Kyungpook National University

School of Electronics Engineering

자율시스템 설계

	Student ID: 2021115004
#보고서 3	**Name: 손창우**
	강의담당교수: 박찬은

1. 강의 내용 요약 및 시뮬레이션 목적

PID 제어기의 필요성

1. **정확한 목표 도달**

   단순히 고정된 속도를 주면 목표 위치에 도달하지 못하거나 오버슈트(초과 도달)할 수 있다. PID는 현재 오차를 반영해 실시간으로 제어 입력을 조정하므로 정밀 제어가 가능하다.

2. **불안정한 시스템 안정화**

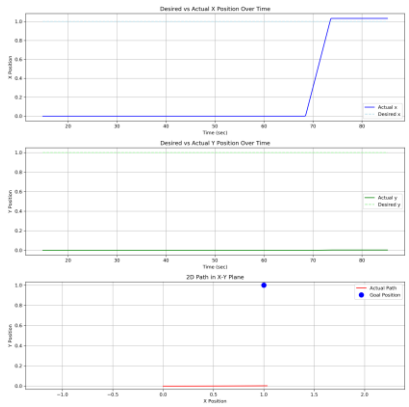
   비례 제어(P)만 쓰면 오차는 줄지만 잔류 오차가 남는다. 여기에 적분(I)을 더하면 잔류 오차를 없애주고, 미분(D)은 급격한 변화(진동)를 억제해 시스템을 **빠르고 부드럽게 수렴**시킨다.

3. **실제 환경에 강인한 제어**

   마찰, 센서 지연, 외란(장애물 등)처럼 이상적인 모델에서 벗어나는 상황에서도 PID는 실시간으로 보정해 비교적 강인하게 동작할 수 있다.

2. 시뮬레이션 결과

1) 일정 거리 주행 후 정지하는 제어기

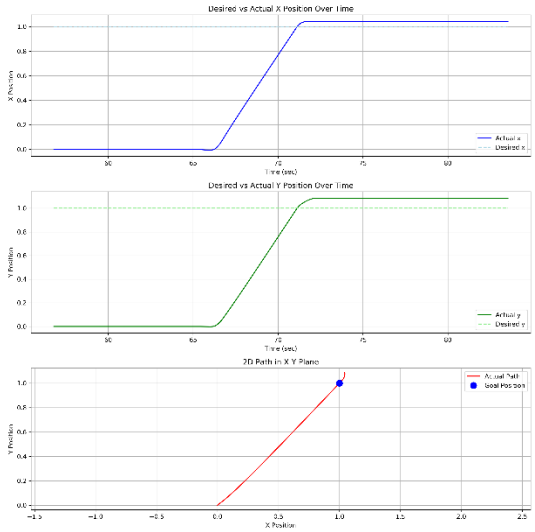



```python
#!/usr/bin/env python3
import rospy
import math
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion

class TurtleBot3PositionController:
    def __init__(self):
        # Initialize ROS node
        rospy.init_node('turtlebot3_position_controller')

        # Publishers and subscribers
        self.cmd_vel_pub = rospy.Publisher('/cmd_vel', Twist, queue_size=10)
        self.odom_sub = rospy.Subscriber('/odom', Odometry, self.odom_callback)

        # Control rate (10 Hz)
        self.rate = rospy.Rate(10)

        # Initialize position and orientation variables
        self.x = 0.0
        self.y = 0.0
        self.theta = 0.0

        # Initial position (for calculating distance moved)
        self.initial_x = None
        self.initial_y = None

        # Target distance to move (in meters)
        self.target_distance = 1.0  # Default: 1m

        # Control parameters
        self.linear_speed = 0.2  # m/s
        self.angular_speed = 0.5  # rad/s

        # Wait for odometry data to start
        rospy.sleep(1)

    def odom_callback(self, msg):
        # Get current position
        self.x = msg.pose.pose.position.x
        self.y = msg.pose.pose.position.y

        # Get orientation
        orientation_q = msg.pose.pose.orientation
        orientation_list = [orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w]
        _, _, self.theta = euler_from_quaternion(orientation_list)

        # Store initial position on first callback
        if self.initial_x is None:
            self.initial_x = self.x
            self.initial_y = self.y

    def stop(self):
        # Create stop command
        cmd = Twist()
        cmd.linear.x = 0.0
        cmd.angular.z = 0.0

        # Send stop command multiple times to ensure the robot stops
        for _ in range(5):
            self.cmd_vel_pub.publish(cmd)
            self.rate.sleep()

        rospy.loginfo("Robot stopped")
```

```python
    def rotate(self, angle):
        # Create rotation command
        cmd = Twist()
        cmd.linear.x = 0.0
        cmd.angular.z = self.angular_speed if angle > 0 else -self.angular_speed

        # Calculate time required to rotate
        time_to_rotate = abs(angle) / abs(cmd.angular.z)

        # Start time
        start_time = rospy.Time.now().to_sec()

        # Rotate for calculated time
        while (rospy.Time.now().to_sec() - start_time) < time_to_rotate:
            self.cmd_vel_pub.publish(cmd)
            self.rate.sleep()

        # Stop after rotation
        self.stop()
        rospy.loginfo(f"Rotated {angle} radians")

    def move_straight(self, distance):
        # Reset initial position
        self.initial_x = self.x
        self.initial_y = self.y

        # Create movement command
        cmd = Twist()
        cmd.linear.x = self.linear_speed
        cmd.angular.z = 0.0

        # Move until target distance is reached
        while not rospy.is_shutdown():
            # Calculate distance moved
            distance_moved = math.sqrt((self.x - self.initial_x)**2 + (self.y - self.initial_y)**2)

            # Check if target distance reached
            if distance_moved >= distance:
                break

            # Publish command
            self.cmd_vel_pub.publish(cmd)
            self.rate.sleep()

        # Stop after reaching target
        self.stop()
        rospy.loginfo(f"Moved {distance} meters")

    def run(self):
        rospy.loginfo("TurtleBot3 Position Controller Started")

        try:
            # First adjust orientation (optional)
            # self.rotate(math.pi/2)  # 90 degrees

            # Then move straight
            self.move_straight(self.target_distance)

            rospy.loginfo("Movement completed")

        except rospy.ROSInterruptException:
            pass

if __name__ == '__main__':
    try:
        controller = TurtleBot3PositionController()
        controller.run()
    except rospy.ROSInterruptException:
        pass
```

self.initial_x, self.initial_y 로 초기 위치를 저장하고 , /odom 데이터를 통해 현재 위치와 거리 계산 . 목표 거리만큼 이동한 경우 self.stop()으로 정지하는 조건문에 의한 단순 주행알고리즘을 구현했다.

2) 목표 좌표로 이동하는 제어기 만들기



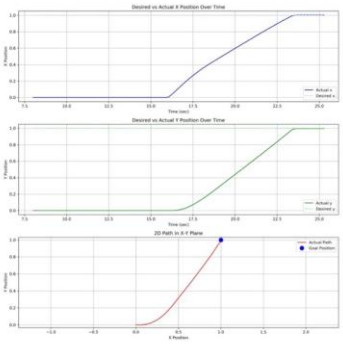

```python
import rospy
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
import math
import tf.transformations

# 목표 위치 설정
goal_x = 1.0
goal_y = 1.0

class GazeboGoToGoalController:
    def __init__(self):
        rospy.init_node('gazebo_go_to_goal_controller', anonymous=True)

        # 현재 위치와 방향 저장 변수
        self.x = 0.0
        self.y = 0.0
        self.theta = 0.0

        # Odometry 구독 (로봇의 위치와 방향 정보 얻기)
        rospy.Subscriber('/odom', Odometry, self.odom_callback)

        # 로봇 제어를 위한 Publisher
        self.pub = rospy.Publisher('/cmd_vel', Twist, queue_size=10)

        # 제어 주기 설정 (10Hz)
        self.rate = rospy.Rate(10)

        # 초기화 완료 메시지
        rospy.loginfo("Gazebo Go-To-Goal Controller initialized")

        # 제어 파라미터 (필요시 조정)
        self.linear_speed = 0.3  # 선속도 (m/s) - 더 느리게 조정
        self.angular_p_gain = 1.5  # 각속도 P 제어 게인

        # 목표 도달 정밀도 설정
        self.distance_threshold = 0.05  # 5cm 이내 도달 기준

    def odom_callback(self, msg):
        # Odometry 메시지에서 위치 정보 추출
        self.x = msg.pose.pose.position.x
        self.y = msg.pose.pose.position.y

        # 쿼터니언에서 오일러 각으로 변환하여 theta(yaw) 추출
        orientation_q = msg.pose.pose.orientation
        orientation_list = [orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w]
        _, _, self.theta = tf.transformations.euler_from_quaternion(orientation_list)
```

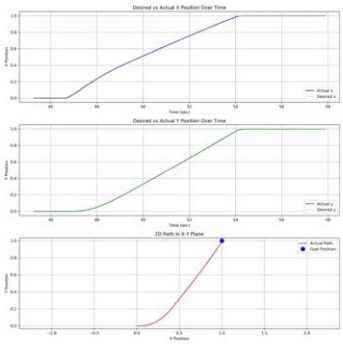
```python
    def run(self):
        # 초기 센서 데이터(Odometry 메시지)를 받기 위한 시간
        rospy.sleep(1.0)

        vel = Twist()

        while not rospy.is_shutdown():
            # 오차 계산 계산
            error_x = goal_x - self.x
            error_y = goal_y - self.y
            distance = math.sqrt(error_x**2 + error_y**2)

            # 목표 방향 계산 (atan2 사용)
            desired_theta = math.atan2(error_y, error_x)
            angle_diff = desired_theta - self.theta

            # 각도 정규화 (-pi ~ pi)
            angle_diff = math.atan2(math.sin(angle_diff), math.cos(angle_diff))

            # 로봇 제어 - 목표 거리 임계값보다 크면 계속 전진
            if distance > self.distance_threshold:
                # 각도 오차가 크면 먼저 회전
                if abs(angle_diff) > 0.3:
                    vel.linear.x = 0.0
                    vel.angular.z = self.angular_p_gain * angle_diff
                else:
                    # 각도가 거의 맞으면 직진하면서 방향 조정
                    vel.linear.x = self.linear_speed
                    vel.angular.z = self.angular_p_gain * angle_diff
            else:
                # 목표에 도달하면 정지
                vel.linear.x = 0.0
                vel.angular.z = 0.0

            # 디버깅 로그
            rospy.loginfo(f"[pose] x={self.x:.2f}, y={self.y:.2f}, θ={self.theta:.2f} | dist={distance:.2f}, angle_err={angle_diff:.2f}")

            # 목표 도달 시 루프 종료
            if distance < self.distance_threshold:
                rospy.loginfo("✅ 목표에 도착했습니다!")
                self.pub.publish(Twist())  # 정지 명령
                break

            # 속도 명령 발행
            self.pub.publish(vel)
            self.rate.sleep()

if __name__ == '__main__':
    try:
        GazeboGoToGoalController().run()
    except rospy.ROSInterruptException:
        pass
```

이 코드는 Gazebo 에서 목표 위치(1.0, 1.0)로 이동하는 P 제어 기반 자율 주행 제어기로, /odom 을 통해 현재 위치와 방향을 받아오고 linear.x 는 고정 속도, angular.z 는 각도 오차에 비례해 제어한다. 방향 오차가 크면 회전만 수행하고, 오차가 작아지면 직진과 회전을 동시에 수행하도록 구현돼 있다. 단순한 P 제어기만 사용했기 때문에 동작은 안정적이나, 회전과 전진 사이 시간이 다소 지연될 수 있다.

초기에는 회전만 하다가 각도 정렬 후 직진한다. 하단 그래프에서는 실제 경로가 목표 좌표로 수렴하며 거의 직선 형태를 이루는 것이 보이며, 이는 등속 직진과 P 제어 회전이 잘 작동했음을 의미한다.

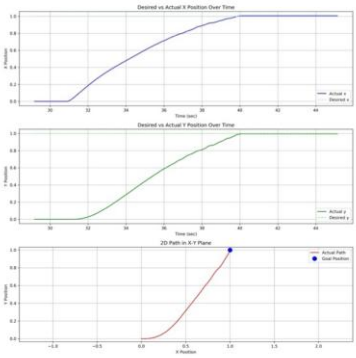
3) 목표 좌표로 이동하는 PID 제어기 만들기





P: 30 I: 0.1 D: 0.1

P: 3 I: 0.1 D: 0.1



P: 6 I: 0.5 D: 0.5

```python
import rospy
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
import math
from time import time
from tf.transformations import euler_from_quaternion

# 목표 위치 설정
goal_x = 1  # Gazebo에서의 적절한 값으로 조정
goal_y = 1  # Gazebo에서의 적절한 값으로 조정

class GoToGoalController:
    def __init__(self):
        rospy.init_node('go_to_goal_controller', anonymous=True)
        self.x = 0.0
        self.y = 0.0
        self.theta = 0.0
        rospy.Subscriber('/odom', Odometry, self.odom_callback)
        self.pub = rospy.Publisher('/cmd_vel', Twist, queue_size=10)
        self.rate = rospy.Rate(10)

        # 시간 초기화
        self.prev_time = time()

        # distance PID
        self.integral_distance = 0.0
        self.prev_distance_error = 0.0

        # angle PID
        self.integral_angle = 0.0
        self.prev_angle_error = 0.0

    def odom_callback(self, msg):
        # Odometry 메시지에서 위치 추출
        self.x = msg.pose.pose.position.x
        self.y = msg.pose.pose.position.y

        # 쿼터니언을 오일러 각으로 변환
        orientation_q = msg.pose.pose.orientation
        orientation_list = [orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w]
        roll, pitch, self.theta = euler_from_quaternion(orientation_list)
```

```python
    def run(self):
        vel = Twist()

        while not rospy.is_shutdown():
            # 시간 차이 계산
            current_time = time()
            dt = current_time - self.prev_time
            self.prev_time = current_time

            # 오차 벡터 계산
            error_x = goal_x - self.x
            error_y = goal_y - self.y
            distance = math.sqrt(error_x**2 + error_y**2)

            # 방향 오차 계산
            desired_theta = math.atan2(error_y, error_x)
            angle_diff = desired_theta - self.theta
            angle_diff = math.atan2(math.sin(angle_diff), math.cos(angle_diff))  # 정규화 [-π, π]

            # --- PID: distance ---
            self.integral_distance += distance * dt
            d_distance = (distance - self.prev_distance_error) / dt if dt > 0 else 0.0
            vx = 30 * distance + 0.1 * self.integral_distance + 0.1 * d_distance  # Gazebo에 맞게 개인 조정
            self.prev_distance_error = distance

            # --- PID: angle ---
            self.integral_angle += angle_diff * dt
            d_angle = (angle_diff - self.prev_angle_error) / dt if dt > 0 else 0.0
            wz = 1.0 * angle_diff + 0.05 * self.integral_angle + 0 * d_angle  # Gazebo에 맞게 개인 조정
            self.prev_angle_error = angle_diff

            # 임계값 기반 정지
            if distance < 0.05:  # Gazebo에 맞게 임계값 조정
                vx = 0.0
            if abs(angle_diff) < 0.05:  # Gazebo에 맞게 임계값 조정
                wz = 0.0

            # 속도 제한 (Gazebo에서는 더 낮은 값이 안전)
            vx = min(vx, 0.2)  # 최대 선속도 제한
            wz = max(min(wz, 0.5), -0.5)  # 최대 각속도 제한

            # 속도 명령 발행
            vel.linear.x = vx
            vel.angular.z = wz
            self.pub.publish(vel)

            # 디버깅 로그
            rospy.loginfo(f"[pose] x={self.x:.2f}, y={self.y:.2f}, θ={self.theta:.2f}")
            rospy.loginfo(f"[error] dist={distance:.3f}, angle_diff={angle_diff:.3f}")
            rospy.loginfo(f"[cmd  ] vx={vx:.3f}, wz={wz:.3f}")

            # 목표 도달 시 정지
            if distance < 0.05:  # Gazebo에 맞게 임계값 조정
                rospy.loginfo("✅ 목표에 도착했습니다!")
                self.pub.publish(Twist())  # 정지 명령
                break

            self.rate.sleep()

if __name__ == '__main__':
    try:
        GoToGoalController().run()
    except rospy.ROSInterruptException:
        pass
```

핵심 제어 방식은 거리 오차를 입력으로 하는 선속도 PID 제어와, 방향 오차(목표 각도와 현재 로봇의 각도 차이)를 입력으로 하는 각속도 PI 제어로 구성되어 있다. 로봇은 실시간으로 Odometry 데이터를 받아 위치와 자세를 계산하며, 거리 오차와 방향 오차를 기반으로 선속도(vx)와 각속도(wz)를 결정하고 이를

/cmd_vel 토픽으로 발행하여 이동한다.

실험에서는 선속도 제어기의 게인만 바꾸어 3가지 조합(P, I, D)을 비교하였으며, 각 조합별로 목표 지점에 도달하는 방식과 경로, 응답 특성 등을 분석하였다. 각속도 게인은 모두 동일하게 유지되었으며, 오차가 작아지면 속도를 제한하고, 특정 임계값 이하로 수렴하면 로봇을 완전히 정지시키는 로직이 포함되어 있다.

첫 번째 실험(P=30, I=0.1, D=0.1)은 고 P 게인을 사용하여 가장 빠른 수렴 속도를 보였다. 거리 오차가 발생하는 즉시 선속도가 최대값(0.2 m/s)까지 포화되며, 직선 형태의 경로를 따라 빠르게 목표 지점으로 접근한다. 오버슈트 없이 빠르게 수렴하는 응답 특성으로 보아, 실질적으로 크리티컬 댐핑에 가까운 동작을 보인다. 강한 비례 항으로 인해 속도가 급격히 상승하지만 제한 조건 덕분에 불안정한 진동은 발생하지 않았으며, 시간 성능이 중요할 경우 가장 적합한 세트로 판단된다.

두 번째 실험(P=3, I=0.1, D=0.1)은 가장 낮은 P 값을 사용하였으며, 전체적으로 움직임이 매우 부드럽고 완만한 곡선을 따라 접근하는 모습을 보인다. 초기 속도 상승이 느리기 때문에 정착 시간은 다소 길며, 제어 출력 자체가 낮아 속도 제한에 거의 도달하지 않는 모습도 관찰된다. 과도한 진동이나 과주행 없이 매우 안정적으로 목표 지점에 도달하였으며, 비교적 안전하고 보수적인 제어가 필요한 상황에 적합하다. 단, 응답성은 낮기 때문에 빠른 이동을 요구하는 상황에서는 비효율적일 수 있다.

세 번째 실험(P=6, I=0.5, D=0.5)은 중간 정도의 P 값에 비해 I와 D 항을 크게 잡은 경우다. 적분 항이 빠르게 누적되면서 중후반부까지 비교적 높은 선속도를 유지하고, 미분 항이 종단에서 속도를 적극적으로 감쇠시키기 때문에 전체 궤적이 S자 형태의 완만한 가속·감속 곡선을 그린다. 결과적으로 진입 구간에서는 세트 A보다 느리지만, 가속이 일단 붙은 이후에는 안정적인 정속 구간이 형성되어 지나치게 긴 주행 시간으로 이어지지는 않았다. 다만 종단에서 미세한 오실레이션(0.95 m 근처에서 흔들리는 현상)이 관찰되었는데, 이는 D 항이 잡음을 어느 정도 증폭한 영향으로 풀이된다. 따라서 이 세트는 움직임을 최대한 부드럽게 하고 기계적 충격을 줄이고자 할 때 유리하지만, 적분 포화(Integral wind-up)에 대한 대비가 없으면 더 긴 경로에서 불안정성이 커질 수 있다.

종합하면, 시간 성능이 가장 중요한 과제라면 P=30, I=0.1, D=0.1 조합이 권장되고, 안전하고 보수적인 주행이 필요할 때는 P=3, I=0.1, D=0.1 조합이 무난하다.

3. 결론 및 느낀점

이번 실습을 통해 이론으로 학습했던 PID 제어기의 파라미터 조정 원리를 실제 터틀봇 시뮬레이

션에 적용해보며, 제어기의 동작 원리와 각 게인이 시스템 반응에 미치는 영향을 직관적으로 체감할 수 있었다. 특히 P, I, D 각각의 요소가 응답 속도, 오버슈트, 안정성 등에 어떻게 작용하는지를 실시간으로 확인하면서 제어 이론이 단순한 공식에 머무르지 않고 실제 시스템과 연결되어 있다는 사실을 생생하게 느낄 수 있었다.

또한 이번 실습을 계기로 단순한 PID 제어를 넘어, 보다 복잡한 제어기법들—예를 들어 칼만 필터를 통한 노이즈 제거나, 센서 데이터를 기반으로 한 자율주행 알고리즘—역시 시뮬레이션 환경에서 구현해보고 싶다는 생각이 들었다. 향후에는 센서 융합 기반의 지능형 제어 기법을 실습에 확장하여, 보다 복잡한 환경에서도 안정적으로 동작할 수 있는 자율 시스템 설계를 시도해보고자 한다.