1_steer_by_wire

October 15, 2024

1 Hardware lab 1: implementing your control law / Steer by wire

In this first hardware lab we are first going to work without the passive arms of our robot and only focus on the motors. The objective of the lab is to informally play with the motor API and understand the possible issues that can arise when trying to control accurately a motor.

Concretely, we will play with the notion of Proportional control (discussed in next week's lecture) and use this to reproduce a steer-by-wire system.

1.1 Preliminaries

First, listen to Garry or Katy for the first few minutes of the lab while they provide some guidance on the platform.

1.2 Communicating with the device

The DICE machines are connected to the platform using a CAN device. This is hidden within a high-level API but you can have a look at the code if you are interested in seeing how all of this works.

To work, open a python console, or edit a file and execute the code that follows. The CAN bus protocol does not work very well with Jupyter notebooks and keyboard interruptions won't work either.

The API is presented in AROMotorControlAPI.md. The only methods that we need are `readPosition` and `applyTorqueToMotor`.

Each method has a `motorid` parameter, taking the value 1 or 2, which points to the motor that you wish to control.

Therefore, to read the position (joint angle value) or a motor, simply call:

[6]:
```
from motor_control.AROMotorControl import AROMotorControl

mc = AROMotorControl()
mc.readPosition(motorid=1)
```

[6]:
```
22.630000000004657
```

The returned value is an angle value in degrees, between 0 and 360.

1.3 A torque controlled motor

The motor is controlled by sending a current command to it, which results in its rotation. The rotational force produced is called a torque (more on this during the lectures).

1.3.1 Torque vs current

This paragraph is just for general knowledge. The torque generated is proportional to the current through the relationship $ = k I $, with τ the torque, I the current (in Amperes), and k a constant given by the manufacturer. In our case, the torque constant provided by the manufacturer is 0.16, which trivially gives you $I = \frac{\tau}{0.16}$. We will trust the manufacturer although this might change from a motor to the other (and is yet another source of approximation).

1.3.2 Sending a command

To send current command you are required to use an exception handling mechanismthat always safely terminates by resetting the control command. Unfortunately this can't be encapsulated in a function as this results in packages loss.

The `run_until` method defined there also allows you to call N times the same method while ensuring an accurate delay `dt` expires before being called again.

Let us use `run_until` to send a constant torque of 0.02 Nm for 2 seconds

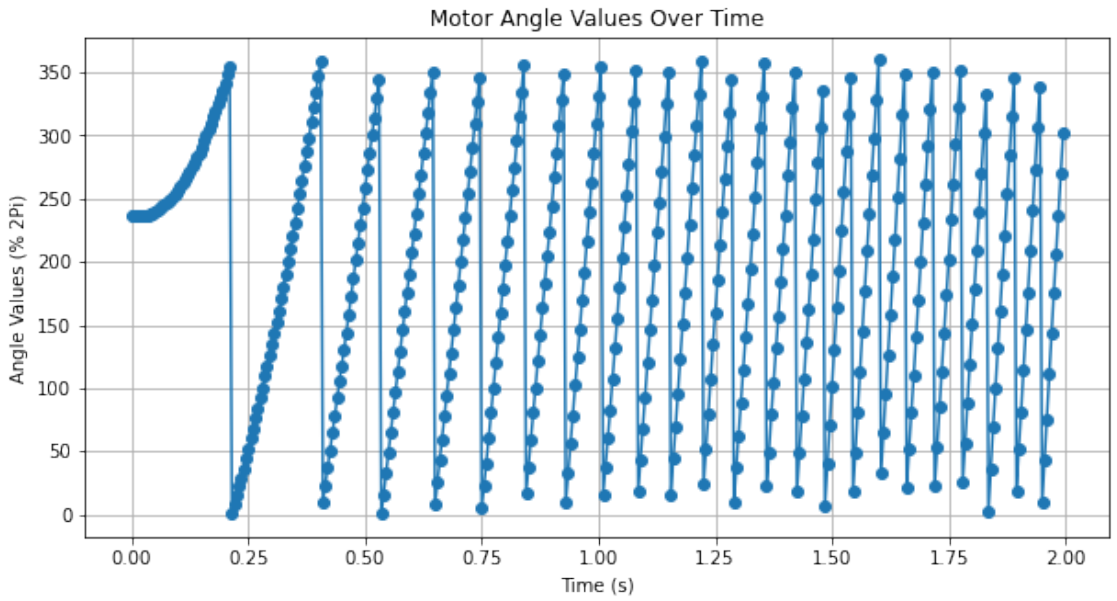
[ ]:
```
from template import run_until

dt = 0.005
N = int(2. / dt)


try:
    run_until(mc.applyTorqueToMotor, N=N, dt=0.005, motorid=1, torque=0.02)
except KeyboardInterrupt:
    print("KeyboardInterrupt received, stopping motors...")
except Exception as e:
    print(f"an error occurred: {e}")
finally:
    mc.applyTorqueToMotor(1, 0)
    mc.applyTorqueToMotor(2, 0)
    print("motors stopped!")
```

Lets make a simple plot of the angle value evolution over time while we are sending this command

[9]:
```
import matplotlib.pyplot as plt

anglevalues =[]

def store_values_and_apply_torques(motorid, torque):
    global anglevalues
    mc.applyTorqueToMotor(motorid=motorid, torque=torque)
```

```
    anglevalues+= [mc.readPosition(motorid)]

try:
    run_until(store_values_and_apply_torques, N=N, dt=0.005, motorid=1,␣
 ↪torque=0.02)
except KeyboardInterrupt:
    print("KeyboardInterrupt received, stopping motors...")
except Exception as e:
    print(f"an error occurred: {e}")
finally:
    mc.applyTorqueToMotor(1, 0)
    mc.applyTorqueToMotor(2, 0)
    print("motors stopped!")



time_values = [i * dt for i in range(len(anglevalues))]

# Plotting the angle values
plt.figure(figsize=(10, 5))
plt.plot(time_values, anglevalues, marker='o', linestyle='-')
plt.title('Motor Angle Values Over Time')
plt.xlabel('Time (s)')
plt.ylabel('Angle Values (% 2Pi)')
plt.grid()
plt.show()
```

```
motors stopped!
```



We can observe that as time progresses the angle values change more rapidly, which seems intuitive of course. But we are going to have to deal with these dynamic behaviours as the course progresses.

1.4 Our first control law

Let us know assume that we want the motor to reach a specific angle value target. We can start by implementing what we call a "proportional" control. We ll measure the distance between the current position and the target position and apply a torque proportional to that distance. This should remind you of inverse kinematics and least square in general. For this let's create the class `PController`, for Proportional control.

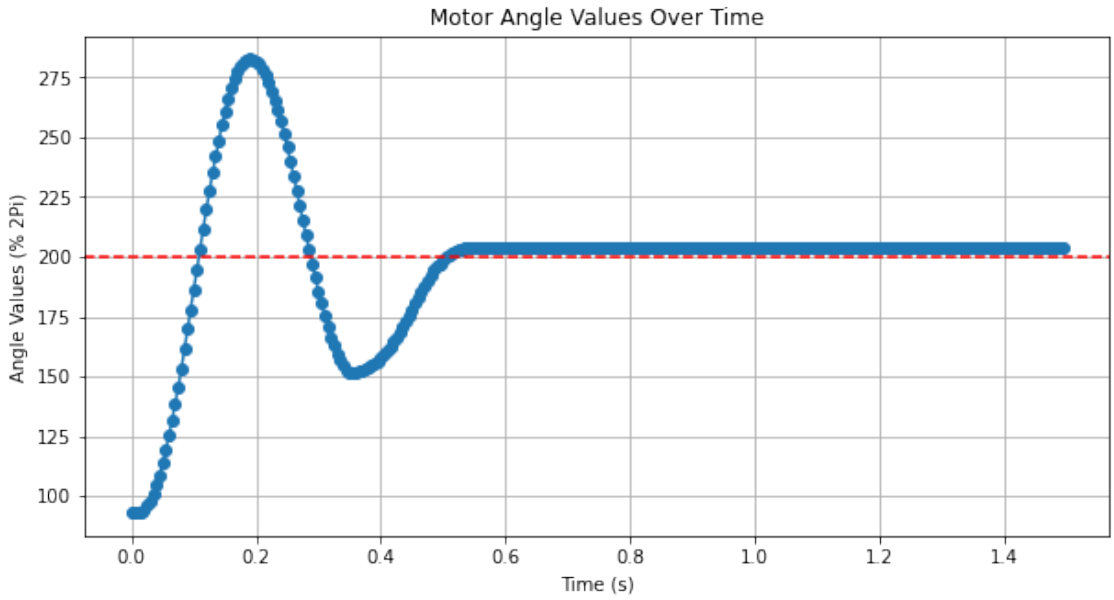
It is parametrised by a K_p variable that will adjust the strength of the signal sent to the motor.

As there is a lot of friction, we'll need to ensure that a minimal signal is sent when the error is not 0 as the motors tend to not move at all before 0.02 nM. This will be implemented in the `clip` function

[1]:
```
def clip(output):
    outabs = abs(output)
    if outabs < 1e-4:
        return 0
    clipped = max(min(outabs, 0.1), 0.02)
    return clipped if output > 0 else -clipped

class PController:

    def __init__(self, Kp):
        self.Kp = Kp

    def shortest_path_error(self, target, current):
        diff = ( target - current + 180 ) % 360 - 180;
        if diff < -180:
                diff = diff + 360
        if (current + diff) % 360 == target:
                return diff
        else:
                return -diff

    def compute(self, target, current):
        error = self.shortest_path_error(target, current)
        output = self.Kp*error
        return clip(output)
```

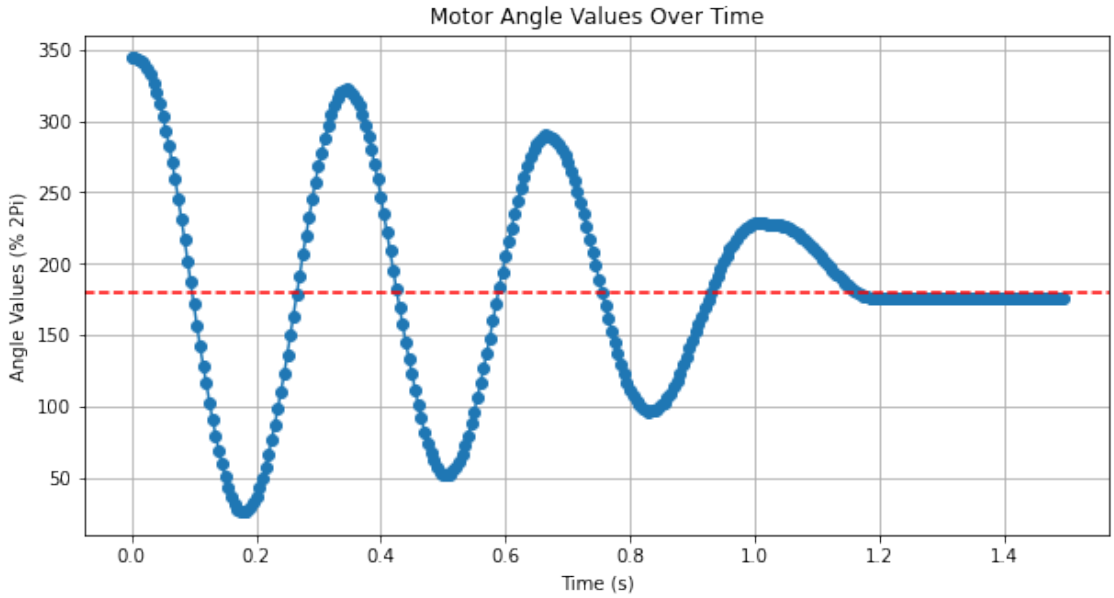
We can now define a method `goTo` for our first motor, that will apply the control law repeatidly for a given period of time

[130]:
```
def goTo(controller, target, time = 1., dt = 0.005, motorid =1):
    anglevalues =[]
    N = (int)(time / dt)

    def oneStep():
        nonlocal anglevalues
        currentAngle = mc.readPosition(motorid)
        anglevalues+=[currentAngle]
        tau = controller.compute(target,currentAngle)
        mc.applyTorqueToMotor(motorid,tau)

    run_until(oneStep, N=N, dt=dt)

    mc.applyTorqueToMotor(1, 0)
    mc.applyTorqueToMotor(2, 0)

    time_values = [i * dt for i in range(len(anglevalues))]
    # Plotting the angle values
    plt.figure(figsize=(10, 5))
    plt.plot(time_values, anglevalues, marker='o', linestyle='-')
    plt.axhline(y=target, color='r', linestyle='--', label='Target Value')  #␣
 ↪Add horizontal line for target
    plt.title('Motor Angle Values Over Time')
    plt.xlabel('Time (s)')
    plt.ylabel('Angle Values (% 2Pi)')
    plt.grid()
    plt.show()

try:

    pc = PController(0.00016)
    goTo(pc, 200, time = 1.5)
except KeyboardInterrupt:
    print("KeyboardInterrupt received, stopping motors...")
except Exception as e:
    print(f"an error occurred: {e}")
finally:
    mc.applyTorqueToMotor(1, 0)
    mc.applyTorqueToMotor(2, 0)
    print("motors stopped!")
```

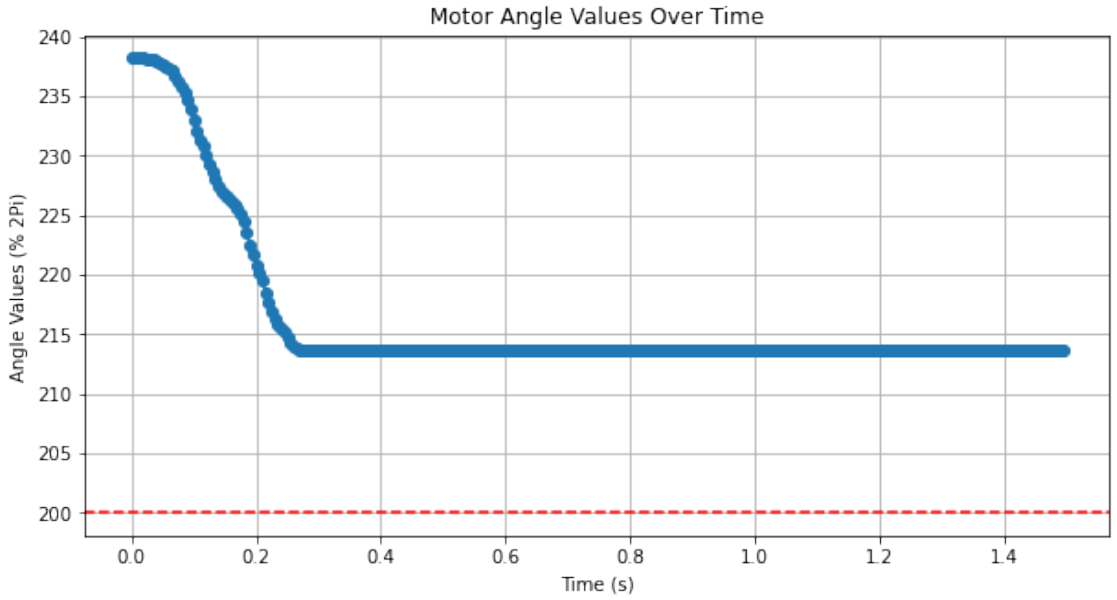



```
motors stopped!
```

Experiment with different values of K_p. A common issue you will encounter is overshooting, when you go beyond the desired angle and creates an oscillating behaviour:



This can be compensated somehow by reducing K_p, but you then there is a risk of never reaching the target (here represented by the red line):



1.5 Adding a damping factor

To mitigate the overshooting behaviour, we can try to emulate the following reasoning: + If you are going fast towards the target, try to slow down not to overshoot + If you are not going fast enough try to accelerate more

This information can be given by tracking the derivative of the error, which we can approximate by looking at the error variation over time. We can now write a PD controller, which includes a gain proportional to the derivative of the error. (Thanks to Jingyang and Zhixin for figuring out that computing the derivative over a longer horizon allowed to compensate for the encoder reading errors);

[ ]:
```
class PDController:

    def __init__(self, Kp, Kd):
        self.Kp = Kp
        self.Kd = Kd
        self.prev_error = 0
        self.positions = []


    def reset(self):
        self.positions = []

    def shortest_path_error(self, target, current):
        diff = ( target - current + 180 ) % 360 - 180;
        if diff < -180:
                diff = diff + 360
        if (current + diff) % 360 == target:
                return diff
```

```
        else:
                return -diff

    def compute(self, target, current, dt):
        self.positions.append(current)
        error = self.shortest_path_error(target, current)
        d_error = 0
        if (len(self.positions) > 2):
                d_error = (self.positions[-1] - self.positions[-3]) / (2*dt)
        output = self.Kp*error - self.Kd * d_error
        return clip(output)


try:
    pdc = PDController(0.00016,0.001*dt)
    goTo(pdc, 200, time = 1.5)
except KeyboardInterrupt:
    print("KeyboardInterrupt received, stopping motors...")
except Exception as e:
    print(f"an error occurred: {e}")
finally:
    mc.applyTorqueToMotor(1, 0)
    mc.applyTorqueToMotor(2, 0)
    print("motors stopped!")
```

Try to tune your K_p and K_d gains until you obtain a behaviour that satisfies you. Usually we talk about PD control or PID control, where I is an integral term. In our domain we rarely use the integral gain, but this will be discussed during the lecture and implemented in next week's tutorial.

Now we can move on to our first application

1.6 Question

Now, write a 30s control loop that does the following: + Motor 1 is configured to track the position of motor 2 + Motor 2 is configured to track the position of motor 1

Manually mess around with the motors while the loop is running and check that things behave as you expect. A similar system is implemented in some of the recent cars where the steering wheel and the wheels are no longer mechanically connected (see https://en.wikipedia.org/wiki/Steer-by-wire)

In [ ]:
```
from motor_control.AROMotorControl import AROMotorControl
mc = AROMotorControl()
mc.readPosition(motorid=2)
```

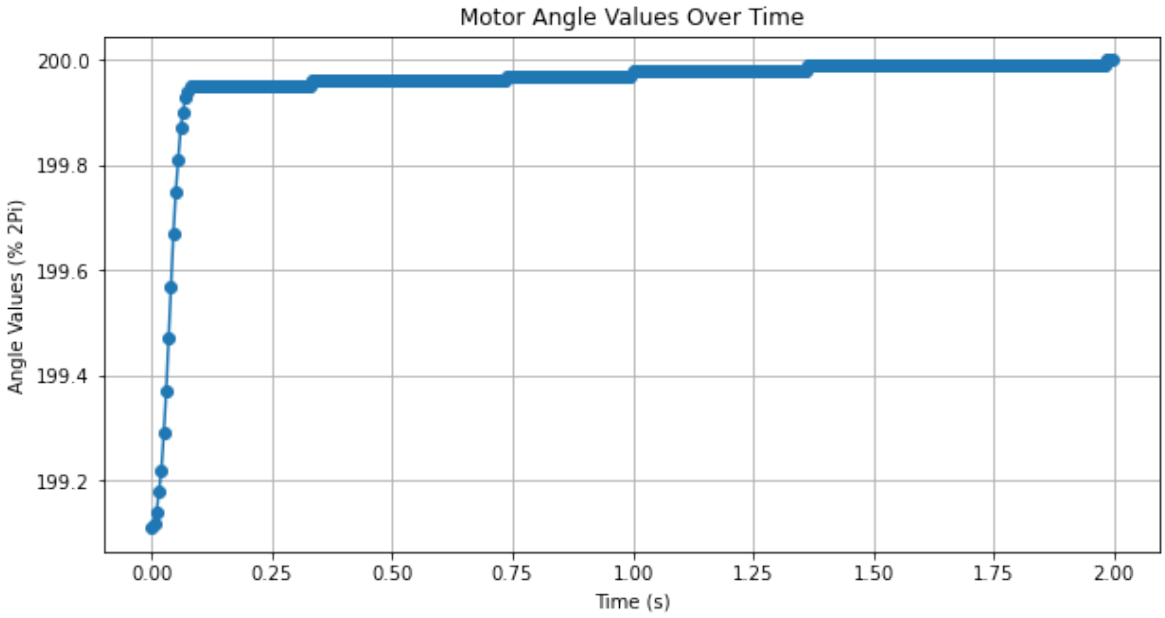
Out[ ]: 199.11000000000058

In [ ]:
```
from template import run_until
dt = 0.005
# Calculate how many iterations the motor should run
# based on a 2-second duration and the time step.
N = int(2. / dt)
try:
    # runs motor1 at 0.02 torque for 2 seconds
    run_until(mc.applyTorqueToMotor, N=N, dt=0.005, motorid=1, torque=0.0
except KeyboardInterrupt:
    print("KeyboardInterrupt received, stopping motors...")
except Exception as e:
    print(f"an error occurred: {e}")
finally:
    mc.applyTorqueToMotor(1, 0) # stop the motor
    mc.applyTorqueToMotor(2, 0)
    # applyTorqueToMotor takes the arguments:
    # motorid: the motor to apply the torque to
    # torque: the torque to apply to the motor
    print("motors stopped!")
```

motors stopped!

In [ ]:
```
import matplotlib.pyplot as plt
anglevalues =[]
def store_values_and_apply_torques(motorid, torque):
    global anglevalues
    mc.applyTorqueToMotor(motorid=motorid, torque=torque)
    anglevalues.append(mc.readPosition(motorid=motorid))

try:
    run_until(store_values_and_apply_torques, N=N, dt=0.005, motorid=2, t
except KeyboardInterrupt:
    print("KeyboardInterrupt received, stopping motors...")
except Exception as e:
    print(f"an error occurred: {e}")
finally:
    mc.applyTorqueToMotor(1, 0)
    mc.applyTorqueToMotor(2, 0)
    print("motors stopped!")


time_values = [i * dt for i in range(len(anglevalues))]

# Plotting the angle values
plt.figure(figsize=(10, 5))
plt.plot(time_values, anglevalues, marker='o', linestyle='-')
plt.title('Motor Angle Values Over Time')
plt.xlabel('Time (s)')
plt.ylabel('Angle Values (% 2Pi)')
plt.grid()
plt.show()
```

motors stopped!



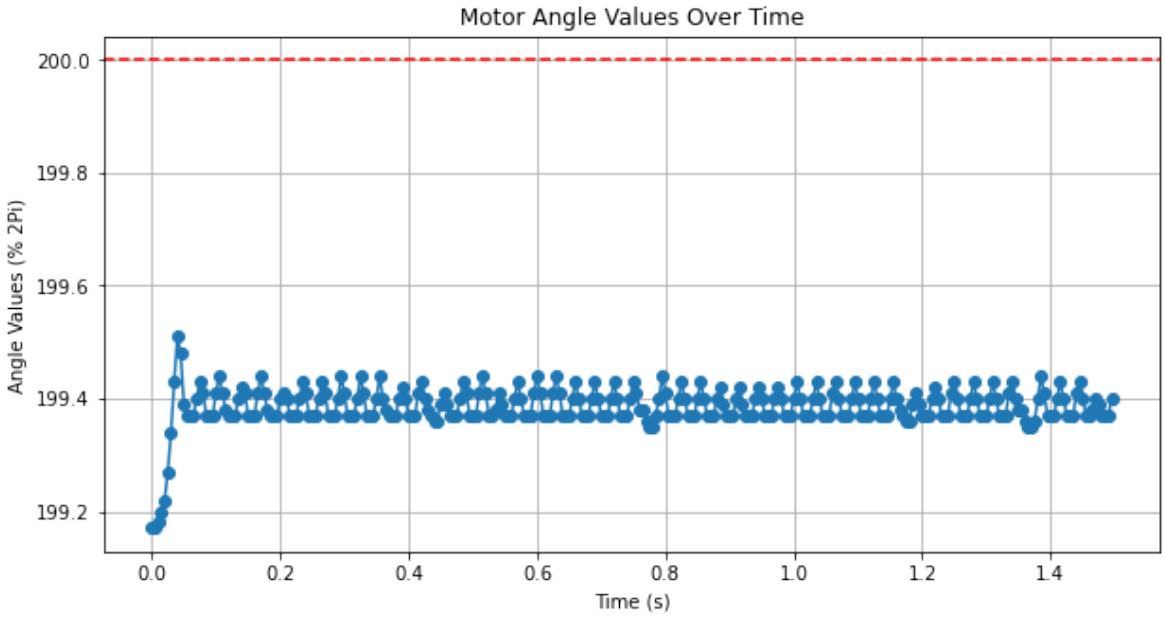
In [ ]:

```python
def clip(output):
    """This function clips the output to a range of -0.1 to -0.02 and 0.0
    this is to ensure that, if there is a lot of friction, a minimal sign
    when the error is not 0 but small, as the motors tend not ro move at
    before 0.02 nM of torque is applied.
    """
    outabs = abs(output)
    if outabs < 1e-4:
        return 0
    clipped = max(min(outabs, 0.1), 0.02)
    return clipped if output > 0 else -clipped

class PController:
    def __init__(self, Kp, Kd):
        self.Kp = Kp
        self.Kd = Kd
        self.elapsed = 0
        self.prev_error = 0

    def reset(self):
        self.elapsed = 0

    def shortest_path_error(self, target, current):
        # difference between target and current
        # you sum 180 to the difference and then take the modulo 360 to g
        # then you subtract 180 to get the difference between the target
        diff = ( target - current + 180 ) % 360 - 180
        # e.g. target = 360, current = 0, diff = (360 - 0 + 180) % 360 -
        # e.g. target = 180, current = 90, diff = (180 - 90 + 180) % 360
        # e.g. target = 90, current = 180, diff = (90 - 180 + 180) % 360
        # e.g. target = 0, current = 360, diff = (0 - 360 + 180) % 360 -
            # in this case, the next line will sum 360 to the diff and re

        if diff < -180:
            diff = diff + 360
        if (current + diff) % 360 == target:
            # if the target is reached, return 0 (the difference)
            return diff
        else:
            # if the target is not reached, return the negative differenc
```

```
            return -diff

    def compute(self, target, current, dt):
        error = self.shortest_path_error(target, current)
        d_error = (error - self.prev_error) if self.elapsed >= dt else 0
        output = self.Kp*error  + self.Kd*d_error
        # Kp is the proportional gain.
        # It is a constant that determines how much the output will chang
        # the error. If Kp is too high, the system will be unstable,
        # if it is too low, the system will be slow.
        if (self.elapsed < 2*dt):
            output = clip(output)
        else:
            output = clip(output)
        self.elapsed += dt
        return output
```

In [ ]:

```
def goTo(controller, target, time = 1., dt = 0.005, motorid =1):
    anglevalues =[]
    N = (int)(time / dt) # number of iterations

    def oneStep():
        nonlocal anglevalues
        currentAngle = mc.readPosition(motorid)
        anglevalues+=[currentAngle]
        tau = controller.compute(target,currentAngle, dt) # returns the t

        mc.applyTorqueToMotor(motorid,tau)

    controller.reset()
    run_until(oneStep, N=N, dt=dt)

    mc.applyTorqueToMotor(1, 0)
    mc.applyTorqueToMotor(2, 0)

    time_values = [i * dt for i in range(len(anglevalues))]
    # Plotting the angle values
    plt.figure(figsize=(10, 5))
    plt.plot(time_values, anglevalues, marker='o', linestyle='-')
    plt.axhline(y=target, color='r', linestyle='--', label='Target Value'
    plt.title('Motor Angle Values Over Time')
    plt.xlabel('Time (s)')
    plt.ylabel('Angle Values (% 2Pi)')
    plt.grid()
    plt.show()

try:
    pc = PController(0.00016, 0.000)
    goTo(pc, 200, time = 1.5, motorid=2)
except KeyboardInterrupt:
    print("KeyboardInterrupt received, stopping motors...")
except Exception as e:
    print(f"an error occurred: {e}")
finally:
    mc.applyTorqueToMotor(1, 0)
    mc.applyTorqueToMotor(2, 0)
    print("motors stopped!")
```




```
motors stopped!
```

In [ ]:
```
print(mc.readPosition(2))
```

```
199.23999999999978
```

In [ ]:
```
"""Now, write a 30s control loop that does the following: + Motor 1 is co
of motor 2 + Motor 2 is configured to track the position of motor 1
Manually mess around with the motors while the loop is running and check
you expect. A similar system is implemented in some of the recent cars wh
the wheels are no longer mechanically connected (see https://en.wikipedia
dt = 0.005
N = (int)(30. / dt) # number of iterations
pc1 = PController(0.00016, 0.000)
pc2 = PController(0.00016, 0.000)
def controlLoop():

    def oneStep():
        angle1 = mc.readPosition(1)
        angle2 = mc.readPosition(2)
        tau1 = pc1.compute(angle2, angle1, dt)
        tau2 = pc2.compute(angle1, angle2, dt)
        mc.applyTorqueToMotor(1, tau1)
        mc.applyTorqueToMotor(2, tau2)
    run_until(oneStep, N=N, dt=dt)

    mc.applyTorqueToMotor(1, 0)
    mc.applyTorqueToMotor(2, 0)


controlLoop()
```

In [ ]:
```
mc.applyTorqueToMotor(1, 0)
mc.applyTorqueToMotor(2, 0)
```

Out[ ]:
```
(28, 0, 0, 147)
```