```
import numpy as np
import softArmParam as P
from tf.transformations import quaternion_matrix
from tf.transformations import euler_from_matrix

class controllerPID:

    def __init__(self):
        # General control parameters
        self.kp = P.kp
        self.ki = P.ki
        self.kd = P.kd
        self.maxAngle = P.maxAngle
        self.beta = (2*P.sigma-P.Ts)/(2*P.sigma+P.Ts) #Used to create the dirty derivative
        self.Ts = P.Ts #The time between samples
        self.F = 0
        self.stringPulledPerStep = P.stringPulledPerStep
        self.L1 = P.L1

        # Variables for theta1, the vertical-angle component of the first joint
        self.theta1 = P.theta1
        self.theta1_prev = 0
        self.theta1dot = 0
        self.theta1dot_prev = 0
        self.theta1Error_prev = 0
        self.theta1Error = 0
        self.theta1Integrator = 0

        # Variables for theta2, the vertical-angle component of the second joint
        self.theta2 = P.theta2
        self.theta2_prev = 0
        self.theta2dot = 0
        self.theta2dot_prev = 0
        self.theta2Error_prev = 0
        self.theta2Error = 0
        self.theta2Integrator = 0

        # Variables for phi1, the horizontal-angle component of the first joint
        self.phi1 = P.phi1
        self.phi1_prev = 0
        self.phi1dot = 0
        self.phi1dot_prev = 0
        self.phi1Error_prev = 0
        self.phi1Error = 0
        self.phi1Integrator = 0

        # Variables for phi2, the horizontal-angle component of the second joint
        self.phi2 = P.phi2
        self.phi2_prev = 0
        self.phi2dot = 0
        self.phi2dot_prev = 0
        self.phi2Error_prev = 0
        self.phi2Error = 0
        self.phi2Integrator = 0

        #Rotation matrices variables
        self.initialMatrix1 = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
        self.initialMatrix2 = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
        self.groundFrame1 = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]

    # This function takes our desired angles, our actual angles, and returns the number of
steps needed to get there.
    def update(self, theta1_r, theta2_r, phi1_r, phi2_r, theta1, theta2, phi1, phi2):
        # Implement Derivative
        #self.theta1dot = self.beta * self.theta1dot_prev + (1 - self.beta) * ((theta1 - se
lf.theta1_prev) / self.Ts)
        #self.theta1dot_prev = self.theta1dot
        #self.theta1 = theta1
```

```
        #self.theta2dot = self.beta * self.theta2dot_prev + (1 - self.beta) * ((theta2 - se
lf.theta2_prev) / self.Ts)
        #self.theta2dot_prev = self.theta2dot
        #self.theta2 = theta2

        self.phi1dot = self.beta * self.phi1dot_prev + (1 - self.beta) * ((phi1 - self.phi1
_prev) / self.Ts)
        self.phi1dot_prev = self.phi1dot
        self.phi1 = phi1

        #self.phi2dot = self.beta * self.phi2dot_prev + (1 - self.beta) * ((phi2 - self.phi
2_prev) / self.Ts)
        #self.phi2dot_prev = self.phi2dot
        #self.phi2 = phi2

        # Implement Integrator
        #self.theta1Error = theta1_r - theta1
        #if abs(self.theta1dot) < 0.05:
        #    self.theta1Integrator = self.theta1Integrator + (self.Ts / 2) * (self.theta1Er
ror + self.theta1Error_prev)
        #self.theta1Error_prev = self.theta1Error

        #self.theta2Error = theta2_r - theta2
        #if abs(self.theta2dot) < 0.05:
        #    self.theta2Integrator = self.theta2Integrator + (self.Ts / 2) * (self.theta2Er
ror + self.theta2Error_prev)
        #self.theta2Error_prev = self.theta2Error

        self.phi1Error = phi1_r - phi1
        if abs(self.phi1dot) < 0.05:
            self.phi1Integrator = self.phi1Integrator + (self.Ts / 2) * (self.phi1Error + s
elf.phi1Error_prev)
        self.phi1Error_prev = self.phi1Error

        #self.phi2Error = phi2_r - phi2
        #if abs(self.phi2dot) < 0.05:
        #    self.phi2Integrator = self.phi2Integrator + (self.Ts / 2) * (self.phi2Error +
self.phi2Error_prev)
        #self.phi2Error_prev = self.phi2Error

        # Create steps with PID control loops
        #self.theta1_unsaturated = (self.kp * (theta1_r - theta1) - self.theta1dot * self.k
d) + self.ki * self.theta1Integrator
        #self.theta1_saturated = self.theta1_unsaturated #self.saturate(theta1_unsaturated)
 FIXME add saturation
        #theta1_steps = self.calculateSteps(theta1_saturated, theta1)
        #if theta1_steps < 0:
        #    bottomMotor1Steps = theta1_steps
        #    topMotor1Steps = theta1_steps / 2 # Step ratio for opposite motors
        #else:
        #    topMotor1Steps = theta1_steps
        #    bottomMotor1Steps = theta1_steps / 2  # Step ratio for opposite motors

        self.phi1_unsaturated = (self.kp * (phi1_r - phi1) - self.phi1dot * self.kd) + self
.ki * self.phi1Integrator
        phi1_saturated = self.phi1_unsaturated #self.saturate(phi1_unsaturated) FIXME add s
aturation
        phi1_steps = self.calculateSteps(phi1_saturated, phi1)
        phi1_steps = phi1_steps/4 # We go 4 steps towards our goal each time we publish

        #self.theta2_unsaturated = (self.kp * (theta2_r - theta2) - self.theta2dot * self.k
d) + self.ki * self.theta2Integrator
        #self.theta2_saturated = self.theta2_unsaturated  # self.saturate(theta2_unsaturate
d) FIXME add saturation
        #theta2_steps = self.calculateSteps(theta2_saturated, theta2)
        #if theta2_steps < 0:
        #    bottomMotor2Steps = theta2_steps
```

```
        #    topMotor2Steps = theta2_steps / 2  # Step ratio for opposite motors
        #else:
        #    topMotor2Steps = theta2_steps
        #    bottomMotor2Steps = theta2_steps / 2  # Step ratio for opposite motors

        #self.phi2_unsaturated = (self.kp * (phi2_r - phi2) - self.phi2dot * self.kd) + sel
f.ki * self.phi2Integrator
        #self.phi2_saturated = self.phi2_unsaturated # self.saturate(phi2_unsaturated) FIXM
E add saturation
        #phi2_steps = self.calculateSteps(phi2_saturated, phi2)
        #if phi2_steps < 0:
        #    leftMotor2Steps = phi2_steps
        #    rightMotor2Steps = phi2_steps / 2 # Step ratio for opposite motors
        #else:
        #    rightMotor2Steps = phi2_steps
        #    leftMotor2Steps = phi2_steps / 2  # Step ratio for opposite motors

        numSteps = ([0 , phi1_steps, 0, 0])
        #([topMotor1Steps, bottomMotor1Steps, leftMotor1Steps, rightMotor1Steps,
        #  topMotor2Steps, bottomMotor2Steps, leftMotor2Steps, rightMotor2Steps])
        return numSteps

        # Create steps open loop control
        # topMotor1Steps = self.calculateSteps(theta1_r, theta1)
        # bottomMotor1Steps = -topMotor1Steps
        # leftMotor1Steps = self.calculateSteps(phi1_r, phi1)
        # rightMotor1Steps = -leftMotor1Steps
        # topMotor2Steps = self.calculateSteps(theta2_r, theta2) + topMotor1Steps
        # bottomMotor2Steps = -topMotor2Steps
        # leftMotor2Steps = self.calculateSteps(phi2_r, phi2) + leftMotor1Steps
        # rightMotor2Steps = -leftMotor2Steps
        # numSteps = ([topMotor1Steps, bottomMotor1Steps, leftMotor1Steps, rightMotor1Steps
,
        #               topMotor2Steps, bottomMotor2Steps, leftMotor2Steps, rightMotor2Steps
])
        # return numSteps

    # This function uses a model to get an estimate on the number of steps needed to move t
o the desired position
    def calculateSteps(self, newAngle, currentAngle):
        newInverter = 1.0
        currInverter = 1.0
        if newAngle < 0.0:
            newInverter = -1.0
        if currentAngle < 0.0:
            currInverter = -1.0
        stepConstant = (2.0*self.L1/self.stringPulledPerStep)
        stepsToDesired = (1.0-np.sin((np.pi/180)*((180.0-abs(newAngle))/2.0)))
        stepsToCurrent = (1.0-np.sin((np.pi/180)*((180.0-abs(currentAngle))/2.0)))
        steps = stepConstant *(newInverter*stepsToDesired - currInverter*stepsToCurrent)
        return steps

    def convertToAngles(self, quaternion1, quaternion2):
        self.groundFrame1 = quaternion_matrix(quaternion1) #Converts quaternion to rotation
 matrix
        #groundFrame2 = quaternion_matrix(quaternion2) #Converts quaternion to rotation mat
rix

        #trueRotation1 = np.transpose(self.initialMatrix1)*groundFrame1
        matrix1 = np.transpose(self.initialMatrix1)
        matrix2 = self.groundFrame1[0:3, 0:3]
        res = [[0 for x in range(3)] for y in range(3)]

        # explicit for loops
        for i in range(len(matrix1)):
            for j in range(len(matrix2[0])):
                for k in range(len(matrix2)):
                     # resulted matrix
```

```
                    res[i][j] += matrix1[i][k] * matrix2[k][j]
        #baseRotation = numpy.transpose(self.initialMatrix2)*groundFrame2
        #trueRotation2 = numpy.transpose(trueRotation1)*baseRotation

        angles1 = euler_from_matrix(res)
        return angles1 #yaw, pitch, roll

    def initializePosition(self, quaternion1, quaternion2):
        self.initialMatrix1 = quaternion_matrix(quaternion1)
        self.initialMatrix1 = self.initialMatrix1[0:3, 0:3]
        #self.initialMatrix2 = quaternion_matrix(quaternion2)

    def getMatrix(self):
        return self.groundFrame1

    # This function is used to keep our system within feasible dynamics.
    def saturate(self, u):
        if abs(u) > self.limit:
            u = self.limit*np.sign(u)
        return u


if __name__ == '__main__':
    #Import message
    #Import GUI commands
    newPhi1 = newPhi2 = newTheta1 = newTheta2 = 0
    truePhi1 = truePhi2 = trueTheta1 = trueTheta2 = 0
    results = controllerPID.update(newTheta1, newTheta2, newPhi1, newPhi2, trueTheta1, true
Theta2, truePhi1, truePhi2)
    #Publish new commands
```