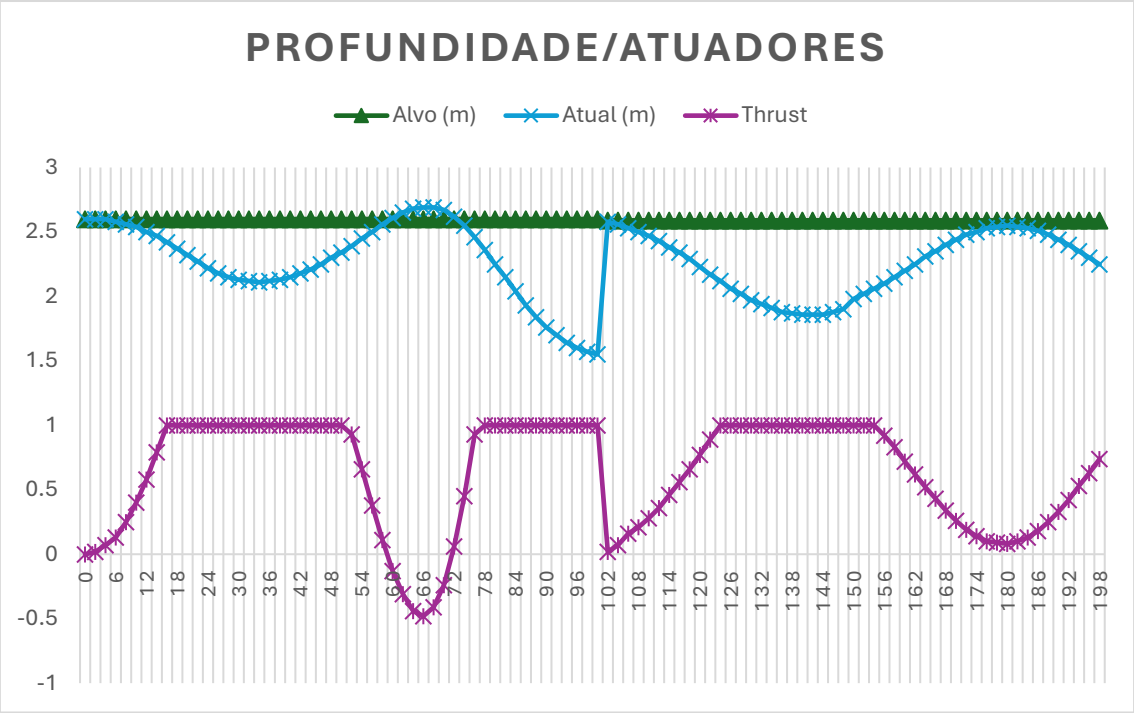

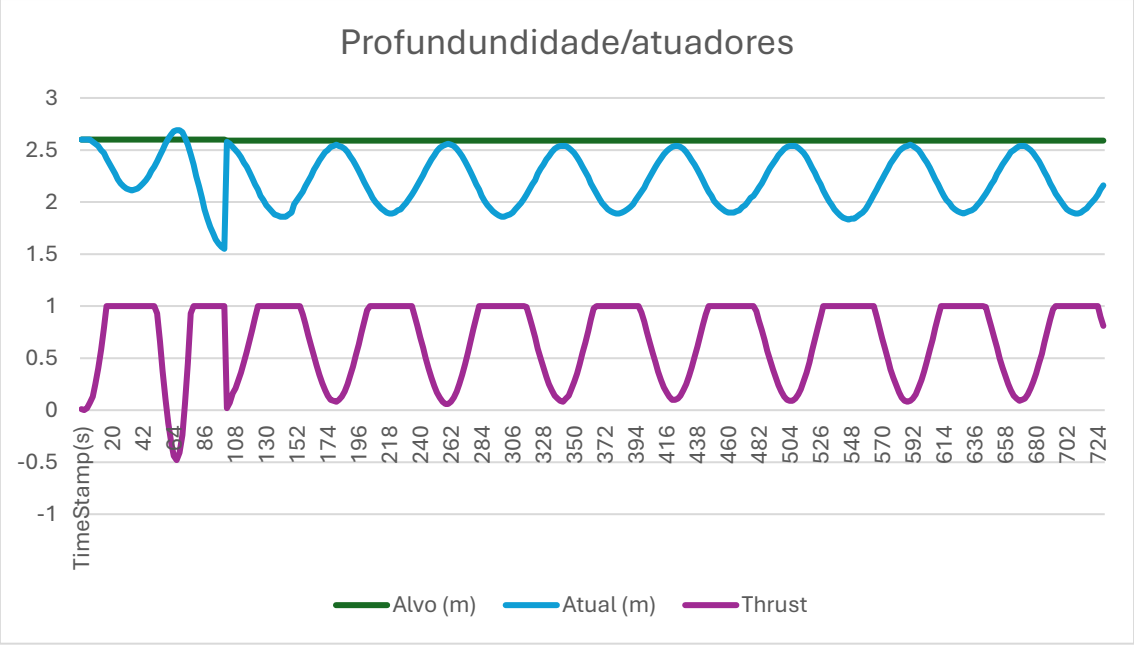

Código:

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)
```