MODULE

User Manual for Establishing Wireless Communication a EMFF Test Bed

August 20, 2018

Anthony Thompson
University of Maryland, College Park
Department of Aerospace Engineering

Contents

Chapter 1

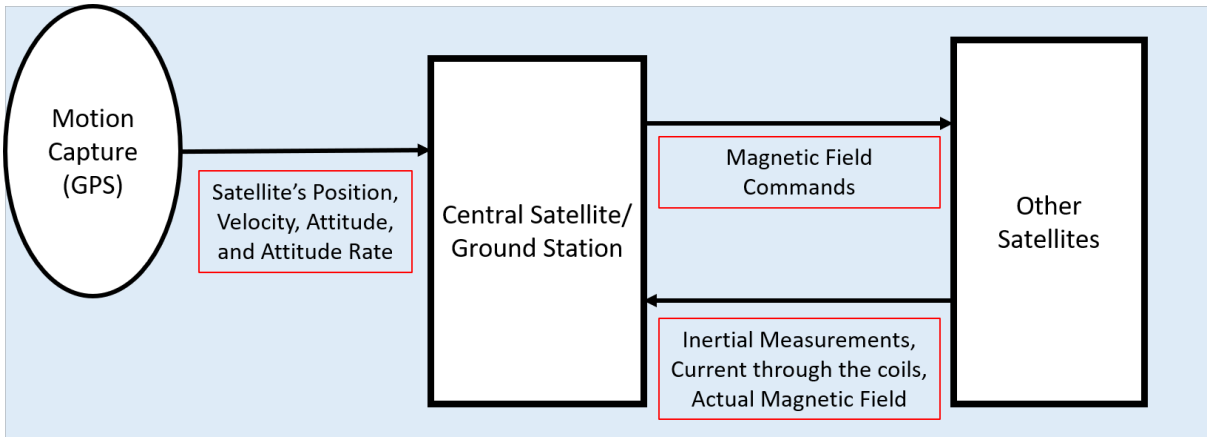
System Overview

1.1 Ground Station and Satellites

The communication network explained in this document was meant to be used for sharing data between multiple satellites. There is a main central satellite called a **Ground Station** that receives all of the data from all the other satellites. This data includes all 3 axis of the accelerometer, gyroscope, and magnetometer, as well as the current going through each coil. The ground station also gets data from the OptiTrack cameras; which is replacing the data we would get from GPS. The ground station also has the task of sending magnetic field commands to each of the satellites. The magnetic field commands is what tells each satellite where and how to move.

So in a nutshell, there are three main systems that need to communicate with each other. The majority of the communication network will be done by using a ROS network, and the satellite will be connected over wifi. The connection from OptiTrack to the ground station is done through MATLAB. There are a few files that you will need to establish this connection, found here. The link should take you to dropbox; where you will be able to find all of the information you'll need on connecting with OptiTrack. For this reason I will not be discussing the connection or communication with OptiTrack in this manual.

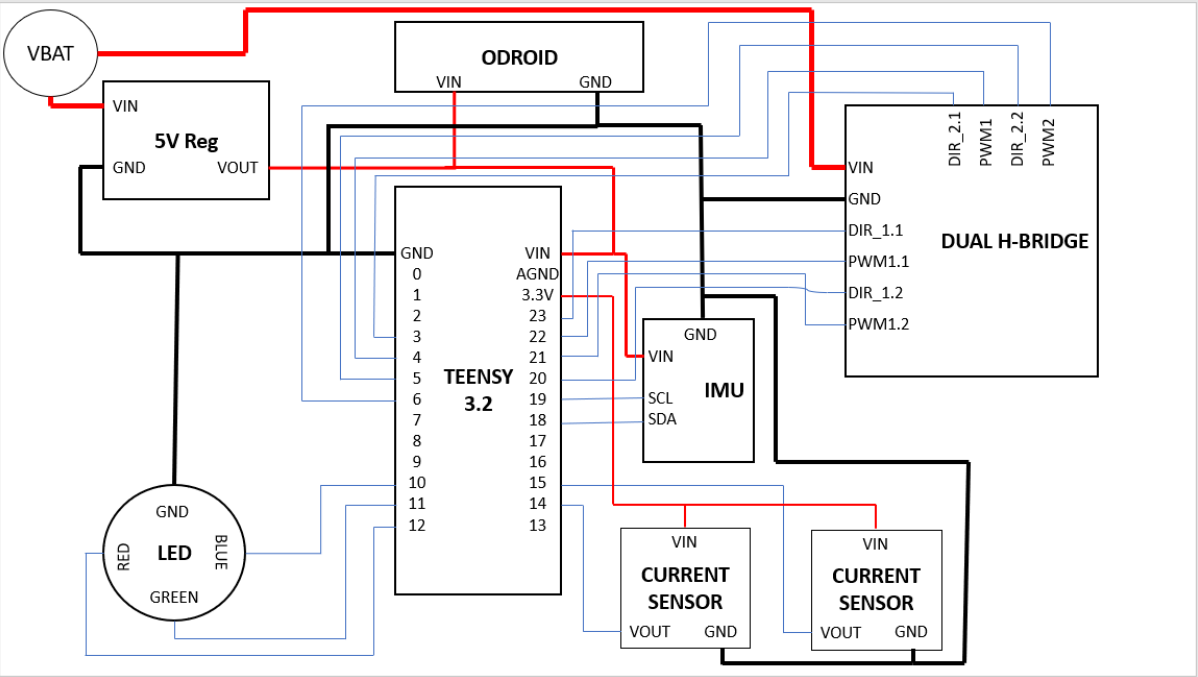
1.2 Block Diagram for Data Transfer



Chapter 2

Electronics

2.1 Circuit Diagrams



2.2 H-Bridge

The dual H-bridge is used to drive current through the magnetic coils. An H-bridge consists of a pair of N-type and P-type transistors. The source of the P-type transistors are connected to the battery, the gate is connected to the IC Driver, and the drain is connected to the coils.

The source of the N-Type transistors is connected to the other end of the current sensor, the gate is connected to the P-Type Transistor gate, and the drain is connected to ground.

A toggle switch and a 20 amp fuse is connected to the battery to add a layer a safety. After the switch is flipped, the voltage spikes up, drops, and then levels out to the battery voltage. This change in voltage activity will either fry the fuse or the transistors. To solve this problem a capacitor is added between battery and the gate of the P-Type transistor. A resistor needs to be added to the circuit to solve this problem.

2.3 BILL OF MATERIALS

This bill of materials does not contain all of the electronic components needed for the H-Bridge.

Item Name	**Description**		
	Part Number	**Store**	**Function**
Odroid XU4		Amazon	Linux Computer w/ ROS
Teensy 3.2	DEV-13736 RoHS	Sparkfun	Micro-Controller
Voltage Regulator D24V50F5	2851	Pololu	Step Down V_Battery
AltIMU-10 v5	2739	Pololu	IMU Measurements
ACS711EX Current Sensor	2453	Pololu	Measure Current in Coils
Status LED	754-1492-ND	DigiKey	Debugging Hardware
Logic Inverter IC	296-2055-5-ND	DigiKey	H-bridge Logic
IC NAND Driver	576-2332-ND	DigiKey	H-bridge Logic
MicroB Cable	CAB-14741 RoHS	Sparkfun	Connect Teensy to Odroid
EdiMax WiFi Adapter		Amazon	Connect to WiFi
SanDisk MicroSD Card		Amazon	Memory for the Odroid
N Type Transistors	IRFB7440PBF-ND	DigiKey	H-Bridge
P Type Transistors	IPP120P04P4L03AKSA1-ND	DigiKey	H-Bridge

Chapter 3

Coils

3.1 Temperature Considerations

These coils can reach 200°C before the insulation on the wire melts. We did not add any temperature sensors to the design yet, but they need to be added for multiple reasons. Temperature sensing needs to be added for safety precautions, accurate calculation of the coils' resistance, and for better temperature control.

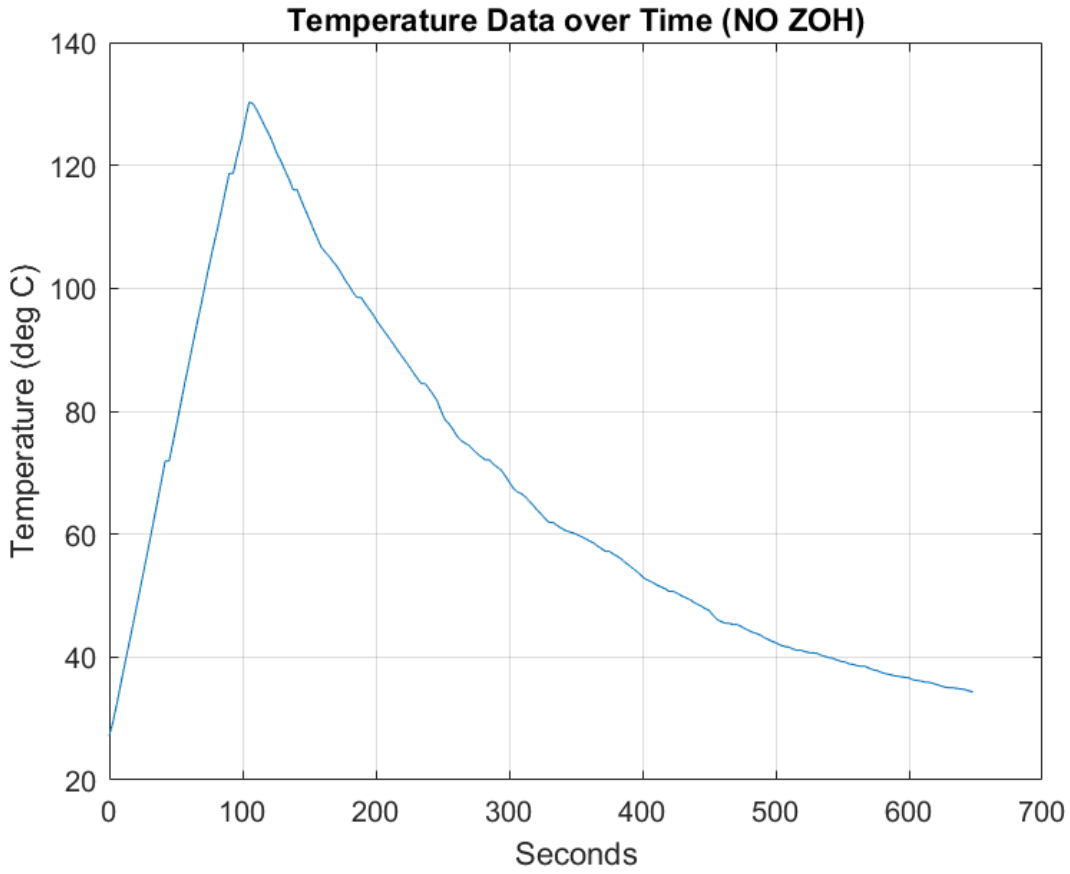
A safe range for touching temperatures of the coils is about 20°C to 65°C. At the moment we have to keep it in mind that the coils are hot and wait a few minutes after each experiment before touching the satellites. Once a temperature sensor is added, it is very easy to add it to the pre-existing circuit and use the status LED to let the user know when the coils are at safe handling temperatures.

The resistance of the coils increases as the temperature increases. In the code on the teensy I use the coils' resistance to calculate the duty cycle for the pwm signal into the H-Bridge. The current method assumes that the resistance is constant; which is good enough for temperatures near the environment's temperature. We can find the more accurate resistance but we need the temperature sensors to do so. We need a temperature sensor for each coil.

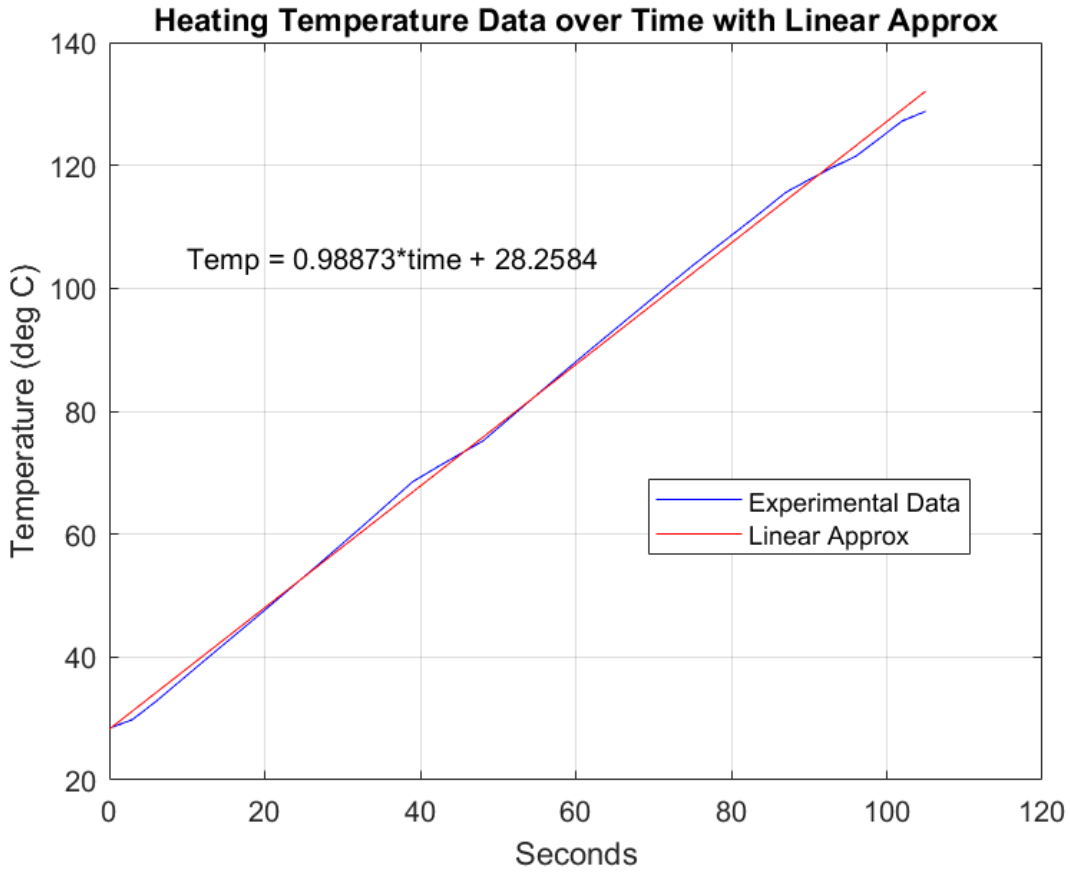
The global duty cycle that we apply to the coils helps extend the time to do experiments, but it will also help us manage the temperatures of the coils. Temperature data can be taken so that when the coils reach their max temp, the global duty cycle can be decreased. This allows the coils to have more time to cool, but they still have the ability to be used during experiments. This should increase the available experiment time even more.

3.2 HEAT DISSIPATION

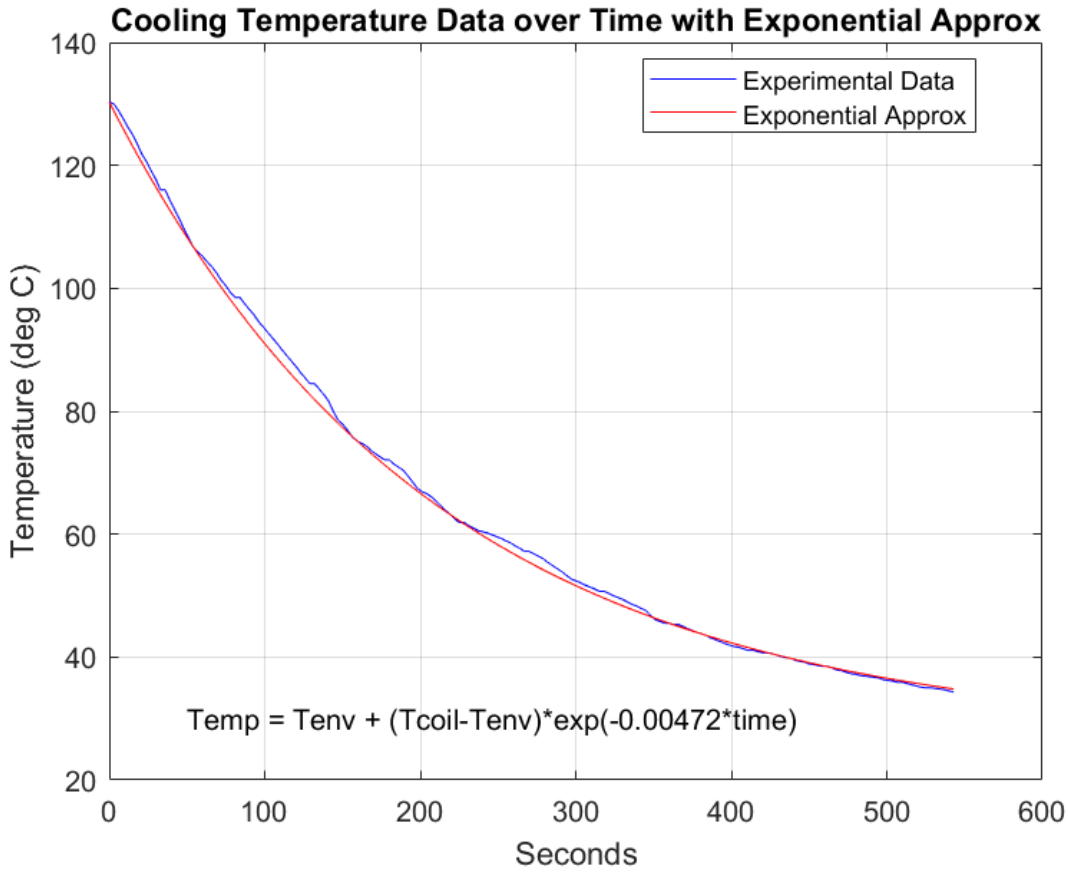
The figure below shows the temperature of the coil with respect to time. In the heating section of the figure the coil had a constant current of 20 amps, and in the cooling section the coil had 0 amps. We stopped heating the coil at about 130°C. As we would expect the coil cooled down exponentially; however, the heating section appears to be more linear than exponential.



The next figure will show you that the best fit line of the heating section. You will see that when the coils are pumped with 20 amps that the temperature increases by 1°C/sec. Until the temperature sensors get add it is good to keep this temperature increase rate in mind when working with the coils. Just keep it in mind that these coils heat up a lot faster than than cool. This is important when applying the global duty cycle. Give the coil more time to cool than time to heat heat (LOW duty cycles).



The figure below is the cooling section with the best fit line. As expected the cooling is exponential. It takes about 9 minutes for the coils to cool back down to room temperature and about 4 minutes to cool down to safe handling temperatures.



3.3 Safely Dealing with Hot Coils

At any given time the coils are being pumped with 280 watts of power. This is over 4 times the amount power input to an incandescent light bulb. Touching the electrical connection to the coils or the batteries while they are powering the coils could probably kill you. This section will explain all of the safety precautions needed to keep you alive and without third degree burns.

1. Do not touch the coils while they are being powered

    - The coils can get up to 200°C before the insulation melts off, so it is safe to say **you should not touch the coils at any point during the experiments and you should give the coils about 5 minutes to cool down to safe temperatures.**
    - As stated above the coils have a power input of 280 watts. Touching the electrical connections will transfer most of the power through you body instead of the coils.

Once the batteries are connected to the coils you should not touch the coils, the h-bridges, or the fuse holder

2. Water should never touch any of the electronics

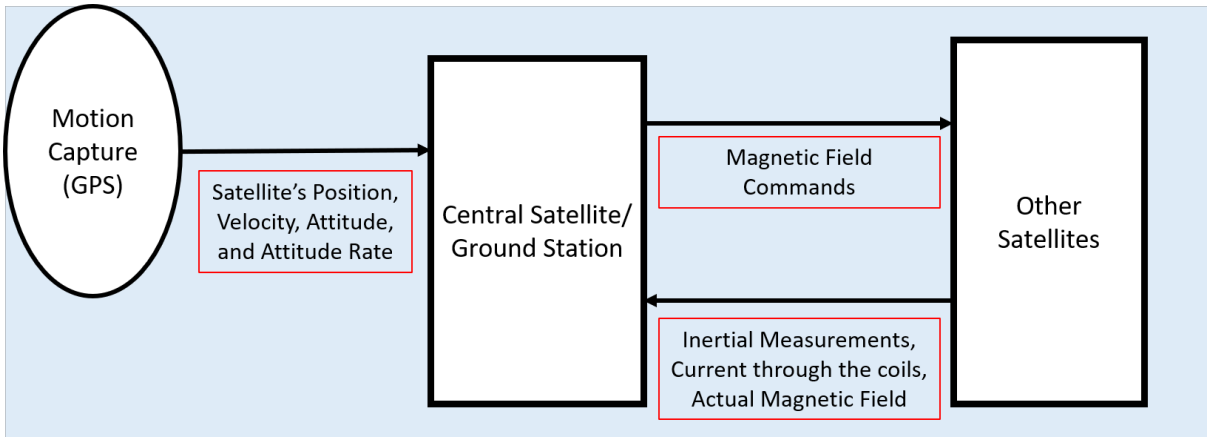
    - Obviously water and electronics do not mix, so when doing experiments **Do not place the batteries or any other electronics on the edge of the satellite. All electronics should be places in the center cutout of the satellite at all times during an experiment.**
    - Make sure the boat does not capsize or sink with any of the electronics on it.

Chapter 4

Communication Methods

4.1 How are the Satellites and Ground Station Communicating

The communication network is set so that there is a ground station computer that sends commands to several independent satellite computers, and each satellite sends their data back to the ground station. The communication between the ground station and each individual satellite is bi-directional and carried out by use Ros.



As seen in the image above the ground station receives inertial data from the motion capture system for each satellite and uses it as feedback for the orientation and position of each satellite. The ground station is connected to each satellite through a wifi connection and communicates by using ROS messages. Each satellite sends their linear acceleration vector, angular velocity vector, magnetometer vector, and the current through each coil to the ground

station; whereas, the ground station only sends a magnetic dipole moment vector in Cartesian coordinates to the each satellite that it wants to control.

4.1.1 Wifi Connection

All devices must be connected to the same wifi network in order for things to work. When updating the software on the odroids (satellites), Internet is a requirement for the wifi network,but during the experiments the wifi networks do not have to have Internet access in order to work, but each satellite must be able to automatically connect to the wifi network.

Once the odroids on each satellite is connected to the wifi, use MobaXterm, or puTTy to connect to it with a ssh connection, run roscore, and then run the roslaunch file used for the experiment. Connecting to the odroid through a ssh connection allows you to control it and run the programs needed for the experiment. If everything is set up properly, the odroids will not need a monitor, keyboard, or mouse in order to function. As soon as the ssh connection is setup you should type:

```
ping www.google.com
```

Running this command pings a signal to google and waits to receive on back. This helps prevent the wifi signal from dropping. If at any point the wifi signal is dropped, cycle the power of the odroid.

4.1.1.1 Wifi Auto-connect on Boot Up

The satellites should be able to connect to wifi on boot up. This is so you do not have to connect the odroid to a monitor, mouse and keyboard, manually connect to the wifi, establish a ssh connection from another computer, and reattach the odroid to the rest of the satellite's electronics. You should be able to just turn the odroid on, automatically connect to the wifi, establish the ssh connection and start the experiments without ever disassembling the electronics.

To do this you you must insert the wifi adapter into the odroid and enable auto login on the odroid. By skipping the login page you are allowing the odroid to boot up and connect to the wifi as soon as you turn it on. There are several ways to make the odroid automatically login, but the way I chose to do it follows below.

MATE

MATE users can achieve the same by adding the following text to the "/usr/share/lightdm/lightdm.conf.d/60-lightdm-gtk-greeter.conf" file. You need root privileges to do this.

```
[SeatDefaults]
greeter-session=lightdm-gtk-greeter
autologin-user=username
```

Remember to replace the value of `autologin-user` with your username.

A second option is to go to "System -> Administration -> Users and Groups" on the top panel. Click on the user you want to set up auto-login for and click "change" on the Password field.

For more infomation of this process, click on this link and follow the instruction under the Linux MATE.

4.1.2 IP Addresses and URIs

When setting up the ssh connection you will need to know the IP address for each odroid. This means that you need to know the IP addresses of each odroid before starting each experiments. The IP addresses can be found by connecting the odroid up to a monitor that accepts an HDMI connection, a mouse, and a keyboard, opening up a terminal window, and typing: **hostname -I**. On a higher level of communication through ROS you may also need the odroids URI. You can get this by typing: **hostname** in the odroids terminal window. You would use this in MATLAB when connecting to ROS.

4.2 ROS NETWORKS

ROS is used to send messages between the satellites and ground station. For more information on ROS and its capabilities visit its home page.

4.2.1 How to Install ROS

The odroids are installed with Ros Kinetic. The install page will walk you through the installation process. **ROS only works with linux computers**

4.2.2 Launch File Explanation

For the satellite reorientation and satellite repositioning experiments we use a ros launch file to create the satellite node. A more in depth explanation what a ros launch file is and how it works can be found here. To run the ros launch file, you must go to the directory where the file is located and type:

roslaunch fileName.launch **(where fileName is the name of the launch file)**

4.2.3 Launch File Template

Below is the launch file that is used for most experiments, but you can find other examples here. This launch file sets up all of the nodes that can be run on the odroid. For most experiments we only need to run the teensy node. We can also set some of the parameters of the node. As seen below we set the port name and baud rate to be fixed values on lines 5 and 6 respectfully. Since the port name is specified on line 5, the teensy USB cable must be connected to the same port each time. Go to the picture of the electronics setup to verify connection 2.1.

test_1.launch

```
1  <launch>
2
3      <!--Arduino -->
4      <node pkg="rosserial_python" type="serial_node.py" name="teensy_node" >
5        <param name="port" value="/dev/ttyACM0" />
6        <param name="baud_rate" value="115200" />
7      </node>
8
9  </launch>
10
```

If you ever need to add more nodes to run on the odroid side of experiments, you can always add to a launch files. Depending on the types of node you need to add will determine the format needed.

4.2.4 Rosserial Connection

The teensy is connected to the odroid by USB and uses a serial connection to communicate with it. To use rosserial for arduino and teensy you have to download some of the drivers and clone a github repository. A step by step description can be found here. For Step 2.1.1 of the tutorial you need to put:

sudo apt-get install ros-kinetic-rosserial-arduino

sudo apt-get install ros-kinetic-rosserial

Once you finish the tutorial you should be able to access the ROS commands and libraries from the arduino IDE.

4.2.5 ROS MATLAB Connection with Examples

ROS is usually meant for Linux machines, but it also comes with MATLAB 2017b and newer versions in the robotics toolkit. If your version of MATLAB does not have the robotics toolkit then you will need to download it from the MATLAB website.

Once you have access to the proper toolkits and libraries setting you the ROS network to the satellites is pretty simple, but some information from each computer is needed before the network can be created. The IP addresses from the master computer and the each respective satellite must be known. Once these are known you have to tell MATLAB who the master and slave nodes are by typing:

setenv('ROS_MASTER_URI' , 'http://masterIP_Address')

setenv('ROS_IP' , 'slaveIP_Address)'

An example of an IP Address is 192.1.168.31. The next step is to initialize the ROS network:

rosinit

A quick way to verify if your ROS network is setup correctly is to type the following command into the command line:

rosnode ping /Insert_Node_Name

You should see the the time it takes to ping the node specified. If the node is never pinged then you are not properly connected to the node. See debugging to solve this problem or go to the MATLAB_ROS tutorials.

4.2.6 Setting up the Communication Network

This section will demonstrate how set up the communication network. Some of this information may feel like it was previously stated above, but these instructions that follow are more sequential.

4.2.6.1 Setting up Communication for the Satellite

The satellites in this experiment will be using an odroid XU4 as a Linux machine with ROS Kinetic downloaded on it. The odroid has the latest version of an Ubuntu 16.04 Image saved on its microSD card. Once the image has been properly formated, a wifi connection to a trusted network needs to established, and ROS Kinetic needs to be downloaded and installed on the odroid to make communicating with the ground station easier.

Using the ROS environment makes it easier for multiple computers and satellites to communicate and share data. This environment sets each computer and satellite as a node and sets the data shared between them as a topic. Before starting any experiments, ROS needs to know two pieces of information. First ROS needs to know who is the master node. Then ROS needs to know what node identification of its satellite. The master node can be permanently set by going to the bottom of the */bashrc* file and typing:

export ROS_MASTER_URI= http:satellite URI

The next line of the */bashrc* file should identify the node ID of the satellite. To specify the node ID of an individual satellite type:

export ROS_NODE_ID= satellite IP address

Once the ROS environment variables are set, initiating the ROS environment is done by typing *roscore* into a Linux terminal. The next step is to set each satellite up as a node. This is done by connecting the teensy 3.2 to the odroid with an USB cable and creating a launch file. In the launch file, name of the node, the baud rate of the teensy, and the USB port that the teensy is connected to is specified. Save the launch file then open up a Linux terminal window. In the terminal type the following to create a satellite node:

roslaunch name_of_launch_file.launch

The communication network for these systems requires that each satellite node both receives and broadcast some data to the ground station computer. The shared data between a satellite and the ground station is known as a topic in ROS. On the satellite's side of things, the topics are created on the teensy. Code on the teensy specifies which topics the satellite receives data from and which topics the satellite sends data to. Since the teensy handles the topics, when the launch file is run, the code on the teensy is executed and the topics are established.

This concludes setting up Communication on the satellite. So far, the satellites have the ability to send data to the ground station but are not yet receiving data. To receive data, communication on the ground station has to be setup.

4.2.6.2 Setting up Communication for the Ground Station

The ground station in this experiment is a desktop computer that is connected to the OptiTrack cameras, and is tasked with doing all of the data analysis and implementing the control algorithms.

Motive is the software interfaced used to run the OptiTrack camera and gather the position, orientation, and their first derivative of each satellite. This data is transfered into MATLAB through a *NatNetML.dll* file. Once the data is in MATLAB, several scripts can be written for processing and analysis.

MATLAB is the main software interface used to setup communication with the satellites. Like the satellites, the ground station uses ROS for its convenience and ease to communicate to other computers, but the ground station facilitates ROS through MATLAB because it uses Windows and not Linux. The process of setting up ROS on the ground station is almost identical to the satellite.

The first step is to specify which computer is the master node. That node has to be the same node that was specified to each of the satellites, and the line that establishes master follows below.

setenv('ROS_MASTER_URI','http:*ground station URI*')

The second piece of code needs to specify the node ID of the ground station as shown below.

setenv('ROS_IP','*ground station IP address*')

The last piece of code needed to establish the ROS network is the initialization. To initialize the ROS network type *rosinit*.

After the ground station node has been created, the next step is the setup the ROS topics. Remember that the topics of a ROS network are the lines where the data is being shared, but each topic can have its own message type. Think of a message type as the structure the data is stored in. For example, a satellites linear acceleration data is in the form of a vector, so

three independent numbers are expected; whereas, the current through a satellite's coils is only a scalar, so only one number is expected. Setting up the topics in MATLAB requires the knowledge of knowing the message types of each topic, and this is all set up before hand in the code on the teensy. An example of setting up a topic that is sending data to a satellite in MATLAB is shown below.

[pub, mSG]=rospublisher('TOPIC','Message_Type')

If a node has data that it is receiving from the ground station, the process is almost identical and is shown below.

sub=rossubscriber('TOPIC','Message_Type')
receive(sub,timeLimit)

The *receive* command tells MATLAB to try to receive a message from the topic specified by *sub* and to wait for a certain amount of time specified by *timeLimit* before quitting. The *receive* command is another way of confirming that the communication network is bidirectional.

At this point the ground station should be able to connect to each satellite via ROS and receive data from each satellite at least once. The last two steps are sending data over a topic and continuously receiving data from a satellite.

Sending data to a satellite is easy once the proper topics have been established. The code needed to send messages follows below.

send(pub,msg_data)

As long as a topic has data on it, that topic will always have that piece of data on it until it gets updated. For this reason sending data to a topic does not need to be in a loop, so unless it is desirable to publish data that is being updated over time, sending it once will suffice for an entire experiment.

Unlike sending data, receiving data requires a loop. This is because the ground station needs to be continuously updated with the data sent from the satellite. Receiving data from a topic also requires that prior knowledge of the message type is known. The code needed to receive data from a topic follows below.

array = *TOPIC*.lastmessage.*message_typeInfo*

Where message type info is structure in which the message type is setup. For example, if the ROS message type was a vector, the message_typeInfo would be X,Y, or Z.

As long as the loop runs for the entirety of the experiment, the ground station will have updated data from the satellites. Setting up communication on the ground station is now complete.

4.2.6.3 Communication Verification

There are numerous way to verify if the communication network is functioning properly. Some of the most popular ways includes pinging each satellite node from the ground station and looking at the rqt_graph of the the ROS network.

Pinging each satellite from MATLAB is very easy. First all steps of creating the communication network must be completed. The next step is to go to the command window and type:

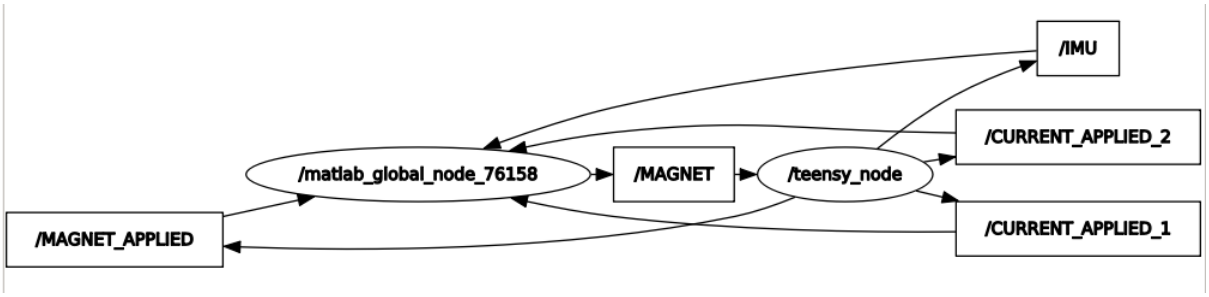
rosnode ping *satellite_node*

MATLAB will try to connect to the specified node with three attempts. If pinging this node fails, the first thing to check is if that satellite and the ground station are on the same wifi network. The next thing to check is if the master node URI is the same for all satellites and the ground station and that all each node ID is correct. This should solve all of the problems; however, if it doesn't then the satellite's and/or ground station's IP addresses have probably been changed by the wifi router.

The other way to verify if the network is working properly is to look at a ROS rqt_graph. For convenience in the future, the ground station should have either MobaXTerm, or VNC viewer installed on them. These softwares allow each satellite to be used as a remote desktop and gives full access to the odroids hard drive. Once installed, open up either software and create a new session or connection and type the IP address of the desired satellite. Once this connection is established, follow the process explained earlier to establish communication on both satellite side and the ground station. Open a Linux terminal and type:

rqt_graph

A window should open that pictorially illustrates all of the active satellite nodes and the ground station communication to each other. An example of this system with one satellite and a ground station is shown below.



4.2.7 ROS Debugging

This section will walk you through some of the common problems faced when dealing with ROS and setting up networks.

4.2.7.1 MATLAB will not connect to the ROS Network or can Send Message but Cannot Receive Messages

If MATLAB does not connect to a ROS Network then it can be one of many problems,but it is most likely to be one of the ones listed below.

1. The ROS Master URI is incorrect

    - You need to decide which computer is going to be the master node. I usually make the master node one of the satellites, so the ROS Master URI should be *http://192.168.1.31:11311* or *http://192.168.32:11311*. In order for this to work though you must also tell the satellites who the master node is. You can do this by going to the *./bashrc* file and typing export ROS_MASTER_URI=http://192.168.1.31:11311 at the bottom of the file.

2. The ROS Node IP Address is incorrect

    - Like with the master URI you need to tell each computer their Node IP address. You can find each computers IP address by opening a terminal window and typing hostname -I on a Linux computer and typing ipconfig /all on Windows. In MATLAB change the ROS_NODE_IP address to the IP address the is associated to that respective computer. On each satellite you have to go to the *./bashrc* file and add export ROS_IP=*satelliteIPaddress* to the end of the file. You only have to add one node IP address per satellite and it should only be the IP address assigned to that computer.

3. The rosinit is the wrong format

    - There are many different formats to initiate the ROS network, but if you follow the format that stated in section 5.2.5 then you should not run into this problem. Make sure that you are following the proper format and verify that the URI's and IP addresses are correct.

4.2.7.2 Roslaunch file errors out

1. The most common reason that the launch file errors out is because it cannot find the teensy on the specified USB port. If you look at the launch file you will see that we specify which port the teensy is plugged into on line 5.

   If you do have the teensy plugged into the correct port and the file still throws an error, then the port you specified is incorrect. you can check the odroid's available ports by opening up a terminal and typing `ls /dev/tty.*` from the home directory. In almost all cases the port you are looking for will be ttyUSB0,ttyUSB1,ttyACM0, or ttyACM1.

2. Another reason that the launch file will not run is if the URI of the master node and if the IP address of the slave node is not correct. To fix this problem, refer to section 5.2.6.1.

3. It is also possible that the format of the launch file is incorrect. If this is the case then look at the type of error that is being thrown and figure out the proper syntax or format of the launch file. Launch file are written in XML so be sure to follows the rules according to that language.

4.2.7.3 Roscore throws an error

1. Most often if roscore is not opening or throws an error it is because the URI of the master node and the slave node are incorrect. See Section 5.2.6.1 to solve this issue.

2. Another common issue is when you clone a git hub repository on the odroid, it doesn't have the correct information according to the new odroid. This is because I creating the repository on a laptop and not an odroid. To fix this problem:

   (a) Open up a terminal window on the odroid.

   (b) Go to the EMFF directory.

   (c) Delete both the build and devel folders by typing `rm -r build` and `rm -r devel`.

(d) Rebuild the EMFF directory by typing catkin_make.

Once catkin_make has successfully run, new build and devel directories will be created and roscore shouldn't throw anymore errors.

4.3 BILL OF MATERIALS

Chapter 5

Software

5.1 Using Motion Capture

5.1.1 Calibration

5.1.2 Selecting a Rigid Body

Selecting a rigid body in motion capture is very simple.

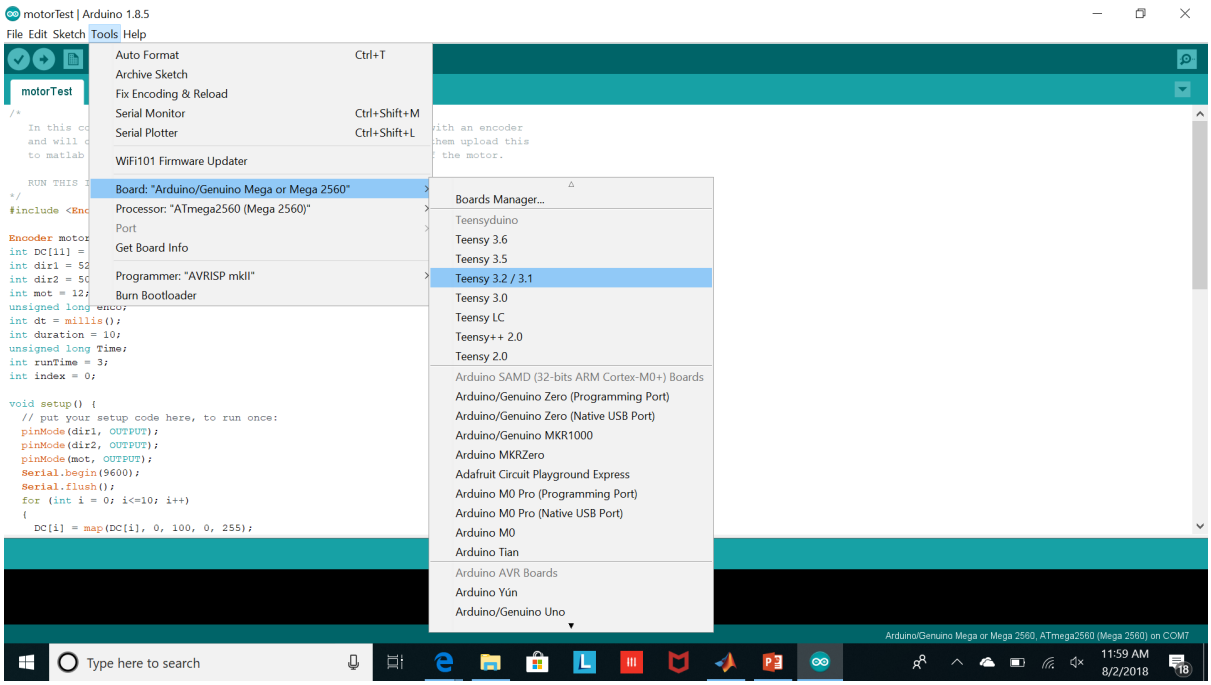
1. Open up the Optitrack Software
2. You should see a bunch of colored points on the screen that represent one or multiple rigid bodies. You now need to select only the points that belong to an individual rigid body by either left clicking and dragging the mouse over the bodies or by hold the shift key and clicking on each point.
3. Once all of the points are selected for the rigid body, right click and select the make a rigid body option.
4. All of the points you've selected should now be a different color and should all be connected to each other. You have now created a rigid body.

Place pictures of this process in a 2X3 grid

5.2 Incorporating the Teensy Boards in Arduino

Teensy uses the same IDE as arduino, but you have to download and installl a few things to be able to upload sketches to your specific teensy board. You can find the website with the files

and instructions on the the teensy website. Once you have successfully installed the teensy add-ons you should be able to select the board you are working with by following the picture below.



5.3 Code on the Teensy

5.3.1 Code Explanation and Walk Through

The code on the teensy can be found in the EMFF directory under the arduino and teesny_32_node folders. In this folder you should find two files: Constants.h and teensy_32_node.ino. The Constants.h file is exactly what it sounds like, and contains all of the systems variables. The teensy_32_node.ino file contains all of the teensy's code and some additional functions. In this next sections I will explain the important sections of the code.

```
14    #include <ros.h>
15    #include <geometry_msgs/Twist.h> //Add the message types
16    #include <geometry_msgs/Pose.h>
17    #include <geometry_msgs/Vector3.h>
18    #include <std_msgs/Float32.h>
19    #include <sensor_msgs/Imu.h>
20    #include <LIS3MDL.h>
21    #include <LSM6.h>
22    #include <Wire.h>
23    #include <math.h>
24
25    #include "constants.h"
26
```

The code above defines all of the libraries being used. Section 6.3.2 will tell you how to include these libraries into the arduino environment.

```
//////////////////////{ PUBLISH }//////////////////////////////////////
geometry_msgs::Vector3 magnetApplied;
std_msgs::Float32 current1;
std_msgs::Float32 current2;
sensor_msgs::Imu IMU;

ros::Publisher magnet_applied("MAGNET_APPLIED", &magnetApplied);
ros::Publisher current_applied_1("CURRENT_APPLIED_1", &current1);
ros::Publisher current_applied_2("CURRENT_APPLIED_2", &current2);
ros::Publisher imu("IMU", &IMU);
////////////////////////////////////////////////////////////////////////
```

This section of code is used for creating the published topics for the ROS network. The first four lines lets the teensy know which libraries to use from ROS and assigns them to a variable. The next four lines creates the topics that the teensy will publish to.

```
//////////////////////{ SUBSCRIBE }//////////////////////////////////////
// Initialized the Subscriber (must know the msg type and the topic that it is subscribing to
 ros::Subscriber<geometry_msgs::Pose> pos("POSITION", &positionCallback ); //Topic == POSITION && Node == pos
 //  ros::Subscriber<geometry_msgs::Twist> velocity("VELOCITY", &velocityCallback ); //Topic == VELOCITY && Node == velocity
 ros::Subscriber<geometry_msgs::Vector3> magnet("MAGNET", &magnetCallback );  //Topic == MAGNET && Node == magnet
////////////////////////////////////////////////////////////////////////
```

The above code sets up the subscribers for our ROS network. These subscribers listen for commands that are being sent from MATLAB. These subscribers also have callback functions that run every time a new message on its topic comes in. The callback functions can be found towards the top of the code.

Insert pub_sub_i2c.PNG

```
146       nh.initNode();
147       nh.subscribe(pos);
148       //  nh.subscribe(velocity);
149       nh.subscribe(magnet);
150
151       nh.advertise(magnet_applied);
152       nh.advertise(current_applied_1);
153       nh.advertise(current_applied_2);
154       nh.advertise(imu);
155
156       Wire.begin(); //Initialize I2C
```

This next segment of code is very important to have. The first line initializes the ROS node. So once this line has been executed you should see the Teensy_Node pop up in the ROS network. The next seven lines will activate the subscribers and publishers of that we have previously created. Lastly, the final line of this segment initializes I2C and enables us to use the various chips on our IMU.

Insert sensorSetup.PNG

```
158     // Make sure the sensors get initialized
159     if (!acc_gyr.init())
160     {
161       //Turn on imu_LED
162       digitalWrite(imu_LED,HIGH);
163       while(1);
164     }
165     acc_gyr.enableDefault();
166     acc_gyr.writeReg(LSM6::CTRL1_XL, 0x3C); // 52 Hz, 8 g full scale
167     acc_gyr.writeReg(LSM6::CTRL2_G, 0x4C); // 104 Hz, 2000 dps full scale
168
169     if (!mag.init())
170     {
171       //Turn on mag_LED
172       digitalWrite(mag_LED,HIGH);
173       while(1);
174     }
175     mag.enableDefault();
176     delay(1000);
177     analogWrite(LED,127);//Turn on good status LED
178     digitalWrite(13,LOW); //Turn off initialization LED
179     time_0=millis(); //Initial Time
180   }
```

In this next segment we are initializing all of our sensors on the the IMU. If any of the initialization processes has an error the debugging LED will turn on to a specific color and the code will be stuck in a loop. If this happens you have to recycle the power on the teensy. Line 178 turns of the status LED on the teensy and to let us know that the initialization period is over. This is different from the debugging LED because it lets us know that we done with the entire initialization process; whereas, the debugging LED just lets us know that the IMU has been initialized and stays on for the entire experiment.

```
241     // Publish the applied values to their topics
242     magnet_applied.publish( &magnetApplied );
243     current_applied_1.publish( &current1 );
244     current_applied_2.publish( &current2 );
245     imu.publish( &IMU );
246
247     nh.spinOnce();
248     delay(10);
```

This last section of code is where we actually publish the data from our sensors. There is also a 10 millisecond delay on the over all loop.

5.3.2 Needed Libraries

We are using several libraries on the teensy, so in order for us to upload the code we have to tell arduino how to find those libraries. If the teensy already has code on it and you do not intend on changing that code then you can ignore the process below. To update the libraries follow the steps below:

1. Locate the EMFF directory on your computer
2. Go into the arduino folder.
3. Go into the lib folder.
4. Copy the ros_lib folder.
5. Go to the folder where you have arduino installed. (This is most likely to be in you Documents folder if you are using Windows)
6. From the Arduino folder you should see a libraries folder. Go into it and paste and ros_lib folder into it.

You will have to do this process every time you add a new library.

5.3.3 Debugging LED

The debugging LED is used to let you know if the code and sensors are working correctly. The LED will change colors depending on the type of problem that is occurring. For example, if all of the sensors are initiated properly and all of the initialization code is complete the LED will be blue.

LED Color	ERROR
Blue	NO ERROR
Red	Accelerometer is not initialized
Green	Magnetometer is not initialized

5.4 CODE IN MATLAB

MATLAB is run on the ground station, and is used to implement the various controllers for the satellites. The ground station code can be found in the MATLAB folder in the EMFF github repository that I have created. To run this code you must have access and include several different files that are used to connect the ground station to motion capture and filter the data from motion capture. All of these additional file needed to connect MATLAB to motion capture can be found here

5.4.1 Setup Motion Capture in MATLAB

In order to get the motion capture data into MATLAB, we must connect the two software packages with a dll file. This is done with the lines of code below.

```
Opti = NET.addAssembly('C:\Users\katar\Dropbox\Quad_experiments\mocap_toolbox\NatNetSDK\lib\x64\NatNetML.dll');
client = NatNetML.NatNetClientML();

client.Initialize('127.0.0.1','127.0.0.1'); % for Local Loopback
```

5.4.1.1 Data Extraction

```
 %% Get data from the vehicles
for i=1:Np
    acc{i}(tt,1)=imu{i}.LatestMessage.LinearAcceleration.X;
    acc{i}(tt,2)=imu{i}.LatestMessage.LinearAcceleration.Y;
    acc{i}(tt,3)=imu{i}.LatestMessage.LinearAcceleration.Z;

    gyro{i}(tt,1)=imu{i}.LatestMessage.AngularVelocity.X;
    gyro{i}(tt,2)=imu{i}.LatestMessage.AngularVelocity.Y;
    gyro{i}(tt,3)=imu{i}.LatestMessage.AngularVelocity.Z;

    magnet{i}(tt,1)=magApplied{i}.LatestMessage.X;
    magnet{i}(tt,2)=magApplied{i}.LatestMessage.Y;
    magnet{i}(tt,3)=magApplied{i}.LatestMessage.Z;

    %time_IMU{Np}(tt)=imu{Np}.LatestMessage.Header.Stamp.Nsec;

    current{i,1}(tt)=currApplied{i}.LatestMessage.Data;
    current{i,2}(tt)=currApplied{i}.LatestMessage.Data;
end
```

The image above shows how to extract data from the satellites. This section of code iterates through all of the satellites in the experiments and stores the sensor data into its corresponding cell array. Once this data is collected the controls and any data analysis can be implemented.