INTELLIGENT ROBOTICS, PORTLAND STATE UNIVERSITY

Spring Report

Eyebrow State Machine

Trenton Ruf

June 19, 2023

1. STATE MACHINE

The main task of this term was to link together the attitude control, altitude control, and eyebrow recognition from previous terms to control a simulated airplane. To manage this I created a state machine.

I. Initialization

The eyebrow_state_machine.py waits for the initial orientation and altitude data from ROSflight attitude and Barometer nodes before starting a control loop. The eyebrow states are sent from another python application faceSocket.py to the state machine over Ipv4 sockets. This is necessary because the version of ROS that ROSflight runs on requires python 2.7, but Mediapipe and Tensorflow libraries that are used for eyebrow detection require python 3.0 or above. This was the most elegant way I found to be able to bridge these requirements.

II. States

There are six states to the machine: neutral, left, right, up, down, failed. They control the rotations of the plane with neutral not doing anything. When the machine moves into the “failed” state it disables the attitude controller and enables the altitude controller while making the altitude controller’s set-point the current altitude. When the machine moves from the “failed” state to any other state it disables the altitude controller and enables the attitude controller while making the attitude controller’s set-point the current orientation. The “failed” state occurs when either the socket connection has failed between eyebrow_state_machine.py and faceSocket.py, or when faceSocket.py cannot detect a face.

III. Rotations

Changing the orientation of the aircraft was done by multiplying the attitude set-point quaternion by a rotation quaternion. I created four separate rotation quaternions for up, down, left, and right. Since I don’t yet have an intuitive understanding of quaternions I made a function to convert to them from Euler angles. On startup of the state machine node creates the rotation quats from these Euler angles so the somewhat computationally expensive conversion only happens once. During the control loop the attitude set-point rotations are applied at 2 degrees per decisecond. I found that this rate makes controlling the aircraft fairly smooth. For increased “smoothness” I could decrease the degree value while increasing the frequency, but I did not find it necessary.

2. Finishing Quaternion PID Tunings

Last term I found the PID gains for the pitch control portion of the attitude controller. This was done with a deterministic system that tested multiple different proportional gains until an osculating output was created. The resulting “ultimate” gain was put through the Ziegler Nichols equations to get proportional, integral, and derivative gains with a low steady state error. I applied this same system to the aileron and rudder portion to finish tuning the whole controller. The initial orientation of the airplane was different for each control surface tuning with a 15 degree offset along their respective axis, but the set-point orientation was the same with a quaternion of (1,0,0,0). I found tuning of the system had to be done in a specific order: elevator, ailerons, then rudder. The elevator had to be tuned before the ailerons because otherwise the plane would fall nose down when the airplane was banked. And the ailerons had to be tuned before the rudder or large yaws would induce a rolling moment.

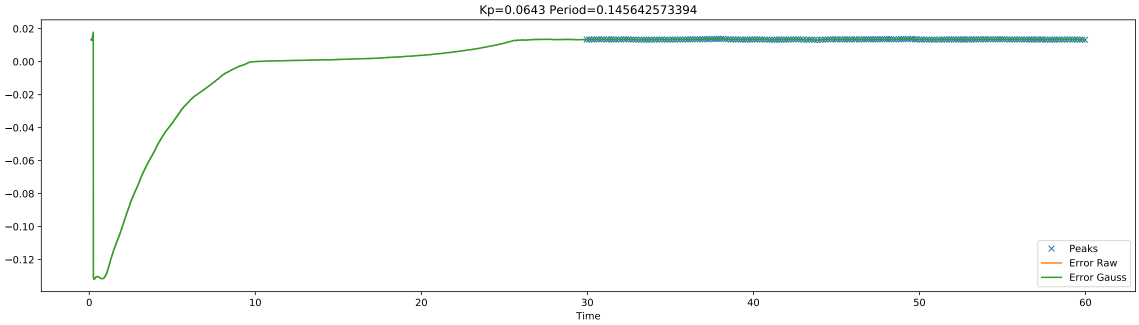

Figure 2.1: Aileron Ultimate Gain

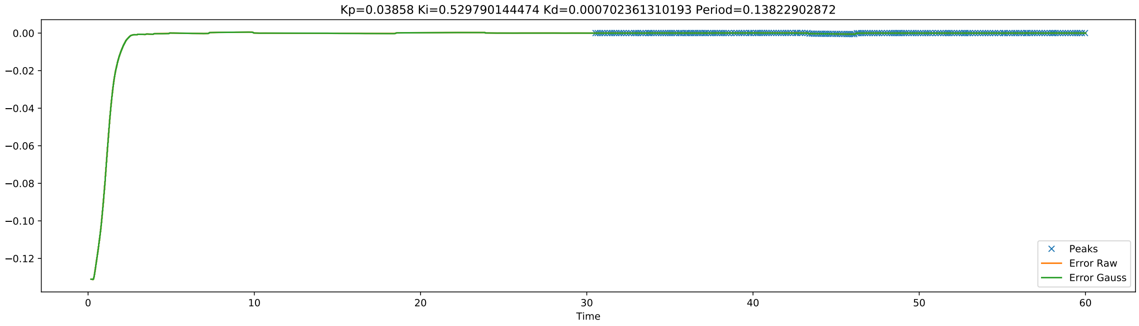

Figure 2.2: Aileron Ziegler Nichols Test

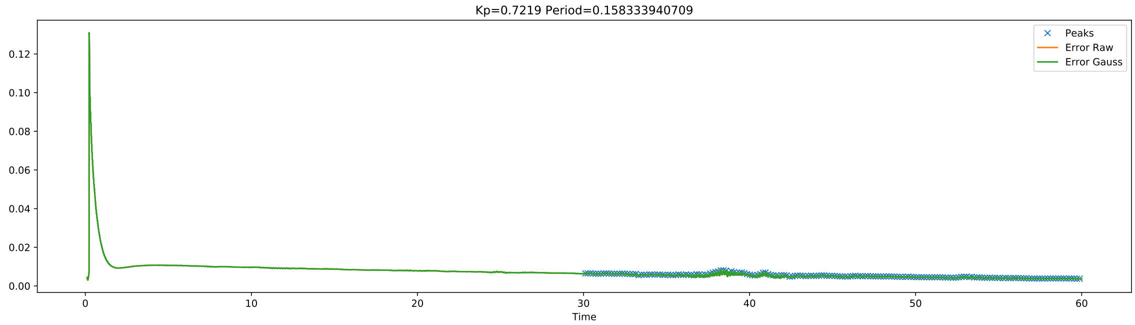

Figure 2.3: Rudder Ultimate Gain

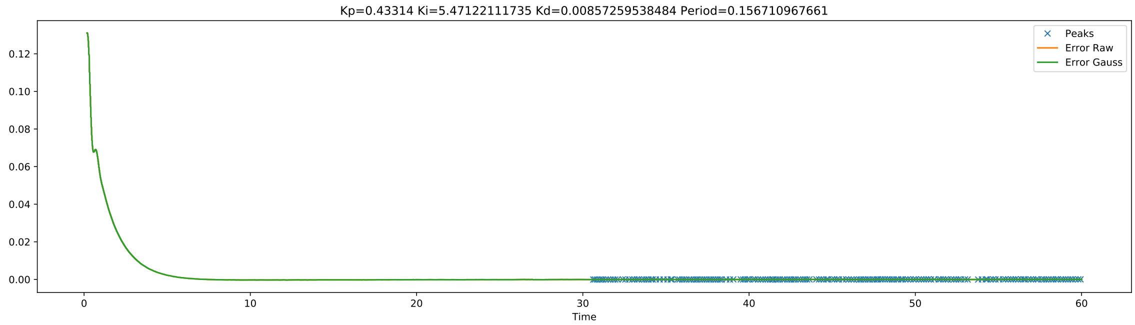

Figure 2.4: Rudder Ziegler Nichols Test

3. Changes to face detection

When looking through the dataset for my eyebrows I found that about half of the labels for the left and right eyebrows were swapped. This ended up being caused by taking eyebrow samples with two different cameras. The camera on my laptop mirrors the image when capturing while the camera on my desktop computer does not. After re-labeling the images correctly and changing the image format from RGB to gray-scale I got much more accurate model than before.

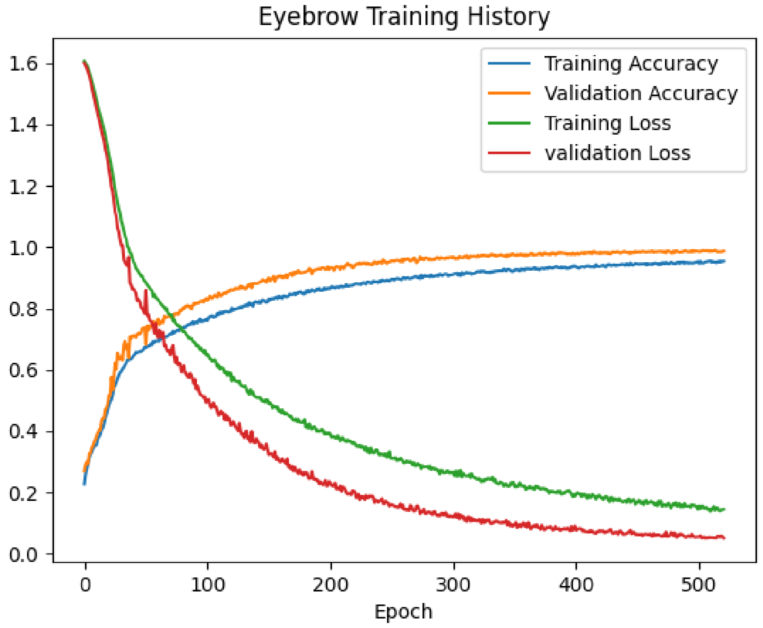

Figure 3.1:

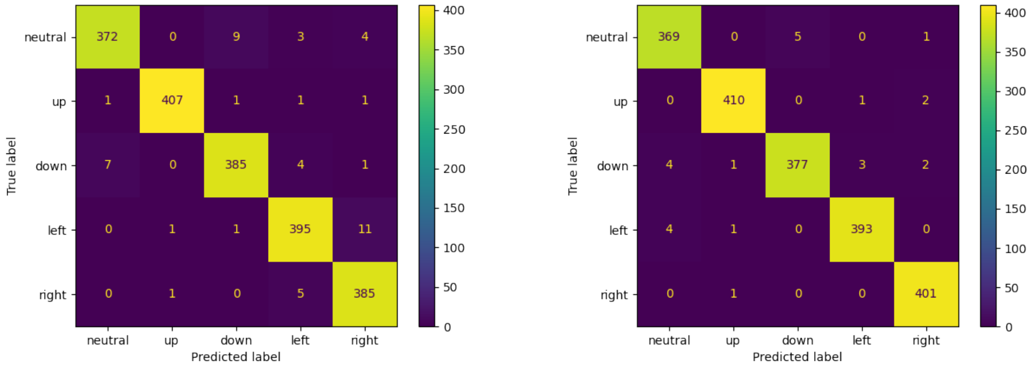

Figure 3.2: RGB images with old dataset on left. Greyscale images with new dataset on right.

As you can see from the Confusion matricies, the new model no longer mistakes right and left commands. This is what initially tipped me off to the problem as it seemed like right and left should have been visually "opposite" and the least likely to be confused.

4. Misc Changes

I made the plane teleport to 10 meters above the ground with a small initial velocity when starting the attitude controller because it helps for testing purposes. The physics simulator takes into account friction and when trying to launch the plane from the ground I would have to max out the plane's thrust to get it airborne and then decrease it dramatically.

I added a runway world file to the gazebo physics simulator startup to have something more interesting to look at rather than an empty world. It also helps when controlling the aircraft because otherwise there is no visual reference for where it is going. This change was not possible until I moved the virtual machine housing the project to a more powerful computer. The processor was not capable of keeping up with the real-time physics simulations on the old machine when also trying to render anything other than an empty world.

The service calls to enable and disable the flight controllers and alter their setpoints were changed to topic publications instead. The rates that the information was being published made more sense for the topic topology.

5. Code

All files related to this project can be found at:
`https://github.com/Trenton-Ruf/Intelligent_Robotics`

Listing 1: attitude_control_gazebo.py

```python
#!/usr/bin/env python

from simple_pid import PID
import math
import time
import numpy as np
import quaternion # using version 2020.11.2.17.0.49
                #numba dependency installed with "sudo apt install python-numba"

import rospy
import rospkg
from rosflight_msgs.msg import Command, Attitude
from nav_msgs.msg import Odometry
from rosflight_control.msg import attitudeSet
from gazebo_msgs.msg import ModelState
from gazebo_msgs.srv import SetModelState
from gazebo_msgs.srv import GetModelState

attitude        = None
aeleronRate     = None
elevatorRate    = None
rudderRate      = None

enable = True;
attitudeSetpoint = np.quaternion(1,0,0,0)

# unrotated unit quaternion
qU = np.quaternion(1,0,0,0)

# Atitude Proportional Controller Gain
Kp = 1

# create PID controllers
elevatorPID = PID(0.03606, 0.33083,0.000983, setpoint=0)
aeleronPID = PID(0.03858, 0.52979, 0.00070, setpoint=0)
rudderPID = PID(0.43314, 5.47122,0.008573, setpoint=0)

elevatorPID.output_limits = (-1,1) # Maximum elevator Deflections
aeleronPID.output_limits = (-1,1) # Maximum aeleron Deflections
rudderPID.output_limits  = (-1,1) # Maximum rudder Deflections

# Create Message Structure
msg = Command()

# msg.ignore = Command.IGNORE_X | Command.IGNORE_Z | Command.IGNORE_F
#msg.ignore = Command.IGNORE_F # Only ignore throttle at first
msg.F = 0.7 # Just for testing
msg.mode = Command.MODE_PASS_THROUGH

# Create publisher
publisher = rospy.Publisher("/fixedwing/command",Command, queue_size=1)

startTime = time.time()

def resetState():
    state_msg = ModelState()
    state_msg.model_name = 'fixedwing'
    state_msg.pose.position.x = 0
    state_msg.pose.position.y = 0
    state_msg.pose.position.z = 20
    state_msg.pose.orientation.x = 0
    state_msg.pose.orientation.y = 0.131
    state_msg.pose.orientation.z = 0
    state_msg.pose.orientation.w = 0.991

    state_msg.twist.linear.x = 8

    rospy.wait_for_service('/gazebo/set_model_state')

    rospy.loginfo("Resetting State")

    try:
        set_state = rospy.ServiceProxy('/gazebo/set_model_state',SetModelState)
        resp = set_state(state_msg)
    except rospy.ServiceExeption, e:
        print("Service call failed: %s" % e)


def attitudeControl(attitudeData):

    global attitudeSetpoint
    global enable

    if not enable:
        return;

    # Get measured attitude as quaternion
    attitudeMeasured = np.quaternion(attitudeData.pose.pose.orientation.w,
                                     attitudeData.pose.pose.orientation.x,
                                     attitudeData.pose.pose.orientation.y,
                                     attitudeData.pose.pose.orientation.z
                                     )

    # rospy.loginfo("attitudeMeasured: " + str(attitudeMeasured))

    # Get the attitude error
    attitudeError =  np.multiply( np.conjugate(attitudeMeasured),  attitudeSetpoint )

    # Since 2 rotations can describe every attitude,
    # find the shorter of both rotations
    if attitudeError.w < 0:
        np.negative(attitudeError)

    # Assume derivative of attitude setpoint is proportional to the attitude error
    attitudeSetpointDerivative = Kp * attitudeError

    # Get angular rate setpoints
    rateSetpoints = np.multiply( (2 * qU) , attitudeSetpointDerivative)

    # Give the PID controllers the new setpoints
    aeleronPID.setpoint  = rateSetpoints.x
    elevatorPID.setpoint = rateSetpoints.y
    rudderPID.setpoint   = rateSetpoints.z

    # Get the Control Surface Deflections from the PID output
    msg.header.stamp = rospy.Time.now()
    msg.x = aeleronPID(attitudeData.twist.twist.angular.x)
    msg.y = elevatorPID(attitudeData.twist.twist.angular.y)
    msg.z = rudderPID(attitudeData.twist.twist.angular.z)
    publisher.publish(msg)

    # Send info to the console for debugging
    """
    rospy.loginfo(
                    "Aeleron setpoint:"+str(round(aeleronPID.setpoint, 4)) +
                    " Elevator setpoint:"+str(round(elevatorPID.setpoint, 4)) +
                    " Rudder setpoint:"+str(round(rudderPID.setpoint, 4))
                    )

    rospy.loginfo(
                    "Aeleron:"+str(round(msg.x, 4)) +
                    " Elevator:"+str(round(msg.y, 4)) +
                    " Rudder:"+str(round(msg.z, 4))
                    )
    """


def attitudeSet_listener(attitudeSet_data):
    global enable
    # Enable always true for debugging
    #enable = attitudeSet_data.enable
    global attitudeSetpoint
    attitudeSetpoint = quaternion.as_quat_array(attitudeSet_data.quaternion)
    rospy.loginfo("Enable: "+ str(enable) + "\nattitudeSetpoint: " + str(attitudeSetpoint))

if __name__ == '__main__':
    try:

        # Init Node
        rospy.init_node('attitude_control')

        # Create attitude listener
        #rospy.Subscriber("/fixedwing/attitude", Attitude, attitudeControl)
        rospy.Subscriber("/fixedwing/truth/NED", Odometry, attitudeControl)

        # Create attitudeSet listener
        rospy.Subscriber("/attitudeSet", attitudeSet, attitudeSet_listener)

        resetState()
        rospy.spin()

    except rospy.ROSInterruptException:
        pass
```

Listing 2: altitude_control_gazebo.py

```
#!/usr/bin/env python

import rospy
from rosflight_msgs.msg import Command, Barometer
from simple_pid import PID
import numpy as np
import skfuzzy.control as ctrl
import time
from rosflight_control.msg import altitudeSet

altitude = None
altitudeSetpoint = 5.
enable = True

# create PID controller
pid = PID(0.0015,0.0004,0.003, setpoint=altitudeSetpoint) # PID tunings will be overwritten with fuzzy logic
pid.output_limits =(-1,1) # Maximum Elevator Deflections

# Time variables for calculating Error Delta
startTime=time.time()
endTime=0
lastPidError = 0

# Create Message Structure
msg = Command()
# need to ignore Aeileron, Rudder
msg.ignore = Command.IGNORE_X | Command.IGNORE_Z
msg.F = 0.7
msg.mode = Command.MODE_PASS_THROUGH

# Create publisher
publisher = rospy.Publisher("/command",Command,queue_size=1)

##########################
# Fuzzy Setup
##########################

# Create five fuzzy variables - two inputs, three outputs
error = ctrl.Antecedent(np.linspace(-5,5,7), 'error')
delta = ctrl.Antecedent(np.linspace(-40,40,7), 'delta')

"""
# Funtional
kp = ctrl.Consequent(np.linspace(0 ,0.00075,7), 'kp')
kd = ctrl.Consequent(np.linspace(0 ,0.0055,7), 'kd')
ki = ctrl.Consequent(np.linspace(0 ,0.00015,7), 'ki')
"""

# Experimental
kp = ctrl.Consequent(np.linspace(-0.000000,0.01,7), 'kp')
kd = ctrl.Consequent(np.linspace(-0.000000,0.02,7), 'kd')
ki = ctrl.Consequent(np.linspace(-0.000000,0.01,7), 'ki')

# Fuzzy Terms
names = ['nb', 'nm', 'ns', 'zo', 'ps', 'pm', 'pb']
error.automf(names=names)
delta.automf(names=names)
kp.automf(names=names)
ki.automf(names=names)
kd.automf(names=names)
```

```python

# So many rules... here we go

# kp rules ################################################
rule0 = ctrl.Rule(antecedent=((error['nb'] & delta['nb']) |
                              (error['nm'] & delta['nb']) |
                              (error['nb'] & delta['nm']) |
                              (error['nm'] & delta['nm'])),
                  consequent=kp['pb'], label='rule kp pb')

rule1 = ctrl.Rule(antecedent=((error['ns'] & delta['nb']) |
                              (error['zo'] & delta['nb']) |
                              (error['ns'] & delta['nm']) |
                              (error['zo'] & delta['nm']) |
                              (error['nb'] & delta['ns']) |
                              (error['nm'] & delta['ns']) |
                              (error['ns'] & delta['ns']) |
                              (error['nb'] & delta['zo'])),
                  consequent=kp['pm'], label='rule kp pm')

rule2 = ctrl.Rule(antecedent=((error['ps'] & delta['nb']) |
                              (error['ps'] & delta['nm']) |
                              (error['zo'] & delta['ns']) |
                              (error['nm'] & delta['zo']) |
                              (error['ns'] & delta['zo']) |
                              (error['nb'] & delta['ps']) |
                              (error['nm'] & delta['ps']) |
                              (error['nb'] & delta['pm'])),
                  consequent=kp['ps'], label='rule kp ps')

rule3 = ctrl.Rule(antecedent=((error['pm'] & delta['nb']) |
                              (error['pb'] & delta['nb']) |
                              (error['pm'] & delta['nm']) |
                              (error['ps'] & delta['ns']) |
                              (error['zo'] & delta['zo']) |
                              (error['ns'] & delta['ps']) |
                              (error['nm'] & delta['pm']) |
                              (error['nb'] & delta['pb'])),
                  consequent=kp['zo'], label='rule kp zo')

rule4 = ctrl.Rule(antecedent=((error['pb'] & delta['nm']) |
                              (error['pm'] & delta['ns']) |
                              (error['ps'] & delta['zo']) |
                              (error['zo'] & delta['ps']) |
                              (error['ps'] & delta['ps']) |
                              (error['ns'] & delta['pm']) |
                              (error['nm'] & delta['pb'])),
                  consequent=kp['ns'], label='rule kp ns')

rule5 = ctrl.Rule(antecedent=((error['pb'] & delta['ns']) |
                              (error['pm'] & delta['zo']) |
                              (error['pb'] & delta['zo']) |
                              (error['pm'] & delta['ps']) |
                              (error['pb'] & delta['ps']) |
                              (error['pm'] & delta['pm']) |
                              (error['ps'] & delta['pm']) |
                              (error['zo'] & delta['pm']) |
                              (error['ns'] & delta['pb']) |
                              (error['zo'] & delta['pb']) |
                              (error['ps'] & delta['pb'])),
                  consequent=kp['nm'], label='rule kp nm')

```

```
rule6 = ctrl.Rule(antecedent=((error['pb'] & delta['pm']) |
                              (error['pm'] & delta['pb']) |
                              (error['pb'] & delta['pb'])),
                  consequent=kp['nb'], label='rule kp nb')

# ki rules ################################################
rule7 = ctrl.Rule(antecedent=((error['nb'] & delta['nb']) |
                              (error['nm'] & delta['nb']) |
                              (error['ns'] & delta['nb']) |
                              (error['nb'] & delta['nm']) |
                              (error['nm'] & delta['nm']) |
                              (error['nb'] & delta['ns'])),
                  consequent=ki['nb'], label='rule ki nb')

rule8 = ctrl.Rule(antecedent=((error['zo'] & delta['nb']) |
                              (error['ns'] & delta['nm']) |
                              (error['nm'] & delta['ns']) |
                              (error['nb'] & delta['zo'])),
                  consequent=ki['nm'], label='rule ki nm')

rule9 = ctrl.Rule(antecedent=((error['ps'] & delta['nb']) |
                              (error['zo'] & delta['nm']) |
                              (error['ps'] & delta['nm']) |
                              (error['ns'] & delta['ns']) |
                              (error['zo'] & delta['ns']) |
                              (error['nm'] & delta['zo']) |
                              (error['ns'] & delta['zo']) |
                              (error['nm'] & delta['ps']) |
                              (error['nb'] & delta['ps'])),
                  consequent=ki['ns'], label='rule ki ns')

rule10 = ctrl.Rule(antecedent=((error['pm'] & delta['nb']) |
                              (error['pb'] & delta['nb']) |
                              (error['pm'] & delta['nm']) |
                              (error['pb'] & delta['nm']) |
                              (error['ps'] & delta['ns']) |
                              (error['zo'] & delta['zo']) |
                              (error['ns'] & delta['ps']) |
                              (error['nm'] & delta['pm']) |
                              (error['nb'] & delta['pm']) |
                              (error['nm'] & delta['pb']) |
                              (error['nb'] & delta['pb'])),
                  consequent=ki['zo'], label='rule ki zo')

rule11 = ctrl.Rule(antecedent=((error['pm'] & delta['ns']) |
                              (error['pb'] & delta['ns']) |
                              (error['ps'] & delta['zo']) |
                              (error['pm'] & delta['zo']) |
                              (error['zo'] & delta['ps']) |
                              (error['ps'] & delta['ps']) |
                              (error['ns'] & delta['pm']) |
                              (error['ns'] & delta['pb'])),
                  consequent=ki['ps'], label='rule ki ps')

rule12 = ctrl.Rule(antecedent=((error['pb'] & delta['zo']) |
                              (error['pm'] & delta['ps']) |
                              (error['ps'] & delta['pm']) |
                              (error['zo'] & delta['pm']) |
                              (error['zo'] & delta['pb'])),
                  consequent=ki['pm'], label='rule ki pm')

rule13 = ctrl.Rule(antecedent=((error['pb'] & delta['ps']) |
                              (error['pm'] & delta['pm']) |
                              (error['pb'] & delta['pm']) |
                              (error['pb'] & delta['pb']) |
                              (error['pm'] & delta['pb']) |
                              (error['ps'] & delta['pb'])),
                  consequent=ki['pb'], label='rule ki pb')

# kd rules ################################################
rule14 = ctrl.Rule(antecedent=((error['nb'] & delta['ns']) |
                              (error['nm'] & delta['ns']) |
                              (error['nb'] & delta['zo']) |
                              (error['nb'] & delta['ps'])),
                  consequent=kd['nb'], label='rule kd nb')

rule15 = ctrl.Rule(antecedent=((error['nb'] & delta['nm']) |
                              (error['ns'] & delta['ns']) |
                              (error['nm'] & delta['zo']) |
                              (error['ns'] & delta['zo']) |
                              (error['nm'] & delta['ps']) |
                              (error['nb'] & delta['pm'])),
                  consequent=kd['nm'], label='rule kd nm')

rule16 = ctrl.Rule(antecedent=((error['nm'] & delta['nm']) |
                              (error['ns'] & delta['nm']) |
                              (error['zo'] & delta['nm']) |
                              (error['ps'] & delta['nm']) |
                              (error['pm'] & delta['nm']) |
                              (error['zo'] & delta['ns']) |
                              (error['zo'] & delta['zo']) |
                              (error['zo'] & delta['ps']) |
                              (error['ns'] & delta['ps']) |
                              (error['zo'] & delta['pm']) |
                              (error['ns'] & delta['pm']) |
                              (error['nm'] & delta['pm'])),
                              #(error['nb'] & delta['pb']) | Test this change , and remove from 'rule kd ps'
                              #(error['nm'] & delta['pb']) |
                  consequent=kd['ns'], label='rule kd ns')

rule17 = ctrl.Rule(antecedent=((error['ns'] & delta['nb']) |
                              (error['zo'] & delta['nb']) |
                              (error['ps'] & delta['nb']) |
                              (error['ps'] & delta['ns']) |
                              (error['ps'] & delta['zo']) |
                              (error['ps'] & delta['ps']) |
                              (error['ps'] & delta['pm']) |
                              (error['zo'] & delta['pb']) |
                              (error['ns'] & delta['pb'])),
                  consequent=kd['zo'], label='rule kd zo')

rule18 = ctrl.Rule(antecedent=((error['nb'] & delta['nb']) |
                              (error['nm'] & delta['nb']) |
                              (error['pm'] & delta['ns']) |
                              (error['pm'] & delta['zo']) |
                              (error['pm'] & delta['ps']) |
                              (error['pb'] & delta['ps']) |
                              (error['pm'] & delta['pm']) |
                              (error['pb'] & delta['pm']) |
                              (error['nb'] & delta['pb']) | # Really? I'm thinkin delta is 'ns' instead
                              (error['nm'] & delta['pb']) | # Really? I'm thinkin delta is 'ns' instead
```

```python
                               (error['ps'] & delta['pb'])),
                  consequent=kd['ps'], label='rule kd ps')

rule19 = ctrl.Rule(antecedent=((error['pb'] & delta['nm']) |
                               (error['pb'] & delta['ns']) |
                               (error['pb'] & delta['zo'])),
                  consequent=kd['pm'], label='rule kd pm')

rule20 = ctrl.Rule(antecedent=((error['pm'] & delta['nb']) |
                               (error['pb'] & delta['nb']) |
                               (error['pm'] & delta['pb']) |
                               (error['pb'] & delta['pb'])),
                  consequent=kd['pb'], label='rule kd pb')

system = ctrl.ControlSystem(rules=[
                                    rule0, rule1, rule2, rule3, rule4, rule5, rule6,
                                    rule7, rule8, rule9, rule10, rule11, rule12, rule13,
                                    rule14, rule15, rule16, rule17, rule18, rule19, rule20
                                    ])
sim = ctrl.ControlSystemSimulation(system, flush_after_run=1000) # lower flush if memory is scarce

altList=[]
# Averages the last three altimeter readings before sending the reading to altitudePID
# Currently not used!
def altimeterFilter(baro):
    altList.insert(0,baro.altitude)
    if len(altList) > 3:
        altList.pop()
        avgAltitude = sum(altList) / float(len(altList))
        altitudePID(avgAltitude)

# Recieves altitude readings and outputs Elevator deflection commands
def altitudePID(baro):

    rospy.loginfo("enable:"+ str(enable) + " altitudeSetpoint: "+ str(altitudeSetpoint))

    # Do nothing if disabled
    global enable
    if not enable:
        return

    altitude = baro.altitude

    global startTime
    global endTime
    global lastPidError
    pid.setpoint = altitudeSetpoint

    # Get error and delta
    pidError= altitudeSetpoint - altitude
    pidDelta = float(pidError - lastPidError)/float(endTime - startTime)
    lastPidError = pidError

    # Reset timer for calculating error delta
    endTime = startTime
    startTime = time.time()

    # Compute Fuzzy Inference
    sim.input['error']= pidError
    sim.input['delta']= pidDelta
    sim.compute()

    #Set pid tunings from fuzzy logic
    pid.tunings = (sim.output['kp'],
                   sim.output['ki'],
                   sim.output['kd']
                   )

    msg.header.stamp = rospy.Time.now()
    msg.y = pid(altitude)
    publisher.publish(msg)
    # Send info to the console for debugging
    rospy.loginfo("Altitude:"+str(round(altitude, 4)) +
                  " Elevator:"+str(round(msg.y, 4)) +
                  " Kp:"+str(round(sim.output['kp'], 4)) +
                  " Ki:"+str(round(sim.output['ki'], 4)) +
                  " Kd:"+str(round(sim.output['kd'], 4)) +
                  " Error:"+str(round(pidError)) +
                  " Delta:"+str(round(pidDelta))
                  )

def altitudeSet_listener(altitudeSet):
    global enable
    global altitudeSetpoint
    enable = altitudeSet.enable
    rospy.loginfo("Raw Enable: "+ str(altitudeSet.enable) )
    altitudeSetpoint = altitudeSet.setPoint

if __name__ == '__main__':
    try:
        # Init Node
        rospy.init_node('altitude_control')

        # Create altitudeSet listener
        rospy.Subscriber("/altitudeSet", altitudeSet, altitudeSet_listener)

        # Create barometer listener
        #rospy.Subscriber("/fixedwing/baro",Barometer, altimeterFilter) # with filter
        rospy.Subscriber("/fixedwing/baro", Barometer, altitudePID ) # bypass filter


        rospy.spin()

    except rospy.ROSInterruptException:
        pass

#msg.x = 0.0 #Aeileron deflection (-1,1)
#msg.z = 0.0 #Rudder deflection (-1,1)
```

Listing 3: eyebrow_state_machine.py

```python
#!/usr/bin/env python

import socket
import math
import numpy as np
import quaternion
import time

# Ros Includes
import rospy
from rosflight_msgs.msg import Command, Attitude, Barometer
# TODO replace Odometry with Attitude after supplimenting Kalman filter with Magnotometer data
from nav_msgs.msg import Odometry
from rosflight_control.msg import altitudeSet, attitudeSet

# Convert euler angles for roll, pitch, and yaw into a quaternion.
def eulerToQuat(roll, pitch, yaw):
    w = math.cos(roll/ 2) * math.cos(pitch/ 2) * math.cos(yaw/ 2) \
        + math.sin(roll/ 2) * math.sin(pitch/ 2) * math.sin(yaw/ 2)
    x = math.sin(roll/ 2) * math.cos(pitch/ 2) * math.cos(yaw/ 2) \
        - math.cos(roll/ 2) * math.sin(pitch/ 2) * math.sin(yaw/ 2)
    y = math.cos(roll/ 2) * math.sin(pitch/ 2) * math.cos(yaw/ 2) \
        + math.sin(roll/ 2) * math.cos(pitch/ 2) * math.sin(yaw/ 2)
    z = math.cos(roll/ 2) * math.cos(pitch/ 2) * math.sin(yaw/ 2) \
        - math.sin(roll/ 2) * math.sin(pitch/ 2) * math.cos(yaw/ 2)
    rospy.loginfo("Quat " + str(w) + " " + str(x) + " " + str(y) + " " + str(z))
    return np.quaternion(w,x,y,z)


class stateMachine():
    # Create States
    neutralState    = 0
    upState         = 1
    downState       = 2
    leftState       = 3
    rightState      = 4
    failedState       = 5 # Eyebrows not detected

    stateSocketDict = {"neutral":0,
                       "up":1,
                       "down":2,
                       "left":3,
                       "right":4,
                       "failed":5}


    def __init__(self):
        # Set inital State
        self.initialState = self.neutralState
        self.currentState = self.initialState
        self.previousState = None

        self.orientation = None
        self.setpoint = None
        self.altitude = None


        # Initialize rotation quaternions
        degreesRot = 2
        thetaRot = degreesRot * math.pi / 180
        self.rotateUp = eulerToQuat(0, thetaRot, 0)
        self.rotateDown = eulerToQuat(0, -thetaRot, 0)
        self.rotateLeft = eulerToQuat(-thetaRot, 0, 0)
        self.rotateRight = eulerToQuat(thetaRot, 0, 0)

        # Loop frequency
        self.hz = 10
        self.period = 1 / self.hz
        self.startTime = time.time()

        # Initialize socket listener
        # Used for recieving eyebrow states from a client
        self.s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        #self.s.bind(("0.0.0.0", 8745))
        #self.s.bind(("socket.gethostname()", 8745))
        self.s.bind(("192.168.106.114", 8745))
        self.s.listen(1)
        self.socketConnected = False
        self.s.settimeout( 2 * self.period )

        # attitudSet Publisher
        self.attitudeSetPub = rospy.Publisher('attitudeSet', attitudeSet, queue_size=1)
        self.attitudeSetMsg = attitudeSet()
        self.attitudeSetMsg.enable = False

        # altitudSet Publisher
        self.altitudeSetPub = rospy.Publisher('altitudeSet', altitudeSet, queue_size=1)
        self.altitudeSetMsg = altitudeSet()
        self.altitudeSetMsg.enable = True


    def connectSocket(self):
        try:
            self.clientsocket, self.address = self.s.accept()
            rospy.loginfo("Connection from " + str(self.address) +" established")
            msg ="Connected to eyebrow_control server"
            self.clientsocket.send(msg.encode("utf-8"))
            self.socketConnected = True
            return True
        except socket.error:
            self.socketConnected = False
            self.currentState = self.failedState
            rospy.loginfo("connectSocket failed")
            return False


    def recieveSocket(self):
        try:
            msg = self.clientsocket.recv(16)
            self.currentState = self.stateSocketDict[ msg.decode("utf-8") ]
            rospy.loginfo("Socket mesg: " + msg)
        except:
            #self.socketConnected = False
            self.currentState = self.failedState
            rospy.loginfo("recieveSocket failed")


    def setAltitude(self, altitudeData):
        self.altitude = altitudeData.altitude


    def setOrientation(self, attitudeData):
        self.orientation = np.quaternion(attitudeData.pose.pose.orientation.w,
                                         attitudeData.pose.pose.orientation.x,
                                         attitudeData.pose.pose.orientation.y,
                                         attitudeData.pose.pose.orientation.z)


    def rotate(self, rotation):
        self.setpoint = rotation * self.setpoint
        rospy.loginfo("rotated quat:" + str(self.setpoint))


    def loop(self):

        # Initialize messages
        self.attitudeSetMsg.quaternion = quaternion.as_float_array(self.setpoint)
        self.altitudeSetMsg.setPoint = float(self.altitude)

        while not rospy.is_shutdown():

            if not self.socketConnected:
                if self.connectSocket():
                    continue
            else:
                self.recieveSocket()

            elapsedTime = time.time() - self.startTime
            if elapsedTime >= self.period:
                # Reset Timer
                self.startTime = time.time()

                if self.currentState == self.failedState:
                    # Start Altitude Hold
                    self.attitudeSetMsg.enable = False
                    self.altitudeSetMsg.enable = True
                    if self.previousState != self.failedState:
                        # Set the altitude hold to the current altitude
                        self.altitudeSetMsg.setPoint = float(self.altitude)

                else:
                    # Stop Altitude Hold
                    self.attitudeSetMsg.enable = True
                    self.altitudeSetMsg.enable = False
                    if self.previousState == self.failedState:
                        # Set the attitude setpoint to the current orientation
                        self.setpoint = self.orientation

                    if self.currentState == self.neutralState:
                        pass # Do nothing
                    elif self.currentState == self.upState:
                        self.rotate(self.rotateUp)
                    elif self.currentState == self.downState:
                        self.rotate(self.rotateDown)
                    elif self.currentState == self.leftState:
                        self.rotate(self.rotateLeft)
                    elif self.currentState == self.rightState:
                        self.rotate(self.rotateRight)

                    self.attitudeSetMsg.quaternion = quaternion.as_float_array(self.
                        ↪ setpoint)

                rospy.loginfo("Current State: " + str(self.currentState) )
                # Publish Topics
                self.attitudeSetPub.publish(self.attitudeSetMsg)
                self.altitudeSetPub.publish(self.altitudeSetMsg)

                # Set previous State
                self.previousState = self.currentState

def main():
    try:
        # Init Node
        rospy.init_node('eyebrow_control')

        # Create state machine class
        eyebrowMachine = stateMachine()

        # Create barometer listener
        rospy.Subscriber("/fixedwing/baro", Barometer, eyebrowMachine.setAltitude)

        # Create attitude listener
        rospy.Subscriber("/fixedwing/truth/NED", Odometry, eyebrowMachine.setOrientation)

        rospy.loginfo("Waiting for initial orientation and altitude")
        while(eyebrowMachine.orientation is None or eyebrowMachine.altitude is None):
            pass

        # Set the initial setpoint to be the same as the first orientation
        eyebrowMachine.setpoint = eyebrowMachine.orientation

        eyebrowMachine.loop()

    except rospy.ROSInterruptException:
        # Print Exception?
        pass

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

Listing 4: faceSocket.py

```python
#!/usr/bin/env python
import cv2
import mediapipe as mp
from mediapipe.python.solutions.drawing_utils import _normalized_to_pixel_coordinates
mp_face_detection = mp.solutions.face_detection
mp_drawing = mp.solutions.drawing_utils

import os
# I don't have an NVidia GPU :(
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2' # disable annoying Tensorflow warnings
import keras
#from keras.models import Sequential
from keras.models import load_model
import numpy as np
import socket

# Overlay text onto user interface
# Input original image, text color, and text contents
# Returns new image with overlayed text
def screenText(img,color,text):
    if color.lower() == "green":
        font_color = (0,255,0)
    elif color.lower() == "black":
        font_color = (0,0,0)
    font = cv2.FONT_HERSHEY_SIMPLEX
    font_size = 0.8
    font_thickness = 2
    x,y = 0,100
    img_text = cv2.putText(img, text, (x,y), font, font_size, font_color, font_thickness, cv2.LINE_AA)
    return img_text

# Crops eyebrows from image
# Input original image and mediapipe face keypoint coordinates
# Returns 50x100 px image containing only eyebrows
def cropDetection(image_input,detection):
    # Example from from https://stackoverflow.com/questions/71094744/how-to-crop-face-detected-via-mediapipe-in-python
    image_rows, image_cols, _ = image_input.shape
    location = detection.location_data
    # Keypoint in order (right eye, left eye, nose tip, mouth center, right ear tragion, and left ear tragion)

    # Get bounding box coordinates
    # Not used since transitioning to eyebrows only instead of full face
    """
    relative_bounding_box = location.relative_bounding_box
    rect_start_point = _normalized_to_pixel_coordinates(
        relative_bounding_box.xmin, relative_bounding_box.ymin, image_cols,
        image_rows)
    rect_end_point = _normalized_to_pixel_coordinates(
        relative_bounding_box.xmin + relative_bounding_box.width,
        relative_bounding_box.ymin + relative_bounding_box.height, image_cols,
        image_rows)
    """

    leftEar = location.relative_keypoints[5]
    leftEarPoint = _normalized_to_pixel_coordinates(
        leftEar.x, leftEar.y, image_cols,
        image_rows)

    rightEar = location.relative_keypoints[4]
    rightEarPoint = _normalized_to_pixel_coordinates(
        rightEar.x, rightEar.y, image_cols,
        image_rows)

    leftEye = location.relative_keypoints[1]
    leftEyePoint = _normalized_to_pixel_coordinates(
        leftEye.x, leftEye.y, image_cols,
        image_rows)

    rightEye = location.relative_keypoints[0]
    rightEyePoint = _normalized_to_pixel_coordinates(
        rightEye.x, rightEye.y, image_cols,
        image_rows)

    # crop image depending on distance between left and right eye
    try:

        xrightEye_relative,yrightEye_relative = rightEyePoint
        xleftEye_relative,yleftEye_relative = leftEyePoint

        xrightEar_relative,yrightEar_relative = rightEarPoint
        xleftEar_relative,yleftEar_relative = leftEarPoint

        yEyeDiff = yrightEye_relative - yleftEye_relative
        xEyeDiff = xrightEye_relative - xleftEye_relative

        xleft = xrightEye_relative + xEyeDiff/2
        xright = xleftEye_relative - xEyeDiff/2

        if yEyeDiff < 0:
            ytop = yrightEye_relative + xEyeDiff/1.5
            ybot = yleftEye_relative + xEyeDiff/8

        else:
            ytop = yleftEye_relative + xEyeDiff /1.5
            ybot = yrightEye_relative + xEyeDiff/8

        crop_img = image_input[int(ytop): int(ybot), int(xleft): int(xright)]
        #cv2.imshow('cropped',crop_img)
        #return crop_img

        resized_crop = cv2.resize(crop_img,(100,50))
        #cv2.imshow('resized_cropped',resized_crop)
        return resized_crop

    except:
        return -1

# predict eyebrow expression
# Input cropped image and Trained model
# Return predicted expression
def checkExpression(img,model):
    #norm = cv2.normalize(img, 0, 1, cv2.NORM_MINMAX)
    #norm = cv2.normalize(img, None, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)
    # convert to greyscale
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    prediction = model.predict(np.expand_dims(gray,axis=0))
    expressions=['neutral','up','down','left','right']
    expression = expressions[np.argmax(prediction)]
    print(expression)
    return expression

model = load_model("./faceModel")

# Socket connection initialization
print("\nStarting Socket Connection")
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("192.168.106.114", 8745))
msg = s.recv(16)
print(msg.decode("utf-8"))

# For webcam input:
cap = cv2.VideoCapture(0)
with mp_face_detection.FaceDetection(
    model_selection=0, min_detection_confidence=0.5) as face_detection:
    while cap.isOpened():
        success, image = cap.read()
        if not success:
            print("Ignoring empty camera frame.")
            # If loading a video, use 'break' instead of 'continue'.
            continue

        # To improve performance, optionally mark the image as not writeable to
        # pass by reference.
        image.flags.writeable = False
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        results = face_detection.process(image)

        # Draw the face detection annotations on the image.
        image.flags.writeable = True
        image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
        if results.detections:
            detection = results.detections[0] # Grab only the closest face
            cropped_img = cropDetection(image,detection)
            if isinstance(cropped_img, int):
                text = "Expression: failed"
                image = screenText(cv2.flip(image,1),"black",text)
                msg = "failed"
                s.send(msg.encode("utf-8"))
            else:
                expression = checkExpression(cropped_img, model)
                mp_drawing.draw_detection(image, detection)
                text = "Expression: " + expression
                image = screenText(cv2.flip(image,1),"green",text)
                msg = expression
                s.send(msg.encode("utf-8"))
        else:
            text = "Expression: failed"
            image = screenText(cv2.flip(image,1),"black",text)
            msg = "failed"
            s.send(msg.encode("utf-8"))


        cv2.imshow('MediaPipe',image)

        keyPress = cv2.waitKey(5) & 0xFF
        if keyPress == 27: # escape key
            break
        elif keyPress == 32: # SpaceBar
            print("spaceBar!")
            break

```

```
cap.release()
```