Código:

Controlador PID:

import time  # Importa a biblioteca time para trabalhar com tempo

class PIDController:  # Define a classe do controlador PID

    """Entradas:

    Kp = Ganho porpocional aka amordecedor

    Ki = erro acumulado ao longo do tempo

    Kd = Ganho derivativo aka instantanio

    Saídas - control\_signal:

    """

    def \_\_init\_\_(self, kp, ki, kd):

        self.kp = kp

        self.ki = ki

        self.kd = kd

        self.previous\_error = 0  # Erro anterior (para o cálculo derivativo)

        self.integral = 0  # Integral do erro (acumula o erro ao longo do tempo)

        self.last\_time = time.time()  # Tempo da última execução (para calcular delta\_time)

    def compute(self, setpoint, current\_depth):

        """Calcula o sinal de controle baseado no erro de profundidade.

        int: control\_signal: antes de ser tratado

        Args:

            setpoint (int): profundidade desejada

            current\_depth (int): profundidade atual

        Returns:

            int: thrust: Sinal pronto para ser colocado

                no código associado ao controlo das hélices

        """

        error = setpoint - current\_depth

        current\_time = time.time()

        delta\_time = current\_time - self.last\_time

        # Integral do erro: acumula o erro multiplicado pelo tempo decorrido

        self.integral += error \* delta\_time

        # Derivativo do erro: calcula a variação do erro dividida pelo tempo decorrido

        d\_error = (error - self.previous\_error) / delta\_time if delta\_time > 0 else 0

        # Sinal de controle PID: combina os três termos (proporcional, integral e derivativo)

        control\_signal = (self.kp \* error) + (self.ki \* self.integral) + (self.kd \* d\_error)

        # Atualiza os valores de erro e tempo para a próxima execução

        self.previous\_error = error

        self.last\_time = current\_time

        # Limit thrust range IMPORTANTE REVER

        thrust = max(min(control\_signal, 1), -1)

        return thrust

# Simulação de um loop de controle

if \_\_name\_\_ == "\_\_main\_\_":  # Verifica se o script está sendo executado diretamente

    kp = 1.0  # Ganho proporcional, ajuste conforme necessário

    ki = 0.05  # Ganho integral, ajuste conforme necessário

    kd = 0.1  # Ganho derivativo, ajuste conforme necessário

    setpoint\_depth = float(input("Digite a profundidade desejada (m): "))  # Entrada do usuário para a profundidade desejada

    current\_depth = 0.0  # Profundidade inicial (simulada como 0)

    controller = PIDController(kp, ki, kd)  # Cria uma instância do controlador PID

    while True:  # Loop contínuo para ajuste dinâmico

        new\_setpoint = input("Digite nova profundidade ou pressione Enter para manter: ")  # Permite que o usuário ajuste o setpoint

        if new\_setpoint:  # Se o usuário fornecer um novo setpoint, atualiza

            setpoint\_depth = float(new\_setpoint)

        control\_output = controller.compute(setpoint\_depth, current\_depth)  # Calcula o sinal de controle com o PID

        # Simula a resposta do veículo ajustando a profundidade com base no sinal de controle

        current\_depth += control\_output \* 0.1  # Ajuste simples da profundidade baseada no controle (multiplicado por 0.1)

        # Exibe a profundidade atual, o sinal de controle e o objetivo desejado

        print(f"Profundidade Atual: {current\_depth:.2f} m | Controle: {control\_output:.2f} | Objetivo: {setpoint\_depth:.2f} m")

        time.sleep(0.1)  # Pausa o loop por 0.1 segundos para simular uma atualização contínua

Código teste PID:

import time

from ROV import ROVsensors

# Inicializações

sensors = ROVsensors()

pingSensor = sensors.connectPing1D("192.168.2.2", 9090)

mavLink = sensors.connectMAVLINK("0.0.0.0", 14550)

# Constantes PID

Kp = 2.0

Ki = 0.0

Kd = 1.0

# Variáveis do PID

prev\_error = 0.0

integral = 0.0

def get\_current\_depth():

    """Lê a profundidade atual do sensor Ping1D com verificação de None"""

    for \_ in range(5):  # Tenta até 5 vezes

        data = sensors.get\_ping1d\_data(ping\_sensor=pingSensor)

        if data and "distance" in data:

            return data["distance"] / 1000  # Convertendo mm para metros

        time.sleep(0.1)

    raise RuntimeError("Falha ao obter leitura válida do sensor Ping1D")

def depth\_hold(target\_depth):

    """Mantém profundidade usando controle PID"""

    global prev\_error, integral

    print(f"Iniciando controle de profundidade para: {target\_depth:.2f} m")

    try:

        while True:

            current\_depth = get\_current\_depth()

            error = target\_depth - current\_depth

            integral += error

            derivative = error - prev\_error

            prev\_error = error

            # Cálculo do sinal de controle (thrust)

            thrust = (Kp \* error) + (Ki \* integral) + (Kd \* derivative)

            thrust = max(min(thrust, 1), -1)  # Limita entre -1 e 1

            # Envia comando de empuxo

            sensors.set\_thrust(thrust, connectionMAVLINK=mavLink)

            # Debug

            print(f"Alvo: {target\_depth:.2f} m | Atual: {current\_depth:.2f} m | Thrust: {thrust:.2f}")

            time.sleep(0.1)

    except KeyboardInterrupt:

        print("Controle de profundidade interrompido pelo usuário.")

def profundidadeInicial():

    """Lê a profundidade inicial para ser usada como referência"""

    profundidade = get\_current\_depth()

    print(f"Profundidade Inicial: {profundidade:.2f} m")

    return profundidade

if \_\_name\_\_ == "\_\_main\_\_":

    profundidade\_alvo = profundidadeInicial()

    depth\_hold(profundidade\_alvo)