Currently just notes

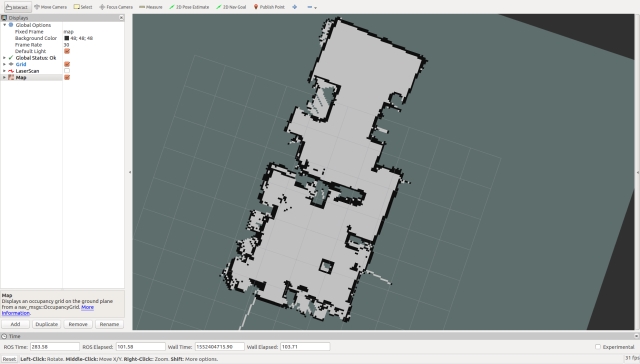
include existing ROS packages for SLAM (simultaneous localization and mapping), a probabilistic localization system,  global and local navigational planning, all to put us well on our way to completing Design Goal 5.

Navigation "Stack"

In older versions of ROS packages could be organised into ROS stacks, this is an outdated term but sometimes you will still see the term stack used especially the "Nav Stack".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

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. Its no perfect but it is possible to use if for navigation.



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 so that you can adjust the parameters and reproduce the map. I prefer a mixture of the two, record the data but also visualise the map in rviz as it is created.

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 topics are not continually published but are latched and any new node 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* 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 it 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 ans 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 5 for Rodney - A Long Time Coming Autonomous Robot (Part 6)   Copy Code

amcl:

# 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: 10.0

# 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: beam

# 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

 We will need to add the the following line near the top the launch file to load the file into the parameter sever.

Hide   Copy Code

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

  The following added to the launch file will launch the node.

Hide   Copy Code

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

It's worth noting that most of the parameters I have changed for from defaults for the nav stack relate to transform tolerances and and sample frequencies, we are asking a lot of the Raspberry Pi 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 it needs to allow for things changing since the map was constructed. If a person or a family pet walking 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 the ROS Wiki page on it.

One of the things to note is that it use two costmaps to plan routes. 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 were the Local costmap comes in. It's updated as the robot moves and where as the Global costmap covers the entire map the Local costmap covers only the immediate location around the robot.

A problem I have encountered on Rodney is that the Local costmap is in th odom frame and although I have tuned the odom parameters (wheel circumference and wheel distance) there are times when the Local costmap can become rotated in axis to the Global one and hence routes through narrow places are blocked off. I can get around this problem by resetting the costmaps before a new nav goal is set and I'll explain this in the Using The Code section.

The odom is always going to be an estimate position but in the future I plan to look at including an IMU (Inertial Measurement Unit) and a Extended Kalman filter to fuse the odom and IMU data to produce a more accurate odom.

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

*base\_local\_planner\_params.yaml*

Hide   Copy Code

move\_base:

controller\_frequency: 1.4

TrajectoryPlannerROS:

holonomic\_robot: false

meter\_scoring: true

max\_vel\_x: 0.25

min\_vel\_x: 0.17

escape\_vel: -0.2

max\_vel\_theta: 1.5

min\_vel\_theta: -1.5

  min\_in\_place\_vel\_theta: 1.2

Here I had to slow down the controller frequency, the holonomic\_robot false value basically states that we are a two wheeled robot using differential drive foe steering. As this node will be responsible for issuing velocity commands we also have to give it some bounds for those velocities.

The costmaps have some common parameters and some which are set for the two different costmaps.

*costmap\_common\_params.yaml*

Hide   Copy Code

obstacle\_range: 2.5

raytrace\_range: 2.0

footprint: [[0.170, 0.090], [0.025, 0.090], [0.025, 0.145], [-0.025, 0.145], [-0.025, 0.090], [-0.170, 0.090], [-0.170, -0.090], [-0.025, -0.090], [-0.025, -0.145], [0.025, -0.145], [0.025, -0.090], [0.170, -0.090]]

observation\_sources: laser\_scan\_sensor

laser\_scan\_sensor:

sensor\_frame: laser

data\_type: LaserScan

topic: scan

marking: true

clearing: true

This gives a footprint for the base including the wheels that stick out from the side and information on how to find the topic with our filtered laser ranges.

*global\_costmap\_params.yaml*

Hide   Copy Code

move\_base:

global\_costmap:

global\_frame: map

robot\_base\_frame: base\_link

static\_map: true

transform\_tolerance: 0.6

update\_frequency: 1.0

*local\_costmap\_params.yaml*

Hide   Copy Code

move\_base:

local\_costmap:

global\_frame: odom

robot\_base\_frame: base\_link

static\_map: false

rolling\_window: true

transform\_tolerance: 0.7

update\_frequency: 1.0

  publish\_frequency: 0.5

Like the main the map the costmaps are latch topics and not continually published. So that I can see the local costmap update in rviz I have changed the publish\_frequency from 0.0 to 0.5 so that is published every 2 seconds. Once the navigation is working it will be prudent to go back to the default so that the map is not continually published.

These files will be loaded into the parameter server with the inclusion of the following lines in the *rodney.launch*file.

Hide   Copy Code

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

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

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

Note that this does note load the costmap common parameters, this file will be loaded twice into different name spaces for the local and global costmaps when the nodes are launched.

To launch the *move\_base*node I have added the following to the *rodney.launch* 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/costmap\_common\_params.yaml" ns="global\_costmap"/>

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

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

</node>

The published topic from move\_base is normally *cmd\_vel*, which is the topic 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*.

….

In the next part of these articles we will changing the *rodney*\_missions node to programatically start autonomous navigation to a new pose.

….

## 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.

At this stage we will use the tool rviz to set the navigation task goals, in the next part we will add code so that this is done programmatically.

### 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:

$ mkdir -p ~/rodney\_ws/src

$ cd ~/rodney\_ws/

$ catkin\_make

Copy the packages face\_recognition, face\_recognition\_msgs, head\_control, pan\_tilt, rodney\_missions, servo\_msgs, speech, thunderborg and tacho\_msgs into the ~/*rodney*\_ws/src folder.

Download the packages ros-keyboard (from <https://github.com/lrse/ros-keyboard>) and rplidar\_ros (from <https://github.com/Slamtec/rplidar_ros>) and copy them both into the ~/*rodney*\_ws/src folder.

Build the code with the following commands:

$ cd ~/rodney\_ws/

$ catkin\_make

Check that the build completes without any errors.

You will also need to compile and download the sketch to either the Arduino Nano or a 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:

$ mkdir -p ~/test\_ws/src

$ cd ~/test\_ws/

$ catkin\_make

Copy the packages rodney, joystick, and ros-keyboard(from <https://github.com/lrse/ros-keyboard>) into the ~/test\_ws/src folder and then build the code with the following commands:

$ 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.

$ 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.

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.

### 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:

$ 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:

$ 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:

$ 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:

$ 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

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

$ 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.

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:

$ 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:

$roscore

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

$ rosparam set use\_sim\_time\_true

$ rosrun gmapping slam\_gmapping

In another terminal playback the rosbag recorded previosusly with the following commands:

$ rosbag play --clock data.bag

Then sit bag whilst 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:

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

You can play with the gampping config 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

### 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.

$ source rodney\_ws/devel/setup.bash

$ roslaunch rodney rodney.launch

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

$ 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:

$ 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, 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 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.

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 manoeuvrer to conduct. Whilst moving the robot you should see the pose estimates converging on the robots position.

#### Setting a Navigation Goal

We are now ready to send the robot on its way but lets 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.

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.

Now there is a route which is possible to navigate from the room at the bottom of the map to the room in the top of the map, but I can see that the costmap has become distorted by the errors in the odom from when I moved the robot to improve the pose estimates.

Open a new terminal on the wokstation and type the following commands to reset the costmaps:

$ source test\_ws/devel/setup.bash

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

$ rosservice call /move\_base/clear\_costmaps

A new more accurate Local Costmap will be displayed in rviz.

Now set the target goal pose by clickin 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.

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 and means 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. In the image below the robot has reached the selected goal.

Now if you want to give it another goal to move to reset the costmaps first.

Video of SLAM Navigation

……………