Using the Raspberry Pi with the cRio robot: Lab Manual

April 2017

Logan Burden

INTRODUCTION

This lab manual is instructional on using the raspberry pi as the primary platform for running code, not the cRio. Initially, you will need to configure the Raspberry Pi. The Raspberry Pi is powered by 5V DC over micro USB, and as such a USB cable has been wired to the power distribution board for this purpose.

Python is the language used for this lab so a working install of Python2 is required. Raspbian has python installed by default, and this configuration will be assumed for the lab.

A Basic understanding of Linux is assumed for this lab manual.

INSTALLING FROM WINDOWS

From: https://www.raspberrypi.org/documentation/installation/installing-images/windows.md

Insert the SD card into your SD card reader. You can use the SD card slot if you have one, or an SD adapter in a USB port. Note the drive letter assigned to the SD card. You can see the drive letter in the left hand column of Windows Explorer, for example **G:**

- Download the Win32DiskImager utility from the Sourceforge Project page as an installer file, and run it to install the software.
- Run the `Win32DiskImager` utility from your desktop or menu.
- Select the image file you extracted earlier.
- Select the drive letter of the SD card in the device box. Be careful to select the correct drive: if you choose the wrong drive you could destroy the data on your computer's hard disk! If you are using an SD card slot in your computer and can't see the drive in the Win32DiskImager window, try using an external SD adapter.
- Click 'Write' and wait for the write to complete.
- Exit the imager and eject the SD card.

INSTALLING FROM LINUX

From: https://www.raspberrypi.org/documentation/installation/installing-images/linux.md

Please note that the use of the dd tool can overwrite any partition of your machine. If you specify the wrong device in the instructions below, you could delete your primary Linux partition. Please be careful.

DISCOVERING THE SD CARD MOUNTPOINT AND UNMOUNTING IT

- Run `df -h` to see what devices are currently mounted.

•If your computer has a slot for SD cards, insert the card. If not, insert the card into an SD card reader, then connect the reader to your computer.

•Run `df -h` again. The new device that has appeared is your SD card. If no device appears, then your system is not automounting devices. In this case, you will need to search for the device name using another method. The `dmesg | tail` command will display the most recent system messages, which should contain information on the naming of the SD card device. The naming of the device will follow the format described in the next paragraph. Note that if the SD card was not automounted, you do not need to unmount later.

•The left column of the results from `df -h` command gives the device name of your SD card. It will be listed as something like `/dev/mmcblk0p1` or `/dev/sdX1`, where X is a lower case letter indicating the device. The last part (`p1` or `1` respectively) is the partition number. You want to write to the whole SD card, not just one partition. You therefore need to remove that part from the name. You should see something like `/dev/mmcblk0` or `/dev/sdX` as the device name for the whole SD card. Note that the SD card can show up more than once in the output of `df`. It will do this if you have previously written a Raspberry Pi image to this SD card, because the Raspberry Pi SD images have more than one partition.

•Now you have noted the device name, you need to unmount it so that files can't be read or written to the SD card while you are copying over the SD image.

•Run `umount /dev/sdX1`, replacing `sdX1` with whatever your SD card's device name is (including the partition number).

•If your SD card shows up more than once in the output of `df` because it has multiple partitions on the SD card. You should unmount all of these partitions.

COPYING THE IMAGE TO THE SD CARD

•In a terminal window, write the image to the card with the command below, making sure you replace the input file `if=` argument with the path to your `.img` file, and the `/dev/sdX` in the output file `of=` argument with the correct device name. This is very important, as you will lose all data on the hard drive if you provide the wrong device name. Make sure the device name is the name of the whole SD card as described above, not just a partition of it. For example: `sdd`, not `sdds1` or `sddp1`, and `mmcblk0`, not `mmcblk0p1`.

```
dd bs=4M if=2017-02-16-raspbian-jessie.img of=/dev/sdX
```

•Please note that block size set to `4M` will work most of the time. If not, please try `1M`, although this will take considerably longer.

•Also note that if you are not logged in as root you will need to prefix this with `sudo`.

COPYING A ZIPPED IMAGE TO THE SD CARD

In Linux it is possible to combine the unzip and SD copying process into one command, which avoids any issues that might occur when the unzipped image is larger than 4GB. This can happen on certain filesystems that do not support files larger than 4GB (e.g. FAT), although it should be noted that most Linux installsations do not use FAT and therefore do not have this limitation.

The following command unzips the zip file (replace 2017-02-16-raspbian-jessie.zip with the appropriate zip filename), and pipes the output directly to the dd command. This in turn copies it to the SD card, as described in the previous section.

```
unzip -p 2017-02-16-raspbian-jessie.zip | sudo dd of=/dev/sdX
bs=4096
```

CHECKING THE IMAGE COPY PROGRESS

- By default, the `dd` command does not give any information about its progress and so may appear to have frozen. It can take more than five minutes to finish writing to the card. If your card reader has an LED, it may blink during the write process.

- To see the progress of the copy operation, you can run the dd command with the status option.

  ```
  dd bs=4M if=2017-03-02-raspbian-jessie.img of=/dev/sdd
  status=progress
  ```

- If you are using an older version of `dd`, the status option may not be available. You may be able to use the `dcfldd` command instead, which will give a progress report about how much has been written.

CHECKING IF THE IMAGE WAS CORRECTLY WRITTEN TO THE SD CARD

- After `dd` has finished copying, you can check what has been written to the SD card by `dd`-ing from the card back to another image on your hard disk; truncating the new image to the same size as the original; and then running `diff` (or `md5sum`) on those two images.

- If the SD card is bigger than the original image size, `dd` will make a copy of the whole card. We must therefore truncate the new image to the size of the original image. Make sure you replace the input file `if=` argument with the correct device name. `diff` should report that the files are identical.

  ```
  dd bs=4M if=/dev/sdX of=from-sd-card.img

  truncate --reference 2017-02-16-raspbian-jessie.img from-sd-
  card.img

  diff -s from-sd-card.img 2017-02-16-raspbian-jessie.img
  ```

- Run `sync`. This will ensure the write cache is flushed and that it is safe to unmount your SD card.

- Remove the SD card from the card reader.

Setting up the Hokuyo Laser Scanner

Install Urgbenri on hostcomputer

https://sourceforge.net/projects/urgbenri/

Installing BreezyLIDAR

Breezy Lidar can be found at:

https://github.com/simondlevy/BreezyLidar

BreezyLidar - Simple, efficient, Lidar access for Linux computers in Python and C++

This repository contains everything you need to start working with the popular Hokuyo URG-04LX Lidar unit on your Linux computer. It is designed for robotics applications on a single-board Linux computer like RaspberryPi or ODROID.

BreezyLidar was inspired by the Breezy approach to Graphical User Interfaces developed by Ken Lambert: an object-oriented Application Programming Interface that is simple enough for beginners to use, but that is efficient enough to scale-up to real world problems. As shown in the following code fragment, the API is extremely simple: a constructor that accepts the port (device) name on which the unit is connected, and method for accessing the scans (range values):

```
from breezylidar import URG04LX

laser = URG04LX('dev/tty/ACM0')

while True:

scan = laser.getScan()

# do something with scan, like SLAM
```

Installing for Python

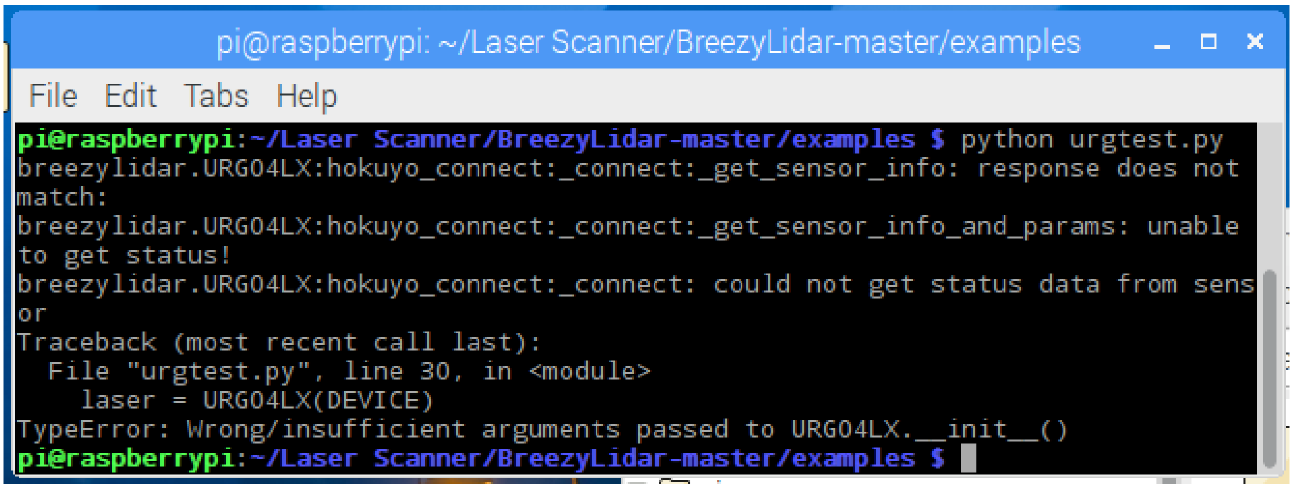
The BreezyLidar installation uses the popular distutils approach to installing Python packages, so all you should have to do is download the repositry, cd to the directory where you put it, and do

```
sudo python setup.py install
```

For a quick demo, you can then cd to `examples` and do

```
make test
```

TROUBLESHOOTING



```
pi@raspberrypi:~/Laser Scanner/BreezyLidar-master/examples $ python urgtest.py
breezylidar.URG04LX:hokuyo_connect:_connect:_get_sensor_info: response does not
match:
breezylidar.URG04LX:hokuyo_connect:_connect:_get_sensor_info_and_params: unable
to get status!
breezylidar.URG04LX:hokuyo_connect:_connect: could not get status data from sens
or
Traceback (most recent call last):
  File "urgtest.py", line 30, in <module>
    laser = URG04LX(DEVICE)
TypeError: Wrong/insufficient arguments passed to URG04LX.__init__()
pi@raspberrypi:~/Laser Scanner/BreezyLidar-master/examples $
```

Sometimes the laser scanner does not initiate properly if it is first called by BreezyLidar . This can be remedied by initiating it with other software or by plugging it into the Lab computer and initiating it in URGBenri standard and then unplugging the USB cable back into the Raspberry Pi without removing the power cable.

Using BreezyLIDAR

Included with Breezy Lidar in the examples folder is urgtest.py. This program can be executed with the bash command $python urgtest.py

Code Below:

```
from breezylidar import URG04LX

import sys

from time import time

DEVICE = '/dev/ttyACM0'

NITER  = 100

laser = URG04LX(DEVICE)

print('===================================================================')

print(laser)

print('===================================================================')

start_sec = time()
```

```
count = 0
for i in range(1, NITER+1):
    sys.stdout.write('Iteration: %3d: ' % i)
    data = laser.getScan()
    if data:
        print('Got %3d data points' % len(data))
        count += 1
    else:
        print('=== SCAN FAILED ===')
elapsed_sec = time() - start_sec
print('%d scans in %f seconds = %f scans/sec' % (count, elapsed_sec, count/elapsed_sec))
```

This is a very basic example that loads data from the laser scanner into an array and returns the length of that array.

Important things to note:

The Laser Scanner is accessible at /dev/ttyACM0

laser.getScan is used to retrieve data. It returns an array of int

Within the array, there are roughly 680 measured points. Values of less than 20 are undefined and should be treated as infinitely far away, values greater than 20 are approximate distances in millimeters.

To find which value in the array corresponds to which angle, viewing the laser scanner in Urgbenri and mouseover the plot, or switch to the array view to find the point in the array associated with the direction.

Setting Up GPIO

You will need the python GPIO module. It is installed by default in Rasbian.

Otherwise it can be found here: https://sourceforge.net/projects/raspberry-gpio-python/

To make sure that it is at the latest version:

```
$ sudo apt-get update
$ sudo apt-get install python-rpi.gpio python3-rpi.gpio
```

To install the latest development version from the project source code library:

```
$ sudo apt-get install python-dev python3-dev
$ sudo apt-get install mercurial
$ sudo apt-get install python-pip python3-pip
$ sudo apt-get remove python-rpi.gpio python3-rpi.gpio
$ sudo pip install hg+http://hg.code.sf.net/p/raspberry-gpio-python/code#egg=RPi.GPIO
$ sudo pip-3.2 install hg+http://hg.code.sf.net/p/raspberry-gpio-python/code#egg=RPi.GPIO
To revert back to the default version in Raspbian:
$ sudo pip uninstall RPi.GPIO
$ sudo pip-3.2 uninstall RPi.GPIO
$ sudo apt-get install python-rpi.gpio python3-rpi.gpio
```

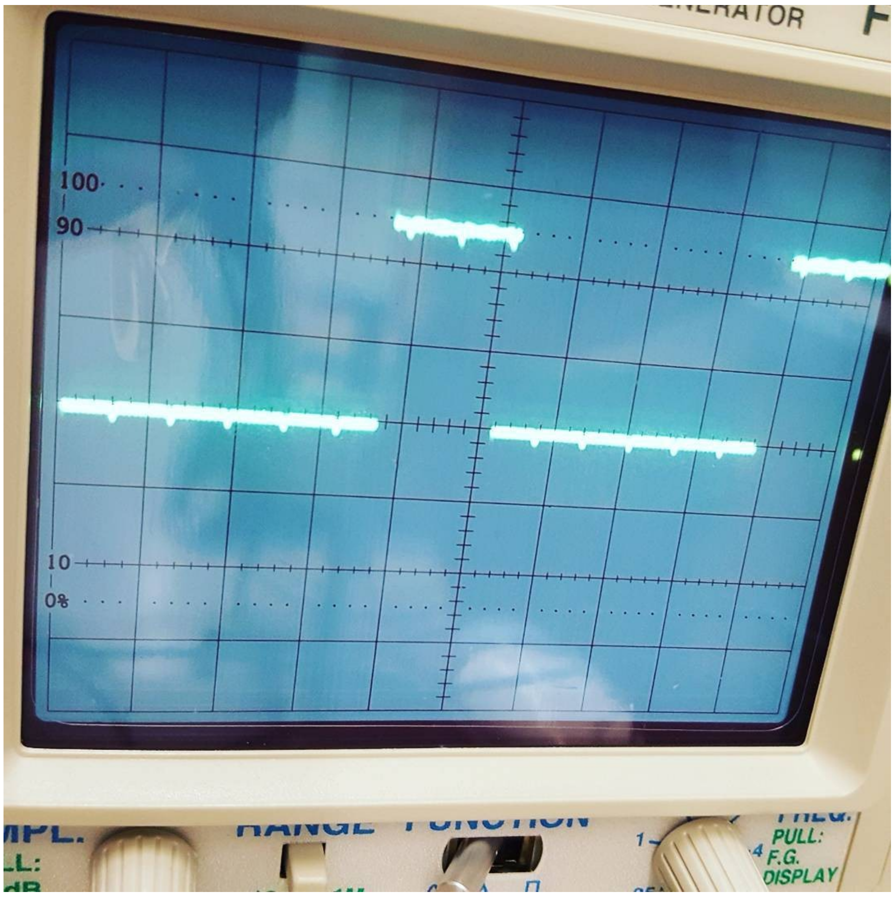
It can also be installed using pip

```
# pip install RPi.GPIO
```

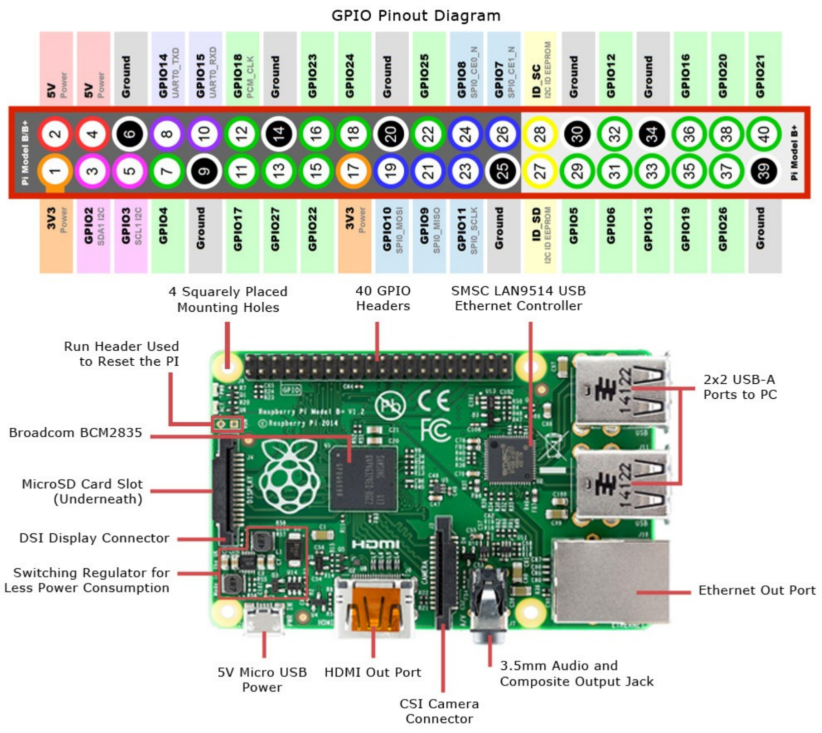
PWM Calibration

This is code that can be used to control the Talon Servo controllers

```
import RPi.GPIO as IO

from fcntl import ioctl

IO.setmode(IO.BOARD)

IO.setup(21, IO.OUT)

IO.setup(35, IO.OUT)

p = IO.PWM(21, 192.3)

d = IO.PWM(35, 192.3)
```



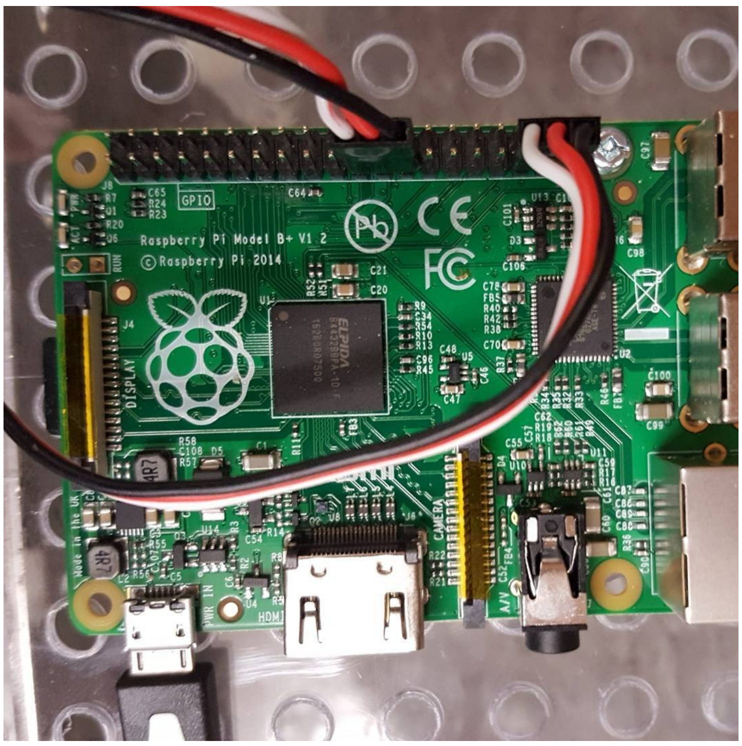
Scope Displaying a 192.3Hz Signal with a duty cycle of 27



This configures IO on pins 21 and 35 and sets the wave speed to 192.3Hz which was measured to be the correct frequency for controlling the Talon Motor controllers.

To properly connect the Servo Controllers to the Raspberry Pi, the black wire for one servo should be connected to the lower right pin when the raspberry pi logo is oriented upwards, and the other should be seven pins to the left (4 pin space between the connectors)

In the diagram to the left the black wires should be on pins 25 and 39 as grounds. Pins 21 and 35 should be connected to the white wires for control.



Raspberry Pi with Talon Servo Controllers attached to pins 21 and 35

Joystick Code

js_linux.py is a sample program modified from https://gist.github.com/rdb/8864666

This Program printed joystick states to the screen, it has been modified to serve as a simple program to drive the robot using a dual stick joystick and GPIO using tank controls. Running this program with $python js_linux.py should allow you to remotely operate a properly configured robot with a USB joystick connected to the Raspberry Pi. The Joystick is assumed to be at /dev/input/js0.

NOTE: If you are using the wireless Joystick, the robot will most likely lurch forward until the joystick is synced. Do not run this code where this occurring would be dangerous or cause damage to the robot or other lab equipment.

Relevant code included below.

```
import os, struct, array
import RPi.GPIO as IO
from fcntl import ioctl
IO.setmode(IO.BOARD)
IO.setup(21, IO.OUT)
IO.setup(35, IO.OUT)
p = IO.PWM(21, 192.3)
d = IO.PWM(35, 192.3)
# Iterate over the joystick devices.
print('Available devices:')
for fn in os.listdir('/dev/input'):
    if fn.startswith('js'):
        print('  /dev/input/%s' % (fn))

# We'll store the states here.
axis_states = {}
button_states = {}

############Name Stuff is her in the source code

axis_map = []
button_map = []

# Open the joystick device.
fn = '/dev/input/js0'
print('Opening %s...' % fn)
jsdev = open(fn, 'rb')

# Get the device name.
```

```
#buf = bytearray(63)
buf = array.array('c', ['\0'] * 64)
ioctl(jsdev, 0x80006a13 + (0x10000 * len(buf)), buf) # JSIOCGNAME(len)
js_name = buf.tostring()
print('Device name: %s' % js_name)
# Get number of axes and buttons.
buf = array.array('B', [0])
ioctl(jsdev, 0x80016a11, buf) # JSIOCGAXES
num_axes = buf[0]
buf = array.array('B', [0])
ioctl(jsdev, 0x80016a12, buf) # JSIOCGBUTTONS
num_buttons = buf[0]
# Get the axis map.
buf = array.array('B', [0] * 0x40)
ioctl(jsdev, 0x80406a32, buf) # JSIOCGAXMAP
for axis in buf[:num_axes]:
    axis_name = axis_names.get(axis, 'unknown(0x%02x)' % axis)
    axis_map.append(axis_name)
    axis_states[axis_name] = 0.0
# Get the button map.
buf = array.array('H', [0] * 200)
ioctl(jsdev, 0x80406a34, buf) # JSIOCGBTNMAP
for btn in buf[:num_buttons]:
    btn_name = button_names.get(btn, 'unknown(0x%03x)' % btn)
    button_map.append(btn_name)
    button_states[btn_name] = 0
print '%d axes found: %s' % (num_axes, ', '.join(axis_map))
print '%d buttons found: %s' % (num_buttons, ', '.join(button_map))
# Main event loop
while True:
    evbuf = jsdev.read(8)
    if evbuf:
        time, value, type, number = struct.unpack('IhBB', evbuf)
        if type & 0x02:
                                                    if (number == 1):
                                                                axis_states[axis] = value
                                                                fvalue = -value / 32767.0
                                                                d.start(fvalue*10 + 27)
                                                                print "%s: %.5f" % ("left", fvalue)
                                                    if (number == 3):
                                                                axis_states[axis] = value
                                                                fvalue = -value / 32767.0
                                                                p.start(-fvalue*10 + 27)
                                                                print "%s: %.5f" % ("right", fvalue)
```

The only code relevant to the operation of the robot are the GPIO inputs and initializers at the top discussed earlier, and the main event loop, the rest is related to getting joystick data. This is a quick and dirty experiment to make sure that the robot behaves correctly when given GPIO signals.

In the main eventloop, axis 1 is the left Y axis and axis 3 is the right Y axis. d.start(int) and p.start(ist) are used to change the duty cycle of the left and right servos. 27 means that the cycle is up 27% of the time. This is a neutral cycle. The servo will move very little or not at all if this cycle is sent.

A cycle greater than 27 but less than ~50 will cause the Servo to move forward, less than 27 but greater than ~0 will cause it to move back. Since the servos are oriented 180 degrees from each other the right servo must be reversed.

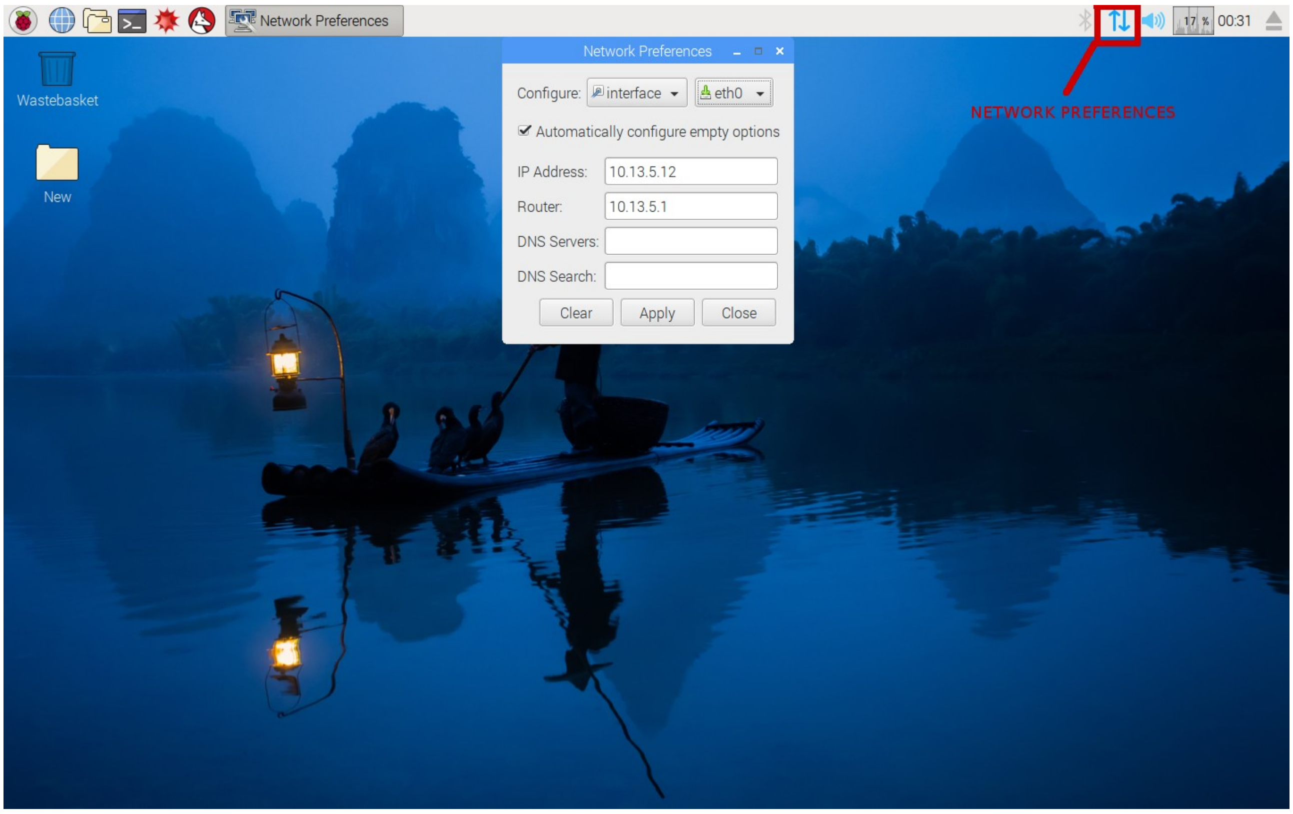
INSTABILITY

Due to the difficulty of running real time code on top of a kernel, there is some instability noticeable when the robot is not moving. The servo controllers will flash lights back and forth between moving forward and backward, and occasionally the robot will jitter. These problems become worse if the raspberry pi is taking input from mouse or keyboard.

These problems could be alleviated with the addition of a breakout board designed to output PWM from hardware, used as an intermediary between the Raspberry Pi and the Servo Controllers.

Teleoperation of the Robot

Since the Raspberry Pi is running Linux, the easiest way to remotely access it is via ssh either from another Linux machine, or via a windows machine using PuTTY. If the raspberry pi is intended to interact with the cRio, it should be on the same subnet. Generally the best way to do this is to right click on the network preferences in the upper right hand corner of the screen next to the processor usage meter, select eth0, and set the network device to use a static IP. The Raspberry pi should be 10.13.5.12, and router should be 10.13.5.1



Network Preferences

Configure: interface eth0

☑ Automatically configure empty options

IP Address: 10.13.5.12

Router: 10.13.5.1

DNS Servers:

DNS Search:

Default Location of Network Preferences, and compatible FIRST static IP settings

Accordingly, the computer you wish to access the Pi from should be connected to the NIPISSING ROBOTICS network, and have that network device set to use a static IP of 10.13.5.X where X is a number between 5 and 9.

Alternatively, both can be set to have dynamic IP addresses, but neither will be able to communicate with the cRIO due to being on a different subnet.

Once both devices are properly configured, run the command $ssh 10.13.5.12 or connect to 10.13.5.12 in PuTTY. If you used Dynamic IP addresses, this will be different, use $ifconfig on the Raspberry Pi to find the correct IP

If you successfully connect to the Raspberry Pi, you should receive a prompt for User name and Password

Username: pi

Password: raspberry

You should now have a CLI for the Raspberry Pi and can now execute files, and execute scripts as though you were using a terminal window on the Pi this eliminates the need for a wired connection to the robot.

Programing Autonomous Behavior

urgtest2.py through urgtest5.py are variants on the original urgtest.py with different data displayed. urgtest4.py and urgtest5.py have basic autonomous behavior included.

urgtest4.py is included below and has only basic instructions for turning away from obstacles.

```
from breezylidar import URG04LX

import sys

#import time

import os, struct, array

import RPi.GPIO as IO

from fcntl import ioctl

from time import time

IO.setmode(IO.BOARD)

IO.setup(21, IO.OUT)

IO.setup(35, IO.OUT)

le = IO.PWM(21, 192.3)

ri = IO.PWM(35, 192.3)

DEVICE = '/dev/ttyACM0'

laser = URG04LX(DEVICE)

print('===============================================================')

print(laser)

print('===============================================================')

start_sec = time()

while True:

    data = laser.getScan()

    turning = False

    for x in range (180, 500):

        if (data[x] >20) and (data [x] < 550):

            turning = True

    if (turning): #turn

                ri.start(20)

        le.start(20)

        print "turning: %d"%(data[350])

    else: #drive forward

        ri.start(30)

        le.start(24)

        print "drive"

 #   time.sleep(1)

else:

    print('=== SCAN FAILED ===')

elapsed_sec = time() - start_sec
```

Basic if statements are used to check for obstacles in front of the robot. 180,500 is a sweep in front of the robot of nearly 180 degrees, and 550mm is the required clearance for the robot to pivot in place. urgtest5.py is similar, with additional if statements to attempt recovery if caught on an obstacle.

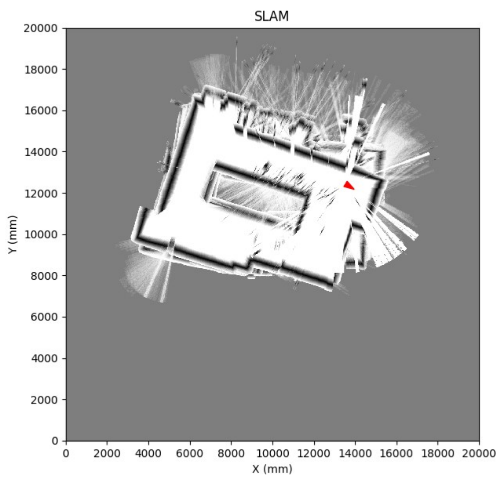
BreezySLAM

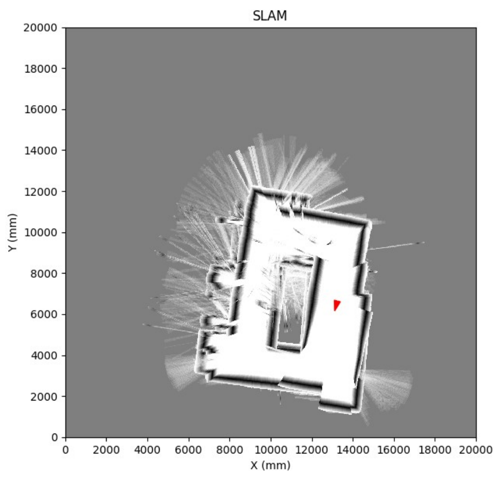
BreezySLAM can be found at: https://github.com/simondlevy/BreezySLAM

BreezySLAM can be installed alongside BreezyLIDAR to demonstrate a SLAM algorythm. Unfortunately, this implementation is to intensive to run on a Raspberry Pi and thus must be done with a USB-IP solution or remotely accessing /dev/ttyACM0 to access the Hokuyo laser scanner from a more powerful system.

Alternatively, as a work around a laptop can simply be attached to the robot, and work in tandem with the raspberry pi, handling slam, navigation and scanning, with the Pi handling servo control.

With some patience, the robot was able to generate some maps of the lab.


SLAM
Y (mm)
X (mm)
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000


SLAM
Y (mm)
X (mm)
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000