## Introduction

The Rodney Robot project is a hobbyist robotic project to design and build an autonomous house-bot using ROS (Robot Operating System). This article is the seventh in the series describing the project.

## Background

In [part 1](https://www.codeproject.com/Articles/1249436/Rodney-A-long-time-coming-autonomous-robot-Part-1), to help define the requirements for our robot, we selected our first mission and split it down into a number of Design Goals to make it more manageable.

The mission was taken from the article, [Let's build a robot!](https://www.codeproject.com/Articles/1115414/Lets-build-a-robot) and was: Take a message to... - Since the robot will [have] the ability to recognize family members, how about the ability to make it the 'message taker and reminder'. I could say 'Robot, remind (PersonName) to pick me up from the station at 6pm'. Then, even if that household member had their phone turned on silent, or was listening to loud music or (insert reason to NOT pick me up at the station), the robot could wander through the house, find the person, and give them the message.

The design goals for this mission were:

* Design Goal 1: To be able to look around using the camera, search for faces, attempt to identify any people seen and display a message for any identified
* Design Goal 2: Facial expressions and speech synthesis. Rodney will need to be able to deliver the message
* Design Goal 3: Locomotion controlled by a remote keyboard and/or joystick
* Design Goal 4: Addition of a laser ranger finder or similar ranging sensor used to aid navigation
* Design Goal 5: Autonomous locomotion
* Design Goal 6: Task assignment and completion notification

In the last part, we added a spinning LIDAR (light detection and ranging) to complete Design Goal 4 and added an IMU to improve the odometry. In this part, I'm going to make use of the ROS Navigation Stack to finally give Rodney Autonomous locomotion. This will include existing ROS packages for SLAM (simultaneous localization and mapping), a probabilistic localization system,  global and local navigational planning, all to put us well on our way to completing Design Goal 5.

## Navigation "Stack"

In older versions of ROS, packages could be organised into ROS stacks. This is an outdated but sometimes you will still see the term used especially the "Nav Stack". This now refers to a number of packages that can be used together for autonomous navigation. In this article we will be not be writing any new code but using existing ROS packages. I'll explain how to configure and launch these packages, however some of them contain many configurable parameters and you should refer to the ROS Wiki for each package to get an understanding of these parameters.

### Mapping

In order for our robot to navigate it will need a map of its world. The node we are going to use to create the map will be run on a Linux workstation and constructs the map from data recorded from the robot sensors. The ROS package is called [gmapping](http://wiki.ros.org/gmapping) and provides laser-based SLAM (Simultaneous Localization and Mapping). The following image shows a map created by gmapping from Rodney's sensors displayed in rviz.

[??? MAP]

The map is stored for later use in two files, a .yaml and a .pgm file. The .pgm file can be edited in a number of image editing tools so you can always tidy up the map and block off areas you may not wish your robot to visit.

You can create maps live but most tutorials will tell you to record the sensor data and then produce the map from the recorded data. This allows you to try different parameters settings when producing the map. I prefer a mixture of the two, record the data but also visualise the map in rviz as it is created. This allows you to ensure you have not missed part of the map you really need.

You create the map by driving your robot slowly in manual mode. To improve the map quality as well as driving slowly it is best to visit a location more than once. I'll show how to record the data and how to create and save the map in the "Using the Code" section.

Once we have a map the robot will need access to the map and as usual with ROS this is done using topics. The node providing these topics is part of the [map\_server](http://wiki.ros.org/map_server) package. Since the map may be large the topic containing the map is not continually published, but is latched and any new nodes requiring the map will be passed a copy. As well as containing the map\_server node the package includes another node called map\_saver which will be used to save the map created by gmapping to disk.

To launch the map\_server node on the robot hardware we will include the following in the *rodney.launch*file.

Hide   Copy Code

<arg name="map\_file" default="second\_floor"/>

<node pkg="map\_server" type="map\_server" name="map\_server" args="$(find rodney)/maps/$(arg map\_file).yaml" output="screen"/>

This implies that we will store our map files in a new folder in the rodney package called maps. If when we call the launch file we don't provide a map\_file parameter the default value of second\_floor will be used.

### Robot Localisation

The next part of this navigation jig-saw puzzle we require is Robot Localisation. As well as keeping track of where it is in the world, the robot will also need to know its starting place. Now the odom and laser sensors are not perfect so the robot can't tell exactly where it is but will maintain an array of candidate poses of possible locations. A pose is a position and orientation in the world. As the robot moves and it narrows down where it thinks it is, it will discard the poses that are less likely to be its position.

For this functionality we will use the [amcl package](http://wiki.ros.org/amcl). AMCL stands for Adaptive Monte Carlo Localization, luckily all we need to do is to configure it and launch it, if you wish to understand it more then may be this [Wikipedia page](https://en.wikipedia.org/wiki/Monte_Carlo_localization) is a good starting place.

As is normal with ROS we will configure the node by loading some configuration data onto the parameter server. This data will be stored in the *amcl\_config.yaml*file stored in the *rodney/config* folder. This is my version of the configuration file but as with these navigation packages they are highly configurable so consult the ROS Wiki for the package.

Hide   Shrink Image 1 for Rodney - A Long Time Coming Autonomous Robot (Part 7)   Copy Code

# Overall filter parameters

min\_particles: 500

max\_particles: 3000

kld\_err: 0.05

kld\_z: 0.99

update\_min\_d: 0.2

update\_min\_a: 0.5

resample\_interval: 1

transform\_tolerance: 0.5

recovery\_alpha\_slow: 0.0

recovery\_alpha\_fast: 0.0

gui\_publish\_rate: 1.0

# Laser model parameters

laser\_max\_beams: 30

laser\_z\_hit: 0.5

laser\_z\_short: 0.05

laser\_z\_max: 0.05

laser\_z\_rand: 0.5

laser\_sigma\_hit: 0.2

laser\_lambda\_short: 0.1

laser\_likelihood\_max\_dist: 2.0

laser\_model\_type: likelihood\_field

# Odometry model parameters

odom\_model\_type: diff

odom\_alpha1: 0.2

odom\_alpha2: 0.2

odom\_alpha3: 0.8

odom\_alpha4: 0.2

odom\_alpha5: 0.1

odom\_frame\_id: odom

base\_frame\_id: base\_footprint

To launch this node with the configuration given we will add the following to the rodney.launch file.

Hide   Copy Code

<node pkg="amcl" type="amcl" name="amcl" output="screen">

<rosparam command="load" file="$(find rodney)/config/amcl\_config.yaml"/>

</node>

It's worth noting that most of the parameters I have changed from defaults for the nav stack relate to transform tolerances and sample frequencies, we are asking the Raspberry Pi to a lot of complex calculations in a short time here!

### Planning a Route

Now we are getting to the business end of the process. If we want to get from point A to point B (or pose A to pose B) using the map something needs to plan the route and command the robot to move. For this we are going to use the [move\_base](http://wiki.ros.org/move_base) package. This package is going to have to use the amcl data, the map, the odom data and the laser data to plan the best route. Not only that, but it needs to allow for things that have changed since the map was constructed. If a person or a family pet walks into the path of the robot it will need to come up with a new route. Again to gain some understanding of this package read its ROS Wiki page.

When planning the route the package will use two costmaps. A costmap shows good and bad places for a robot to be. A good place is out in the open and a bad place would be against a wall. One costmap is the Global costmap and is based on the map and remains static. This map is used to plan the global route, but as said before what if things have changed from the map. This is where the Local costmap comes in. It's updated as the robot moves and whereas the Global costmap covers the entire map the Local costmap covers only the immediate location around the robot.

The package move\_base can use a number of global and local planners that are available. As long as it adheres to the nav\_core::BaseGlobalPlanner interface for global planning and nav\_core::BaseLocalPlannerinterface for local planning, you can even write your own planners.

There are default planners you can use or you can declare which planners you are using in the configuration files. I'm going to use [global\_planner](http://wiki.ros.org/global_planner) and  and [dwa\_local\_planner](http://wiki.ros.org/dwa_local_planner).

We need a number of configuration files for the move\_base package which will be loaded into the parameter sever. Each file will be located in the rodney/config folder.

base\_local\_planner\_params.yaml

Hide   Shrink Image 2 for Rodney - A Long Time Coming Autonomous Robot (Part 7)   Copy Code

controller\_frequency: 5.0

use\_grid\_path: true

base\_global\_planner: global\_planner/GlobalPlanner

base\_local\_planner: dwa\_local\_planner/DWAPlannerROS

DWAPlannerROS:

acc\_lim\_x: 0.75

acc\_lim\_y: 0.0

acc\_lim\_theta: 1.75 # wiki calls it acc\_lim\_th but it is acc\_lim\_theta

acc\_lim\_trans: 1.0

max\_trans\_vel: 0.25

min\_trans\_vel: 0.1

max\_vel\_x: 0.25 # The maximum foward velocity

min\_vel\_x: -0.1

max\_vel\_y: 0.0

min\_vel\_y: 0.0

max\_rot\_vel: 2.5

min\_rot\_vel: 0.3

yaw\_goal\_tolerance: 0.3

xy\_goal\_tolerance: 0.25

sim\_time: 2.0

sim\_granularity: 0.1

vx\_samples: 3

vy\_samples: 10

vth\_samples: 20

path\_distance\_bias: 32.0

goal\_distance\_bias: 24.0

occdist\_scale: 0.05

forward\_point\_distance: 0.3

stop\_time\_buffer: 0.5

scaling\_speed: 0.25

max\_scaling\_factor: 0.2

oscillation\_reset\_dist: 0.05

prune\_plan: true

trans\_stopped\_vel: 0.1

rot\_stopped\_vel: 0.1

angular\_sim\_granularity: 0.1

twirling\_scale: 0.0

oscillation\_reset\_angle: 0.2

use\_dwa: true

Here we set the planners we are going to use and a large number of parameters for the local planner. Again refer to the relevant Wiki for each parameter. Most of the parameters can be set dynamically using the dynamic reconfiguration which we have used in the previous articles on Rodney. Values are also very particular to each robot and its environment. For example at one point I increased the maximum speed and rotational velocity thinking that it would be able to turn quicker around tight corners, but ended up with the robot getting stuck up against objects. It would appear that by going faster the robot got it's self into problem areas before it had time to calculate it was a bad area.

The costmaps have some common parameters between the global and local costmaps and some parametsr which are particular to either the global or local costmap.

The common parameters are stored in the *costmap\_common\_params.yaml* file:

Hide   Copy Code

obstacle\_range: 2.5

raytrace\_range: 3.0

footprint: [[0.170, 0.145], [-0.170, 0.145], [-0.170, -0.145], [0.170, -0.145]] # Simple rectangle taking in the base and wheels

inflation\_radius: 0.55

observation\_sources: laser\_scan\_sensor

laser\_scan\_sensor:

sensor\_frame: laser

data\_type: LaserScan

topic: scan

marking: true

clearing: true

For the footprint I started with a complex shape which took into account the wheels sticking out from the base. But in order to make it less complex for the calculations I changed it to a simple rectangle which included the base and the wheels.

The parameters that a particular to the global costmap are stored in the global\_costmap\_params.yaml file:

Hide   Copy Code

global\_costmap:

global\_frame: map

robot\_base\_frame: base\_footprint

static\_map: true

transform\_tolerance: 0.5

update\_frequency: 1.0

Those that are particular to the local costmap are stored in the local\_costmap\_params.yaml file:

Hide   Copy Code

local\_costmap:

global\_frame: odom

robot\_base\_frame: base\_footprint

static\_map: false

rolling\_window: true

transform\_tolerance: 0.5

update\_frequency: 1.0

publish\_frequency: 0.5

width: 2.5

height: 2.5

Again to help the calculations I'm using a relatively small local costmap of 2.5x2.5 metres around the robot. Remember the local costmap is used to plan the route around objects that may have moved since the static map, which the global costmap is based on, was created. Increasing the width and height allows the robot to plan further ahead but at the expense of number of calculations the processor will need to carry out.

It's probably worth noting at this point that although I did have the navigation working on the Raspberry Pi 3B I have upgraded to a Raspberry Pi 3B+ to take advantage of the slightly faster processor.

To launch the move\_base node and to load all the parameters add the following to the rodney.lunch file:

Hide   Copy Code

<node pkg="move\_base" type="move\_base" name="move\_base" respawn="false" output="screen">

<rosparam command="load" file="$(find rodney)/config/base\_local\_planner\_params.yaml"/>

<rosparam command="load" file="$(find rodney)/config/costmap\_common\_params.yaml" ns="global\_costmap"/>

<rosparam command="load" file="$(find rodney)/config/costmap\_common\_params.yaml" ns="local\_costmap"/>

<rosparam command="load" file="$(find rodney)/config/global\_costmap\_params.yaml"/>

<rosparam command="load" file="$(find rodney)/config/local\_costmap\_params.yaml"/>

<remap from="cmd\_vel" to="demand\_vel"/>

</node>

The published topic from move\_base is normally cmd\_vel, which is the topic name which our thunderborg node subscribes to. Since we wish to be able to drive the robot manually our rodney node was set up to publish on the cmd\_vel topic either data for manual driving or autonomous depending on the current mode of the robot. We therefore remap the topic name here from cmd\_vel to demand\_vel.

One other change I want to make to the rodney.launch file, is to be able to launch the software either with the nav stack enabled or not. The default will be to have the nav stack enabled but if we want to drive the robot remotely in order to produce a new map we don't want to run the map server and we don't want to launch the other nodes that will be looking for a map. I have therefore wrapped the the sections for launching the various parts of the nav stack in "group unless" tag.

The complete rodney.launch file is reproduced below:

Hide   Shrink Image 3 for Rodney - A Long Time Coming Autonomous Robot (Part 7)   Copy Code

<?xml version="1.0" ?>

<launch>

*<!-- Static transforms in the system -->*

<node pkg="rodney" type="static\_broadcaster.py" name="static\_broadcaster\_node"/>

*<!-- Load each of the config files into the parameter server -->*

<rosparam command="load" file="$(find pan\_tilt)/config/config.yaml"/>

<rosparam command="load" file="$(find face\_recognition)/config/config.yaml"/>

<rosparam command="load" file="$(find head\_control)/config/config.yaml"/>

<rosparam command="load" file="$(find rodney\_missions)/config/config.yaml"/>

*<!-- map and localization system -->*

<arg name="no\_nav" default="false"/>

<group unless="$(arg no\_nav)">

<arg name="map\_file" default="second\_floor"/>

<node pkg="map\_server" type="map\_server" name="map\_server" args="$(find rodney)/maps/$(arg map\_file).yaml" output="screen"/>

<node pkg="amcl" type="amcl" name="amcl" output="screen">

<rosparam command="load" file="$(find rodney)/config/amcl\_config.yaml"/>

</node>

</group>

*<!-- Launch the camera node from one of its launch files -->*

<include file="$(find raspicam\_node)/launch/camerav2\_1280x960.launch" />

*<!-- Start all the nodes that make up Rondey -->*

*<!-- Starting with those written for the project -->*

<node pkg="pan\_tilt" type="pan\_tilt\_node" name="pan\_tilt\_node"/>

<node pkg="face\_recognition" type="face\_recognition\_node.py" name="face\_recognition\_node"/>

<node pkg="head\_control" type="head\_control\_node" name="head\_control\_node"/>

<node pkg="speech" type="speech\_node" name="speech\_node"/>

<node pkg="rodney\_missions" type="rodney\_missions\_node.py" name="rodney\_missions" output="screen"/>

<node pkg="rodney" type="rodney\_node" name="rodney" output="screen">

<rosparam command="load" file="$(find rodney)/config/config.yaml"/>

</node>

<node pkg="thunderborg" type="thunderborg\_node.py" name="thunderborg\_node">

<rosparam command="load" file="$(find thunderborg)/config/config.yaml"/>

</node>

*<!-- Teensy.*

*Use the defaults /dev/ttyACM0 (or teensy if dev rules updated) and 500000 -->*

<arg name="serial\_port" default="/dev/teensy"/>

<arg name="baud\_rate" default="500000"/>

<node pkg="rosserial\_python" type="serial\_node.py" name="serial\_node" output="screen">

<param name="port" value="$(arg serial\_port)"/>

<param name="baud" value="$(arg baud\_rate)"/>

</node>

*<!-- The RPLidar and laser filter node*

*Have created symbolic link for /dev/ttyUSBn to be rplidar -->*

<node pkg="rplidar\_ros" type="rplidarNode" name="rplidar\_node" output="screen">

<param name="serial\_port" type="string" value="/dev/rplidar"/>

<param name="serial\_baudrate" type="int" value="115200"/>

<param name="frame\_id" type="string" value="laser"/>

<remap from="scan" to="scan\_filter\_input"/>

</node>

<node pkg ="laser\_filters" type="scan\_to\_scan\_filter\_chain" name="scan\_to\_scan\_filter\_chain" output="screen">

<rosparam command="load" file="$(find rodney)/config/laser\_filter\_config.yaml"/>

<remap from="scan" to="scan\_filter\_input"/>

<remap from="scan\_filtered" to="scan"/>

</node>

*<!-- The robot face -->*

<node pkg="homer\_robot\_face" type="RobotFace" name="RobotFace"/>

*<!-- Add calibration to raw imu data -->*

<node pkg="imu\_calib" type="apply\_calib" name="imu\_calib" output="screen">

<param name="calib\_file" value="$(find rodney)/config/imu\_calib.yaml"/>

</node>

*<!-- Node to fuse motor encoder and IMU data for odom -->*

<node pkg="robot\_localization" type="ekf\_localization\_node" name="ekf\_localization\_node">

<remap from="odometry/filtered" to="odom"/>

<rosparam command="load" file="$(find rodney)/config/robot\_localization.yaml"/>

</node>

*<!-- Navigation -->*

<group unless="$(arg no\_nav)">

<node pkg="move\_base" type="move\_base" name="move\_base" respawn="false" output="screen">

<rosparam command="load" file="$(find rodney)/config/base\_local\_planner\_params.yaml"/>

<rosparam command="load" file="$(find rodney)/config/costmap\_common\_params.yaml" ns="global\_costmap"/>

<rosparam command="load" file="$(find rodney)/config/costmap\_common\_params.yaml" ns="local\_costmap"/>

<rosparam command="load" file="$(find rodney)/config/global\_costmap\_params.yaml"/>

<rosparam command="load" file="$(find rodney)/config/local\_costmap\_params.yaml"/>

<remap from="cmd\_vel" to="demand\_vel"/>

</node>

</group>

</launch>

Well that's it for now. In the next article we will changing the rodney\_missions node to programmatically run autonomous navigation and combine it with the face recognition functionality to wander through the house searching for a known person to deliver the message to.

In the "Using the Code" section we will create a map and use rviz to input poses which the robot will autonomously navigate to.

## Robot Hardware

A full size image of the current circuit diagram is available in the diagrams zip folder along with a full size copy of the image from rqt\_graph showing all the nodes and topics.

A complete bill of material for the project so far is available here ???.

In part 1 of the article, I referenced the Ubiquity Robot Image which I use on the Raspberry Pi. Instructions on how to install the image, install extra software and configure it for the project are available here ???.

## Using the Code

As usual, I'll run the code on the robot hardware and run the test tools and manual control nodes on a Linux PC. I'll refer to this PC as the workstation in the details below.

### Building the ROS Packages on the Pi (Robot Hardware)

If not already done, create a catkin workspace on the Raspberry Pi and initialise it with the following commands:

Hide   Copy Code

$ mkdir -p ~/rodney\_ws/src

$ cd ~/rodney\_ws/

$ catkin\_make

Copy the packages face\_recognition, face\_recognition\_msgs, head\_control, imu\_calib, pan\_tilt, rodney, rodney\_missions, ros-keyboard, rplidar-ros, servo\_msgs, speech, tacho\_msgs and thunderborg into the ~/rodney\_ws/src folder.

Build the code with the following commands:

Hide   Copy Code

$ cd ~/rodney\_ws/

$ catkin\_make

Check that the build completes without any errors.

You will also need to compile and download the sketch to the Teensy 3.5.

### Building the ROS Packages on the Workstation

On the workstation, we want to run the keyboard, joystick and heartbeat nodes so that we can control the actual robot hardware remotely.

Create a workspace with the following commands:

Hide   Copy Code

$ mkdir -p ~/test\_ws/src

$ cd ~/test\_ws/

$ catkin\_make

Copy the packages rodney, joystick, odom\_test and ros-keyboard into the ~/test\_ws/src folder, and then build the code with the following commands:

Hide   Copy Code

$ cd ~/test\_ws/

$ catkin\_make

Check that the build completes without any errors.

### Tip

When running ROS code and tools on a workstation and the Raspberry Pi, there can be a lot of repeat typing of commands at a number of terminals. In the next sections, I have included the full commands to type but here are a few tips that can save you all that typing.

On the Raspberry Pi, to save typing "source devel/setup.bash", I have added it to the .bashrc file for the Raspberry Pi.

Hide   Copy Code

$ cd ~/

$ nano .bashrc

Then add "source /home/ubuntu/rodney\_ws/devel/setup.bash" to the end of the file, save and exit.

When running test code and tools on the workstation, it also needs to know where the ROS master is so I have added the following to the .bashrc file for the workstation.

Hide   Copy Code

alias rodney='source ~/test\_ws/devel/setup.bash; \

export ROS\_MASTER\_URI=http://ubiquityrobot:11311'

Then by just typing "rodney" at a terminal, the two commands are run and a lot of typing is saved.

You can also save some typing as some ROS tools support TAB completion. For example, type rosrun rosserial\_ and then press the tab key to auto complete rosrun rosserial\_python.

### Watching the Log File

As the robot navigates around you may like to keep an eye on log messages. These will be displayed in the terminal that you use to launch the code from but that's not helpful if the robot is movining and in a different room. You can view the log at the workstation with th following commands:

Hide   Copy Code

$ export ROS\_MASTER\_URI=http://ubiquityrobot:11311

$ rqt\_console

[??? image of rqt\_console]

### Creating a Map

On the robot hardware, run the following commands to start all the current nodes in the system with the exception of the navigation stack:

Hide   Copy Code

$ source rodney\_ws/devel/setup.bash

$ roslaunch rodney rodney.launch no\_nav:=True

On the workstation, run the following commands to start the remote control nodes:

Hide   Copy Code

$ source test\_ws/devel/setup.bash

$ export ROS\_MASTER\_URI=http:*//ubiquityrobot:11311*

$ roslaunch rodney remote.launch

A small window whose title is "**ROS keyboard input**" should be running. When entering keyboard strokes ensure the small window has the focus.

Next we will start recording the transforms and laser scan messages so that we can show how to create a map from recorded data. In a terminal start the recording with the following commands:

Hide   Copy Code

$ export ROS\_MASTER\_URI=http:*//ubiquityrobot:11311*

$ rosbag record -O data.bag /scan /tf

Here we are going to start slam\_gmapping so that we can see a live map being created. I'll also limit the size of the map. In a terminal on the workstation run the following commands:

Hide   Copy Code

$ export ROS\_MASTER\_URI=http:*//ubiquityrobot:11311*

$ rosparam set slam\_gmapping/xmax 10

$ rosparam set slam\_gmapping/ymax 10

$ rosparam set slam\_gmapping/xmin -10

$ rosparam set slam\_gmapping/ymin -10

$ rosparam set slam\_gmapping/delta 0.05

$ rosrun gmapping slam\_gmapping

As an alternative I have created a test package called mapping\_launch that contains a launch file called mapping.launch which will set the parameters and start gmapping. This package is available in the Robotics-test-code folder.

In another terminal on the workstation start rviz with the following commands:

Hide   Copy Code

$ source test\_ws/devel/setup.bash

$ export ROS\_MASTER\_URI=http:*//ubiquityrobot:11311*

$ roslaunch rodney rviz.launch

Configure rviz so that:

* Fixed frame is map
* LaserScan is displaying the /scan topic
* TF is displaying the base\_link
* Map is displaying the /map topic

Using the joystick and/or keyboard enter manual mode, ensure the LIDAR motor is running and manually drive the robot around its world. Go slow and visit each location at least twice. The map being created should be visible on rviz.

[??? IMAGE OF PART OF A MAP]

Once you have created the map press **Ctrl-C** in the terminal running rosbag to finish storing the message.

We can now save the map that is visible in rviz, to disk. With slam\_gmapping still running enter the following command in the terminal that was running rosbag:

Hide   Copy Code

$ rosrun map\_server map\_saver -f my\_first\_map

This will result in two files been saved, my\_first\_map.yaml and my\_first\_map.pgm

You can at this point if you wish regenerate the map from the rosbag file with different gmapping parameters. Here we don't want any of the existing nodes running on the workstation so shutdown and close all the terminals. You can also shutdown the robot.

At the workstation we need a ROS master running (in the previous setup this was running automatically on the robot hardware), enter the following in a terminal:

Hide   Copy Code

$ roscore

In another terminal use rosparam to set the required gmapping parameters and then type the following:

Hide   Copy Code

$ rosparam set use\_sim\_time\_true

$ rosrun gmapping slam\_gmapping

In another terminal playback the rosbag recorded previously with the following command:

Hide   Copy Code

$ rosbag play --clock data.bag

Then sit back while gmapping creates the map. If you wish you could start rviz (without the export ROS\_MASTER\_URI command) and watch the map being created. Once all the messages have been played back store the new map to disk with the following command:

Hide   Copy Code

$ rosrun map\_server map\_saver -f my\_second\_map

You can play with the gmapping configuration parameters here as much as you like and keep playing back the recoded bag file to see the differences the parameters have on the generated map.

The following video shows map creation.

[Video of map creattion ???]

### Autonomous Navigation

Now the bit we have all been waiting for, autonomous navigation.

On the robot hardware, run the following commands to start all the current nodes in the system including the navigation stack. I'm going to use my default map but you can set the map by adding "map\_file:=my\_first\_map" to the end of the roslaunch command.

Hide   Copy Code

$ source rodney\_ws/devel/setup.bash

$ roslaunch rodney rodney.launch

On the workstation, run the following commands to start the remote control node:

Hide   Copy Code

$ source test\_ws/devel/setup.bash

$ export ROS\_MASTER\_URI=http:*//ubiquityrobot:11311*

$ roslaunch rodney remote.launch

A small window whose title is "**ROS keyboard input**" should be running. When entering keyboard strokes ensure the small window has the focus.

In another terminal on the workstation start rviz with the following commands:

Hide   Copy Code

$ source test\_ws/devel/setup.bash

$ export ROS\_MASTER\_URI=http:*//ubiquityrobot:11311*

$ roslaunch rodney rviz.launch

#### Localising the Robot

Configure rviz to display the robot model or base\_link axis, laser scan, map and pose estimates. Also ensure that the map is the fixed frame.

We can see from the display that the laser scan does not match the map, the pose estimates are spread around and I know the model/axis is not facing the same direction as the real robot. So before we give the robot a navigational goal we need to improve its localisation.

The image below shows a poorly localised robot. The red lines are the laser scan and the green arrows are the pose estimates.

[??? image as described above]

The first operation we will carry out is to give the robot an improved localisation using rviz. Click the "2D Pose Estimate" button, estimate the real location and pose of the robot and click/drag the large green arrow on the map to set the initial pose. You can keep doing this until you get the laser scan close to matching the map.

We now have a good initial pose but the pose estimates are still out. We can improve these by driving the robot around in manual mode. Spinning on the spot is a good manoeuvre to conduct. Whilst moving the robot you should see the pose estimates converging on the robots position.

[??? image with better pose estimate]

#### Setting a Navigation Goal

We are now ready to send the robot on its way but let’s first take a look at the costmaps that will be used for planning the route. Select Global Planning in rviz to display the Global Costmap. For the Global Costmap I like to select "Draw Behind" so that the map is washed out behind the main map.

From the costmap you can see the open spaces the planner will try and use and the risker places like up against a wall.

[??? image global costmap]

Now the Global Costmap was constructed from the main map and will be used to plan an ideal route, but the actual movement of the robot will be governed by the Local Costmap and this will be generated on the fly as the sensor data arrives. This will allow the robot to avoid objects that weren't there when the map was created, i.e. a sleeping family pet.

Select Local Planning in rviz to display the Local Costmap. I like to have this map superimposed on top of the main map.

[??? image local costmap]

Now set the target goal pose by clicking the "2D Nav Goal" button, click/drag the large green arrow on the map to set the goal. Now I have deliberately left the robot in manual mode so it will not move and this gives us a chance to examine the global plan which is shown as a thin green line in the image below.

[??? image of route]

We can put the robot into to autonomous mode by giving the **ROS keyboard input**window the focus and pressing the "1" key (not on the numeric keypad). This is the request to run mission 1 which is currently empty so all it does is take the robot out of manual mode so that the velocities generated by the navigation stack and not the joystick/keyboard will be sent to the thunderorg node which controls the motors.

Hopefully the actual robot will navigate to the goal pose and you can monitor the progress on rviz.

I have noticed that after setting a number of poses to move to eventually despite the clear route on the costmaps the nav stack fails to calculate a route. You can clear this problem with the following command at the workstation:

Hide   Copy Code

$ rosservice call /move\_base/clear\_costmaps

[??? Video of SLAM Navigation]

## Points of Interest

??? State what we did and what we will do next. Also invite people to fork my repositories and submit pull-requests

## History

??? Keep a running update of any changes or improvements you've made here.