Introduction

The general objective of this project was to determine how to manipulate images received from a camera at points all around itself and narrow down the data to find all objects of a solid color. Our chosen color was neon green for tennis balls to make it easier for the gimbal to select those specific HSV values. Essential to this goal was the use of Haar cascade object detection to detect objects of interest which were tennis balls in our case. Upon the identification of a tennis ball, our program would draw an ellipse at the location of the ball, giving us a bounded area within our full image where the ball is located, knowing this location would allow us to perform further manipulations on the image at the current point of view. A user should expect the program to detect multiple tennis balls on the screen and draw ellipses around each ball it detects. To allow the gimbal to both manipulate the image data to find objects and move the motors toward the object, the CPU needs to have a single process running both the controller code and detector code simultaneously. To do this, we use a first thread for the controller and create a second thread for the detector from the system code. They also need to share data between them so that the controller knows from the detector where exactly the objects are. For this purpose, we create a shared object that is used in both threads. However, to implement this correctly and avoid memory issues, we need a lock for the process. Otherwise, we will run into hard-to-fix errors and bugs. For example, if both threads were accessing the shared object at the same time, then the controller could find that there are three tuples within the deltas list and the detector could update it right after to an empty list, but the controller would still attempt to access all three items in the list. Hence, the lock provides a sort of security within the integrated system that protects it from distorting data.

Shared Data Structure

The shared data structure has five variables: the lock, the stop flag, the frame counter, new-data flag, and the deltas list. The lock variable, as described above, is used to prevent both threads from accessing the shared object at the same time to prevent memory issues. Secondly, the stop flag is a boolean variable used by the system code to tell the detector to stop using the camera. The frame counter variable is a now deprecated variable used by the controller to know if there has not been an object in the past ten frames or more. We initially used this counter variable when we were only sending the largest object in-frame from the detector to the controller. We have since begun sending all objects from the detector to the controller, so this variable became obsolete, as can be seen by its lack of usage in the detector and controller (we just forgot to comment it out in the class definition). Moreover, the new-data variable is a boolean variable used to tell the controller if there is new data from the detector to read in. Lastly, the deltas variable consists of a list containing the coordinates of all neon green objects (mostly only tennis balls) currently in the frame. This is what the controller uses to orient the gimbal toward the objects.

Flowcharts

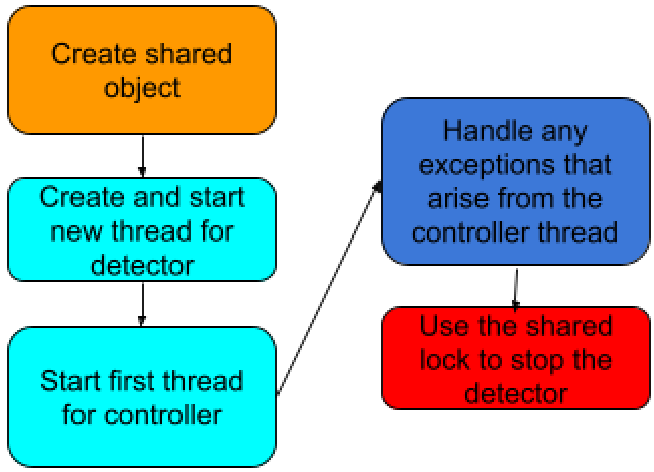

Figure 1: System code flowchart

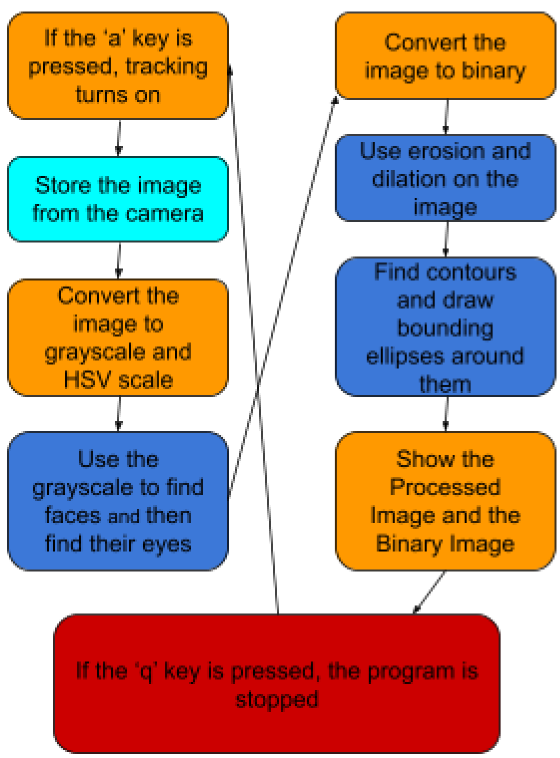

Figure 2: Detector code flowchart

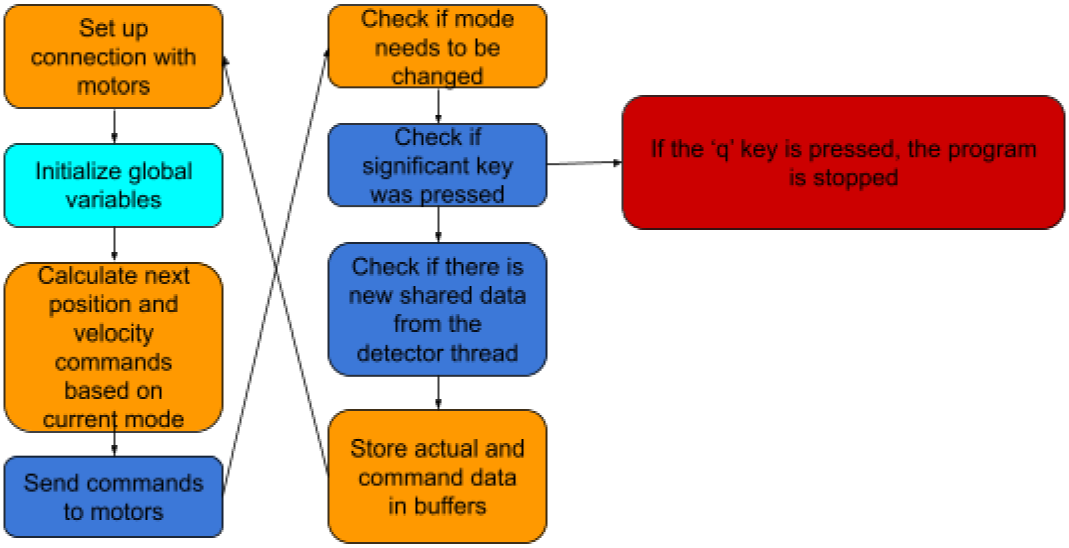

Figure 3: Controller code flowchart

The detector utilizes the shared object to send over the object data by assigning a list of object coordinates to the deltas variable within the shared object. The controller then receives this data and uses it to add to its objects_seen global variable. At the end of its execution, it graphs the objects_seen to show where it saw objects around itself while scanning.

Conclusions

After finishing our project, we are proud that we completed our objective of tracking multiple objects. Near the end of our project, we noticed that there was substantial latency in the motor. We believe that it is likely due to three main reasons. The first being that our code still had the face detector implemented into it so it was wasting processing time on useless tasks, additionally there were also many variables in our code that were left unused wasting more processing time. We also noticed that having two users connected to the pi would contribute to higher latency. Additionally, we would have liked to have implemented steps 6 and 7 but because of time constraints we had to settle for completing steps 1-5.

```python

"""goals6faces.py

Run the face (and eye) detectors and show the results.
"""

# ------------------ SETUP THE KEYBOARD INTERFACE ------------------
# Import the necessary packages
import atexit, select, sys, termios

# Import OpenCV
import cv2
from math import pi
import numpy as np

def detector(shared):

    # Set up a handler, so the terminal returns to normal on exit.
    stdattr = termios.tcgetattr(sys.stdin.fileno())
    def reset_attr():
        termios.tcsetattr(sys.stdin.fileno(), termios.TCSAFLUSH, stdattr)
    atexit.register(reset_attr)

    # Switch terminal to canonical mode: do not wait for <return> press.
    newattr    = termios.tcgetattr(sys.stdin.fileno())
    newattr[3] = newattr[3] & ~termios.ICANON
    termios.tcsetattr(sys.stdin.fileno(), termios.TCSAFLUSH, newattr)

    # Define the kbhit() and getch() functions.
    def kbhit():
        return sys.stdin in select.select([sys.stdin], [], [], 0)[0]
    def getch():
        return sys.stdin.read(1)
    #-----------------------------------------------------------------------


    # Set up video capture device (camera).  Note 0 is the camera number.
    # If things don't work, you may need to use 1 or 2?
    camera = cv2.VideoCapture(0, cv2.CAP_V4L2)
    if not camera.isOpened():
        raise Exception("Could not open video device: Maybe change the cam number?")

    # Change the frame size and rate.  Note only combinations of
    # widthxheight and rate are allowed.  In particular, 1920x1080 only
    # reads at 5 FPS.  To get 30FPS we downsize to 640x480.
    camera.set(cv2.CAP_PROP_FRAME_WIDTH,  640)
    camera.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
    camera.set(cv2.CAP_PROP_FPS,           30)

    # Get the face/eye detector models from XML files.  Instantiate detectors.
    faceXML = "haarcascade_frontalface_default.xml"
    eyeXML1 = "haarcascade_eye.xml"
    eyeXML2 = "haarcascade_eye_tree_eyeglasses.xml"

    faceDetector = cv2.CascadeClassifier(faceXML)
    eyeDetector  = cv2.CascadeClassifier(eyeXML1)

    # Pick some colors.  Note OpenCV uses BGR color codes by default.
    blue  = (255,  0,  0)
    green = (  0,255,  0)
    red   = (  0,  0,255)
    white = (255,255,255)

    hues = []
    sats = []
    vals = []

    scalepan = -(pi/4)/517
    scaletilt = (pi/8)/260

    work = False

    # Keep scanning, until 'q' hit IN IMAGE WINDOW.
    while True:
        if kbhit():
            ch = getch()
            if (ch == 'a'):
                print("\nStart tracking!")
                work = True

        # Grab an image from the camera.  Often called a frame (part of sequence).
        ret, frame = camera.read()

```

```
        # Convert the image to gray scale.
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # Grab the faces - the cascade detector returns a list faces.
        faces = faceDetector.detectMultiScale(gray,
            scaleFactor = 1.2,
            minNeighbors = 5,
            minSize = (30,30),
            flags = cv2.CASCADE_SCALE_IMAGE)

        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        center = hsv[240, 320, :]
        # print(center)
        if work:
            hues.append(center[0])
            sats.append(center[1])
            vals.append(center[2])

        # Process the faces: Each face is a bounding box of (x,y,w,h)
        # coordinates.  Draw the bounding box ON THE ORIGINAL IMAGE.
        for i in range(len(faces)):
            # Grab the first face.
            face = faces[i]

            # Grab the face coodinates.
            (x, y, w, h) = face

            # Draw the bounding box on the original color frame.
            cv2.rectangle(frame, (x, y), (x+w-1, y+h-1), green, 3)


            # Also look for eyes - only within the region of the face!
            # This similarly a list of eyes relative to this region.
            eyes = eyeDetector.detectMultiScale(gray[y:y+h,x:x+w])

            # Process the eyes: As before, eyes is a list of bounding
            # boxes (x,y,w,h) relative to the processed region.
            for (xe,ye,we,he) in eyes:
                # Can you draw circles around the eyes?  Consider the function:
                radius = 20
                cv2.circle(frame, (x + xe + we//2, y + ye + he//2), radius, blue, 2)

        xA = 320
        yA = [0, 480]
        cv2.line(frame, (xA,yA[0]), (xA,yA[1]), (0,0,255), 1)
        xB = [0, 640]
        yB = 240
        cv2.line(frame, (xB[0],yB), (xB[1],yB), (0,0,255), 1)

        hmin, hmax = 20, 40
        smin, smax = 29, 144
        vmin, vmax = 106, 255

        binary = cv2.inRange(hsv, (hmin, smin, vmin), (hmax, smax, vmax))
        # removing erroneous points
        iter = 1
        cv2.erode(binary, None, iterations=iter)
        cv2.dilate(binary, None, iterations=iter)
        # adding missing points
        iter = 1
        cv2.dilate(binary, None, iterations=iter)
        cv2.erode(binary, None, iterations=iter)

        (contours, hierarchy) = cv2.findContours(binary, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
        contours = sorted(contours, key=cv2.contourArea)

        orange = (31, 95, 255)

        MIN_CUTOFF = 500
        MAX_CUTOFF = 5000

        objects_found = []
        for contour in contours:
            if MIN_CUTOFF < cv2.contourArea(contour):
                # ellipses
                ellipse = cv2.fitEllipse(contour)
                (xe, ye), (w, h), angle = ellipse
                ratio = cv2.contourArea(contour)/(pi * (w/2) * (h/2))
                if 0.7 <= ratio <= 1.08:
                    ellipse = cv2.fitEllipse(contour)
                    (xe, ye), (w, h), angle = ellipse
                    cv2.ellipse(frame, ellipse, orange, 3)
```

```python
                    radius = 4
                    cv2.circle(frame, (int(xe), int(ye)), radius, orange, -1)
                    objects_found.append((xe, ye))

        if shared.lock.acquire():
            if len(objects_found) > 0:
                objects_found = np.array(objects_found)
                objects_found[:, 0] = scalepan * (objects_found[:, 0] - xA)
                objects_found[:, 1] = scaletilt * (objects_found[:, 1] - yB)
                shared.deltas = objects_found
                shared.counter = 0
                shared.newdata = True
                print("At least one object found")
            else:
                print("No object found")
                shared.deltas = objects_found
                shared.counter += 1
            shared.newdata = True
            shared.lock.release()

        # cv2.drawContours(frame, contours_drawn, -1, (255,128,0), 2)

        # Show the image with the given title.  Note this won't actually
        # appear (draw on screen) until the waitKey(1) below.
        cv2.imshow('Binary Image', binary)
        cv2.imshow('Processed Image', frame)

        # Check for a key press IN THE IMAGE WINDOW: waitKey(0) blocks
        # indefinitely, waitkey(1) blocks for at most 1ms.  If 'q' break.
        # This also flushes the windows and causes it to actually appear.
        if (cv2.waitKey(1) & 0xFF) == ord('q'):
            print("q hit")
            break

        # Check if the main thread signals this loop to end.
        if shared.lock.acquire():
            stop = shared.stop
            shared.lock.release()
            if stop:
                print("Stopping in detector")
                break

    # Close everything up.
    camera.release()
    cv2.destroyAllWindows()
```

```python
# Import useful packages
import hebi
import numpy as np              # For future use
import matplotlib.pyplot as plt
import enum

from math import pi, sin, cos, asin, acos, atan2, sqrt
from time import sleep, time

# ------------------- SETUP THE KEYBOARD INTERFACE -------------------
# Import the necessary packages
import atexit, select, sys, termios

transform = np.array([1, -1])
MAX_TILT = pi/2

def controller(shared):
    '''goals3democode.py

  Demo code for Goals 3
  '''

    # Set up a handler, so the terminal returns to normal on exit.
    stdattr = termios.tcgetattr(sys.stdin.fileno())
    def reset_attr():
        termios.tcsetattr(sys.stdin.fileno(), termios.TCSAFLUSH, stdattr)
    atexit.register(reset_attr)

    # Switch terminal to canonical mode: do not wait for <return> press.
    newattr    = termios.tcgetattr(sys.stdin.fileno())
    newattr[3] = newattr[3] & ~termios.ICANON
    termios.tcsetattr(sys.stdin.fileno(), termios.TCSAFLUSH, newattr)

    # Define the kbhit() and getch() functions.
    def kbhit():
        return sys.stdin in select.select([sys.stdin], [], [], 0)[0]
    def getch():
        return sys.stdin.read(1)
    #-------------------------------------------------------------------

    #
    #  HEBI Initialization
    #
    #  Create the motor group, and pre-allocate the command and feedback
    #  data structures.  Remember to set the names list to match your
    #  motor.
    #
    names = ['4.5', '6.2']
    group = hebi.Lookup().get_group_from_names(['robotlab'], names)
    if group is None:
        print("Unable to find both motors " + str(names))
        raise Exception("Unable to connect to motors")

    command  = hebi.GroupCommand(group.size)
    feedback = hebi.GroupFeedback(group.size)

    feedback = group.get_next_feedback(reuse_fbk=feedback)

    class Mode(enum.Enum):
        Hold = 0
        Spline = 1
        Scanning = 2

    #
    #  Pre-allocate the storage.
    #
    T = 30.0                     # 5 seconds
    N = int(100 * T)            # 100 samples/second.
    Time = [0.0] * N
    PAct = np.zeros((2, N))
    PCmd = np.zeros((2, N))
    VAct = np.zeros((2, N))
    VCmd = np.zeros((2, N))
    P_error = np.zeros((2, N))
    V_error = np.zeros((2, N))
    objectpt = np.zeros((2, N))

    object_pan = 0
    object_tilt = 0
    # A_objang = 0.73
    # A_mtrvel = 1.75
    T_latency = 0.2
    objects_seen = []

    def graph_scanned_objects(objects_seen):
        objects = np.resize(objects_seen, (len(objects_seen), 2))
        plt.scatter(objects[:, 0], objects[:, 1], marker='x', color='b')
        plt.title("Objects detected during scan")
        plt.xlabel("Theta_pan")
```

```
        plt.ylabel("Theta_tilt")
        plt.xlim(-1.5, 1.5)
        plt.ylim(-1, 1)
        plt.show()

    def calc_spline(t_move, p0, v0, pf, vf):
        a = p0
        b = v0
        c = 3 * (pf - p0) * (1/(t_move ** 2)) - vf * (1/t_move) - 2 * v0 * (1/t_move)
        d = -2 * (pf - p0) * (1/(t_move ** 3)) + vf * (1/(t_move ** 2)) + v0 * (1/(t_move ** 2))
        return a, b, c, d

    def splinetime(p0, pf, v0, vf):
        v_max = np.array([1.75, 1.75])/4
        a_max = v_max/.25
        return max(np.maximum(1.5*(abs(pf-p0)/v_max + abs(v0)/a_max + abs(vf)/a_max), 0.01))

    #
    #  Execute the movement.
    #
    # Initialize the index and time.
    MIN_MOVEMENT_TIME = 1
    scan_once = False
    index = 0
    t     = 0.0
    t_period = 20
    t_hold = 5
    mode = Mode.Hold
    next_mode = Mode.Hold
    p0 = np.array(feedback.position[:2])
    v0 = np.zeros(2)
    p_hold = p0
    vf = np.array([1/t_period, 1/(4*t_period)])
    pf = np.zeros(2)
    A = np.array([1.0, 0.5])
    t0 = t
    t_move = 3

    # a, b, c, d = calc_spline(t_move, p0, v0, pf, vf)
    while True:
        # Read the actual data. This blocks (internally waits) 10ms for the data.
        feedback = group.get_next_feedback(reuse_fbk=feedback)
        pact = np.array(feedback.position[0:2])
        vact = np.array(feedback.velocity[0:2])

        # Compute the commands for this time step.
        if mode is Mode.Hold:
            pcmd = p_hold
            vcmd = np.zeros(2)
        elif mode is Mode.Spline:
            pcmd = a + b * (t - t0) + c * (t - t0) ** 2 + d * (t - t0) ** 3
            vcmd = b + 2 * c * (t - t0) + 3 * d * (t - t0) ** 2
        elif mode is Mode.Scanning:
            pcmd = A*np.sin(2*pi*(t-t0)/(np.array([1, 2])*t_period))
            vcmd = A*(2*pi/(np.array([1, 2])*t_period))*np.cos(2*pi*(t-t0)/(np.array([1, 2])*t_period))
        else:
            print("bad error happened")
            break


        # Send the commands.  This returns immediately.
        command.position = pcmd
        command.velocity = vcmd
        group.send_command(command)

        if mode is Mode.Spline and t >= t0 + t_move:
            p_hold = pf
            t0 = t
            if next_mode == Mode.Scanning:
                mode = Mode.Scanning
            else:
                mode = Mode.Hold
        elif scan_once and t >= t0 + 2*t_period:
            print("\nScanned once\n")
            scan_once = False
            t0 = t
            pf = np.zeros(2)
            vf = np.zeros(2)
            t_move = splinetime(pcmd, pf, vcmd, vf)
            tf = t0 + t_move
            a, b, c, d = calc_spline(t_move, pcmd, vcmd, pf, vf)
            graph_scanned_objects(objects_seen)
            mode = Mode.Spline
            next_mode = Mode.Hold
        elif kbhit():
            # Take action, based on the character.
            ch = getch()
            if   (ch == 'a'):
                t0 = t
```

```
                pf = np.array([0, -pi/8])*transform
                vf = 0
                t_move = splinetime(pcmd, pf, vcmd, vf)
                tf = t0 + t_move
                a, b, c, d = calc_spline(t_move, pcmd, vcmd, pf, vf)
                mode = Mode.Spline
                next_mode = Mode.Hold
                print(’\nGo to A’)
            elif (ch == ’b’):
                t0 = t
                pf = np.array([pi/4, -pi/8])*transform
                vf = 0
                t_move = splinetime(pcmd, pf, vcmd, vf)
                tf = t0 + t_move
                a, b, c, d = calc_spline(t_move, pcmd, vcmd, pf, vf)
                mode = Mode.Spline
                next_mode = Mode.Hold
                print(’\nGo to B’)
            elif (ch == ’c’):
                t0 = t
                pf = np.array([0, -pi/4])*transform
                vf = np.zeros(2)
                t_move = splinetime(pcmd, pf, vcmd, vf)
                tf = t0 + t_move
                a, b, c, d = calc_spline(t_move, pcmd, vcmd, pf, vf)
                mode = Mode.Spline
                next_mode = Mode.Hold
                print(’\nGo to C’)
            elif (ch == ’d’):
                t0 = t
                pf = np.array([0, -pi/8])*transform
                vf = 0
                t_move = splinetime(pcmd, pf, vcmd, vf)
                tf = t0 + t_move
                a, b, c, d = calc_spline(t_move, pcmd, vcmd, pf, vf)
                mode = Mode.Spline
                next_mode = Mode.Hold
                print(’\nGo to D’)
            elif (ch == ’s’): # scan once
                objects_seen = []
                scan_once = True
                t0 = t
                pf = np.zeros(2)
                vf = np.array([1/t_period, 1/(4*t_period)])
                t_move = splinetime(pcmd, pf, vcmd, vf)
                tf = t0 + t_move
                a, b, c, d = calc_spline(t_move, pcmd, vcmd, pf, vf)
                mode = Mode.Spline
                next_mode = Mode.Scanning
                print(’\nStart single scan’)
            elif (ch == ’q’):
                # If we want to quit, break out of the loop.
                print(’\nQuitting’)
                break
            else:
                # Report the bad press.
                print("’The key ’%c’ doesn’t do anything" % ch)

        # Check whether we have a new object location.
        if shared.lock.acquire(timeout=0.001):
            if shared.newdata:
                #Compute the object angles.
                object_angles = shared.deltas
                for i in range(len(object_angles)):
                    object_angles[i, :] += pact - vact*T_latency
                for point in object_angles:
                    objects_seen.append((point[0], point[1]))
                # if object_tilt < MAX_TILT:
                #     t0 = t
                #     pf = np.array([object_pan, object_tilt])
                #     vf = 0
                #     t_move = max(splinetime(pcmd, pf, vcmd, vf), MIN_MOVEMENT_TIME)
                #     tf = t0 + t_move
                #     a, b, c, d = calc_spline(t_move, pcmd, vcmd, pf, vf)
                #     mode = Mode.Spline
                #     next_mode = Mode.Hold
                    # print(’\nObject found\n’)
                # Clear the data.
                shared.newdata = False
            # elif shared.counter >= 10 and mode != Mode.Scanning and next_mode != Mode.Scanning:
            #     t0 = t
            #     pf = np.zeros(2)
            #     vf = np.array([1/t_period, 1/(4*t_period)])
            #     t_move = splinetime(pcmd, pf, vcmd, vf)
            #     tf = t0 + t_move
            #     a, b, c, d = calc_spline(t_move, pcmd, vcmd, pf, vf)
            #     mode = Mode.Spline
            #     next_mode = Mode.Scanning
            #     print("Start scanning, counter =", shared.counter)
```

```python

            shared.lock.release()

        # Store the data for this time step (at the current index).
        if index < N:
            Time[index] = t
            PAct[:, index] = pact
            PCmd[:, index] = pcmd
            VAct[:, index] = vact
            VCmd[:, index] = vcmd
            P_error[:, index] = pact - pcmd
            V_error[:, index] = vact - vcmd
            objectpt[:, index] = np.array([object_pan, object_tilt])

        # Advance the index/time.
        index += 1
        t     += 0.01

    #
    #  Plot.
    #
    # Create a plot of position and velocity, actual and command!
    fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)

    ax1.plot(Time[0:index], PCmd[0, 0:index], color='red', linestyle='--', label='Pan pcmd')
    ax1.plot(Time[0:index], VCmd[0, 0:index], color='blue', linestyle='-',  label='Pan vcmd')
    ax1.plot(Time[0:index], objectpt[0, 0:index], color='green', linestyle='--', label='Object Pan')

    #ax1.plot(Time[0:index], PAct[0, 0:index], color='blue', linestyle='-',  label='PanAct')
    #ax1.plot(Time[0:index], PCmd[0, 0:index], color='blue', linestyle='--', label='PanCmd')
    #ax1.plot(Time[0:index], objectpt[0, 0:index], color='red', linestyle='--', label='Object Pan')
    #ax1.plot(Time[0:index], P_error[0, 0:index], color='red', linestyle='--', label='PanP error')
    #ax2.plot(Time[0:index], VAct[0, 0:index], color='blue', linestyle='-',  label='PanAct')
    #ax2.plot(Time[0:index], VCmd[0, 0:index], color='blue', linestyle='--', label='PanCmd')
    #ax2.plot(Time[0:index], V_error[0, 0:index], color='red', linestyle='--', label='PanV error')

    ax2.plot(Time[0:index], PCmd[1, 0:index], color='yellow', linestyle='--', label='Tilt pcmd')
    ax2.plot(Time[0:index], VCmd[1, 0:index], color='purple', linestyle='-',  label='Tilt vcmd')
    ax2.plot(Time[0:index], objectpt[1, 0:index], color='orange', linestyle='--', label='Object Tilt')

    #ax1.plot(Time[0:index], PAct[1, 0:index], color='green', linestyle='-',  label='TiltAct')
    #ax1.plot(Time[0:index], PCmd[1, 0:index], color='green', linestyle='--', label='TiltCmd')
    #ax1.plot(Time[0:index], objectpt[1, 0:index], color='red', linestyle='--', label='Object Tilt')
    #ax1.plot(Time[0:index], P_error[1, 0:index], color='yellow', linestyle='--', label='TiltP error')
    #ax2.plot(Time[0:index], VAct[1, 0:index], color='green', linestyle='-',  label='TiltAct')
    #ax2.plot(Time[0:index], VCmd[1, 0:index], color='green', linestyle='--', label='TiltCmd')
    #ax2.plot(Time[0:index], V_error[1, 0:index], color='yellow', linestyle='--', label='TiltV error')


    ax1.set_title('Step 3 data')
    ax1.set_ylabel('Pan Position (rad)')
    ax2.set_ylabel('Tilt Position (rad)')
    ax2.set_xlabel('Time (s)')

    ax1.grid()
    ax2.grid()
    ax1.legend()
    ax2.legend()

    plt.show()
```

```python
"""goals8system.py

  This is the main script that coordinates both the controller and
  the detector.  Feel free to play/edit/...

"""

# Import the system parts
import threading
import traceback

# Import your pieces.  Change the "from" names (being the file names)
# to the file names of your two code parts.  Also, this is a great way
# to quickly switch which detector to run.  E.g. you could import from
# "facedetector" in place or "balldetector".
from goals9controller4   import controller
from goals9balldetector4 import detector
# from goals8facedetector import detector     # Alternate option


#
#   Shared Data
#
class Shared:
    def __init__(self):
        # Thread Lock.  Always acquire() this lock before accessing
        # anything else (either reading or writing) in this object.
        # And don't forget to release() the lock when done!
        self.lock = threading.Lock()

        # Flag - stop the detection.  If this is set to True, the
        # detection should break out of the loop and stop.
        self.stop = False

        self.counter = 0
        self.newdata = False  # Flag to indicate fresh, new data
        self.deltas = [] # Relative pan angle, relative tilt angle for each object detected
        # self.deltapan = 0.0  # Relative pan angle
        # self.deltatilt = 0.0  # Relative tilt angle


#
#   Main Code
#
def main():
    # Prepare the shared data object.
    shared = Shared()

    # Create a second thread.
    thread = threading.Thread(target=detector, args=(shared,))

    # Start the second thread with the detector.
    print("Starting second thread")
    thread.start()      # Equivalent to detector(shared) in new thread

    # Use the primary thread for the controller, handling exceptions
    # to gracefully to shut down.
    try:
        controller(shared)
    except BaseException as ex:
        # Report the exception
        print("Ending due to exception: %s" % repr(ex))
        traceback.print_exc()

    # Stop/rejoin the second thread.
    print("Stopping second thread...")
    if shared.lock.acquire():
        shared.stop = True
        shared.lock.release()
    thread.join()       # Wait for thread to end and re-combine.

if __name__ == "__main__":
    main()
```