Implementing Odometry and SLAM Algorithms on a Raspberry Pi to Drive a Rover

Research · May 2015

DOI: 10.13140/RG.2.1.1747.8569

1 author:

Patrick Butterly
Institute of Technology Blanchardstown

Institute of Technology Blanchardstown

Implementing Odometry and SLAM Algorithms on a Raspberry Pi to Drive a Rover

Authors
B00055556 – Patrick Butterly
B00056835 – Joshua Daly
B00056837 – Leigh Morrish

Supervisor
Pr. Arnold Hensman

A thesis submitted in fulfilment of the requirements for the degree of Bachelor of Science (Ordinary) in Computing – May 2014

Contents

Abstract 3
Introduction 4
Requirement Specification 5
Design Specification 7
  Programming Languages 8
  APIs 8
Implementation 9
  Wireless Communication 9
    Configuration 9
    Explanation 10
  Controlling the Raspberry Pi via a Laptop 11
  Static IP 12
  Communicating with the Rover 14
  Host to Arduino 14
    Message Structure 14
    Example 14
  Arduino to Host 15
    Odometry Reports 15
    Example 15
  Scan Reports 16
    Example 17
  Available Commands 18
Arduino Operation 19
    Processing Commands 20
Returning Data 22
  Odometry 22
    Example 23
  Scan Results 24
  Rover Locomotion 26
SLAM Implementation 27
  System Design 27
  Event Processing 29
  EKF SLAM Algorithm 30

Efficiency 31
Landmark Extraction 32
Graphical Representation 33
Thread Safety 35
Testing 36
Test 1: Sonar accuracy 36
Test 2: Odometry 38
Conclusion 40
Further Work 41
Hardware 41
Software 41
References 42
Appendix 44
List of Suppliers 44
Hobby Components 44
Maplin 44
RobotShop 44
Radionics 44
List of Components 45

Abstract

This paper explores the problem of implementing a teleoperated navigation system on a mobile robot using low cost equipment by critically analysing the current trends in mobile robotics. In the hobbyists robotics community there is a current lack of teleoperated robots capable of simultaneously mapping and localising themselves within an environment. The work presented here builds upon the research of a past student at the Institute of Technology Blanchardstown who set out to create such a robot. In order to address that issue this paper looks at tracking a mobile robot's position within its environment using commodity mouse sensors and electro magnetic compasses. It then goes on to explore how a simple microcomputer can aid a mobile robot in constructing a consistent map of its environment using a mapping algorithm.

Introduction

Simultaneous Localisation and Mapping (SLAM) is a complex robotics problem that scientists have been working on for nearly thirty years (Durrant-Whyte, H. & Bailey, T. 2006). The focus of this project is on using an off-the-shelf controller to run a SLAM algorithm called EKF SLAM (Riisgaard and Blas, 2005). This project was taken on because of our interest in SLAM and how we could integrate it into existing equipment.

In May 2013 a fourth year student at the Institute of Technology Blanchardstown, Sebastian Poenar set out to build an autonomous rover capable of navigating and mapping an indoor environment. The robot would use several internal sensors connected to an Arduino microcontroller (Poenar, S. 2013) to collect data about its physical surroundings, this information would then be processed on a host PC and from it a real-time map would be created.

Overall Sebastian's project fell short of its intended goal and many of the envisioned features remained unimplemented. The robot's behaviour was limited to wandering and recording the distance to any objects encountered. In all Sebastian's final result left much to be desired. Further efforts could have been made to construct a dynamic map from the data being recorded by the host, it this technical challenge that is the central focus of this project.

We have inherited the rover and made some modifications and improvements, as well as reprogramming the Arduino. We chose to add a microcomputer to the mix in order to give more processing power for SLAM to run more efficiently. This system will be fully wireless and the resulting data will be transmitted back to a laptop.

Requirement Specification

On completion of this project, a rover will be assembled and programmed with Simultaneous Localisation and Mapping algorithms allowing it to traverse an unknown area and then build a map of the area traversed. The rover will be driven manually from an external system, allowing the rover to be fully wireless up to a distance of 5m.

The rover will be controlled remotely, and the algorithms will be implemented on a simple microcontroller, a system with very limited power. For the project to be a success the finished product must be capable of:

- Traversing an unknown environment using distance based sensors
- Mapping that environment
- Traversing the map
- Responding to Manual Command
- Coping with various constraints while performing desired actions
- Wireless Connectivity between Rover and Laptop
- Live Updates of position and orientation relative to the environment

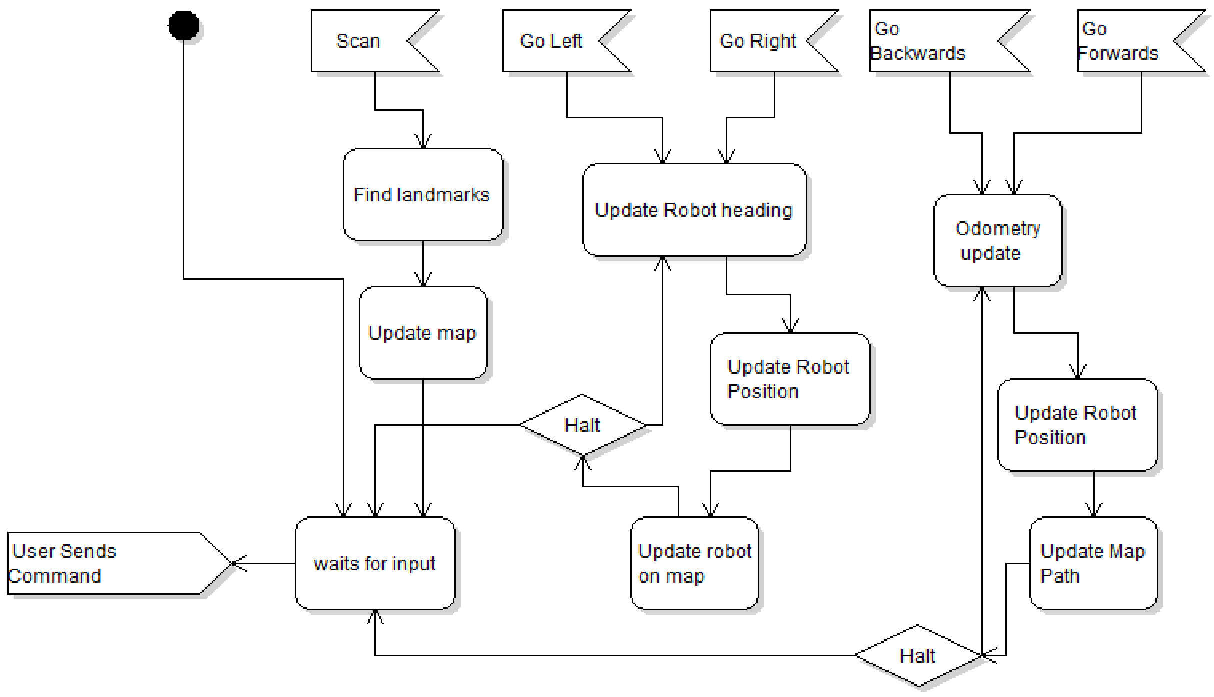


Figure 1 - Activity Diagram

To accomplish this, the following is necessary:

- A chassis capable of housing small chipsets
- Small electric motors capable of moving the chassis
- A microcontroller
- A mountable computing unit
- An odometry measurement device
- A sonar distance sensor
- A servo unit
- Cables/Connections
- Wireless Adaptors
- Various power sources

- SLAM algorithm
- Manual Control functions
- Scanning functions
- Map Making algorithm
- Connection/Communication software

Design Specification

This project is designed around using a Raspberry Pi to power the algorithms and instruct the Arduino to control the motors of the rover and to monitor the input of the sensors. This is a challenge because of the limited processing power of the Raspberry Pi which is 700mhz. Due to these power limitations some SLAM algorithms have not been considered for this project due to their need for high processing power. Other systems that have greater processing power have had a lot of success with various other SLAM algorithms and has led it to become very popular recently and is therefore well documented.

Our system will have a limited version of SLAM running on the Raspberry Pi which will be instructing the Arduino on its movements in the Environment. The Arduino will be monitoring the various sensors and send the information it gathers from the sensors to the Raspberry Pi for processing. The Raspberry Pi will input the information into the SLAM algorithm which calculates the path taken. The Raspberry Pi will then translate this path onto a map. We can drive the rover via the Raspberry Pi and the Arduino by instructing the Arduino to drive the motors. The rover is not able to scan the area while moving, and any readings taken while moving would be inaccurate, so the rover can only scan while in a stationary position. It can pick up on nearby landmarks and draws them on the map as lines.

In our research we discovered *SLAM for Dummies* (Riisgaard and Blas, 2005) and decided to work on a similar tangent to their work. We have very different Rovers for carrying out movement and slightly more efficient algorithms for our landmark and map drawing. We decided to use a Raspberry Pi rather than follow their design of using a whole Laptop because of our interest in working with the Raspberry Pi and testing the limits of its performance. It is also more cost effective for us to use the Raspberry Pi instead of a Laptop, and the small size of the Raspberry Pi means we can have a compact rover with the Raspberry Pi mounted on top.

Programming Languages

The implementation will be split between two programming languages Arduino's subset of C++ and Mono's cross platform C#. All of the programming for Arduino will be done using the official IDE, and will be contained within a single sketch *Robot.ino* along with its accompanying header *Robot.h*. The rest of the code will be wrote using MonoDevelop in an Lubuntu virtual machine.

We have chosen to use C# because the SLAM algorithm that we plan to use was originally wrote in it so rather than reinvent the wheel we will build upon it. Another advantage of using C# is the fact that it is implemented using portable *bytecode*, allowing us to run it on Lubuntu (x86) and the Raspberry Pi (ARM) without having to recompile. It also compiles faster than architecture dependent languages such as C++ which in turn cuts down on build times, although its execution is marginally slower.

APIs

We have decided to use the PS/2 Arduino Library written by Chris J. Kiick to enable the Arduino to communicate with the optical mouse sensor. We have found a bug in the original source, which was fixed by changing an old header from <WProgram.h> to <Arduino.h>, but otherwise it worked fine.

We will be using the HMC5883L Library for communicating between the HMC5883L and the Arduino. The version we are using was released by Love Electronics in 2011 under the GNU GPL version 3 Licence. We found this very helpful as it saved us from writing the low level code in order to interface with the HMC5883L.

We have decided upon GTK# to create our GUI interface, buttons, labels etc. It was chosen over other solutions such as WinForms as it is no longer being developed by Microsoft or Mono and it does not support the native look-and-feel.

Lastly we will be using the Cairo Vector Graphics Library for drawing a simple 2D representation of the map created by the rover. While this is not the best library for fast real time rendering the rovers update rate is slow enough for this not be an issue. In future projects we would recommend using something else, such as OpenGL which is more suited to real time rendering.

Implementation

Wireless Communication

We had some difficulty in connecting the rover with our remote system, namely between a Laptop and the Raspberry Pi. We found that the daemon wpasupplicant was not linking effectively with /etc/network/interfaces. This problem seems to be undocumented and specifically on the Raspberry Pi, not other Linux systems. We temporarily fixed this issue by making this symbolic link:

```
/etc/network/interfaces /etc/wpa_supplicant/wpa_supplicant.conf
```

Another issue we have faced is with our outdated hardware. We are using a fairly old router that does not support DHCP, so all our addresses are static. The addresses are as follows:

```
Router/Gateway:    192.168.2.1

Raspberry Pi :wlan0:          192.168.2.3

Raspberry Pi :eth0:10.42.0.2

Connected laptop1:            192.168.2.5
```

The interface eth0 is set up so that we can tunnel into the Raspberry Pi from a Laptop without needing to set up the Raspberry Pi with its own display etc. and is useful for quickly troubleshooting problems if the wifi fails to connect automatically.

Configuration

/etc/network/interfaces

```
auto lo

        iface lo  inet loopback



        iface      eth0 inet static

                   address   10.42.0.2

                   netmask  255.255.255.0

                   gateway  10.42.0.1



auto      wlan0

          allow-hotplug        wlan0

          iface      wlan0 inet static

                     address   192.168.2.3

                     netmask  255.255.255.0

                     gateway  192.168.2.1
```

```
wireless   essid default
```

Explanation

The line “auto wlan0” is important as it defines the interface on which the following commands will be implemented on. The “auto” is not commonly used when working with the wpa supplicant but if it isn’t used in this situation it will not connect to the wireless network.

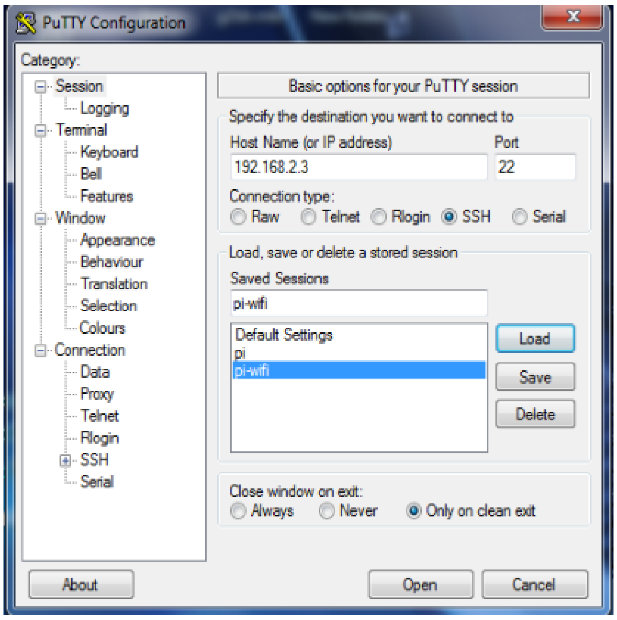
“Allow-hotplug” tells the Linux Kernel to bring up the interface and is usually used instead of “auto”, but because of the Raspberry Pi having issues connecting, both are needed to ensure connectivity.

The line “iface wlan0 inet static” commands the Raspberry Pi to connect to the interface to connect via the static IP. The details of the IP are just below the command. These details would usually be in the wpa supplicant.

***!Note:** It is very important to configure the the SSID as shown above “wireless essid default”, default being the SSID of the router. If you configure it as usual, i.e. “essid default”, it will result in the Raspberry Pi appearing to be connected to a connection called default, but there will be no actual connection.*

Controlling the Raspberry Pi via a Laptop

After the connection is set up, we can connect to the Raspberry Pi using Remote Desktop or an SSH Client(we used Putty on Windows OS). The direct SSH setup is the easiest and most useful to make changes and run commands. To do this, you can enable SSH from the Raspberry Pi Boot/Configure menu that comes with the Raspbian OS by scrolling down to advanced options ->SSH-> enable. Then you can connect using putty by putting the IP address of the Raspberry Pi in the Hostname (or IP address) field and clicking open. This closes the window and opens a terminal(command prompt) window. If the connection was successful you will be prompted to log in with your details (username and password entered when the OS was installed).You can now enter commands into the window just like you would on the Raspberry Pi's normal terminal.



Specify the destination you want to connect to

Host Name (or IP address): 192.168.2.3
Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Load, save or delete a stored session

Saved Sessions: pi-wifi

Default Settings
pi
pi-wifi

Close window on exit: Always Never [✓ Only on clean exit]

Figure 2 - Putty Configuration

The next step is to be able to use the desktop remotely. For this projects purposes this is only necessary so that the map can be viewed as it is drawn on the Raspberry Pi. Otherwise, commands are just entered on the terminal.

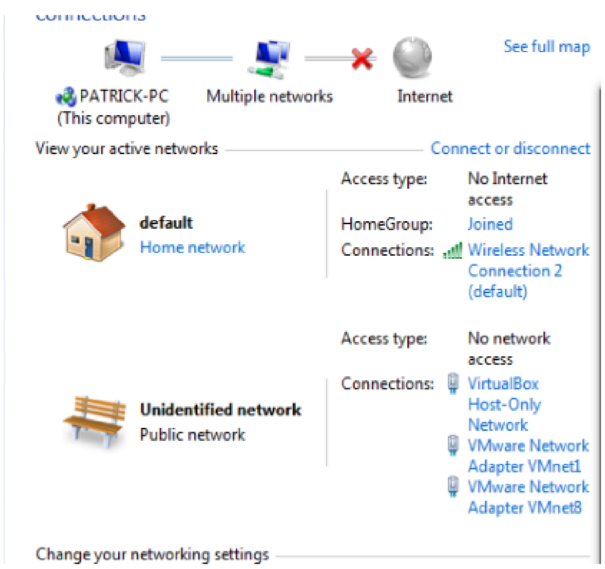
There are many different types of Remote Desktop software but in this project we are using one provided with the Windows OS. No extra configurations are needed to use the software, only the IP address, username and password. The Control View window will then open and be ready for use.

!Note: *Another Remote Desktop software we experimented with was Real VNC. It required installation of a VNC server and only allowed access to specific iterations which the user preconfigured. This setup would be better suited to a project with more security concerns than our own.*

Static IP

Statically assigning an IP address is a significant advantage for operating this type of system which automatically connects to a router, but it is essential in this case to be able to connect to the Wi-Fi as as the router available does not have built-in DHCP functionality.

To connect the Windows 7 Laptop to the router, we have to open the network and sharing centre. This can be done from either the control panel or the Wi-Fi connectivity Icon on the task bar.



PATRICK-PC (This computer) — Multiple networks — Internet

See full map

View your active networks — Connect or disconnect

default
Home network

Access type:	No Internet access
HomeGroup:	Joined
Connections:	Wireless Network Connection 2 (default)

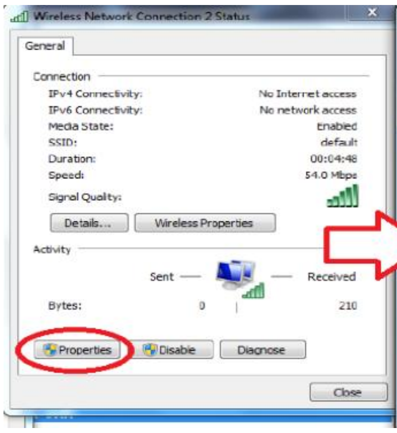
Unidentified network
Public network

Access type:	No network access
Connections:	VirtualBox Host-Only Network VMware Network Adapter VMnet1 VMware Network Adapter VMnet8

Change your networking settings

Figure 3 - Network Center - The network and sharing centre should present you with a window similar to what is shown above.

As you can see, the wireless is already connected to the router named “default”. By clicking the connection we can edit the properties by clicking Properties, Internet Protocol Version 4, and edit its properties as shown below.


Wireless Network Connection 2 Status

General

Connection

IPv4 Connectivity: No Internet access
IPv6 Connectivity: No network access
Media State: Enabled
SSID: default
Duration: 00:04:48
Speed: 54.0 Mbps
Signal Quality:

Details... | Wireless Properties

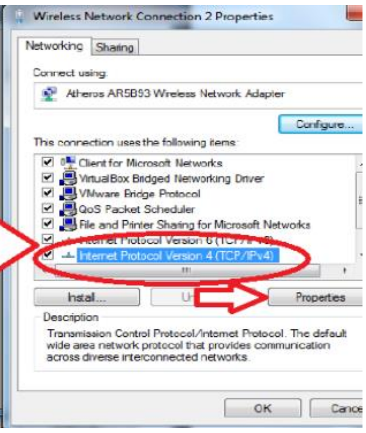
Activity

Sent — Received

Bytes: 0 | 210

Properties | Disable | Diagnose

Close


Wireless Network Connection 2 Properties

Networking | Sharing

Connect using:

Atheros AR5B93 Wireless Network Adapter

Configure...

This connection uses the following items:

- [✓] Client for Microsoft Networks
- [✓] VirtualBox Bridged Networking Driver
- [✓] VMware Bridge Protocol
- [✓] QoS Packet Scheduler
- [✓] File and Printer Sharing for Microsoft Networks
- [✓] Internet Protocol Version 6 (TCP/IPv6)
- [✓] Internet Protocol Version 4 (TCP/IPv4)

Install... | U | Properties

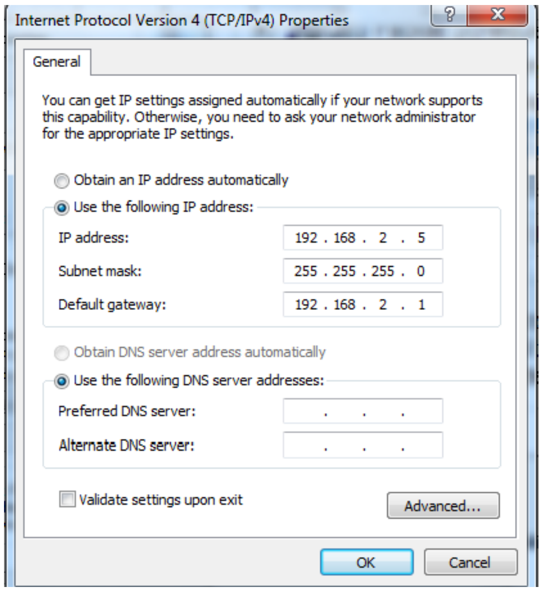
Description

Transmission Control Protocol/Internet Protocol. The default wide area network protocol that provides communication across diverse interconnected networks.

OK | Cance

Figure 4 - Connection Status   Figure 5 - Connection Properties

Now we can edit the computers IP settings. You can now enter the static IP address, subnet mask and default gateway. The default gateway is the address of your router. You should be able to configure the router by entering the default gateway into the address bar of your browser.


Internet Protocol Version 4 (TCP/IPv4) Properties

General

You can get IP settings assigned automatically if your network supports this capability. Otherwise, you need to ask your network administrator for the appropriate IP settings.

( ) Obtain an IP address automatically
(●) Use the following IP address:

IP address: 192 . 168 . 2 . 5
Subnet mask: 255 . 255 . 255 . 0
Default gateway: 192 . 168 . 2 . 1

( ) Obtain DNS server address automatically
(●) Use the following DNS server addresses:

Preferred DNS server: . . .
Alternate DNS server: . . .

[ ] Validate settings upon exit

Advanced...

OK | Cancel

Figure 6 - IP addresses

***!Note:** If you have access to a router capable of DHCP you can bypass static configuration shown above, but you will still need to statically configure the IP address on the Raspberry Pi or use a tool to find out what its IP address is every time you connect to it.*

Communicating with the Rover

All low level communication to and from the rover's sensors and motors is done via the Arduino USB serial interface. The rover will respond to a set of programmed commands such as forward, backward, rotate left, rotate right, halt, and scan. It will also constantly report its change in odometry, except when it is performing a scan.

Host to Arduino

Message Structure

The Arduino USB serial interface works on 8 bit data packets which is the set standard for serial communication. This means that when sending a data type such as a 32 bit integer the data is actually split into 4 separate packets (32 / 8 = 4).

The rover interprets serial data as 8 bit char data types, it can accept up to a maximum of two chars per message. The value of each char is interpreted using the ASCII table in which alphanumeric characters are assigned a value between 0 and 255. A command is in the form:

[command character]\n

A list of all of the available command characters can be found under Available Commands. All messages sent to the rover must be terminated by a newline '\n'.

Example

The below pseudo code instructs the robot to begin moving forward using the 'w\n' command:

```
create serial port
set serial port baud rate
set serial port time outs
open serial port

wait 2000 milliseconds for the robot to boot
write "w\n" to the serial port
close serial port
```

Arduino to Host

Odometry Reports

As long as the rover is not performing a scan it will report back its change in odometry about every 500 milliseconds. The odometry reports are in the form:

o,[x],[y],[theta]\n

Where “o” is an odometry message header which tells the receiver what the message contains, x is the last reported x displacement, y is the last reported y displacement, and theta is the robots current heading in radians. Odometry reports are terminated by a newline character '\n' and delimited using a coma ','.

Example

The below pseudocode constantly reads the odometry data stream:

```
create serial port
set serial port baud rate
set serial port time outs
open serial port


wait 2000 milliseconds for the robot to boot
while true:
    read a line from the serial port


close serial port
```

Scan Reports

When the rover is given the command to begin scanning it halts all movement and goes into scan mode. While in scan mode odometry reports are temporarily disabled and the rover assumes that its odometry will not change for the duration of the scan. During the scan the rover will collect 180 readings from its servo mounted sonar, after it has collected all 180 readings it will immediately send them down the serial line. Scan reports are in the form:

s,[reading],[reading],[reading],....[reading],[reading],[reading]\n

Where "s" is a scan message header which tells the receiver what the message contains, [reading] is the distance at x degrees. The first reading was taken at 0° and the last at 179°. Scan reports are terminated by a newline character '\n' and delimited using a coma ','.

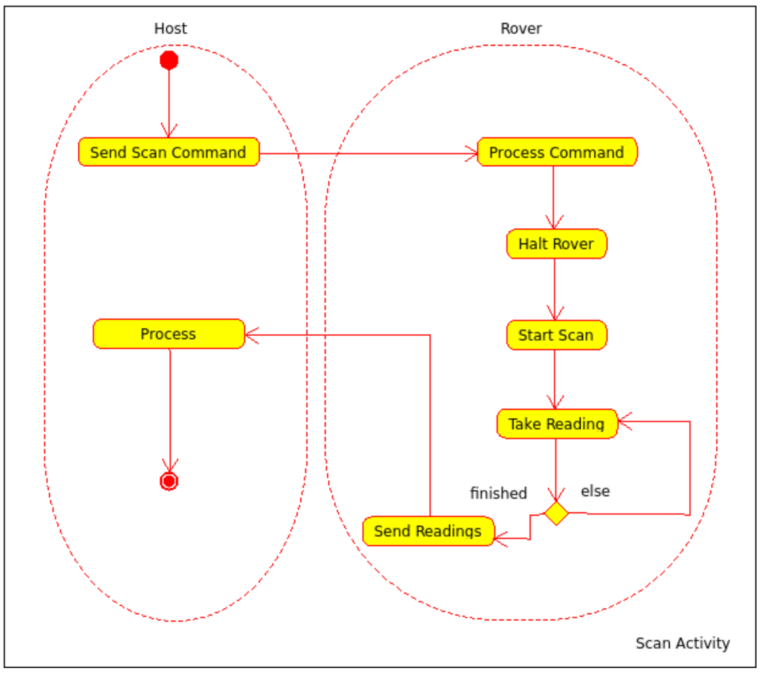


Figure 7 - Scan Report

Example

The below pseudo initiates a scan and reads the result:

```
create serial port
set serial port baud rate
set serial port time outs
open serial port

wait 2000 milliseconds for the robot to boot
    write "e\n" to the serial port

    wait for a response
    read in the line of readings

close serial port
```

Available Commands

Below is a list of all the available commands that the rover will accept:

Syntax	**Description**	**Expected Result**
w\n	Instructs the rover to move forward.	The rover will move forward.
s\n	Instructs the rover to move backward.	The rover will move backward.
a\n	Instructs the rover to rotate left.	The rover will rotate left.
d\n	Instructs the rover to rotate right.	The rover will rotate right.
q\n	Instructs the rover to halt.	The rover will halt all motion.
e\n	Instructs the rover to begin scanning.	The rover will start a scan.

Arduino Operation

The sketch uploaded to the Arduino is designed to be as lightweight and as efficient as possible. This is important because unlike traditional systems such as modern desktops Arduino microcontrollers are limited to Kilobytes of memory, not Gigabytes. The Uno used in this project has 32K of memory. Some of the larger Arduinos have considerably more memory, such as the 2560 Mega which has 256K of memory (Arduino 2014). It is very easy to exhaust this limited amount of memory. For example, creating an array larger than 32K will result in undefined behaviour.

!Note: *Another common cause of erratic behaviour when programming for the Arduino platform is illegal memory access. Attempting to access an array index that does not exist can lead to seemingly random behaviour and unlike an operating system the Arduino does not have any memory protection, something which one of the authors learnt the hard way.*

With that in mind the Arduino Uno in this project acts purely as a slave for a more powerful host system, it understands a set number of commands and has little to no intelligence of its own. The Arduino can be thought of as the spinal cord of the rover, all data to and from the rovers sensors and motors must pass through it, while some external host acts as the rover's brain. In this section we will discuss some of the Arduino's core functionality using snippets from the source code and supported with pseudo code, in order to gain a complete understanding of how the Arduino operates.

Processing Commands

Towards the top of the main loop in the Arduino source code there is a conditional *if* statement similar to the following:

```
// Check to see if at least one character is available.
if (Serial.available())
{
        // Redeclare this every time to clear the buffer.
        char buffer[MAX_CHARACTERS];

        bytes = Serial.readBytesUntil(terminator, buffer, MAX_CHARACTERS);

        if (bytes > 0)
        {
                processCommand(buffer);
        }
}
```

During each iteration of the main loop this statement is evaluated once. The outer if statement checks the serial line for available data. If this condition is true the Arduino will attempt to read from the serial line. Note the use of Serial.readBytesUntil(), this function will read data from the serial line until it reads a terminator, the determined length has been read, or it times out (Arduino function library). It also returns an integer value containing the number of bytes read, if it returns a value greater than 0 then the function processCommand() is called.

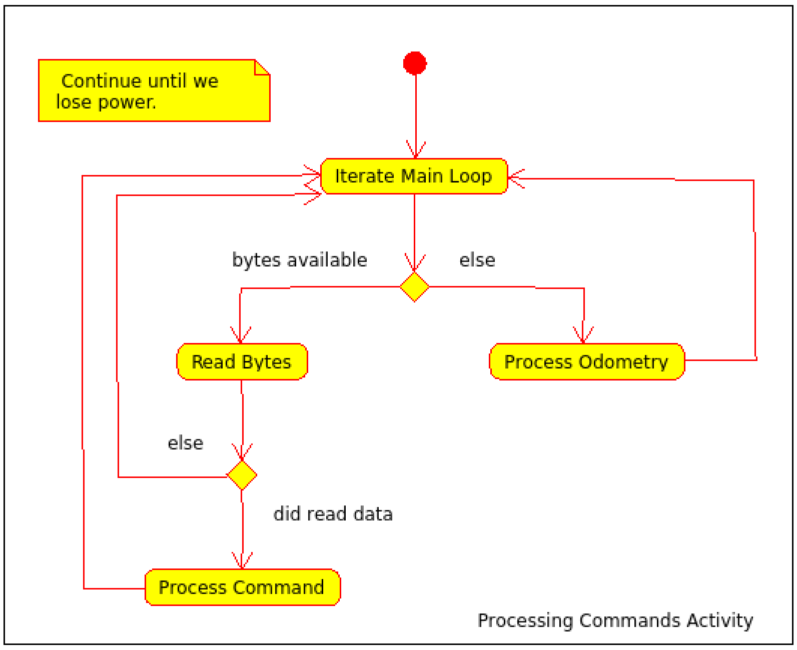


Figure 8 - Processing Commands

The *processCommand()* function contains a lengthy *case* statement that examines the first character of a message, and if the character corresponds to one of the available commands it takes further action. If however it does not recognise the command it reports an error.

```
void processCommand(char command[])
{
        switch (tolower(command[0]))
        {
        case 'w':
#if DEBUG
                Serial.println("Forward");
#endif
                goForward();
                break;
        ... <output omitted> ...
        default:
                Serial.print("Unknown command \"");
                Serial.print(command[0]);
                Serial.println("\"");
        }
}
```

Returning Data

Odometry

We mentioned above how the rover reports two different types of events; its change in odometry and the results of a scan. While we provided sample pseudo code for reading this data we did not yet cover exactly how it is collected. As it turns out it is fairly simple, the pseudocode below covers the steps involved when collecting odometry data:

```
while arduino has power
    if the time since last message is 0
        store the current time
    check the serial line
        process a message

    tell the mouse we want data
    skip the acknowledgement
    skip the status
    store the x movement
    store the y movement

    read rotation from compass
    correct any wrap around

    if current time - stored time >= message rate
        send the odometry message
        reset the stored time, x, and y values

loop
```

The process is made up of a number of sequential reads from both the mouse sensor and the compass. At the start of the loop we check to see if we have stored a time since the last message was sent. When we reach the end of the loop we take the stored time away from the current time and if the elapsed period is equal to or greater than the message rate the data is sent. After the message is sent all of the variables involved are reset.

***!Note:** A better solution would have been to use Arduino Timers (Arduino, 2014), the current implementation makes a best effort to send a message roughly every 500 milliseconds but it is more than likely sent after that period. Timers however can be run independently of the main loop and will execute at exactly the time specified but they are more complicated to implement. If this were a real time application where the difference between a few milliseconds meant success or failure it would have mattered more.*

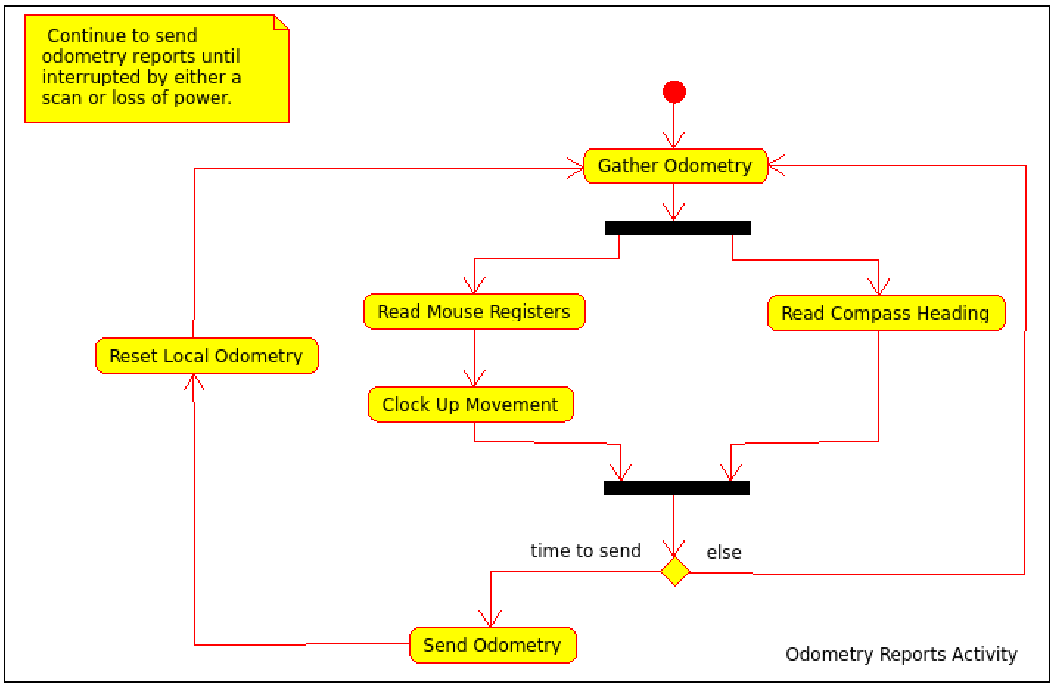


Figure 9 - Odometry Reports

Example

Assuming this is the first iteration of the main loop and the time since start up is 10 milliseconds then:

storedTime = 10 milliseconds

At the end of this iteration the current time is 260 milliseconds since startup and our message rate is every 500 miliseconds so:

currentTime - storedTime = 250 milliseconds

Since 250 is not >= 500 we loop again, the currentTime is now 530 milliseconds so:

currentTime - storedTime = 520 milliseconds

Now 520 is >= 500 so the message is sent and the storedTime is reset to 0.

Scan Results

Unlike odometry, the results of a scan are not returned at any predefined interval. Instead, as discussed earlier, a scan must be initiated by the host. When a scan is initiated all movement is halted and odometry reports are suspended for its duration. The code behind collecting the range readings and sending them down the serial line is very straight forward and just involves a number of iterations:

```
void scan()
{
        for (unsigned char pos = 0; pos < 180; pos++)
        {
                sonarServo.write(pos);
                delay(25);                          // Waits 25ms for the servo to
                                                    // reach the position.
                distances[pos] = takeReading();     // Store sonar reading at current
                                                    // position.
        }

        sonarServo.write(90);

        // Send readings back to the host.
        Serial.print(scanReadingsHeader);

        for (unsigned char i = 0; i < 180; i++)
        {
                Serial.print(",");
                Serial.print(distances[i], DEC);
        }

        Serial.println(); // Message terminated by \n.
}
```

The first for loop deals with moving the range finders servo in steps of 1 degree. It gives the servo 25 milliseconds to move into position before taking a reading from the sonar. It takes a total of 180 readings, storing each one into the corresponding index in the distances[] array. After it has finished taking readings the servo is returned to its original position and the scan readings are sent down the serial line.

Rover Locomotion

The four geared motors that drive the rover are not directly connected to the Arduino. This task is instead carried out by the Adafruit (Adafruit 2014) expansion shield. It is not safe to drive the motors using the Arduino directly because of the high current requirements (up to 600mA per motor), and attempting to do so would burn out the Arduino's chip. The Adafruit shield is designed to drive motors with a voltage rating between 6V-36V and a current draw of 0.6A (peak of 1.2A). The Arduino can control each motor via the Adafruit shield using the AFMotor library (Adafruit 2010).

A motor as defined by the AFMotor library can be in one of three states:

- FORWARD
- BACKWARD
- RELEASE

RELEASE is a stopped state. We are not going to cover the principle of DC motor operation here but if you want to know more about how they work we refer you to a book in the ITB Library *Building Robot Drive Trains.* For now all you need to know is that any one of the four motors can be in a different state.

The rovers locomotion model is based on Skid Steering. Basically, it drives like a tank. It travels in straight lines forwards or backwards and turns on the spot by driving opposite wheels in opposite directions. The corresponding functions in the Arduino source code are goForward, goBackward, turnLeft, turnRight, and halt.

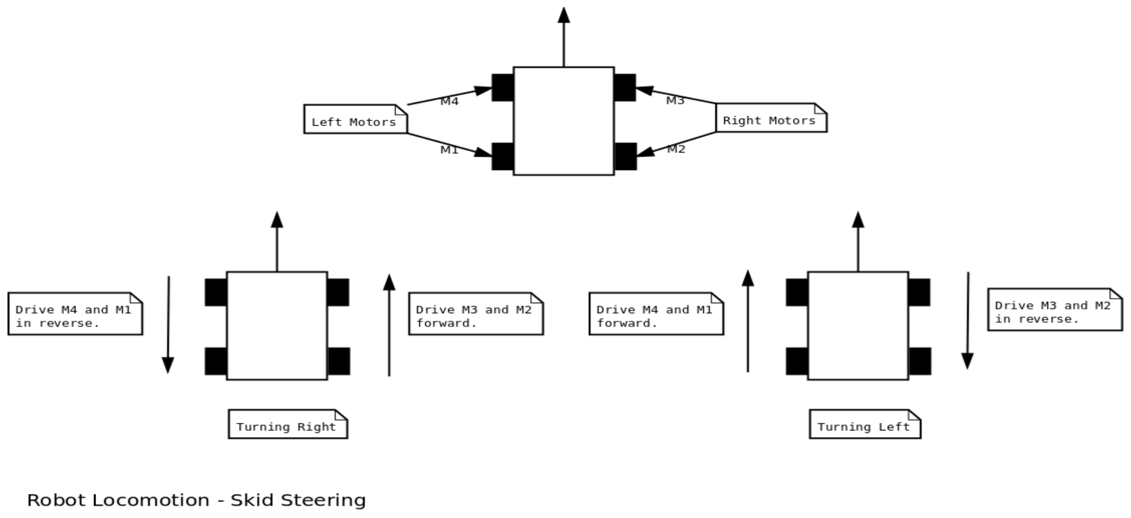


Figure 10 - Rover Locomotion

SLAM Implementation

System Design

The core of the system is built upon three popular software design patterns: the Model-View-Controller (MVC) pattern, the Observer pattern, and the Singleton pattern. Of these the most important is the MVC. Using it we were able to break the system down into smaller reusable units, decreasing any internal coupling. For example, separating the *Robot* classes implementation details from the view allows you to represent the *Robot* graphically any way you want, whether it is a triangle, square, or an image doesn't matter. The only class to use the Singleton pattern is the *SerialProxy*, as there will only ever be one connection to the serial port at any one time it made made sense to use a singleton as opposed to passing references to constructors. The observer pattern is used throughout the system and is discussed under the *Event Processing* section.

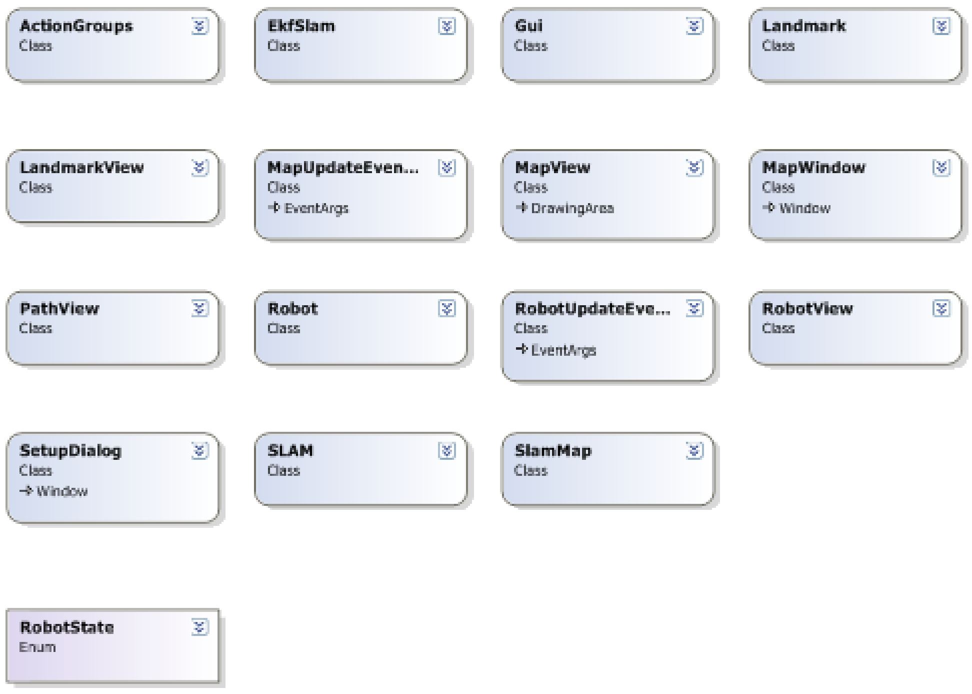


Figure 11 - Classes

The system is split into two separate projects, *Driver* and *SLAM*. Both are contained within the same MonoDevelop solution named *SLAM*. Between the two there are a total of 15 classes and one enum. Of those 15, 5 are views, 4 models, 4 event types, 2 controllers, and 1 SLAM algorithm. At the time of writing the lines of code (loc) stands at 2984, with over 900 in the SLAM algorithm alone.

Each project produces its own output:

- Driver produces: “driver.exe”
- SLAM produces: “SLAM.exe”

The “driver.exe” can be used standalone to monitor the data on the serial port. “SLAM.exe” depends on “driver.exe” and cannot be used without it, they must be in the same path. To start the program on Linux we open a terminal and *cd* into the folder containing both executables and type:

```
mono SLAM.exe
```

The *mono* runtime will then begin to execute the byte code contained within “SLAM.exe”, the version of *mono* must be 4.0 or greater and it will not work with anything older.

When the system is running there are two separate threads of execution, the main thread running GTK and the algorithm, and the read thread with the *SerialProxy*. These two threads of execution communicate using events which is discussed in the next chapter.

Event Processing

At the heart of our system is the event model. Every change in the rover is reported using events, whether it is a simple rotation or complete scan. Attached to these events are *observers* which are programmed to react to them. All of the events are raised via the *SerialProxy.* The *OdometryUpdated* event contains the status of the rover at a given time, and when a scan is completed a *Scanned* event is raised. Any class can observe the events being raised on the *SerialProxy* by providing an appropriate handler:

```
proxy.OdometryUpdated += new EventHandler<OdometryUpdateEventArgs> (SerialProxy_OdometryUpdate);
```

In the case of the *SLAM* class the handler is simply the *SerialProxy_OdometryUpdate* methods signature which takes an *object* (the sender) and an *OdometryUpdateEventArgs* which contains the event data x, y, and theta:

```
private void SerialProxy_OdometryUpdate (object sender, OdometryUpdateEventArgs e)

{

        // Do something here with the event data

}
```

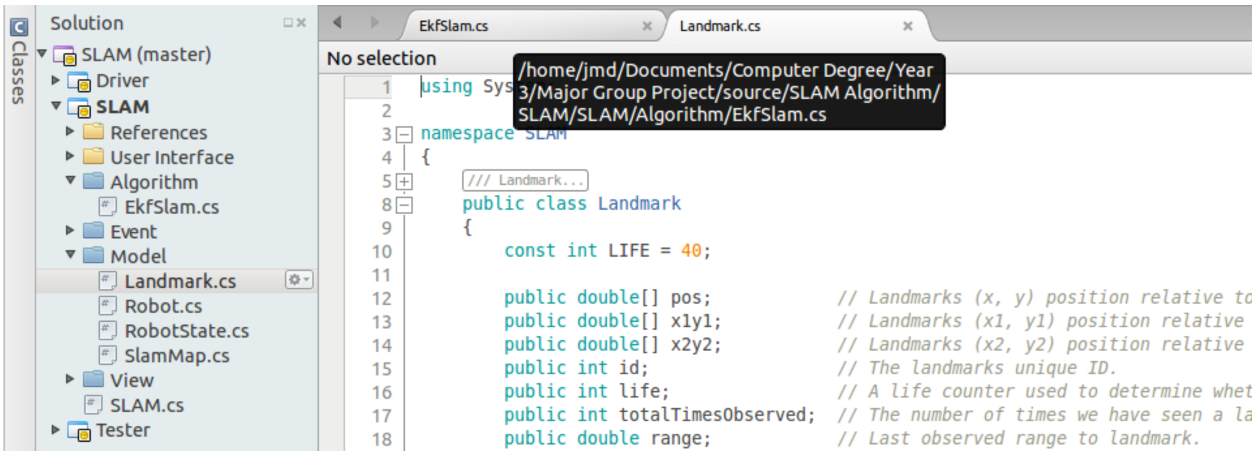
This method will be called whenever the *OdometryUpdated* event is raised on the *SerialProxy*. The real powerful thing about event driven systems is that there can be multiple observers to the same event. Take the case of the *RobotUpdated* event. Several classes may be interested in these updates; a map may want to redraw the robots position, or a landmark may need to update itself based on the new position, and so on. In our system, the event processing works its way along a chain of classes with the same event being propagated to several observers. In the case of the *Robot* class the processing of an *OdometryUpdateEventArgs* may lead to another event being raised, in this case *RobotUpdated,* which has its own set of observers.

EKF SLAM Algorithm

The SLAM algorithm deployed in our solution was first presented in the research paper *SLAM for Dummies* (Riisgaard and Blas, 2005) and is based on the Extended Kalman Filter and Random Sampling Consensus (RANSAC) techniques. It was designed to extract two types of landmarks (straight lines and spikes), although we only ever extracted lines. The algorithm looks for lines within the permissible range of the sonar (3m). A line landmark must be at least 0.5m long and 0.5m away from any other line landmark to be considered. Any landmarks that appear at the rovers position are discarded as bad readings. A landmark can be re-observed any number of times, and every time that it is not observed its life counter is decremented. When its life value reaches 0 it is removed. Re-observing the landmark resets its life counter.

The theory behind the algorithms implementation is covered in sufficient detail in Riisgaard and Blas's research paper and so will not be covered again here. Instead we will be focusing on its application and computational efficiency.

Perhaps the most significant flaw with the solution presented in *SLAM for Dummies* is the lack of any sample usage provided with the paper itself. Apart from the 82 pages of C# source code attached to the appendix the reader is left with very little idea of how it can actually be applied. After close analysis we discovered that out of the 82 pages of code only 44 of them would be of any use to us, the remaining 38 were specific to their mobile platforms architecture. Out of those 44 pages, two classes were extracted; the confusingly named EKF algorithm titled "Landmarks" and the inner class "landmark". These classes became "EkfSlam.cs" and "Landmark.cs" in our project.



/home/jmd/Documents/Computer Degree/Year 3/Major Group Project/source/SLAM Algorithm/SLAM/SLAM/Algorithm/EkfSlam.cs

```
using Sys
 
namespace SLAM
{
    /// Landmark...
    public class Landmark
    {
        const int LIFE = 40;

        public double[] pos;             // Landmarks (x, y) position relative tc
        public double[] x1y1;            // Landmarks (x1, y1) position relative
        public double[] x2y2;            // Landmarks (x2, y2) position relative
        public int id;                   // The landmarks unique ID.
        public int life;                 // A life counter used to determine whet
        public int totalTimesObserved;   // The number of times we have seen a la
        public double range;             // Last observed range to landmark.
```

Figure 12 - Project Setup

Efficiency

The solution to SLAM presented in the paper *SLAM for Dummies* is for the most part inefficient. A lengthy study of the source code revealed a number of flaws in its implementation. There is very little use of recursive functions, if any. Instead, everything is carried out using iteration. In general the algorithmic complexity of each method in *Big O* notation is *O(n^2)* or in some cases simply *O(1)*. For example, in its worse case scenario the iteration process in *RemoveBadLandmarks* will run for the entire length of the database twice making it up to *NxN* or *O(n^2)*. So as the database grows the time taken to execute this method will increase exponentially*:*

```
for (int k = 0; k < DBSize + 1; k++){

        <output omitted>

        for (i = 0; i < 4; i++) {

        <output omitted>

        }

        if (inRectangle) {

                if (landmarkDB [k].life <= 0){

                        for (int kk = k; kk < DBSize; kk++).{

                                <output omitted>

                        }

                }

        }

}
```

The method with the poorest performance that we found was *ExtractLineLandmarks* this method contains four nested loops similar to the above putting it in the class of *O(n^3)* or *O(2n)*. As a result this method scales extremely poorly as the database size increases, making it too expensive to be considered useful, something one of the authors later admitted (Riisgaard 2005). Other inefficiencies include the creation of a very large array of *Landmark* objects that is hardly utilised in *ExtractLineLandmarks,* it could easily be replaced with a dynamic structure such as a *LinkedList* or even an *ArrayList*:

```
// Have a large array to keep track of found landmarks.

Landmark[] tempLandmarks = new Landmark[400];
```

Both of the mentioned methods are called every time a scan has been performed meaning that their poor execution times have a significant impact on the entire solution.

Landmark Extraction

In order to extract any type of landmark we must obtain two pieces of information:

- The rovers global position (x, y, heading).
- A number of range readings separated by fixed intervals.

Our rovers global position is reported at a rate of about every 500 milliseconds. The x and y coordinates are calculated based on the rovers displacement along a straight line reported by the optical flow sensor, combined with its current heading from the magnetic compass. The range readings are collected from the servo mounted sonar with a resolution of 1° per scan. A total of 170 readings are taken. All distance based readings are in meters and the rovers rotation is reported in radians.

An important factor to keep in mind is that the greater the number of readings taken the more accurate the map will be. The authors of this algorithm used far superior hardware, namely a SICK laser rangefinder with a price tag of around $1000, capable of covering 0° to 180° in steps of half a degree (Adept Mobile Robotics, 2012). In comparison to that we were using a $30 sonar combined with a hobby RC servo that can cover between 0° to 179° in steps of 1°. So from the beginning we were more than aware that our results would be less accurate.

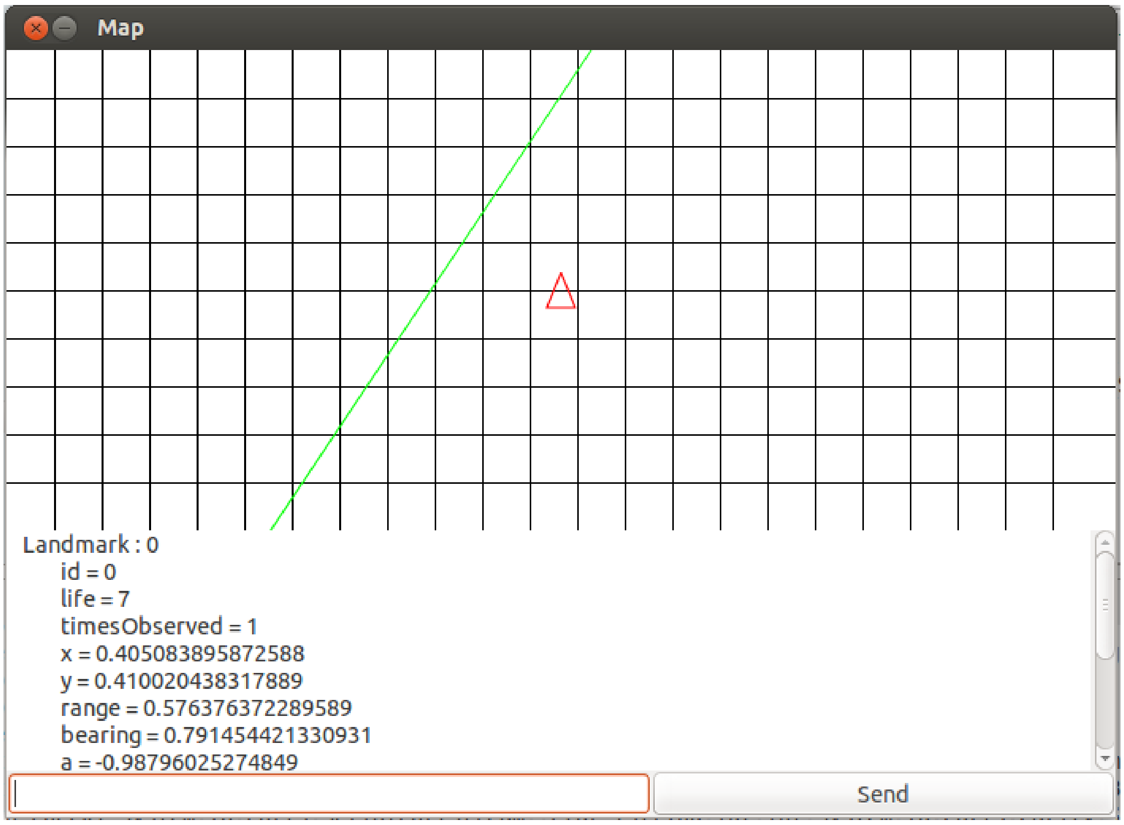
Once this information has been gathered, searching for and extracting any landmarks that may exist just requires an instance of the EKF algorithm. The algorithm instance holds a database which contains every landmark that has been observed. Landmarks are added, updated, and removed from this database. The code for extracting landmarks can be found under the "SLAM.cs" file:

```
private void SerialProxy_Scan (object sender, ScanEventArgs e)
{
    Gdk.Threads.Enter ();
    try
    {
      ekfSlam.RemoveBadLandmarks (e.Readings.ToArray (), robot.Position);
      // Extract any landmarks then update the slam database with any new landmarks.
      ekfSlam.UpdateAndAddLineLandmarks (ekfSlam.ExtractLineLandmarks (e.Readings.ToArray (),
        robot.Position));
      // Now update the model.
      map.UpdateLandmarks (ekfSlam.GetDB ());
    }
    finally
    {
      Gdk.Threads.Leave ();
    }}
```

Graphical Representation

The graphical end of the system is designed to be as abstract from the implementation details as possible. Every model based class has a corresponding view. For example, the *Robot* class that contains all of the implementation details such as the code to control it via the *SerialProxy* is separated from the drawing end of the system which doesn't need to know these details, it is only concerned with the robot's location and rotation. This allows a developer to make changes to the system in isolation and a model could potentially have multiple visual representations all in separate classes of their own.

The most important view class in our system is the *MapWindow* which inherits from the GTK *Window* superclass. It contains all of the other sub views and controls for manipulating the rover. Its center is dominated by a *MapView*, followed by textbox containing landmark data, and at the very bottom the command interface to the rover:



```
Landmark : 0
    id = 0
    life = 7
    timesObserved = 1
    x = 0.405083895872588
    y = 0.410020438317889
    range = 0.576376372289589
    bearing = 0.791454421330931
    a = -0.98796025274849
```

Figure 13 - Sample Map

All of the commands listed in the *Available Commands* section can be sent to the rover via this interface by hitting the send button or the return key on a keyboard. The *MapView* in the center of the window is actually a *DrawingArea* that is passed to each view class in sequence, meaning they all share the same drawing context:

```
DrawingArea area = (DrawingArea)sender;
Cairo.Context cairoContext = Gdk.CairoHelper.Create (area.GdkWindow);

// Draw the background and grid.
<output omitted>

if (robotView.Robot.PathPointList.Count > 1)
   pathView.Draw (cairoContext, centerX, centerY, 1.0);

robotView.Draw (cairoContext, centerX, centerY, 1.0);
landmarkView.Draw (cairoContext, centerX, centerY, 1.0);

((IDisposable)cairoContext.GetTarget()).Dispose ();
((IDisposable)cairoContext).Dispose ();
```

The drawing code is fairly straight forward and requires little explanation. The only thing that you must be aware of when drawing to a computer screen is the fact that (0, 0) is in the top left, so *x* increases to the right and *y* downwards.

Thread Safety

Something to keep in mind when updating anything in a user interface is thread safety. In our system there are two separate threads of execution; the GTK thread containing the core of the system, and serial read thread. The GTK thread is receiving updates in the form of events from the serial thread around every 500ms but by default GTK is not thread safe (nor is WinForms or SWING) so updating GTK from outside of its thread will cause it to crash. However it can be made "thread aware" meaning that locks can be applied when needed. The problem and its solutions are covered in detail on the Mono projects website (Mono Project, 2014).

The solution that we applied involved GDK locking. While this not the recommended approach, it works in our case and it also required the least amount of time to implement. A better approach would be to use the *ThreadNotify* solution (Mono Project, 2014). Any events that are raised from outside of GTK are protected using GDK locking:

```
private void SerialProxy_OdometryUpdate (object sender, OdometryUpdateEventArgs e)

{

        Gdk.Threads.Enter ();

        try

        {

                robot.UpdateOdometry (e);

        }

        finally

        {

                Gdk.Threads.Leave ();

        }

}
```

Testing

Test 1: Sonar accuracy

The robot was placed approximately 0.3m directly in front of the target landmark. The results from 10 scans were:

0.2997, 0.3005, 0.1810, 0.2989, 0.2990, 0.3001, 0.1668, 0.2995, 0.3001, 0.2949

From this result set we can tell that the sonar is fairly accurate apart from in two of the above cases. The 3rd and 7th reading were off by up to 0.1332m, but from these results we can assume that the distance readings for our scans will be accurate approximately 80% of the time.

Apart from the two exceptional cases the range is accurate up to a point of 0.01m, which is within our acceptable limits of 0.01-0.02m when using sonar to determine landmark range. The outliers will be corrected by the EKF SLAM algorithm which will treat them as errors. This however makes the possibility of making the rover autonomous a more difficult challenge.

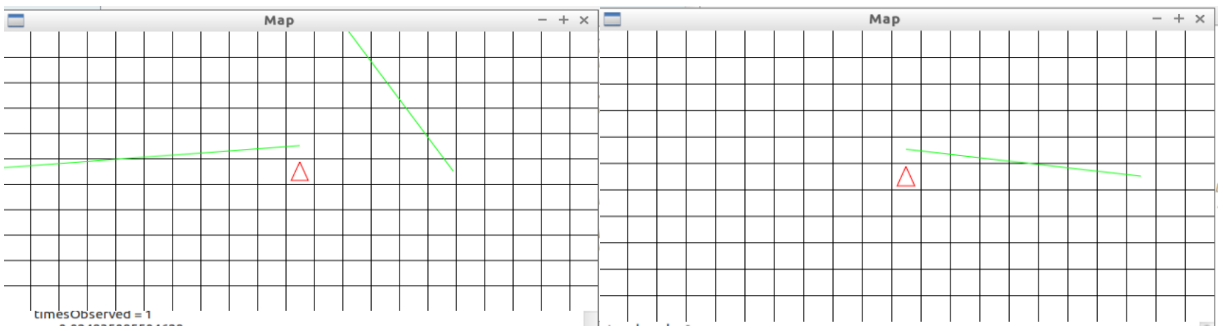


Figure 14 - Sample Landmarks

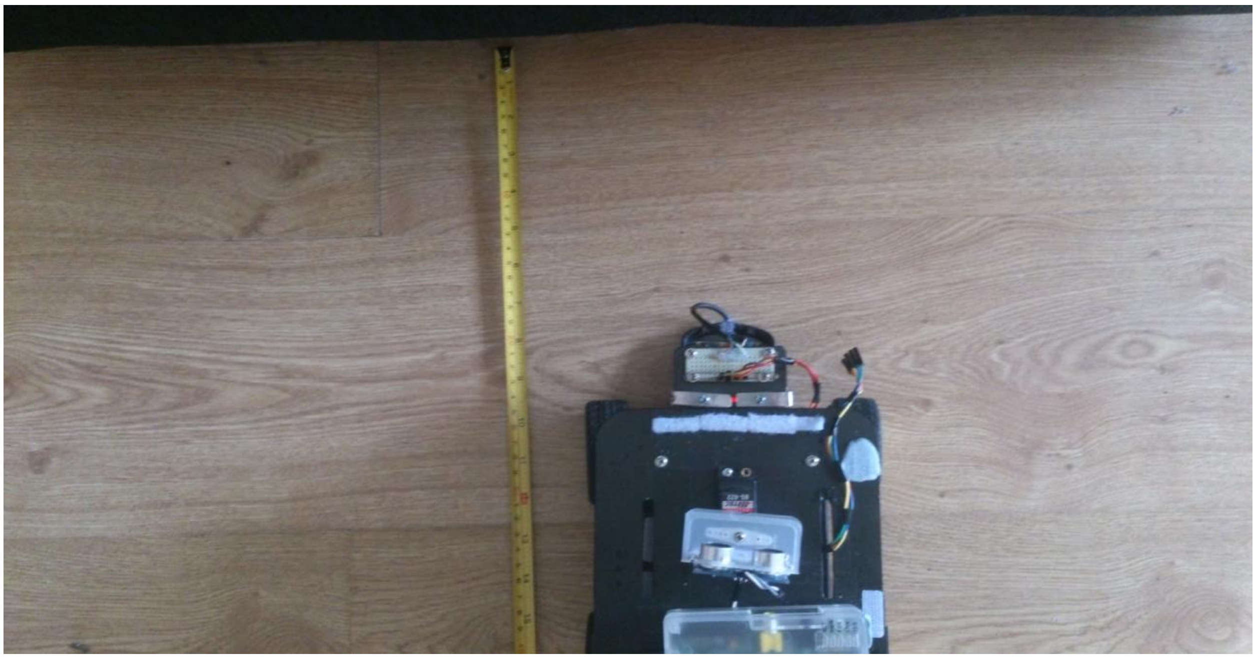


Figure 15 - Odometry Test

From the images above we can see that the position of the slope drawn on the map varies greatly. In an attempt to mitigate this the SLAM algorithm was updated to include data changes. For future work we advise that this change be pursued further as the SLAM algorithm provided does not correct all landmark values as it runs, it simply reaffirms if they exist or not. We did not proceed with updating the algorithm due to time constraints.

Test 2: Odometry

We noted after several tests on different surfaces that carpeted surfaces gave the best distance readings as opposed to lino or tile, in extreme cases the robot failed to read any movement while travelling on these surfaces. We concluded that this was due to the reflective nature of lino and tiled surfaces, which affects the precision of the optical flow sensor.

The table below shows distance measurements taken on a carpeted surface when travelling 30cm. The test was performed without the compass so as to only test the odometry without the curve skew. One set of tests was performed with the Arduino running off the Raspberry Pi's power supply, and the other set the Arduino has its own power supply:

Test Number	**Low Power (cm travelled)**	**Separate Source (cm travelled)**
1	21	30.5
2	23.5	24
3	25.5	31
4	31	32
5	31	32
6	23	33
7	21.5	24.5
8	22	33
9	25	30
10	21	31

Results show that the distance travelled using low power versus having dedicated power sources is significant. Power is a major issue when working with this type of equipment and low power leads to poor results.

We anticipated an error tolerance of between 1-3cm because of the delay in the odometry sensor, but in practise the results are accurate to under 1cm. It updates approx. every 0.2 seconds which gives the map inaccurate readings but it seems to work out. Any outliers in the readings are attributed to power fluctuations.

Another test was performed where a set of 5 readings was taken on the carpet, but the rover was aligned with the grain of the carpet. The purpose of this test was to see if the carpet added any resistance to the rover. The results are as follows: 22cm, 24cm, 31cm, 30cm, 29cm

From the below tests, based on one metre runs you can see a much higher error rate when the compass is added. There are several contributing factors to this result. First is the effect of nearby electronics and magnetic fields on the compass. In different areas of the room the accuracy of the map varied significantly as it passed under electronics such as an overhead projector, desktop computers, or large masses of metal.

With an average error rate of over 25% when the rotation is taken into account it makes it all but impossible to implement automation using the generated map, however it can easily be seen that the error is in the compass directions as the error rate for the other input devices was <5%.

Test Number	**x**	**y**	**Heading**	**Error**
1	0.485	0.729	5.4	0.271
2	0.461	0.764	5.5	0.236
3	0.367	0.686	5.6	0.314
4	0.476	0.698	5.4	0.302
5	0.467	0.716	5.6	0.284
6	0.470	0.692	5.4	0.308
7	0.408	0.720	5.5	0.28
8	0.445	0.782	5.58	0.218
9	0.433	0.679	5.42	0.321
10	0.484	0.752	5.45	0.248

Average claimed: 0.7218m

Average error: 0.2782m

Conclusion

The rover in its current state is what was set out to achieve. This projects system was low on processing power but still managed to successfully orientate and map itself and its surroundings. The project expanded and improved upon what was just a simple rover, and was done so by adding the Raspberry Pi and better sensors. There were difficulties making connections between the Arduino and the Raspberry Pi and between the Raspberry Pi and the remote laptop, but these were overcome like so many other small problems, to have a successful project to demonstrate. After extensively studying the SLAM for Dummies research paper that this project concludes that the algorithms were not effective or scalable. In retrospect a more effective attempt at running SLAM algorithms on low processing power would have employed gridSLAM or fastSLAM.

Further Work

Hardware

Task	Description	Importance
1	The front left motor is malfunctioning, it spins constantly even when told to stop. Becomes a real problem when the motor power supply is low. Replace it with a new one from www.dfrobot.com.	HIGH
2	The soldering on the motor shield is very poor and could be part of the issue with the malfunctioning motor. Replace it with a new pre-soldered motor shield from HobbyComponents.	HIGH
3	The HMC5883L magnetic compass needs to be calibrated to account for the declination angle at its location and any drift. It tends to throw the robots heading off.	NORMAL
4	Separate the servo’s power supply from the Arduino to save draining the 9V battery every scan. Instead run it off the 5AA cells that power the motors.	NORMAL
5	Get the Parallax Laser Range Finder from RobotShop, likely to be more accurate than the Ping))).	LOW

Software

Task	Description	Importance
1	Account for any noise and drift in both the compass and the mouse optical flow sensor to get more accurate results.	HIGH
2	Find a better SLAM algorithm, possibly one based on a grid such as GridSLAM on openslam.org. A grid based representation would make implementing autonomous navigation easier.	HIGH
3	Remove the GUI on the Raspberry Pi, run a headless algorithm, and instead view it on an external machine.	NORMAL
4	Use button controls on the GUI rather than the command line.	LOW

References

Arduino. (2014). Compare board specs. Available: http://arduino.cc/en/Products.Compare. Last accessed 30th April 2014.

Adafruit. (2014). Adafruit Motor/Stepper/Servo Shield for Arduino kit. Available: http://www.adafruit.com/products/81. Last accessed 30/04/2014.

Adafruit. (2010). Adafruit-Motor-Shield-library. Available: https://github.com/adafruit/Adafruit-Motor-Shield-library. Last accessed 30th April 2014.

Riisgaard, S. and Blas, M. (2005). Slam for Dummies. 1st ed. Available:

http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-412j-cognitive-robotics-spring-2005/projects/1aslam_blas_repo.pdf. Last accessed 30th April 2014.

Riisgaard, S. (2005). Slam for Dummies Personal Comments. Available:

http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-412j-cognitive-robotics-spring-2005/projects/soren_project.pdff. Last accessed 30th April 2014.

Adept Mobile Robotics. (2012). SICK LMS-200 Laser Rangefinder. Available: http://robots.mobilerobots.com/wiki/SICK_LMS-200_Laser_Rangefinder. Last accessed 30th April 2014.

Mono Project. (2014). Best Practices. Available: http://mono-project.com/Best_Practices. Last accessed 30th April 2014.

Poenar, S. (2013) Arduino Autonomous Rover. Undergraduate Thesis. Institute of Technology Blanchardstown

Knight, W. (2013) Driverless Cars Are Further Away Than You Think, MIT Technology Review, October 22nd 2013. Available:

http://www.technologyreview.com/featuredstory/520431/driverless-cars-are-further-away-than-you-think/. Last accessed 14/11/2013.

Durrant-Whyte, H. & Bailey, T. 2006, "Simultaneous localization and mapping: part I", IEEE Robotics & Automation Magazine, vol. 13, no. 2, pp. 99-110.

Appendix

List of Suppliers

Hobby Components
A quality UK supplier of electronic components. The HMC5883L compass and cables were obtained from here. For the case of the compass HobbyComponents price was a 10th of what RobotShop was charging. Delivery within 3 - 5 working days.

Website: www.hobbycomponents.com

Maplin
Local supplier to the UK and Ireland, off the self stock. The batteries, Pi case, and various other components were sourced from Maplin. The real advantage when compared to the Internet suppliers is that it can be collected on the day. However they lack the more exotic components such as sensors and usually everything comes at a higher price.

Website: www.maplin.co.uk

RobotShop
A huge online supplier of robots and components, very large range, good source of information. The DFPirate kit and Parallax Ping))) came from here, a little expensive at times.

Website: www.robotshop.com

Radionics
Global supplier with a branch in Ireland, similar to Maplin but more focused, normally cheaper. Although nothing was sourced from here they have a decent selection.

Website: www.radionics.rs-online.com

List of Components

Pirate DFRobot Platform

<u>Description:</u>

General purpose mobile robotics platform constructed using high strength aluminium alloy. It comes equipped with four geared 3V-6V DC motors for locomotion, three tiered layers, one for motor and power supply mounting, a second for microcontroller mounting, and a third layer for additional components.

<u>Data sheet:</u>

http://www.robotshop.com/media/files/pdf/instruction-mannual-rob0003.pdf

Last accessed: 30th April 2014

<u>Image:</u>

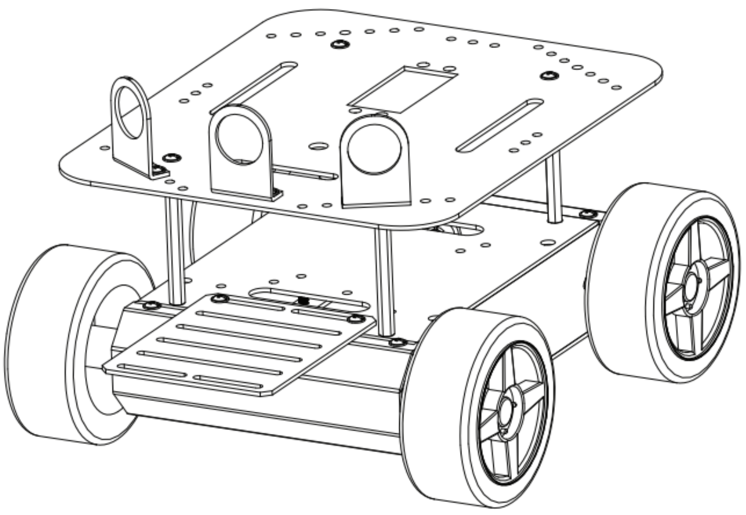


Figure 16 - Pirate DFRobot Platform

Arduino Uno Rev 3

Description:

One of the most common Arduino microcontroller boards, based on the 8bit ATMega 328 processor. The Uno comes with 14 digital I/O pins, 6 of which provide Pulse Width Modulation (PWM), and another 6 pseudo analog pins.

The Arduino Uno gives the rover the ability to communicate with all of the low level equipment such as the motors, compass, sonar, and so on.

Data sheet: http://arduino.cc/en/uploads/Main/Arduino_Uno_Rev3-schematic.pdf

Last accessed: 30th April 2014

Image:

Figure 17 - Arduino Uno Rev 3

Adafruit Motor Shield

<u>Description:</u>
A stackable expansion shield that gives the Uno the ability to drive up to four 3V-36V motors providing 0.6A per motor (1.2A peak). When attached to the Arduino it occupies all of the digital pins except for pin 13, the remaining 6 analog pins can be accessed using breakout pin headers.

An important feature included with the Adafruit motor shield is the option to split the motor power supply from the Arduino, this means that you can drive much higher voltage motors than the Arduino's peak voltage of 21V.

<u>Data sheet:</u>

<u>http://learn.adafruit.com/downloads/pdf/adafruit-motor-shield.pdf</u>

Last accessed: 30th April 2014

<u>Image:</u>

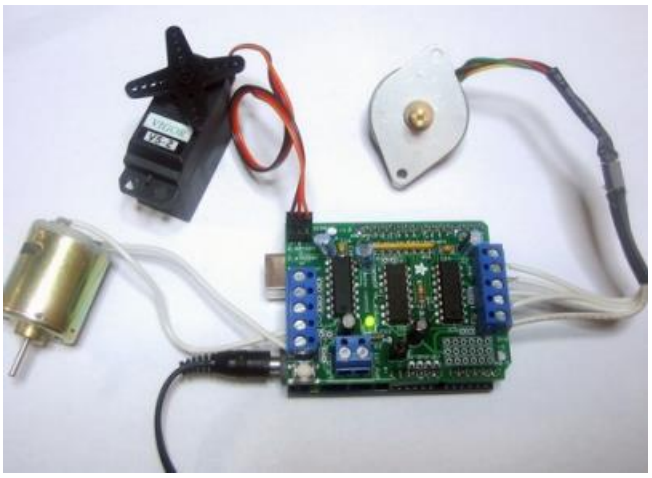


Figure 18 - Adafruit Motor Shield

HMC5883L Compass

Description:

Honeywell's HMC5883L is a tiny surface mount based electro magnetic compass capable of measuring the strength of the earth's magnetic field in 3 axis. It is designed as a low cost magnetometry solution with an accuracy between 1 and 2°.

One of its most important features is the simple interface it uses for communication; the I2C protocol, which the Arduino can easily participate in using its built in Wire library.

Data sheet:

http://jameco.com/Jameco/Products/ProdDS/2150248.pdf

Last accessed: 30th April 2014

Image:

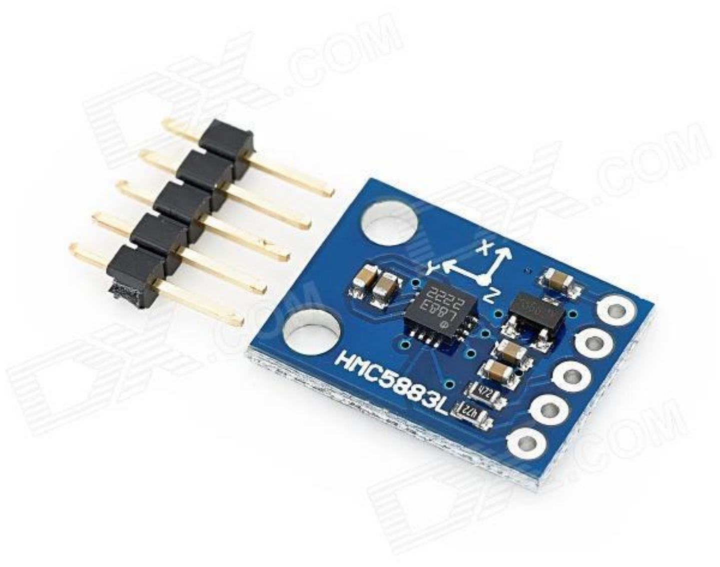


Figure 19 - HMC5883L Compass

Parallax Ping)))

Description:

Ultrasonic sensor that calculates the distance to an object based on the speed of sound and the duration that it took for the pulse it sent to return. The frequency of the pulse is ultrasonic and is well above human hearing.

The Parallax Ping))) is a non-contact sensor capable of measuring distances between 2cm and 3m and uses a single SIG pin for communication purposes, the sensor is widely used in the Arduino community and comes with example code.

Data sheet:

http://www.parallax.com/sites/default/files/downloads/28015-PING-Sensor-Product-Guide-v2.0.pdf

Last accessed: 30th April 2014

Image:

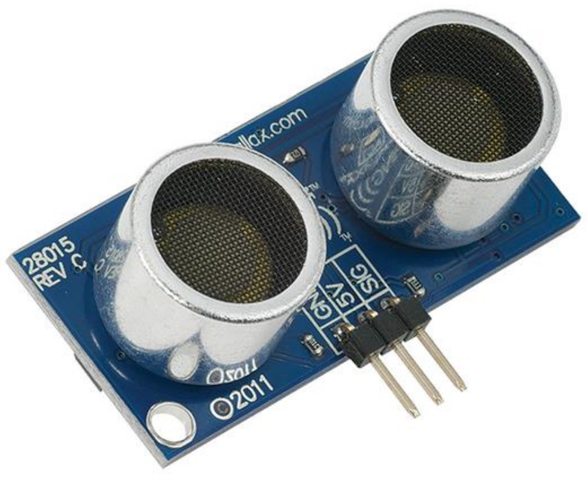


Figure 20 - Parallax Ping)))

Hitec HS-422 Servo

Description:

Hobby RC servo capable of covering an arc of rotation between 0° and 179°. It can easily be positioned using the Arduino Servo library which uses PWM. The HS-422 is used in conjunction with the Ping))) to create a 2D range finder capable of sweeping an area.

Data sheet:

http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Robotics/hs422-31422S.pdf

Last accessed: 30th April 2014

Image:

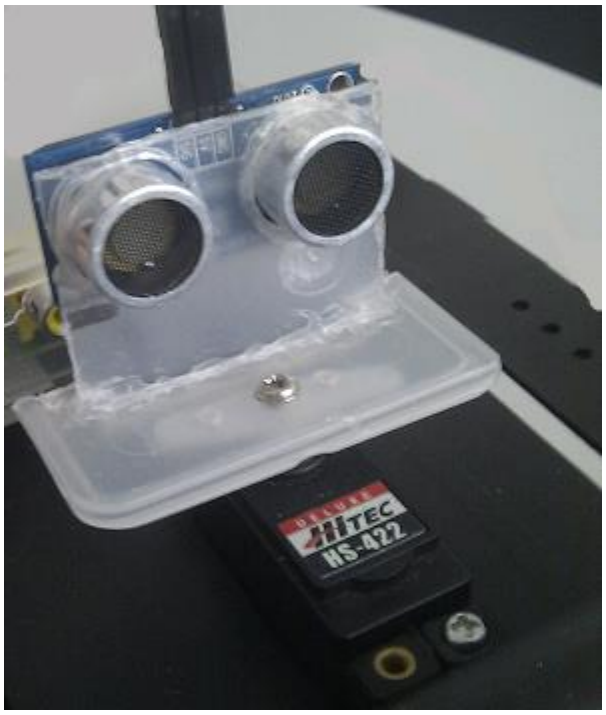


Figure 21 - Hitec HS-422 Servo with Sonar Mounted

Mouse Sensor PAW3504

Description:

The PAW3504 is a CMOS process optical mouse sensor single chip with USB interface that serves as a non-mechanical motion estimation engine for implementing a computer mouse.

Data sheet:

http://html.alldatasheetpt.com/html-pdf/333281/PIXART/PAW3504/1548/7/PAW3504.html

Last accessed: 30th April 2014

Image:

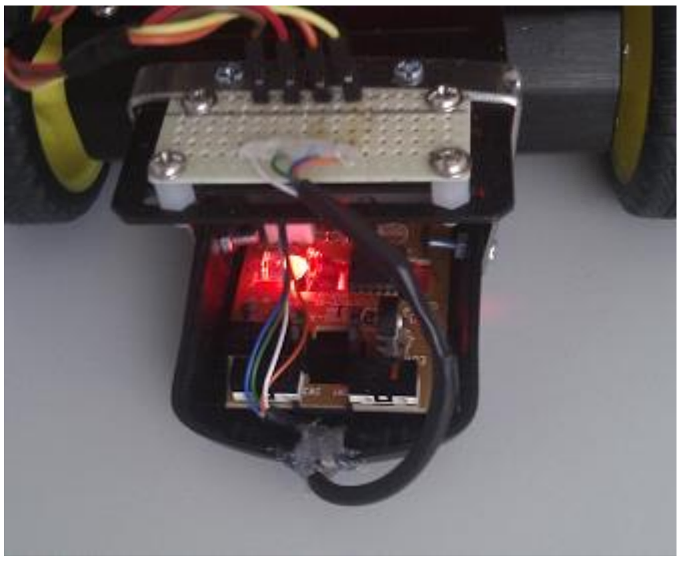


Figure 22 - Mouse Odometer Attached to the Rover

Raspberry Pi Model B

Description:

The Raspberry Pi is a credit-card sized computer. Model B is the higher-spec variant of the Raspberry Pi, with 512MB of RAM, two USB ports and a 100mb Ethernet port. The SoC is a Broadcom BCM2835. This contains an ARM1176JZFS, with floating point, running at 700Mhz; and a Videocore 4 GPU. The GPU is capable of BluRay quality playback, using H.264 at 40MBits/s. It has a fast 3D core accessed using the supplied OpenGL ES2.0 and OpenVG libraries.

Data sheet:

http://www.raspberrypi.org/help/faqs/

Last accessed: 30th April 2014

Image:

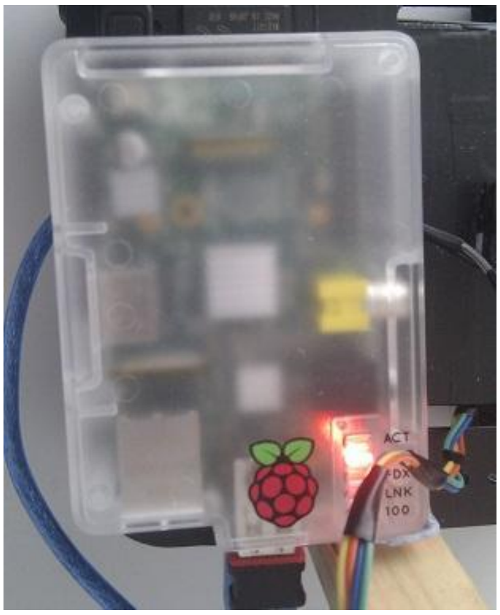


Figure 23 - Raspberry Pi in its Case on the Rover

Nano Wifi Dongle

Description:

The USB 2.0 N150 is a dongle for wireless networking between a device that has a USB port and another device transmitting a wireless signal. It is backward compatible with 802.11b/g and Nano Size.

Data sheet:

http://www.maplin.co.uk/p/maplin-single-band-n150-nano-usb-network-adapter-a71lb

Last accessed: 30th April 2014

Image:

Figure 24 - Maplin's Nano Wifi Dongle