Code Supplements

This document acts as a code supplement for the ideas described in the main report.

SIFT detection in python and OpenCV

```
# Important NOTE: Use opencv <= 3.4.2.16 as
# SIFT is no longer available in
# opencv > 3.4.2.16
import cv2

# Loading the image
# alternatively you can load your camera frame here
img = cv2.imread('geeks.jpg')

# Converting image to grayscale
gray= cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

# Applying SIFT detector
sift = cv2.xfeatures2d.SIFT_create()
kp = sift.detect(gray, None)

# Marking the keypoint on the image using circles
img=cv2.drawKeypoints(gray ,
          kp ,
          img ,
          flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

cv2.imwrite('image-with-keypoints.jpg', img)
```

ORB feature detectors

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('simple.jpg',0)
# Initiate ORB detector
orb = cv.ORB_create()
# find the keypoints with ORB
kp = orb.detect(img,None)
# compute the descriptors with ORB
kp, des = orb.compute(img, kp)
# draw only keypoints location,not size and orientation
img2 = cv.drawKeypoints(img, kp, None, color=(0,255,0), flags=0)
plt.imshow(img2), plt.show()
```

Feature matching in OpenCV using ORB

```
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt
```

```
# instead of img1 and img2 being different images
# you can use them to store successive frames of
# the camera feed
img1 = cv.imread('box.png',cv.IMREAD_GRAYSCALE)          # queryImage
img2 = cv.imread('box_in_scene.png',cv.IMREAD_GRAYSCALE) # trainImage
# Initiate ORB detector
orb = cv.ORB_create()
# find the keypoints and descriptors with ORB
kp1, des1 = orb.detectAndCompute(img1,None)
kp2, des2 = orb.detectAndCompute(img2,None)

# create BFMatcher object
bf = cv.BFMatcher(cv.NORM_HAMMING, crossCheck=True)
# Match descriptors.
matches = bf.match(des1,des2)
# Sort them in the order of their distance.
matches = sorted(matches, key = lambda x:x.distance)
# Draw first 10 matches.
img3 = cv.drawMatches(img1,kp1,img2,kp2,matches[:10],None,flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
plt.imshow(img3),plt.show()
```

Trackers

```
import cv2
import sys

(major_ver, minor_ver, subminor_ver) = (cv2.__version__).split('.')

if __name__ == '__main__' :

    # Set up tracker.
    # Instead of MIL, you can also use

    tracker_types = ['BOOSTING', 'MIL','KCF', 'TLD', 'MEDIANFLOW', 'GOTURN', 'MOSSE', 'CSRT']
    tracker_type = tracker_types[2]

    if int(minor_ver) < 3:
        tracker = cv2.Tracker_create(tracker_type)
    else:
        if tracker_type == 'BOOSTING':
            tracker = cv2.TrackerBoosting_create()
        if tracker_type == 'MIL':
            tracker = cv2.TrackerMIL_create()
        if tracker_type == 'KCF':
            tracker = cv2.TrackerKCF_create()
        if tracker_type == 'TLD':
            tracker = cv2.TrackerTLD_create()
        if tracker_type == 'MEDIANFLOW':
            tracker = cv2.TrackerMedianFlow_create()
        if tracker_type == 'GOTURN':
            tracker = cv2.TrackerGOTURN_create()
        if tracker_type == 'MOSSE':
            tracker = cv2.TrackerMOSSE_create()
        if tracker_type == "CSRT":
            tracker = cv2.TrackerCSRT_create()

    # Read video
    video = cv2.VideoCapture("videos/chaplin.mp4")

    # Exit if video not opened.
    if not video.isOpened():
        print "Could not open video"
        sys.exit()

    # Read first frame.
    ok, frame = video.read()
    if not ok:
        print 'Cannot read video file'
        sys.exit()

    # Define an initial bounding box
    bbox = (287, 23, 86, 320)

    # Uncomment the line below to select a different bounding box
    bbox = cv2.selectROI(frame, False)

    # Initialize tracker with first frame and bounding box
    ok = tracker.init(frame, bbox)

    while True:
```

```
        # Read a new frame
        ok, frame = video.read()
        if not ok:
            break

        # Start timer
        timer = cv2.getTickCount()

        # Update tracker
        ok, bbox = tracker.update(frame)

        # Calculate Frames per second (FPS)
        fps = cv2.getTickFrequency() / (cv2.getTickCount() - timer);

        # Draw bounding box
        if ok:
            # Tracking success
            p1 = (int(bbox[0]), int(bbox[1]))
            p2 = (int(bbox[0] + bbox[2]), int(bbox[1] + bbox[3]))
            cv2.rectangle(frame, p1, p2, (255,0,0), 2, 1)
        else :
            # Tracking failure
            cv2.putText(frame, "Tracking failure detected", (100,80), cv2.FONT_HERSHEY_SIMPLEX, 0.75,(0,0,255),2)

        # Display tracker type on frame
        cv2.putText(frame, tracker_type + " Tracker", (100,20), cv2.FONT_HERSHEY_SIMPLEX, 0.75, (50,170,50),2);

        # Display FPS on frame
        cv2.putText(frame, "FPS : " + str(int(fps)), (100,50), cv2.FONT_HERSHEY_SIMPLEX, 0.75, (50,170,50), 2);

        # Display result
        cv2.imshow("Tracking", frame)

        # Exit if ESC pressed
        k = cv2.waitKey(1) & 0xff
        if k == 27 : break
```

Evasive maneuvres based on tracking output

This Pseudo code block summarises the second DAA suggestion. Each part of the Pseudocode can be found above as code snippets

```
# Assuming you have a history of tracked frames from the previous step
# Pseudocode to take evasive measures

# Detect suitable Orb/SIFT features from the frame that could be theorised as obstacles
# Retrieve ROI (Regions of Interest) from the detected keypoints. This could be done by just taking a random rectangle around the keyp
# Pass this ROI into the aforementioned trackers to track the obstacles
# if there are multiple detections, use each detection innitialise a separate tracker.
# This would give you individual tracklets for each detection.
# Construct a velocity vector from the past n frames of each tracklet.
# Utilise the velocity estimate of each tracklet to plan your evasive maneuvres.
```

Background subtraction based Detection

Assuming you have opened your camera and started streaming frames

```
# convert the frames to grayscale
grayA = cv2.cvtColor(prev_frame, cv2.COLOR_BGR2GRAY)
grayB = cv2.cvtColor(current_frame, cv2.COLOR_BGR2GRAY)

# plot the image after frame differencing
plt.imshow(cv2.absdiff(grayB, grayA), cmap = 'gray')
plt.show()


diff_image = cv2.absdiff(grayB, grayA)

# perform image thresholding
ret, thresh = cv2.threshold(diff_image, 30, 255, cv2.THRESH_BINARY)

# plot image after thresholding
```

```
plt.imshow(thresh, cmap = 'gray')
plt.show()


# apply image dilation
kernel = np.ones((3,3),np.uint8)
dilated = cv2.dilate(thresh,kernel,iterations = 1)

# plot dilated image
plt.imshow(dilated, cmap = 'gray')
plt.show()

# plot vehicle detection zone
plt.imshow(dilated)
cv2.line(dilated, (0, 80),(256,80),(100, 0, 0))
plt.show()

# get contours of the detection
valid_cntrs = []

for i,cntr in enumerate(contours):
    x,y,w,h = cv2.boundingRect(cntr)
    if (x <= 200) & (y >= 80) & (cv2.contourArea(cntr) >= 25):
        valid_cntrs.append(cntr)

# count of discovered contours
len(valid_cntrs)

dmy = col_images[13].copy()

cv2.drawContours(dmy, valid_cntrs, -1, (127,200,0), 2)
cv2.line(dmy, (0, 80),(256,80),(100, 255, 255))
plt.imshow(dmy)
plt.show()
```

Use the contours derived in the detection step as your regions of interest in your tracking step.

Optical Flow based Detection and Tracking

This code snippet acts as a supplement POC for the first DAA suggestion in the report. The script already gives you the direction of motion of the foreground object versus background. And so you can design your drone control to do the exact opposite of the foreground motion. I would advise running this script with the laptop webcam and just waving your hand around slowly. You can move it around faster and see if tracks motion. The direction of the track is printed on the screen. My suggestion is to go with this one as it is very robust, up to certain speeds of the foreground object. So this can be used for evasive actions.

```
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt

from scipy.stats import mode
from argparse import ArgumentParser

if __name__ == '__main__':
    ap = ArgumentParser()
    ap.add_argument('-rec', '--record', default=False, action='store_true', help='Record?')
    ap.add_argument('-pscale', '--pyr_scale', default=0.5, type=float,
                    help='Image scale (<1) to build pyramids for each image')
    ap.add_argument('-l', '--levels', default=3, type=int, help='Number of pyramid layers')
    ap.add_argument('-w', '--winsize', default=15, type=int, help='Averaging window size')
    ap.add_argument('-i', '--iterations', default=3, type=int,
                    help='Number of iterations the algorithm does at each pyramid level')
    ap.add_argument('-pn', '--poly_n', default=5, type=int,
                    help='Size of the pixel neighborhood used to find polynomial expansion in each pixel')
    ap.add_argument('-psigma', '--poly_sigma', default=1.1, type=float,
                    help='Standard deviation of the Gaussian that is used to smooth derivatives used as a basis for the polynomial exp
    ap.add_argument('-th', '--threshold', default=10.0, type=float, help='Threshold value for magnitude')
    ap.add_argument('-p', '--plot', default=False, action='store_true', help='Plot accumulators?')
    ap.add_argument('-rgb', '--rgb', default=False, action='store_true', help='Show RGB mask?')
    ap.add_argument('-s', '--size', default=10, type=int, help='Size of accumulator for directions map')

    args = vars(ap.parse_args())

    directions_map = np.zeros([args['size'], 5])
```

```
cap = cv.VideoCapture(0)
if args['record']:
    h = int(cap.get(cv.CAP_PROP_FRAME_HEIGHT))
    w = int(cap.get(cv.CAP_PROP_FRAME_WIDTH))
    codec = cv.VideoWriter_fourcc(*'MPEG')
    out = cv.VideoWriter('out.avi', codec, 10.0, (w, h))

if args['plot']:
    plt.ion()

frame_previous = cap.read()[1]
gray_previous = cv.cvtColor(frame_previous, cv.COLOR_BGR2GRAY)
hsv = np.zeros_like(frame_previous)
hsv[:, :, 1] = 255
param = {
    'pyr_scale': args['pyr_scale'],
    'levels': args['levels'],
    'winsize': args['winsize'],
    'iterations': args['iterations'],
    'poly_n': args['poly_n'],
    'poly_sigma': args['poly_sigma'],
    'flags': cv.OPTFLOW_LK_GET_MIN_EIGENVALS
}

while True:
    grabbed, frame = cap.read()
    if not grabbed:
        break

    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    flow = cv.calcOpticalFlowFarneback(gray_previous, gray, None, **param)
    mag, ang = cv.cartToPolar(flow[:, :, 0], flow[:, :, 1], angleInDegrees=True)
    ang_180 = ang/2
    gray_previous = gray

    move_sense = ang[mag > args['threshold']]
    move_mode = mode(move_sense)[0]

    if 10 < move_mode <= 100:
        directions_map[-1, 0] = 1
        directions_map[-1, 1:] = 0
        directions_map = np.roll(directions_map, -1, axis=0)
    elif 100 < move_mode <= 190:
        directions_map[-1, 1] = 1
        directions_map[-1, :1] = 0
        directions_map[-1, 2:] = 0
        directions_map = np.roll(directions_map, -1, axis=0)
    elif 190 < move_mode <= 280:
        directions_map[-1, 2] = 1
        directions_map[-1, :2] = 0
        directions_map[-1, 3:] = 0
        directions_map = np.roll(directions_map, -1, axis=0)
    elif 280 < move_mode or move_mode < 10:
        directions_map[-1, 3] = 1
        directions_map[-1, :3] = 0
        directions_map[-1, 4:] = 0
        directions_map = np.roll(directions_map, -1, axis=0)
    else:
        directions_map[-1, -1] = 1
        directions_map[-1, :-1] = 0
        directions_map = np.roll(directions_map, 1, axis=0)

    if args['plot']:
        plt.clf()
        plt.plot(directions_map[:, 0], label='Down')
        plt.plot(directions_map[:, 1], label='Right')
        plt.plot(directions_map[:, 2], label='Up')
        plt.plot(directions_map[:, 3], label='Left')
        plt.plot(directions_map[:, 4], label='Waiting')
        plt.legend(loc=2)
        plt.pause(1e-5)
        plt.show()

    loc = directions_map.mean(axis=0).argmax()
    if loc == 0:
        text = 'Moving down'
    elif loc == 1:
        text = 'Moving to the right'
    elif loc == 2:
        text = 'Moving up'
    elif loc == 3:
        text = 'Moving to the left'
    else:
        text = 'WAITING'

    hsv[:, :, 0] = ang_180
```

```
        hsv[:, :, 2] = cv.normalize(mag, None, 0, 255, cv.NORM_MINMAX)
        rgb = cv.cvtColor(hsv, cv.COLOR_HSV2BGR)

        frame = cv.flip(frame, 1)
        cv.putText(frame, text, (30, 90), cv.FONT_HERSHEY_COMPLEX, frame.shape[1] / 500, (0, 0, 255), 2)

        k = cv.waitKey(1) & 0xff
        if k == ord('q'):
            break
        if args['record']:
            out.write(frame)
        if args['rgb']:
            cv.imshow('Mask', rgb)
        cv.imshow('Frame', frame)
        k = cv.waitKey(1) & 0xff
        if k == ord('q'):
            break

    cap.release()
    if args['record']:
        out.release()
    if args['plot']:
        plt.ioff()
    cv.destroyAllWindows()
```