SeBAR (Self-Balancing Assistant Robot)

윤대영 · 권민환 · 온유리

지도교수: 김용태

한경국립대학교 전가전자제어공학과(ICT로봇공학전공)

E-mail : toueg1236@gmail,com, minhwan2315@naver.com, on16aug@gmail.com

요 약

본 연구는 자율적으로 균형을 유지하며 이동할 수 있는 자가 균형 로봇(Self-Balancing Robot)을 개발하는 것을 목표로 한다. 로봇은 자이로센서를 이용한 센서 데이터 수집, PID 제어 알고리즘을 통한 균형 유지 메커니즘을 통합하여 이를 수행한다. 기존의 자가 균형 로봇은 주로 안정된 실내 환경에 초점을 맞추었지만, 본 로봇은 폼 체인지를 통해 험난한 환경에서도 안정적으로 작동할 수 있도록 설계되었다. 자가 균형 로봇은 균형을 유지하며 사용자를 따라 이동하고, 제스처를 인식하여 컨트롤러 없이도 안정적으로 작동할 수 있다. 로봇 제어에는 ROS2 운영체제를 사용하였다.

자가 균형 로봇, 객체 인식, 폼 체인지, 주행

gyroscope, PID control, ROS2, Object detection

Ⅰ. 서 론

1.1 개발동기 및 필요성

최근 산업용 로봇의 수요가 급증하고 있다. 이는 생산성 향상, 인력 문제 해결, 위험 작업 최소화 등의 장점을 제공하기 때문이다. 그림1은 산업용 로봇 및 서비스용 로봇 시장 규모 및 전망을 나타낸 그래프이다. 그림1에서 볼 수 있듯이, 산업용 로봇 및 서비스용 로봇 시장의 규모는 연평균 성장률(CAGR)이 각각 25.2%와 46.2%에 이를 것으로 예상된다. 이러한 성장은 산업 전반에 걸쳐 로봇 도입의 필요성을 더욱 부각시키고 있다.

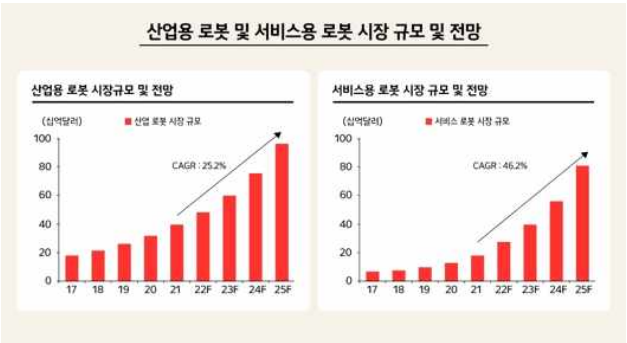


그림1. 산업용 로봇 시장 규모 및 전망

그러나 산업 재해는 여전히 중요한 문제로 남아있다. 산업 현장에서 작업자는 신체적, 정신적 피로를 겪으며, 이는 안전사고로 이어질 수 있다. 중대산업재해의 발생 현황을 보면, 여전히 많은 사고가 발생하고 있으며, 이는 작업 환경의 안전성 향상이 시급함을 보여준다. 밸런스 로봇의 도입은 이러한 문제를 해결하는 데 중요한 역할을 할 수 있다. 로봇은 작업자의 신체적 부담을 줄이고, 위험한 작업을 대신 수행하며, 안정성을 높일 수 있다.

따라서, 본 연구에서는 자율적으로 균형을 유지하며 이동할 수 있는 자가 균형 로봇(Self-Balancing Robot)을 개발하고자 한다. 이러한 로봇은 산업 현장에서 작업자의 부담을 덜고, 안전한 작업 환경을 조성하며, 다양한 응용 분야에서 활용될 수 있을 것이다.

1.2 기존 방법 및 문제점

1.2.1 제한된 자율성

기존 자가 균형 로봇은 사용자가 컨트롤러로 제어해야 한다. 이는 사용자가 직접적인 제어를 지속적으로 수행해야 하므로, 로봇의 자율성이 크게 제한된다. 이러한 제어 방식은 로봇의 활용성을 저하시킬 뿐만 아니라, 사용자의 피로를 증가시키고, 장시간 사용 시 효율성을 떨어뜨린다.

1.2.2 불안정한 균형 제어

기존 자가 균형 로봇은 대부분 2륜 구조로 제작되어 균형을 유지하도록 설계되었다. 그러나 이러한 설계는 균형 제어가 필요 없는 상황에서도 불안정함을 초래할 수 있다. 특히, 로봇이 정지해 있을 때나, 이동 중에 외부 충격을 받을 때 불안정한 상태가 발생할 수 있다. 이는 로봇의 안정성을 저하시켜, 다양한 환경에서의 활용에 제약을 가져온다.

따라서 이러한 문제를 해결하기 위해 자가 균형 로봇의 설계 및 제어 시스템을 개선할 필요성이 있다. 기존 로봇의 제한된 자율성을 극복하기 위해 본 연구에서는 사용자가 컨트롤러를 사용하지 않고도 자율적으로 균형을 유지하고 이동할 수 있는 시스템을 개발하고자 한다. 또한, 기존의 2륜 구조로 인한 불안정을 해결하기 위해 본 로봇은 다양한 환경에서도 안정적으로 작동할 수 있도록 폼 체인지 기능을 도입하였다. 이러한 자가 균형 로봇은 실내외 환경 모두에서 안정적으로 작동할 수 있으며, 개인 운송, 물류, 교육 등 다양한 응용 분야에서 활용 가능성을 극대화할 것이다.

1.3 본 작품의 독창성

1.3.1 균형을 잡으며 민감한 물건 배송

본 연구의 자가 균형 로봇은 민감한 물건을 안전하게 운송할 수 있도록 설계되었다. 로봇은 자이로센서와 가속도계를 이용한 정밀한 균형 제어 시스템을 갖추고 있어, 이동 중에도 물건의 흔들림을 최소화할 수 있다. 이는 특히 섬세한 제품이나 깨지기 쉬운 물건을 운송할 때 큰 장점을 제공하며, 고품질의 배송을 보장한다. 또한, 이 기능을 통해 로봇은 물류, 운송 등 다양한 분야에서 활용 가능성을 극대화할 수 있다.

1.3.2 객체 인식 기술 활용

본 연구에서 개발한 자가 균형 로봇은 객체 인식 기술을 접목하여 기능을 향상시켰다. 로봇은 독립적으로 작업을 수행할 수 있으며, 이는 작업 효율성과 유연성을 크게 향상시킨다. 객체 인식 기술을 통해 로봇은 사람의 손 모양을 인지하고, 주행을 제어할 수 있으며, 인지한 사람을 따라 이동할 수 있다. 이러한 첨단 기술의 통합으로 로봇은 작업을 간소화하고 다양한 응용 분야에서 높은 생산성을 보장할 수 있다.

1.3.3 폼 체인지 기능을 통한 다양한 환경 적응

본 연구의 자가 균형 로봇은 실내 및 실외 환경 모두에서 효과적으로 작동할 수 있도록 폼 체인지 기능을 갖추고 있다. 로봇은 울퉁불퉁한 바닥이나 장애물이 많은 환경에서도 안정적인 작동을 위해 6륜 모드로 전환하고, 평탄한 지형이나 좁은 공간에서는 2륜 모드로 민첩하게 움직인다. 이 설계는 로봇이 다양한 환경 조건에서도 부드럽게 움직일 수 있도록 하며, 이를 통해 실내외 모든 환경에서 안정적이고 효율적으로 사용할 수 있다. 다양한 지형과 작업 조건에 적응할 수 있는 이 기능은 로봇의 응용 가능성을 크게 확장시킨다.

II. 작품설명

2.1 전체 시스템 구성

2.1.1 전체 시스템 구성도

그림 2는 로봇의 전체 시스템 구성도를 나타낸다. ZED 카메라는 SBC(AGX Xavier)와 연결되어 ROS 토픽을 통해 3D 매핑 및 인식 기능을 제공한다. 또한, Iahrs[1](자이로 센서)도 SBC와 ROS 토픽으로 연결되어 로봇의 자세 정보를 제공한다. SBC는 Dynamixel 프로토콜을 통해 MCU(OpenCR1.0)에 명령을 전달하며, MCU는 다시 Dynamixel 프로토콜을 이용하여 로봇의 동작을 제어하는 Dynamixel MX-106 모터를 구동한다. 이 통신을 통해 로봇의 움직임을 원활하게 조정하고 제어할 수 있다.

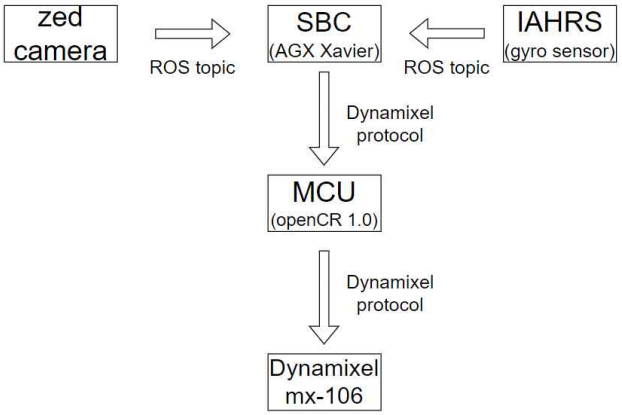


그림 2.전체 시스템 구성도

2.1.2 시스템 흐름도

그림 3은 로봇 작동 시스템의 흐름도이다. 로봇이 초기화 및 시작되면, 센서 값을 수집하여 로봇의 균형을 유지하는 제어 시스템에 전달한다. 동시에 카메라는 제스처 인식을 통해 로봇의 동작 명령을 수신한다. 인식된 제스처에 따라 모터 제어 신호가 발행되고, 이를 통해 모터가 동작하여 로봇이 이동하거나 작업을 수행한다. 이러한 시스템 흐름은 로봇이 안정적으로 균형을 유지하면서도, 사용자 제스처에 따라 유연하게 동작할 수 있도록 한다. 이를 통해 로봇은 다양한 환경에서 효과적으로 작동하며, 작업의 효율성과 유연성을 크게 향상시킬 수 있다.

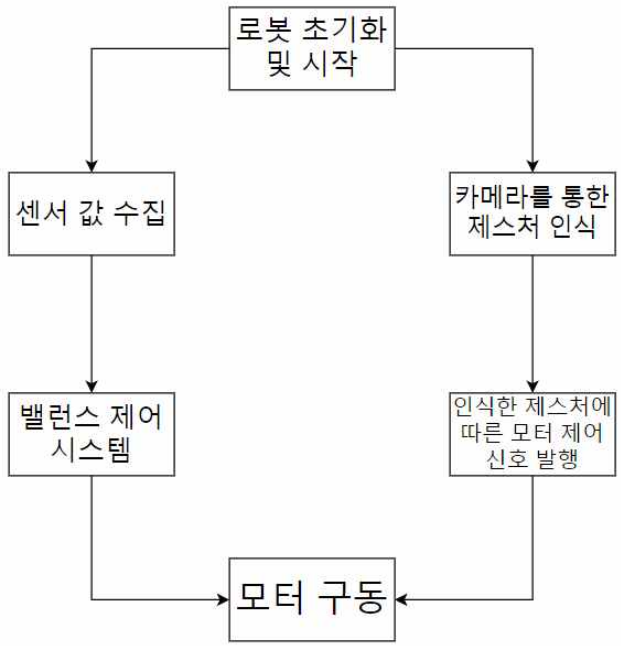


그림 3. 전체 시스템 흐름도

2.2 주요 기능 및 특징

로봇의 주요 기능은 크게 3가지로 분류할 수 있다.

2.2.1 자가 균형 기능

로봇은 움직이는 동안 안정성을 유지할 수 있는 자가 균형 기능을 갖추고 있다. 이 기능은 특히 고르지 않은 지면에서도 로봇이 부드럽게 작동하도록 하며, 자이로센서와 가속도계를 이용한 정밀한 균형 제어 시스템을 통해 구현된다. 이를 통해 로봇은 다양한 지형에서도 안정적인 이동이 가능하다.

2.2.2 제스처를 인식하여 주행

로봇은 손동작을 인식하여 조종하는 기능을 갖추고 있다. 제스처 인식 기술을 통해 사용자는 리모컨 없이 직관적으로 로봇을 조종할 수 있다. 이를 통해 사용자는 더욱 편리하게 로봇을 제어할 수 있으며, 로봇의 이동성과 유연성을 극대화할 수 있다.

2.2.3 폼 체인지 기능

로봇은 지형이나 운행 모드에 맞게 형태를 바꿀 수 있도록 설계된 폼 체인지 기능을 가지고 있다. 이 기능을 통해 로봇은 다양한 환경과 상황에 적응할 수 있으며, 균형이 필요 없는 경우에는 6륜 모드로, 균형 유지가 필요한 경우에는 2륜 모드로 전환하여 주행할 수 있다. 이를 통해 로봇은 안정성과 적응성을 동시에 확보할 수 있다.

또한, 로봇의 전체 시스템은 ROS2(Robot Operating System) 환경에서 패키지를 통해 데이터 교환 및 통신이 이루어진다. 이러한 통합을 통해 다양한 기능을 원활하게 조정하고 동기화할 수 있다.

2.3 개발 환경

표 1은 본 연구에서 로봇을 제작하는 데 이용된 개발 환경을 나열한 것이다.

표 1. 개발 환경

구분	항목	세부내용
Module	NVIDIA Jetson AGX Xavier	로봇 전제 시스템을 제어하는 SBC
jetpack[2]	5.1.1	소프트웨어 개발 플랫폼
Distribution	ubuntu 20.04	로봇 개발 환경
Platform	ROS2 Foxy	로봇의 기능을 통합 제어하기 위한 로봇 시스템 개발 프레임워크
CUDA	11.4.315	NVIDIA GPU 성능을 활용하기 위한 프레임워크
Language	C++, Python	하드웨어 제어 및 다양한 기능을 사용하기 위한 언어
YOLO	YOLOv5	과실 인식을 위한 객체 감지 알고리즘
Roboflow	-	datasets 관리 및 전처리 기능 수행
Pytorch	1.12.0	YOLO 알고리즘을 사용 기반이 되는 딥러닝 프레임워크
ZED SDK	4.0.8	ZED 카메라를 사용하기 위한 소프트웨어 개발 키트
Inventor	-	작품의 3D 모델링을 위함

2.4 상세 기능

2.4.1 제어 및 통신

로봇의 제어 및 통신은 ROS 토픽[3](ROS Topic)을 기반으로 이루어진다. IMU 데이터를 연속적으로 발행하여 로봇의 위치나 움직임을 실시간으로 계산할 수 있도록 한다. 퍼블리셔는 정해진 주소(imu/data)에 데이터를 지속적으로 발행하며, 이 토픽을 구독하는 노드들이 데이터를 수신하여 로봇의 제어에 활용한다.

ROS 토픽은 비동기적으로 데이터를 주고받을 수 있는 기능을 제공하며, 하나의 퍼블리셔와 다수의 구독자가 동시에 데이터를 주고받을 수 있다. 이를 통해 로봇 시스템 내 다양한 모듈들이 효율적으로 통신하고, 실시간으로 데이터를 처리할 수 있다. 예를 들어, 로봇의 균형 유지 시스템은 IMU 데이터를 실시간으로 수신하여 즉각적인 균형 조정이 가능하며, 주행 모듈은 제스처 인식 모듈에서 전달된 명령을 수신하여 로봇의 이동 경로를 조정한다.

또한, ROS 토픽을 이용한 통신 구조는 확장성과 유연성이 뛰어나다. 새로운 센서나 모듈을 추가할 때, 기존의 퍼블리셔와 구독자 구조를 그대로 유지하면서 필요한 데이터를 손쉽게 주고받을 수 있다. 이는 시스템 업그레이드와 유지 보수를 용이하게 하며, 로봇의 기능을 확장할 수 있는 기반을 제공한다.

이와 같은 ROS 기반의 통신 및 제어 시스템은 로봇이 복잡한 환경에서도 안정적으로 동작할 수 있도록 지원하며, 다양한 센서 데이터와 제어 명령을 효과적으로 교환할 수 있게 한다.

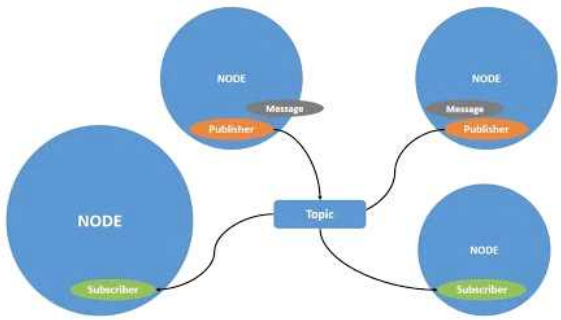


그림 4. ROS Topic의 통신 예시

2.4.2 자가 균형 기능을 위한 PID 제어

로봇의 자가 균형 기능은 PID 제어 기법[4]을 사용하여 구현된다. PID 제어는 로봇의 목표 각도와 센서로 측정된 현재 각도의 차이를 이용하여 균형을 유지한다. 이 차이(오차)를 바탕으로 비례(P), 적분(I), 미분(D) 계수를 적용하여 모터 출력을 조정한다. 이러한 조정을 통해 로봇은 외부 변화에 신속하게 대응하며 안정적인 자세를 유지한다.

PID 제어의 구체적 동작은 다음과 같다.

비례 제어(P): 현재 오차에 비례하여 조정하는 방식으로, 오차가 클수록 더 큰 보정 값을 적용하여 로봇의 균형을 맞춘다.

적분 제어(I): 오차의 누적 값을 고려하여 장기적인 오차를 수정하는 방식으로, 지속적인 작은 오차를 보정하여 균형을 더 정밀하게 유지한다.

미분 제어(D): 오차의 변화율을 고려하여 빠른 응답을 유도하는 방식으로, 갑작스러운 외부 변화에 신속하게 대응할 수 있도록 한다.

센서 데이터 처리 및 통합

로봇은 자이로센서에서 수집된 센서 데이터를 실시간으로 처리하여 현재 각도를 측정한다. 이 데이터는 ROS 토픽을 통해 지속적으로 업데이트되며, PID 제어 알고리즘은 이 데이터를 기반으로 실시간으로 모터 출력을 조정한다. 이러한 통합 시스템은 로봇이 복잡한 지형에서도 균형을 유지할 수 있도록 한다.

실시간 균형 유지

PID 제어를 통해 로봇은 외부 충격이나 지형 변화와 같은 외부 요인에 신속하게 반응할 수 있다. 예를 들어, 로봇이 갑작스러운 경사면을 만날 경우, PID 제어는 경사면에 해당하는 오차를 측정하고 즉각적으로 모터 출력을 조정하여 균형을 유지하며, 로봇이 넘어지지 않고 안정적인 주행을 할 수 있게 한다.

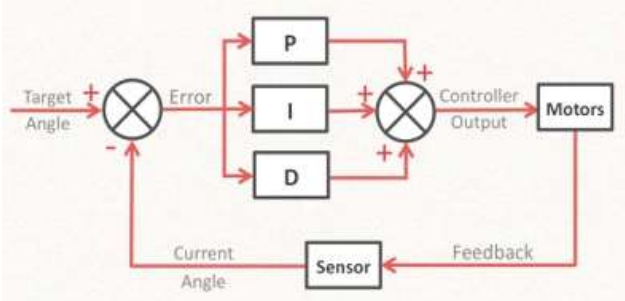


그림 5. PID 제어 흐름도

2.4.3 제스처 및 사람 인식

제스처 데이터셋은 roboflow의 오픈 데이터셋을 다운받아 사용하였다.

학습 데이터는 207장, 검증 데이터는 20장, 테스트 데이터는 10장으로 총 237장의 데이터셋을 사용하였다. 다운받은 데이터셋을 Google Colab 환경에서 GPU를 사용하여 batch:5 epoch: 300으로 학습을 돌려 train data를 만들어 주었다.

그림 6. 학습을 위한 제스처 데이터셋

학습 데이터에 따라 실시간으로 제스처의 클래스를 구분하여 탐지를 한다.

탐지된 제스처를 bounding box의 중심점을 기준으로 계산하여 X(가로), Y(세로)를 구하고 계산된 bounding box의 크기를 이용하여 Z(거리)룰 계산하여 topic으로 발행한다.

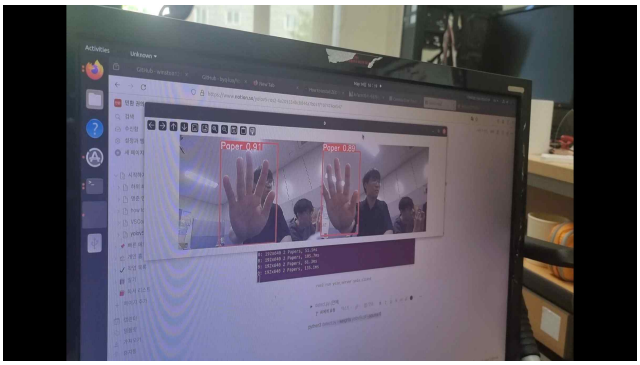


그림 7. 실시간 객체 인식중인 사진

2.4.4 주행

로봇의 주행은 4개의 동력 없는 보조바퀴와 4개의 동력 있는 바퀴 중, wheel모드의 모터가 장착 된 양측 2개의 바퀴로 구동한다. 선 폼에서는 2개의 바퀴로 주행하며, 앉은 폼에서는 4개의 보조바퀴를 포함하여 6개의 바퀴로 주행한다.

주행 모터의 속도 통일

주행모드 모터의 목표 토크의 값 범위는0 ~ 2047이다. 0 ~ 1,023 범위의 값을 사용하면 CCW 방향으로 토크가 가해지며 0으로 설정하면 정지한다. 1,024 ~2,047 범위의 값을 사용하면 CW 방향으로 토크가 가해지며 1,024으로 설정하면 정지한다. 제작 단계에서 양 쪽 모터가 거울처럼 대칭되도록 배치하였으므로, 1번 모터 속도에서 -1024 하고 음수일 때 2048을 더하여 속력을 유지하면서 속도를 일관되도록 한다.

실제 거리 계산

카메라를 통해 들어오는 픽셀의 크기와 좌표 값으로 실제 거리와 위치를 추정한다. 임의의 데이터를 기준으로 실제 크기 및 거리와 픽셀의 크기 및 거리의 비율을 구한다. 실제 크기 / 픽셀 크기 x 초점거리를 각 차원별로 구한 후 거리를 계산한다.

다음은 수평, 수직, 대각선 좌표 길이를 구하는 수식이다.

$$D\_horizontal = f \times \frac{W\_real}{w\_object}$$

$$w\_object\_new = w\_object \times \frac{D\_horizontal}{f}$$

수식 1. 수평 길이를 구하는 수식

$$D\_vertical = f \times \frac{H\_real}{h\_object}$$

$$h\_object\_new = h\_object \times \frac{D\_vertical}{f}$$

수식2. 수직 길이를 구하는 수식

$$D\_real = \sqrt{x\_horizontal^2 + y\_vertical^2}$$

수식3. 좌표 대각선 거리를 구하는 수식

계산된 거리를 바탕으로 좌측 속력과 우측 속력의 비율을 결정한다.

표 2. 센서에 대한 고유 매개변수

[LEFT_CAM_HD]	
fx	700.819
fy	700.819
cx	665.465
cy	371.953
k1	0.174318
k2	0.0261121

fx 및 fy 는 초점 거리(픽셀), cx 와 cy는 픽셀 단위의 광학 중심 좌표, k1 과 k2는 왜곡 매개변수이다. 수평 초점 거리 fx와 수직 초점 거리 fy가 동일하므로 계산식에서는 초점 거리 f로 통일한다.

2.4.5 폼 체인지

로봇의 폼 체인지는 낮은 자세의 6륜 주행에서 높은 자세의 2륜 주행으로 오간다. 자세의 높이를 높임으로서 카메라의 사야가 확장되고 더 다이나믹한 움직임이 가능하며, 낮춤으로서 주행 안정성을 높인다. 이는 유기적으로 연결 된 다리관절과 연결된 모터의 회전각으로 전환된다. 초기상태의 각 정보를 앉은 폼으로 하여 외부 신호에 따라 선 폼으로 전환할 때, 앉은 폼의 각에서 선 폼의 상수만큼 추가한 각 정보만큼 동작한다.

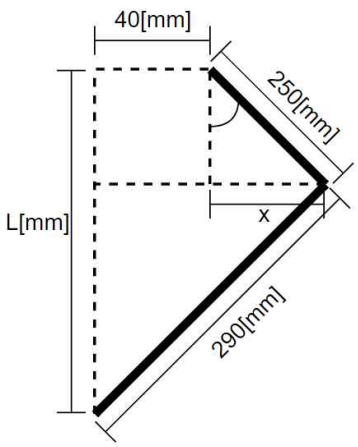


그림 8. 각 계산

creeping mode의 높이는 220mm, standing mode의 목표 높이는 420mm로 로봇의 최상단 높이가 사람의 허리 높이만큼 올라오게 한다. 임의의 x를 사용하여 식을 새우면 $L = \sqrt{250^2 - x^2} + \sqrt{290^2 - (40+x)^2}$ L이 537일 때, x는 90이라는 결론이 도출된다. 이를 삼각비로 계산하면 선 폼의 목표 각은 -31.72°가 된다. 앉은 폼의 각은 -20.556°이므로 폼 체인지 명령을 받을 때, 11.164°씩 움직이게 한다.

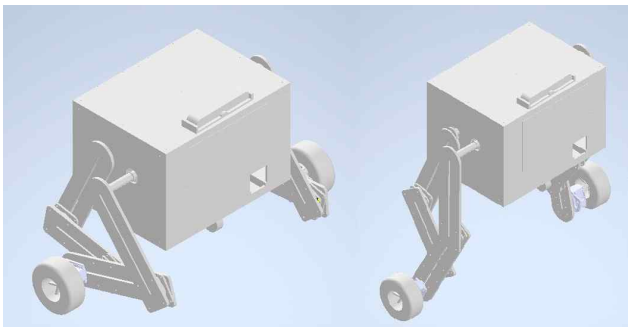


그림 9. 하드웨어 폼 체인지

III. 제작과정

3.1 사용 재료

표 3은 본 연구에서 로봇 제작에 있어 사용된 재료이다.

표 3. 사용 재료

부품	세부 내용	수량
NVIDIA Jetson AGX Xavier	전체 시스템을 제어하는 고성능 SBC	1개
Dynamixel MX-106	패킷 통신을 사용하여 움직임을 제어	4개
ZED Camera[5]	제스처 인식을 위한 스테레오 카메라	각 1개
OPENCR 1.0	로봇의 동작을 제어하는 컨트롤러 보드(MCU)	1개
U2D2	모터 제어를 위한 통신 프로토콜 변환기	1개
배터리	로봇의 동력을 제공하는 14.8V 리튬 폴리머 배터리	2개
IAHRS (RB_SDA-v1)	로봇의 균형 제어를 위한 값을 측정하는 자이로 센서	1개
Display	제스처 인식을 실시간으로 확인할 수 있는 디스플레이	1개
프레임	파이프, 알루미늄 프로파일, 아크릴, 카본 등 작품의 뼈대	-
바퀴	로봇의 이동을 가능하게 하는 바퀴	2개
필라멘트	하드웨어 부품 제작 (3D 프린팅)	-
기타 부속품	케이블, 커넥터, 나사, 허브 , 베어링 등 로봇의 부속품	-

3.2 하드웨어

로봇의 경첩과 프레임과 같은 맞춤형 소형 부품들은 3D프린터를 사용하여 출력하였고, 5mm이하의 평면형 큰 부피의 골격은 레이저 컷팅기를 이용하여 아크릴을 재단하였다. 설계 단계에서부터 아크릴과 3D 출력물을 혼용하여 사용할 것을 염두에 두어, 비용적인 면과 시간적인 면에서 자원을 크게 절약할 수 있었다.

다리의 설계 중, 관절부의 부드러운 회전을 위하여 베어링이 사이에 끼워진 회전판을 나사로 고정하였다. 한쪽 다리에 동력 공급을 위해 고정된 관절을 제외한 나머지 3개의 관절이 마름모 구성으로 서로 지지하고 있어, 고정된 관절의 각도에 따라 일정한 모양을 유지 할 수 있게 한다.

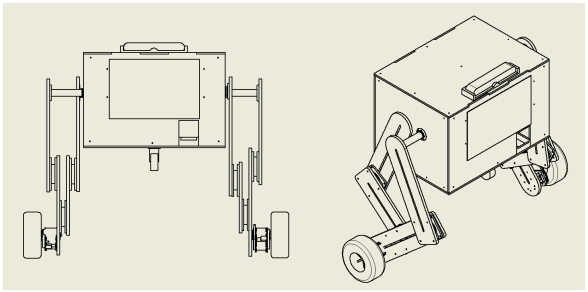


그림 10. 하드웨어 구상도

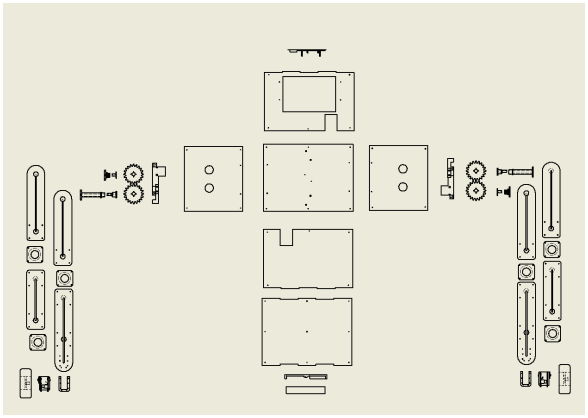


그림 11. 하드웨어 설계도

사용하는 MCU(OpenCR)와 SBC(AGX Xaiver)의 요구되는 입력 전압의 플러그 타입이 달라 각각 따로 배터리를 연결하여 동작한다. 5V의 저전압을 요구하는 센서의 경우 SBC(AGX Xaiver)를 경유하여 전압을 낮춰 공급한다. 12V를 요구하는 Display는 MCU(OpenCR)와 SBC(AGX Xaiver)를 경유하지 않고 배터리에 직접 연결하여 12V 전압을 받아 동작한다.

4개의 모터 전부를 직렬로 전원을 공급하였을 때 전압이 낮아지며 모든 모터에 동일한 전력공급이 되지 않는 것을 방지하기 위하여, Dynamixel에 공급되는 전력은 2개씩 직렬로 연결 된 2쌍의 모터들에 병렬로 연결한다.

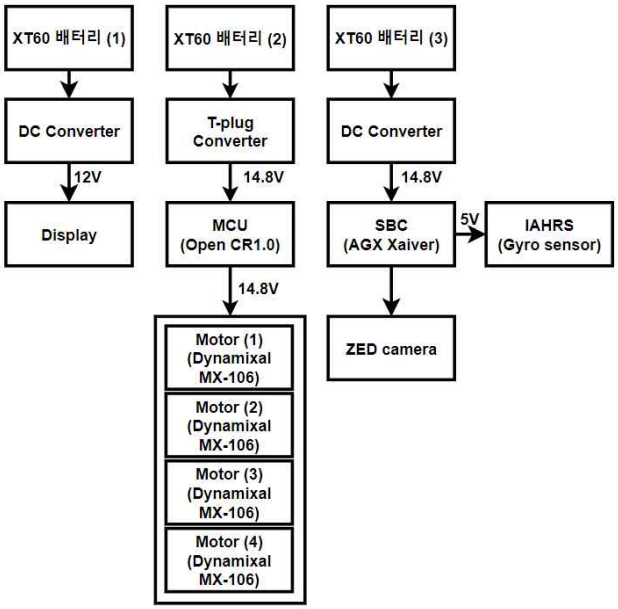
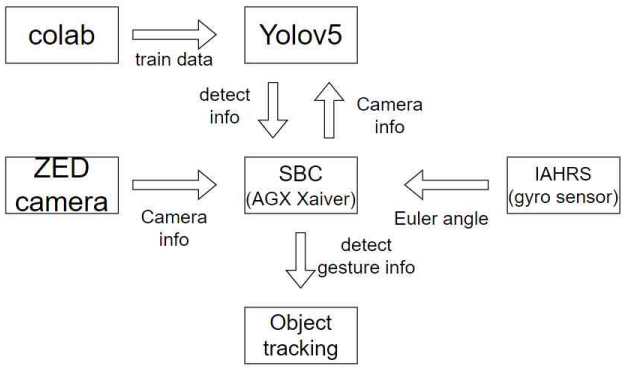


그림 12. 하드웨어 흐름도



3.3 소프트웨어

그림 13. 소프트웨어 구조도

3.3.1 소프트웨어 구조도

그림 13은 본 작품의 소프트웨어 구조도를 나타낸다.
이 구조는 로봇의 다양한 센서와 컴포넌트 간의 데이터 흐름과 상호작용을 보여준다. 주요 구성 요소와 그 기능은 다음과 같다.

Colab은 YOLOv5 모델의 학습 데이터를 제공한다. 학습된 모델은 로봇의 SBC(AGX Xavier)에 업로드되어 실시간 객체 탐지에 사용된다. YOLOv5는 ZED 카메라에서 입력된 영상을 분석하여 객체를 탐지하고, 탐지된 정보를 SBC에 전달한다.

ZED 카메라는 로봇의 시각 정보를 제공하는 역할을 하며, 실시간으로 영상을 SBC에 전달한다. SBC는 시스템의 중앙 제어 장치로, 다양한 센서와 컴포넌트로부터 데이터를 수집하고 처리한다. YOLOv5 모델의 객체 탐지 결과와 IMU 센서의 데이터를 통합하여 로봇의 움직임과 균형을 제어한다. SBC는 ZED 카메라로부터 입력된 영상 정보를 수신하고, IMU 센서로부터는 자세 정보를 수신하며, 제스처 인식 정보를 통해 로봇의 동작을 제어한다.

IAHRS는 자이로센서로, 로봇의 자세 정보를 제공하며,

로봇의 현재 각도를 측정하여 SBC에 전달한다. Object Tracking 모듈은 YOLOv5로부터 받은 객체 탐지 정보를 사용하여 특정 객체를 추적하며, 이 정보는 SBC로 전달되어 로봇의 이동 경로와 동작을 제어하는 데 사용된다.

이와 같은 구조를 통해 자율 균형 로봇은 다양한 센서로부터 실시간 데이터를 수집하고, 이를 바탕으로 객체를 탐지하고 추적하며, 균형을 유지할 수 있다. 이러한 통합된 시스템은 로봇이 복잡한 환경에서도 안정적으로 동작할 수 있도록 한다.

3.3.2 YOLOv5

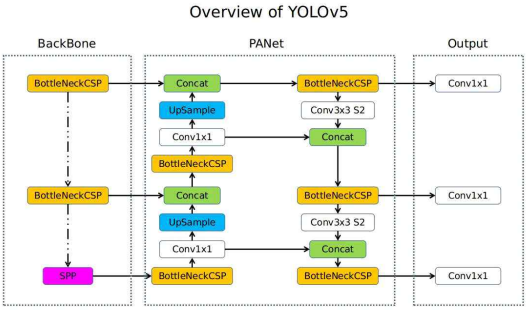


그림 14. YOLOv5 개요

본 작품에서는 제스처 및 사람 인식을 위해 YOLOv5[6]를 사용하였다. YOLOv5는 객체 탐지를 위한 최신 딥러닝 모델 중 하나로, 빠르고 정확한 객체 탐지를 가능하게 하는 구조를 갖추고 있다. 위의 그림 14는 YOLOv5의 전체 구조를 보여준다. 이 모델은 Backbone, PANet, Output 세 가지 주요 부분으로 구성된다.

Backbone은 이미지의 특징을 추출하는 역할을 한다. YOLOv5의 Backbone은 여러 개의 BottleNeckCSP 모듈로 구성되어 있으며, 이는 Convolutional Neural Network (CNN)의 일종으로, 이미지의 특징을 효율적으로 추출하고 전달하는 역할을 한다. 마지막에는 SPP(Spatial Pyramid Pooling) 모듈이 위치하여 다양한 크기의 필터를 사용해 특징 맵을 추출한다.

PANet(Path Aggregation Network)은 Backbone에서 추출된 특징을 통합하고 강화하는 역할을 한다. 이 과정에서 특징 맵의 크기를 조절하며, 보다 정교한 특징을 추출하여 객체 탐지의 성능을 향상시킨다.

Output 부분은 최종적으로 객체 탐지 결과를 출력하는 역할을 한다. PANet을 통해 강화된 특징 맵을 기반으로 객체의 위치와 클래스 정보를 예측한다.

즉 YOLOv5는 Backbone에서 특징을 추출하고, PANet에서 다양한 레이어의 특징을 통합 및 강화하며, Output에서 최종 객체 탐지를 수행한다. 이러한 구조는 빠르고 정확한 객체 탐지를 가능하게 하여, 실시간 애플리케이션에서 매우 유용하게 사용될 수 있다. YOLOv5는 특히 객체 인식 및 추적, 자율 주행 등 다양한 분야에서 활용될 수 있는 강력한 모델이다.

3.4 소스코드

3.4.1 자가 균형을 위한 소스코드

```
class ImuSubscriber : public rclcpp::Node {
public:
    ImuSubscriber()
    : Node("imu_subscriber"),
      balance_controller_(4.0, 0.1, 0.2)  // PID 파라미터를 적절하게 설정
    {
        motor_pub_ = this->create_publisher<std_msgs::msg::Float64>("/motor_command", 10);
        imu_subscription_ = this->create_subscription<sensor_msgs::msg::Imu>(
            "imu/data", rclcpp::SensorDataQoS(), std::bind(&ImuSubscriber::callback, this, std::placeholders::_1));

        portHandler_ = dynamixel::PortHandler::getPortHandler(DEVICENAME.c_str());
        packetHandler_ = dynamixel::PacketHandler::getPacketHandler(PROTOCOL_VERSION);
```

```
class PIDController {
public:
    PIDController(double kp, double ki, double kd)
    : kp_(kp), ki_(ki), kd_(kd), prev_error_(0), integral_(0) {}

    double compute(double error) {
        double p = kp_ * error;
        integral_ += error;
        double i = ki_ * integral_;
        double d = kd_ * (error - prev_error_);
        prev_error_ = error;
        return p + i + d;
    }

private:
    double kp_, ki_, kd_;
    double prev_error_;
    double integral_;
};
```

```
class BalanceController {
public:
    BalanceController(double kp, double ki, double kd)
    : pid_controller_(kp, ki, kd) {}

    double compute(double error) {
        return pid_controller_.compute(error);
    }

private:
    PIDController pid_controller_;
};
```

그림 15. 밸런싱 제어를 위한 PID 제어 소스코드

그림 15는 로봇의 균형을 제어하기 위한 PID 제어 알고리즘을 구현한 코드이다. 이 코드는 ROS 노드인 ImuSubscriber 클래스를 통해 IMU 데이터를 구독하고, PID 제어를 통해 로봇의 균형을 유지하는 기능을 수행한다. PIDController 클래스는 비례(P), 적분(I), 미분(D) 계수를 사용하여 제어 신호를 계산하며, Balance Controller 클래스는 PIDController 클래스를 이용해 균형 제어를 수행한다. IMU 데이터를 수신한 후, callback 함수에서 현재 각도를 계산하고, 목표 각도와의 차이(오차)를 구한 뒤, PID 제어 신호를 생성하여 모터에 적용함으로써 로봇의 균형을 유지한다. 이를 통해 로봇은 외부 변화에 신속하게 대응하며, 안정적인 자세를 유지할 수 있다.

```
while(rclcpp::ok())
    {
    rclcpp::spin_some(node);
    if (serial_fd >= 0)
    {
        const int max_data = 10;
        double data[max_data];
        int no_data = 0;
        no_data = iahrs.SendRecv("gtn", data, max_data);    // Read angular_velocity _ wx, wy, wz
        if (no_data >= 3)
        {
            // angular_velocity
            iahrs.imu_data_msg.angular_velocity.x = _pIMU_data.dAngular_velocity_x = data[0]*(M_PI/180.0);
            iahrs.imu_data_msg.angular_velocity.y = _pIMU_data.dAngular_velocity_y = data[1]*(M_PI/180.0);
            iahrs.imu_data_msg.angular_velocity.z = _pIMU_data.dAngular_velocity_z = data[2]*(M_PI/180.0);
        }
        no_data = iahrs.SendRecv("atn", data, max_data);    // Read linear_acceleration     unit: m/s^2
        if (no_data >= 3)
        {
            //// linear_acceleration   g to m/s^2
            iahrs.imu_data_msg.linear_acceleration.x = _pIMU_data.dLinear_acceleration_x = data[0] * 9.80665;
            iahrs.imu_data_msg.linear_acceleration.y = _pIMU_data.dLinear_acceleration_y = data[1] * 9.80665;
            iahrs.imu_data_msg.linear_acceleration.z = _pIMU_data.dLinear_acceleration_z = data[2] * 9.80665;
        }
        no_data = iahrs.SendRecv("etn", data, max_data);    // Read Euler angle
        if (no_data >= 3)
        {
            // Euler _ rad
            _pIMU_data.dEuler_angle_Roll  = data[0]*(M_PI/180.0);
            _pIMU_data.dEuler_angle_Pitch = data[1]*(M_PI/180.0);
            _pIMU_data.dEuler_angle_Yaw   = data[2]*(M_PI/180.0);
        }

        tf2::Quaternion q;
        q.setRPY(_pIMU_data.dEuler_angle_Roll , _pIMU_data.dEuler_angle_Pitch, _pIMU_data.dEuler_angle_Yaw);
        // orientation
        iahrs.imu_data_msg.orientation.x = q.x();
        iahrs.imu_data_msg.orientation.y = q.y();
        iahrs.imu_data_msg.orientation.z = q.z();
        iahrs.imu_data_msg.orientation.w = q.w();
        iahrs.imu_data_msg.header.stamp = node->now();
        iahrs.imu_data_msg.header.frame_id = "imu_link";

        // publish the IMU data
        iahrs.imu_data_pub->publish(iahrs.imu_data_msg);
```

그림 16. 밸런싱 제어를 위한 센서 값 퍼블리시 소스코드

그림 16은 로봇의 IMU 데이터를 처리하고 퍼블리싱하기 위한 센서 드라이버 패키지이다. 이 파일은 ROS 2 환경에서 IMU 데이터를 수집하고, 이를 필요한 형태로 변환하여 배포하는 기능을 구현한다. IAHRS 클래스는 IMU 데이터를 배포하고, 서비스 제공을 통해 데이터를 초기화하는 기능을 수행한다. IMU 데이터는 시리얼 통신을 통해 수집되며, 필요한 변환을 거쳐 실시간으로 배포된다. 메인 함수에서는 ROS 2 노드를 초기화하고 실행하여 데이터를 수집하고 배포하는 루프를 실행한다. 이를 통해 로봇은 실시간으로 IMU 데이터를 처리하고 배포하여, 안정적인 동작을 지원할 수 있다.

3.4.2 실시간 객체 탐지를 위한 소스코드

```
class LoadStreams(Node):
    def __init__(self, sources='file.streams', img_size=640, stride=32, auto=True, transforms=None, vid_stride=1):
        super().__init__('yolo_streams')  # ROS 노드를 초기화합니다.
        torch.backends.cudnn.benchmark = True  # 고정 크기 추론에 더 빠름
        self.mode = 'stream'
        self.img_size = img_size
        self.stride = stride
        self.vid_stride = vid_stride  # 비디오 프레임 속도 간격
        sources = Path(sources).read_text().rsplit() if os.path.isfile(sources) else [sources]
        # source가 파일이면 해당 파일에서 source를 읽고 그렇지 않으면 직접 사용
        n = len(sources)
        self.sources = [clean_str(x) for x in sources]  # 나중을 위해 소스 이름 정리
        self.imgs, self.fps, self.frames, self.threads = [None] * n, [0] * n, [0] * n, [None] * n
        self.bridge = CvBridge()  # CvBridge 인스턴스 추가
        self.w, self.h = [0] * n, [0] * n
        self.threads = []
        for i, s in enumerate(sources):  # 인덱스, 소스
            self.threads.append(threading.Thread(target=self.start_ros_subscriber))
            self.threads[i].start()
            # rospy.Subscriber("/camera/color/image_raw", Image, self.update)
            # print("self.imgs", self.imgs)
            # 비디오 스트림에서 프레임 읽기 시작
```

```
    st = f'{i + 1}/{n}: {s}... '
    if urlparse(s).hostname in ('www.youtube.com', 'youtube.com', 'youtu.be'):  # 소스가 YouTube 비디오인 경우
        # YouTube 형식 예: 'https://www.youtube.com/watch?v=Zgi9g1ksQHc' 또는 'https://youtu.be/Zgi9g1ksQHc'
        check_requirements(('pafy', 'youtube_dl==2020.12.2'))
        import pafy
        s = pafy.new(s).getbest(preftype='mp4').url  # YouTube URL
    s = eval(s) if s.isnumeric() else s  # 예: s = '0' 지역 웹캠 s = 0은 로컬 웹캠을 나타냄. 그렇지 않으면 그대로 사용
    if s == 0:
        assert not is_colab(), '--source 0 웹캠은 Colab에서 지원되지 않습니다. 로컬 환경에서 명령을 다시 실행하십시오.'
        assert not is_kaggle(), '--source 0 웹캠은 Kaggle에서 지원되지 않습니다. 로컬 환경에서 명령을 다시 실행하십시오.'

    w = 640
    h = 480
    fps = 30
    self.frames[i] = float('inf')
    self.fps[i] = fps  # 이 소스의 프레임 수와 fps를 설정합니다.
    # _, self.imgs[i] = cap.read()  # 첫 번째 프레임 보장
    # self.threads[i] = Thread(target=self.update, args=([i]), daemon=True)
    #  ## 이 소스에서 프레임을 지속적으로 업데이트하기 위해 새 스레드 시작
    LOGGER.info(f'{st} 성공 ({self.frames[i]} 프레임 {w}x{h} at {self.fps[i]:.2f} FPS)')
    # self.threads[i].start() ## 스레드 시작
    while any(v is None for v in self.imgs):
        time.sleep(0.1)  # 이미지가 로드 될 때까지 대기
while not all(t.is_alive() for t in self.threads):  # 모든 스레드가 시작될 때까지 대기
    time.sleep(0.1)
LOGGER.info('')  # 새 줄
```

```
def start_ros_subscriber(self):
    self.create_subscription(
        Image,
        'zed/zed_node/stereo_raw/image_raw_color',
        # '/camera/color/image_raw',
        self.update,
        QoSProfile(depth=10, reliability=ReliabilityPolicy.RELIABLE))
    # rospy.Subscriber("/camera/aligned_depth_to_color/image_raw", Image, self.update)
    rclpy.spin(self)

def update(self, data):
    # print("ddddd")
    try:
        for i in range(len(self.sources)):
            self.imgs[i] = self.bridge.imgmsg_to_cv2(data, "bgr8")
            # height, width, _ = self.imgs[i].shape  # 이미지의 차원을 얻습니다
            # print("Image resolution: {}x{}".format(width, height))
            # print("self.imgs", self.imgs)
    except CvBridgeError as e:
        LOGGER.warning('경고 ⚠ 이미지 변환 중 CvBridgeError 발생: ', e)
```

그림 17. 카메라 데이터를 불러오는 코드

그림 17은 기존 dataloder를 ROS메시지를 사용하여 데이터를 처리할 수 있도록 start_ros_subscriber(self), update(self, data)를 정의하여 수정하였다.

```
FILE = Path(__file__).resolve()
ROOT = FILE.parents[0]  # YOLOv5 root directory
if str(ROOT) not in sys.path:
    sys.path.append(str(ROOT))  # add ROOT to PATH
ROOT = Path(os.path.relpath(ROOT, Path.cwd()))  # relative

from models.common import DetectMultiBackend
from utils.dataloaders import IMG_FORMATS, VID_FORMATS, LoadImages, LoadScreenshots, LoadStreams
from utils.general import (LOGGER, Profile, check_file, check_img_size, check_imshow, check_requirements, colorstr, cv2,
                           increment_path, non_max_suppression, print_args, scale_boxes, strip_optimizer, xyxy2xywh)
from utils.plots import Annotator, colors, save_one_box
from utils.torch_utils import select_device, smart_inference_mode


class Yolo(Node):
    def __init__(self):
        super().__init__('yolo_ros')
        self.object_detect_pub = self.create_publisher(Yolodetect, 'yolo_info' , 10)
# object_detect_pub = rospy.Publisher('object_detect', Yolodetect, queue_size = 10)
```

```
for obj in detected_objects:
    object_detect = Yolodetect()
    object_detect.center.x = obj['center_x']
    object_detect.center.y = obj['center_y']
    object_detect.length_1.x = obj['length_x_1']
    object_detect.length_1.y = obj['length_y_1']
    object_detect.length_2.x = obj['length_x_2']
    object_detect.length_2.y = obj['length_y_2']
    object_detect.name = obj['name']
    object_detect.num = obj['num']
    self.object_detect_pub.publish(object_detect)
    xyxy, cls, conf = obj['xyxy'], obj['cls'], obj['conf'] # 추가

    if save_txt:  # Write to file
        xywh = (xyxy2xywh(torch.tensor(xyxy).view(1, 4)) / gn).view(-1).tolist()  # normalized xywh
        line = (cls, *xywh, conf) if save_conf else (cls, *xywh)  # label format
        with open(f'{txt_path}.txt', 'a') as f:
            f.write(('%g ' * len(line)).rstrip() % line + '\n')

    if save_img or save_crop or view_img:  # Add bbox to image
        c = int(cls)  # integer class
        label = None if hide_labels else (names[c] if hide_conf else f'{names[c]} {conf:.2f}')
        annotator.box_label(xyxy, label, color=colors(c, True))
    if save_crop:
        save_one_box(xyxy, imc, file=save_dir / 'crops' / names[c] / f'{p.stem}.jpg', BGR=True)
```

```
# Stream results
im0 = annotator.result()
if view_img: ##위에서 디스플레이 검사를 한다면 이 코드가 실행됨. view_img = True
    if platform.system() == 'Linux' and p not in windows:
        windows.append(p)
        cv2.namedWindow(str(p), cv2.WINDOW_NORMAL | cv2.WINDOW_KEEPRATIO)  # allow window resize (Linux)
        cv2.resizeWindow(str(p), im0.shape[1], im0.shape[0])
    cv2.imshow(str(p), im0)
    cv2.waitKey(1)  # 1 millisecond

    # cv2.namedWindow('Depth', cv2.WINDOW_NORMAL)
    # cv2.imshow('Depth', depth_frame)
    # cv2.waitKey(1)
```

그림 18. 실시간 객체 인식을 하는 소스코드

그림 18(detect.py)은 ROS통신을 하며 바운딩 박스의 객체의 중심점을 통해 계산하고 바운딩 박스의 크기를 통해 객체와의 거리를 계산할 수 있도록 수정하였다.

그 결과 X(가로), Y(세로), Z(거리)를 topic으로 발행한다.

Ⅳ. 최종 결과물

4.1 외관 및 동작

4.1.1 로봇 외관

그림 14는 로봇의 외관 사진이다. 몸체, 다리, 바퀴가 베어링, 3d 프린팅 부품 등으로 연결되어 있는 모습이다. 중앙부는 시스템 동작에 필요한 여러 모듈과 부품들이 아크릴 뼈대의 박스 안에 들어있다. 본체의 정면에는 실시간 인식에 필요한 zed 카메라와 인식을 확인하기 위한 display가 설치되어 있다.

그림 19 로봇의 외관

4.1.2 동작 (영상 링크)

● 로봇이 앞뒤로 주행하며 방향 전환하는 영상 : https://www.youtube.com/watch?v=RfzLx-0OVYY

● 실시간 객체 인식 - 실시간으로 제스처를 인식하는 것을 확인 하는 영상 : https://www.youtube.com/watch?v=9AaTPkjsMks

● 프로토타입 밸런스 제어 - 프로토타입이 PID 제어를 통해 밸런스를 제어하는 영상 : https://youtu.be/sEzjfjThum0

4.2 성능 결과

Yolo에서는 모델의 성능(정확도)을 mAP(Mean average precision)을 통해 확인한다. mAP와 정확도는 비례한다. AP(Average precision)계산에서는 precision-recall 곡선을 사용한다.

IoU(Intersection over union)는 객체 검출 모델의 정확도를 측정하는 평가지표로 합성곱 신경망을 사용한 객체 검출 모델(R-CNN, Faster R-CNN, YOLO, etc.)등을 평가할 때 사용한다.(IoU = 교집합 영역 넓이/합집합 영역 넓이)

Precision-recall(PR곡선): confidence threshold 값에 따른 precision과 recall의 값의 변화를 그래프로 표현한 것이다.

Precision(정밀도): 모델이 예측한 결과의 Positive 결과가 얼마나 정확한지 나타내는 값. 검출 결과들 중 옳게 검출한 비율을 의미한다.

Recall(재현율): 모델의 예측한 결과가 얼마나 Positive 값들을 잘 찾는지 측정하는 것. 실제 옳게 검출된 결과물 중에서 옳다고 예측한 것의 비율을 의미한다.

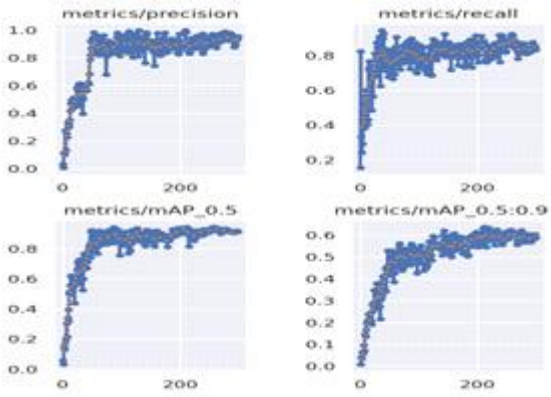


그림 20. 학습 후 나온 mAP값

mAP의 값을 비교해보면 그래프가 많이 흔들리는 것을 볼 수 있다. mAP값이 정확도에 비례하기 때문에 detection의 정확도가 떨어져 class를 detection하는 것에 정확도와 인식률이 약간 떨어지는 것을 확인할 수 있었다. 따라서 data의 수를 늘리거나 학습 수, 학습률을 높여 mAP의 값을 높여 인식률 및 정확도를 높일 계획이다.

Ⅴ. 결 론

본 연구의 자율 균형 로봇은 자동으로 균형을 잡는 기능을 통해 물류, 건설, 산업 분야 등에서 안정적이고 효율적으로 활용될 수 있다. 이 로봇은 다양한 지형에서도 안정적으로 움직이며 넘어짐을 방지하는 기능을 제공한다. 자동 균형 조정을 통해 사용자가 다른 작업에 집중할 수 있어 작업 효율이 증가하고, 건설, 물류 등 여러 산업 분야에서 폭넓게 활용될 수 있다. 특히, 제스처 인식 기능을 통해 사용자는 직관적인 조작이 가능하며, 폼 체인지 기능을 통해 로봇이 다양한 환경에 적응할 수 있다.

안정적인 운동으로 인해 다양한 지형에서도 넘어짐을 방지할 수 있으며, 자동 균형 조정을 통해 사용자는 다른 작업에 집중할 수 있어 작업 효율이 크게 증가한다. 이러한 기능들은 로봇이 건설, 물류 등 여러 산업 분야에서 폭넓게 활용될 수 있는 기반을 마련한다. 특히, 로봇의 자율성과 유연성은 작업 환경의 변화에 빠르게 대응할 수 있어, 다양한 응용 분야에서 활용 가능성을 크게 높인다.

프로젝트 초기에는 재료 가공 어려움과 계획된 일정보다 느린 진행으로 일부 기능을 수행하는 데 어려움이 있었다. 프로젝트의 기능을 회의할 당시, 과한 의욕으로 프로젝트에 많은 기능을 추가하길 원했고 그 과정이 오히려 독이 되어 기능 개발에 차질이 많이 생겼다. 이러한 문제로 인해 기존 목표 중 일부 기능을 수행할 수 없었다.

이 과정을 통해 이론을 배우는 것과 그 이론을 실제 하드웨어에 적용하여 구현하는 것 사이에 큰 차이가 있다는 것을 깨달았다. 또한, 이론으로 배운 것들을 코드를 사용하여 적용하는 데 또 다른 능력이 요구되며, 단순히 이론만 배워서는 안 된다는 것을 느꼈다.

본 프로젝트를 통해 이와 비슷한 기능의 로봇을 보았을 때, 더 많은 기술을 보고 이해할 수 있는 능력이 향상되었다. 이 연구를 통해 이론적 지식을 실제로 구현하는 과정을 경험하면서, 로봇 공학에 대한 깊은 이해와 실용적인 기술을 습득할 수 있었다. 이는 앞으로의 연구와 개발에 큰 도움이 될 것이며, 다양한 산업 분야에서 자율 균형 로봇의 응용 가능성을 더욱 넓힐 수 있는 기반이 될 것이다.

References

[1] https://robor.co.kr/?page_id=827 - IAHRS 센서 매뉴얼

[2] jetpack SDK 참고 - https://developer.nvidia.com/embedded/jetpack-sdk-511

[3] ros2 topic 발행 참조 - https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html

[4] PID 제어 참고 https://velog.io/@717lumos/Control-PID-%EC%A0%9C%EC%96%B4

[5] zed ros2 참고 - https://github.com/stereolabs/zed-ros2-wrapper

[6] yolov5 참고 - https://github.com/ultralytics/yolov5