Drone Assisted Energy Delivery

FINAL REPORT

Team Number: 40
Advisers: Randall Geiger & Degeng Chen

Team Members/Roles:
Garrett Lies: Embedded Systems - Drone Automation
Drew Underwood: Embedded Systems - Drone Automation
Abdullah Al-Obaidi: Power Systems - Image Processing
Khalifa Aldaheri: Power Systems - Drone Automation
Ahmed Al-Hulayel: Control Systems - Energy Delivery/Landing Station

Team Email: sdmay18-40@iastate.edu
Team Website: http://sdmay18-40.sd.ece.iastate.edu

Revised: April 23, 2018

Table of Contents

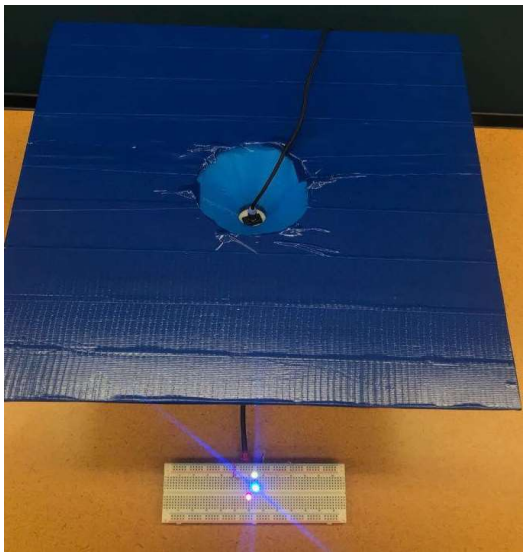
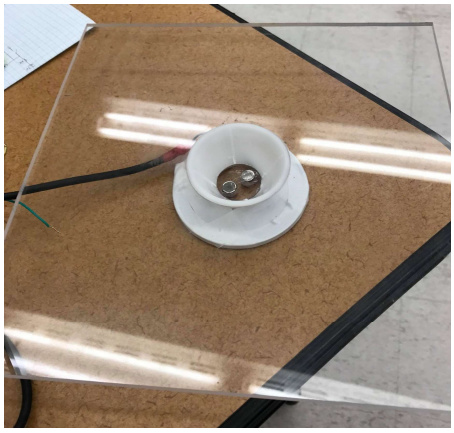


Figure 2. Old Landing Station



1 Introduction

1.1 Acknowledgement

We would like to thank our two advisors Dr. Geiger and Dr. Chen as they have been a huge help throughout the development of this project. Also thank you to our peers which have given us feedback throughout the past year, and to ETG for helping us with our hardware.

1.2 Problem and Project Statement

There has been an increasing trend in the use of IOT (Internet of Things) devices and other embedded systems. It may not be feasible to connect these devices to the power grid, and alternative methods may not be available. To bring energy to these devices, a new method of delivering energy may be required.

Our team is working on solving this problem through the use of drone assisted energy delivery. In the near future it is possible that a large network of drones will be used to delivery energy to surrounding devices. We will begin working on a simple proof of concept to bring this idea to life. To accomplish this task we will need to design a landing station for a drone to establish connection to the device being charged, and we will need to fully automate the process of the drone flying from one location to the landing station, charging the device, and returning to its home location.

1.3 Operational Environment

Ideally, the drone should be able to navigate through dynamic environments and various weather conditions. A final product might use smaller more inexpensive drones, so that if one is destroyed, the financial loss is minimized. We are focusing on a proof of concept, and therefore will not be making any requirements about the environment. It is assumed that the drone should operate in an ideal environment with no obstacles.

1.4 Intended Users and uses

Two broad types of users are intended for the use of this product. The first user type is a typical consumer. A consumer will have the ability to call a drone for energy delivery at any given time. The drone then flies to the consumer, charges their electronic device, and leaves.

A second type of user is Industry. Industry users will be wanting to delivery energy to many stationary and non-stationary devices. An industry user may want additional customizable options to allow more efficient energy delivery, and delivery on a large scale. Bringing energy delivery to an Industry user will require a large network of drones all working together.

1.5 Assumptions and Limitations

This product has no scaling limitations. The end product will be flexible, allowing for large networks, and various sizes of drones. Power stations for recharging the drone will be assumed to have connection into the power grid, or some other means of gathering energy. Each device needing to be charged will require a nearby docking station for the drone to land on while delivery energy. Due to drone regulations, the use of drone automated energy delivery must follow all of these regulations. For example, these drones cannot fly within five miles of an airfield.

Assumptions:

1. **The internet of things will expand rapidly:** Our project is built on the idea that the rapid expansion of IoT will continue and nodes will be placed in locations far away from the power grid.
2. **Autonomous Drone flights will be legal:** We expect that the US government will remove the ban on autonomous drone flights in the near future.
3. **Drone prices decline:** Drone technologies is improving continuously which is leading to prices dropping steadily. Cheapness of drones makes our project economically feasible.

4. **Collision avoidance implemented:** As stated in the environment section we will assume drones will be able to avoid collisions in a dynamic environment

5. **Drone will be charged at home location:** Drone being used in final product should have a simple way of charging itself at the home station.

Limitations:

1. **Weight carrying ability:** Our drone can carry a maximum of two extra lbs. This limits the weight of the additional charging battery the drone can carry, and therefore the amount of charge which can be delivered.

1.6 Expected End Product and Deliverables

The final end product will consist of a drone which can automatically fly from one location to another. The drone will be able to land, charge a device, and return to the original location. The charged device will continue to be powered after the drone leaves through the use of a battery. In order to accomplish this, a custom docking station will be built to allow the delivery of energy to the drone and electronic device. Lastly, a proposal will be made to guide the next group through a list of tasks which will need to be done next.

Drone automation: Our drone will need to be able to fly from one location to another, charge a device, and return home. To accomplish this, we will automate the takeoff of the drone, flying to the location with GPS, using image processing to accurately move above the landing station, land the drone, disarm the motors, and then return home after giving the drone time to charge the device.

Docking station: The docking station is necessary for the drone to land at the target device and make a reliable connection to delivery energy. We will need to create a landing station which the drone can accurately land on, while also establishing a solid connection between the drone and the device being charged. This may require several prototypes and exploration into various methods for establishing a reliable connection between these two devices.

2. Specifications and Analysis

2.1 Proposed Design

Drone Energy Delivery project:

Our project is divided into two categories, programming the drone to go from point A to B, and charging the device. As a result of knowing what our focuses and capabilities are, we have divided ourselves into two teams.

Drew and Garrett are the two primarily working on the automation of the drone. They are responsible for automating the flight, and figuring out how to land the drone accurately. In order to accomplish this task they will be working closely with the other team to ensure the solution will work properly.

The other team, composed of Ahmed, Abdullah, and Khalifa, is responsible of figuring out a way to deliver the energy from the charging portable to the ioT devices. This team will have to figure out a way to connect reliably to the device. This can be done through any method, but they must also ensure that the drone is able to disconnect from the device as well.

Together, our two teams will be working on a solution to deliver energy through drone assisted energy delivery by presenting the following deliverables: Fully automated system for flying the drone from point A to point B, landing, taking off, and returning home. A way to reliably connect to a device in order to deliver energy. And a landing station which allows the drone to locate and connect to the device. For the demonstration of this project, we do not require a success rate of 100%. As long as the drone is able to carry out the mission some percentage of the time, then that is acceptable.

2.2 Design Analysis

Our team has made a lot of progress on both the landing station and the automation since first beginning this project. We stuck with two separate teams for a

while, but as time went on we had to move more members to automation. We were able to finish the landing station, but the automation is not quite there yet.

The landing station has gone through several different prototypes. We explored the possibility of using wireless charging, which we decided was not worth the energy loss. Our first prototype used multiple conducting magnets to create a connection using ground as one polarity of the magnet and the input as the other. This worked out fairly well, but the magnets were too strong for the drone to take off after connection, even with really small magnets. We instead decided to use a USB charge with a magnetic connector which is much weaker (Figure 2).

In order to help the drone land on the landing station we have included a larger landing platform, and a cone to act as a funnel for directing the wire connection coming from the drone to the device to be charged. This way the drone does not need to land directly on the target in order to establish a connection. The funnel will guide the wire, and the magnets will ensure a connection is made. We have also used blue tape to cover the base of the landing station. This is used by the image processing to locate the geometric center of the base by filtering for color (Figure 1).

For automation we decided to mainly focus using the Intel recommended library of Dronekit. Dronekit is python library which abstracts the mavlink messages being sent from the compute board to the flight controller. We are also using OpenCV library in python to help us with the image processing. One benefit of using Dronekit is that both Dronekit and OpenCV utilize python, so communication between the two is easy.

Before we could do anything with the drone we had to go through a long process of research and installation. We first had to flash the drone to the latest version of the operating system which was a custom Yocto installation. We then had to flash several other hardware related pieces such as the flight controller. After this was complete, we were required to properly calibrate the drone to ensure stable flight. Into our second semester, Intel released a version of Ubuntu compatible with the drone which we later flashed to in order to use the camera libraries required for image processing. Each of these stages had their own difficulties and troubleshooting.

Once the system was ready, we were able to begin coding. Automation was a steep learning curve as no one on our team had previous experience with drones or embedded systems outside of what was learned in required courses. With a lot of trial and error we were able to make significant progress on the automation. We began with simple tests such as arming and disarming the motors. After these were working, we moved onto taking off and landing, movement by GPS waypoints, and explored the possibility of movement through velocity settings, as well as yaw pitch and roll settings. Weather proved to be problematic as we found out many of these tests required GPS in order to run, otherwise the drone has no sense of location and refused to carry out commands.

We found out towards the end of our project that the Dronekit library was not fully supported for the PX4 architecture that our drone was using. This severally limited our control over the drone, but we were able to find clever workarounds to our problems. We spent a large amount of time trying to solve these issues as it was too late to learn the alternative libraries available to us, which also did not interface with the image processing. As more issues became apparent, it seemed that we would not be able to accomplish the task due to the dronekit limitations as well as flight stability issues with the drone.

Abdullah ended up taking over the image processing to ease the burden of the automation problems. He was able to successfully locate the landing station by filtering images for objects of a certain size and color. We also came up with an algorithm which took advantage of our knowledge of the landing station base dimensions to determine how far away the drone was from the target. All of the image processing had been completed, tested, and found working. All the image processing coding was done in Python through a 3rd party library called OpenCv.

We assigned Ahmed to work on the Landing/charging station and implement a new and more efficient design by which the wire connection will be ensured. Ahmed started working on a new idea, which is using the USB wire consisted of magnet. Ahmed has started learning how to use SolidWorks in order to make a new design for the base by creating a cone, cylindrical spare with hole in the middle to place the USB male to female, and a solid base to stabilize it keeping in mind that we need to minimize the budget for the parts. Then have assembled the parts together and made multiple simple tests to

ensure that when the drone lands, it will establish a reliable connection. So, to do this, Ahmed and Abdullah had to pick a color that the camera can reorganize easily in the image processing. As a result, we had to make the landing station's color to be Blue colored. We tested how the camera picks the blue color easily and then find the center point, it all worked well.

3 Testing and Implementation

We took an incremental approach to testing as simulation software was often unreliable for our libraries. This involved running many tests to try and gain an understanding of the different degrees of freedom in our system. We were then able to combine our understanding into more complex tests to carry out the automation tasks.

Testing for the landing station was far simpler. All we needed to do here, was ensure that a connection was being made by using a simple LED circuit on a breadboard. We then had to test the probability that the drone would be able to hit the target in order to make the connection, and whether or not the drone could take off once charging was complete.

1. Preliminary testing
    a. Prep for Takeoff
        i. Battery charging and attachment
        ii. Stable power supplied for drone, safe connection
        iii. Get familiar with remote control, prior to attaching propellers ensure all systems are stable before any flight takes place.
    b. Flight Testing
        i. Attach propellers and begin simple testing
        ii. Indoor Takeoff/land within a few feet
2. Communication between ground and drone
    a. QGroundControl
        i. Connect to our drone
        ii. Understand proper instruction delivery to drone

iii. Repeat preliminary flight tests with automated instruction

b. Begin Mission testing
    i. Approach automated path guidance via pre-programmed instruction
    ii. Drone will takeoff, reach destination, and then return home

c. Conclude Communications and Preliminary testing
    i. Drone can safely takeoff/land
    ii. Drone can reach destination and return home

3. Begin sensory testing
    a. Real-Sense camera
        i. Have a thorough understanding of Real-Sense library and how to relay proper instruction to drone when necessary
        ii. Repeat preliminary flight testing with Real-Sense data flow
        iii. Attempt non-hazardous obstacle detection
            1. Slowly approach a wall
            2. Approach a person
        iv. Ensure data feed is reliable and accurate
        v. Begin testing of obstacle detection, approach specified target
    b. Height, Height of Interest
        i. Establish accuracy of height detection
        ii. Find limits of height detection within reasonable boundary
        iii. Understand how to utilize height of interest for obstacle avoidance and general landing precautions
    c. Conclude Sensory Testing
        i. Drone can now safely approach a destination with real-time input
        ii. Sensory data feed is accurate, reliable, and understood
4. Battery Attachment
    a. Separate battery attached to drone
        i. Reliably connected
        ii. Does not hinder drone performance
        iii. Make adjustments to modified battery life

1. Fixed battery will bring us close to drone's max flight weight

iv. Repeat preliminary testing with attached battery

b. Battery fixed connectors

i. Battery needs to be adjusted in such a way that it can attach to a specified target once the drone arrives

ii. Modify Height of interest and "box" around the drone for in flight calculations if necessary

iii. Repeat preliminary testing with battery connectors

c. Energy Deliverance

i. Create appropriate destination for energy delivery

ii. Repeat drone sensory testing on our created destination for delivery

iii. Establish accuracy of drone automated guidance. Make adjustments if necessary

d. Conclude Battery Attachment and energy delivery testing

i. Drone can reach destination accurately with sensory input

ii. Drone and fixed battery can safely land and attach to specified charging target

iii. Drone can safely remove itself from charging target and return home

5. (Best Case Scenario) Complete Energy Delivery Automation, proceed to obstacle avoidance

a. Repeat Preliminary testing with obstacle detection and avoidance on fixed paths

b. Proceed to moving obstacle detection

c. Conclude obstacle avoidance

3.1 Interface Specifications

Software specifications:

1. QGroundControl - Standard UAV communication tool which allows us to connect to our drone and issue flight instructions

2. MavLINK Protocol - Standard air to ground communication protocol for relaying reliable and secure information

3.2 HARDWARE AND SOFTWARE

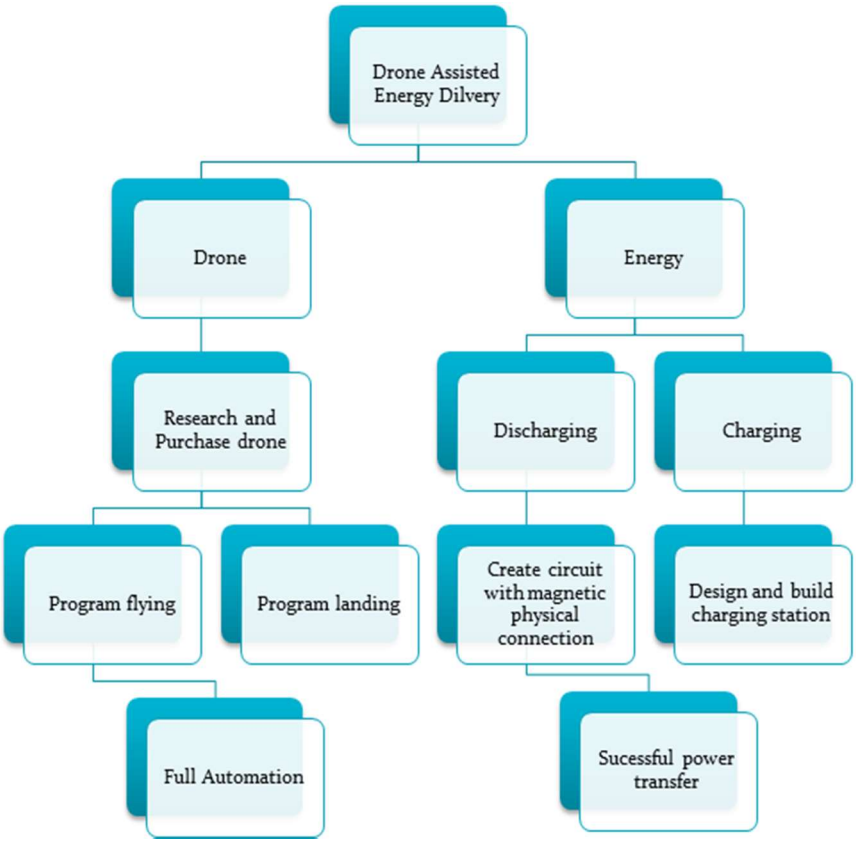
Software:

1. RealSense and OpenCV library - RealSense camera on the drone using OpenCV for image processing to detect landing station
2. Dronekit - Library containing functions which abstract the process of sending mavlink messages
3. Python - All automation and image processing done in python

Hardware:

1. LiPo Battery - Relatively compact and powerful, can be extremely dangerous if not handled properly and should be used responsibly
2. Intel Compute Board - brains behind the drone, sends mavlink commands to the flight controller, handles most of the processing load
3. USB Standard - Used USB connectors throughout the project as a way of transferring energy to a device

3.3 Process



Above is a diagram showing how the work was broken up for this project. This diagram represents our initial plan, but has been changed slightly throughout our project lifetime. One change that was mage was the inclusion of image processing as the GPS was not accurate enough to bring us to our target. There are two clearly defined branches, the drone and energy branches which various members focused on different branches.

3.4 Results

We were able to create the landing station without many issues, and are happy with the final product. The landing station went through several iterations, with the final design being easily disassembled if necessary. No issues were found in the final design, although the base could have been a little bit larger to prevent the drone from tipping the landing station.

The image processing software can filter an object of a predefined size and color, and find its location in 3D space. The location accuracy is accurate within 1-2 cm; the (0,0,0) location is defined to be the camera location, and the object we are filtering for is defined in xyz coordinates in cm units.

Automation has come a long way but is not quite finished. We ran into several issues, and hit a major roadblock towards the end of the project when it was discovered that Dronekit could not implement the task properly. This was due to an unknown issue with the Dronekit library and the PX4 architecture. This limited our ability to control the drone to only GPS waypoints which were not accurate enough to move the drone directly over the landing station. We planned a work around which would mimic positional movements by canceling GPS waypoint commands during mid flight. Unfortunately, we found out that the process of canceling waypoint commands is not responsive enough and can be delayed by several seconds. This was too much error to move the drone over the landing station.

We also ran into flight stability issues which several other users of our drone have mentioned to Intel. During flight, our drone is constantly drifting off despite the calibration being correct. We were unsure if this was due to an error in the controls, sensors, or balance. This problem was out of our control as we have no access to the control systems which ensure mavlink commands are carried out successfully.

Dronekit is currently working on full compatibility with the PX4 architecture, but the technology is not there yet. We explored several solutions of accomplishing our task, but without the full compatibility it appears to be impossible to complete successfully. We still created a finished program which works assuming the stability issues have been fixed. If we had to repeat this project again, we would have learned the MAVROS library which would hopefully give us the control and stability that we require.

4 Closing Material

4.1 Conclusion

We were not able to meet all the requirements set out by our advisor, but we have made significant progress. Our team has been working hard for the past year to solve all of the issues we have encountered, and managed to create a system which is just shy of working. Unfortunately our automation hit a brick wall towards the end as limitations became more visible, and we encountered issues that were beyond our control. It is quite possible that in a year or two, when Dronekit releases full support for PX4 that our current tests will actually accomplish the task.

4.2 References

[1]"DroneKit by 3D Robotics", *DroneKit*, 2017. [Online]. Available: http://dronekit.io/. [Accessed: 05- Dec- 2017].

[2]"intel-aero/meta-intel-aero", *GitHub*, 2017. [Online]. Available: https://github.com/intel-aero/meta-intel-aero/wiki. [Accessed: 05- Dec- 2017].

4.3 Appendices

I. Operation Manual

To carry out the final demonstration of our project, or any test for that matter, there are several steps which need to be followed. These include the proper charging of the battery, connecting via ssh to the drone, space requirements, preparation, locating and launching the program, and cancellation of the program if anything goes wrong.

1. Proper Charging of the Battery

The battery for our drone is an external LiPo battery which contains multiple cells. This battery needs to be fully charged and balanced before use, otherwise a fire or explosion could occur. Operating when the battery is below 75% voltage may cause permanent damage, the drone should not be run for more than 20 minutes of flight before recharging is required.

In order to charge the battery, connect the LiPo battery to the charger while plugged into an outlet. Make sure the battery is disconnected from the drone before charging. The charger contains an LED light to indicate the progress of charging. A solid red light means the battery is charging, an alternating red and green light means the battery is balancing, and a solid green light signals the battery is ready to be used. If the light flashes red, then something is wrong and the battery should be removed. During charging, the battery should start as solid red, move to alternating red and green, and then to solid green.

2. Connecting via SSH

Once the battery is fully charged, the drone can be turned on by pressing the power button once. To turn off the drone, hold the power button until the power light flashes. In order to launch the program from a laptop, a connection to the drone must be established. While on the same wifi as the drone (IASTATE), a connection can be made by SSH with the following address: sdmay18-40-drone.ece.iastate.edu username: sdmay1840 password: aadgk1840

3. Preparation

Several preparations must be made before running any code. Some tests can be run inside without the propellers attached, but if flight is intended, the drone must be outside in an open area clear of nearby people and obstacles. The propellers can then be attached by matching the letter on the propeller to the letter on the drone. A tether is recommended to be attached to the drone incase something goes wrong. In addition, the

controller should be on and connected to the drone in order to take over if anything happens. This can be done by turning on the controller while holding down the bind button. A beep will indicate that the controller is connected successfully. It may help to try and use the controller to ensure that it is connected before running any programs.

4. Launching the Program

In order to launch the program, locate the file by accessing the Desktop directory, sdmay1840 folder, and the our tests folder. There are several other directories containing all of our tests. The final test is located in the final tests directory and named test6_final.py. This program can be launched by typing the command python test6_final.py into the terminal.

5. Cancellation

In the event that something goes wrong during the test, the program can be canceled by hitting ctrl C in the terminal. This will cause the program to stop, with the drone maintaining the state that it was last in. From this point on, control can be passed over to the remote controller.

II. Alternative Designs

Throughout the progress of our project we have came up with several alternative designs. The landing station initially was designed to use conducting magnets as a way to move current to the charging device. We found that even for the smallest magnets we could acquire, the drone was not able to overcome the force of the magnets during takeoff.

We also explored several different solutions for automation. We first planned on using the Dronekit library to complete our project, but towards the end of the project we also expanded into using mavros. Mavros was too complicated and low level to get a good enough grasp on within the limited time frame, and so we did not pursue it too far. We also came up with, implemented, and tested several different ways of accomplishing the automation task in Dronekit.

The final method we were forced into was the use of GPS waypoints. We also explored the possibility of controlling the drone through velocity commands as well as yaw pitch and roll. Through our testing we found that the drone had no response to the commands. With further research it was discovered that these commands were not supported on our drone's architecture.

III. Other Considerations

I hope that it is clear that our team has worked extremely hard on this project from start to finish. We have put in many hours, and solved many unseen issues. Although the project did not meet the requirements given by our advisors, we do believe that we have done the best job that we could have given our choice in drone and libraries. If given another chance, we would have explored an entirely new approach with our newfound knowledge. We would also likely work with a drone such as DJI which we know has a guaranteed working library.

IV. Code

All of the code can be found on our senior design team's github, but we will also include the final demonstration code as well.

```
################################################################################
##################
# For this test the drone should fly 5 meters up, 10 meters north, 10 meters east (Target location)
# The drone will then look for the landing station using Abdullas image processing
# The drone will then take several small movements until it is above the landing station
# the drone will then land and disarm
# after 20 seconds, the drone will rearm and return home
################################################################################
##################
# https://github.com/PX4/Firmware/blob/master/Tools/mavlink_px4.py

# Import DroneKit-Python
from dronekit import connect, Command, LocationGlobal
from pymavlink import mavutil
```

```
# Image Processing
import cv2
import numpy as np
import imutils
import math


def create_mask(color, image) :
        #green mask
        green_min = np.array([45,50,50])
        green_max=np.array([75,255,255])

        #blue mask
        blue_min = np.array([100,100,100])
        blue_max=np.array([120,255,255])

        if color == "blue":
                mask_min = blue_min
                mask_max = blue_max
        elif color == "green":
                mask_min = green_min
                mask_max = green_max

        hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
        mask = cv2.inRange(hsv, mask_min, mask_max)
    masked_image = cv2.bitwise_and(image , image , mask = mask)
        return masked_image

def find_centroid (mask, original, obj_width, focalLength):
            gray = cv2.cvtColor(mask, cv2.COLOR_BGR2GRAY)
            blurred = cv2.GaussianBlur(gray, (5,5), 0)
            thresh = cv2.threshold(blurred, 60, 255, cv2.THRESH_BINARY)[1]
            cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
            cnts = cnts[0] if imutils.is_cv2() else cnts[1]
            for c in cnts:
                    area = cv2.contourArea(c)
                    marker = cv2.minAreaRect(c)
                    dis = find_distance(obj_width, focalLength, marker[1][0])
                    M = cv2.moments(c)
                cX = int(M["m10"]/(M["m00"]+0.0001))
                cY = int(M["m01"]/(M["m00"]+0.0001))
                    x,y = find_pos_diff(cX,cY, 320,240, focalLength, dis)
                    if area > 500 :
```

```
                    if (300 < cX < 350) & (220 < cY < 260) :
                            cv2.rectangle(original, (0,0) , (30,30) , (0,255,0), -1)

                    cv2.drawContours(original, [c], -1, (0,255,0), 2)
                    cv2.circle(original, (cX, cY), 7 , (255,255,255), -1)
                    cv2.line(original, (cX, cY), (320,240), (0,0,255), 2)
                    cv2.putText(original, "Target location : (%.2f cm, %.2f cm, %.2f cm) " % (x,y,dis),
(0,50), 1, 1, (0,0,255))
            return mask, original

def find_pos_diff (cX, cY, dX, dY, focal_length, distance):
        pX = cX - dX
        pY = dY - cY
        x = (pX * distance)/focal_length
        y = (pY * distance)/focal_length
        return x,y

def find_distance(fixed_width, focalLength, perWidth ):
        return(fixed_width * focalLength)/ (perWidth +0.0001)



# Basic code
import time, sys, argparse, math


################################################################################
##################
# Settings
################################################################################
##################

connection_string       = 'tcp:127.0.0.1:5760'
MAV_MODE_AUTO   = 4

################################################################################
##################
# Init
################################################################################
##################

# Connect to the Vehicle
print "Connecting"
vehicle = connect(connection_string, wait_ready=False)

def PX4setMode(mavMode):
    vehicle._master.mav.command_long_send(vehicle._master.target_system,
vehicle._master.target_component,
                                mavutil.mavlink.MAV_CMD_DO_SET_MODE, 0,
                                mavMode,
                                0, 0, 0, 0, 0, 0)
```

```
def get_location_offset_meters(original_location, dNorth, dEast, alt):
    """
    Returns a LocationGlobal object containing the latitude/longitude `dNorth` and `dEast` metres from the
    specified `original_location`. The returned Location adds the entered `alt` value to the altitude of the `original_location`.
    The function is useful when you want to move the vehicle around specifying locations relative to
    the current vehicle position.
    The algorithm is relatively accurate over small distances (10m within 1km) except close to the poles.
    For more information see:
    http://gis.stackexchange.com/questions/2951/algorithm-for-offsetting-a-latitude-longitude-by-some-amount-of-meters
    """
    earth_radius=6378137.0 #Radius of "spherical" earth
    #Coordinate offsets in radians
    dLat = dNorth/earth_radius
    dLon = dEast/(earth_radius*math.cos(math.pi*original_location.lat/180))

    #New position in decimal degrees
    newlat = original_location.lat + (dLat * 180/math.pi)
    newlon = original_location.lon + (dLon * 180/math.pi)
    return LocationGlobal(newlat, newlon,original_location.alt+alt)




################################################################################
##################
# Listeners
################################################################################
##################

home_position_set = False

#Create a message listener for home position fix
@vehicle.on_message('HOME_POSITION')
def listener(self, name, home_position):
    global home_position_set
    home_position_set = True



################################################################################
##################
# Start mission example
################################################################################
```

```
##################

# wait for a home position lock
# check test2_home_location.py for alternative way to get home position
while not home_position_set:
   print "Waiting for home position..."
   time.sleep(1)

# Display basic vehicle state
print " Type: %s" % vehicle._vehicle_type
print " Armed: %s" % vehicle.armed
print " System status: %s" % vehicle.system_status.state
print " GPS: %s" % vehicle.gps_0
print " Alt: %s" % vehicle.location.global_relative_frame.alt

# Change to AUTO mode
PX4setMode(MAV_MODE_AUTO)
time.sleep(1)

# Load commands
# might give us issues
cmds = vehicle.commands
cmds.clear()

home = vehicle.location.global_relative_frame

# takeoff to 5 meters
# might need to increase this, maybe 10?
wp = get_location_offset_meters(home, 0, 0, 5);
cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_TAKEOFF, 0, 1, 0, 0, 0, 0, wp.lat, wp.lon, wp.alt)
cmds.add(cmd)

# move 10 meters north
wp = get_location_offset_meters(wp, 10, 0, 0);
cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_WAYPOINT, 0, 1, 0, 0, 0, 0, wp.lat, wp.lon, wp.alt)
cmds.add(cmd)

# move 10 meters east
wp = get_location_offset_meters(wp, 0, 10, 0);
cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_WAYPOINT, 0, 1, 0, 0, 0, 0, wp.lat, wp.lon, wp.alt)
cmds.add(cmd)

#Arm motors
while not vehicle.armed:
        print "waiting for arming, armable = %s" % vehicle.armable
        vehicle.armed = True
        time.sleep(1)
```

```
time.sleep(3)

# Upload mission
print "uploading missions"
cmds.upload()
time.sleep(1)

# monitor mission execution
nextwaypoint = vehicle.commands.next
while nextwaypoint < len(vehicle.commands):
   if vehicle.commands.next > nextwaypoint:
      display_seq = vehicle.commands.next+1
      print "Moving to waypoint %s" % display_seq
      nextwaypoint = vehicle.commands.next
   time.sleep(1)

# Should be near landing station, find center of station and move towards it
# Will need to be changed once calibrated
pos = find_location("blue")
startX = pos[0]
startY = pos[1]
xComplete = False
yComplete = False

# Desired distance
wp = get_location_offset_meters(wp, startY, startX, 0);

#loop until done
while xComplete == False && yComplete == False:
   print "Going to landing station"
   cmds.clear()
   if xComplete == False
      #Find direction to move west/east
      if startX > 0
         #Move east
         wp2 = get_location_offset_meters(wp, 0, 10, 0);
         cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_WAYPOINT, 0, 1, 0, 0, 0, 0, wp2.lat, wp2.lon, wp2.alt)
         cmds.add(cmd)
      else
         #Move west
         wp2 = get_location_offset_meters(wp, 0, -10, 0);
         cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_WAYPOINT, 0, 1, 0, 0, 0, 0, wp2.lat, wp2.lon, wp2.alt)
         cmds.add(cmd)
   if yComplete == False
          #Find direction to move north/south
      if startY > 0
         #Move north
         wp2 = get_location_offset_meters(wp, 10, 0, 0);
         cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
```

```
mavutil.mavlink.MAV_CMD_NAV_WAYPOINT, 0, 1, 0, 0, 0, 0, wp2.lat, wp2.lon, wp2.alt)
        cmds.add(cmd)
      else
        #Move south
        wp2 = get_location_offset_meters(wp, -10, 0, 0);
        cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_WAYPOINT, 0, 1, 0, 0, 0, 0, wp2.lat, wp2.lon, wp2.alt)
        cmds.add(cmd)
  #Goto desired
  cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_WAYPOINT, 0, 1, 0, 0, 0, 0, wp.lat, wp.lon, wp.alt)
  cmds.add(cmd)

  #Upload the commands
  cmds.upload()
  time.sleep(0.3)

  #go to next command
  vehicle.commands.next = vehicle.commands.next + 1
  if xComplete == False && yComplete == False
    time.sleep(0.3)
    vehicle.commands.next = vehicle.commands.next + 1

  #Check if we reached the destination
  pos2 = find_location("blue")
  if startX > 0
    if pos2[0] < 0
      xComplete = True
    else
      xComplete = False
  else
    if pos2[0] > 0
      xComplete = True
    else
      xComplete = False
  if startY > 0
    if pos2[1] < 0
      yComplete = True
    else
      yComplete = False
  else
    if pos2[1] > 0
      yComplete = True
    else
      yComplete = False

#Land
cmds.clear()
wp = get_location_offset_meters(wp, 0, 0, 5);
cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_LAND, 0, 1, 0, 0, 0, 0, wp.lat, wp.lon, wp.alt)
```

```
cmds.add(cmd)
cmds.upload()
time.sleep(1)

# wait for the vehicle to land
while vehicle.commands.next > 0:
        print "waiting for landing"
    time.sleep(1)

# Disarm vehicle
print ("disarming")
vehicle.armed = False
time.sleep(1)

#Charge for 20 seconds
time.sleep(20)

# Attempt to return home, get cmds, clear them, retrace steps, and upload
# Load commands
# might give us issues
cmds.clear()

# takeoff to 5 meters
# might need to increase this, maybe 10?
wp = get_location_offset_meters(wp, 0, 0, 5);
cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_TAKEOFF, 0, 1, 0, 0, 0, 0, wp.lat, wp.lon, wp.alt)
cmds.add(cmd)

# move 10 meters west
wp = get_location_offset_meters(wp, 0, -10, 0);
cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_WAYPOINT, 0, 1, 0, 0, 0, 0, wp.lat, wp.lon, wp.alt)
cmds.add(cmd)

# move 10 meters south
wp = get_location_offset_meters(wp, -10, 0, 0);
cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_WAYPOINT, 0, 1, 0, 0, 0, 0, wp.lat, wp.lon, wp.alt)
cmds.add(cmd)

# land
wp = get_location_offset_meters(wp, 0, 0, 5);
cmd = Command(0,0,0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_LAND, 0, 1, 0, 0, 0, 0, wp.lat, wp.lon, wp.alt)
cmds.add(cmd)

#Arm motors
while not vehicle.armed:
        print "waiting for arming, armable = %s" % vehicle.armable
        vehicle.armed = True
```

```
            time.sleep(1)

# Upload mission
print "uploading missions"
cmds.upload()
time.sleep(1)

# monitor mission execution
nextwaypoint = vehicle.commands.next
while nextwaypoint < len(vehicle.commands):
   if vehicle.commands.next > nextwaypoint:
      display_seq = vehicle.commands.next+1
      print "Moving to waypoint %s" % display_seq
      nextwaypoint = vehicle.commands.next
   time.sleep(1)

# wait for the vehicle to land
while vehicle.commands.next > 0:
            print "waiting for landing"
      time.sleep(1)

# Disarm vehicle
print "disarming"
vehicle.armed = False
time.sleep(5)

# Close vehicle object before exiting script
print "complete"
vehicle.close()
time.sleep(1)
```