**카메라가 왜 필요하냐?**

다음 경우를 생각하셈.

1. 눈을 감고 똥@꼬 긁기 🡪 쉬움

* 눈 감아도 너 팔이 어디 있는지 쉽게 알 수 있음.
* 이를 자기수용성 감각(proprioceptive state)라고 함. 로봇의 자기수용성 감각은 조인트 각도임. 예시로, 너는 항상 팔의 조인트 각도를 알고 있기 때문에, 너 팔이 니 몸에 닿고있는지 판단 (로봇의 경우 자기 충돌 감지; self-collision detection)가능함.

1. 완전히 어두운 상황에서 불 스위치 키러 가기 --> 어려움.

즉, 2번과 같은 외부 환경과의 상호작용을 위해서 로봇은 카메라가 필요함.

그럼 카메라가 뭐냐? 카메라는, 3차원 실세계 공간의 점들에서 나온 빛(ray)들을 스크린 공간에 옮겨 적는(mapping) 기계임.

* 3차원 실세계 공간의 한 점
* 2차원 스크린 공간의 한 점(픽셀)
* A지점에서 B지점으로 옮겨 적는 matrix

수학적으로 이해 못해도 카메라는 입력 로부터 변환 관계 를 적용해서 출력 내보내는 것만 알면 댐.

더 정확하게는, 보통 이런 수식으로 계산함

물리적 의미는 임의의 기준좌표 에서 정의된 한 점 가 있을 때, 이를 3D 카메라 기준에서는 어디 있는지 파악하고, 이를 다시 스크린 기준으로 옮기는거임. 어려운 말로는 이렇게 지정함.

* : 내부 변환 매트릭스 (Intrinsic matrix)
  + 카메라 좌표계와 스크린 공간의 관계 정의
  + 카메라 소프트웨어나 하드웨어에 의해 좌우됨
* : 외부 변환 매트릭스 (Extrinsic matrix)
  + 글로벌 좌표계와 카메라 좌표간 관계 정의
  + 카메라 위치에 의해 좌우됨

이 개@소리가 왜나오냐? 일반적 카메라는 2D 픽셀값만을 줌. 그러나 로봇은 3D공간에서 움직임. 사람 눈깔 하나는 납작한 2D정보만을 받기에 3차원 공간 정보를 얻기 위해 눈을 2개 쓰는거랑 똑같은거임. 따라서 너가 카메라로 어떤 작업을 하기 위해서는 너는 카메라 정보부터 3D 정보를 어떻게든지 뽑아내야 함.

물론 너가 직접 할필요는 없음. 관련 코드가 아주 잘 나오거든. 너가 할일은 과 을 구해주기만 하면 됨. 어케구하냐? 이제부터 설명함.

**내부 변환 변수 구하기**

내부 변수는, 위에서 기술한대로, 카메라 자체에 설정된 값임.

이를 알아내기 위해서는 몇 개 준비물이 필요함: 종이, 프린터, 평평한 판

1. 체커 보드 프린트하기

* 여기(<https://markhedleyjones.com/projects/calibration-checkerboard-collection>)에서 다운받을 수 있음. 혹은 내가 저장해둔 파일 보셈(A4, 8X6, 30mm)
* 나는 A4, 8X6, 30mm로 설정함. 해당 스펙을 명시해주기만 하면 알아서 구해줌.
* 주의사항 1: 프린트시 여백 0, 원본 크기로 뽑아야함.
* 주의사항 2: 프린트 후 자로 직접 재서 너가 원한대로 나왔는지 봐야 함. 다시 뽑기 귀찮으면 코드를 너가 측정한 길이 기준으로 돌려도 됨.

1. 납작한 판에 붙이기

* 체커보드는 무적권 납작해야 함. 가볍고 평평한 판에 붙일 것(들고 움직여야 됨/카메라를 움직이거나)

1. 실행 환경 구성

* vscode사용을 권장함.
* 터미널에 입력
  + conda create -n cobot\_py311 python=3.11 (이미 있으면 안해도 됨)
  + conda activate cobot\_py311
  + pip install opencv-python numpy (이미 했으면 안해도 됨)
* VSCode에서 F1 -> Python:Select Interpreter -> cobot\_py311

1. 추정 코드 실행

* Camera\_intrinsic\_extimation/main.py를 실행한 후, 설정값 확인.
* 필수 설정값: PATTERN\_SIZE, SQUARE\_SIZE
* 보드가 잡힌 상태로 space를 누르면 샘플 모아짐.
* 때려 맞추는 기술이기 때문에, 샘플 수가 많을수록 좋음(일반적으로 30개~50개)
* 여러 방향에서 체커보드를 보셈. -> 정확도 향상을 위해 중요함
  + 카메라 중앙 & 카메라 외각
  + 가까이서 & 멀리에서
  + 보드를 정면에서, 위로 기울여서, 옆으로 기울여서, 회전해서

|  |  |
| --- | --- |
|  |  |
| 체커보드 (3cm) | 실행화면(무지개 선 보임?) |

* 샘플을 충분히 모았다고 판단되면, esc를 눌러서 종료.

1. 결과 확인

* Esc를 누른 후 기다리면, 파일이 저장됨. (파일명: calibration\_result\_TIMESTAMP.json)
* 이를 열어보면, cx,cy,fx,fy,distortion등 내부 변환 매트릭스 관련 정보가 들어있음.
* 즉, 너는 이제 이 값을 “땡겨오기”만 하면 된다는거지. 내가 다 해놨음 ㅋ

1. 재측정

* 동일 카메라에 대해서 한번 측정하면, 영원히 쓸 수 있음.
* 그러나, 카메라 zoom을 했다거나, 너가 무슨 내부 변수를 바꿨다면 재측정해야함.

**외부 변환 매트릭스 구하기**

카메라로부터 나온 값은, 말 그대로 카메라 기준임.

* 카메라: “내 기준 5cm앞에 있어!!! 와서 잡어!!!!”
* 너, 로봇, & 우리가 설정한 어떤 기준 좌표계: “ㅅㅂ 니 위치는 조상님이 알려주노”

따라서 핵심은, 너 맘에 드는 지점을 기준 좌표계로 잡고, 기준 좌표계와 로봇, 그리고 기준 좌표계와 카메라의 관계를 설정하는 것임.

여기서 기준 좌표계는 1개의 AprilTag를 사용하겠음.

준비물은 다음과 같음: 종이, 프린터, 칼, 자

1. AprilTag 출력

* 여기(<https://chaitanyantr.github.io/apriltag.html>)에 들어가셈.
* 기술적 디테일 몰라도댐. 궁금하면 gpt한테 물어보셈. 다음과 같이 설정
  + Tag Family: tag36h11
  + Tag ID: 0
  + Total Size: 별로 안중요함.
  + Tag Size: 너 맘에드는걸로. 실험적으로, 40mm이상 하면 충분함.
* 주의사항 1: 프린트시 여백 0, 원본 크기로 뽑아야함.
* 주의사항 2: 프린트 후 자로 직접 재서 너가 원한대로 나왔는지 봐야 함. 다시 뽑기 귀찮으면 코드를 너가 측정한 길이 기준으로 돌려도 됨.

1. 태그 자르기

* 태그를 정사각형으로 자르셈.
* 여백을 Tag Size의 10%정도 주면 됨. 예시로, 너가 40mm 짜리 뽑았다면 계산하기 쉽게 5mm 혹은 10mm주셈. 한 변의 길이가 50mm(여백 5mm 경우) 혹은 60mm(여백 10mm 경우) 되게

1. 실행 환경 구성

* vscode사용을 권장함.
* 터미널에 입력
  + conda create -n cobot\_py311 python=3.11 (이미 있으면 안해도 됨)
  + conda activate cobot\_py311
  + pip install opencv-python numpy pupil\_apriltags (이미 했으면 안해도 됨)
* VSCode에서 F1 -> Python:Select Interpreter -> cobot\_py311

1. 실행

* Localization\_april\_tag/0\_visualize\_any\_tag\_realtime.py를 실행한 후, 설정값 확인.
* 코드 내 필수 설정/확인값: CALIB\_JSON\_PATH, TAG\_FAMILLY, TAG\_SIZE\_M
* 마찬가지로 태그 휘어지면 인식 안되거나 저하됨! 평평한 곳에 둘 것.

|  |  |
| --- | --- |
|  |  |
| 카메라 기준 프레임. | 여기 뜨는 값은 좌측 프레임 기준임.  카메라에서 멀어지면, z값이 커지는거지.  카메라 밑으로 움직이면, y값이 커지는거고. |

1. 결과 확인

* 이게 의미하는 바가 뭐냐? 카메라 기준 어떤 점과의 3D관계를 알 수 있다는 거임.
  + 예시로, 우리는 ID:0을 기준 좌표계로 쓰기로 했음.
  + 해당 파일이 보여주는 건 카메라가 (0,0,0)일 때 위치 정보임. (회전 정보도 알수있음!)
  + 이를 뒤집어 ID:0을 기준(0,0,0)으로 설정하면 기준 좌표계 대비 카메라 위치 가능.
    - 내 1m앞에 너가 있으면 너기준으로는 내가 1m뒤에 있는거 맞잖아 ㅋ
    - 물론 방향도 중요함. 서로 마주볼수도 있는거니까 – 이것도 코드가 다 처리가능
* 다시 돌아가서,
  + 카메라: “내 기준 5cm앞에 있어!!! 와서 잡어!!!!”
  + 너 & 우리가 설정한 기준 좌표계(ID:0): ㅇㅋ 확인
  + 로봇: “ㅅㅂ 어딨냐고”

기본 아이디어: ID:0의 기준 좌표계를 로봇 기준 좌표계와 일치시키자.

**로봇 to cam 변환 매트릭스 구하기**

몇가지 알아낸 사실:

* 카메라에서 읽기만 하면 카메라 위치 대비 마커 위치 알아낼 수 있음.
* 반대로, 마커 위치를 기준(0,0,0)으로 하면 카메라 위치 알아낼 수 있음. 어케하는진 모르겠지만 코드가 이것까지 고려해서 알아서 계산해줄거임.
* ID별로 각각 동적인 위치 추적이 가능함.

그리고 우리의 가정:

* 카메라 세팅 딱, 마커 세팅 딱, 로봇 베이스 세팅 딱 하면 이 셋은 움직일 일 없음.
  + 굳이 컴퓨터 돌려가면서 동적으로 변수 업데이트 해줄 필요없잖아.
  + 물론 니가 세팅 후에 위치를 옮긴다면 다시 구해줘야겠지.
  + 아니면 자동차 로봇같은경우는 매 시간마다 업데이트 해줘야 하고.
  + 근데 우리는 로봇팔인데 알빠노?

제안하는 아이디어:

로봇 베이스 모서리에 마커 4개 붙이고, 평균낸 위치가 우리 기준 좌표계임.

그리고, 이는 당연히 로봇 좌표계와도 일치함. (로봇은 항상 베이스를 (0,0,0)으로 보거든)

1. 여러 April tag 출력

* Tag ID: 1,2,3,4
* Tag Family: tag36h11
* Tag Size: 편의를 위해 기존과 동일하게.

1. April tag 로봇 베이스 모서리에 부착

* 태그 여백 길이가 모두 같고, 로봇 베이스에 균일하게 붙여야댐.
* 이때 순서가 중요함. 시계방향으로 각각 1,2,3,4 붙여야 함.
* 코드 내부적으로는
  + 동시에 ID:1과 ID:3을 보면 이 둘을 평균내서 센터를 구함
  + 동시에 ID:2과 ID:4을 보면 이 둘을 평균내서 센터를 구함
  + 그 외에는 계산 안됨. 왜 안되는지 생각해보셈.
* 부착시 각 태그의 회전도 중요함.
  + “좌표계”라는건 위치 뿐만 아니라 회전도 정의하기 때문.
  + 정사각형이라고 막붙인다? 위치는 맞는데 회전이 안맞음.
  + 이해안되면 Localization\_april\_tag/0\_visualize\_any\_tag\_realtime.py 켜고 태그를 한바퀴 돌려보셈.

🡪 이렇게 해서 만들어놓은게 base.pdf로 저장되어있음. 그냥 뽑아서 로봇밑에 깔고 쓰셈

* 근데 잘 보면, 카메라에서 잡히는 Tag와 로봇 베이스 간 방향이 다를거임.
* 로봇은 항상 앞쪽이 +x, 왼쪽이 +y, 위쪽이 +z임. 이를 맞춰주는 코드 들어있음.
* 수학적으로는, 로 계산됨.

|  |  |
| --- | --- |
|  |  |
| 나쁜세팅: Tag 위치는 OK, 회전이 안맞음 | 좋은세팅: 위치 OK 회전 OK |

1. 코드 실행

Localization\_april\_tag/1\_compute\_extrinsic.py

1. 결과

* 이제 extrinsic\_calibration\_result.json이 저장됬음.
* T\_camera\_world가 뭐냐? 무섭게 생겼는데 디테일은 알 필요 없고, 카메라 좌표계 기준 우리 기준 좌표계(world) 의 변환 관계임. 딱 한가지, T\_camera\_world와 T\_world\_camera의 관계를 한번 보셈. gpt한테 물어보던가.
* 아무튼 이를 사용하면 우리 기준 좌표계 기준 카메라 위치 도 알아낼 수 있음!
* 근데 우리는 기준 좌표계를 로봇 좌표계랑 일치시켰으니까 임. 즉,
  + 카메라: “내 기준 5cm앞에 있어!!! 와서 잡어!!!!”
  + 로봇 좌표계 == 기준 좌표계: “확인 지금 딱 기다리라노”

**물체 잡기를 위한 인식**

이제 너는 intrinsic, extrinsic을 구해서 파일로 저장했음. 말은 어려운데 별거아님. 어차피 코드가 다 치는거.

다시 복기하자면

* intrinsic은 카메라가 바뀌었거나 줌 같은거 건드리지 않으면 새로 안구해줘도 됨.
* Extrinsic은 환경을 새로 세팅할 때, 혹은 너가 실수로 로봇 베이스나 카메라를 움직였을 때 새로 구해줘야 댐.

아무튼 로봇과 카메라가 extrinsic 측정 이후 안움직인다고 치자. 그럼 tag 1,2,3,4는 무시하고 그냥 저장한 파일들 불러와서 쓰면되잖아? 이걸 바탕으로 카메라가 본 위치를 로봇 기준으로 넘겨줄 수 있다.

추후 태그 없이 100% 이미지 기반으로 확장 가능하지만, 지금은 간단한 예제를 구상한다. 물체 및 목적지 위치 파악을 위한 예제다.

1. 환경 설정.

* 물체: Tag ID: 5
* 수집 통: Tag ID: 6

여기에서는 실제 물체를 들어올리는 걸 하진않음. 다만 그 직전까지 가보려고 함.

1. 코드 실행

* Localization\_april\_tag/2\_read\_object\_and\_target.py실행하셈
* 해당 코드는, Tag ID:5,6에 대해 기존에 카메라 기준 태그 위치뿐만 아니라, 로봇(=global)기준 태그 위치도 반환함.
* 이전과 차이점은, 더 이상 로봇 위치를 동적으로 추정하지 않는다는 것임. 그냥 어디 적어놓고 사용함. 이는 불필요한 컴퓨터 연산량 줄여 속도 증가 목적임.

1. 결과

* 이게 의미하는게 뭐냐?

1. 칼리브레이션은 완료됬고 카메라가 보면, 로봇이 집으러 갈수 있다는것임. 로봇 기준 태그 위치 떤져주면, 로봇은 알아서 잡으러 갈것임.

2. 카메라를 창에 띄우는 작업 하지 않아도, (내가) 태그 위치를 얻어내는 코드 잘 쳐놓으면 너는 그냥 태그 값을 땡겨 쓸수 있다는 뜻임. 이런 식으로 말이지

Handle = TrackerHandle(ID=5, Size=8츠)

object\_pose = Handle.get\_pos()

Print(object\_pose) # [0.2, 0, 0.4]m, 로봇 프레임 기준

Robot.move\_to(object\_pose) # 잡으러감

|  |  |
| --- | --- |
|  |  |
|  |  |

**다음 컨텐츠 예고**

* **로봇 프레임 기준 태그(ID:5)된 물체 위치 기반으로 들어올리고 목표지점까지 이송**
* **춘식이 잡으러 가기**