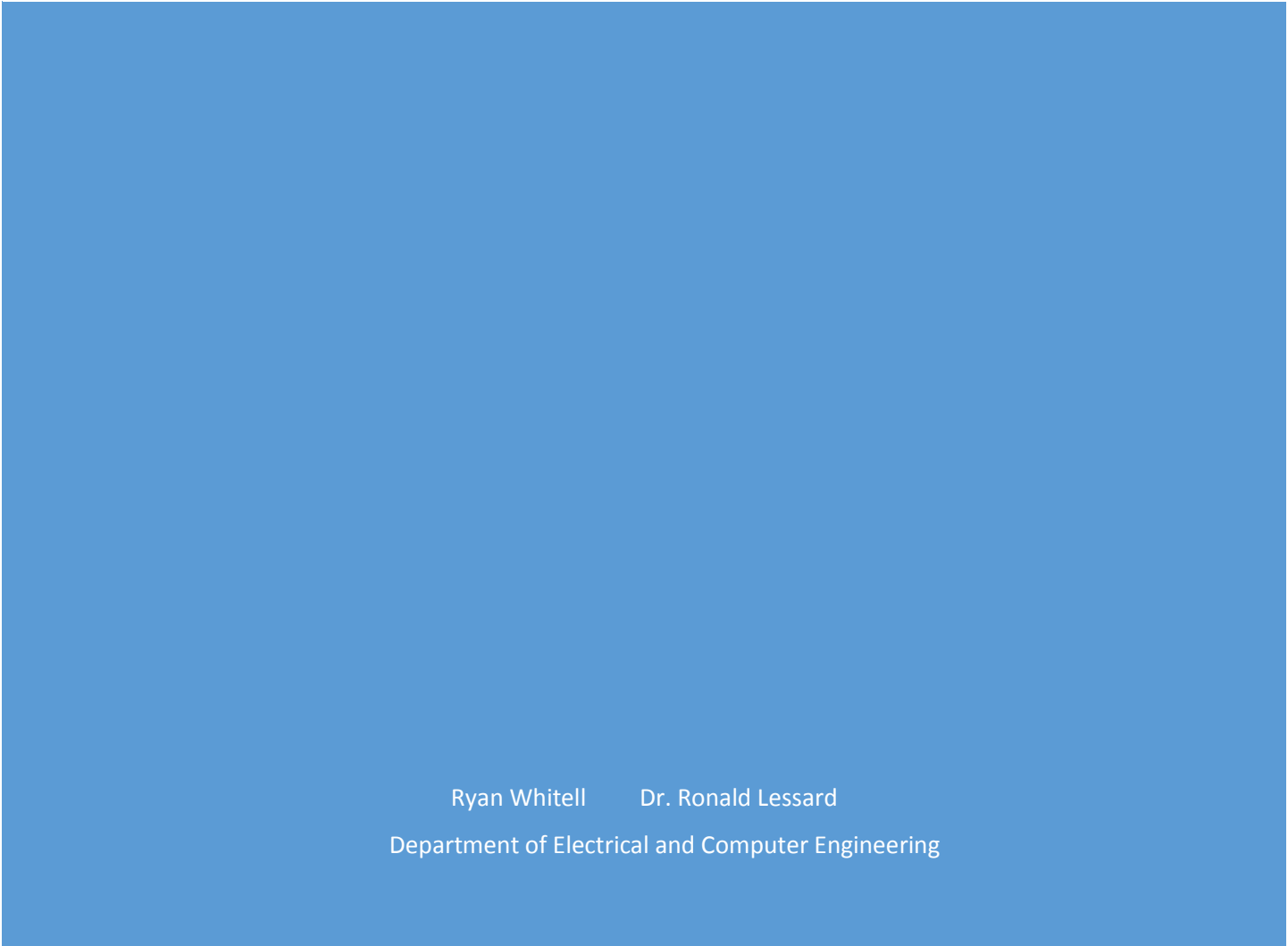




CONTROL OF COLLABORATING ENTITIES: AN INTRODUCTORY APPROACH TO TARGET LOCATION SYSTEMS



Ryan Whitell Dr. Ronald Lessard
Department of Electrical and Computer Engineering

Contents

Abstract	2
Introduction.....	2
Background.....	3
Key Words and Materials.....	3
Raspberry Pi	3
XBee and Explorer Dongle	4
Methods.....	4
Introduction	4
Getting the Raspberry Pis to Talk.....	5
Testing the RSSI/Distance Relationship	7
Analyzing the RSSI/Distance Data	7
Building a Simulation Environment.....	8
Testing	10
Full Scale Test.....	10
Discussion	12
Summary and Conclusions	13
References	14
Appendix.....	15
Appendix A - Configuring the Pis	15
Appendix B - Configuring the XBees	16
Appendix C - Distance Test Code	17
Appendix D - Analyzing the RSSI/Distance Data	18
Appendix E - Simulation Code and Results	21
Appendix F - Test Results	30

Abstract

Ryan Whitell (Dr. Ronald Lessard)

Control of Collaborating Entities: An Introductory Approach to Target Location Systems

Department of Electrical and Computer Engineering

Swarms, formations and teams of collaborating entities have many advantages over complex single system entities performing the same function. Deploying multiple small robots rather than a more complex unit to perform certain actions can decrease overall cost and increase the flexibility and robustness of the entire system. For example, a formation of quadcopter drones could be deployed in a search and rescue environment to locate a target. Two current challenges facing collaborating entities are communications between units and autonomy. This research lays the foundation of *understanding* a novel triangulating target locating system using a formation of three drones without direction finders or non-traditional triangulation methods. The two dimensional concept is proven here in simulation using MATLAB. Then the concept is demonstrated in a field test with PI computers communicating over XBee dongles. This manual SCADA system can be improved further by implementing quadcopters to quickly and autonomously locate a target.

Introduction

For many applications, teams of robots can prove to be cheaper and more effective than a single complex unit. In space missions, for example, smaller satellites are much cheaper to build and easier to launch- if the satellites are small enough, they can piggyback on larger rockets that have unused space. A more earth-bound example can be seen in the realm of quadcopters. Quadcopter technology is rapidly advancing. Sensors are becoming smaller and more accurate, flying mechanics are becoming more precise and air time is increasing. The purpose of this research is to conceptualize a system to aide in Search and Rescue (SAR) missions.

Currently, most SAR missions are done either by foot or by helicopter. These methods are slow, dangerous and one dimensional. Quadcopters could conduct quicker and more efficient SAR missions. The deployment of drones is starting to become more common in aiding SAR teams by taking overhead snapshots of disaster areas. These snapshots are helpful but don't come close to the potential quadcopters have for aiding in SAR missions. This research aims to improve the functionality of drone systems in one key area: formation-flying. A flying formation of quadcopter drones can sweep a larger area than one alone and could pinpoint any signals coming off of devices, such as a cellphones, and converge upon the target.

Using a team of three Raspberry Pis communicating through XBee radio modules a formation and location algorithm will be simulated, first in MATLAB and then a full scale test on a football field will be realized- the results of which can be implemented by a team of quadcopters much more efficiently, autonomously, and over a larger area than that tested.

Background

Key Words and Materials

Drone / Copter – Used interchangeably. Quadcopters with the ability to fly precisely and communicate with each other and the environment.

RSSI – Received signal strength indicator. A measurement of power present in a received radio signal.

XBee – Radio modules. IEEE 802.15.4 protocol.

Packet/Frame – Used interchangeably. The string of bits present in an API communication frame between XBees.

Xbee Explorer Dongle – USB-to-Serial converter. Connects XBees through USB.

XCTU – Human Machine Interface.

MTU – Master terminal unit. Refers to a laptop equipped with the XCTU software and an XBee.

Raspberry Pi – Credit card computer. Able to run python programs and interact with XBee radio modules. Simulates a drone.

USB Battery Pack – Supplies Power to the Raspberry Pi in the field.

Raspberry Pi

The Raspberry Pi is a credit card sized system on a chip running an ARM processor. This project utilizes the Raspbian operating system and model A+ Raspberry Pis. An out of the box Raspberry Pi needs to be configured as in Appendix A. To write and run python programs on the Raspberry Pi requires a few commands in the terminal. Every time the pi boots up it requests an ID and password, by default these are “pi” and “raspberrypi”, respectively. From the terminal type:

```
sudo -i
```

This will invoke full access to the Pi as the superuser. It is wise to create a new directory to save your programs. First, change the directory from the root to “pi”, and then make a new directory called “programs” by issuing the commands:

```
cd /home/pi
```

```
mkdir programs
```

```
cd programs
```

To create and edit files use the pico command. To edit a file, type “pico” and then the file as a .py file:

```
pico existingPythonScript.py
```

To create a new file, issue the same command but give the file a unique name- to check what is already in the directory use the “ls” command. Make sure to append the “.py” to specify python.

Run these programs using the “python” command.

An example of how this should look in the terminal:

```
root@raspberrypi:/home/pi/programs# pico HelloWorld.py
```

```
# This is the file, save by pressing ctr-x
```

```
print "Hello World"
```

```
root@raspberrypi:/home/pi/programs# python HelloWorld.py
```

```
Hello World
```

XBee and Explorer Dongle

The XBee is a radio frequency module produced by Digi International Inc. This project uses the Series 1, IEEE 802.25.4 standard with wire antenna. The Xbee modules are configured as in Appendix B. The Explorer Dongle from SparkFun is a USB-to-Serial converter and is used to connect the XBees to each Raspberry Pi and the MTU (laptop).

Methods

Introduction

A proof of concept for this project was done in code using MATLAB. This initial step provided a background of knowledge necessary to move on. Because the drones only have a signal strength reading to work with, a robust solution needed to be conceptualized before any full scale testing was done. The program had to be written with certain restrictions; the testing area had to be the size of a football field and the RSSI distance estimation had a maximum error of about 30m. An algorithm was developed that utilized a formation of drones to find a target at an unknown location while keeping within these constraints. All simulations and tests are conducted in two dimensions and have a distance restriction of about 90m. The algorithms developed will have to work in three dimensions (with a few small, manageable adjustments) and be scalable to greater distances.

Raspberry Pi's were used to simulate the drones and XBee radio modules were used for communication and signal generation. Some of the concepts and algorithms were developed and tested strictly in MATLAB before any of the hardware was used. These initial simulations provided a solid foundation to work on but no testing was done leading up to this point in determining how the Raspberry Pis and XBees might behave. Therefore, the relationship of the distances between XBee modules and their

corresponding RSSI readings on the football field had to be determined before moving on. Once this relationship was established a more robust simulation was developed and a full scale test was realized. But first, the Pis must communicate.

Getting the Raspberry Pis to Talk

Getting the Raspberry Pis to communicate serially through XBees meant first understanding how the XBees communicate to each other and then determining how to implement these communications in code. There are many different variations and modes for the XBees to send and receive data but only in API (Application Programming Interface) mode will the Raspberry Pis be able to easily determine the RSSI value of an incoming signal. API mode allows data to be sent and received as packets of data or “frames.” A frame is broken down in the following way:

- Start Delimiter: Start delimiter of the API frame. Always set to 0x7E.
- Length: Number of Bytes between length and checksum fields.
- Frame Type: Indicates the frame type of the API frame. This project uses 0x00- Long Address Transmit Requests that get received as 0x80- Long Address Received Packets.
- Frame ID: Identifies the UART data frame for the host to match with a subsequent response. If zero, no response is requested. Set to zero, responses would add unwanted noise to the system and cause program errors.
- 64-bit Dest. Address: Set to the 64-bit address of the destination device. This is the MAC Address of the XBee.
- Options: Bitfield to enable various command options. Ignored.
- RF Data: Packet Payload (up to 256 bytes)
- Checksum: FF minus 8-bit sum of bytes between the length and the checksum fields.
- RSSI: Received Signal Strength Indicator (Receive Packets)
- Source Address: Source Address for Receive Packets (00 00 Universal)

Figure 1 shows an example of how packets are sent and received. XBee A sends out a transmit request containing the MAC address of XBee B. XBee B picks up the request, confirms the address and reads the data. The RSSI value is included as part of the “receive packet” frame.

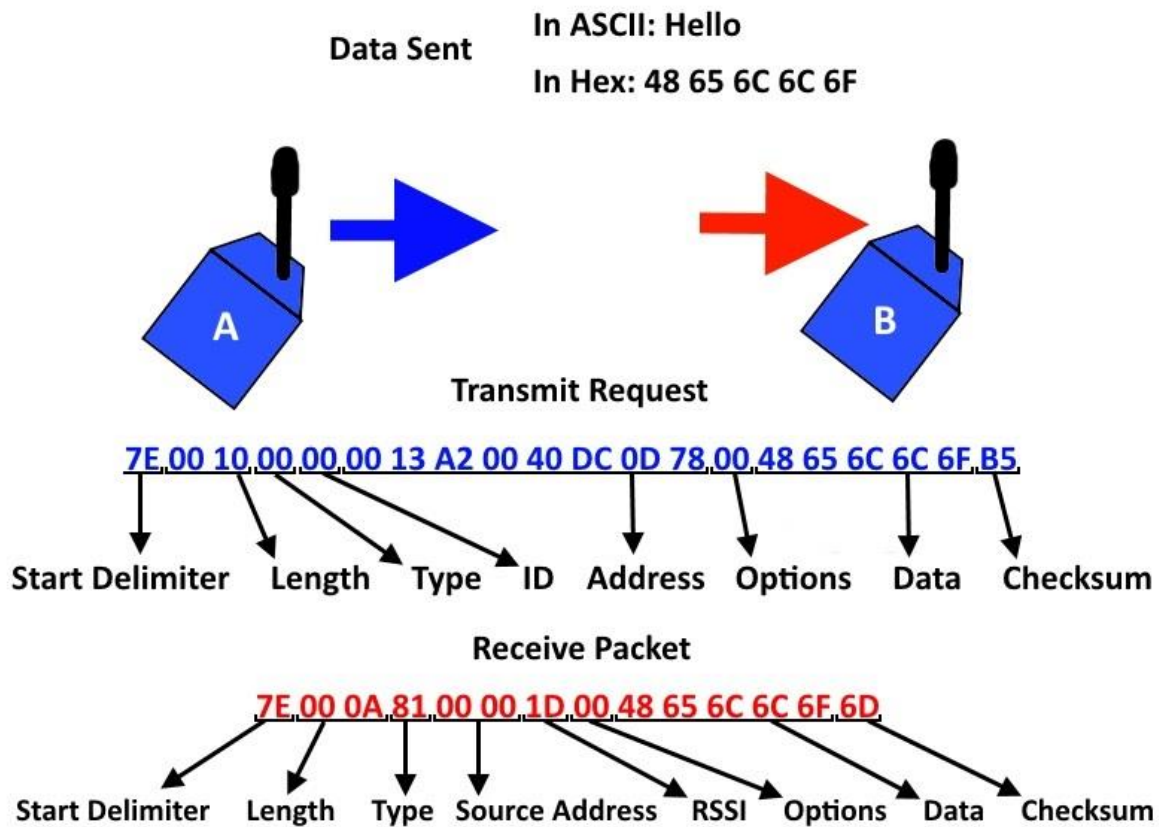


Figure 1: Example of XBee API protocol communication sending "Hello" from XBee-A to XBee-B

The communication between two XBees can be handled in python by use of the XBee library written by Paul Malmsten which, “provides a semi-complete implementation of the XBee binary API protocol and allows a developer to send and receive the information they desire without dealing with the raw communication details” [1]. The library is imported into a python program with the line:

```
from xbee import XBee
```

Initializing an XBee object requires some information about the XBee (the serial port that the XBee is connected to and the baud rate) and the python serial library:

```
import serial

SERIALPORT = "/dev/ttyUSB0"

BAUDRATE = 9600

ser = serial.Serial(SERIALPORT,BAUDRATE)
```

```
xbee = XBee(ser)
```

The function for receiving data continuously reads the serial port and converts incoming packets into a python dictionary. If XBee B of Figure 1 were connected to a Raspberry Pi, the received frame would look as follows:

```
dictionary = xbee.wait_read_frame()

print dictionary

>> {'rf_data': 'Hello', 'rssi': '+', 'source_addr': '\x00\x00', 'id':
'rx', 'options': '\x00'}
```

Notice the RSSI value and Frame ID are in ASCII notation.

The function for sending data works the opposite way- converting a python dictionary into a frame and transmitting:

```
hello = 'Hello'

xbee.send("tx_long_addr", frame_id='\x00',
dest_addr='\x00\x13\xA2\x00\x40\xDC\x0D\x78', data=hello)
```

Testing the RSSI/Distance Relationship

Tracking and locating a target using only RSSI values can be either simple or complex depending on many factors including but not limited to: multipathing, interference from objects or people, outside noise/signals, etc...To limit these factors, all tests were done on a football field. This wide open area provided a good approximation of a “free space” environment.

Testing for distance data was straightforward. Two wooden stools were used to prop up the Raspberry Pis and make them easy to move around. RPI-A was placed on one stool at the 0 yard marker and three Raspberry Pis (RPI-B,C,D) were placed on the second stool at the 0 yard marker. The MTU sends a start command to RPI-A. RPI-A runs a program that pings RPI-B and records the RSSI value into an array. It repeats this processes for RPI-C and RPI-D a number of times. It then averages the value and sends it back to the MTU where it is recorded. The stool is moved one yard and the process is repeated. Taking multiple readings from multiple Raspberry Pis and averaging them increases the accuracy of the measurement. This is important because the RSSI values fluctuate a bit, sometimes a lot. The code used for this test is shown in Appendix C.

Analyzing the RSSI/Distance Data

Equation 1 is used for estimating the distance given the RSSI value for the XBee [2].

$$distance = 10^{\frac{RSSI-A}{10n}} \quad (1)$$

Parameter A is the RSSI value at a reference distance, in this case 1 meter, and was found to be 43. Parameter n is the path loss exponent from the log-distance path loss model. This value was obtained in MATLAB by finding the least amount of error for n values ranging from 0 to 4 and was determined to be 1.8. An n value of 2 corresponds to free space. Values greater than 2 represent environments with obstacles and path loss or attenuation. The football field is a good approximation of free space, however a value below 2 indicates some mulitpathing or outside sources are still disrupting the data. Appendix D contains the data and the MATLAB code used to analyze it. As shown in Figure 2, the data collected follows closely to the equation.

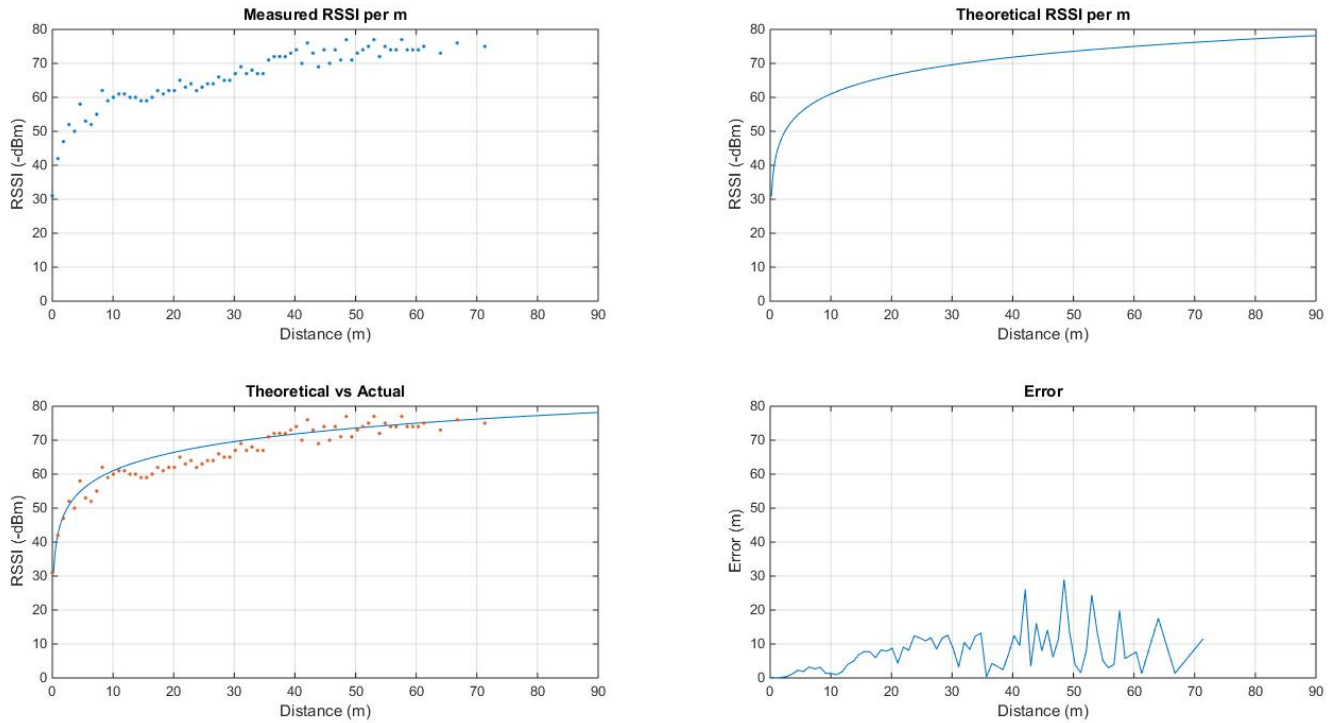


Figure 2: Analysis of the distance data. Equation 1 best represents the data trend.

Building a Simulation Environment

Building a simulation in MATLAB provided a working algorithm and a proof of concept. With a few minor adjustments to the code the simulation became the main working program for the full scale test. By using a formation of drones a target can be found through trilateration. The algorithm can predict where the target is based off of all three drone RSSI readings and then move towards the predicted point. This process is repeated until the RSSI value of any drone is less than 52 dBm, meaning that this drone is likely 3 meters away from the target. Figure 3 presents a flow chart that demonstrates the algorithm:

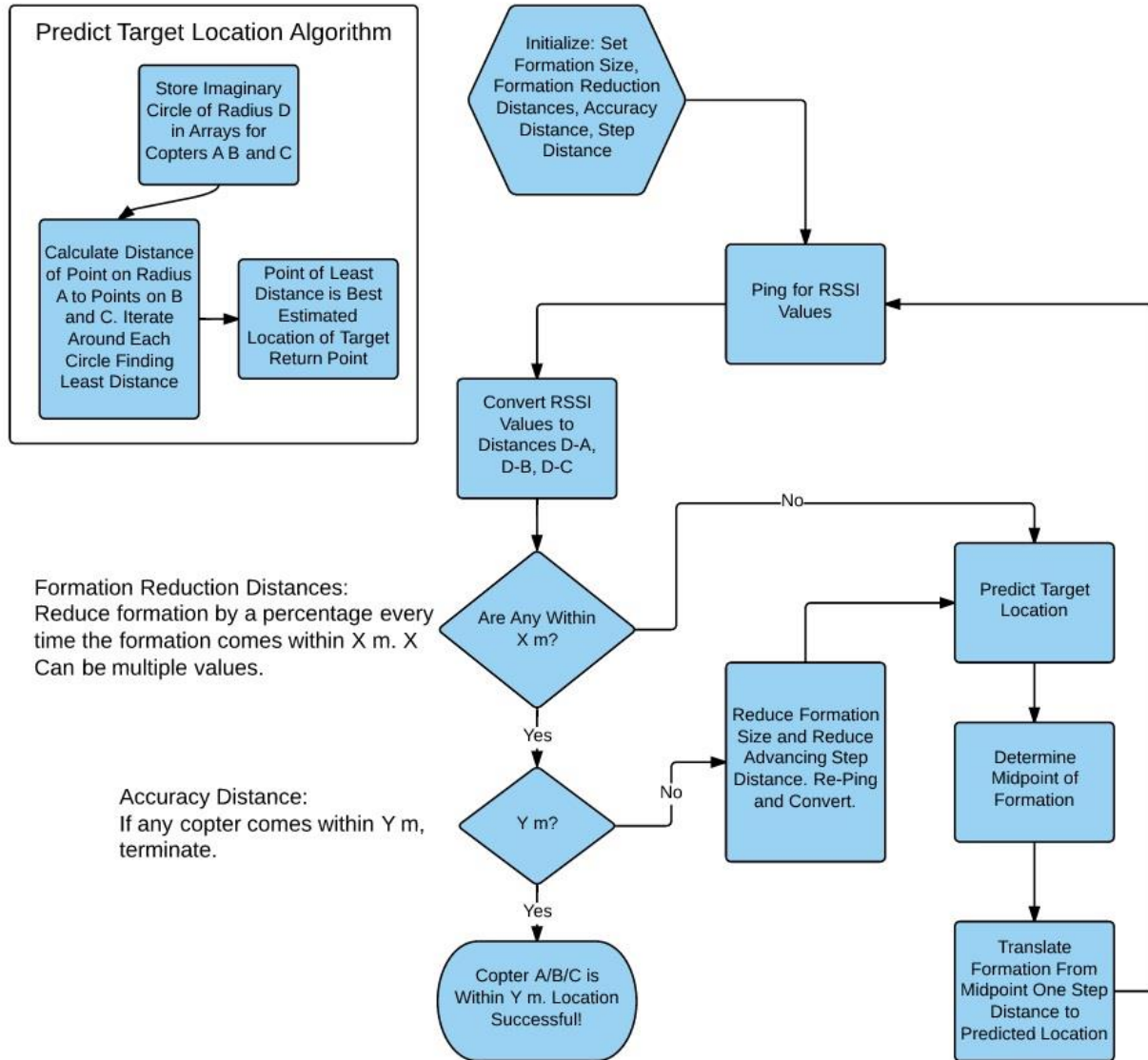


Figure 3: Algorithm flow chart.

The initial setup places a target and a formation on the football field. The formation in this case being an equilateral triangle of side length 20 meters (although this is arbitrary, the formation is flexible as long as it is large enough as mentioned later). Because the range of the XBees are only about 90 meters, the formation and the target must be placed within the bounds of the football field. Any initial setup

adhering to these boundaries should have the formation tending towards and eventually finding the target. Figure 4 shows the simulation results (for the code that produces this result refer to Appendix E). The black dots are the midpoint of the formation as it moves- they are plotted every time the formation steps leaving a trail.

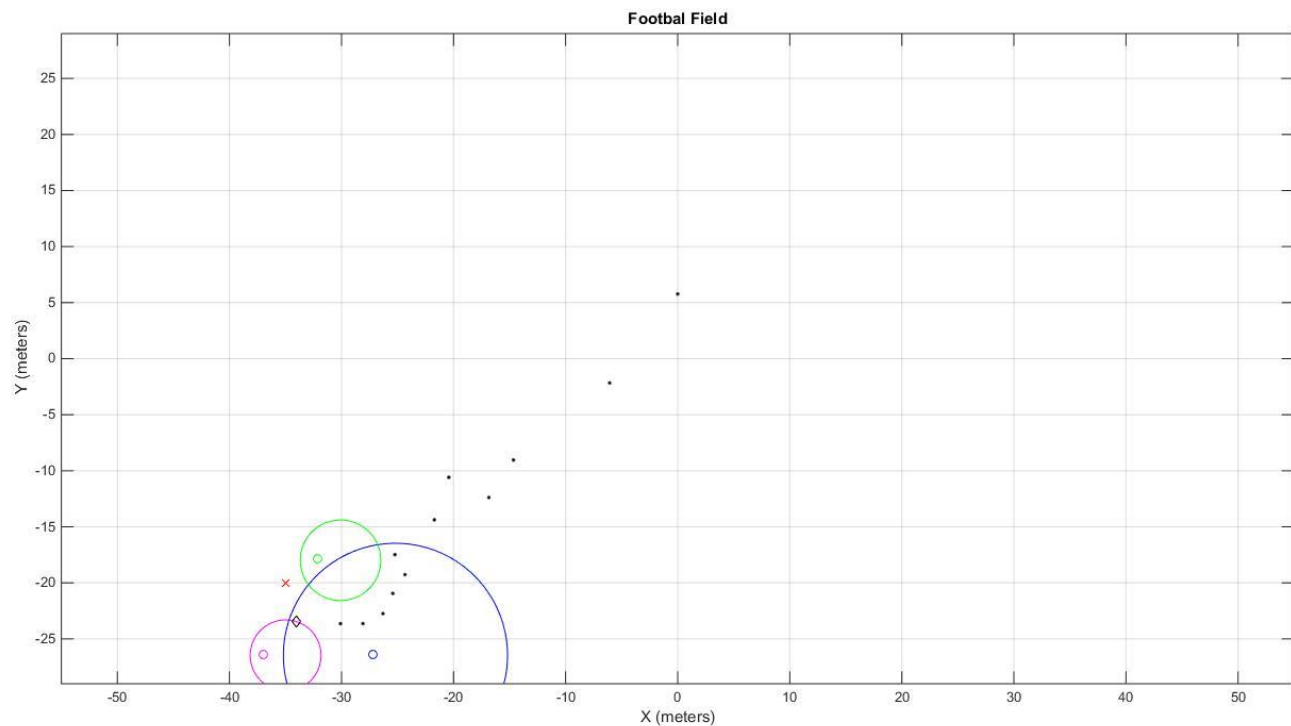


Figure 4: Simulation result. The target is marked as the X and the circles are plotted for help visualizing the algorithm. The algorithm predicts the location of the target and is plotted as a diamond. This diamond represents the best estimation of direction for the formation to move towards- It is the spot closest to each circle.

Testing

Full Scale Test

For the full scale test, the simulation code in MATLAB was modified to prompt the user for the RSSI values and then outputs an X and a Y value in meters to move the formation, which was done manually. The RSSI values were obtained by the MTU sending a command to each Raspberry Pi in the formation one at a time. The MTU uses the XCTU software to send and receive commands. The Pi receiving the command from the MTU pings the target multiple times and sends back the average value to the MTU. The Pis were running near identical programs as in the RSSI/Distance test (Appendix C). The Pis were propped up onto equal height cardboard boxes to limit multipath and noise. Three boxes were placed in a formation near the 50 yard line and the target box was put near the sideline at 10 yards. Figures 5 and 6 show the initial setup, and figure 7 shows the final result. The process of retrieving RSSI values, imputing them into MATLAB, and manually moving the boxes was repeated until the target box was found.

The formation tended towards the target every step except once. It took only about seven steps and two formation reductions to get within 3 meters of the target. Copter C ended up being 1 meter from the target.

Refer to Appendix F for the MATLAB code and console input/output.



Figure 5: Raspberry Pi on boxes at around mid-field



Figure 6: Target Pi at the 10 yard line.



Figure 7: The formation successfully located and converged upon the target.

Discussion

The full scale test was more of a success than predicted. On first try, the formation tended towards the target very efficiently. Considering the imprecise nature of measuring and moving the boxes by hand, the algorithm seems to be very flexible. As long as the formation stays relatively stable, the predicted point will almost always be in the general direction of the target.

The MATLAB program used an initial Cartesian reference frame and translations to determine the current formation. When working with quadcopters it may not be necessary for a reference frame to be established. If the quadcopters can determine each other's relative positions they can determine and hold their formation before pinging for RSSI values and locating the target, regardless of where they are in space. This type of flexibility would reduce the need for the drones to record their movements. This would be useful especially in windy environments that would cause the copters to veer off course and

reduce the accuracy of position monitoring techniques such as dead reckoning or velocity integration. This type of system would require some sort of distance sensors between copters or GPS, however.

The full scale test was conducted in two dimensions, the formation and the target being on the same level. Real drones would fly much higher than the target and enter three dimensional space. By inspection, this could be handled either by creating imaginary spheres instead of circles or by converting the RSSI readings into a radius length provided that the copters can determine their height above the target. Further investigation is needed on this aspect.

The full scale test also had all of the computations handled by the MTU in MATLAB. An autonomous system would have one of the drones designated the captain. The captain would request the RSSI readings and positions from the other drones and handle the location estimating. It would then communicate back the results to the other drones and possibly a controlling unit on the ground. Using more drones could slow down and complicate the calculations but would provide a more accurate location point. Teams of drones could also take shorter steps and more frequently run the algorithm, limiting the impact of the stepping away from the target which happens every so often.

A real search and rescue system would encounter many problems not considered in this research. By using XBees, the radio signals were isolated in frequency and protocol. Picking up a lost hikers cell phone or wifi signal would require different hardware or protocols. Also, the environment will most likely be much more noisy or extreme. Many things should be considered if building towards a final product. But in a controlled environment this system works well as an undergraduate lab or final project.

Summary and Conclusions

This research proved the concept that a team of drones can locate and converge upon a target. This project can be repeated through learning radio communication, algorithm development, feedback, and embedded systems. This project lays a foundation for a target location system using formation flying drones and leaves room for advances and improvement.

References

- [1]P. Malmsten, “python-xbee Documentation”, Release 2.1.0, 2013.
- [2]B. Lee and W. Chung, 'Multitarget Three-Dimensional Indoor Navigation on a PDA in a Wireless Sensor Network', *IEEE Sensors Journal*, vol. 11, no. 3, pp. 799-807, 2011.

Appendix

Appendix A - Configuring the Pis

The Raspberry Pi model A+ comes bare out of the box and needs an operating system to be installed. The easiest way to achieve this is to purchase a micro sd card with an operating system boot loader. For this project, an 8GB micro sd card with NOOBS was purchased for each model A+ and was used to install the Raspbian operating system on each Pi. Raspbian is a clone of Debian, a common Linux distribution, optimized to run on the Raspberry Pi hardware.

Because the Raspberry Pi was developed in the UK, it is useful to change the keyboard layout to US standard. This can be done by typing the command:

```
sudo raspi-config
```

While in the command window make that SSH is enabled and then navigate to the “internationalization” options and change the keyboard layout.

In order to download the python libraries needed for this project to work the Pi’s must be connected to the internet. Because the A+ model only has one USB port, the Pi needs to be configured so that it automatically connects to a wifi network when a USB wifi dongle is connected. In the command line type:

```
sudo nano /etc/wpa_supplicant/wpa_supplicant.conf
```

This will open up the wpa supplicant. In this file, below the other text, add the lines:

```
network={  
    ssid="network name"  
    psk=" network password"  
}
```

Change the content in the quotations to match your network, keeping the quotations. Save and exit the file. When the wifi dongle is plugged in your Raspberry Pi should automatically connect to your wifi network. Get the IP address (this shows up on the last line of a successful boot with the wifi dongle plugged in, right before it asks you for the login ID). Use a remote SSH client like PuTTY to run the Pi headless (if you are using a model B you do not need to do this, the model A+ only has one USB port taken up by the wifi dongle so to issue commands you need to run it headless).

When the Pi is successfully connected to the internet, connect to it using an SSH client by inputting IP address. It should ask you to login on the headless terminal. Login and run these commands in order to download all of the necessary libraries for this project:

```
sudo apt-get update  
sudo apt-get install python-dev  
sudo apt-get install python-pip  
sudo shutdown -r now
```


Appendix B - Configuring the XBees

Configuration of the Xbees is most easily done using Digis' XCTU software. Because the Series 1 XBees use the 802.15.4 function set, the out of the box firmware does not have to be changed. Plug the XBee into a laptop using an XBee Explorer Dongle and discover it using the XCTU software. In the configuration window, change and write the appropriate settings. The configuration window is shown in Figure 8.

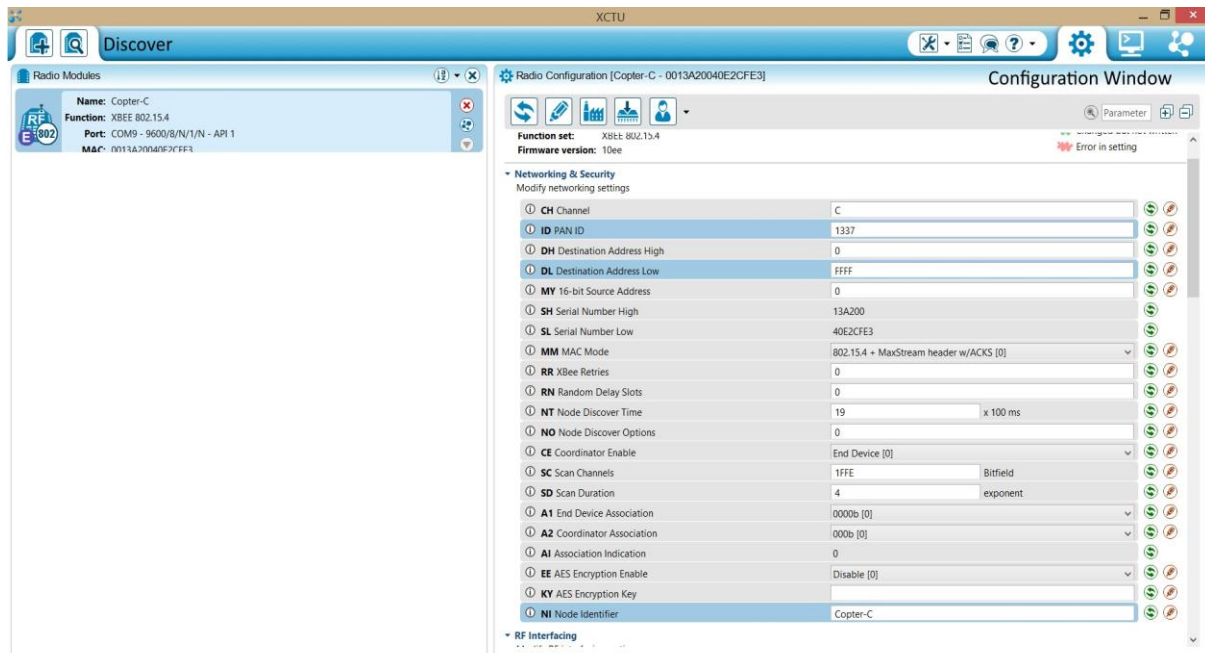


Figure 8: XCTU software interface.

The modified settings for this project were:

- PAN ID: 1337
- Destination Address Low: FFFF
- Node Identifier: Copter-A, Copter-B, Copter-C, Target and MTU
- All XBees were set to API mode

Appendix C - Distance Test Code

The following program is run by Raspberry Pi A. RPI-A waits for the 'GETRSSI' command from the MTU. Once it receives the command it sends a command to another RPI running a similar program, that RPI sends back a packet containing RSSI information, which RPI-A stores in an array. It repeats this process with the other two RPI's sitting at the same location 300 times and then averages the data. The result is then sent back to the MTU.

```
import serial, time, datetime, sys
from xbee import XBee
import numpy as np

print 'Time Delay Switch to XBee'
time.sleep(10)
print 'Delay Over'

SERIALPORT = "/dev/ttyUSB0"
BAUDRATE = 9600

ser = serial.Serial(SERIALPORT, BAUDRATE)

xbee = XBee(ser)

while True:
    try:
        command = xbee.wait_read_frame()
        total = np.array([],int)
        if command['rf_data'] == 'GETRSSI':
            for x in range(0,300):
                xbee.send("tx_long_addr",
frame_id='\x00', dest_addr='\x00\x13\xA2\x00\x40\xE2\xCF\xE3',
data='\x50\x49\x4E\x47')

                frame = xbee.wait_read_frame()
                rssi_ASCII = frame['rssi']
                rssi_int = ord(rssi_ASCII)
                total = np.append(total,rssi_int)
                time.sleep(0.01)
                xbee.send("tx_long_addr", frame_id='\x00',
dest_addr='\x00\x13\xA2\x00\x40\xE2\xD0\xAA', data='\x50\x49\x4E\x47')
                frame = xbee.wait_read_frame()
                rssi_ASCII = frame['rssi']
                rssi_int = ord(rssi_ASCII)
                total = np.append(total,rssi_int)
                time.sleep(0.01)
                xbee.send("tx_long_addr", frame_id='\x00',
dest_addr='\x00\x13\xA2\x00\x40\xDC\x0D\x78', data='\x50\x49\x4E\x47')
                frame = xbee.wait_read_frame()
                rssi_ASCII = frame['rssi']
```

```

        rssi_int = ord(rssi_ASCII)
        total = np.append(total, rssi_int)
        avg_rssi = str(int(np.mean(total)))
        time.sleep(0.01)
        xbee.send("tx_long_addr", frame_id='\x00',
dest_addr='\x00\x13\xA2\x00\x40\xE2\xCF\xC5', data=avg_rssi)
    except KeyboardInterrupt:
        break

ser.close()

```

The following program is run by Raspberry Pis B,C, and D. They sit and wait for the 'PING' command and then send back a transmission containing RSSI data for RPI-A to store in an array.

```

import serial, time, datetime, sys
from xbee import XBee

print 'Time Delay Switch to XBee'
time.sleep(10)
print 'Delay Over'

SERIALPORT = "/dev/ttyUSB0"
BAUDRATE = 9600

ser = serial.Serial(SERIALPORT, BAUDRATE)

xbee = XBee(ser)

while True:
    try:
        command = xbee.wait_read_frame()
        if command['rf_data'] == 'PING':
            xbee.send("tx_long_addr", frame_id='\x00',
dest_addr='\x00\x13\xA2\x00\x40\xD8\x76\x2E',
data='\x50\x49\x4E\x47\x42\x41\x43\x4B')
    except KeyboardInterrupt:
        break

ser.close()

```

Appendix D - Analyzing the RSSI/Distance Data

MATLAB code:

```

function n = optimize()
% This function determines the appropriate n value corresponding to the
% least amount of error

%Data
rss_i = [31 42 47 52 50 58 53 52 55 62 59 60 61 61 60 60 59 59 60 62 61 62 62
65 63 64 62 63 64 64 66 65 65 67 69 67 68 67 67 71 72 72 72 73 74 70 76 73 69
74 70 74 71 77 71 73 74 75 77 72 75 74 74 77 74 74 74 75 73 76 75];
yards= [00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 70 73 78];

meters = yards*0.9144;
dist = 10.^((rss_i-43)/41);
old_error = sum(abs(dist - meters));
for x = 0:0.1:4
    dist = 10.^((rss_i-43)/(10*x));
    new_error = sum(abs(dist - meters));
    if new_error < old_error
        old_error = new_error
        n = x;
    end
end
end

```

```

function distance_sabine()
% This function plots the recorded data

one_meter = [43];
rss_i = [31 42 47 52 50 58 53 52 55 62 59 60 61 61 60 60 59 59 60 62 61 62 62
65 63 64 62 63 64 64 66 65 65 67 69 67 68 67 67 71 72 72 72 73 74 70 76 73 69
74 70 74 71 77 71 73 74 75 77 72 75 74 74 77 74 74 74 75 73 76 75];
yards= [00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 70 73 78];
meters = yards*0.9144

subplot(2,2,1)
plot(meters,rss_i, '.');
axis([0 90 0 80]);
grid on
title('Measured RSSI per m')
ylabel('RSSI (-dBm)')
xlabel('Distance (m)')

subplot(2,2,2)
db = 31:1:90
dist = 10.^((db-43)/18)
plot(dist,db)
axis([0 90 0 80]);
grid on
title('Theoretical RSSI per m')
ylabel('RSSI (-dBm)')
xlabel('Distance (m)')

```

```
subplot(2,2,3)
plot(dist,db)
hold on
plot(meters,rssi,'. ');
axis([0 90 0 80]);
grid on
title('Theoretical vs Actual')
ylabel('RSSI (-dBm)')
xlabel('Distance (m)')

dist = 10.^((rssi-43)/18)
error = abs(dist - meters)
subplot(2,2,4)
plot(meters,error)
hold on
axis([0 90 0 80]);
grid on
title('Error')
ylabel('Error (m)')
xlabel('Distance (m)')
```

Appendix E - Simulation Code and Results

MATLAB Code:

```
function initialize() % Run initialize function to start simulation

% Declare and initialize copter formation
global A B C;

A = set_se2(-10,0,0);
B = set_se2(10,0,0);
C = set_se2(0,sqrt(20^2-10^2),0);

% Plot copter formation and target
plot_position(A,'A');
plot_position(B,'B');
plot_position(C,'C');
T = set_target(-35,-20);

% Locate and converge upon target
EXIT_FLAG = 0;
within = Inf;
while 1
    x = 10;
    within = is_D_m_from_target(A,B,C,T,within);
    if within == 10
        within = 10;
        reduce_formation_size(A,B,C);
        while 1
            x = 4;
            within = is_D_m_from_target(A,B,C,T, within);
            if within == 5
                within = 5;
                reduce_formation_size(A,B,C);
                while 1
                    x = 2;
                    within = is_D_m_from_target(A,B,C,T, within);
                    if within == 3
                        EXIT_FLAG = 1;
                        break;
                    end
                    if EXIT_FLAG, break, end
                    advance(A,B,C,T,x)
                end
            end
            if EXIT_FLAG, break, end
            advance(A,B,C,T,x)
        end
    end
    if EXIT_FLAG, break, end
    advance(A,B,C,T,x)
end
```

```

end
function position_global = set_se2(x, y, yaw)
% SET_SE2
% This function initializes a copter using a Special Euclidian
% Transformation Matrix in two-dimensional space. Easily updatable
% Two three dimensions

position_global = [cosd(yaw) -sind(yaw) x; sind(yaw) cosd(yaw) y; 0 0 1];
end
function plot_position(position_global,identifier)
% PLOT_POSITION
% This function plots the position of the copter that is passed in as an
% argument

global APLLOT BPLLOT CPLLOT;

if identifier == 'A'
    delete(APLLOT);
    APLLOT = plot(position_global(1,3),position_global(2,3),'mo');
end
if identifier == 'B'
    delete(BPLLOT);
    BPLLOT = plot(position_global(1,3),position_global(2,3),'bo');
end
if identifier == 'C'
    delete(CPLLOT);
    CPLLOT = plot(position_global(1,3),position_global(2,3),'go');
end

grid on
title('Football Field')
xlabel('X (meters)')
ylabel('Y (meters)')
axis equal
axis([-55,55,-29,29])
hold on
end
function within = is_D_m_from_target(A,B,C,T, current)
% This function determines if any of the quad copters are "D" meters from the
% target and returns the distance.

rssi_a = get_rssi(A,T);
rssi_b = get_rssi(B,T);
rssi_c = get_rssi(C,T);

within = Inf;

if current > 10
    if (rssi_a < 62)
        fprintf('Copter A is within 10 meters from the target \n')
        within = 10;
    end
    if (rssi_b < 62)
        fprintf('Copter B is within 10 meters from the target \n')
        within = 10;
    end
end

```

```

        end
        if (rssi_c < 62)
            fprintf('Copter C is within 10 meters from the target \n')
            within = 10;
        end
    elseif current > 5
        if (rssi_a < 56)
            fprintf('Copter A is within 5 meters from the target \n')
            within = 5;
        end
        if (rssi_b < 56)
            fprintf('Copter B is within 5 meters from the target \n')
            within = 5;
        end
        if (rssi_c < 56)
            fprintf('Copter C is within 5 meters from the target \n')
            within = 5;
        end
    else
        if (rssi_a < 52)
            fprintf('Copter A is within 3 meters from the target \n')
            within = 3;
        end
        if (rssi_b < 52)
            fprintf('Copter B is within 3 meters from the target \n')
            within = 3;
        end
        if (rssi_c < 52)
            fprintf('Copter C is within 3 meters from the target \n')
            within = 3;
        end
    end
end
end

```

```

function rssi = get_rssi(copter, target)
% This function uses the data from the RSSI/Distance test to
% Return the RSSI value closest to the distance
% This simulates the readings that might be obtained from the full scale
% Test

% Data Model
rssi_measured = [31 42 43 47 52 50 58 53 52 55 62 59 60 61 61 60 60 59 59 60
62 61 62 62 65 63 64 62 63 64 64 66 65 65 67 69 67 68 67 67 71 72 72 72 73 74
70 76 73 69 74 70 74 71 77 71 73 74 75 77 72 75 74 74 77 74 74 74 75 73 76
75];
yards= [00 01 1.09361 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 70 73 78];

distance = sqrt((copter(1,3)-target(1))^2 + (copter(2,3)-target(2))^2);
meters = yards*0.9144;
diff = abs(distance - meters(1));

for x = 2:71
    new_diff = abs(distance - meters(x));
    if new_diff < diff
        diff = new_diff;
    end
end

```



```

        rssi = rssi_measured(x);
    end
end
end


---


function reduce_formation_size(CA,CB,CC)
% This function reduces the formation size by a factor of 15%

global A B C rssi_a rssi_b rssi_c;

A = set_rotation(A, 0);
B = set_rotation(B, 180);
C = set_rotation(C, 270);

percentage = 0.15;

factor = 1 - (percentage*2);
dist_ab = sqrt((CA(1,3)-CB(1,3))^2+(CA(2,3)-CB(2,3))^2);
dist = dist_ab*percentage;
distc = abs(CA(2,3)-CC(2,3)) - sqrt((factor*dist_ab)^2-(factor*...
    dist_ab/2)^2);

fprintf('Reduce Formation Size \n');

A = local_translation(A,'A',dist,0);
B = local_translation(B,'B',dist,0);
C = local_translation(C,'C',distc,0);
end


---


function new_position_global = set_rotation(position_global,yaw)
% This function sets the rotation of the copter that is passed in as an
% argument in the Special Euclidian Matrix

new_position_global = [cosd(yaw) -sind(yaw)...
    position_global(1,3); sind(yaw) cosd(yaw)...
    position_global(2,3); 0 0 1];
end


---


function new_position_global =
local_translation(position_global,identifier,x,y)
% This function takes as its input arguments a copter, a string
% identifying which copter it is (A,B,C) and an x and y translation value.
% This function translates the copter by concatenating matrices

new_position_global = position_global*[1 0 x;0 1 y;0 0 1];

plot_position(new_position_global,identifier);
end


---


function point = locate_target(A,B,C,T)
% This is the main running algorithm for locating the target. This
% algorithm uses the rssi level readings from each copter to create
% "imaginary" circles (trilateration) and then outputs the point on copter
% A's circle that
% is closest to both copter B and C's circles. This point should be close
% to the target.

global rssi_a rssi_b rssi_c;

```

```

% Read in the rssi levels (this is a distance in m, in real practice it will
% be an rssi level reading which will have to be converted to m)
rssi_a = get_rssi(A,T);
rssi_b = get_rssi(B,T);
rssi_c = get_rssi(C,T);
rad_a = convert_rssi(rssi_a);
rad_b = convert_rssi(rssi_b);
rad_c = convert_rssi(rssi_c);

% Plot the imaginary circles for visualization
plot_circle(A(1,3),A(2,3),rad_a,'A');
plot_circle(B(1,3),B(2,3),rad_b,'B');
plot_circle(C(1,3),C(2,3),rad_c,'C');

% Create 1001 plot points
angle=0:(2*pi)/1000:2*pi;

% Initialize the imaginary circles in a 2x10001 matrix
% Each column corresponds to a point (x,y) on the circle
AR = [rad_a*cos(angle)+A(1,3);rad_a*sin(angle)+A(2,3)];
BR = [rad_b*cos(angle)+B(1,3);rad_b*sin(angle)+B(2,3)];
CR = [rad_c*cos(angle)+C(1,3);rad_c*sin(angle)+C(2,3)];

% Initialize a large distance for error checking
distance = 1000;

% Iterate around every circle and check if the total distance between
% points is a minimum
for i = 1:7:1001
    for j = 1:7:1001
        for k = 1:7:1001
            new_distance = sqrt((AR(1,i)-BR(1,j))^2+...
            (AR(2,i)-BR(2,j))^2) + sqrt((BR(1,j)-CR(1,k))^2+...
            (BR(2,j)-CR(2,k))^2) + sqrt((AR(1,i)-CR(1,k))^2+...
            (AR(2,i)-CR(2,k))^2);
            if new_distance < distance
                distance = new_distance;
                point = [AR(1,i);AR(2,i)];
            end
        end
    end
end

if distance == 1000
    fprintf('Too much error, optimize and re-try\n');
    return
end

% Plot a black diamond at the point
global BLACKDIAMOND;
delete(BLACKDIAMOND);
BLACKDIAMOND = plot(point(1),point(2),'kd');
end
function distance = convert_rssi(rssi)

```

```

% This function uses the distance equation to estimate the distance
% corresponding to the given RSSI reading

distance = 10.^((rssi-43)/18);
end

function plot_circle(x,y,r,identifier)
% PLOT_CIRCLE
% This function plots a circle for visualization of the main algorithm

ang=0:(2*pi)/1000:2*pi;
xp=r*cos(ang);
yp=r*sin(ang);

global CIRCLEPLOT A CIRCLEPLOTB CIRCLEPLOT C;

if identifier == 'A'
    delete(CIRCLEPLOT A);
    CIRCLEPLOT A = plot(x+xp,y+yp,'m');
elseif identifier == 'B';
    delete(CIRCLEPLOTB);
    CIRCLEPLOTB = plot(x+xp,y+yp,'b');
elseif identifier == 'C'
    delete(CIRCLEPLOT C);
    CIRCLEPLOT C = plot(x+xp,y+yp,'g');
end
hold on
end

function midpoint = determine_midpoint(A,B,C)
% This function determines the midpoint of the formation and plots it
% as a black dot

dist_ab = sqrt((A(1,3)-B(1,3))^2+(A(2,3)-B(2,3))^2);
y = (1/sqrt(3))*(dist_ab/2);
midpoint = [A(1,3)+(dist_ab/2);A(2,3)+y;1];

plot(midpoint(1),midpoint(2),'k.')
end

function yaw = determine_yaw(midpoint,point)
% This function determines the rotation angle corresponding to the
% midpoint of the formation (first argument) and the point
% generated from the LOCATE_TARGET
% algorithm (or any (x,y) point as the second argument).

x = midpoint(1);
y = midpoint(2);
a = point(1);
b = point(2);

if x == a
    if y > b
        yaw = 270;
    else
        yaw = 90;
    end
elseif b == y

```

```

        if x > a
            yaw = 180;
        else
            yaw = 0;
        end
    else
        yaw = atand(abs((y-b)/(x-a)));
        if a<x && b>y
            yaw = (90-yaw)*2 + yaw;
        elseif a<x && b<y
            yaw = 180 + yaw;
        elseif a>x && b<y
            yaw = 360 - yaw;
        end
    end
end
end

```

```

function advance(CA,CB,CC,CT,x)
% This function predicts the target location using the
% locate target function and advances the midpoint of the formation
% towards the predicted location

global A B C rssi_a rssi_b rssi_c;

point = locate_target(CA,CB,CC,CT);

mid = determine_midpoint(CA,CB,CC);
yaw = determine_yaw(mid,point);

A = set_rotation(CA, yaw);
B = set_rotation(CB, yaw);
C = set_rotation(CC, yaw);

A = local_translation(A, 'A', x, 0);
B = local_translation(B, 'B', x, 0);
C = local_translation(C, 'C', x, 0);
end

```

Figure 9 shows a visualization of each step of the formation. The formation is reduced in graph 3 and in graph 7.

A collection of different starting points for the formation and for the target is presented in Figure 10.

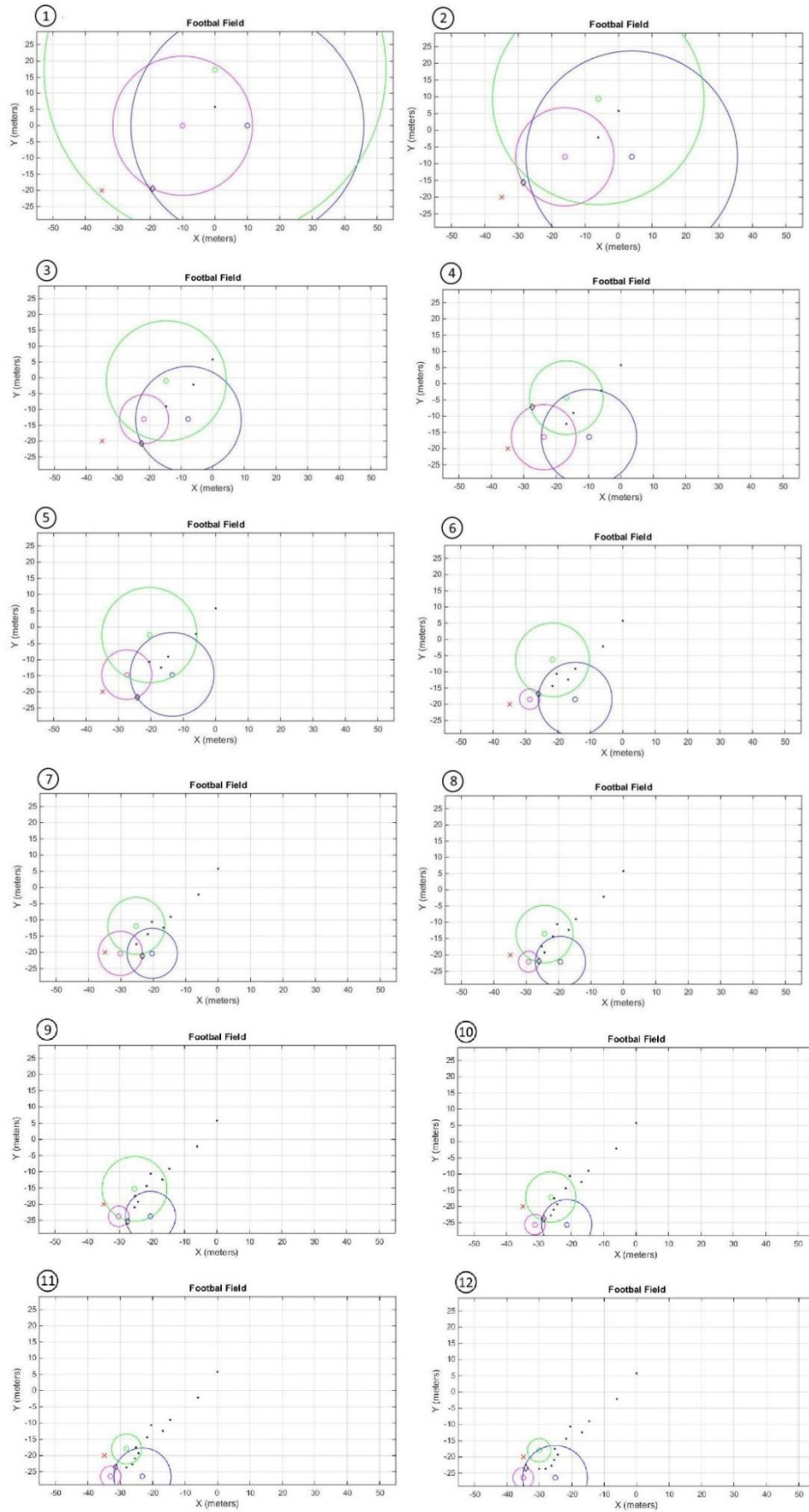


Figure 9: The graph is redrawn each time the formation advances a step.

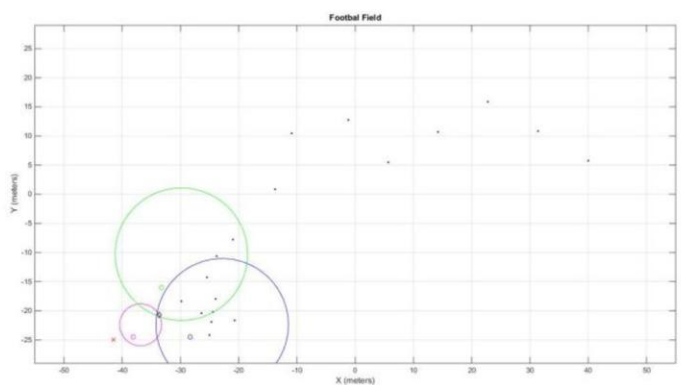
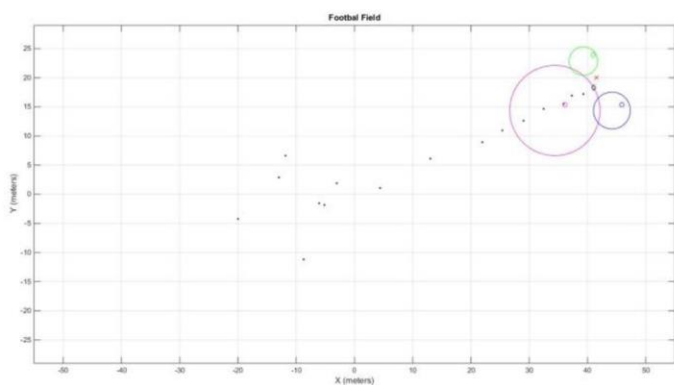
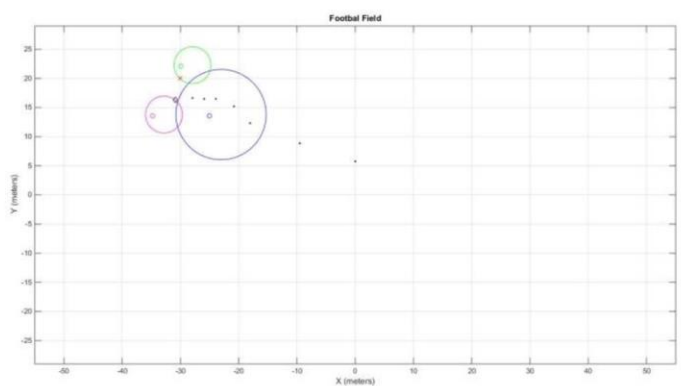
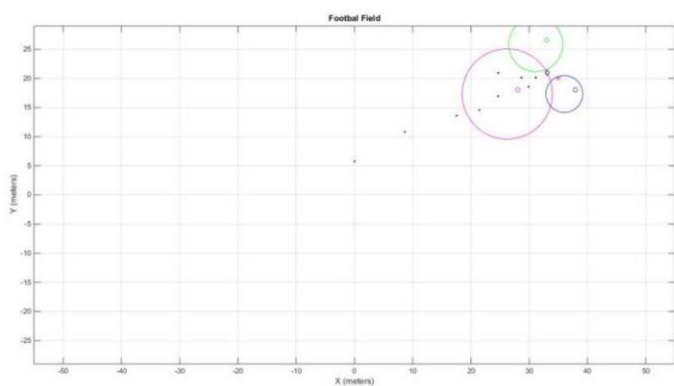
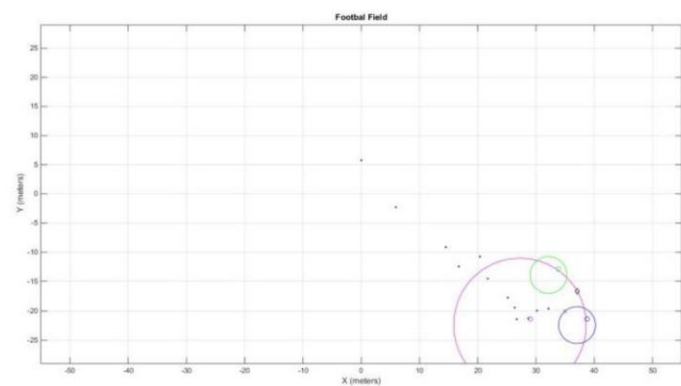
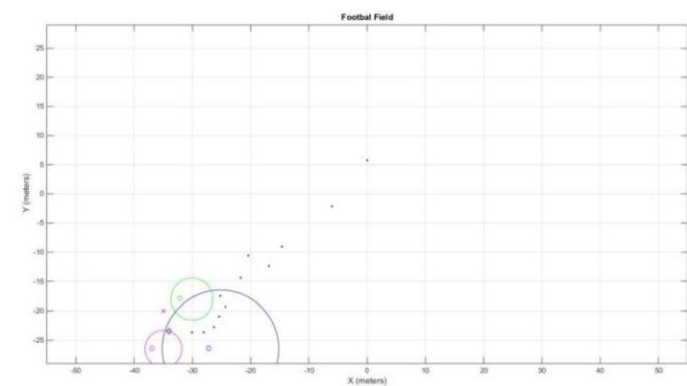


Figure 10: Successful simulations with different starting locations.

Appendix F - Test Results

MATLAB CODE:

```
function initialize() % Run initialize function to start simulation

% Decalare and initailize copter formation
global A B C;

% Give the formation an initial referance frame
A = set_se2(-10,0,0);
B = set_se2(10,0,0);
C = set_se2(0,sqrt(20^2-10^2),0);

%Locate and converge upon target
EXIT_FLAG = 0;
within = Inf;
while 1
    get_rssi();
    x = 10;
    within = is_D_m_from_target(within);
    if within == 10
        within = 10;
        reduce_formation_size(A,B,C);
        advance(A,B,C,x)
        while 1
            get_rssi()
            x = 4;
            within = is_D_m_from_target(within);
            if within == 5
                within = 5;
                reduce_formation_size(A,B,C);
                advance(A,B,C,x)
                while 1
                    get_rssi()
                    x = 2;
                    within = is_D_m_from_target(within);
                    if within == 3
                        EXIT_FLAG = 1;
                        break;
                    end
                    if EXIT_FLAG, break, end
                    advance(A,B,C,x)
                end
            end
            if EXIT_FLAG, break, end
            advance(A,B,C,x)
        end
    end
    if EXIT_FLAG, break, end
```

```

    advance(A,B,C,x)
end
end


---


function within = is_D_m_from_target(current)
% This function determines if any of the quad copters are "D" meters from the
% target and returns the distance.

global rssi_a rssi_b rssi_c;

within = Inf;

if current > 10
    if (rssi_a < 62)
        fprintf('Copter A is within 10 meters from the target \n')
        within = 10;
    end
    if (rssi_b < 62)
        fprintf('Copter B is within 10 meters from the target \n')
        within = 10;
    end
    if (rssi_c < 62)
        fprintf('Copter C is within 10 meters from the target \n')
        within = 10;
    end
elseif current > 5
    if (rssi_a < 56)
        fprintf('Copter A is within 5 meters from the target \n')
        within = 5;
    end
    if (rssi_b < 56)
        fprintf('Copter B is within 5 meters from the target \n')
        within = 5;
    end
    if (rssi_c < 56)
        fprintf('Copter C is within 5 meters from the target \n')
        within = 5;
    end
else
    if (rssi_a < 52)
        fprintf('Copter A is within 3 meters from the target, success! \n')
        within = 3;
    end
    if (rssi_b < 52)
        fprintf('Copter B is within 3 meters from the target, success! \n')
        within = 3;
    end
    if (rssi_c < 52)
        fprintf('Copter C is within 3 meters from the target, success! \n')
        within = 3;
    end
end
end
end


---


function position_global = set_se2(x, y, yaw)
% SET_SE2
% This function initializes a copter using a Special Euclidian
% Transformation Matrix in two-dimensional space. Easily updatable

```



```

% Two three dimensions

position_global = [cosd(yaw) -sind(yaw) x; sind(yaw) cosd(yaw) y; 0 0 1];
end

function get_rssi()
% This function propts the user for the RSSI value

global rssi_a rssi_b rssi_c;
prompt = 'RSSI A: ';
rssi_a = input(prompt);
prompt = 'RSSI B: ';
rssi_b = input(prompt);
prompt = 'RSSI C: ';
rssi_c = input(prompt);
end

function reduce_formation_size(CA,CB,CC)
% This function reduces the formation size by a factor of 15%

global A B C rssi_a rssi_b rssi_c;

A = set_rotation(A, 0);
B = set_rotation(B, 180);
C = set_rotation(C, 270);

percentage = 0.15;

factor = 1 - (percentage*2);
dist_ab = sqrt((CA(1,3)-CB(1,3))^2+(CA(2,3)-CB(2,3))^2);
dist = dist_ab*percentage;
distc = abs(CA(2,3)-CC(2,3)) - sqrt((factor*dist_ab)^2-(factor*...
    dist_ab/2)^2);

fprintf('Reduce Formation Size \n');

A = local_translation(A,'A',dist,0);
B = local_translation(B,'B',dist,0);
C = local_translation(C,'C',distc,0);

fprintf('Reduce Distance A: %d\n', dist);
fprintf('Reduce Distance B: %d\n', dist);
fprintf('Reduce Distance C: %d\n', distc);
end

function new_position_global = set_rotation(position_global,yaw)
% This function sets the rotation of the copter that is passed in as an
% argument in the Special Euclidian Matrix

new_position_global = [cosd(yaw) -sind(yaw)...
    position_global(1,3); sind(yaw) cosd(yaw)...
    position_global(2,3); 0 0 1];
end

function new_position_global =
local_translation(position_global,identifier,x,y)
% This function takes as its input arguments a copter, a string
% identifying which copter it is (A,B,C) and an x and y translation value.

```

```

% This function translates the copter by concatenating matrices

new_position_global = position_global*[1 0 x;0 1 y;0 0 1];
end
function point = locate_target(A,B,C)
% This is the main running algorithm for locating the target. This
% algorithm uses the rssi level readings from each copter to create
% "imaginary" circles (trilateration) and then outputs the point on copter
A's circle that
% is closest to both copter B and C's circles. This point should be close
% to the target.

global rssi_a rssi_b rssi_c;

% Convert the RSSI readings to a distance
rad_a = convert_rssi(rssi_a);
rad_b = convert_rssi(rssi_b);
rad_c = convert_rssi(rssi_c);

% Create 1001 plot points
angle=0:(2*pi)/1000:2*pi;

% Initialize the imaginary circles in a 2x10001 matrix
% Each column corresponds to a point (x,y) on the circle
AR = [rad_a*cos(angle)+A(1,3);rad_a*sin(angle)+A(2,3)];
BR = [rad_b*cos(angle)+B(1,3);rad_b*sin(angle)+B(2,3)];
CR = [rad_c*cos(angle)+C(1,3);rad_c*sin(angle)+C(2,3)];

% Initialize a large distance for error checking
distance = 1000;

% Iterate around every circle and check if the total distance between
% points is a minimum
for i = 1:7:1001
    for j = 1:7:1001
        for k = 1:7:1001
            new_distance = sqrt((AR(1,i)-BR(1,j))^2+...
                (AR(2,i)-BR(2,j))^2) + sqrt((BR(1,j)-CR(1,k))^2+...
                (BR(2,j)-CR(2,k))^2) + sqrt((AR(1,i)-CR(1,k))^2+...
                (AR(2,i)-CR(2,k))^2);
            if new_distance < distance
                distance = new_distance;
                point = [AR(1,i);AR(2,i)];
            end
        end
    end
end

if distance == 1000
    fprintf('Too much error, optimize and re-try\n');
    return
end
end
function distance = convert_rssi(rssi)
% This function uses the lod-distance equation to estimate the distance

```

```

% cooresponding to the given RSSI reading

distance = 10.^((rssi-43)/18);
end

function midpoint = determine_midpoint(A,B,C)
% This function determines the midpoint of the formation

dist_ab = sqrt((A(1,3)-B(1,3))^2+(A(2,3)-B(2,3))^2);
y = (1/sqrt(3))*(dist_ab/2);
midpoint = [A(1,3)+(dist_ab/2);A(2,3)+y;1];
end

function yaw = determine_yaw(midpoint,point)
% This function determines the rotation angle corresponding to the
% midpoint of the formation (first argument) and the point
% generated from the LOCATE_TARGET
% algorithm (or any (x,y) point as the second argument).

x = midpoint(1);
y = midpoint(2);
a = point(1);
b = point(2);

if x == a
    if y > b
        yaw = 270;
    else
        yaw = 90;
    end
elseif b == y
    if x > a
        yaw = 180;
    else
        yaw = 0;
    end
else
    yaw = atand(abs((y-b)/(x-a)));
    if a<x && b>y
        yaw = (90-yaw)*2 + yaw;
    elseif a<x && b<y
        yaw = 180 + yaw;
    elseif a>x && b<y
        yaw = 360 - yaw;
    end
end
end
end

function advance(CA,CB,CC,x)
% This function predicts the target location using the
% locate target function and advances the midpoint of the formation
% towards the predicted location

global A B C rssi_a rssi_b rssi_c;

old_a = A;
old_b = B;
old_c = C;

```

```

point = locate_target(CA,CB,CC);

mid = determine_midpoint(CA,CB,CC);
yaw = determine_yaw(mid,point);

A = set_rotation(CA, yaw);
B = set_rotation(CB, yaw);
C = set_rotation(CC, yaw);

A = local_translation(A, 'A', x, 0);
B = local_translation(B, 'B', x, 0);
C = local_translation(C, 'C', x, 0);

hyp_a = sqrt((old_a(1,3)-A(1,3))^2+(old_a(2,3)-A(2,3))^2);
AX = cosd(yaw)*hyp_a;
AY = sind(yaw)*hyp_a;
fprintf('Translate Locations: (%d,%d)\n', AX, AY);
end

```

MATLAB console input/output:

```

>> initialize
RSSI A: 64
RSSI B: 68
RSSI C: 69
Translate Locations: (-4.195137e+00,-9.077490e+00)
RSSI A: 66
RSSI B: 70
RSSI C: 67
Translate Locations: (-9.111290e+00,4.121212e+00)
RSSI A: 67
RSSI B: 63
RSSI C: 63
Translate Locations: (8.614768e+00,5.077969e+00)
RSSI A: 63
RSSI B: 68
RSSI C: 67
Translate Locations: (-9.902633e+00,1.392073e+00)
RSSI A: 64
RSSI B: 65
RSSI C: 66
Translate Locations: (-6.166934e-01,-9.980966e+00)
RSSI A: 53
RSSI B: 61
RSSI C: 62
Copter A is within 10 meters from the target
Copter B is within 10 meters from the target
Reduce Formation Size

```

Reduce Distance A: 3.000000e+00
Reduce Distance B: 3.000000e+00
Reduce Distance C: 5.196152e+00
Translate Locations: (-7.510184e+00,-6.602813e+00)
RSSI A: 44
RSSI B: 53
RSSI C: 56
Copter A is within 5 meters from the target
Copter B is within 5 meters from the target
Reduce Formation Size
Reduce Distance A: 2.100000e+00
Reduce Distance B: 2.100000e+00
Reduce Distance C: 3.637307e+00
Translate Locations: (-3.398450e+00,-2.109630e+00)
RSSI A: 49
RSSI B: 48
RSSI C: 39
Copter A is within 3 meters from the target, success!
Copter B is within 3 meters from the target, success!
Copter C is within 3 meters from the target, success!