INTELLIGENT ROBOTICS, PORTLAND STATE UNIVERSITY

Winter Report

Quaternion Attitude Control of a Simulated Airplane

Trenton Ruf

March 26, 2023

1. CHANGES TO LAST TERM'S CODE

I made the fuzzy-altitude controller from last term into a ROS service server. This is to control the attitude setpoint and enable/disable the controller from a seperate ROS node. My plan is to have the altitude and attitude controller nodes given commands by a master "State Machine" node.

2. QUATERNION CONTROL

I. controller layout

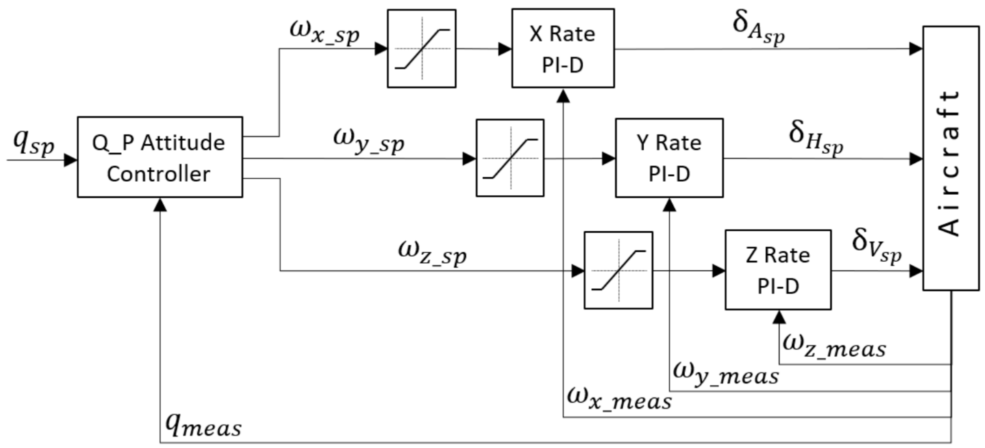


Figure 2.1: Quaternion-Based controller schematic from [1]

The quaternion attitude controller was modeled after the one described in Quaternion Attitude Control System of Highly Maneuverable Aircraft [1]. It is a cascading controller that takes a quaternion setpoint (q_{sp}) and quaterneon measured (q_{meas}) as the initial intputs to a proportional controller. The outputs of the proportional controller are angular velocity setpoints for their respective x, y, and z access PID controllers. The secondary inputs to these PIDs are the current measured angular velocities. The final outputs are the positions of the airplane control surfaces (rudder, aeleron, and elevator).

II. Translating to python

To perform quaternion math I installed the numpy-quaternion version 2020.11.2.17.0.49 since that was the last version to support python 2.7. A dependancy of numpy-quaternion is numba, which also must be explicitly installed to version 0.34.0. For the mathmatical notation for each step, please check pages 4-6 of Quaternion Attitude Control System of Highly Maneuverable Aircraft [1]. Below is a snippet of code from the file attitude_control_gazebo.py that shows my implementation of these steps.

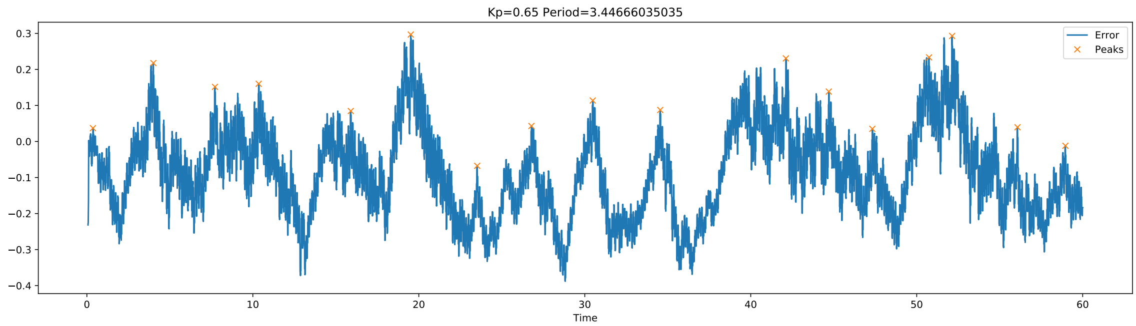
```
import numpy as np
import quaternion

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

# 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

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

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

aeleronPID.setpoint   = rateSetpoints.x
elevatorPID.setpoint = rateSetpoints.y
rudderPID.setpoint    = rateSetpoints.z

# Get the positions for each control surface
msg.header.stamp = rospy.Time.now()
msg.x = aeleronPID(measuredangVelocity.x)
msg.y = elevatorPID(measuredangVelocity.y)
msg.z = rudderPID(measuredangVelocity.z)
```

```
# Publish the ROS commands
publisher.publish(msg)
```

I've set the gain Kp for the first Proportion controller to be a static 1 for now. I'm thinking to adjust this gain depending on the airspeed of the plane. The control surfaces have more influence over the aircraft at higher airspeeds, so I think making this gain inversly proportional to airspeed will allow for better control.

III. Controller Tuning

I am attempting to tune each PID controller seperately. My plan was to find the "ultimate gain" as described by the Ziegler-Nicholas method. In which the Integral and Derivative Gains are set to 0, and the Proportional gain is adjusted until the system reaches a steady oscilation. The fist PID I tuned was for the elevator.I attempted to create a system to automatically find the ultimate gain. It works by recording the attitude error over time and determining when the error peaks appear. The more equadistant the peaks are then the more stable the oscillation. The peaks were found with the scipy.signal.find_peaks() function from the SciPy python library. It is a deterministic sytem where the plane was given an initial orientation of 8 degrees pitched up, a setpoint of level pitch, and 8m/s of initial velocity. The system would alter the gain value between 60 second trials to try to find the most stable oscillation. There was a Lot of troubleshooting during this process. For Example:



I thought the above graph showed a lot of noise so I added a 1-Dimensional Gaussian filter to help estimate actual error peaks. The filter is also a function of the SciPy library.
But it turns out it wasn't noise, but a consuquence of using gain values far too high.
Another Picture HERE
This shows a gain with a steady oscilation, but it is involves the airplane repeatedly stalling.
I added a criteria to be minimizing the average distance between the error peaks and troughs and this is the Ultimate gain my final system came up with. The peaks only show on the graph half way through the trial because I set it to analyse after 30 seconds into the trials, allowing it to stabilize before calculating any period.
Plugging the ultimate gain into the Ziegler-Nichols formula:
$K_p = 0.7K_u$
$K_i = 1.2K_u/T_u$
$K_d = 0.075K_uT_u$
The controller gives minimal overshoot and stabalizes with around 0 error. I am very satisfied with this result.

3. THINGS TO DO

When trying to tune the system I was initually using the rosflight simulated IMU. It has a built in Kalman filter, but even so the attitude would drift about 1 degree every 3 minutes. For a bandaid fix I changed from using the IMU to getting the exact attitude and velocity information from the Gazebo simulation's model. The IMU method will work but I will need to supliment the IMU data with a seperate truth reference. I'll try using the simulated magnatometer.

After tuning the aeleron and rudder PIDs, next term will be focused on integrating the altitude and attitude controllers with the eyebrow detection convolution network. My current plan is to use a quaternion rotation $p' = qpq^{-1}$ to control pitch up/down and bank left/right. I will have the altitude controller do the "neutral" state.

I mostly work with embedded C. Therefore I'm still learning a lot about python. Something irritating I've found is that python has no static variables. I got around that by making too many variables global. In the future I will make functions that requires a static variable to be part of a class instead.

I. 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

import rospy
from rosflight_msgs.msg import Command, Attitude
from nav_msgs.msg import Odometry
from simple_pid import PID
import numpy as np
import math
import quaternion # using version 2020.11.2.17.0.49
                        #numba dependency installed with "sudo apt install python-numba"
from scipy.signal import find_peaks
from scipy.ndimage.filters import gaussian_filter1d
import rospkg
from gazebo_msgs.msg import ModelState
from gazebo_msgs.srv import SetModelState
from gazebo_msgs.srv import GetModelState

from rosflight_control.srv import controller_set
import time
import matplotlib.pyplot as plt

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

enable = True;
# Initial setpoint value
attitudeSetpoint = np.quaternion(1,0,0,0)
#attitudeSetpoint = np.quaternion(0.991,0,0.131,0) # 15 degrees pitch up
#attitudeSetpoint = np.quaternion(0.998,0,0.07,0) # 8 degrees pitch up

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

# Atitude Proportional Controller Gain
```

```python
Kp = 1

# create PID controllers
#elevatorPID = PID(0.055,0,0, setpoint=0)
elevatorPID = PID(0.1,0,0, setpoint=0) # getting integral

#aeleronPID = PID(0.1,0,0, setpoint=0)
#rudderPID = PID(0.1,0,0, setpoint=0)

aeleronPID = PID(0,0,0, setpoint=0)
rudderPID = PID(0,0,0, 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 getFixedwingHeight():
    rospy.wait_for_service('/gazebo/get_model_state')
    try:
        get_state = rospy.ServiceProxy('/gazebo/get_model_state',GetModelState)
        resp = get_state('fixedwing',"")
        height = float(resp.pose.position.z)
        rospy.loginfo("fixedwing height: "+str(height))
        return height

    except rospy.ServiceExeption, e:
        print("Service call failed: %s" % e)



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 plotPID(x,y,gain):
    title = 'Kp='+str(gain)
    y_gauss = gaussian_filter1d(y, sigma=150)
    #peaks, properties = find_peaks(y_gauss,width=100,prominence=0.2)
    peaks, properties = find_peaks(y_gauss,prominence=0.0009)
    if len(peaks) != 0:
        peak_timestamps = [x[i] for i in peaks]
        peak_values = [y_gauss[i] for i in peaks]
        plt.plot(peak_timestamps, peak_values, 'x',label="Peaks")
        avg_period = (peak_timestamps[-1] - peak_timestamps[0]) / len(peak_timestamps)
        title += (' Period='+str(avg_period))
        rospy.loginfo("Plot Peaks: "+str(peaks))

    plt.rcParams["figure.figsize"] = (20,5)
    plt.plot(x,y, label = "Error Raw")
    plt.plot(x,y_gauss, label = "Error Gauss")
    plt.title(title)
    plt.xlabel('Time')
    plt.legend()
    #plt.savefig(str(gain)+'_PID.pdf')
    plt.show()
    plt.clf()


def findPeriodDeviation(_x,_y):
    deviation = -1
    y_gauss = gaussian_filter1d(_y, sigma=150)
    #peaks, properties = find_peaks(y_gauss,width=100,prominence=0.22)
    peaks, properties = find_peaks(y_gauss,prominence=0.0009)
    rospy.loginfo("Peak indicies: "+str(peaks))
    if len(peaks) > 2:
        peak_timestamps = [_x[i] for i in peaks if i > 3000]
        avg_period = (peak_timestamps[-1] - peak_timestamps[0]) / len(peak_timestamps)

        #peak_debug = [_y[i] for i in peaks]
        #rospy.loginfo("Peak error: "+str(peak_debug))

        deviation = 0
        for index,timestamp in enumerate(peak_timestamps[:-1]):
            period = peak_timestamps[index+1] - timestamp
            deviation += abs(avg_period - period)
        deviation/len(peak_timestamps[:-1])
    return deviation

def findHeightDeviation(_x,_y):
    deviation = -1
    peaks, properties = find_peaks(_y,width=100,prominence=0.22)
    rospy.loginfo("Peak indicies: "+str(peaks))
    if len(peaks) > 2:
        peak_heights = [_y[i] for i in peaks]
        avg_height = (peak_heights[-1] - peak_heights[0]) / len(peak_heights)
        deviation=0
        for index,height in enumerate(peak_heights):
            deviation += abs(avg_height - height)
        return deviation/len(peak_heights)
    return deviation

```

```
ult_x = []
ult_y = []
y=[]
x=[]
best_gain = -1
old_best_gain = -1
inc_gain = 0.01
curr_gain = 0.1
target_gain = 0.2
precision = 2 # decimal points of precision
precision_count = 1
trial_duration = 60
def findUltimateGain(pid,error):
    global best_gain
    global inc_gain
    global curr_gain
    global target_gain
    global old_best_gain
    global startTime
    global ult_x
    global ult_y
    global precision_count

    # Only log the error and the time until trial finished
    elapsed_time = time.time() - startTime
    x.append(elapsed_time)
    y.append(error)
    if elapsed_time > trial_duration:
        plotPID(x,y,curr_gain) # Debug only
        startTime = time.time()
        rospy.loginfo("curr_gain: "+str(curr_gain))
        rospy.loginfo("best_gain: "+str(best_gain))
        rospy.loginfo("target_gain: "+str(target_gain))
        rospy.loginfo("inc_gain: "+str(inc_gain))
    else:
        return 0


    # Test curr against best
    deviation = findPeriodDeviation(x,y)
    #deviation = findHeightDeviation(x,y)
    rospy.loginfo("deviation: "+str(deviation))
    if deviation >=0 and getFixedwingHeight() > 3:
        ult_deviation = findPeriodDeviation(ult_x,ult_y)
        #ult_deviation = findHeightDeviation(ult_x,ult_y)
        rospy.loginfo("ult_deviation: "+str(ult_deviation))
        if ult_deviation < 0 or deviation < ult_deviation:
            best_gain = curr_gain
            ult_x = list(x)
            ult_y = list(y)

    # Reset x, y
    del x[:]
    del y[:]

    # if reach end of test interval
    if round(curr_gain, precision) == round(target_gain, precision):
        if precision == precision_count:
            plotPID(ult_x,ult_y,best_gain)
            rospy.signal_shutdown(0)
```

```
            curr_gain = float(best_gain - inc_gain)
            target_gain = float(best_gain + inc_gain - (float(inc_gain) / 10))
            inc_gain = (float(inc_gain) / 10)
            old_best_gain = best_gain
            precision_count += 1

    # Increment step
    curr_gain += inc_gain
    if round(curr_gain, precision) == round(old_best_gain, precision):
        curr_gain += inc_gain

    # Disable PID controller
    #pid.auto_mode = False

    # Set new PID proportional gain
    pid.Kp = curr_gain

    # Reset model state
    resetState()

    # Enable PID controller
    #pid.set_auto_mode(True, last_output=0.0)

    return 0


ziegler_started=False
def testZieglerNichols(pid,error,ult_gain,ult_period):
    global ziegler_started
    global kp
    global ki
    global kd
    if not ziegler_started:
        kp=ult_gain * 0.6
        ki=ult_gain * 1.2 / ult_period
        kd=ult_gain * 0.075 * ult_period
        pid.Kp=kp
        pid.Ki=ki
        pid.Kd=kd
        ziegler_started = True
        resetState()

    elapsed_time = time.time() - startTime
    x.append(elapsed_time)
    y.append(error)
    if elapsed_time > trial_duration:
        plotPID(x,y, str(kp) + ' Ki='+str(ki) + ' Kd='+str(kd))
        rospy.signal_shutdown(0)



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 = np.multiply(Kp, attitudeError)
    attitudeSetpointDerivative = Kp * attitudeError

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

    aeleronPID.setpoint  = rateSetpoints.x
    elevatorPID.setpoint = rateSetpoints.y
    rudderPID.setpoint   = rateSetpoints.z

    msg.header.stamp = rospy.Time.now()
    msg.x = aeleronPID(attitudeData.twist.twist.angular.x) # potentially wrong
    msg.y = elevatorPID(attitudeData.twist.twist.angular.y)
    msg.z = rudderPID(attitudeData.twist.twist.angular.z) # potentially wrong
    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))
                )
    """

    #findUltimateGain(elevatorPID, attitudeError.y)
    testZieglerNichols(elevatorPID,attitudeError.y,0.11,0.075)


def getControl(request):
    global attitudeSetpoint
    global enable
    attitudeSetpoint = request.setPoint
    enable = request.enable
    return []

if __name__ == '__main__':
    try:

        # Init Node
        rospy.init_node('attitude_control')
```

```python

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


        # Create service
        service = rospy.Service("attitude_set" , controller_set, getControl)

        resetState()
        rospy.spin()

    except rospy.ROSInterruptException:
        pass
```

REFERENCES

[1] M. Gołąbek, M. Welcer, C. Szczepanski, M. Krawczyk, A. Zajdel, and K. Borodacz, "Quaternion attitude control system of highly maneuverable aircraft," *Electronics*, vol. 11, p. 3775, 11 2022.