Integrating ROS, a Raspberry Pi, and a SeaPerch

Authors:
John McLinden
Contact: john_mclinden@my.uri.edu
Drew Mulcare
Contact: drewmichael1227@gmail.com

Setting Up Your Raspberry Pi:

This initial project is running on Ubuntu 20.04 on a Raspberry Pi (Model 3B/3B+) and ROS Noetic. Previous attempts at following the official ROS beginner tutorials (http://wiki.ros.org/ROS/Tutorials) using Raspbian Buster ran into issues when trying to get the beginner tutorial software (though it is almost certainly feasible, see the tutorials at http://wiki.ros.org/ROSberryPi). We followed both the tutorials for ROS Kinetic and ROS Melodic, but both versions seemed to be more compatible with Ubuntu than with Raspbian. Some benefits using Ubuntu include:

- ROS officially targets Ubuntu 20.04
- It allows us to use ROS Noetic, the most recent version of ROS (07/22/2020)

Installing and configuring Ubuntu on the raspberry pi was done by following the tutorial found at: https://ubuntu.com/tutorials/how-to-install-ubuntu-on-your-raspberry-pi#1-overview.
Information about setting up ROS Noetic for Ubuntu can be found at: http://wiki.ros.org/noetic/Installation/Ubuntu.

Below are the steps we took to configure ROS Noetic on a Raspberry Pi 3B/B+ running Ubuntu 20.04.

1. Using the Raspberry Pi imager (https://www.raspberrypi.org/downloads/), flash an Ubuntu 20.04 image to a microSD card (at least 8 GB).
2. Before attempting to boot the image on the pi, open the network-config file now found on the microSD card. There should be a section that looks like this:

```
#wifis:
#  wlan0:
#    dhcp4: true
#    optional: false
#    access-points:
#      <wifi network name>:
#        password: <"wifipassword">
```

   Uncomment the section and replace `<wifi network name>` with the name of your wifi network (may need to be in quotes if there is a space in the name). Replace

`<"wifipassword">` with your wifi password. Save the file, safely remove the microSD card, and insert it into the Pi to begin your first boot.

3. After booting, you will need to change the default password (default password is "ubuntu").
4. (Optional, but required to run the beginner tutorials on your pi): Install a desktop environment. The environment we are currently using is lubuntu, as it is relatively lightweight, though still fairly slow.
   ```
   sudo apt install lubuntu-desktop
   ```
   Afterwards, you should be able to set up your Pi as normal ()
5. Setup your sources list to accept packages from packages.ros.org:
   ```
   sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
   ```
6. Setup your keys:
   ```
   sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
   ```
7. Ensure that everything is updated:
   ```
   sudo apt update
   ```
8. Install ROS Noetic. The desktop version is sufficient to run all of the beginner tutorials (some include rqt/rviz).
   ```
   sudo apt install ros-noetic-desktop
   ```
9. Edit ~./setup.bash file to allow for automatic sourcing of /opt/ros/noetic/setup.bash on every new shell:
   ```
   echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc

   source ~/.bashrc
   ```
   From here, you should be able to follow the ROS beginner tutorials: http://wiki.ros.org/ROS/Tutorials. Be sure to configure the Pi for interfacing by following the steps in the next section.

Configuring Pi for Interfacing:

I2C- In order to properly access the I2C line on the Raspberry Pi with Ubuntu, here are a few steps that you need to take:

1. Install raspi-config. This is normally only intended for use in Raspbian, but appears to work in Ubuntu. This is also an older version of raspi-config. Details about this workaround to access the I2C bus are found here: https://askubuntu.com/questions/1130052/enable-i2c-on-raspberry-pi-ubuntu
   a.
      ```
      wget https://archive.raspberrypi.org/debian/pool/main/r/raspi-config/raspi-config_20160527_all.deb -P /tmp
      ```

   b. `sudo apt-get install libnewt0.52 whiptail parted triggerhappy lua5.1 alsa-utils -y`
   c. `sudo apt-get install -fy`
   d. `sudo dpkg -i /tmp/raspi-config_20160527_all.deb`
   e. `sudo mount /dev/mmcblk0p1 /boot`
2. Run raspi-config:
   `sudo raspi-config`
   In the GUI, select “Advanced Interfacing Options”, and enable the I2C bus.
3. Create an I2C user group
   a. `sudo groupadd i2c`
4. Change the group’s ownership
   a. `sudo chown :i2c /dev/i2c-1`
5. Change file permissions
   a. `sudo chmod g+rw /dev/i2c-1`
6. Add your user to the I2C group
   a. `sudo usermod -aG i2c username`
   b. Replace `username` with your username
7. Reboot so the changes take effect
   a. Logging off and logging on again also works
   b. The goal is to get `i2cdetect -y 1` to work
8. Make the changes permanent
   a. `sudo su`
   b. `# echo 'KERNEL=="i2c-[0-9]*", GROUP="i2c"' >> /etc/udev/rules.d/10-local_i2c_group.rules`

This information was adapted from a tutorial that can be found at:
https://lexruee.ch/setting-i2c-permissions-for-non-root-users.html

GPIO- In order to use the GPIO pins on the Raspberry Pi with Ubuntu, here are a few steps that you need to take:

Python 3

1. `sudo apt-get update`
2. `sudo apt-get upgrade`
3. `sudo apt-get install python3-pip python3-dev`
4. `sudo pip3 install RPi.GPIO`

Python 2

1. `sudo apt-get update`
2. `sudo apt-get upgrade`
3. `sudo apt-get install python-pip python-dev`
4. `sudo pip install RPi.GPIO`

Both

1. Similarly, create a GPIO user group:
   a. `sudo groupadd gpio`
2. Change ownership of the directory /dev/mem to the gpio group
   a. `sudo chown :gpio /dev/mem`
3. Change file permissions
   a. `sudo chmod g+rw /dev/mem`
4. Add your user to the group
   a. `sudo usermod -aG gpio username`
   b. Replace `username` with your username
5. Navigate to `/etc/udev/rules.d` and create a text file `99-gpio.rules`.
   a. `cd /etc/udev/rules.d`
   b. `sudo nano 99-gpio.rules`
6. Add the following line to the file:
   a. `SUBSYSTEM =="bcm2835-gpiomem", GROUP="gpio", MODE="0660"`
7. Reboot

If this does not work, there is another way to do this, in which the Pi will run a shell file at bootup that will run these commands each time. The steps below, which were adapted from https://stackoverflow.com/a/39225774, show you how.

1. Download the launcher script from GitHub
   a. `cd /bin/`
   b. `sudo wget https://raw.githubusercontent.com/dmichael1227/ROSPerch/master/rosperch/change-gpio-perms.sh`
   c. `sudo chmod 755 change-gpio-perms.sh`
2. Add this file to the Crontab so it runs on boot
   a. `sudo crontab -e`
   b. Add this to the very end of the file that opens up:
      i. `@reboot sudo sh /bin/change-gpio-perms.sh &`
3. Reboot
   a. `sudo reboot`

This information was adapted to from https://raspberrypi.stackexchange.com/questions/81570/installing-rpi-gpio-on-ubuntu-core, which just covers Python 2, but we found that it is compatible with Python 3.

NOTE: It is important to note that any code that uses the GPIO pins will need to be run as the root user. UPDATE: this may no longer apply if the steps above were followed to create a new set of udev rules. However, the following explanation of how subprocesses work could still be useful.

For example:

Python 3

```
sudo python3 file.py
```

Python 2

```
sudo python file.py
```

However, with ROS, things are a bit more complicated. Using `rosrun` to run a script cannot be executed as a root user. As a workaround, we have used short scripts called with the `subprocess` library in our ROS node scripts when we need to access the GPIO library. An example of this can be found in the `led_receiver.py` script. Upon receiving a Boolean message on the topic `ledstuff`, the script calls another script, `led_blinker.py`, with the necessary root privileges to use the library and the received message as an argument (a cumbersome process that should be changed, if possible). Our current understanding of this issue is that the GPIO library is not intended for Ubuntu, so the permissions that are normally set automatically in Raspbian are not set during Ubuntu installation.

In Python, this work-around can be accomplished using the subprocess library, including the `subprocess.call()` command. This command allows you to create and communicate with subprocesses. More information on the use of this command can be found at: https://pymotw.com/2/subprocess/ and https://queirozf.com/entries/python-3-subprocess-examples. Being able to spawn and communicate with other processes means that within one Python script, `subprocess.call()` can spawn another script, and, most notably, it can run it with root privileges using `sudo`. Running the script with root level privileges allows for the use of GPIO pins, which solves the problem we were encountering. More information about subprocesses is available under the **Subprocess Note** section.

Setting Up a Ubuntu Virtual Machine:

Using Ubuntu on a virtual machine provides a Linux environment to develop code and familiarity with Linux without the computing power limitations that a Raspberry Pi brings and without the potential damage that comes from dual-booting a PC for the first time. (Using a Linux machine or dual-booting Linux are best saved for more experienced users.) The Linux virtual machine also makes connecting to the Raspberry Pi easier.

The following instructions are adapted from the “Course Preparation Instructions” PDF available at https://rsl.ethz.ch/education-students/lectures/ros.html#course_material and the tutorial at https://medium.com/riow/how-to-open-a-vmdk-file-in-virtualbox-e1f711deacc4. The course materials that ETH Zurich provides include a YouTube series with accompanying presentations, exercises, and relevant material for learning ROS. These lessons are based in C++, but the very first lecture video and accompanying exercise are very educational introductions, regardless of programming language.

1. Download the “Ubuntu_18_04_ROS_COURSE.zip” file from https://polybox.ethz.ch/index.php/s/iSwkLBhRJNKuv8a
    a. This file is about 5.6GB, so make sure you have room
2. Download VirtualBox from https://www.virtualbox.org/wiki/Downloads and install it
3. Open VirtualBox and click the button to create a new machine

4. Name your machine
    a. Take note of the destination listed after “Machine Folder:”
    b. If it does not do so automatically, change the “Type:” field to “Linux” and change the “Version:” field to “Ubuntu (64-bit)”
    c. Hit “Next” to continue
5. Stepping away from VirtualBox for a moment, unzip the “Ubuntu_18_04_ROS_COURSE.zip” folder to the destination that you noted in the previous step, and then pull VirtualBox back up to continue configuring your machine
6. Choose however much memory (RAM) you would like to be allocated to the virtual machine, and click “Next”
    a. The virtual machine runs fine with 1024 MB of memory (RAM), but the machine does tend to lag when you have multiple terminals and an internet browser open, so going higher is helpful depending on how you will use the machine
7. For a “Hard disk,” choose the “Use an existing virtual hard disk file” option, and use the file shaped button on the right.
8. From here, choose the “Add” button from the top left. This will open up your file explorer, where you can navigate to the destination the “Ubuntu_18_04_ROS_COURSE.zip” was extracted to in Step 5.
9. Choose the file named “Ubuntu_18_04_ROS_Course-cl1.vmdk” and press open
10. Click through any prompts that show up, and finally click the “Create” button to create the virtual machine
11. To run the virtual machine, select it from the menu on the left, and click the green “Start” arrow on the top menu bar.
    a. The default password for this image is “student”

Alternatively, you can download Ubuntu 20.04, which supports ROS Noetic, however you will have to install ROS on that machine manually.

1. Download the file from https://sourceforge.net/projects/linuxvmimages/files/VirtualBox/U/20.04/Ubuntu_20.04_VB.zip/download
2. Unzip the file and open VIrtualBox
3. Select “Tools” and then “Import”
4. Where it asks for the file location, navigate to wherever the file was unzipped to in Step 2 and select the file
5. Continue clicking through the prompts to create the machine, and then boot it
    a. Default username and password are both “ubuntu”
6. Follow the steps at http://wiki.ros.org/noetic/Installation/Ubuntu to install ROS Noetic. The commands from these steps are listed in order below.
    a. `sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'`
    b. `sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654`
    c. `sudo apt update`

d. `sudo apt install ros-noetic-desktop-full`

e. `source /opt/ros/noetic/setup.bash`

f. `echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc`

g. `source ~/.bashrc`

7. From there, follow the steps at http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment to set up and configure your workspace environment.

ROS Tutorials:

The ROS tutorials, available at http://wiki.ros.org/ROS/Tutorials, are a great resource for getting started, and cover the use of both C++ and Python. (We opted for the Python route to allow for faster prototyping and ease of programming.) The tutorials walk you through configuring the environment, and introduce you to ROS's Nodes, Topics, Messages, Publishers, Subscribers, and other fundamental core concepts of ROS. The scripts (specifically `talker.py` and `listener.py`) for these sections of the tutorials are actually the basis for what we used to create the LED Blinker.

Later on in the tutorial, .bag files are introduced. These files can be used to log and store data, whether it be commands, sensor data, or robot behavior. We found that this particular type of file could potentially be used to record behaviors and play them back to your robot later on, or even just extract the data. This information is covered here and here.

LED Blinker:

Here is an example of a ROS talker and listener that takes a raw input from the user in the form of either `0` or `1`, and then turns the LED off or on respectively. To accomplish this, a string called `command_input` is defined as whatever the user puts in. (In the case of the current state of the code, only input values of 0 and 1 are valid.) Then, due to the nature of the `input()` command, the string is then converted to a `float`, which is then converted to a `bool`. This allows `led_reciever.py` to publish the user's input to the `led_command` topic, which is then read by the subscriber, `led_reciever.py`. The subscriber reads the message, and then, using `subprocess.call()` as outlined in the **Configuring Pi for Interfacing** section's note, calls a Python script, `led_blinker.py` using the Boolean message as an argument. This argument takes the form of `data.data` in this line of code:

```
subprocess.call(['sudo','python3','led_blinker.py','%s' % data.data])
```

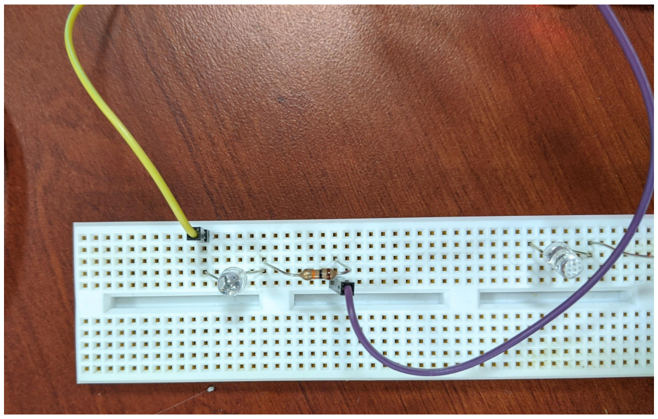
The led_test Package and the talker.py/listener.py Scripts

The talker.py/listener.py scripts are meant to be run in conjunction, similar to those found on the ROS tutorial site. To run these, in three separate shells, run:

1. `roscore`
2. `rosrun led_test talker.py`
3. `rosrun led_test listener.py`

Assuming the bme280 is wired in accordance with the tutorial that the `talker.py` script is largely based on:
https://www.raspberrypi-spy.co.uk/2016/07/using-bme280-i2c-temperature-pressure-sensor-in-python/

The `talker.py` script should begin outputting the temperature as a string to the ledstuff topic, while the `listener.py` script should blink the LED (connected to pin 8 of the GPIO header) and print the topics read on the ledstuff topic. This script demonstrates an interface with an external sensor that is capable of sending/receiving sensor data in the ROS framework.



To set up the circuit (pictured above) for this example, use the output from GPIO pin 8 (purple wire) on the Raspberry Pi to power the LED, and use any of the Pi's ground pins to ground it (yellow wire). Be sure to add a resistor between the power from the Pi and the LED. Refer to https://www.raspberrypi.org/documentation/usage/gpio/ for the pinout of the Raspberry Pi. The code for this example can be found on our GitHub: github.com/dmichael1227/led_test

The temp_read_publish.py and temp_listen_convert.py Scripts:

`temp_read_publish.py` and `temp_listen_convert.py` are an extension of the concepts covered in the (led_test) `talker.py` and `listener.py` scripts. The latter pair of scripts communicates the temperature read by the sensor as a string and prints it in Celcius by using the standard message type String. While the standard message types are useful for sending simple messages, ROS messages are capable of greater complexity, sending multiple values of multiple data types at once. To this end, we have created the BME message type to handle the publishing of everything useful that we receive from the BME280 sensor. This was done by adding the `BME.msg` file to the `/msg` directory in the led_test package. This is a very simple file consisting of three lines:

```
float64 temp
float64 pressure
float64 humidity
```

The lines above state that this message type consists of three 64-bit float values: temperature, pressure, and humidity. The BME.msg file was included in the `add_message_files()` section of `CMakeLists.txt` for the led_test package. The `temp_read_publish.py` script is analogous to the `talker.py` script; it reads the data from the BME280 sensor and publishes to a topic (this time bme280 instead of chatter). The difference here is that it now communicates temperature, pressure, and humidity instead of only temperature. Similarly, `temp_listen_convert.py` is analogous to `listener.py` in that it receives the data sent (this time listening on the bme280 topic) by the other script. `temp_listen_convert.py` receives the temperature, pressure, and humidity values and converts the temperature to Fahrenheit with a simple calculation before displaying the values in the log.

Making the Package:

To get this ROS package to work properly, you must first run the command `catkin_create_pkg led_test std_msgs rospy roscpp` from the `~/catkin_ws/src` directory created in the beginner tutorials. From there, you should clone the GitHub repository into a *different folder*, and then place the contents of that folder in the `~/catkin_ws/src/led_test` folder, overwriting any duplicates. From the command line, the easiest way to do this is remove the files individually, and then copy them over. Finally, change the directory back to `~/catkin_ws` and run the `catkin_make` command to build the package.

Subprocess Note:

The example below is a line of code within a listener that uses `subprocess.call()` to run a different Python script with root privileges, thus allowing for GPIO usage. (The `data.data` at the end sends an argument to the `led_blinker.py` program, and is talked about more under the **LED Blinker** heading). `subprocess.call(['sudo','python3','led_blinker.py','%s' % data.data])`

Another command in the subprocess library is `subprocess.Popen()`. Popen differs slightly from call, as it is a more general use function that does not wait for the process to finish to continue the original script. Call is useful in cases where the output of the called subprocess is required later on in the original script, while Popen allows the called script to run in the background without the original waiting for it to complete, which is useful in situations where a delay in listening could lead to missed messages. An example of Popen usage is available in the `led_receiver.py` script, where the `blink_test.py` script is called before calling the listener function. The led blinks 3 times while the listener initializes. If `subprocess.call()` were used, the listener would only initialize after the LEDs finished blinking. Aside from increased complexity, another drawback is that the ROS scripts that call these other scripts **MUST BE RUN FROM THE /scripts DIRECTORY** in the package to be able to locate the scripts that interact with the GPIO interface (i.e. cannot "rosrun" from just anywhere).

Pushing Changes to the Git from the Raspberry Pi:
This section is using our led_test repository as an example and is based on the tutorial found here: https://uoftcoders.github.io/studyGroup/lessons/git/branches/lesson/

1. Fork the repository on github: https://github.com/dmichael1227/led_test
2. Clone it to a directory you're comfortable working in:
   `git clone https://github.com/dmichael1227/led_test.git`
3. Create a new branch (make sure you're in the directory you just cloned):
   `git checkout -b branchName`
4. Make your edits, and add your changes:
   `git add yourfilename` or `git add -A` to add all files.
5. Commit your changes
   `git commit -m "Your Message"`
6. Push the new branch to your forked version:
   `git push origin branchName`
7. Submit a pull request on GitHub. Assuming that it is accepted, merge with the main and delete the branch on GitHub and locally on the Pi:
   `git checkout master && git pull upstream master && git branch -d branchName`

NOTE: If you have multi-factor authentication enabled on your account, simply inputting your username and password will not work. GitHub has a tutorial at https://docs.github.com/en/github/authenticating-to-github/creating-a-personal-access-token about creating a personal access token. Follow the tutorial to create one, and then use the token as your password when prompted in the command line.

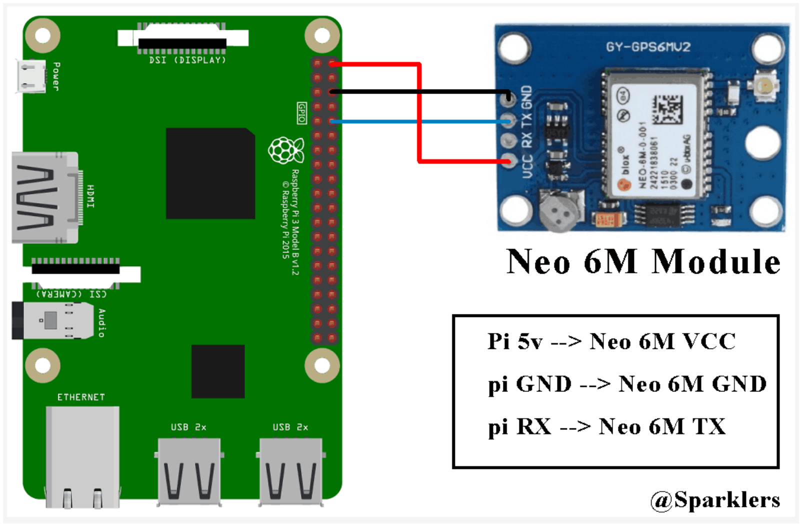
Running ROS on Multiple Machines:
The documentation for ROS provides a tutorial on running it across multiple machines at http://wiki.ros.org/ROS/Tutorials/MultipleMachines. When executed on two Raspberry Pis, both running Ubuntu 20.04, this tutorial worked without any major issues. One important step that needs to be taken in order for the two machines to work is letting ROS know what your IP address is. To check if ROS knows which IP to subscribe and receive on, run the `echo $ROS_IP` command. If this command prints out a blank block of text, you need to run `export ROS_IP=machine_ip_addr`, replacing `machine_ip_addr` with the IP address of your machine. Other than that, the tutorial provided in the ROS Documentation worked, as proven through our test of running the `led_controller.py` and `led_reciever.py` scripts on different Raspberry Pis, successfully blinking the LED.

To make following the tutorial easier, change the hostname of the Pi to allow you to use a name, just like `hal` and `marvin` in the tutorials, rather than typing in the typing in the IP addresses each time. To do this, run `sudo nano /etc/hostname` and replace the old name, likely `ubuntu`, with your desired hostname. From there, run `sudo nano /etc/hosts` and update any reference to the old hostname, if any, to reflect the new one. Then, run `sudo reboot`. The final step is to let ROS know the hostname, just like giving it the IP address in the paragraph above. Run `export ROS_HOSTNAME=your_hostname`, replacing `your_hostname` with the hostname you just assigned your machine.

NEO 6M GPS Module:

GPIO Pin Setup: Please note that this setup requires the Raspberry Pi to be running Raspbian, and was successfully tested on Raspbian Buster. Here are some instructions on setting up the NEO 6M GPS Module to work with the Raspberry Pi, as well as some example programs where the GPS logs data to a `.csv` file and then maps it.

1. Wire the module to the Pi as shown in the image below, which was taken from the Sparklers the Makers GitHub page referenced at the end of this tutorial



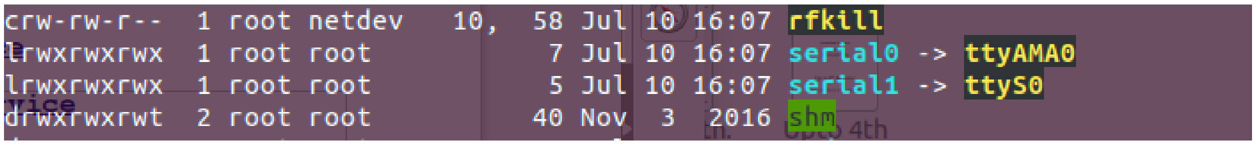
2. Update the Pi.
   a. `sudo apt-get update`
   b. `sudo apt-get upgrade`
3. Edit the `/boot/config.txt` file to enable UART, the Serial Interface, and disable Bluetooth (this makes the GPIO pin UART the primary UART.
   a. `sudo nano /boot/config.txt`
   b. Add the following lines:

```
dtparam=spi=on
dtoverlay=pi3-disable-bt
core_freq=250
enable_uart=1
force_turbo=1
```

   c. Now save and exit by pressing `ctrl+x`, type `y`, and then press `enter`.
4. Next, make a copy of the `/boot/cmdline.txt` file.
   a. `sudo cp /boot/cmdline.txt /boot/cmdline_backup.txt`
   b. `sudo nano /boot/cmdline.txt`
5. Now edit it to disable the UART serial console.
   a.

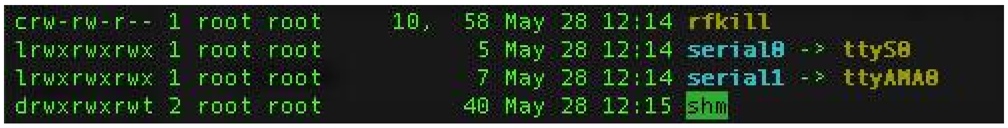
```
dwc_otg.lpm_enable=0 console=tty1 root=/dev/mmcblk0p2
rootfstype=ext4 elevator=deadline fsck.repair=yes rootwait quiet
splash plymouth.ignore-serial-consoles
```

   b. Now save and exit by pressing `ctrl+x`, type `y`, and then press `enter`.
6. Now reboot the Pi
   a. `sudo reboot`
7. After rebooting the Pi, make sure you are near a window, outside, or are in a location where the module can get a GPS signal.
   a. There will be a blinking LED on the NEO Module when it receives a signal from GPS satellites. Depending on the specific chip, the light could flash green, blue, purple, or even red. The exact colour is not important, just as long as one flashes.
   b. If the light does not blink within five minutes, move closer to a window or to a place under the open sky. Keep in mind that the module could take as long as 15 minutes to acquire its first satellite fix.
8. Once the LED is blinking, run the following command:
   a. `sudo cat /dev/ttyAMA0`
   b. This should print out streams of data, and can be stopped with `ctrl+c`. If there are error messages, this is likely due to a weak signal or a lack of signal. Essentially, this step verifies that the module is working.
9. Next, we need to make sure the Pi knows where the module is wired to.
   a. Run the `ls -l /dev` command.
   b. If your output looks like the image below (from the Sparklers the Makers GitHub page), run the following commands:



```
crw-rw-r--  1 root netdev   10,  58 Jul 10 16:07 rfkill
lrwxrwxrwx  1 root root           7 Jul 10 16:07 serial0 -> ttyAMA0
lrwxrwxrwx  1 root root           5 Jul 10 16:07 serial1 -> ttyS0
drwxrwxrwt  2 root root          40 Nov  3  2016 shm
```

   c. `sudo systemctl stop serial-getty@ttyAMA0.service`
   d. `sudo systemctl disable serial-getty@ttyAMA0.service`
   e. However, if your output looks like the image below (from the Sparklers the Makers GitHub page), run the following commands:



```
crw-rw-r-- 1 root root     10,  58 May 28 12:14 rfkill
lrwxrwxrwx 1 root root           5 May 28 12:14 serial0 -> ttyS0
lrwxrwxrwx 1 root root           7 May 28 12:14 serial1 -> ttyAMA0
drwxrwxrwt 2 root root          40 May 28 12:15 shm
```

   f. `sudo systemctl stop serial-getty@ttyS0.service`
   g. `sudo systemctl disable serial-getty@ttyS0.service`
10. Next, install some of the libraries that you'll need to run the scripts we're about to use:
    a. `sudo apt-get install python-pandas python3-pandas`
    b. `sudo pip3 install folium`
    c. `sudo pip install pynmea2`
11. From there, download the code from GitHub.
    a. `cd`
    b. `mkdir git`
    c. `cd ~/git`
    d. `git clone http://github.com/dmichael1227/gps_test.git`

12. To make sure the module works properly, move the code to its own directory.
    a. `cd ~/git/gps_test`
    b. `mv ~/git/gps_test/pi_tests ~/`
13. Before we use the GPS module at all, we need to update the GPS module so it knows what the date and time is
    a. `cd ~/pi_tests`
    b. `sudo python gps_update.py`
    c. This program will continuously spit out values, but you can end it as soon as it starts spitting out the proper time. This can be done by pressing `ctrl+c`.
14. Now, run the first program to make sure we are getting a good GPS reading.
    a. `sudo python initial_gps_test.py`
    b. Running this program should continuously print you `Latitude= xx.xxx and Longitude= xx.xxx`.
15. Next, let's log the GPS data in a `.csv` file. This program should both print it to `raw_gps_data.csv` as well as print the `Latitude= xx.xxx and Longitude= xx.xxx` continuously on the screen.
    a. `sudo python initial_data_logger.py`
16. You can verify this by opening the `raw_gps_data.csv` file with a spreadsheet editor, and you should see three columns of data that contain the latitude, longitude, and timestamp in that order. This data may be useful for you, but there need to be headers to map out the information. You can either add these yourself, +or you can let the program do it for you.
    a. `sudo python gps_data_logger.py`
17. Now, when you open up the `gps_mapping_data.csv` file, you will find that the data columns are named `Latitude`, `Longitude`, and `Timestamp`. This means that the data can now be mapped out
    a. One way to do this is to use a website that takes an uploaded `.csv` file and plots it out. An example of such a site is:
        i. https://www.gpsvisualizer.com/map_input?form=data
    b. Another way is using the folium library. An example of this is built into the `gps_mapper.py` script.
        i. First, edit the gps_mapper.py script to reflect your location
            1. `nano gps_mapper.py`
            2. Replace the `[41.519917,-71.29445]` part of the `mapPATH` definition with the coordinates you want your map to center on.
            3. You can also edit the `zoom` of the map as well.
        ii. `sudo python3 gps_mapper.py`
18. Open the RobotPath.html file in your preferred web browser and view your map!
    The information for this module tutorial was compiled from the following sources:

NOTE: There is also a library called `gps`, with commands such as `cgps` and `mgpsmon`. This particular library does not function properly when paired with this GPS module on Raspbian. It works every once in a while, but then spontaneously and continuously malfunctions, as such we are using the `pynmea2` library. The downside is the `gps` library can provide readings for longitude, latitude, speed, heading, altitude, time, as well as the error for all of those readings, whereas `pynmea2` only does latitude, longitude, and time. However, this seems to be a fair trade-off since we will not need all of those readings.

https://sparklers-the-makers.github.io/blog/robotics/use-neo-6m-module-with-raspberry-pi/
https://stackoverflow.com/questions/42879408/writing-variables-to-a-csv-row-python
https://ozzmaker.com/how-to-save-gps-data-to-a-file-using-python/
http://comet.lehman.cuny.edu/owen/teaching/datasci/foliumLab.html
https://projects.raspberrypi.org/en/projects/mapping-the-weather/
https://www.raspberrypi.org/forums/viewtopic.php?t=168440
https://github.com/FranzTscharf/Python-NEO-6M-GPS-Raspberry-Pi/blob/master/Neo6mGPS.py

Launcher File Setup: We can set the Pi up so that the `gps_data_logger.py` will run at boot, regardless of whether or not you are connected to your Pi. The file that does this, `launcher.sh`, is already written and included in the GitHub Repository that was cloned earlier on in this section. That being said, the Pi still needs to know to start the program at boot.

1. Edit the `launcher.sh` file and uncomment the execute command.
    a. `cd ~/pi_tests`
    b. `nano launcher.sh`
    c. Remove the `#` in front of the `#sudo python initial_data_logger.py`, so that it reads `sudo python initial_data_logger.py`.
    d. Now save and exit by pressing `ctrl+x`, type `y`, and then press `enter`.
2. Make the `launcher.sh` file executable
    a. `chmod 755 launcher.sh`
3. Test the file, it should run the Python script, you can exit out with `ctrl+c`.
    a. `sh launcher.sh`
4. Make a logs directory for error logs.
    a. `cd`
    b. `mkdir logs`
5. Have the `launcher.sh` file run at boot.
    a. `sudo crontab -e`
        i. This command opens the crontab and may prompt you to choose an editor, choose whichever one you are most comfortable with.
    b. Add this line at the bottom:
        i. `@reboot sh /home/pi/pi_tests/launcher.sh >/home/pi/logs/cronlog 2>&1`
    c. Now save and exit by pressing `ctrl+x`, type `y`, and then press `enter`.
6. Reboot.
    a. `sudo reboot`
7. If the code doesn't start running, you can check the error logs for troubleshooting.
    a. `cd ~/logs`

b. `cat cronlog`

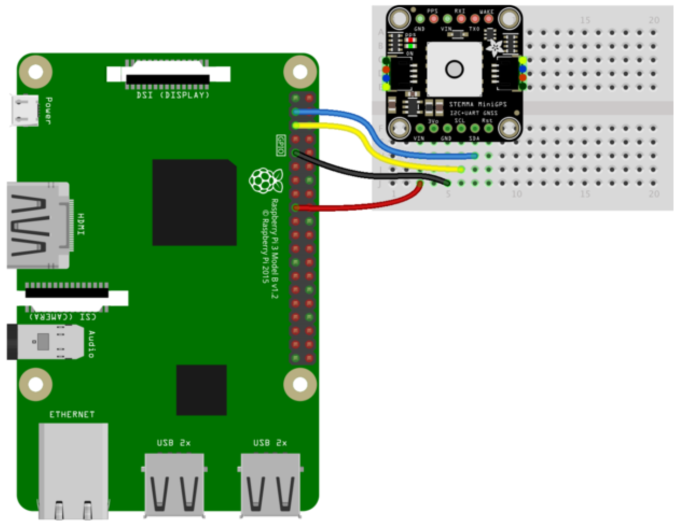
The information for this tutorial was compiled from the following source: https://www.instructables.com/id/Raspberry-Pi-Launch-Python-script-on-startup/

Adafruit Mini GPS PA1010D:

This GPS module allows for communication over UART, like the NEO 6M, and over I2C. This helps to solve the problem of the Pi not booting if the NEO 6M was attached to the UART GPIO pins and the Pi was running Ubuntu. This issue was likely due to a boot configuration issue that essentially led to the Ubuntu OS not knowing the GPS Module was supposed to be plugged in there, and thus assuming that it was a potential attack, hence the Pi not booting up. This could be solved with a USB to UART cable, as done in the **gps_comm Package**, but it can also be solved by using a different, I2C capable, module, as done with this Adafruit one.

There are only three sets of steps to get this module working on Ubuntu, and then you should be all set to integrate it into programs.

1. Wire the module to the Raspberry Pi's I2C line, following the diagram at the end of this section, which was taken from Adafruit's tutorial at
   a. learn.adafruit.com/adafruit-mini-gps-pa1010d-module/
2. Follow the steps under the I2C section of the **Configuring Pi for Interfacing** header
3. Follow Adafruit's tutorial (link below) to make sure your sensor is working properly and install the necessary library. Make sure to change the code so it is configured to use I2C and not UART.
   a. https://learn.adafruit.com/adafruit-mini-gps-pa1010d-module/circuitpython-python-i2c-usage
   b. `sudo pip3 install adafruit-circuitpython-gps`



Adafruit's Mini GPS Module I2C Wiring Guide

You can also download and run the `simple_test_gps.py` from the master branch of our repository at https://github.com/dmichael1227/gps_test to have the Pi read data from the module, and then log it to a .csv file.

The gps_comm Package:

The gps_comm package contains two Python scripts: `gps_talker.py` and `gps_listener.py`. `gps_talker.py` uses the `pynmea2` and `pyserial` libraries to read the serial port the GPS module is attached to and convert the nmea “sentences” to a usable format. A serial object is created to read from the port (default `/dev/ttyUSB0`) and `pynmea2` is used to parse the serial data read from the device. The latitude and longitude are extracted from the resulting object and converted to float64 before being passed in the previously created GPS message type to the `gps_listener.py` script.

This code is based on the talker/listener scripts from the ROS beginner tutorials (referenced earlier) as well as sample code from the pynmea2 library’s Git page: https://github.com/Knio/pynmea2

RosPerch - Set Up:

This is how ROS was integrated with the UTAP 2020 daughterboard and a Raspberry Pi 3B running Ubuntu 20.04 with ROS Noetic. Follow the steps under **Setting Up Your Raspberry Pi** and **Configuring Pi for Interfacing** to get your Pi set up. Also, please note that only the commands that need to be entered into the terminal will be formatted `like this` for the entirety of this section.

1. Build the daughterboard as described in the UTAP_2020_RPi.pdf document that is available at https://github.com/jdicecco/UTAP. Additionally, build a standard SeaPerch.
2. We then took the GPIO pin definitions, the code to spin a motor, and the code to turn on an LED from UTAP_2020.py (from the same repository as above), and merged that with the led_test package we made earlier. This code gets used later on in Step 6.
    a. Plugged the code to make the motor go full throttle forwards and turn on both LEDs into the led_blinker.py code, so a true input would turn them on, and a false input would turn them off.
3. Set up GitHub on the Pi. (Be sure to alter the commands so they reflect *your* name, email, etc. instead of the examples.)
    a. This step is adapted from wiki.paparazziuav.org/wiki/Github_manual_for_Ubuntu.
    b. `cd ~/.ssh`
        i. If there are any warnings or errors about the directory not existing, ignore it for now and move to the next step.
    c. `ssh-keygen -t rsa -C "your_email@youremail.com"`
    d. Press enter twice
    e. Input your desired password as prompted
        i. Afterwards, ensure you are in the correct directory again
        ii. `cd ~/.ssh`
    f. `mkdir key_backup`
    g. `cp id_rsa* key_backup`
    h. `nano id_rsa.pub`
        i. Copy the contents of the file. You may have to get creative with how you copy it, especially if SSHing into the Pi. Transferring the file to your PC

with FileZilla, opening it with Notepad++, and copying it worked when we tried it.

i. Add it into your GitHub Account Settings
    i. Sign in to the GitHub website, go to your Account Settings
    ii. Click “SSH and GPG keys”
    iii. Click “New SSH key”
    iv. Paste the contents of the file you copied into the “Key” field and then click “Add SSH key”

j. `ssh-add`

k. `git config --global user.name "Your Name"`

l. `git config --global user.email you@example.com`

m. Basically what this does is replace you having to input your GitHub username and password every time you push a change to you just having to input a shorter password. Additionally, it opens up a new means of cloning the repository, and also eliminates the need for the multi-factor authentication workaround that we had to do in the **Pushing Changes to the Git from the Raspberry Pi** section. This also allows you to run the git push/pull commands without any arguments when you need to get the most updated version of the repository.

4. Clone the GitHub Repository and link it to your catkin workspace.
    a. `cd ~/git`
    b. `git clone git@github.com:dmichael1227/ROSPerch.git`
        i. Alternatively, if you do not set up the SSH Keys:
        ii. `git clone https://github.com/dmichael1227/ROSPerch.git`
    c. `cd ROSPerch/`
    d. `git init`
        i. This initializes the repository in your current directory.
    e. `ln -s ~/git/ROSPerch/rosperch ~/catkin_ws/src/`
        i. You can check that it is linked properly by running the following commands:
        ii. `cd ~/catkin_ws/src`
        iii. `ls`
        iv. You should see the “rosperch” directory listed, usually in a lighter blue.
5. Build the package!
    a. `cd ~/catkin_ws`
    b. `catkin_make`
6. Run the tester code to make sure everything works.
    a. Open three separate terminals.
    b. In terminal one, run `roscore`.
    c. In terminal two, run `rosrun rosperch receiver.py`.
    d. In terminal three, run `rosrun rosperch controller.py`.
    e. If you input a 1 into the terminal running controller.py, the two side motors on the SeaPerch should start running, and the two LEDs on the daughterboard should light up.

    f. If you input a 0 into the terminal running controller.py, the two side motors on the SeaPerch should stop running, and the two LEDs on the daughterboard should light up.
        i. If the motors were not running in the first place, nothing would happen.
7. For controlling the ROSPerch via the handheld controller, run the following command:
    a. `rosrun rosperch joy_controller.py`
8. For the ROSPerch to run a predetermined path, in this case a square, run the following commands in separate terminals:
    a. `roscore`
    b. `rosrun rosperch launcher_for_auto_square.py`
    c. `rosrun rosperch auto_square.py`
        i. Please note that this code is currently fine tuned for our ROSPerch, but can be easily updated to work with any ROSPerch. The code is also easily adaptable to any pattern that is desired, so long as the distances are input, thus making this a great framework for sensor integration later on.
        ii. *Additional information on these scripts is available in the* ***ROSPerch - auto_square.py and launcher_for_auto_square.py*** *section below.*
9. For the ROSPerch to be controlled from the command line, run the following commands, in this order, in separate terminals:
    a. `roscore`
    b. `rosrun rosperch mission_commands.py`
    c. `rosrun rosperch auto_driver.py`
        i. Please note that this code is currently fine tuned for our ROSPerch, but can be easily updated to work with any ROSPerch. The code is also acting as a framework code that is easily adaptable to any sensor integration later on.
        ii. *Additional information on these scripts is available in the* ***ROSPerch - auto_driver.py and mission_commands.py*** *section below*

ROSPerch - auto_square.py and launcher_for_auto_square.py

The `auto_sqaure.py` script and `launcher_for_auto_square.py` script are intended to be run together in ROS (after launching `roscore`). The `launcher_for_auto_square.py` script is fairly simple; it queries the user for a `Y` or `N` value in order to launch the `auto_test.py` script. A value of `Y` will begin the sequence of function calls in the `auto_sqaure.py` script, while a value of `N` shuts off the forward propulsion motors. The `auto_sqaure.py` script contains a function that takes in a set direction for each of the two forward propulsion motors (True = forwards, False = backwards) and a time duration for which the motors should be running. Currently, upon receiving the start signal from the `launcher_for_auto_square.py` script, the `auto_sqaure.py` script begins a deterministic sequence of function calls to attempt to have the ROSPerch follow a predetermined path with no feedback. In the case of these specific scripts, the sequence directs the ROSPerch to drive in a square pattern, however this code can be easily modified and adapted to many other patterns desired.

ROSPerch - auto_driver.py and mission_commands.py

The `auto_driver.py` and `mission_commands.py` scripts are extensions of the `auto_sqaure.py` script and `launcher_for_auto_square.py` script. They share similar functionality, but with several key differences. The `mission_commands.py` script allows the user to send commands by typing `drive`, `leftturn`, `rightturn`, or `stop`. The code is designed to filter out human factors when it comes to the commands, which means that capitalization and spaces between the direction and the turn for the `left` and `right turn` commands are taken into account. Additionally, the script is also set up to filter out invalid commands. After entering a mission command, the user is then prompted to enter a parameter: distance (in meters) if the user selected `drive` and degrees if the user selected `left turn` or `right turn`. Should the user select stop, a dummy parameter variable of 1 is assigned `stop`, as due to the nature of the scripts, a parameter must be assigned *(it is important to note that currently, the stop command does not work and thus the motors will not stop the ROSPerch)*. The entered command and parameter is sent to the `auto_driver.py` script in a `Commands` message. After sending the command, the `mission_commands.py` script waits for a message on the `systemstate` topic to prompt the user to input another command.

Upon receiving a command, the `auto_driver.py` script parses the command and parameter values and executes them by driving the motors in the desired direction for the desired duration. This approach is extremely naive: it assumes that each motor outputs a constant velocity and that there are no changes in the external forces acting on the ROSPerch (no waves, currents, different density liquids, etc.). Stringing these commands together to form a mission also assumes that any resulting drift from the last command is negligible. The ROSPerch has no understanding of its relative position barring the information from previous commands. Upon completion of the last received command, the `auto_driver.py` script sends a message on the `systemstate` topic to inform the `mission_commands.py` script that the ROSPerch is ready to receive the next command. This structure is the foundation of an automated ROS system. The `mission_commands.py` script serves as a placeholder for a central decision-making node that, in the future, could integrate sensor data in real-time decision making to determine what the next command will be. Forcing the script to wait for a ready signal ensures that commands will only be sent when the system is ready to receive them. The only downside of this structure is that in order for the scripts to run properly, `mission_commands.py` must be run *first* and then `auto_driver.py`.

The `auto_driver2.py` and `mission_commands2.py` scripts (found in the *wip* section of the scripts directory) attempt to remedy this by introducing a set of handshake functions that establish that both nodes are ready to communicate. The `mission_commands2.py` script includes a handshake publisher thread that starts a publisher that runs continuously after start-up. This publisher publishes a True boolean value once per second on the `command_to_driver` topic. Upon receiving this for the first time, the `auto_driver2.py` script publishes True on the `systemstate` topic at 10 Hz for one second. The scripts then proceed normally after this sequence; the `mission_commands2.py` script prompts the user for command input to be sent to the `auto_driver2.py` script. These scripts do not work consistently; it appears that sometimes after receiving a message on the `command_to_driver` topic, the `auto_driver2.py` script can fail to publish the ready signal (tested by monitoring the `systemstate` topic), causing the `mission_commands2.py` script to never query the user for input. After attempting to publish a value to the topic using `rostopic pub -1 /systemstate /std_msgs/Bool True` It seems that this issue doesn't usually arise the first time the nodes are launched- getting progressively less likely to work during successive attempts until reboot. It also seems that restarting `roscore` may help too. We hypothesize that these issues may arise due to improper shutdown of the nodes or involved topics- further investigation is required to be sure.

To launch these scripts, run the following commands in separate terminals, in this order:

1. `roscore`
2. `rosrun rosperch mission_commands.py`
3. `rosrun rosperch auto_driver.py`

ROSPerch - 9dof_pub.py:

The `9dof_pub.py` script reads from the FXOS8700 3-axis accelerometer/magnetometer and the FXAS21002c gyroscope, converts readings into heading, roll, yaw, and yaw tilt, and publishes heading, roll, yaw, yaw tilt, and acceleration in x, y, and z to a single topic. This code is based largely on the UTAP base code: https://github.com/jdicecco/UTAP as well as the ROS tutorials: http://wiki.ros.org/ROS/Tutorials.

ROSPerch - SSH into Pi Over Direct Ethernet Connection:

In order to establish an ssh connection with the Pi in situations where you may be without a wireless network, a connection between the ethernet port of a laptop and the Pi can be used. (Please note that this assumes that you have SSH enabled on your Pi.)

1. Plug one end of an ethernet cord into the Pi, and the other into the laptop.
2. Press the `Windows key` on the laptop, type `cmd` and press `enter`.
3. Enter `ssh username@machinename.local` into the Command Prompt and press `enter`.
    a. Be sure to change the *username* and *machine name* to reflect the username and hostname of the Pi.
4. Enter your password, press `enter`, and you're in!

ROSPerch (WIP)- Absolute Orientation Kalman Filter (Kalman_Filter.py)

The `Kalman_Filter.py` script is an attempted adaptation of the master's thesis entitled "An Attitude Heading Reference System Using a Low-Cost Inertial Measurement Unit" by Matthew Leccadito. This Kalman filter is intended to estimate both quaternion attitude heading and gyroscope bias, fusing information from the gyroscope, magnetometer, and accelerometer. The state vector X_k consists of the four quaternion values and the three gyroscope bias values:

$$X\_k = [q_s,q_x,q_y,q_z,bias_x,bias_y,bias_z]^T$$

To initialize X_k, we use $[1,0,0,0]^T$ for the initial quaternion values (no rotation) and the average bias value recorded from ~1 minute of gyro data taken while the IMU was stationary.

To initialize the error covariance matrix Q_k, we use the outputs of the `Var_Measurement.py` script. This script runs for a maximum of 10 minutes and logs the gyroscope and accelerometer values collected every tenth of a second. It returns the variance of these values once interrupted by the user. The values currently used for Q_k come from roughly three minutes of recording. The initial values of Q_k currently used are:

```
Q_k = numpy.array([[0.0001,0,0,0,0,0,0], #var(qs)
                   [0,0.0001,0,0,0,0,0], #var(qx)
                   [0,0,0.0001,0,0,0,0], #var(qy)
                   [0,0,0,0.0001,0,0,0], #var(qz)
                   [0,0,0,0,0.038,0,0], #var(gyro bias x)
                   [0,0,0,0,0,0.024,0], #var(gyro bias y)
                   [0,0,0,0,0,0,0.015]]) #var(gyro bias z)
```

An issue left unaddressed by the source document is the relationship between the recorded variances (variance of accelerometer data) and the variance of the different quaternion components. The highest accelerometer variance of the three axes (0.0001) was used, but a more appropriate measure is being investigated.

The noise measurement covariance matrix R_k contains the variance of the three accelerometer data channels from a previous dataset. The `Noise_Measurement.py` script records the accelerometer data and provides the variance of the data corresponding to each axis of the accelerometer. The initial values used for R_k are:

```
R_k = numpy.array([[0.367,0,0], #Accel cov x
                   [0,0.719,0], #Accel cov y
                   [0,0,0.269]]) #Accel cov z
```

As explained in the original document, hardware limitations were present in the original application of the algorithm at its operating frequency (50 Hz) because of its intent to be used on an unmanned aerial vehicle (the currently implemented operating frequency is 10 Hz). Because of this, floating point values in the original implementation were set to single precision. The smallest value that could be represented, 10^{-8}, was used for each entry on the diagonal of the noise model covariance matrix. Double precision was used in our implementation, so this value has been changed to 10^{-16} in our noise model covariance matrix, P_k.

The general workflow of the filter is as follows:

1. Measurements are taken from the IMU and corrected
    a. Hard iron offset is subtracted from magnetometer values
    b. Gyroscope bias is subtracted from gyroscope value
2. Store measurements in the Jacobian matrix, delta_f_delta_X.

```
delta_f_delta_X = numpy.array([[0,-omega_x,-omega_y,-omega_z,qx,qy,qz],
                               [omega_x,0,omega_z,-omega_y,-qs,qz,-qy],
                               [omega_y,-omega_z,0,omega_x,-qz,-qs,qx],
                               [omega_z,omega_y,-omega_x,0,qy,-qx,-qs],
                               [0,0,0,0,0,0,0],
                               [0,0,0,0,0,0,0],
                               [0,0,0,0,0,0,0]])
```

Where omega_x, omega_y, and omega_z are the corrected gyro values and qs, qx, qy, and qz are the quaternion values.

3. Determine the transition matrix, T_k.

   $$T\_k = I_{7x7} + ½*delta\_t*delta\_f\_delta\_X$$

   Where delta_t is the time between samples.
4. Determine predicted state vector X_predict.

   $$X\_predict = T\_k*X\_k$$
5. Determine the error covariance matrix Q_predict

   $$Q\_predict = T\_k*Q\_k*T\_k^T + P\_k$$
6. Determine the measurement transition matrix A_k (denoted A_k here).

   $$\phi_{nonlinear} = \arctan(2(q_y q_z + q_s q_x)/(1 - 2(q_x^2 + q_y^2)))$$

   $$\theta_{nonlinear} = -\arcsin(2(q_x q_z - q_s q_y))$$

   $$\psi_{nonlinear} = \arctan(2(q_x q_y + q_s q_z)/(1 - 2(q_y^2 + q_z^2)))$$

   $$A_k = \begin{bmatrix} \frac{\partial\phi}{\partial q_s} & \frac{\partial\phi}{\partial q_x} & \frac{\partial\phi}{\partial q_y} & \frac{\partial\phi}{\partial q_z} \\ \frac{\partial\theta}{\partial q_s} & \frac{\partial\theta}{\partial q_x} & \frac{\partial\theta}{\partial q_y} & \frac{\partial\theta}{\partial q_z} \\ \frac{\partial\psi}{\partial q_s} & \frac{\partial\psi}{\partial q_x} & \frac{\partial\psi}{\partial q_y} & \frac{\partial\psi}{\partial q_z} \end{bmatrix}$$

   The equations for each partial derivative were completed using a partial derivative calculator.
7. Compute Kalman gain K_k

   $$K\_k = Q\_predict*A\_k^T*(A\_k*Q\_predict*A\_K^T+R\_k)^{-1}$$
8. Determine measured Euler angle based on accelerometer/magnetometer readings:

```
phi_acc = math.atan2(-accel_y,accel_z)
theta_acc = math.asin(-accel_x/math.sqrt(accel_x*accel_x+accel_y*accel_y+accel_z*accel_z))
psi_acc = math.atan2(mag_y,mag_x)
```

```
z_k = numpy.transpose(numpy.array([phi_acc,theta_acc,psi_acc]))
```

9. Determine the nonlinear measured euler angle values and store them in h_k

```
phi_nl = math.atan2(2*(qy*qz+qs*qx),(1-2*(qx*qx+qy*qy)))
theta_nl = -math.asin(2*(qx*qz-qs*qy))
psi_nl = math.atan2(2*(qx*qy+qs*qz),(1-2*(qy*qy+qz*qz)))
```

```
h_k = numpy.transpose(numpy.array([phi_nl,theta_nl,psi_nl]))
```

10. Determine the innovation matrix y_k

```
y_k = numpy.subtract(z_k,h_k)
```

11. Update state matrix X_k

```
X_k = numpy.add(X_predict,numpy.dot(K_k,y_k))
```

12. Normalize quaternion values in updated X_k. This step is omitted from the original document, but explained here, at the top of page 9190: https://www.mdpi.com/1424-8220/11/10/9182/pdf

```
norm = math.sqrt(X_k[0]*X_k[0]+X_k[1]*X_k[1]+X_k[2]*X_k[2]+X_k[3]*X_k[3])
X_k[0] = X_k[0]/norm
X_k[1] = X_k[1]/norm
X_k[2] = X_k[2]/norm
X_k[3] = X_k[3]/norm
```

13. Update the error covariance matrix Q_k

```
Q_k = numpy.subtract(Q_predict,numpy.matmul(numpy.matmul(K_k,A_k),Q_predict))
```

This implementation is a work in progress. As it currently stands, the output values do not seem to converge.

Additional References:

A majority of our references are included within the steps that they were used to create, but here are websites and documents that were used as additional references about ROS, Ubuntu, GitHub, and other project related topics.

General ROS References

https://gist.github.com/drmaj/20b365ddd3c4d69e37c79b01ca17587a

Covers building ROS Melodic with Python 3 rather than Python 2.7, was used during testing different ROS distros.

https://rsl.ethz.ch/education-students/lectures/ros.html#course_material

Course materials for a C++ centric ROS course from ETH Zurich, first lecture and exercise are very useful for general ROS intro, other lectures focus more on ROS with C++.

https://surfertas.github.io/ros/2017/03/06/ros-husky-robocup.html

Goes over building a .launch file for Exercise Session 1 of the above course website and provides an example solution.

https://answers.ros.org/question/253445/arduino-cmd_vel-and-odom-with-pololu-motors-w-encoder/

Example of using ROS with the motor drivers that are used on the UTAP daughterboard, only this example is run on an Arduino with C++ rather than a Raspberry Pi with Python, but still helpful for example integration.

http://wiki.ros.org/ROS/NetworkSetup

Goes over different parts of configuring ROS to connect to a network, thus allowing you to communicate with ROS across multiple machines.

http://wiki.ros.org/ROS/Tutorials/MultipleRemoteMachines

Information about connecting to machines/robots that are not on the same network, was not very practical to test in the current situation.

http://wiki.ros.org/Packages

Outlines what the common files and directories are in a ROS package and defines what they are.

https://answers.ros.org/question/217107/does-a-roslaunch-start-roscore-when-needed/

Clarifies that using roslaunch will autostart roscore if it is needed but not already running.

__ROS - Launch File__

https://answers.ros.org/question/12216/launch-file-how-to-write-it/

This has an example of how a .launch file would be written.

http://www.clearpathrobotics.com/assets/guides/kinetic/ros/Launch%20Files.html

Provides a more detailed picture of what the .launch file should look like. The file starts with <launch> and ends with </launch>. Within those tags, there is a node tag that does essentially the same thing as launching the program with rosrun, including auto launching roscore if it is needed. The pkg in the tag is the ROS package name, the type is the name of the executable file, in our case Python script, that is being launched, and the name part of the tag should be the same as the type part, but can be used to overwrite the name of the node if necessary. Basically, for every Python script you need to call, you need a node tag, since the scripts are technically ROS nodes. Be sure to order the node tags in the order they need to be launched!

http://wiki.ros.org/roslaunch/XML

Official documentation on the .launch file's XML content.

__ROS - Robot Navigation__

https://circuitpython.readthedocs.io/projects/fxas21002c/en/latest/api.html
https://cdn-learn.adafruit.com/assets/assets/000/040/671/original/FXAS21002.pdf?1491475056

Documentation and datasheet for the NXP FXAS21002C gyroscope, part of the NXP Precision 9DoF Breakout.

https://circuitpython.readthedocs.io/projects/fxos8700/en/latest/api.html https://cdn-learn.adafruit.com/assets/assets/000/043/458/original/FXOS8700CQ.pdf?1499125614

Documentation and datasheet for the NXP FXOS8700 accelerometer and magnetometer, part of the NXP Precision 9DoF Breakout.

https://learn.adafruit.com/nxp-precision-9dof-breakout/

Documentation for the NXP Precision 9DoF Breakout.

https://learn.turtlebot.com/2015/02/01/14/

Information about using ROS to move a robot to a specific point on a map, seems to need to have some sort of vision sensor on it to work. This tutorial is also specific to the TurtleBot.

https://www.southampton.ac.uk/~fangohr/teaching/python/book/html/16-scipy.html

The SciPy library offers a range of numerical methods for Python. When paired with numpy, it can do math with vectors and matrices, and paired with matplotlib it enables you to plot and visualize the data. This is potentially useful for more “complex” math problems involving integration.

https://answers.ros.org/question/335062/how-to-create-a-simple-map-for-simulation-in-stdr-simulator/?answer=335079#post-id-335079
http://wiki.ros.org/map_server

The answer to this forum post (first link) shows how to create a map file that ROS can read. Essentially, you create a simple map in a drawing software like MS Paint or GIMP, save it as a .png, and then create a text file that provides the information ROS needs to know about (examples include image name and the resolution (in meters per pixel)), and save it in the same directory as the image file, with the same name as the image file, but with the .yaml extension. Additional information can be found in the map_server Package Summary (second link) under the Map format header.

https://www.cs.bham.ac.uk/internal/courses/int-robot/2017/notes/amcl.php

More information about using ROS to move a robot to a specific point on a map, still needs some sort of vision sensor or environment scanner. If you use the information in the link above to create a map, this tutorial shows how to visualize it using rviz (skip the step about running roscore and the Pioneer driver).

https://answers.ros.org/question/227390/how-to-bring-map-in-rviz-indigo-gazebo/

Another way to view a map file in rviz.

https://answers.ros.org/question/233257/rospack-error-package-map_server-not-found/

The map_server package might not be installed with the specific version of ROS that you download, so this is the command to download it. Just be sure to change “indigo” to whatever distribution of ROS you are using.

Python References

https://pythonprogramminglanguage.com/user-input-python/

Reviews commands to get raw inputs from users in Python 3 and gives example uses.

https://stackoverflow.com/questions/22990069/text-game-convert-input-text-to-lowercase-python-3-0

Covers how to change a user's raw input into all lower-case to eliminate code errors stemming from differences in capitalization. Simply change `.lower()` to `.upper()` for upper-case.

General Linux References

https://learning.oreilly.com/library/view/robot-operating-system/9781484234051/

This book provides a brief introduction to Linux operating systems, with most of the focus on Ubuntu, as well as some basic commands. It also covers the absolute basics of ROS with both C++ and Python.

Ubuntu References

https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-vnc-on-ubuntu-18-04

Installing VNC on a Ubuntu system so it can be used when you do not have access to a monitor, mouse, and keyboard for the Raspberry Pi.

https://linuxize.com/post/how-to-enable-ssh-on-ubuntu-18-04/

Enabling SSH on a Ubuntu system so it can be used when you do not have access to a monitor, mouse, and keyboard for the Raspberry Pi.

https://www.howtogeek.com/341944/how-to-clone-your-raspberry-pi-sd-card-for-foolproof-backup/

Making a copy of the image of Ubuntu that runs on the Raspberry Pi to allow for backups and cloning the SD card.

https://www.raspberrypi.org/forums/viewtopic.php?t=36856
https://unix.stackexchange.com/questions/118716/unable-to-write-to-a-gpio-pin-despite-file-permissions-on-sys-class-gpio-gpio18
https://stackoverflow.com/questions/30938991/access-gpio-sys-class-gpio-as-non-root

These three links cover the GPIO issue that we were having, and show other people having the same issues, and how they worked around the issue.

https://www.cyberciti.biz/faq/ubuntu-change-hostname-command/
Changing the hostname of a Ubuntu system, useful for the above configuration.

Raspbian References

https://www.ionos.com/digitalguide/server/configuration/provide-raspberry-pi-with-a-static-ip-address/
Setting a static IP address for Raspberry Pi running Raspbian.

https://www.instructables.com/id/ROS-Melodic-on-Raspberry-Pi-4-RPLIDAR/
Has an image file of Rasbian Buster (and Raspbian Buster Lite) with ROS Melodic installed. Using this image file helps speed up the installation of ROS on the Pi.

Virtual Machine References

https://www.liberiangeek.net/2013/09/copy-paste-virtualbox-host-guest-machines/
Setting up VirtualBox to allow for copy and paste between host and guest machines.

GitHub References

http://sethrobertson.github.io/GitBestPractices/
Some GitHub best practices.

https://uoftcoders.github.io/studyGroup/lessons/git/branches/lesson/
Using GitHub on Linux within a terminal.

https://docs.github.com/en/github/using-git/ignoring-files
Goes over what .gitignore files are and how to use them.

https://stackoverflow.com/questions/115983/how-can-i-add-an-empty-directory-to-a-git-repository/180917#180917
Setting up a .gitignore file to add an “empty” directory to a GitHub Repository.

https://stackoverflow.com/questions/18216991/create-a-tag-in-a-github-repository
Adding tags to GitHub repositories and specific commits.

https://www.markdownguide.org/basic-syntax/
https://www.markdownguide.org/extended-syntax/

Syntax for markdown files, used as formatting guides to make README.md on ROSPerch GitHub Repository.

https://docs.github.com/en/github/managing-your-work-on-github/linking-a-pull-request-to-an-issue

Has information on linking pull requests and issues, both manually and automatically with keywords.

Underwater Vehicle References

https://ieeexplore.ieee.org/document/6107001

IEEE paper on implementing ROS on the Yellowfin AUV.

https://ieeexplore.ieee.org/document/8729755

IEEE presentation report on using ROS on a REMUS 100 AUV, references the Python based libraries and packages that they built to interface with the AUV, however these packages and libraries do not seem to be publicly available.

https://www.naval-technology.com/projects/remus-100-automatic-underwater-vehicle/

Article covering the sensors/navigation tools used by the REMUS AUV.

Additional GPS Sensor References

https://askubuntu.com/questions/891662/why-does-cgps-s-give-me-no-results

Tip that helped setting up gpsd when using a USB adapter.