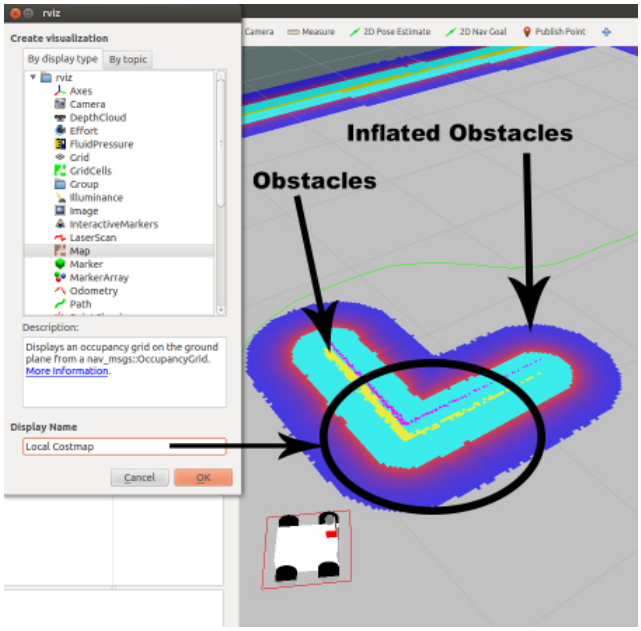
- **Topic:** local_costmap/robot_footprint
- **Type:** geometry_msgs/Polygon



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:

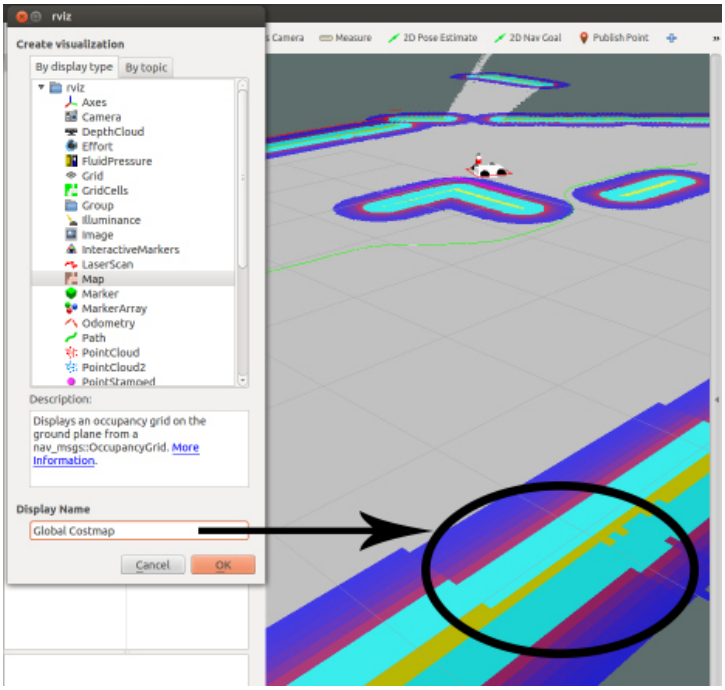
- **Topic:** move_base/local_costmap/costmap
- **Type:** nav_msgs/OccupancyGrid



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:

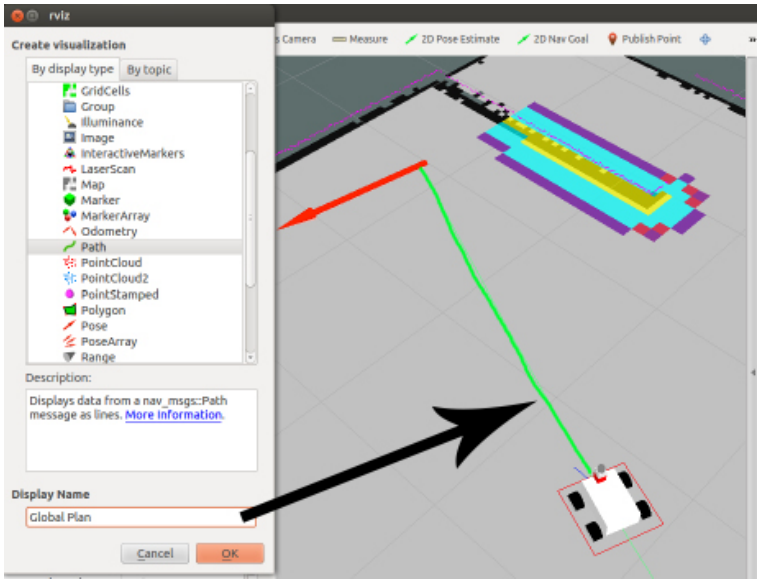
- Topic: /move_base/global_costmap/costmap
- Type: nav_msgs/OccupancyGrid



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.

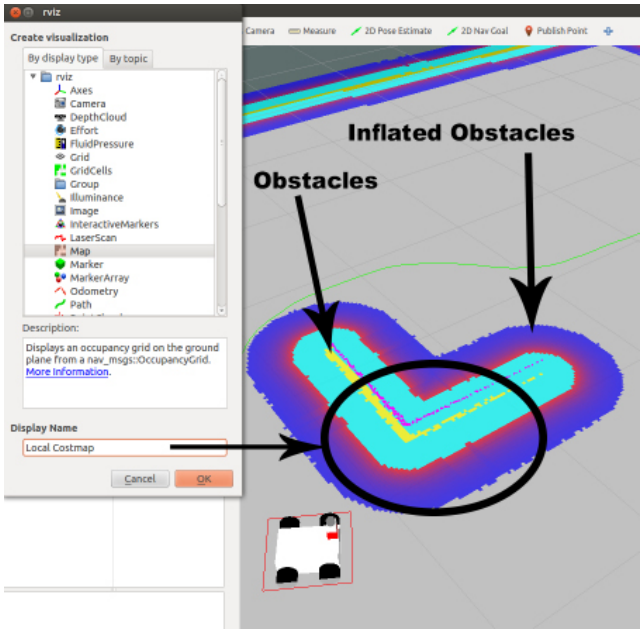
- Topic: TrajectoryPlannerROS/global_plan
- Type: nav_msgs/Path



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:

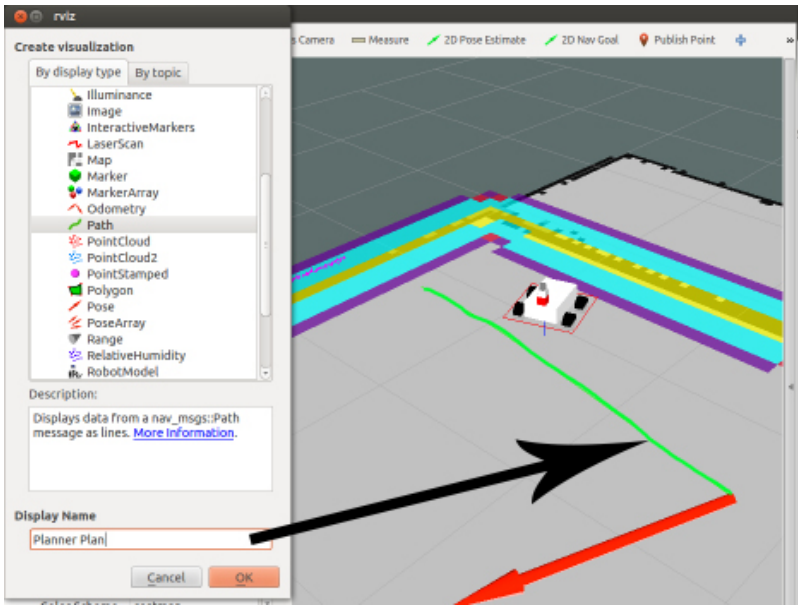
- Topic: TrajectoryPlannerROS/local_plan
- Type: nav_msgs/Path



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:

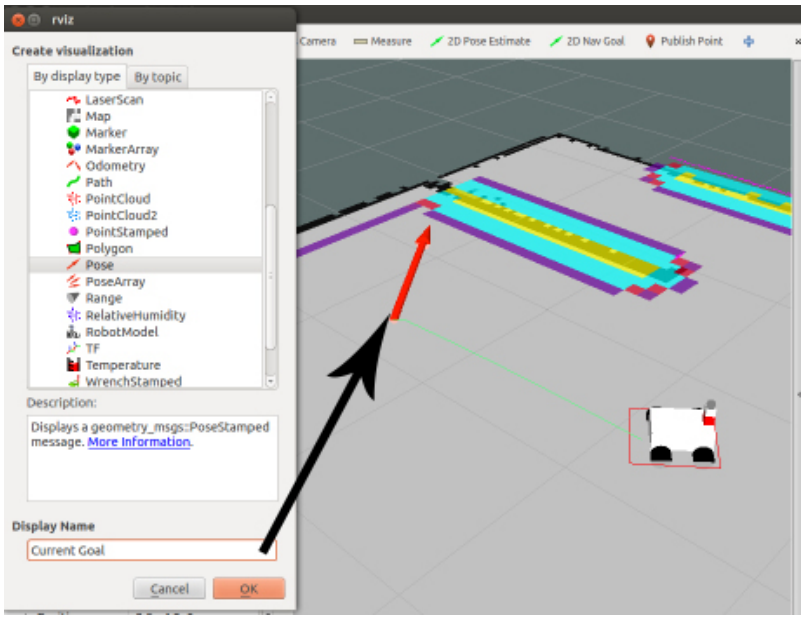
- Topic: NavfnROS/plan
- Type: nav_msgs/Path



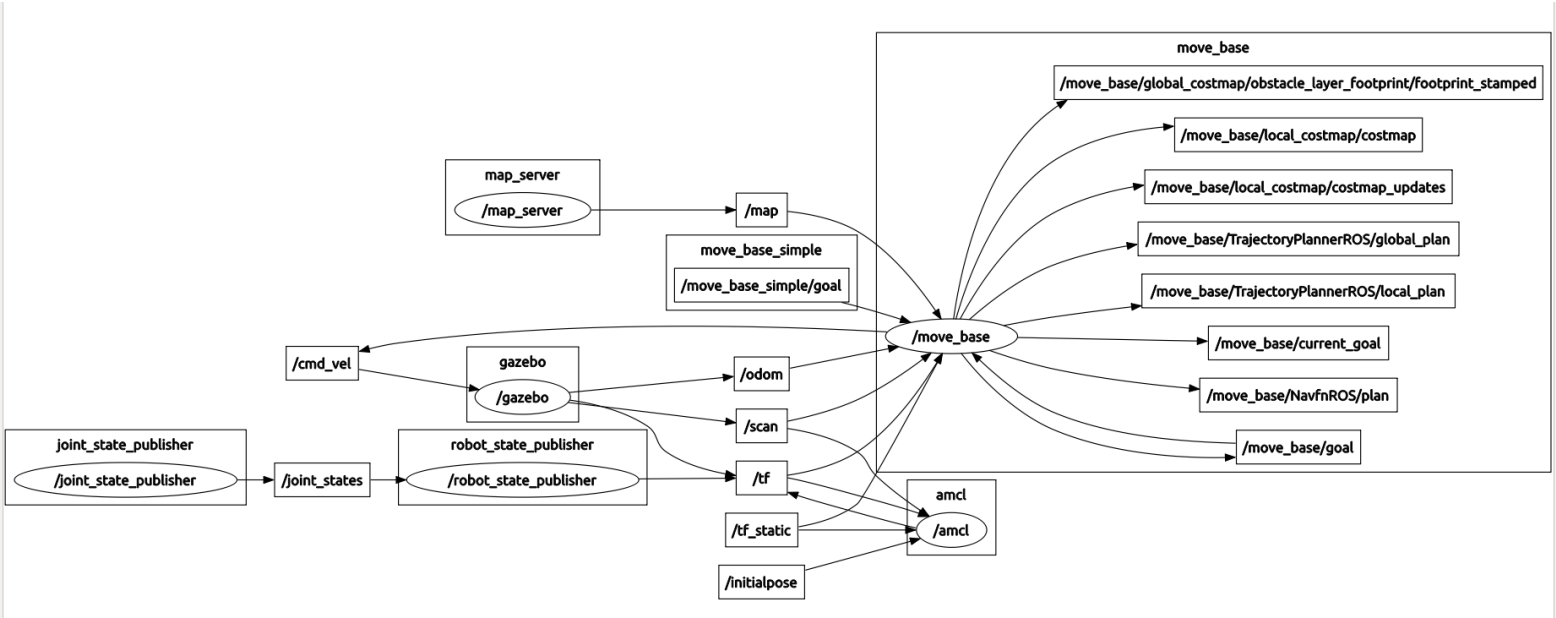
The current goal

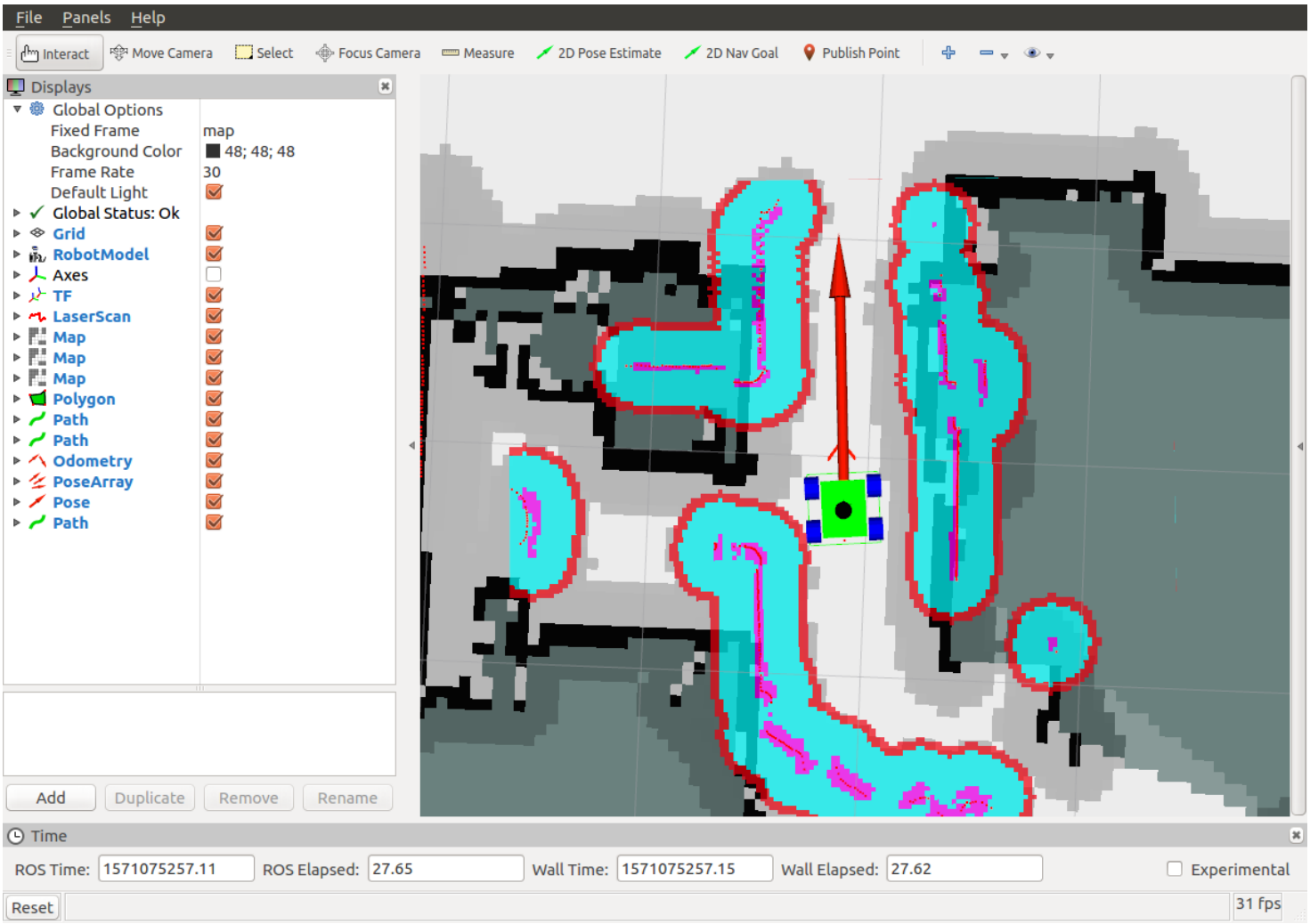
This shows the goal pose that the navigation stack is attempting to achieve. You can see it as a red arrow, and it is displayed after you put in a new 2D nav goal. It can be used to find out the final position of the robot:

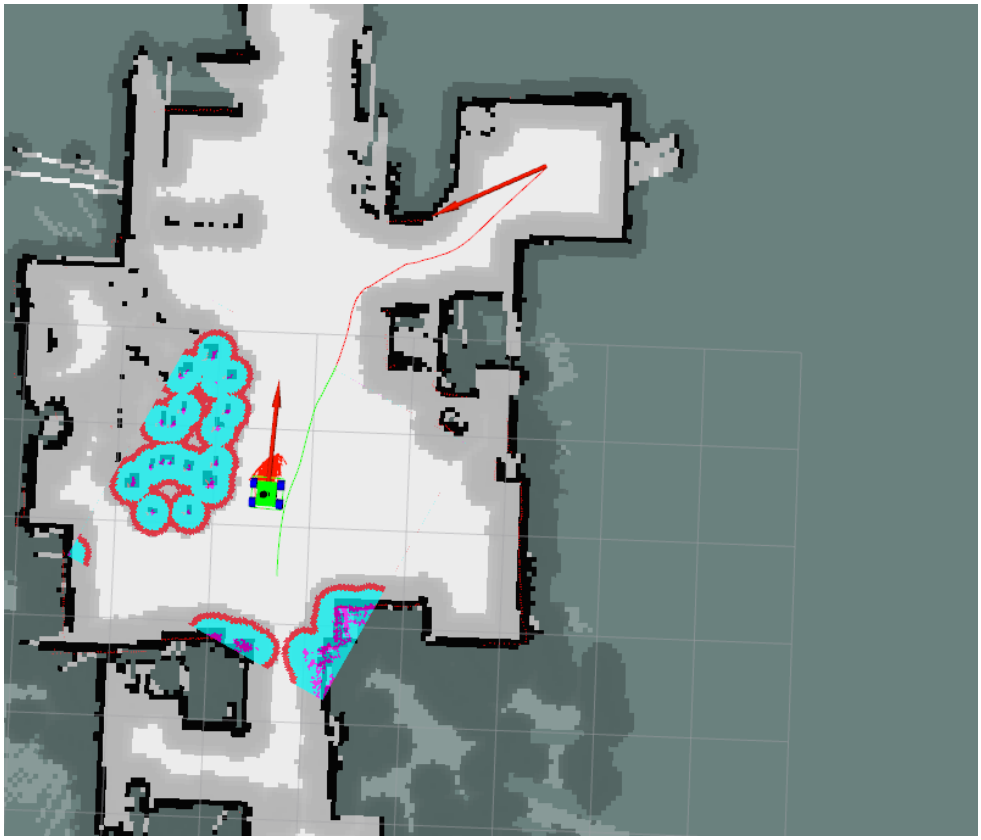
- Topic: current_goal
- Type: geometry_msgs/PoseStamped



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.


move_base
/move_base/global_costmap/obstacle_layer_footprint/footprint_stamped
/move_base/local_costmap/costmap
/move_base/local_costmap/costmap_updates
/move_base/TrajectoryPlannerROS/global_plan
/move_base/TrajectoryPlannerROS/local_plan
/move_base/current_goal
/move_base/NavfnROS/plan
/move_base/goal
map_server
/map_server
/map
move_base_simple
/move_base_simple/goal
/move_base
/cmd_vel
gazebo
/gazebo
/odom
/scan
/tf
joint_state_publisher
/joint_state_publisher
/joint_states
robot_state_publisher
/robot_state_publisher
amcl
/amcl
/tf_static
/initialpose


File
Panels
Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global Options
Fixed Frame
map
Background Color
48; 48; 48
Frame Rate
30
Default Light
Global Status: Ok
Grid
RobotModel
Axes
TF
LaserScan
Map
Map
Map
Polygon
Path
Path
Odometry
PoseArray
Pose
Path
Add
Duplicate
Remove
Rename
Time
ROS Time:
1571075257.11
ROS Elapsed:
27.65
Wall Time:
1571075257.15
Wall Elapsed:
27.62
Experimental
Reset
31 fps

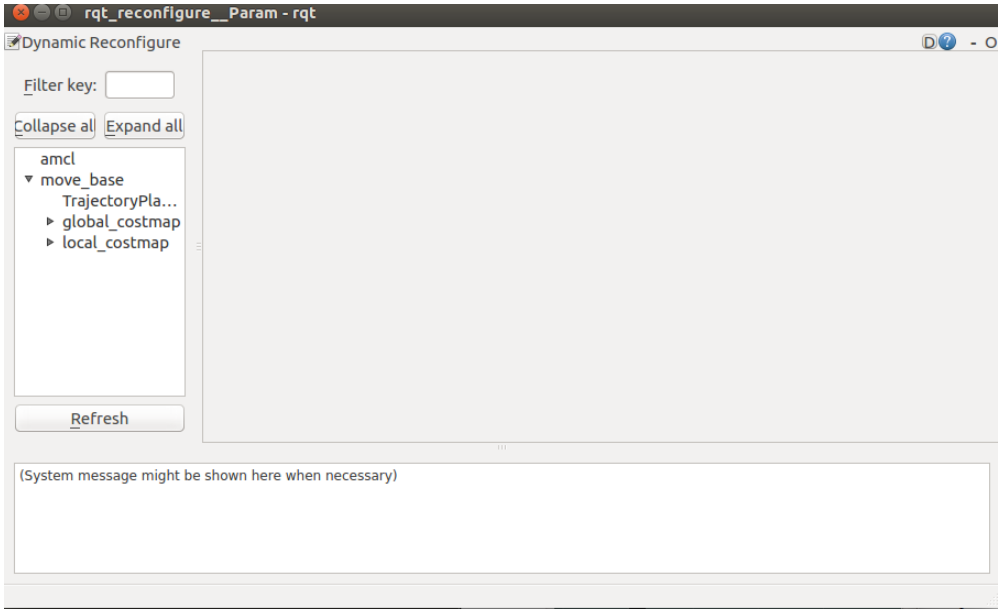


4.6 Modifying paramaters with rqt_reconfigure

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:

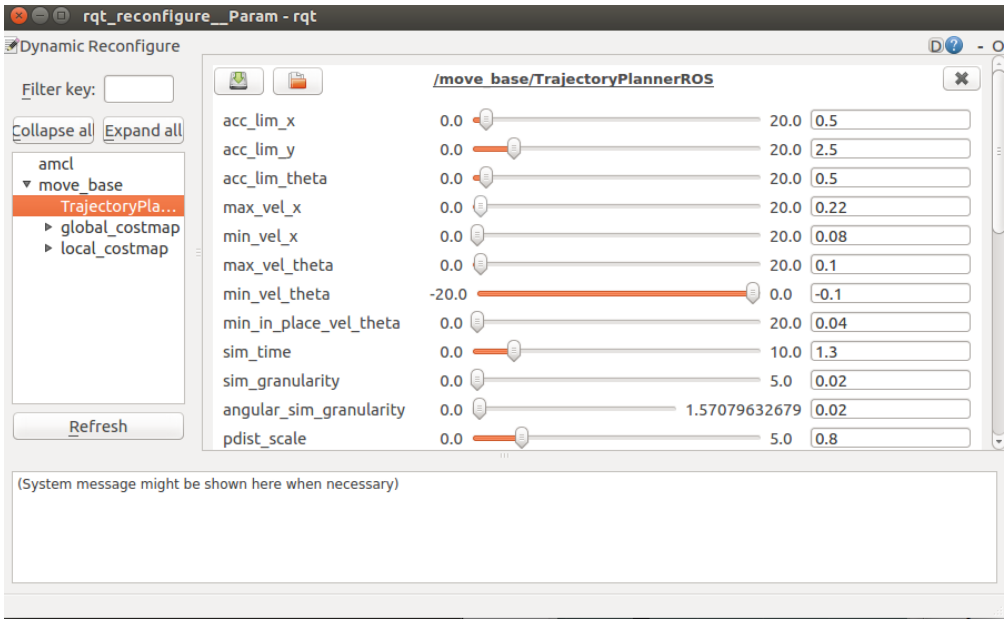
$ rosrun rqt_reconfigure rqt_reconfigure

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.



4.7 Avoiding obstacles

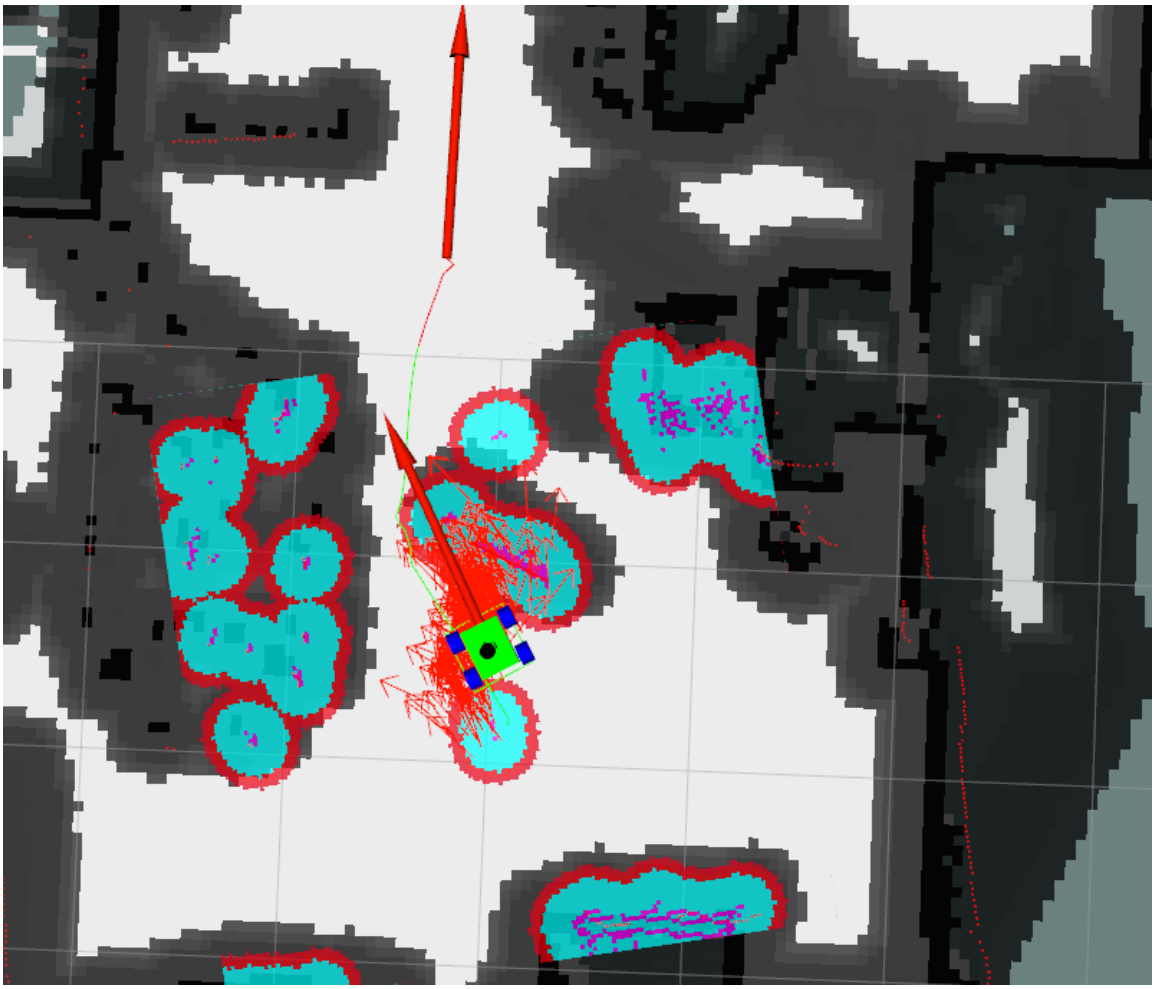
A great functionality of the navigation stack is the recalculation of the path if it finds obstacles during the movement. You can easily see this feature by adding an object in front of the robot. In our case i added a wood board in front of the robot. The navigation stack detects the new obstacle, and 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 usefull 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:



4.8 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.

4.9 Summary

Now you should br 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 reusability, 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.