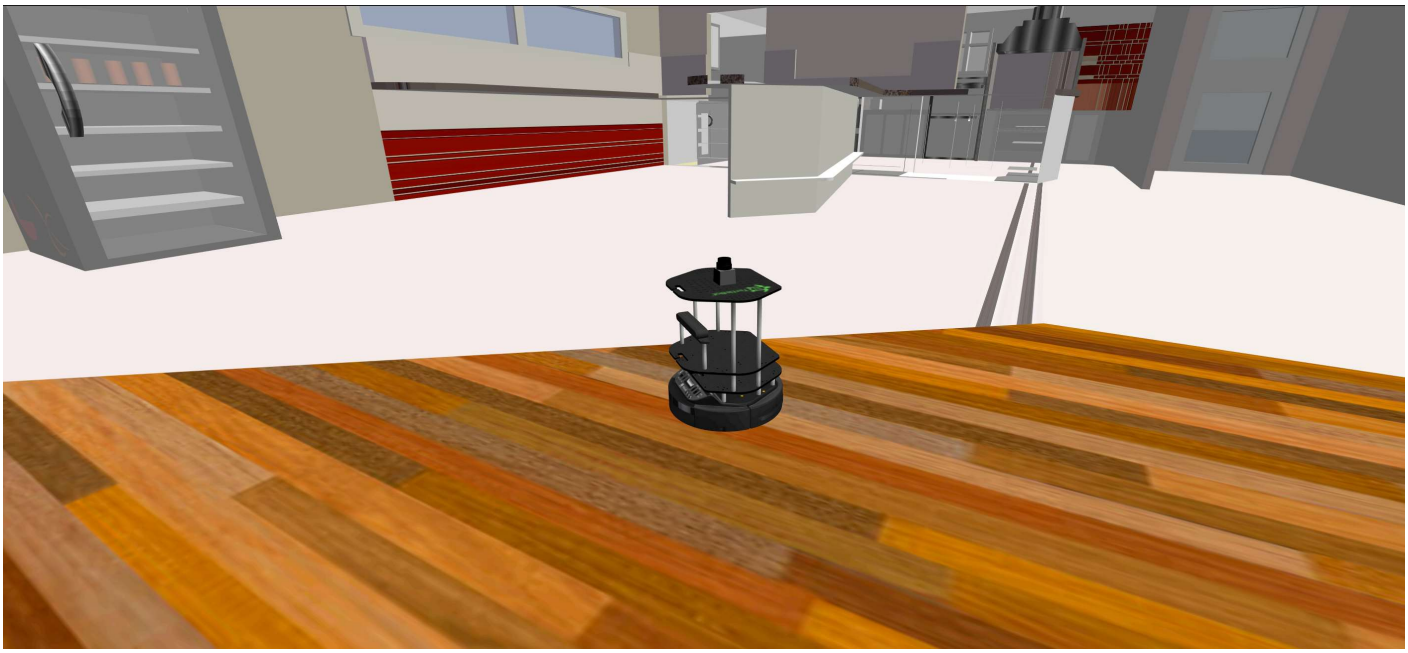
ROS Navigation in 5 Days

Unit 1: Basic Concepts



- Summary -

Estimated time of completion: **2 hours**

What you will learn with this unit?

- What is the ROS Navigation Stack?
- What do I need to work with the Navigation Stack?
- What is the move_base node and why it is so important?
- Which parts take place in the move_base node?

- End of Summary -

Let's imagine the next scenario. You're chilling in your lab, listening to your favourite 80s music remix while carrying out a detailed investigation about the last kitten videos that have been uploaded to Youtube... when suddenly, your boss comes into the room and says the magic words: ¡Hey (Insert Your Name Here)! I have a new project for you. I need you to make this robot navigate autonomously with ROS, and I need it for yesterday.

Probably you'll start to sweat, while asking yourself questions like: ¿Navigate? ¿For yesterday? ¿With ROS? ¿What the hell do I need? ¿Where do I start? ¿How does BB-8 climb stairs?

Don't panic! Keep Calm... Robot Ignite Academy comes to rescue you!!

1.1 What do you need to perform robot navigation with ROS?

1.1.1 First, you need a map

The very first thing you need in order to perform Navigation is... of course, a **Map**. But not a treasure Map, or a Map of the state of Wisconsin... No!! You need a Map of the environment where you want your robot to navigate at. Sounds pretty obvious, right? But how could possibly any Robot perform autnomous navigation without providing it with a map of the environment? It just can't.

So first thing I need to do is to get a Map of the environemnt. Great!! But wait... How do I get a this map? Where do I get it from?

You use the robot to create it, of course! A Map is just a representation of an environment created from the sensor readings of the robot (for example, from the laser, among others). So just by moving the robot around the environment, you can create an awesome **Map** of it! In terms of ROS Navigation, this is knows as **Mapping**. Would you like to see an example of how this works? Then let's do it!

Wait a second! How rude by my part! I didn't even made the proper introductions yet...

In the top right corner of your screen you have the simulation window. In this window you will meet different robots during the course (depending on the Chapter you are) that will help you in the process of learning. In this first Chapter (Basic Concepts), you'll be working with the lovely Kobuki located in a cafeteria. I'm sure you'll get along well!

So now we've made the proper introductions... let's start building a map with the Kobuki robot!

- Example 1.1 -

First, you need to launch the map builder program.

Execute the following command in the WebShell number #1 in order to launch the Mapping demo.

▶ **Execute in Shell #1**

In [ ]:
```
roslaunch turtlebot_navigation_gazebo gmapping_demo.launch
```

Next, you need a program to manually move the robot around, so that you can show the environment to the robot. Execute the following command in the WebShell number #2 in order to launch the keyboard teleoperation program. You will use this programs to move the robot by pressing keys in the keyboard, so that you can move the robot around.

▶ Execute in Shell #2

In [ ]:
```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

Hit the icon with a screen in the top-right corner of the IDE window

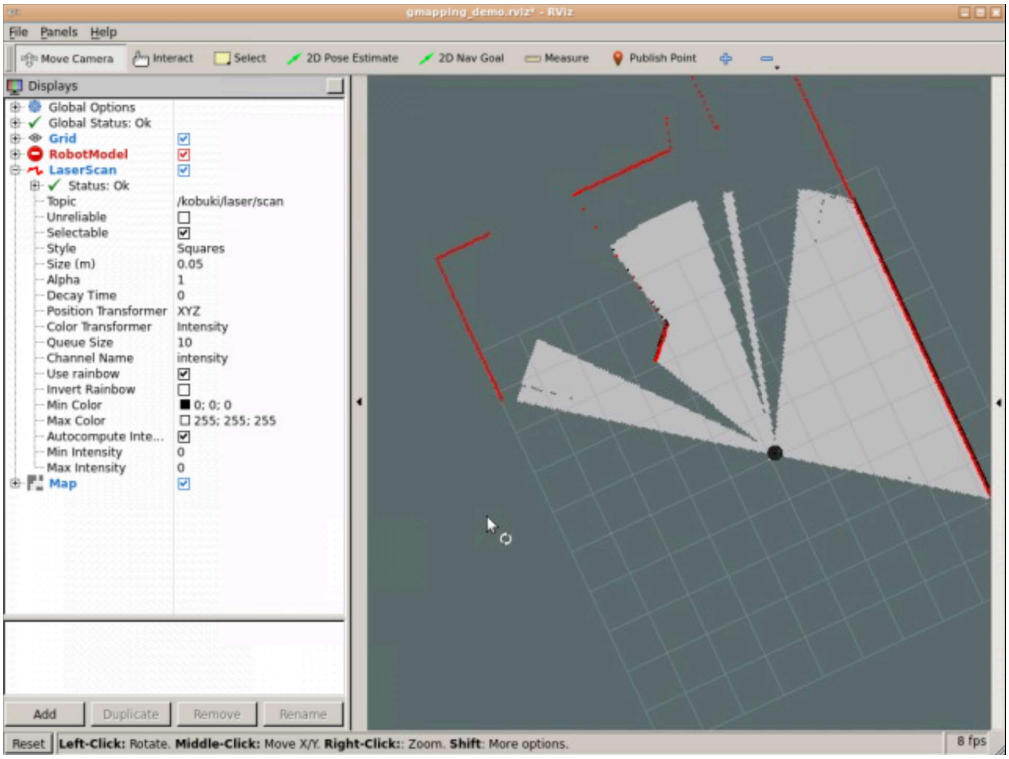
in order to open the Graphic Interface.

Now, execute the following command in the WebShell number #3 in order to start RViz with a predefined configuration.

▶ Execute in Shell #3

In [ ]:
```
roslaunch turtlebot_rviz_launchers view_mapping.launch
```

A new window should appear on the Graphic Interface tab containing the Rviz application.



Start moving the Robot around the cafeteria using the keyboard teleop program. Remember that in order to move the robot you need to have the focus of the browser on the WebShell #2 (the one that is executing the keyboard teleop program).

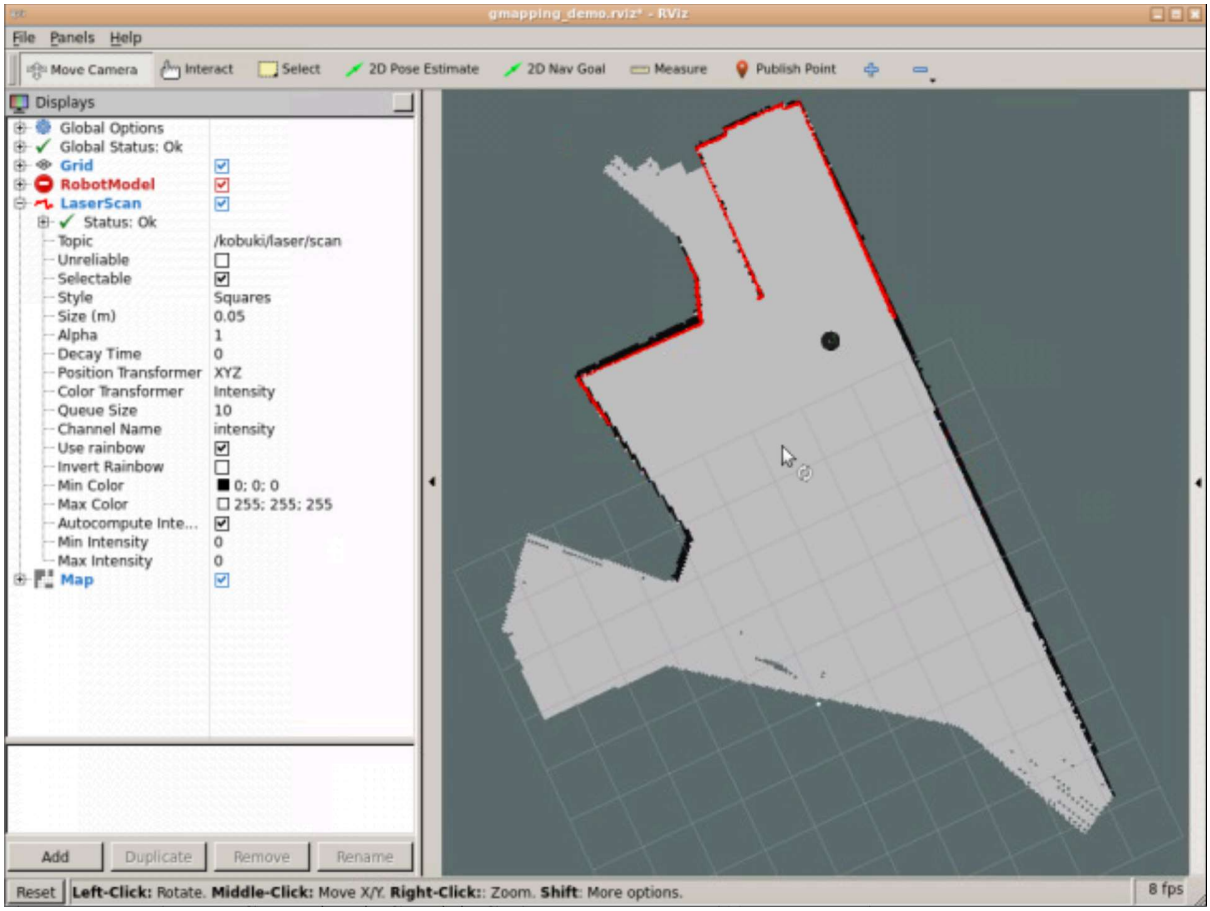
Once you start moving the robot, you will see how the map is created on the Graphic Interface tab.

The basic keys in order to move the robot with the keyboard are the following:

i	**Move forward**
,	**Move backward**
j	**Turn left**
l	**Turn right**
k	**Stop**
q z	**Increase / Decrease Speed**

- End of Example 1.1 -

- Expected Result for Example 1.1 -



- End of Expected Result -

- Notes for Example 1.1 -

Check the following Notes in order to complete the Example:

Note 1: It's not necessary to map the whole room. This exercise is only a demo of the mapping process.</i>

Note 2: You can try to figure out what is happening by checking different topics. You can get an idea of the topics involved in the process by looking at the launch files code, or in RViz configuration. Use the webshells for this purpose.

Note 3: You can change Rviz configuration as you want, and check how it affects the visualization of the mapping process.

- End of Notes -

Awesome, right? You'll probably have a lot of questions about how this whole process works... but remember this is just the Basic Concepts Unit. For now, you just need to get a general picture of it. You'll learn how the whole process works and how to apply this to your own robot on the **Mapping Unit (Chapter 3)**. Just be patient!

1.1.2 Next you need to localize the robot on that map

So now, let's move on. You've seen that you need a Map in order to Navigate autonomously with your robot, but is it enough? What do you think?

As you may imagine, the answer is NO. You have a Map of the environment, yes, but this is completely useless if your robot doesn't know **WHERE it is with respect to this map**. This means, in order to perform a proper Navigation, your robot needs to know in which **position** of the Map it is located and with which **orientation** (that is, which direction the robot is facing) at every moment. In terms of ROS Navigation, this is known as **Localization**. Let's see a quick demonstration of localization.

Let's see how the Kobuki robot can localize itself in the map we previously built.

- Example 1.2 -

IMPORTANT: Make sure to stop all the previous programs running in your Web Shells (by pressing Ctrl+C) before starting with this example.

Execute the following command in the WebShell number #1 in order to start the Localization demo.

▶ Execute in Shell #1

In [ ]:
```
roslaunch turtlebot_navigation_gazebo amcl_demo.launch
```

Execute the following command in the WebShell number #2 in order to launch the keyboard teleop program.

▶ Execute in Shell #2

In [ ]:
```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

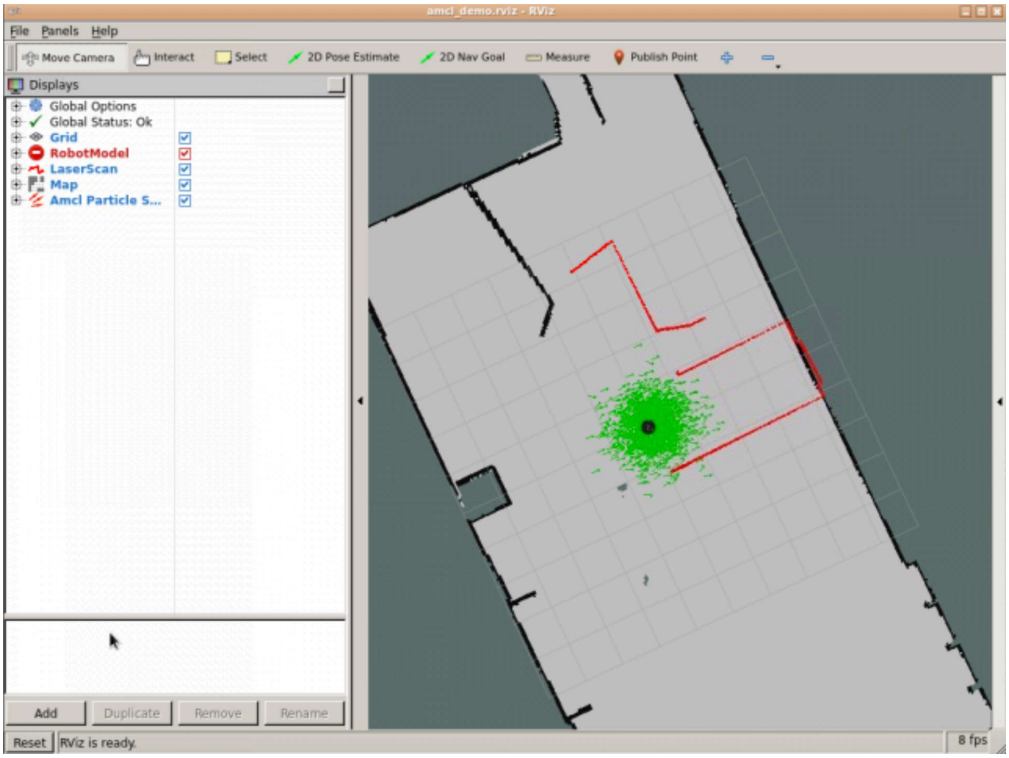
Execute the following command in the WebShell number #3 in order to start RViz with a predefined configuration.

▶ Execute in Shell #3

In [ ]:
```
roslaunch turtlebot_rviz_launchers view_localization.launch
```

Start moving the Robot around the cafe using the keyboard teleop. Remember that in order to move the robot you need to have the focus of the browser on the WebShell #2 (the one that is executing the keyboard control program).

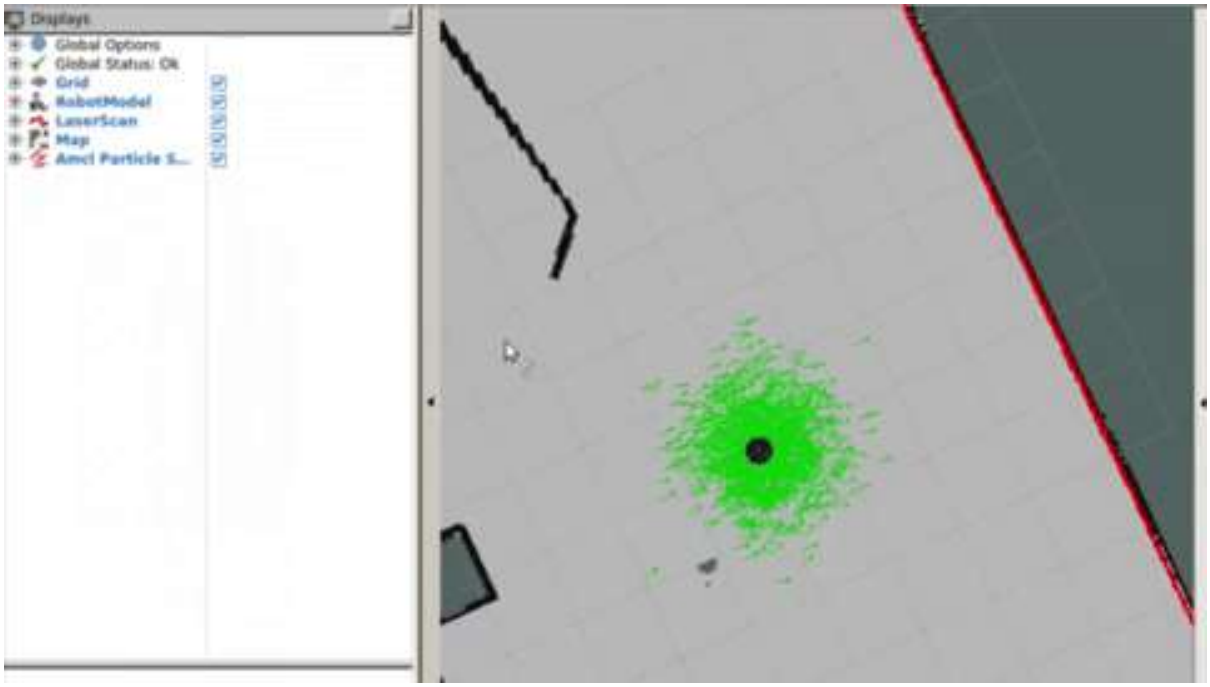
You should see on the Rviz window the map of the cafeteria, a representation of the Kobuki on that map, and a lot of green arrows. Those green arrows represent location guesses of the robot in the map. That is, the green arrows are guesses that the localization algorithm is doing in order to figure out where in the map the robot is located. The arrows will concentrate on the most likely location when you move the robot. Let's try this.



Start moving the Robot around the cafe using the keyboard teleop program and see how the green arrows modify their position on the map, adjusting to the actual location of the robot in the map.

- End of Example 1.2 -

- Expected Result for Example 1.2 -



- End of Expected Result -

- Notes for Example 1.2 -

Check the following Notes in order to complete the Example:

Note 1:: You can try to figure out what is happening by checking different topics. You can get an idea of the topics involved in the process by looking at the launch files code, or in RViz configuration.

Note 2:: You can change Rviz configuration as you want, and check how it affects the visualization of the localization process.

Note 3:: You can help the robot localize by providing it its location on the map. To do that, go to the Rviz app. Then press the button *2d Pose Estimate* and go to the map and drag and drop at the location the robot is (more or less). The green arrows will be spread arround that location.

- End of Notes -

I bet you're getting thrilled about this whole thing, right? But I'm also sure you still have more questions. Remember, this is just the Basic Concepts Unit. In the **Localization Unit (Chapter 3)** you will learn how to configure the localization system for your robot, and how to get information from it.

1.1.3 Now you can send goal locations to the robot

Thought you're done? Not at all! You still need a couple of things in order to perform ROS Navigation. For now we've already covered the first block, which is building our own **Map** of the environment and being able to **Localize** the robot in it. So what's next? We should start Navigating autonomously in it, right?

For this, we'll need some kind of system which tells the robot **WHERE** to go, at first, and **HOW** to go there, at last. In ROS, we call this system the **Path Planning**. The Path Planning basically takes as input the current location of the robot and the position where the robot wants to go, and gives us as an output the best and fastest path in order to reach that point. Let's see an example of how this works.

- Example 1.3 -

IMPORTANT: Make sure to stop all the programs running in your Web Shells (by pressing Ctrl+C) before starting with this example.

Execute the following command in the WebShell number #1 in order to start the Path Planning demo.

▶ Execute in Shell #1

In [ ]:
```
roslaunch turtlebot_navigation_gazebo move_base_demo.launch
```

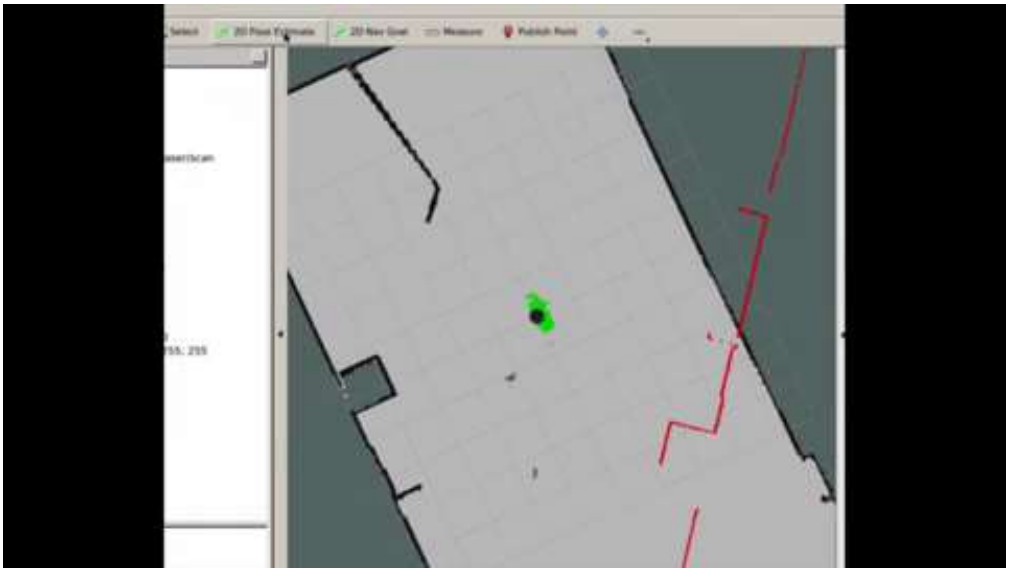
Execute the following command in the WebShell number #2 in order to start RViz with a predefined configuration.

▶ Execute in Shell #2

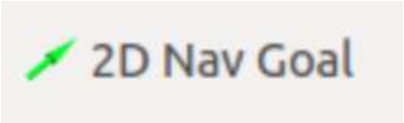
In [ ]:
```
roslaunch turtlebot_rviz_launchers view_planning.launch
```

Use the 2D Pose Estimate tool in Rviz to Localize the Robot. There are two ways of localizing the robot in the map: one is moving the robot around until it localizes itself (as we did in the previous exercise) and another one is to provide the location to the algorithm manually (because we as humans, we know where the robot is located in the map). The second approach is a lot faster and it is the one we are going to do now.

Press the "2D Pose Estimate" button at the top menu in RViz. Then, in the RViz central panel, press in the position in the map where the robot is. Indicate also the orientation of the robot, by clicking, dragging and dropping in the direction of the robot orientation.

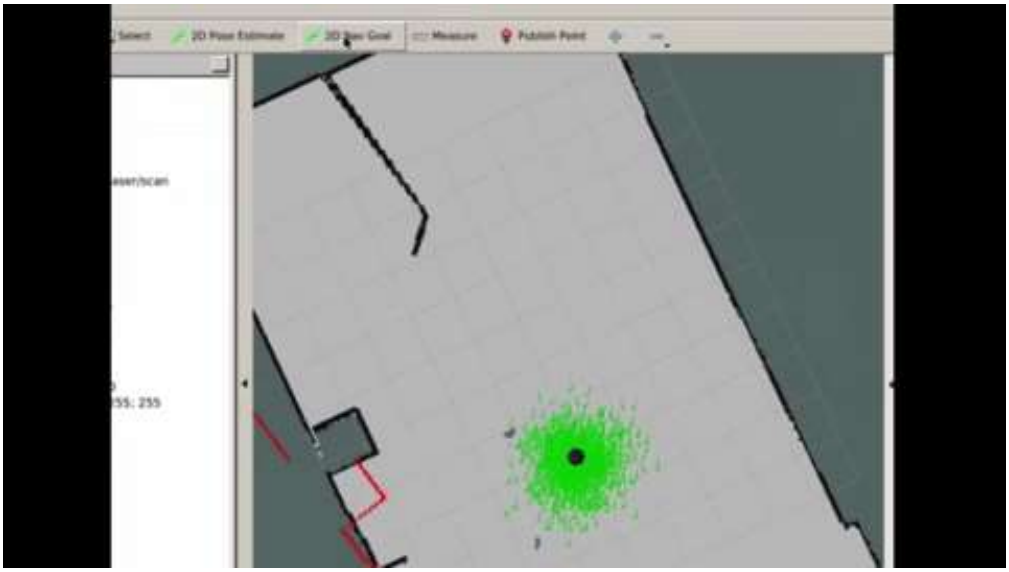


Use the 2D Nav Goal tool in Rviz to send a Goal to the Robot.



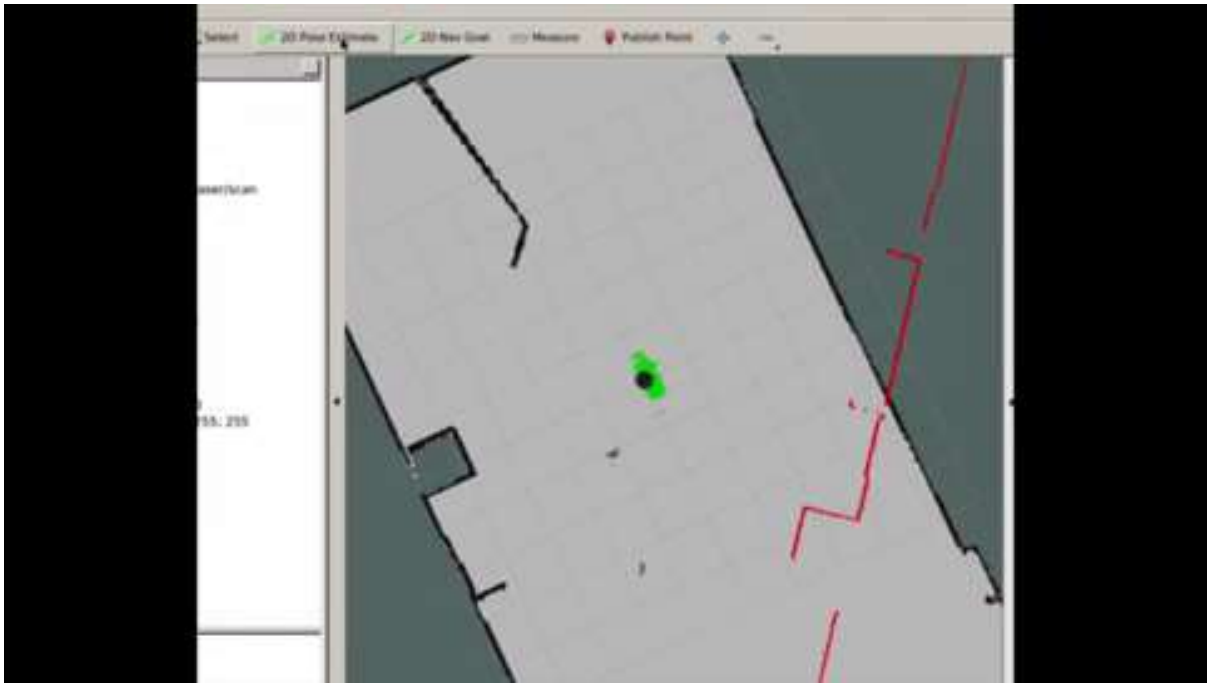
Press the "2D Nav Goal" button at the top menu in RViz. Then, in the RViz central panel, press in the position in the map where you want your robot to go. Indicate also the orientation at the end of the movement.

At this point, you should see a green line appearing in the Rviz window, on top of the map. That is the path calculated to go from the current location to the goal location. Also, the robot will start moving towards that goal (check the simulation window).



- End of Example 1.3 -

- Expected Result for Example 1.3 -



- End of Expected Result -

- Notes for Example 1.3 -

Check the following Notes in order to complete the Example:

Note 1:: The 2D Pose Estimate allows you to tell the system what's the initial position and orientation of the robot. The robot needs to be localized in the map, in order to be able to receive a location to go (Navigation Goal).

Note 2:: The 2D Nav Goal allows you to send to the robot the position in the map where you want it to navigate to.

Note 3:: In both cases, after setting the position by pressing in RViz's central panel map, you'll have to set up the orientation as well, by pointing the green arrow that appears to the correct direction (drag and drop).

- End of Notes -

I'm sure you're quite impressed right now, am I right? Sure!!

But how does it know the best Path to follow? Maybe it uses the Map we created previously in order to calculate this Path? Hmmm... don't go so fast, buddy! As I told you before, we also have a Unit we're we will discuss largely about all this topics, and that's no other than the **Path Planning Part 1 (Chapter 4)**. On that unit you will learn how to set up a path planner for your own robot and how to ask to it for destinations and status using your own programs.

1.1.4 Finally, you need to avoid obstacles

Anyway, there is actually a question that it is relevant for the next topic we're going to introduce... HOW does ROS manage to avoid, for instance, a table in the environment? What happens if someone or something suddenly walks into the path of the robot? Will the robot know it's there? Will it be able to avoid that obstacle? All of this questions are related to the same topic, which is called **Obstacle Avoidance**.

Basically, the Obstacle Avoidance system breaks the big picture (Map) into smaller pieces, which updates in real-time using the data it's getting from the sensors. This way, it assures it won't be surprised for any sudden change in the environment, or by any obstacle that appears in the way. Let's see an example of how this works.

- Example 1.4 -

IMPORTANT: Make sure to stop all the programs running in your Web Shells (by pressing Ctrl+C) before starting with this example.

Execute the following command in the WebShell number #1 in order to start the Obstacle Avoidance demo.

▶ Execute in Shell #1

In [ ]:
```
roslaunch turtlebot_navigation_gazebo move_base_demo.launch
```

Execute the following command in the WebShell number #2 in order to copy to your workspace the URDF file of the obstacle we are going to spawn into the simulation.

▶ Execute in Shell #2

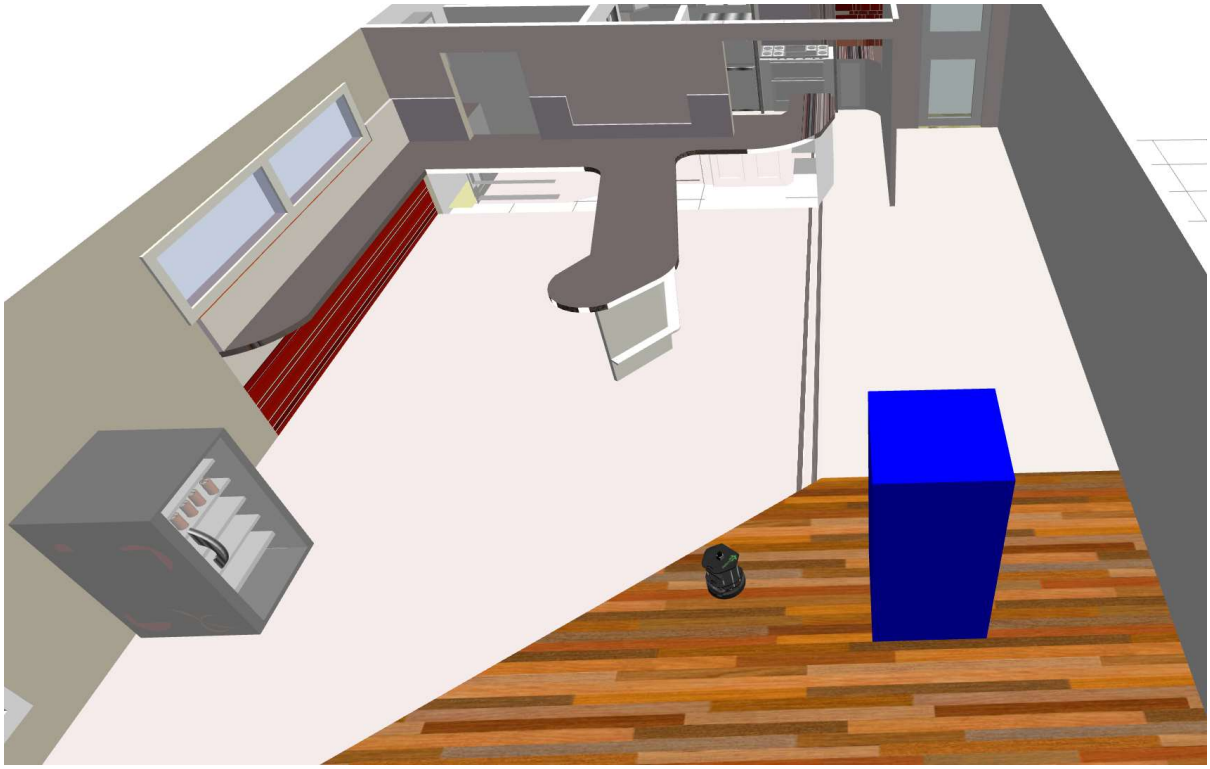
In [ ]:
```
cp /home/simulations/public_sim_ws/src/all/turtlebot/turtlebot_navigation_gaze
```

Execute the following command in the WebShell number #2 in order to spawn the obstacle in the room.

▶ Execute in Shell #2

In [ ]:
```
rosrun gazebo_ros spawn_model -file /home/user/catkin_ws/src/object.urdf -urdf
```

If everything goes fine, you should a blue box spawning into the simulation. Like this:



Execute the following command in the WebShell number #2 in order to start RViz with a predefined configuration.

▶ Execute in Shell #2

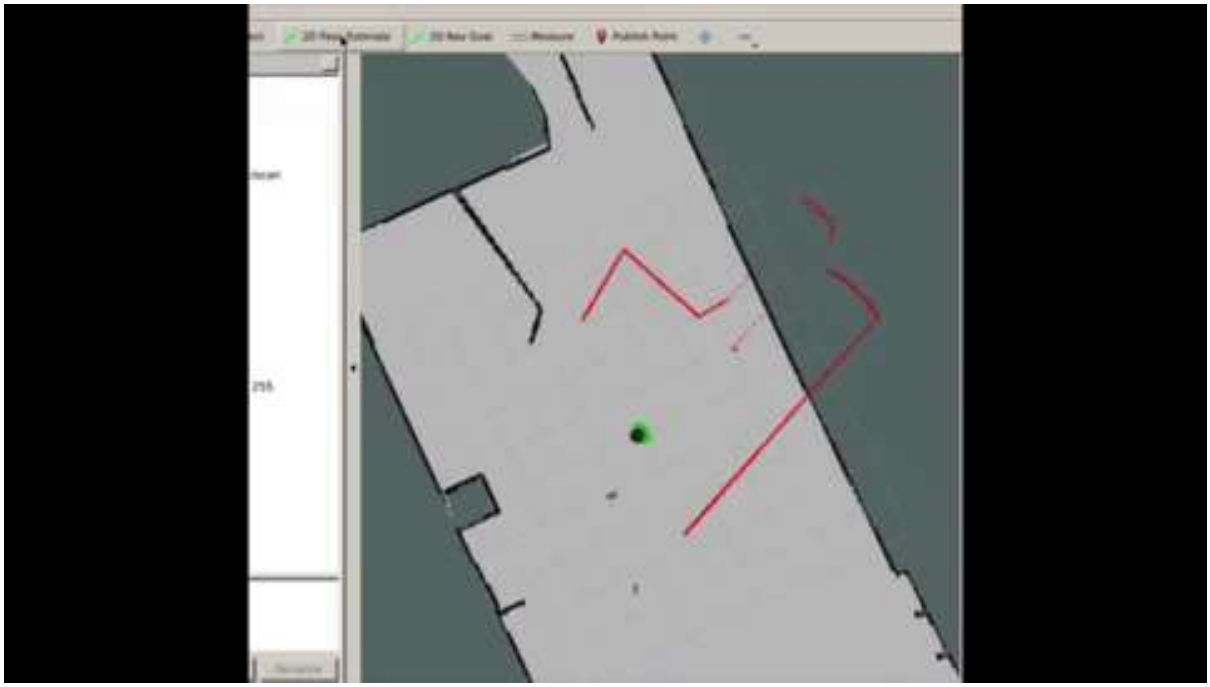
In [ ]:
```
roslaunch turtlebot_rviz_launchers view_planning.launch
```

Use the 2D Pose Estimate and 2D Nav Goal tools in RViz (as in the previous example) in order to tell the robot WHERE it is and WHERE you want it to go. Provide to the robot a location that enforces the robot to avoid the obstacle to check how the system works.

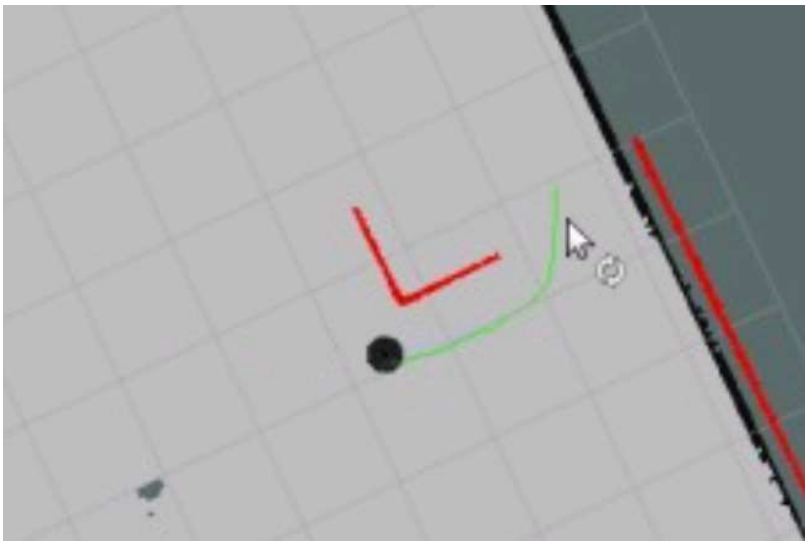
Set a Pose Goal where the robot has to somehow avoid the obstacle you've just spawned into the simulation.

- End of Example 1.4 -

- Expected Result for Example 1.4 -



Blue box detected by the laser of the Kobuki robot:



- End of Expected Result -

- Notes for Example 1.4 -

Check the following Notes in order to complete the Example:

Note 1: The command to spawn the object in the simulation is just a way to introduce a new obstacle to the simulation. It has NOTHING to do with Obstacle Avoidance. Included here for teaching purposes only. If you want to learn more about how to affect the simulation, enroll our course **Gazebo in 5 days**.

Note 2: Use the **2D Nav Goal** to send the robot to different places of the Caffe. Select a destination point that requires the robot to avoid the newly inserted object, so you can see how the path is being modified by the obstacle and how the robot is avoiding it and not colliding with it.

IMPORTANT: When you have finished with this exercise, make sure to remove again the object inserted by using the following command:

In [ ]:
```
rosservice call /gazebo/delete_model "model_name: 'my_object'"
```

- End of Notes -

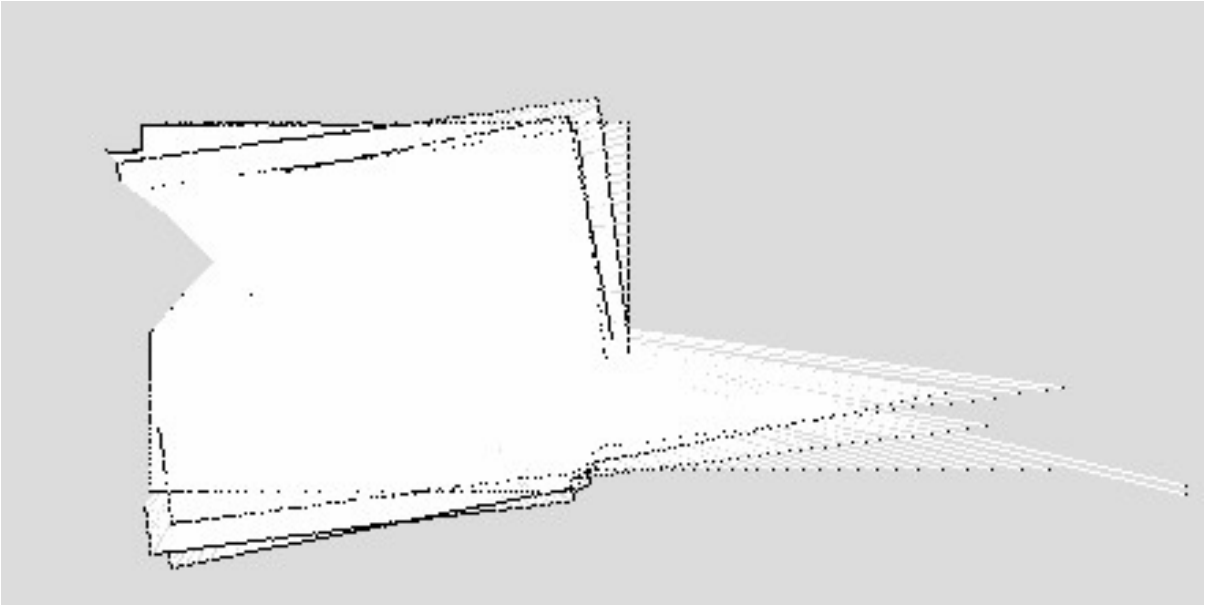
At this point we both know what you are thinking, and what I'm going to say... so let's make it easy. See you back in **Path Planning Part 2 (Chapter 5)** with more details about obstacle avoidance!

So... what do you think? By now, you've already gone through different examples of each one of the parts involved in ROS Navigation. But... Do you think this whole system would work right out of the box? For any kind of robot? The answer is **NO**. It won't.

In order to have the ROS Navigation Stack working properly, you need to provide some data to the system , depending on which is the robot you want to use and how it is built. That is, you need to have your robot properly configured.

Robot Configuration is extremely important in all the navigation modules. For instance, in the Mapping system, if you don't tell the system WHERE does your robot have the laser mounted on, which is the laser's orientation, which is the position of the wheels in the robot, etc., it won't be able to create a good and accurate Map. And as you may already know at this point, **if we don't have a good Map in ROS Navigation, we have NOTHING!**

The following image is an example of how a Map file would look like, if the laser of the robot is not properly configured:



1.2 How to Configure your Robot

Robot configuration and definition is done in the **URDF** files of the robot. URDF (Unified Robot Description Format) is an XML format that describes a robot model. It defines its different parts, dimensions, kinematics, dynamics, sensors, etc...

Every time you see a 3D robot on ROS, a URDF file is associated with it.

For instance, let's have a look at how the laser of the Kobuki robot is defined in the URDF file of the robot:

In [ ]:

```
<joint name="laser_sensor_joint" type="fixed">
    <origin xyz="0.0 0.0 0.435" rpy="0 0 0"/>
    <parent link="base_link"/>
    <child link="laser_sensor_link"/>
</joint>

<link name="laser_sensor_link">
        <inertial>
                <mass value="1e-5"/>
                <origin xyz="0 0 0" rpy="0 0 0"/>
                <inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e
        </inertial>
        <collision>
                <origin xyz="0 0 0" rpy="0 0 0"/>
                <geometry>
                        <box size="0.1 0.1 0.1"/>
                </geometry>
        </collision>
        <visual>
                <origin xyz="0 0 0" rpy="0 0 0"/>
                <geometry>
                        <mesh filename="package://hokuyo/meshes/hokuyo.dae"/>
                </geometry>
        </visual>
</link>
```

IMPORTANT: The XML code shown above is just a section of the URDF file that defines the Kobuki robot. In this case, it's the section where the laser is defined.

As you can see, it's defining several things regarding the laser:

- It defines the position and orientation of the laser regarding the base ("base_link") of the robot.
- It defines inertia related values
- It defines collision related values. These values will provide the real physics of the laser.
- It defines visual values. These values will provide the visual elements of the laser. This is just for visualization purposes.

These files are usually placed into a package named ***yourrobot*_description.**

In this Course, though, you won't be covering URDF files. I've introduced you to them because it is important that you know about how robot configuration and definition is handled in ROS, but you won't go any further. If you are interested in learning more about this topic, you can have a look at the **Robot Creation with URDF** (https://www.robotigniteacademy.com/en/course/robot-creation-with-urdf-ros/details/) Course of the **Robot Ignite Academy**.

But wait!! Don't get too excited yet. You may not be covering this topic now, but you will still have to handle other configuration issues during this Course. I'm talking about the configuration of the many parameters that take place during the Navigation process. These parameters will allow you to modify the behavior of the different phases involved in the Navigation process (such as Mapping, Localization, etc.). But do not worry for now, you'll learn to configure the navigation system while you progress through the Chapters of this Course, as well as to use some handy configuration tools.

For now, just remember this one thing: "With great Power comes great Responsibility". Oops!! I think I'm starting to mix things... I ment: "**With great Configuration comes great Navigation**".

So... what do you think? You've already seen (overviewed) the main Basic Concepts you need to know about ROS Navigation. And you may be tempted to think you already know everything you need in order to impress your boss... but let me tell you that you're very WRONG. You still have a lot to learn, my young Padawan.

This was just an **introduction** to the different Units you're going to see during this course, but it's just a glimpse. You haven't even finished with the Basic Concepts Chapter!! So stop daydreaming and get back to work!

1.3 The Navigation Stack

During the previous examples, you've been launching many different ROS nodes. Each one of these nodes launched their own ROS programs in order to execute different tasks. And all of these was contained, as always happens in ROS, in packages. But... where did all of these packages came from? What are they? Are they connected between them or are they totally independent?

As you may have already figured out, you've been using what's known in ROS as **The Navigation Stack**.

The Navigation Stack is a **set of ROS nodes and algorithms** which are used to autonomously move a robot from one point to another, avoiding all obstacles the robot might find in its way. The ROS Navigation Stack comes with an implementation of several navigation related algorithms which can help you perform autonomous navigation in your mobile robots.

The Navigation Stack will **take as input** the **current location of the robot**, the **desired location the robot wants to go (goal pose)**, the **Odometry data** of the Robot (wheel encoders, IMU, GPS...) and **data from a sensor such as a Laser**. In exchange, it will **output** the necessary **velocity commands** and send them to the mobile base in order to **move the robot to the specified goal position**.

Summarizing, we can say that **the main objective of the Navigation Stack is to move a robot from a position A to a position B, assuring it won't crash against obstacles, or get lost in the process**.

1.4 Hardware Requirements

The ROS Navigation Stack is generic. That means, it can be used with almost any type of moving robot, but there are some hardware considerations that will help the whole system to perform better, so they must be considered. These are the requirements:

- The Navigation package will work better in differential drive and holonomic robots. Also, the mobile robot should be controlled by sending velocity commands in the form:
  x, y (linear velocity)
  z (angular velocity)
- The robot should mount a planar laser somewhere around the robot. It is used to build the map of the environment and perform localization.
- Its performance will be better for square and circular shaped mobile bases.

Following there is a figure with the basic building blocks of the Navigational stack taken from the ROS official website (http://wiki.ros.org/navigation/Tutorials/RobotSetup (http://wiki.ros.org/navigation/Tutorials/RobotSetup)). Maybe this doesn't make much sense to you right now, but by the end of this Course, you will be fully capable to understand this diagram and each block it forms part of it. For now, just try to get a global idea.

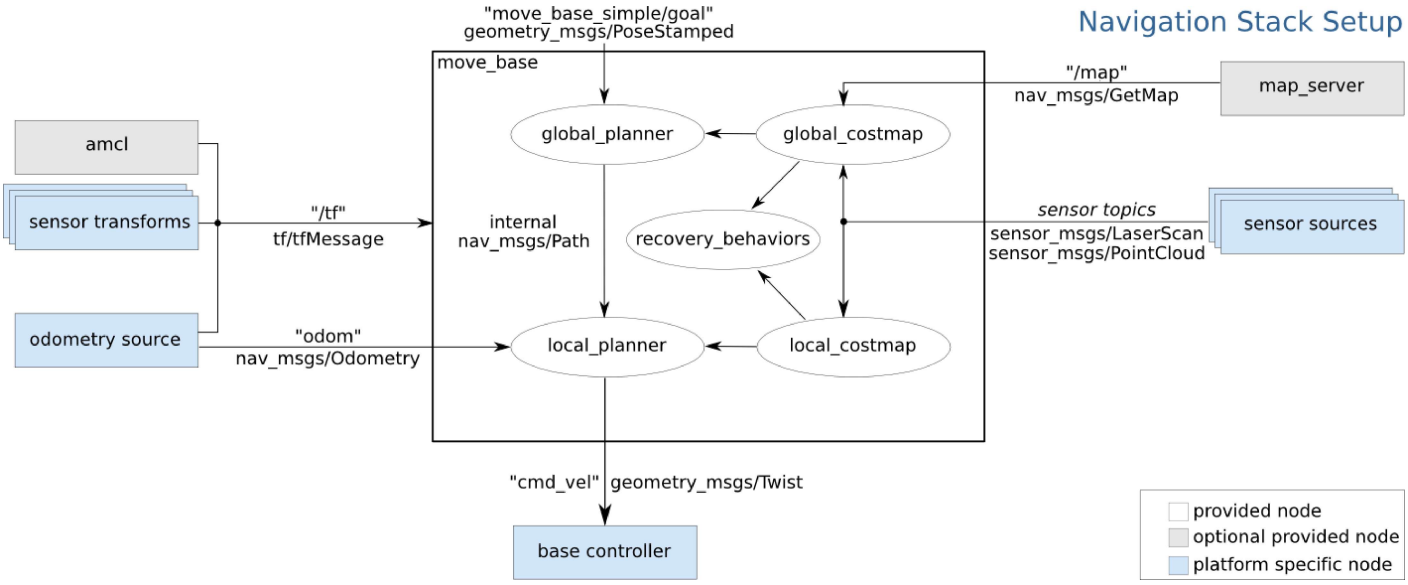


Fig.1.1 - Navigation Stack Setup (figure from ROS Wiki)

According to the shown diagram, we must provide some functional blocks in order to work and communicate with the Navigation stack. Following are brief explanations of all the blocks which need to be provided as input to the ROS Navigation stack:

- **Odometry source**: Odometry data of a robot gives the robot position with respect to its starting position. Main odometry sources are wheel encoders, IMU, and 2D/3D cameras (visual odometry). The odom value should publish to the Navigation stack, which has a message type of nav_msgs/ Odometry. The odom message can hold the position and the velocity of the robot.
- **Sensor source**: Sensors are used for two tasks in navigation: one for localizing the robot in the map (using for example the laser) and the other one to detect obstacles in the path of the robot (using the laser, sonars or point clouds).
- **sensor transforms/tf**: the data captured by the different robot sensors must be referenced to a common frame of reference (usually the ***base_link***) in order to be able to compare data coming from different sensors. The robot should publish the relationship between the main robot coordinate frame and the different sensors' frames using ROS transforms.
- **base_controller**: The main function of the base controller is to convert the output of the Navigation stack, which is a Twist (**geometry_msgs/Twist**) message, into corresponding motor velocities for the robot.

- Exercise 1.1 -

Check that the system you're working in fullfils these requirements.

Execute the following command in the WebShell number #1 in order to get a list of all the topics running in the system.

▶ Execute in Shell #1

In [ ]:
```
rostopic list
```

Look for the following topics:

`/cmd_vel`

`/kobuki/laser/scan`

`/odom`

`/tf`

Can you find them? Are they actives? Great! Each one of this topics has its own functionality in the Navigation process. Can you guess the purpose of each of these topics?

Try to get all the information you can about this topics, since you'll need to know them well.

▶ Execute in Shell #1

In [ ]:
```
rostopic info /cmd_vel
rostopic info /odom
rostopic info /kobuki/laser/scan
rostopic info /tf
```

Get also information about the messages they use.

▶ Execute in Shell #1

In [ ]:
```
rosmsg show geometry_msgs/Twist
rosmsg show nav_msgs/Odometry
rosmsg show sensor_msgs/LaserScan
rosmsg show tf2_msgs/TFMessage
```

- End of Exercise 1.1 -

- Expected Result for Exercise 1.1 -

Info /cmd_vel:

```
user ~ $ rostopic info /cmd_vel
Type: geometry_msgs/Twist

Publishers: None

Subscribers:
 * /gazebo (http://ip-172-31-34-208:48232/)
```

Info /odom:

```
user ~ $ rostopic info /odom
Type: nav_msgs/Odometry

Publishers:
 * /gazebo (http://ip-172-31-34-208:48232/)

Subscribers: None
```

Info /kobuki/laser/scan:

```
user ~ $ rostopic info /kobuki/laser/scan
Type: sensor_msgs/LaserScan

Publishers:
 * /gazebo (http://ip-172-31-34-208:48232/)

Subscribers: None
```

Info /tf:

```
user ~ $ rostopic info /tf
Type: tf2_msgs/TFMessage

Publishers:
 * /robot_state_publisher (http://ip-172-31-34-208:48020/)
 * /gazebo (http://ip-172-31-34-208:48232/)

Subscribers: None
```

Twist message:

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

Odometry message:

```
user ~ $ rosmsg show nav_msgs/Odometry
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
  float64[36] covariance
```

LaserScan message:

```
user ~ $ rosmsg show sensor_msgs/LaserScan
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

tfMessage message:

```
user ~ $ rosmsg show tf/tfMessage
geometry_msgs/TransformStamped[] transforms
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  string child_frame_id
  geometry_msgs/Transform transform
    geometry_msgs/Vector3 translation
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion rotation
      float64 x
      float64 y
      float64 z
      float64 w
```

- **/cmd_vel**: Receives the output of the Navigation Stack and transforms the commands into motor velocities.
- **/kobuki/laser/scan**: Provides the Laser readings to the Stack.
- **/odom**: Provides the Odometry readings to the Stack.
- **/tf**: Provides the Transformations to the Stack.

- End of Expected Result -

1.5 The move_base node

This is the most important node of the Navigation Stack. It's where most of the "magic" happens.

The main function of the **move_base node** is to **move a robot from its current position to a goal position** with the help of other Navigation nodes. This node links the global planner and the local planner for the path planning, connecting to the rotate recovery package if the robot is stuck in some obstacle, and connecting global costmap and local costmap for getting the map of obstacles of the environment.

Following is the list of all the packages which are linked by the move_base node:

- **global-planner**
- **local-planner**
- **rotate-recovery**
- **clear-costmap-recovery**
- **costmap-2D**

Following are the other packages which are interfaced to the move_base node:

- **map-server**
- **AMCL**
- **gmapping**

Don't panic! Remember this is just an introduction to the Basic Concepts of ROS Navigation. We'll have a deeper look at all this elements in further Units. For now, just try to get a general idea of how the ROS Navigation Stack works. Try to get familiar with some of the names you read, since you'll find out very soon that some of them are very important!

1.6 Summary

The Navigation Stack is a set of ROS nodes an algorithms, which work together in order to move a robot from a position A to a position B, avoiding the obstacles it may find in its way. In order to do this, the Navigation Stack requires many data inputs (of different kinds). In exchange, it will give as an output the necessary command velocities in order to safely move the robot to the desired position.