Hawkeye: Unmanned Search and Rescue Missions through Intelligent Drones Guided by Computer Vision and Dynamic Pathfinding

Adiyan Kaul

Sohan Vichare

Table of Contents

Inspiration, Proposal, and Plan

Inspiration

Inspiration

The use of drones in search and rescue missions holds much promise. As of now, various government agencies are using drones to assist in search and rescue missions. However, we noticed one thing about drones being used in search and rescue missions - *they are all controlled by people.* We strongly believe that autonomously controlled drones, that is, drones that can control themselves, can make search and rescue missions that currently require massive amounts of manpower due to their sheer size much more efficient.

Brief Project Proposal

Build and program a drone that can accomplish the following:

1. Take off and fly itself around a user-defined area of land
2. Detect people
3. Bring these people to user-defined safe points
    a. Detect and avoid obstacles (for use in rough, mountainous, environments)
4. Collect relevant data
5. Complete the above as efficiently as possible

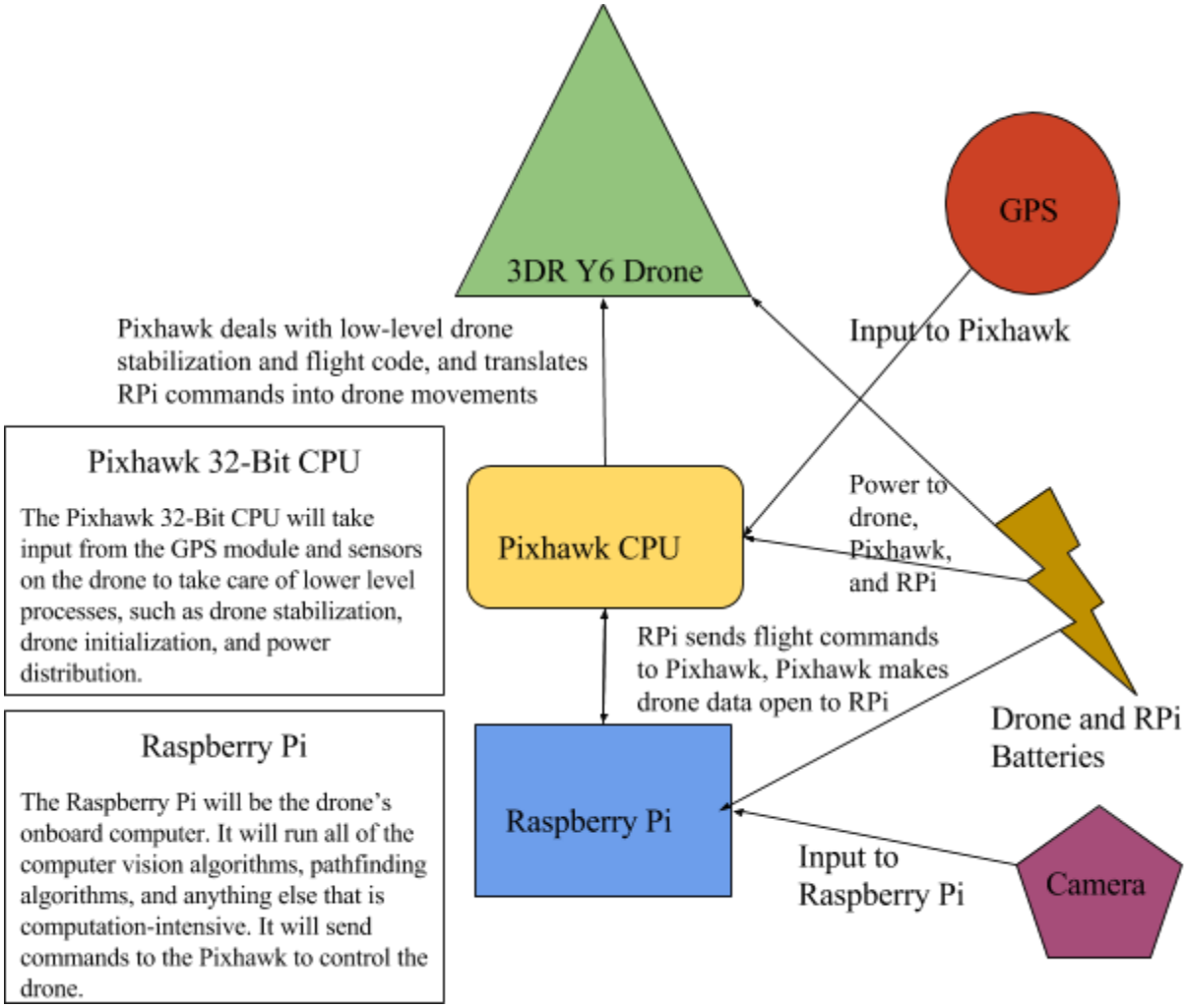
The complete Engineering Project Detailed Research Plan is attached at the end of this journal.

Plan

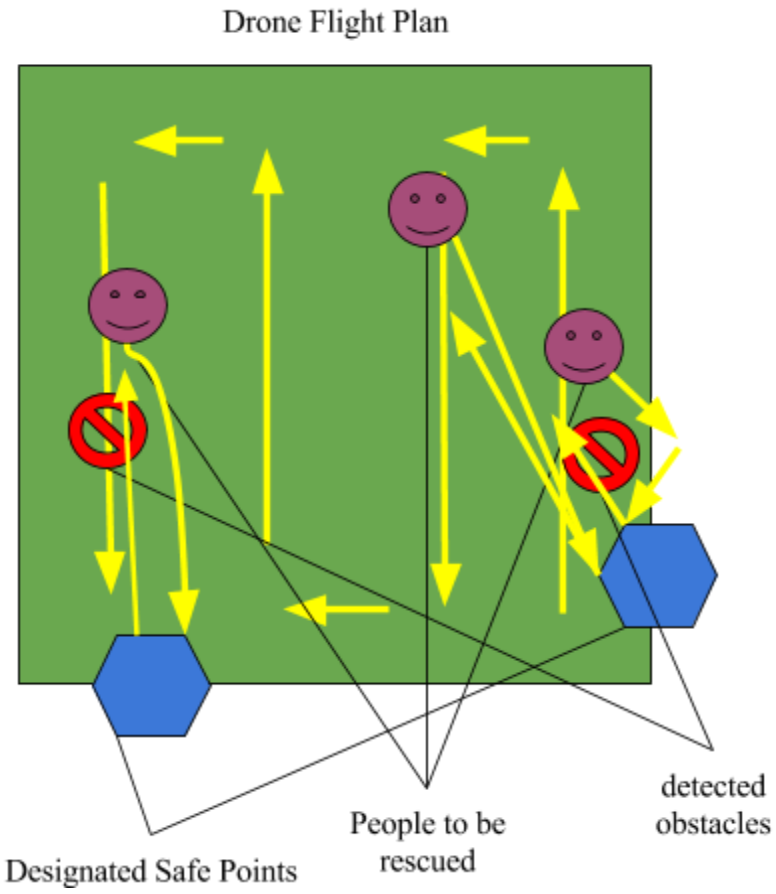
<u>Materials</u>

1. 3DR Y6 2014 DIY Drone Kit (Drone body)
2. 3DR GPS Module (Drone GPS)
3. Pixhawk 32-Bit with ArduPilot (Drone CPU)
4. Raspberry Pi 2 B+ (Onboard Drone Companion Computer)
5. Raspberry Pi Camera (Camera for Person and Object Recognition)
6. 3DR Y6 Lithium-Polymer 4S 5200mAh Battery (Drone battery)
7. Anker Astro Battery (Raspberry Pi Onboard Battery)
8. MicroUSB Cable (Connect Raspberry Pi to Pixhawk)

Build Plan Diagram



Flight Plan Diagram



Drone will:

1. Divide X by Y grid into 5 by 5 meter "chunks"
2. Analyze each "chunk" for people using computer vision
3. If people are found, it will bring these people back to designated safe locations, and avoid detected obstacles while doing this
4. Continue looping through chunks

Drone Specifications

3DR Y6 Body Specs

Motors: 6

Weight (without battery): 1200 grams

Weight (with batteries): 2100 grams

Wingspan: 40 centimeters

Propeller Length: 23 centimeters

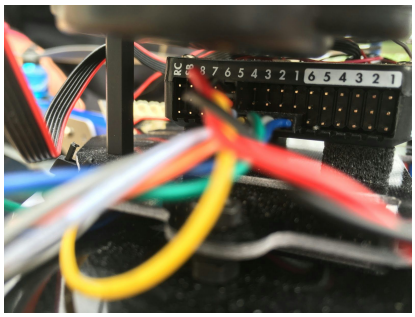
Height: 29 centimeters

Pixhawk Drone CPU Specs

Processor: 168 MHz / 252 MIPS Cortex-M4F

Height: 1.8 centimeters

Width: 7.8 centimeters



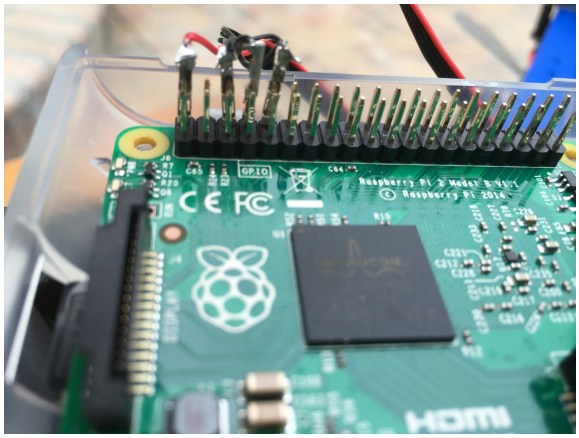
Raspberry Pi Onboard Computer Specs

Processor: A 900MHz quad-core ARM Cortex-A7 CPU

RAM: 1GB

Operating System: Debian Linux

Dimensions: 104mm x 75mm x 23mm



Person Recognition Code

Overview

1. Wrote code with Opencv
2. Used Haar Cascade Classifiers to determine what was a person
3. Found the classifier granted by OpenCV far too inaccurate so we trained the classifier to make it far more accurate
4. Optimized it to work real time and efficiently

B.) Code Samples

1. Code for frontal face detection

code	explanation
1.import numpy as np import cv2 2. face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml') eye_cascade = cv2.CascadeClassifier('haarcascade_eye.xml') 3.img = cv2.imread('insert image file name here'')	1. These are the necessary imports when working with face detection. Cv2 is the opencv library and numpy is a math library 2. This creates the classifiers which deal with finding people. The face cascade finds faces and the eye cascade finds eyes 3. This sends the image to opencv and

	converts it into a matrix with the rgb values .
4. rects = face_cascade.detectMultiScale(img, scaleFactor=1.3, minNeighbors=4, minSize=(30, 30),flags=cv2.CASCADE_SCALE_IMAGE)	4. This finds the actual faces with the help of the parameters. The img is the matrix which was declared in step 3, the scale factor specifies how much the image size is reduced at each image scale, the minimum neighbors specifies how many neighbors each candidate should have, and the min size is the minimum possible size of the object.
5. for (x, y, w, h) in rects: cv2.rectangle(img, (x, y), (x+w, y+h), (0, 255, 0), 2) roi_color = img[y:y+h, x:x+w] eyes = eye_cascade.detectMultiScale(roi_color) for (ex,ey,ew,eh) in eyes: cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh), (0,255,0),2)	5. This draws the actual rectangles around the face and the eyes. The cv2.rectangle is the opencv function for drawing a rectangle. The roi_color is a subsection of the picture and the eyes are found in that section.
6. cv2.imshow('img',img) cv2.waitKey(0) cv2.destroyAllWindows()	6. This finally shows the resulting image after all the recognition and escapes when the escape key is pressed.

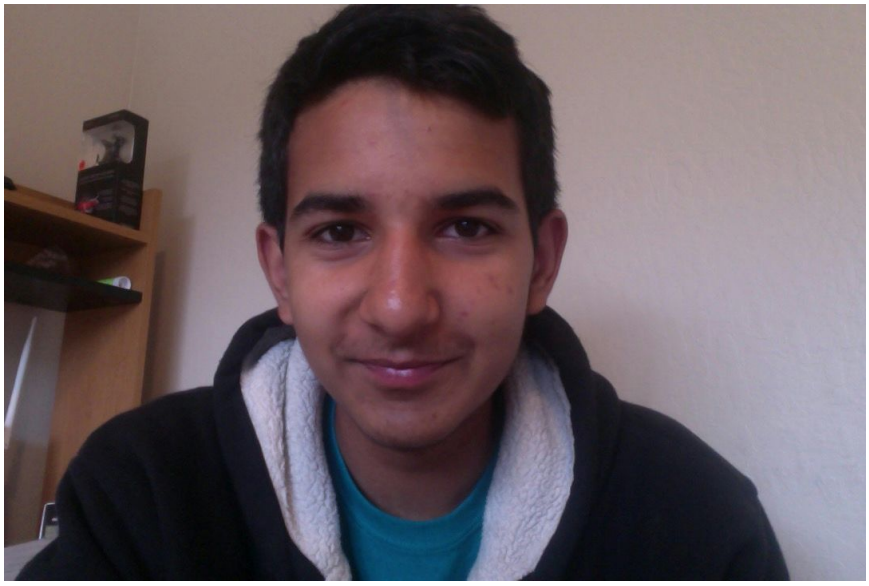
2. Pedestrian Detection

Code	Explanation
1.import numpy as np import cv2 import imutils from imutils.object_detection import non_max_suppression	1. These are all the imports necessary for the code to work.
2. hog = cv2.HOGDescriptor() hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())	2. This creates a hog descriptor or a histogram of oriented gradients which is a feature descriptor used in computer vision for the purpose of object detection
3. image = cv2.imread('insert file name here'') orig = image.copy()	3. This sends the image to opencv and converts it into a matrix with the rgb values .
4. (rects, weights) = hog.detectMultiScale(image, winStride=(4, 4), padding=(8, 8), scale=1.05)	4. This detects the people in the image using the parameters. Image refers to the step 3 image, win Stride dictates the "step size" in both the *x* and *y* location of the sliding window, the padding is a tuple which indicates the number of pixels in both the *x* and *y* direction in which the sliding window ROI is "padded" prior to HOG feature extraction,and the scale controls the factor in which our image is resized at each layter of the image pyramid, ultimately influencing the *number* of levels in the image pyramid.
5. for (x, y, w, h) in rects: cv2.rectangle(orig, (x, y), (x + w, y + h), (0, 0, 255), 2)	5. This draws the rectangles around all the people the previous step.
6. rects = np.array([[x, y, x + w, y + h] for (x, y, w, h) in rects]) pick = non_max_suppression(rects, probs=None, overlapThresh=0.65)	6. This applies non-maxima suppression to the bounding boxes using a fairly large overlap threshold to try to maintain overlapping boxes that are still people. This ensures that the people detection is accurate.
7.for (xA, yA, xB, yB) in pick: cv2.rectangle(image, (xA, yA), (xB, yB), (0, 255, 0), 2)	7. This draws the proper rectangles around the people based on the last step.

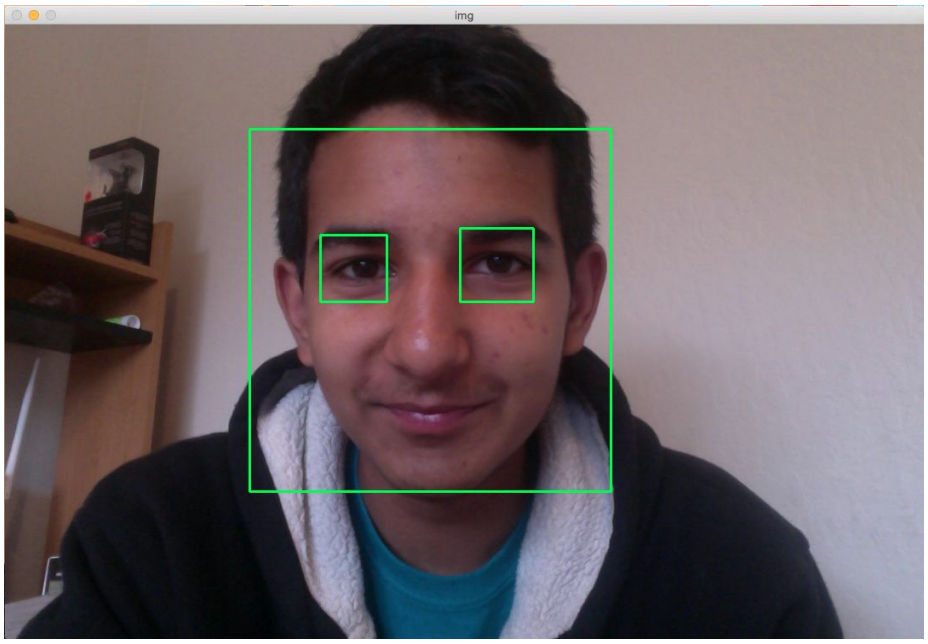
8. cv2.imshow("Before NMS", orig) cv2.imshow("After NMS", image) cv2.waitKey(0)	8. This displays two images, one with all the rectangles and one with the accurate rectangles.

Debrief

1. Here is an image that we experimented with:



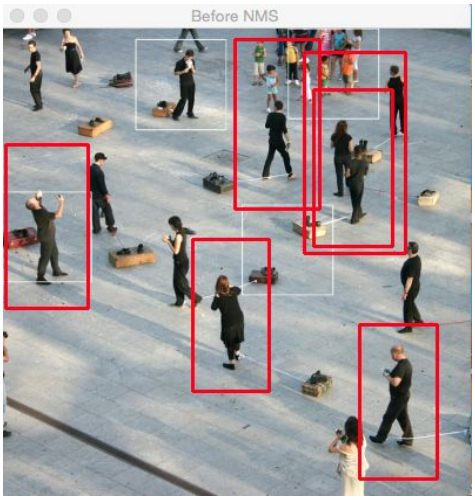
2. Here is the result:



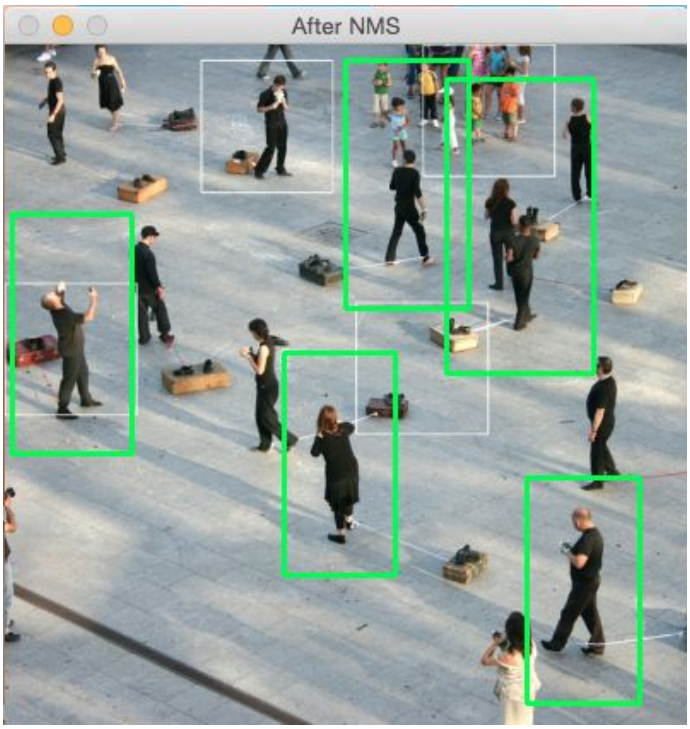
3. As can be seen the code was able to find my face and drew a rectangle around my face and my eyes.This took place in around 3 seconds showing its precision and accuracy

4. Here is an example of a drone's camera view:

5. The following two pictures shows the output of the people detection from the Hog classifier.
    a. The prelim image:



b. The accurate image:



This highlights the accuracy of the Hog Descriptor as it was able to recognize most of the people in the image with remarkable accuracy. All of the rectangles the program drew were actual people and this is how we are finding the people with the drone. Although the code was not able to recognize every single person in the picture, it was still very accurate as every rectangle it drew was a real person.

Pathfinding Code

Overview

Use

The pathfinding code will be used to the drone lead people back to user-defined safe points around things that they cannot get across. Computer Vision code (see above) will identify and input the locations of points that people will not be able to cross.

Requirements

1. Dynamic Path Replanning
    a. the drone has to navigate itself in an environment which it does not fully know - the path planning algorithm must be able to take input and dynamically replan a path with a new obstacle on the fly
2. Latitude and Longitude
    a. the path planning algorithm must be able to take in inputs of latitude and longitude coordinates for safe points and obstacles
3. Optimized
    a. the path planning algorithm must be optimized (NOT brute force) as it will be run alongside computer vision algorithms on the Raspberry Pi

Plan

Based on our requirements, we decided to base our algorithm off the D* Lite Path Planning Algorithm (based on this paper: http://idm-lab.org/bib/abstracts/papers/aaai02b.pdf, printout attached at back of report). This algorithm, developed by Professor Sven Koenig, not only is not processing-intensive but also allows for dynamic path replanning.

Comparison to A* and Djikstra's Algorithm

A* and Djikstra's algorithm are two very popular path-planning algorithms. However, both fall short when it comes to replanning a path upon addition or subtraction of a new obstacle

in the environment. D* Lite, on the other hand, has the power to dynamically change the costs of surrounding nodes, so essentially the path can be replanned around the newly added obstacle while most of the original path remains intact.

Code Walkthrough

Due to space constraints, the actual D* Lite code is not included here. To view it, go to page 49

D* Lite Algorithm Pseudocode (Taken From *D* Lite*, by Sven Koenig and Maxim Likhachev)

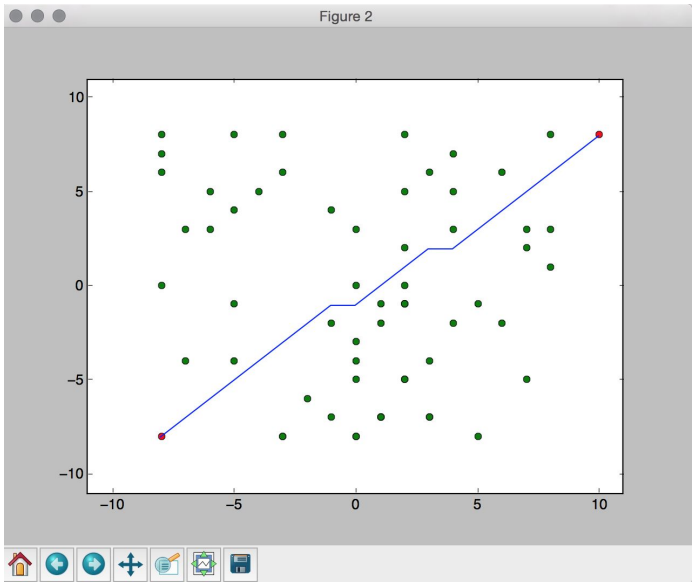
```
procedure CalculateKey(s)
{01} return [min(g(s), rhs(s)) + h(s, s_goal); min(g(s), rhs(s))];
procedure Initialize()
{02} U = ∅;
{03} for all s ∈ S rhs(s) = g(s) = ∞;
{04} rhs(s_start) = 0;
{05} U.Insert(s_start, CalculateKey(s_start));
procedure UpdateVertex(u)
{06} if (u ≠ s_start) rhs(u) = min_{s' ∈ Pred(u)} (g(s') + c(s', u));
{07} if (u ∈ U) U.Remove(u);
{08} if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));
procedure ComputeShortestPath()
{09} while (U.TopKey() < CalculateKey(s_goal) OR rhs(s_goal) ≠ g(s_goal))
{10}    u = U.Pop();
{11}    if (g(u) > rhs(u))
{12}       g(u) = rhs(u);
{13}       for all s ∈ Succ(u) UpdateVertex(s);
{14}    else
{15}       g(u) = ∞;
{16}       for all s ∈ Succ(u) ∪ {u} UpdateVertex(s);
procedure Main()
{17} Initialize();
{18} forever
{19}    ComputeShortestPath();
{20}    Wait for changes in edge costs;
{21}    for all directed edges (u, v) with changed edge costs
{22}       Update the edge cost c(u, v);
{23}       UpdateVertex(v);
```

D* Lite Pathfinding Algorithm Simple Explanation

D* Lite begins (Initialize() method above) by finding the ideal shortest path between the "start" and the "end." It divides the world into a grid, and each point is a node. D* Lite uses a novel method, taken from the Lifelong Planning A* algorithm to cheaply adapt to obstacles as they block the computed path. Each node on the specified path calculates its own "g" value, which is the cost of getting to that node from the start. However, the algorithm also computes another "g" value based on the node's neighboring node's "g" values, which is called "rhs." The minimum value is stored in "rhs" and later used to update the node's "g" value. During this estimating process, neighboring nodes take into account the new obstacles so that "g" is updated with an "rhs" that has taken into account the new obstacles or free spaces.

Debrief

D* Lite Results



As can be seen, the D* Lite pathfinding code successfully found the shortest path between a randomly generated field of obstacles. Here are a few more outputs:

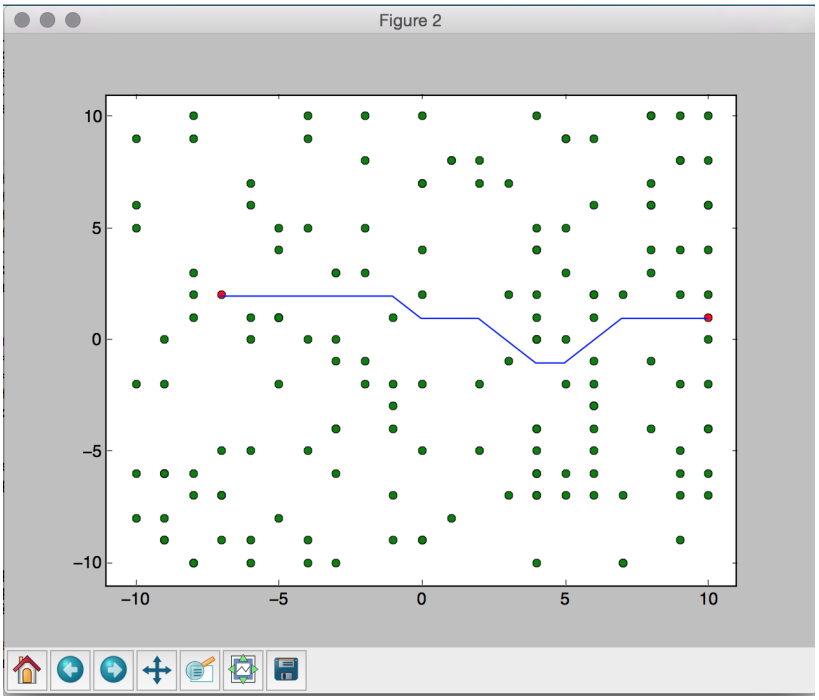


Figure 2
10
5
0
−5
−10
−10
−5
0
5
10

Drone Flight Code

Overview

1. Wrote Python code to get the drone to fly
2. Used dronekit python to do so (http://dronekit.io)

B.) Code Samples

Code	Explanation
1. from dronekit import connect, VehicleMode, LocationGlobalRelative import time 2. print 'Connecting to vehicle;' vehicle = connect("/dev/ttyACM0", wait_ready=True) 3. def arm_and_takeoff(aTargetAltitude): """ Arms vehicle and fly to aTargetAltitude. """ print "Basic pre-arm checks" # Don't try to arm until autopilot is ready while not vehicle.is_armable: print " Waiting for vehicle to initialise..." time.sleep(1) print "Arming motors" # Copter should arm in GUIDED mode	1. These are the necessary imports for the code to work. 2. This declares the vehicle variable and sets it to the drone using the connection. 3. This defines a function arm_and_takeoff which does a lot of different things. First it confirms the vehicle is armed before it can take off. Then using the vehicle.simple_takeoff() function it starts the propeller. Once it has reached that height the function will break.

vehicle.mode = VehicleMode("GUIDED")
vehicle.armed = True
 	
# Confirm vehicle armed before attempting to take off 	
while not vehicle.armed:
print " Waiting for arming..."
time.sleep(1)
 	
print "Taking off!"
vehicle.simple_takeoff(aTargetAltitude) # Take off to target altitude
 	
# Wait until the vehicle reaches a safe height before processing the goto (otherwise the command 	
# after Vehicle.simple_takeoff will execute immediately). 	
while True:
print " Altitude: ", vehicle.location.global_relative_frame.alt
#Break and return from function just below target altitude.
if vehicle.location.global_relative_frame.alt>=aTargetAltitude*0.95:
print "Reached target altitude"
break
time.sleep(1)
 	
4. arm_and_takeoff(1)
	4. This code calls the arm_and_takeoff function declared above which flies the drone up one meter because of the passed in 1 parameter. After the

print "Set default/target airspeed to 3" vehicle.airspeed=3 print "Returning to Launch" vehicle.mode = VehicleMode("LAND") #Close vehicle object before exiting script print "Close vehicle object" vehicle.close()	drone has reached the height of one meter, the mode of the vehicle is changed to RTL. This signifies return to launch which means the drone flies back to the location it started from. The drone is then disarmed and turned off.

Project Summary

Person Recognition:

This was also extremely successful and was able to detect people at a remarkable accuracy rate. We fed the program lots and lots of pictures and found the accuracy to be right about 80%. When we were testing we realized that the program often didn't find people who were laying down. To solve this problem we trained our own hog classifier. This was done in three steps. **Step 1)** We prepared some training images of the objects you want to detect (positive samples). Also we will prepared some images with no objects of interest (negative samples).**Step 2)** We then detected HOG features of the training sample and use this features to train an SVM classifier (also provided in OpenCV).**Step 3)** Use the coefficients of the trained SVM classifier in HOGDescriptor.setSVMDetector() method. This was able to get a lot more people but it still wasn't as accurate as we would like it to be. However, the person recognition was accurate enough for us to be able to get people with accuracy and be able to save them.

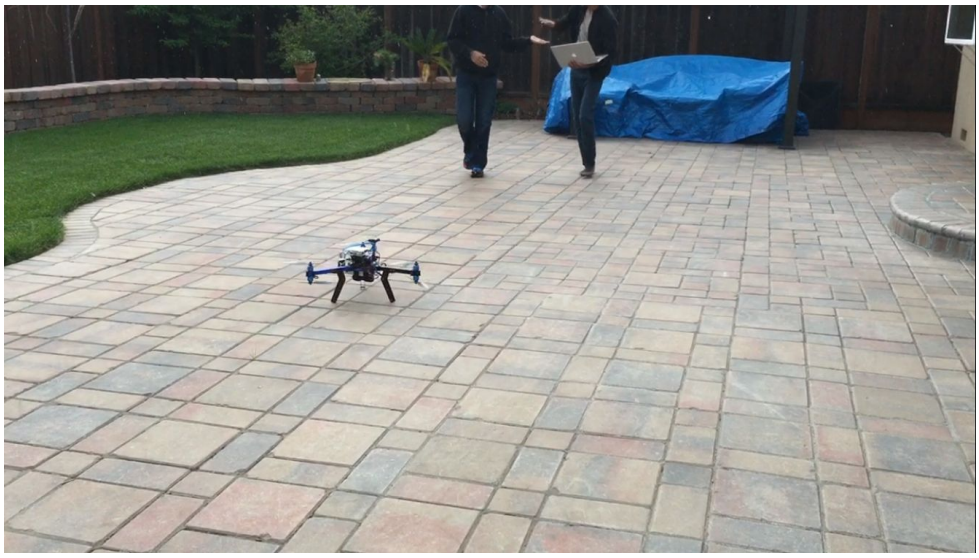
Pathfinding:

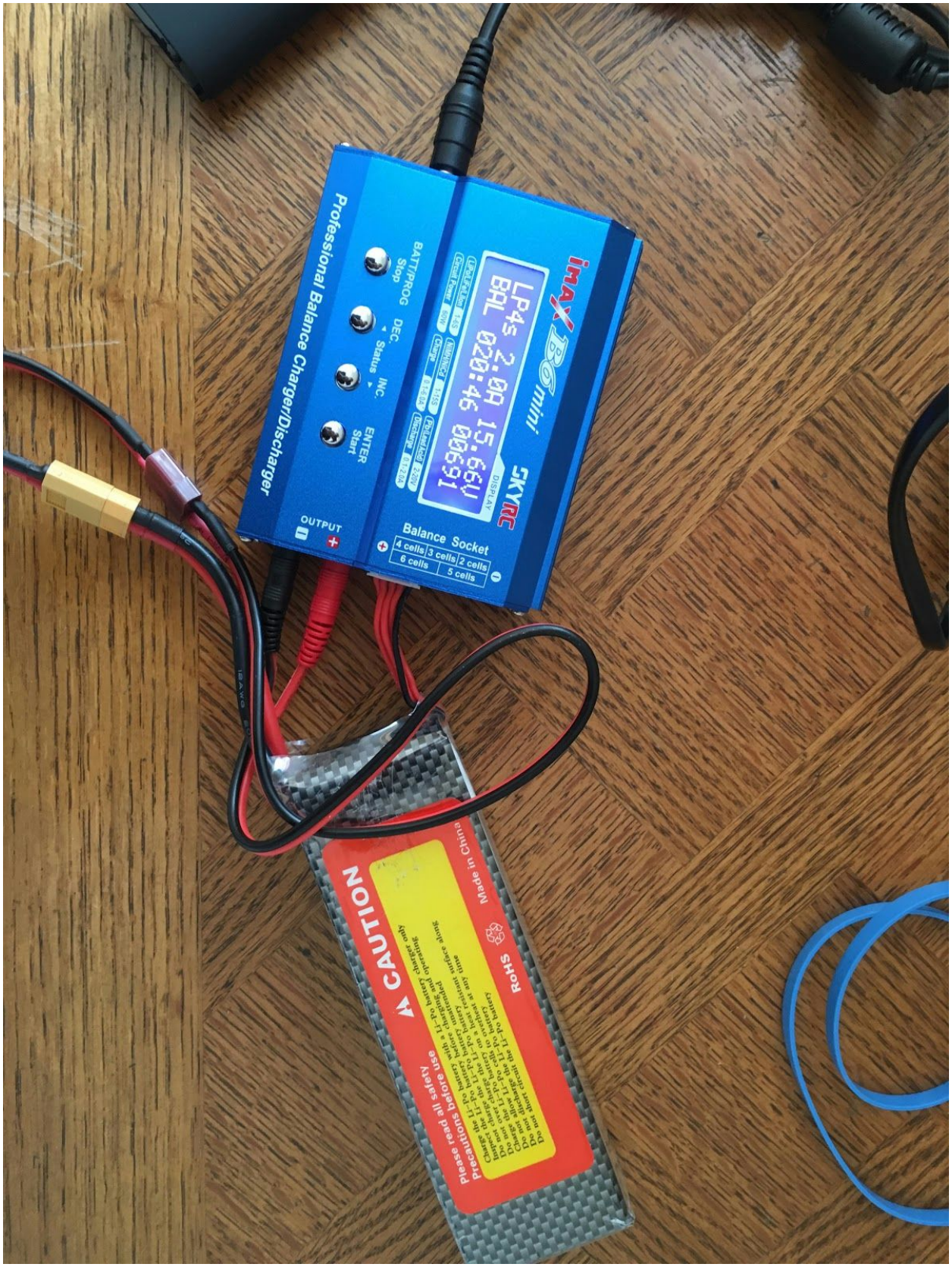
We were successfully able to implement the D* Lite pathfinding algorithm to dynamically find the shortest path between the drone and the safe "goal" location while avoiding obstacles to the person. Due to its grid based processing, the algorithm worked perfectly after we modified its "world" to be a grid of latitude and longitudinal points.

Drone Flight:

The drone flight code worked successfully. The Raspberry Pi sent commands to the Pixhawk, which then interfaced with the drone's motors to control the drone. We had to put set times behind each command we sent to the Pixhawk to allow them to run and/or continuously monitor the location of the drone in the world frame to make sure that one command was not sent while another was not finished running. The drone code we wrote made it easy to interface with the pixhawk, and is reusable, so that we can utilize it in the future to easily control a drone.

Photos




iMAX B6 mini
Professional Balance Charger/Discharger
SKYRC
LP4s 2.0A 15.66V
BAL 020:46 00691
BATT/PROG
Stop
DEC.
Status
INC.
ENTER
Start
OUTPUT
Balance Socket
4 cells
3 cells
2 cells
6 cells
5 cells
CAUTION
Made in China
RoHS


Battery Pack
14.8V 35C
5500 mAh
RoHS


Crimping Tool

Search and Rescue Code

```
import numpy as np
import math
import cv2
from picamera.array import PiRGBArray
from picamera import PiCamera
from dronekit import connect, VehicleMode, LocationGlobalRelative
from pymavlink import mavutil # Needed for command message
definitions
import time
from dronekit_sitl import SITL
from imutils.object_detection import non_max_suppression



#defines drone variable
vehicle = connect("/dev/ttyACM0", wait_ready=True);
#defines variable t
face_cascade =
cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
# initialize the camera and grab a reference to the raw camera
capture
camera = PiCamera()
rawCapture = PiRGBArray(camera)
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())



#define location class that stores latitude and longitude
class Location:
   "Basic Location class, used to store latitude and longitude"

   def __init__(self, latitude, longitude):
      self.latitude = latitude
      self.longitude = longitude
```

```
#define SafeLocation class, which stores location data, which people
are at each Safe Location
class SafeLocation:
    "Safe Location class, used to store data for a safe location. Has
two attributes, one is location (a location object which stores the
location), and the second is peopleArrivedArray, which is an array
that stores the people who have reached this location."

    def __init__(self, location):
        self.location = location
        #an array of Person objects, used to store what people are
already at the safeLocation
        self.peopleArrivedArray = [];

#define helper method to print arrays with locations so that they are
easy to see
def printLocationArray(locationArray):
    for item in locationArray:
        print item.latitude
        print item.longitude
        print " "

#define helper method to get an array of the locations object from an
array of a person, safelocation, or any other object that has
location as a property
def getLocationsArray(safeLocsArray):
        returnArray = []
        for item in safeLocsArray:
            returnArray.append(item.location);
        return returnArray;

#distance formula function, finds distance between two location
objects, accounting for the earth's spherical shape (assumes that
earth is a sphere)
def distanceBetweenLocations(location1, location2):

    # Convert latitude and longitude to spherical coordinates in
radians.
    degreesToRadians = math.pi/180.0

    # phi = 90 - latitude
    phi1 = (90.0 - location1.latitude)*degreesToRadians
    phi2 = (90.0 - location2.latitude)*degreesToRadians
```

```
    # theta = longitude
    theta1 = location1.longitude*degreesToRadians
    theta2 = location2.longitude*degreesToRadians

    # Compute spherical distance from spherical coordinates.

    # For two locations in spherical coordinates
    # (1, theta, phi) and (1, theta', phi')
    # cosine( arc length ) =
    #    sin phi sin phi' cos(theta-theta') + cos phi cos phi'
    # distance = rho * arc length

    cos = (math.sin(phi1)*math.sin(phi2)*math.cos(theta1 - theta2) +
           math.cos(phi1)*math.cos(phi2))
    arc = math.acos( cos )

    # multiply and return arc by the right distance unit, miles or
kilometers
    unit = 1;

    return arc * unit;

def closestSafePoint(droneLocation, safePointLocationArray):
        #define array that will be returned
        closestPointIndexArray = [];
        usedIndexArray = [];
        closestDist = 1000;
        closestIndex = 0;

        for index, item in enumerate(safePointLocationArray):
            for ind, itm in enumerate(safePointLocationArray):
                if ind not in usedIndexArray:
                    dist = distanceBetweenLocations(itm.location,
droneLocation);
                    if dist <= closestDist:
                        closestDist = dist;
                        closestIndex = ind;

            if closestIndex not in usedIndexArray:
                usedIndexArray.append(closestIndex);
                closestPointIndexArray.append(closestIndex);
                closestDist = 1000;
```

```
        return closestPointIndexArray.sort();

#takes a "picture" of what the picamera is seeing by saving the array
def takePicture():
    camera.capture(rawCapture, format="bgr")
    image = rawCapture.array
    return image


# gets the number of people in the image
def readImage(image):
    (rects, weights) = hog.detectMultiScale(image, winStride=(4, 4),
     padding=(8, 8), scale=1.05)
    rects = np.array([[x, y, x + w, y + h] for (x, y, w, h) in
rects])
    pick = non_max_suppression(rects, probs=None, overlapThresh=0.65)
    count=0
    for (xA, yA, xB, yB) in pick:
         count=count+1
    return count

def findNumPeople():
    img =takePicture();
    people= readImage(img)
    return people
'''dx = R*cos(theta)
   = 500 * cos(135 deg)
   = -353.55 meters

dy = R*sin(theta)
   = 500 * sin(135 deg)
   = +353.55 meters

delta_longitude = dx/(111320*cos(latitude))
                = -353.55/(111320*cos(41.88592 deg))
                = -.004266 deg (approx -15.36 arcsec)

delta_latitude = dy/110540
               = 353.55/110540
               =  .003198 deg (approx 11.51 arcsec)

Final longitude = start_longitude + delta_longitude
```

```
                = -87.62788 - .004266
                = -87.632146

Final latitude = start_latitude + delta_latitude
               = 41.88592 + .003198
               = 41.889118
'''

def get_distance_metres(aLocation1, aLocation2):
    """
    Returns the ground distance in metres between two LocationGlobal
objects.

    This method is an approximation, and will not be accurate over
large distances and close to the
    earth's poles. It comes from the ArduPilot test code:

https://github.com/diydrones/ardupilot/blob/master/Tools/autotest/com
mon.py
    """
    dlat = aLocation2.lat - aLocation1.lat
    dlong = aLocation2.lon - aLocation1.lon
    return math.sqrt((dlat*dlat) + (dlong*dlong)) * 1.113195e5


#initializes drone
#connection string is the port at which the rpi connects to drone
#isSimulator is a boolean value for whether this is being used on the
actual drone or a simulator
def initializeDrone(connectionString, isSimulator):
    if isSimulator:
        sitl = SITL()
        sitl.download('copter', '3.3', verbose=True)
        sitl_args = ['-I0', '--model', 'quad',
'--home=-35.363261,149.165230,584,353']
        sitl.launch(sitl_args, await_ready=True, restart=True)
        print "Connecting to vehicle on: 'tcp:127.0.0.1:5760'"
        vehicle = connect('tcp:127.0.0.1:5760', wait_ready=True)
        # print information about vehicle
        print "Global Location: %s" % vehicle.location.global_frame
        print "Global Location (relative altitude): %s" %
vehicle.location.global_relative_frame
```

```
        print "Local Location: %s" % vehicle.location.local_frame
#NED
        print "Attitude: %s" % vehicle.attitude
        print "Velocity: %s" % vehicle.velocity
        print "GPS: %s" % vehicle.gps_0
        print "Groundspeed: %s" % vehicle.groundspeed
        print "Airspeed: %s" % vehicle.airspeed
        print "Battery: %s" % vehicle.battery
        print "EKF OK?: %s" % vehicle.ekf_ok
        print "Last Heartbeat: %s" % vehicle.last_heartbeat
        print "Rangefinder: %s" % vehicle.rangefinder
        print "Rangefinder distance: %s" %
vehicle.rangefinder.distance
        print "Rangefinder voltage: %s" % vehicle.rangefinder.voltage
        print "Heading: %s" % vehicle.heading
        print "Is Armable?: %s" % vehicle.is_armable
        print "System status: %s" % vehicle.system_status.state
        print "Mode: %s" % vehicle.mode.name    # settable
        print "Armed: %s" % vehicle.armed
    else:
        # Connect to the Vehicle
        print 'Connecting to vehicle;'
        vehicle = connect(connectionString, wait_ready=True)
        # print information about vehicle
        print "Global Location: %s" % vehicle.location.global_frame
        print "Global Location (relative altitude): %s" %
vehicle.location.global_relative_frame
        print "Local Location: %s" % vehicle.location.local_frame
#NED
        print "Attitude: %s" % vehicle.attitude
        print "Velocity: %s" % vehicle.velocity
        print "GPS: %s" % vehicle.gps_0
        print "Groundspeed: %s" % vehicle.groundspeed
        print "Airspeed: %s" % vehicle.airspeed
        print "Battery: %s" % vehicle.battery
        print "EKF OK?: %s" % vehicle.ekf_ok
        print "Last Heartbeat: %s" % vehicle.last_heartbeat
        print "Rangefinder: %s" % vehicle.rangefinder
        print "Rangefinder distance: %s" %
vehicle.rangefinder.distance
        print "Rangefinder voltage: %s" % vehicle.rangefinder.voltage
        print "Heading: %s" % vehicle.heading
        print "Is Armable?: %s" % vehicle.is_armable
```

```
        print "System status: %s" % vehicle.system_status.state
        print "Mode: %s" % vehicle.mode.name    # settable
        print "Armed: %s" % vehicle.armed

#flies a vehicle to a target altitude
def arm_and_takeoff(aTargetAltitude):
    #Arms vehicle and fly to aTargetAltitude.

    print "Basic pre-arm checks"
    # Don't try to arm until autopilot is ready
    while not vehicle.is_armable:
        print " Waiting for vehicle to initialise..."
        time.sleep(1)

    print "Arming motors"
    # Copter should arm in GUIDED mode
    vehicle.mode    = VehicleMode("GUIDED")
    vehicle.armed   = True

    # Confirm vehicle armed before attempting to take off
    while not vehicle.armed:
        print " Waiting for arming..."
        time.sleep(1)

    print "Taking off!"
    vehicle.simple_takeoff(aTargetAltitude)
    # Wait until the vehicle reaches a safe height before processing
the goto (otherwise the command
    #  after Vehicle.simple_takeoff will execute immediately).
    while True:
        print " Altitude: ",
vehicle.location.global_relative_frame.alt
        #Break and return from function just below target altitude.
        if
vehicle.location.global_relative_frame.alt>=aTargetAltitude*0.95:
            print "Reached target altitude"
            break
        time.sleep(1)

#stops program and counts down for a certain number of seconds
def countdown(amtTime):
    i = 0
    while i <= amtTime:
```

```
        print("COUNTDOWN: "+str(amtTime-i))
        time.sleep(1)
        i = i+1

#stops program and countrs down for a certain number of seconds while
displaying the altitude
def countdownAlt(amTime):
    o = 0
    while o <= amTime:
        print("COUNTDOWN: "+str(amTime-o))
        print " Altitude: ",
vehicle.location.global_relative_frame.alt
        time.sleep(1)
        o = o+1

#initializes and takes off drone
#alt: alitutude in meters
#countdownSeconds = number of seconds till takeoff
#isSim = true if we are using a simulator
#connectString = port at which the rpi is connected to drone
"/dev/tty/ACM0" for usb
def takeOff(alt, countdownSeconds, isSim, connectString):
    initializeDrone("/dev/tty/ACM0", isSim)
    countdown(countdownSeconds)
    vehicle.airspeed=3
    arm_and_takeoff(alt)

def moveVehicle(velocity_x, velocity_y, velocity_z, duration):
    """
    Move vehicle in direction based on specified velocity vectors.
    """
    msg =
vehicle.message_factory.set_position_target_local_ned_encode(
        0,       # time_boot_ms (not used)
        0, 0,    # target system, target component
        mavutil.mavlink.MAV_FRAME_LOCAL_NED, # frame
        0b0000111111000111, # type_mask (only speeds enabled)
        0, 0, 0, # x, y, z positions (not used)
        velocity_x, velocity_y, velocity_z, # x, y, z velocity in m/s
        0, 0, 0, # x, y, z acceleration (not supported yet, ignored
in GCS_Mavlink)
        0, 0)    # yaw, yaw_rate (not supported yet, ignored in
GCS_Mavlink)
```

```
    # send command to vehicle on 1 Hz cycle
    for x in range(0,duration):
        vehicle.send_mavlink(msg)
        time.sleep(1)

#goto helper function for drone
def goToLocation(targetLocation, gotoFunction=vehicle.simple_goto):
    currentLocation=vehicle.location.global_relative_frame
    targetDistance=get_distance_metres(currentLocation,
targetLocation)
    gotoFunction(targetLocation)

    while vehicle.mode.name=="GUIDED": #Stop action if we are no
longer in guided mode.

remainingDistance=get_distance_metres(vehicle.location.global_frame,
targetLocation)
        print "Distance to target: ", remainingDistance
        if remainingDistance<=targetDistance*0.01: #Just below
target, in case of undershoot.
            print "Reached target"
            break;
        time.sleep(2)



########ACTUAL PROGRAM STARTS HERE ALL FUNCTIONS GO ABOVE
THIS#############
safePointArray = [SafeLocation(Location(0,0)),
SafeLocation(Location(0,0)), SafeLocation(Location(0,0)),
SafeLocation(Location(0,0))]
squareLength = input("Input side length of square to survey in
meters: ")
takeOff(5,5,True,"/dev/tty/AMA0")
# Set airspeed using attribute
vehicle.airspeed = 1 #m/s
# Set groundspeed using attribute
vehicle.groundspeed = 1 #m/s
homeL = vehicle.home_location
homeLat = homeL.lat
homeLong = homeL.lon
```

```
home = Location(homeLat, homeLong)
currLoc = Location(0,0);
numFiveMeterSegments = int(round(squareLength/5))
trackFiveMeterSegments = numFiveMeterSegments
for i in range(0,numFiveMeterSegments):
    #move the vehicle north for 5 seconds at a speed of 1 m/s
    moveVehicle(1,0,0,5)
    #update currLoc variable to store location
    currLoc =
Location(vehicle.location.global_frame.lat,vehicle.location.global_fr
ame.lon)
    trackFiveMeterSegments = trackFiveMeterSegments - 1
    #finds num people
    numPeople = findNumPeople()
    if numPeople >= 1:
        currLoc =
Location(vehicle.location.global_frame.lat,vehicle.location.global_fr
ame.lon)
        index = closestSafePoint(currLoc, safePointArray)
        safePointLocation =
LocationGlobalRelative(safePointArray[index[0]].location.latitude,
safePointArray[index[0]].location.longitude, 5)
        goToLocation(safePointLocation)
        time.sleep(5)
        goToLocation(currLoc)

    if trackFiveMeterSegments == 0:
        break;

print "Returning to Launch"
vehicle.mode    = VehicleMode("LAND")
countdownAlt(30)

#Close vehicle object before exiting script
print "Close vehicle object"
vehicle.close()
```

Pathfinding Code

This code was interfaced with python through a C++ helper library called Boost Python. We wrote wrappers in C++ that made compiled code available through a python library. The only drawback to this is that it needs to be recompiled from source every different system it is run on. The wrapper is named dstarimplement.cpp.

Dstar.cpp

```
/*BY SOHAN VICHARE AND ADIYAN KAUL, BASED ON AND BUILT FROM
 * James Neufeld (neufeld@cs.ualberta.ca)
 * and Arek Sredzki (arek@sredzki.com)
 */

#include "Dstar.h"

#ifdef USE_OPEN_GL
#ifdef MACOS
#include <OpenGL/gl.h>
#else
#include <GL/gl.h>
#endif
#endif

/* void Dstar::Dstar()
 * -------------------------
 * Constructor sets constants.
 */
Dstar::Dstar() {

  maxSteps = 80000;  // node expansions before we give up
  C1       = 1;      // cost of an unseen cell

}

/* float Dstar::keyHashCode(state u)
```

```
 * --------------------------
 * Returns the key hash code for the state u, this is used to compare
 * a state that have been updated
 */
float Dstar::keyHashCode(state u) {

  return (float)(u.k.first + 1193*u.k.second);

}

/* bool Dstar::isValid(state u)
 * --------------------------
 * Returns true if state u is on the open list or not by checking if
 * it is in the hash table.
 */
bool Dstar::isValid(state u) {

  ds_oh::iterator cur = openHash.find(u);
  if (cur == openHash.end()) return false;
  if (!close(keyHashCode(u), cur->second)) return false;
  return true;

}

/* void Dstar::getPath()
 * --------------------------
 * Returns the path created by replan()
 */
list<state> Dstar::getPath() {
  return path;
}

int state::getX(){
    return x;
}

int state::getY(){
    return y;
}

/* bool Dstar::occupied(state u)
 * --------------------------
 * returns true if the cell is occupied (non-traversable), false
 * otherwise. non-traversable are marked with a cost < 0.
 */
bool Dstar::occupied(state u) {

  ds_ch::iterator cur = cellHash.find(u);
```

```
  if (cur == cellHash.end()) return false;
  return (cur->second.cost < 0);
}

/* void Dstar::init(int sX, int sY, int gX, int gY)
 * --------------------------
 * Init dstar with start and goal coordinates, rest is as per
 * [S. Koenig, 2002]
 */
void Dstar::init(int sX, int sY, int gX, int gY) {

  cellHash.clear();
  path.clear();
  openHash.clear();
  while(!openList.empty()) openList.pop();

  k_m = 0;

  s_start.x = sX;
  s_start.y = sY;
  s_goal.x  = gX;
  s_goal.y  = gY;

  cellInfo tmp;
  tmp.g = tmp.rhs =  0;
  tmp.cost = C1;

  cellHash[s_goal] = tmp;

  tmp.g = tmp.rhs = heuristic(s_start,s_goal);
  tmp.cost = C1;
  cellHash[s_start] = tmp;
  s_start = calculateKey(s_start);

  s_last = s_start;

}
/* void Dstar::makeNewCell(state u)
 * --------------------------
 * Checks if a cell is in the hash table, if not it adds it in.
 */
void Dstar::makeNewCell(state u) {

  if (cellHash.find(u) != cellHash.end()) return;

  cellInfo tmp;
  tmp.g       = tmp.rhs = heuristic(u,s_goal);
```

```
  tmp.cost    = C1;
  cellHash[u] = tmp;

}

/* double Dstar::getG(state u)
 * ---------------------------
 * Returns the G value for state u.
 */
double Dstar::getG(state u) {

  if (cellHash.find(u) == cellHash.end())
    return heuristic(u,s_goal);
  return cellHash[u].g;

}

/* double Dstar::getRHS(state u)
 * ---------------------------
 * Returns the rhs value for state u.
 */
double Dstar::getRHS(state u) {

  if (u == s_goal) return 0;

  if (cellHash.find(u) == cellHash.end())
    return heuristic(u,s_goal);
  return cellHash[u].rhs;

}

/* void Dstar::setG(state u, double g)
 * ---------------------------
 * Sets the G value for state u
 */
void Dstar::setG(state u, double g) {

  makeNewCell(u);
  cellHash[u].g = g;
}

/* void Dstar::setRHS(state u, double rhs)
 * ---------------------------
 * Sets the rhs value for state u
 */
double Dstar::setRHS(state u, double rhs) {

  makeNewCell(u);
```

```
  cellHash[u].rhs = rhs;

}

/* double Dstar::eightCondist(state a, state b)
 * --------------------------
 * Returns the 8-way distance between state a and state b.
 */
double Dstar::eightCondist(state a, state b) {
  double temp;
  double min = fabs(a.x - b.x);
  double max = fabs(a.y - b.y);
  if (min > max) {
    double temp = min;
    min = max;
    max = temp;
  }
  return ((M_SQRT2-1.0)*min + max);
}

/* int Dstar::computeShortestPath()
 * --------------------------
 * As per [S. Koenig, 2002] except for 2 main modifications:
 * 1. We stop planning after a number of steps, 'maxsteps' we do this
 *    because this algorithm can plan forever if the start is
 *    surrounded by obstacles.
 * 2. We lazily remove states from the open list so we never have to
 *    iterate through it.
 */
int Dstar::computeShortestPath() {

  list<state> s;
  list<state>::iterator i;

  if (openList.empty()) return 1;

  int k=0;
  while ((!openList.empty()) &&
         (openList.top() < (s_start = calculateKey(s_start))) ||
         (getRHS(s_start) != getG(s_start))) {

    if (k++ > maxSteps) {
      fprintf(stderr, "At maxsteps\n");
      return -1;
    }


    state u;
```

```
    bool test = (getRHS(s_start) != getG(s_start));

    // lazy remove
    while(1) {
      if (openList.empty()) return 1;
      u = openList.top();
      openList.pop();

      if (!isValid(u)) continue;
      if (!(u < s_start) && (!test)) return 2;
      break;
    }

    ds_oh::iterator cur = openHash.find(u);
    openHash.erase(cur);

    state k_old = u;

    if (k_old < calculateKey(u)) { // u is out of date
      insert(u);
    } else if (getG(u) > getRHS(u)) { // needs update (got better)
      setG(u,getRHS(u));
      getPred(u,s);
      for (i=s.begin();i != s.end(); i++) {
        updateVertex(*i);
      }
    } else {   // g <= rhs, state has got worse
      setG(u,INFINITY);
      getPred(u,s);
      for (i=s.begin();i != s.end(); i++) {
        updateVertex(*i);
      }
      updateVertex(u);
    }
  }
  return 0;
}

/* bool Dstar::close(double x, double y)
 * --------------------------
 * Returns true if x and y are within 10E-5, false otherwise
 */
bool Dstar::close(double x, double y) {

  if (isinf(x) && isinf(y)) return true;
  return (fabs(x-y) < 0.00001);
```

```
}

/* void Dstar::updateVertex(state u)
 * --------------------------
 * As per [S. Koenig, 2002]
 */
void Dstar::updateVertex(state u) {

  list<state> s;
  list<state>::iterator i;

  if (u != s_goal) {
    getSucc(u,s);
    double tmp = INFINITY;
    double tmp2;

    for (i=s.begin();i != s.end(); i++) {
      tmp2 = getG(*i) + cost(u,*i);
      if (tmp2 < tmp) tmp = tmp2;
    }
    if (!close(getRHS(u),tmp)) setRHS(u,tmp);
  }

  if (!close(getG(u),getRHS(u))) insert(u);

}

/* void Dstar::insert(state u)
 * --------------------------
 * Inserts state u into openList and openHash.
 */
void Dstar::insert(state u) {

  ds_oh::iterator cur;
  float csum;

  u    = calculateKey(u);
  cur  = openHash.find(u);
  csum = keyHashCode(u);
  // return if cell is already in list. TODO: this should be
  // uncommented except it introduces a bug, I suspect that there is
a
  // bug somewhere else and having duplicates in the openList queue
  // hides the problem...
  //if ((cur != openHash.end()) && (close(csum,cur->second))) return;

  openHash[u] = csum;
  openList.push(u);
```

```
}

/* void Dstar::remove(state u)
 * --------------------------
 * Removes state u from openHash. The state is removed from the
 * openList lazilily (in replan) to save computation.
 */
void Dstar::remove(state u) {

  ds_oh::iterator cur = openHash.find(u);
  if (cur == openHash.end()) return;
  openHash.erase(cur);
}


/* double Dstar::trueDist(state a, state b)
 * ---------------------------
 * Euclidean cost between state a and state b.
 */
double Dstar::trueDist(state a, state b) {

  float x = a.x-b.x;
  float y = a.y-b.y;
  return sqrt(x*x + y*y);

}

/* double Dstar::heuristic(state a, state b)
 * ---------------------------
 * Pretty self explanitory, the heristic we use is the 8-way distance
 * scaled by a constant C1 (should be set to <= min cost).
 */
double Dstar::heuristic(state a, state b) {
  return eightCondist(a,b)*C1;
}

/* state Dstar::calculateKey(state u)
 * ---------------------------
 * As per [S. Koenig, 2002]
 */
state Dstar::calculateKey(state u) {

  double val = fmin(getRHS(u),getG(u));

  u.k.first  = val + heuristic(u,s_start) + k_m;
  u.k.second = val;

  return u;
```

```
}

/* double Dstar::cost(state a, state b)
 * --------------------------
 * Returns the cost of moving from state a to state b. This could be
 * either the cost of moving off state a or onto state b, we went
with
 * the former. This is also the 8-way cost.
 */
double Dstar::cost(state a, state b) {

  int xd = fabs(a.x-b.x);
  int yd = fabs(a.y-b.y);
  double scale = 1;

  if (xd+yd>1) scale = M_SQRT2;

  if (cellHash.count(a) == 0) return scale*C1;
  return scale*cellHash[a].cost;

}
/* void Dstar::updateCell(int x, int y, double val)
 * --------------------------
 * As per [S. Koenig, 2002]
 */
void Dstar::updateCell(int x, int y, double val) {

   state u;

  u.x = x;
  u.y = y;

  if ((u == s_start) || (u == s_goal)) return;

  makeNewCell(u);
  cellHash[u].cost = val;

  updateVertex(u);
}

/* void Dstar::getSucc(state u,list<state> &s)
 * --------------------------
 * Returns a list of successor states for state u, since this is an
 * 8-way graph this list contains all of a cells neighbours. Unless
 * the cell is occupied in which case it has no successors.
 */
void Dstar::getSucc(state u,list<state> &s) {
```

```
  s.clear();
  u.k.first  = -1;
  u.k.second = -1;

  if (occupied(u)) return;

  u.x += 1;
  s.push_front(u);
  u.y += 1;
  s.push_front(u);
  u.x -= 1;
  s.push_front(u);
  u.x -= 1;
  s.push_front(u);
  u.y -= 1;
  s.push_front(u);
  u.y -= 1;
  s.push_front(u);
  u.x += 1;
  s.push_front(u);
  u.x += 1;
  s.push_front(u);

}

/* void Dstar::getPred(state u,list<state> &s)
 * ---------------------------
 * Returns a list of all the predecessor states for state u. Since
 * this is for an 8-way connected graph the list contails all the
 * neighbours for state u. Occupied neighbours are not added to the
 * list.
 */
void Dstar::getPred(state u,list<state> &s) {

  s.clear();
  u.k.first  = -1;
  u.k.second = -1;

  u.x += 1;
  if (!occupied(u)) s.push_front(u);
  u.y += 1;
  if (!occupied(u)) s.push_front(u);
  u.x -= 1;
  if (!occupied(u)) s.push_front(u);
  u.x -= 1;
  if (!occupied(u)) s.push_front(u);
  u.y -= 1;
```

```
  if (!occupied(u)) s.push_front(u);
  u.y -= 1;
  if (!occupied(u)) s.push_front(u);
  u.x += 1;
  if (!occupied(u)) s.push_front(u);
  u.x += 1;
  if (!occupied(u)) s.push_front(u);

}

/* void Dstar::updateStart(int x, int y)
 * --------------------------
 * Update the position of the robot, this does not force a replan.
 */
void Dstar::updateStart(int x, int y) {

  s_start.x = x;
  s_start.y = y;

  k_m += heuristic(s_last,s_start);

  s_start = calculateKey(s_start);
  s_last  = s_start;

}

/* void Dstar::updateGoal(int x, int y)
 * --------------------------
 * This is somewhat of a hack, to change the position of the goal we
 * first save all of the non-empty on the map, clear the map, move
the
 * goal, and re-add all of non-empty cells. Since most of these cells
 * are not between the start and goal this does not seem to hurt
 * performance too much. Also it free's up a good deal of memory we
 * likely no longer use.
 */
void Dstar::updateGoal(int x, int y) {

  list< pair<ipoint2, double> > toAdd;
  pair<ipoint2, double> tp;

  ds_ch::iterator i;
  list< pair<ipoint2, double> >::iterator kk;

  for(i=cellHash.begin(); i!=cellHash.end(); i++) {
    if (!close(i->second.cost, C1)) {
      tp.first.x = i->first.x;
      tp.first.y = i->first.y;
```

```
      tp.second = i->second.cost;
      toAdd.push_back(tp);
    }
  }

  cellHash.clear();
  openHash.clear();

  while(!openList.empty())
    openList.pop();

  k_m = 0;

  s_goal.x  = x;
  s_goal.y  = y;

  cellInfo tmp;
  tmp.g = tmp.rhs =  0;
  tmp.cost = C1;

  cellHash[s_goal] = tmp;

  tmp.g = tmp.rhs = heuristic(s_start,s_goal);
  tmp.cost = C1;
  cellHash[s_start] = tmp;
  s_start = calculateKey(s_start);

  s_last = s_start;

  for (kk=toAdd.begin(); kk != toAdd.end(); kk++) {
    updateCell(kk->first.x, kk->first.y, kk->second);
  }


}

/* bool Dstar::replan()
 * --------------------------
 * Updates the costs for all cells and computes the shortest path to
 * goal. Returns true if a path is found, false otherwise. The path
is
 * computed by doing a greedy search over the cost+g values in each
 * cells. In order to get around the problem of the robot taking a
 * path that is near a 45 degree angle to goal we break ties based on
 *  the metric euclidean(state, goal) + euclidean(state,start).
 */
bool Dstar::replan() {
```

```
  path.clear();

  int res = computeShortestPath();
  //printf("res: %d ols: %d ohs: %d tk: [%f %f] sk: [%f %f] sgr:
(%f,%f)\n",res,openList.size(),openHash.size(),openList.top().k.first
,openList.top().k.second, s_start.k.first,
s_start.k.second,getRHS(s_start),getG(s_start));
  if (res < 0) {
    fprintf(stderr, "NO PATH TO GOAL\n");
    return false;
  }
  list<state> n;
  list<state>::iterator i;

  state cur = s_start;

  if (isinf(getG(s_start))) {
    fprintf(stderr, "NO PATH TO GOAL\n");
    return false;
  }

  while(cur != s_goal) {

    path.push_back(cur);
    getSucc(cur, n);

    if (n.empty()) {
      fprintf(stderr, "NO PATH TO GOAL\n");
      return false;
    }

    double cmin = INFINITY;
    double tmin;
    state smin;

    for (i=n.begin(); i!=n.end(); i++) {

      //if (occupied(*i)) continue;
      double val  = cost(cur,*i);
      double val2 = trueDist(*i,s_goal) + trueDist(s_start,*i); //
(Euclidean) cost to goal + cost to pred
      val += getG(*i);

      if (close(val,cmin)) {
        if (tmin > val2) {
          tmin = val2;
          cmin = val;
          smin = *i;
```

```
        }
      } else if (val < cmin) {
        tmin = val2;
        cmin = val;
        smin = *i;
      }
    }
    n.clear();
    cur = smin;
  }
  path.push_back(s_goal);
  return true;
}

#ifdef USE_OPEN_GL

void Dstar::draw() {

  ds_ch::iterator iter;
  ds_oh::iterator iter1;
  state t;

  list<state>::iterator iter2;

  glBegin(GL_QUADS);
  for(iter=cellHash.begin(); iter != cellHash.end(); iter++) {
    if (iter->second.cost == 1) glColor3f(0,1,0);
    else if (iter->second.cost < 0 ) glColor3f(1,0,0);
    else glColor3f(0,0,1);
    drawCell(iter->first,0.45);
  }

  glColor3f(1,1,0);
  drawCell(s_start,0.45);
  glColor3f(1,0,1);
  drawCell(s_goal,0.45);

  for(iter1=openHash.begin(); iter1 != openHash.end(); iter1++) {
    glColor3f(0.4,0,0.8);
    drawCell(iter1->first, 0.2);
  }


  glEnd();

  glLineWidth(4);
  glBegin(GL_LINE_STRIP);
  glColor3f(0.6, 0.1, 0.4);
```

```
  for(iter2=path.begin(); iter2 != path.end(); iter2++) {
    glVertex3f(iter2->x, iter2->y, 0.2);
  }
  glEnd();

}

void Dstar::drawCell(state s, float size) {

  float x = s.x;
  float y = s.y;


  glVertex2f(x - size, y - size);
  glVertex2f(x + size, y - size);
  glVertex2f(x + size, y + size);
  glVertex2f(x - size, y + size);


}

#else
void Dstar::draw() {}
void Dstar::drawCell(state s, float z) {}
#endif
```

Dstar.h

```
/*BY SOHAN VICHARE, BASED ON AND BUILT FROM
 * James Neufeld (neufeld@cs.ualberta.ca)
 * and Arek Sredzki (arek@sredzki.com)
 */

#ifndef DSTAR_H
#define DSTAR_H

#include <math.h>
#include <stack>
#include <queue>
#include <list>
#include <stdio.h>
#include <ext/hash_map>
```

```
using namespace std;
using namespace __gnu_cxx;

class state {
 public:
  int x;
  int y;
  int getX();
  int getY();
  pair<double,double> k;

  bool operator == (const state &s2) const {
    return ((x == s2.x) && (y == s2.y));
  }

  bool operator != (const state &s2) const {
    return ((x != s2.x) || (y != s2.y));
  }

  bool operator > (const state &s2) const {
    if (k.first-0.00001 > s2.k.first) return true;
    else if (k.first < s2.k.first-0.00001) return false;
    return k.second > s2.k.second;
  }

  bool operator <= (const state &s2) const {
    if (k.first < s2.k.first) return true;
    else if (k.first > s2.k.first) return false;
    return k.second < s2.k.second + 0.00001;
  }


  bool operator < (const state &s2) const {
    if (k.first + 0.000001 < s2.k.first) return true;
    else if (k.first - 0.000001 > s2.k.first) return false;
    return k.second < s2.k.second;
  }

};

struct ipoint2 {
  int x,y;
};

struct cellInfo {

  double g;
  double rhs;
```

```
  double cost;

};

class state_hash {
 public:
  size_t operator()(const state &s) const {
    return s.x + 34245*s.y;
  }
};


typedef priority_queue<state, vector<state>, greater<state> > ds_pq;
typedef hash_map<state,cellInfo, state_hash, equal_to<state> > ds_ch;
typedef hash_map<state, float, state_hash, equal_to<state> > ds_oh;


class Dstar {

 public:

  Dstar();
  void   init(int sX, int sY, int gX, int gY);
  void   updateCell(int x, int y, double val);
  void   updateStart(int x, int y);
  void   updateGoal(int x, int y);
  bool   replan();
  void   draw();
  void   drawCell(state s,float z);

  list<state> getPath();

 private:

  list<state> path;

  double C1;
  double k_m;
  state s_start, s_goal, s_last;
  int maxSteps;

  ds_pq openList;
  ds_ch cellHash;
  ds_oh openHash;

  bool   close(double x, double y);
  void   makeNewCell(state u);
  double getG(state u);
```

```
  double getRHS(state u);
  void   setG(state u, double g);
  double setRHS(state u, double rhs);
  double eightCondist(state a, state b);
  int    computeShortestPath();
  void   updateVertex(state u);
  void   insert(state u);
  void   remove(state u);
  double trueDist(state a, state b);
  double heuristic(state a, state b);
  state  calculateKey(state u);
  void   getSucc(state u, list<state> &s);
  void   getPred(state u, list<state> &s);
  double cost(state a, state b);
  bool   occupied(state u);
  bool   isValid(state u);
  float  keyHashCode(state u);
};

#endif
```

DstarImplement.cpp

```
 //PYTHON WRAPPER FOR DSTARLITE ALGORITHM

#include "Dstar.h"
#include <iostream>
#include <list>
#include "python.hpp"
using namespace std;

Dstar *dstar = new Dstar();
list<state> mypath;

void dStarInit(int startX, int startY, int goalX, int goalY)
{
  dstar->init(startX,startY,goalX,goalY);
}

void dStarUpdateGoal(int goalX, int goalY)
{
   dstar->updateGoal(goalX, goalY);
}
```

```
void dStarUpdateStart(int startX, int startY)
{
  dstar->updateStart(startX, startY);
}

void dStarAddBlock(int xCoor, int yCoor)
{
  dstar->updateCell(xCoor, yCoor, -1);
}

void dStarUpdateCell(int xCoor, int yCoor, int cost)
{
  dstar->updateCell(xCoor, yCoor, cost);
}

void dStarReplan()
{
  dstar->replan();
}

void dStarUpdatePathVar()
{
  mypath = dstar->getPath();
}

int dStarGetPathLength()
{
  mypath = dstar->getPath();
  return mypath.size();
}

int dStarGetXAtIndex(int index)
{
  int xToReturn;
  auto front = mypath.begin();
  for (int i = 0; i <= index; i++){
       std::list<state>::iterator it = mypath.begin();
       std::advance(it, i);
       xToReturn = it->getX();
  }
  return xToReturn;
}

int dStarGetYAtIndex(int index)
{
  int yToReturn;
  auto front = mypath.begin();
  for (int i = 0; i <= index; i++){
```

```
        std::list<state>::iterator it = mypath.begin();
        std::advance(it, i);
        yToReturn = it->getY();
  }
  return yToReturn;
}

#include <boost/python/module.hpp>
#include <boost/python/def.hpp>
using namespace boost::python;

BOOST_PYTHON_MODULE(dstarlite)
{
    def("init", dStarInit);
    def("addBlock", dStarAddBlock);
    def("replan", dStarReplan);
    def("updatePathVar", dStarUpdatePathVar);
    def("getPathLength", dStarGetPathLength);
    def("getXAtIndex", dStarGetXAtIndex);
    def("getYAtIndex", dStarGetYAtIndex);
    def("updateCell", dStarUpdateCell);
    def("updateGoal", dStarUpdateGoal);
    def("updateStart", dStarUpdateStart);
}
```

PathAlgorithm.py

```
# -*- coding: utf-8 -*-

#BY SOHAN VICHARE Adiyan Kaul 2015
#testing out dstarlite dynamic pathfinding algorithm and logic for
finding closest safe point

#import needed dependiences
from random import randint
import cv2
import math
import matplotlib.pyplot as plot
import matplotlib.patches as mpatches
import matplotlib.font_manager as fm
import dstarlite

#define location class that stores latitude and longitude
```

```
class Location:
   "Basic Location class, used to store latitude and longitude"

   def __init__(self, latitude, longitude):
      self.latitude = latitude
      self.longitude = longitude


#define Person class, which stores location data, closest safe point
data for a person
class Person:
    "Person class, used to store data for a person. Has two
attributes, location (which stores the person's location), and
safePointArray (which stores an array of safePoints in order of which
is closest to the location)"
    def __init__(self,location):
        self.location = location
        #an array of safe points, in order of what is closest to the
person
        self.safePointArray = [];

    #define function that finds the closes safe point to a given
location, returns an array of array indexes, where the first one is
the closest and so on
    def closestSafePoint(self, safePointLocationArray):
        #define array that will be returned
        closestPointIndexArray = [];
        usedIndexArray = [];
        closestDist = 1000;
        closestIndex = 0;

        for index, item in enumerate(safePointLocationArray):
            for ind, itm in enumerate(safePointLocationArray):
                if ind not in usedIndexArray:
                    dist = distanceBetweenLocations(itm.location,
self.location, "");
                    if dist <= closestDist:
                        closestDist = dist;
                        closestIndex = ind;

            if closestIndex not in usedIndexArray:
                usedIndexArray.append(closestIndex);
                closestPointIndexArray.append(closestIndex);
                closestDist = 1000;

        return closestPointIndexArray;
```

```
#define SafeLocation class, which stores location data, which people
are at each Safe Location
class SafeLocation:
    "Safe Location class, used to store data for a safe location. Has
two attributes, one is location (a location object which stores the
location), and the second is peopleArrivedArray, which is an array
that stores the people who have reached this location."

    def __init__(self, location):
        self.location = location
        #an array of Person objects, used to store what people are
already at the safeLocation
        self.peopleArrivedArray = [];

#define helper method to print arrays with locations so that they are
easy to see
def printLocationArray(locationArray):
    for item in locationArray:
        print item.latitude
        print item.longitude
        print " "

#define helper method to get an array of the locations object from an
array of a person, safelocation, or any other object that has
location as a property
def getLocationsArray(safeLocsArray):
        returnArray = []
        for item in safeLocsArray:
            returnArray.append(item.location);
        return returnArray;


#define helper class to plot a location array's points, takes in
array of location Arrays
def plotLocationArray(locationArray, color):
    xcoords = []
    ycoords = []
    for item in locationArray:
        xcoords.append(item.latitude);
        ycoords.append(item.longitude);
    plot.plot(xcoords,ycoords,color);
    plot.axis([0, 30, 0, 30])

#distance formula function, finds distance between two location
objects, accounting for the earth's spherical shape (assumes that
earth is a sphere)
def distanceBetweenLocations(location1, location2, distanceUnit):
```

```
    # Convert latitude and longitude to spherical coordinates in
radians.
    degreesToRadians = math.pi/180.0

    # phi = 90 - latitude
    phi1 = (90.0 - location1.latitude)*degreesToRadians
    phi2 = (90.0 - location2.latitude)*degreesToRadians

    # theta = longitude
    theta1 = location1.longitude*degreesToRadians
    theta2 = location2.longitude*degreesToRadians

    # Compute spherical distance from spherical coordinates.

    # For two locations in spherical coordinates
    # (1, theta, phi) and (1, theta', phi')
    # cosine( arc length ) =
    #    sin phi sin phi' cos(theta-theta') + cos phi cos phi'
    # distance = rho * arc length

    cos = (math.sin(phi1)*math.sin(phi2)*math.cos(theta1 - theta2) +
           math.cos(phi1)*math.cos(phi2))
    arc = math.acos( cos )

    # multiply and return arc by the right distance unit, miles or
kilometers
    unit = 1;
    if distanceUnit == "miles":
        unit = 3960
    if distanceUnit == "kilometers":
        unit = 6737

    return arc * unit;

#function to get a location array from dstarlite -> returns an array
of locations which is the steps for the path the drone will take
def dStarLiteGetLocationArray():
    arrToReturn = [];
    dstarlite.updatePathVar();
    pathLength = dstarlite.getPathLength();
    for getindex in range(0, pathLength):
        latitud = dstarlite.getXAtIndex(getindex);
        longitud = dstarlite.getYAtIndex(getindex);
        locashun = Location(latitud,longitud);
        arrToReturn.append(locashun);
    return arrToReturn
```

```
#initialize dstarlite at a random location and give it a random goal
plot.figure(2);
x1 = randint(-10, -8)
y1 = randint(-10, 8)
x2 = randint(8, 10)
y2 = randint(8, 10)
plot.axis([-11,11,-11,11])
plot.plot([x1,x2],[y1,y2],'ro');
dstarlite.init(x1, y1, x2, y2);

#add random obstacles
for x in range(0,90):
    blockX = randint(-10, 10)
    blockY = randint(-10, 10)
    dstarlite.addBlock(blockX, blockY);
    plot.plot([blockX],[blockY],'go');

#replan and update the path variable for dstar
dstarlite.replan();
dstarlite.updatePathVar();
pathArray = dStarLiteGetLocationArray();
xArr = [];
yArr = [];
for item in pathArray:
    xArr.append(item.latitude);
    yArr.append(item.longitude);
plot.plot(xArr,yArr,'b');

printLocationArray(dStarLiteGetLocationArray());

#define arrays that store safe location and person location variables
safeLocationArray = [];
personArray = [];

#change plot
plot.figure(1);

#populate personLocs
for x in range(0, 4):
    lat = randint(1,20)
    lon = randint(1,20)
    loc = Location(lat,lon);
    p = Person(loc)
    personArray.append(p);

#populate safeLocs
for x in range(0, 2):
    lat = randint(1,20)
```

```
    lon = randint(1,20)
    loc = Location(lat,lon);
    s = SafeLocation(loc);
    safeLocationArray.append(s);


#plot both arrays on the graph
plotLocationArray(getLocationsArray(personArray),'ro');
plotLocationArray(getLocationsArray(safeLocationArray),'bo');

#LOGIC TO FIND CLOSEST SAFE POINT TO EACH PERSON AND DRAW A LINE
BETWEEN THE RANDOMLY GENERATED POINTS
for item in personArray:
    colorsArray = ['m','k','y','c','g','r','b',]
    personLocation = item.location
    plot.plot(personLocation.latitude,personLocation.longitude,'bo')
    closestSafeLocIndexArray =
item.closestSafePoint(safeLocationArray);
    for index, item in enumerate(closestSafeLocIndexArray):
        closestSafeLoc = safeLocationArray[item];
        plot.plot(closestSafeLoc.location.latitude,
closestSafeLoc.location.longitude, 'ro');
        plot.plot([closestSafeLoc.location.latitude,
personLocation.latitude],[closestSafeLoc.location.longitude,
personLocation.longitude],colorsArray[index]);

#show plot and create legend
safe_points = mpatches.Patch(color='red', label='Red dots = Safe
Points (designated by user)')
person_points = mpatches.Patch(color='blue', label='Blue dots =
People (found by drone)')
line = mpatches.Patch(color="magenta", label='1st option')
line1 = mpatches.Patch(color="black", label='2nd option')
line2 = mpatches.Patch(color="yellow", label='3rd option')
line3 = mpatches.Patch(color="cyan", label='4th option')
line4 = mpatches.Patch(color="green", label='5th option')
line5 = mpatches.Patch(color="red", label='6th option')
line6 = mpatches.Patch(color="blue", label='7th option')
prop = fm.FontProperties(size=14)
plot.legend(handles=[safe_points, person_points, line, line1, line2,
line3, line4, line5, line6], prop=prop)
plot.show();
```

testflight1.py

```
from dronekit import connect, VehicleMode, LocationGlobalRelative
import time

# Connect to the Vehicle
print 'Connecting to vehicle;'
vehicle = connect("/dev/ttyACM0", wait_ready=True)

# vehicle is an instance of the Vehicle class
print "Global Location: %s" % vehicle.location.global_frame
print "Global Location (relative altitude): %s" %
vehicle.location.global_relative_frame
print "Local Location: %s" % vehicle.location.local_frame    #NED
print "Attitude: %s" % vehicle.attitude
print "Velocity: %s" % vehicle.velocity
print "GPS: %s" % vehicle.gps_0
print "Groundspeed: %s" % vehicle.groundspeed
print "Airspeed: %s" % vehicle.airspeed
print "Battery: %s" % vehicle.battery
print "EKF OK?: %s" % vehicle.ekf_ok
print "Last Heartbeat: %s" % vehicle.last_heartbeat
print "Rangefinder: %s" % vehicle.rangefinder
print "Rangefinder distance: %s" % vehicle.rangefinder.distance
print "Rangefinder voltage: %s" % vehicle.rangefinder.voltage
```

```
print "Heading: %s" % vehicle.heading
print "Is Armable?: %s" % vehicle.is_armable
print "System status: %s" % vehicle.system_status.state
print "Mode: %s" % vehicle.mode.name    # settable
print "Armed: %s" % vehicle.armed

def arm_and_takeoff(aTargetAltitude):
    """
    Arms vehicle and fly to aTargetAltitude.
    """

    print "Basic pre-arm checks"
    # Don't try to arm until autopilot is ready
    while not vehicle.is_armable:
        print " Waiting for vehicle to initialise..."
        time.sleep(1)


    print "Arming motors"
    # Copter should arm in GUIDED mode
    vehicle.mode    = VehicleMode("GUIDED")
    vehicle.armed   = True

    # Confirm vehicle armed before attempting to take off
    while not vehicle.armed:
        print " Waiting for arming..."
        time.sleep(1)

    print "Taking off!"
    vehicle.simple_takeoff(aTargetAltitude) # Take off to target
altitude

    # Wait until the vehicle reaches a safe height before processing
the goto (otherwise the command
    #  after Vehicle.simple_takeoff will execute immediately).
    while True:
        print " Altitude: ",
vehicle.location.global_relative_frame.alt
        #Break and return from function just below target altitude.
        if
vehicle.location.global_relative_frame.alt>=aTargetAltitude*0.95:
            print "Reached target altitude"
            break
        time.sleep(1)

def countdown(amtTime):
    i = 0
    while i <= amtTime:
```

```
        print("COUNTDOWN: "+str(amtTime-i))
        time.sleep(1)
        i = i+1

def countdownAlt(amTime):
    o = 0
    while o <= amTime:
        print("COUNTDOWN: "+str(amTime-o))
        print " Altitude: ",
vehicle.location.global_relative_frame.alt
        time.sleep(1)
        o = o+1

countdown(20)

print "Set default/target airspeed to 1"
vehicle.airspeed=1
vehicle.groundspeed =1

arm_and_takeoff(1)

print "Returning to Launch"
vehicle.mode    = VehicleMode("LAND")
countdownAlt(30)

#Close vehicle object before exiting script
print "Close vehicle object"
vehicle.close()
```

testflight2.py

```
# Import DroneKit-Python
from dronekit import connect, VehicleMode
import time

# Connect to the Vehicle.
print "Connecting to vehicle on: 'tcp:127.0.0.1:5760'"
vehicle = connect("/dev/ttyACM0", wait_ready=True)
from dronekit import connect, VehicleMode, LocationGlobalRelative
from pymavlink import mavutil # Needed for command message
definitions
import time

# vehicle is an instance of the Vehicle class
```

```
print "Global Location: %s" % vehicle.location.global_frame
print "Global Location (relative altitude): %s" %
vehicle.location.global_relative_frame
print "Local Location: %s" % vehicle.location.local_frame    #NED
print "Attitude: %s" % vehicle.attitude
print "Velocity: %s" % vehicle.velocity
print "GPS: %s" % vehicle.gps_0
print "Groundspeed: %s" % vehicle.groundspeed
print "Airspeed: %s" % vehicle.airspeed
print "Battery: %s" % vehicle.battery
print "EKF OK?: %s" % vehicle.ekf_ok
print "Last Heartbeat: %s" % vehicle.last_heartbeat
print "Rangefinder: %s" % vehicle.rangefinder
print "Rangefinder distance: %s" % vehicle.rangefinder.distance
print "Rangefinder voltage: %s" % vehicle.rangefinder.voltage
print "Heading: %s" % vehicle.heading
print "Is Armable?: %s" % vehicle.is_armable
print "System status: %s" % vehicle.system_status.state
print "Mode: %s" % vehicle.mode.name    # settable
print "Armed: %s" % vehicle.armed

def arm_and_takeoff(aTargetAltitude):
    """
    Arms vehicle and fly to aTargetAltitude.
    """

    print "Basic pre-arm checks"
    # Don't try to arm until autopilot is ready
    while not vehicle.is_armable:
        print " Waiting for vehicle to initialise..."
        time.sleep(1)


    print "Arming motors"
    # Copter should arm in GUIDED mode
    vehicle.mode    = VehicleMode("GUIDED")
    vehicle.armed   = True

    # Confirm vehicle armed before attempting to take off
    while not vehicle.armed:
        print " Waiting for arming..."
        time.sleep(1)

    print "Taking off!"
    vehicle.simple_takeoff(aTargetAltitude) # Take off to target
altitude
```

```
    # Wait until the vehicle reaches a safe height before processing
the goto (otherwise the command
    #  after Vehicle.simple_takeoff will execute immediately).
    while True:
        print " Altitude: ",
vehicle.location.global_relative_frame.alt
        #Break and return from function just below target altitude.
        if
vehicle.location.global_relative_frame.alt>=aTargetAltitude*0.95:
            print "Reached target altitude"
            break
        time.sleep(1)

def countdown(amtTime):
    i = 0
    while i <= amtTime:
        print("COUNTDOWN: "+str(amtTime-i))
        time.sleep(1)
        i = i+1

def countdownAlt(amTime):
    o = 0
    while o <= amTime:
        print("COUNTDOWN: "+str(amTime-o))
        print " Altitude: ",
vehicle.location.global_relative_frame.alt
        time.sleep(1)
        o = o+1

def countdownLoc(amTime):
    o = 0
    while o <= amTime:
        print("COUNTDOWN: "+str(amTime-o))
        print " Altitude: ", vehicle.location.global_relative_frame
        time.sleep(1)
        o = o+1

def moveVehicle(velocity_x, velocity_y, velocity_z, duration):
    """
    Move vehicle in direction based on specified velocity vectors.
    """
    msg =
vehicle.message_factory.set_position_target_local_ned_encode(
        0,       # time_boot_ms (not used)
        0, 0,    # target system, target component
        mavutil.mavlink.MAV_FRAME_LOCAL_NED, # frame
        0b0000111111000111, # type_mask (only speeds enabled)
        0, 0, 0, # x, y, z positions (not used)
```

```
        velocity_x, velocity_y, velocity_z, # x, y, z velocity in m/s
        0, 0, 0, # x, y, z acceleration (not supported yet, ignored
in GCS_Mavlink)
        0, 0)    # yaw, yaw_rate (not supported yet, ignored in
GCS_Mavlink)


    # send command to vehicle on 1 Hz cycle
    for x in range(0,duration):
        vehicle.send_mavlink(msg)
        time.sleep(1)

countdown(20)

print "Set default/target airspeed to 1"
vehicle.airspeed=1
vehicle.groundspeed =1

arm_and_takeoff(1)
#move the vehicle north for 5 seconds at a speed of 1 m/s
moveVehicle(1,0,0,5)
countdownLoc(5)

print "Returning to Launch"
vehicle.mode    = VehicleMode("LAND")
countdownAlt(30)

#Close vehicle object before exiting script
print "Close vehicle object"
vehicle.close()
```

testflight3.py

```
from dronekit import connect, VehicleMode, LocationGlobalRelative
from pymavlink import mavutil # Needed for command message
definitions
import time
from picamera.array import PiRGBArray
from picamera import PiCamera
import cv2
import numpy as np
```

```
# Connect to the Vehicle
print 'Connecting to vehicle;'
vehicle = connect("/dev/ttyACM0", wait_ready=True)

# vehicle is an instance of the Vehicle class
print "Global Location: %s" % vehicle.location.global_frame
print "Global Location (relative altitude): %s" %
vehicle.location.global_relative_frame
print "Local Location: %s" % vehicle.location.local_frame    #NED
print "Attitude: %s" % vehicle.attitude
print "Velocity: %s" % vehicle.velocity
print "GPS: %s" % vehicle.gps_0
print "Groundspeed: %s" % vehicle.groundspeed
print "Airspeed: %s" % vehicle.airspeed
print "Battery: %s" % vehicle.battery
print "EKF OK?: %s" % vehicle.ekf_ok
print "Last Heartbeat: %s" % vehicle.last_heartbeat
print "Rangefinder: %s" % vehicle.rangefinder
print "Rangefinder distance: %s" % vehicle.rangefinder.distance
print "Rangefinder voltage: %s" % vehicle.rangefinder.voltage
print "Heading: %s" % vehicle.heading
print "Is Armable?: %s" % vehicle.is_armable
print "System status: %s" % vehicle.system_status.state
print "Mode: %s" % vehicle.mode.name    # settable
print "Armed: %s" % vehicle.armed
camera = PiCamera()
rawCapture = PiRGBArray(camera)
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())


def arm_and_takeoff(aTargetAltitude):
    """
    Arms vehicle and fly to aTargetAltitude.
    """

    print "Basic pre-arm checks"
    # Don't try to arm until autopilot is ready
    while not vehicle.is_armable:
        print " Waiting for vehicle to initialise..."
        time.sleep(1)


    print "Arming motors"
    # Copter should arm in GUIDED mode
    vehicle.mode    = VehicleMode("GUIDED")
    vehicle.armed   = True
```

```
    # Confirm vehicle armed before attempting to take off
    while not vehicle.armed:
        print " Waiting for arming..."
        time.sleep(1)

    print "Taking off!"
    vehicle.simple_takeoff(aTargetAltitude) # Take off to target
altitude

    # Wait until the vehicle reaches a safe height before processing
the goto (otherwise the command
    #  after Vehicle.simple_takeoff will execute immediately).
    while True:
        print " Altitude: ",
vehicle.location.global_relative_frame.alt
        #Break and return from function just below target altitude.
        if
vehicle.location.global_relative_frame.alt>=aTargetAltitude*0.95:
            print "Reached target altitude"
            break
        time.sleep(1)

def countdown(amtTime):
    i = 0
    while i <= amtTime:
        print("COUNTDOWN: "+str(amtTime-i))
        time.sleep(1)
        i = i+1

def countdownAlt(amTime):
    o = 0
    while o <= amTime:
        print("COUNTDOWN: "+str(amTime-o))
        print " Altitude: ",
vehicle.location.global_relative_frame.alt
        time.sleep(1)
        o = o+1
def countdownLoc(amTime):
    o = 0
    while o <= amTime:
        print("COUNTDOWN: "+str(amTime-o))
        print " Altitude: ", vehicle.location.global_relative_frame
        time.sleep(1)
        o = o+1

def moveVehicle(velocity_x, velocity_y, velocity_z, duration):
    """
    Move vehicle in direction based on specified velocity vectors.
```

```
    """
    msg =
vehicle.message_factory.set_position_target_local_ned_encode(
        0,       # time_boot_ms (not used)
        0, 0,    # target system, target component
        mavutil.mavlink.MAV_FRAME_LOCAL_NED, # frame
        0b0000111111000111, # type_mask (only speeds enabled)
        0, 0, 0, # x, y, z positions (not used)
        velocity_x, velocity_y, velocity_z, # x, y, z velocity in m/s
        0, 0, 0, # x, y, z acceleration (not supported yet, ignored
in GCS_Mavlink)
        0, 0)    # yaw, yaw_rate (not supported yet, ignored in
GCS_Mavlink)


    # send command to vehicle on 1 Hz cycle
    for x in range(0,duration):
        vehicle.send_mavlink(msg)
        time.sleep(1)

def takePicture():
    camera.capture(rawCapture, format="bgr")
    image = rawCapture.array
    return image


# gets the number of people in the image
def readImage(image):
    (rects, weights) = hog.detectMultiScale(image, winStride=(4, 4),
     padding=(8, 8), scale=1.05)
    count=0
    for (xA, yA, xB, yB) in rects:
         count=count+1
    return count

def findNumPeople():
    img =takePicture();
    people= readImage(img)
    return people


countdown(20)

print "Set default/target airspeed to 1"
vehicle.airspeed=1
vehicle.groundspeed =1

arm_and_takeoff(1)
```

```
moveVehicle(0.5,1,0,2)
countdownLoc(2);

count = findNumPeople()
countdown(1)
print(count)


print "Returning to Launch"
vehicle.mode    = VehicleMode("LAND")
countdownAlt(30)

#Close vehicle object before exiting script
print "Close vehicle object"
vehicle.close()
```

testflight4.py

```
from dronekit import connect, VehicleMode,
LocationGlobalRelative,LocationGlobal
from pymavlink import mavutil # Needed for command message
definitions
import time
from picamera.array import PiRGBArray
from picamera import PiCamera
import cv2
import numpy as np
import math



# Connect to the Vehicle
print 'Connecting to vehicle;'
vehicle = connect("/dev/ttyACM0", wait_ready=True)

# vehicle is an instance of the Vehicle class
print "Global Location: %s" % vehicle.location.global_frame
```

```
print "Global Location (relative altitude): %s" %
vehicle.location.global_relative_frame
print "Local Location: %s" % vehicle.location.local_frame    #NED
print "Attitude: %s" % vehicle.attitude
print "Velocity: %s" % vehicle.velocity
print "GPS: %s" % vehicle.gps_0
print "Groundspeed: %s" % vehicle.groundspeed
print "Airspeed: %s" % vehicle.airspeed
print "Battery: %s" % vehicle.battery
print "EKF OK?: %s" % vehicle.ekf_ok
print "Last Heartbeat: %s" % vehicle.last_heartbeat
print "Rangefinder: %s" % vehicle.rangefinder
print "Rangefinder distance: %s" % vehicle.rangefinder.distance
print "Rangefinder voltage: %s" % vehicle.rangefinder.voltage
print "Heading: %s" % vehicle.heading
print "Is Armable?: %s" % vehicle.is_armable
print "System status: %s" % vehicle.system_status.state
print "Mode: %s" % vehicle.mode.name    # settable
print "Armed: %s" % vehicle.armed
camera = PiCamera()
rawCapture = PiRGBArray(camera)
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())


def arm_and_takeoff(aTargetAltitude):
    """
    Arms vehicle and fly to aTargetAltitude.
    """

    print "Basic pre-arm checks"
    # Don't try to arm until autopilot is ready
    while not vehicle.is_armable:
        print " Waiting for vehicle to initialise..."
        time.sleep(1)


    print "Arming motors"
    # Copter should arm in GUIDED mode
    vehicle.mode    = VehicleMode("GUIDED")
    vehicle.armed   = True

    # Confirm vehicle armed before attempting to take off
    while not vehicle.armed:
        print " Waiting for arming..."
        time.sleep(1)

    print "Taking off!"
```

```
    vehicle.simple_takeoff(aTargetAltitude) # Take off to target
altitude

    # Wait until the vehicle reaches a safe height before processing
the goto (otherwise the command
    #  after Vehicle.simple_takeoff will execute immediately).
    while True:
        print " Altitude: ",
vehicle.location.global_relative_frame.alt
        #Break and return from function just below target altitude.
        if
vehicle.location.global_relative_frame.alt>=aTargetAltitude*0.95:
            print "Reached target altitude"
            break
        time.sleep(1)

def countdown(amtTime):
    i = 0
    while i <= amtTime:
        print("COUNTDOWN: "+str(amtTime-i))
        time.sleep(1)
        i = i+1

def countdownAlt(amTime):
    o = 0
    while o <= amTime:
        print("COUNTDOWN: "+str(amTime-o))
        print " Altitude: ",
vehicle.location.global_relative_frame.alt
        time.sleep(1)
        o = o+1
def moveVehicle(velocity_x, velocity_y, velocity_z, duration):
    """
    Move vehicle in direction based on specified velocity vectors.
    """
    msg =
vehicle.message_factory.set_position_target_local_ned_encode(
        0,       # time_boot_ms (not used)
        0, 0,    # target system, target component
        mavutil.mavlink.MAV_FRAME_LOCAL_NED, # frame
        0b0000111111000111, # type_mask (only speeds enabled)
        0, 0, 0, # x, y, z positions (not used)
        velocity_x, velocity_y, velocity_z, # x, y, z velocity in m/s
        0, 0, 0, # x, y, z acceleration (not supported yet, ignored
in GCS_Mavlink)
        0, 0)    # yaw, yaw_rate (not supported yet, ignored in
GCS_Mavlink)
```

```
    # send command to vehicle on 1 Hz cycle
    for x in range(0,duration):
        vehicle.send_mavlink(msg)
        time.sleep(1)

def goToLocation(targetLocation, gotoFunction=vehicle.simple_goto):
    currentLocation=vehicle.location.global_relative_frame
    targetDistance=get_distance_metres(currentLocation,
targetLocation)
    gotoFunction(targetLocation)

    while vehicle.mode.name=="GUIDED": #Stop action if we are no
longer in guided mode.

remainingDistance=get_distance_metres(vehicle.location.global_frame,
targetLocation)
        print "Distance to target: ", remainingDistance
        if remainingDistance<=targetDistance*0.01: #Just below
target, in case of undershoot.
            print "Reached target"
            break;
        time.sleep(2)
def get_distance_metres(aLocation1, aLocation2):
    """
    Returns the ground distance in metres between two LocationGlobal
objects.

    This method is an approximation, and will not be accurate over
large distances and close to the
    earth's poles. It comes from the ArduPilot test code:

https://github.com/diydrones/ardupilot/blob/master/Tools/autotest/com
mon.py
    """
    dlat = aLocation2.lat - aLocation1.lat
    dlong = aLocation2.lon - aLocation1.lon
    return math.sqrt((dlat*dlat) + (dlong*dlong)) * 1.113195e5

def takePicture():
    camera.capture(rawCapture, format="bgr")
    image = rawCapture.array
    return image


# gets the number of people in the image
def readImage(image):
    (rects, weights) = hog.detectMultiScale(image, winStride=(4, 4),
```

```
      padding=(8, 8), scale=1.05)
    count=0
    for (xA, yA, xB, yB) in rects:
         count=count+1
    return count

def findNumPeople():
    img =takePicture();
    people= readImage(img)
    return people


countdown(20)

print "Set default/target airspeed to 1"
vehicle.airspeed=1
vehicle.groundspeed =1

arm_and_takeoff(1)
moveVehicle(1,0,0,5)

count = findNumPeople()
print(count)

lat = 0;
lon = 0;
alt = 1;
a_location = LocationGlobal(lat, lon, alt)

if(count>1):
    goToLocation(a_location)


print "Returning to Launch"
vehicle.mode    = VehicleMode("LAND")
countdownAlt(30)

#Close vehicle object before exiting script
print "Close vehicle object"
vehicle.close()
```