아래와 같은 순서로 전체 파이프라인을 구성하면, 자동으로 3가지 색(빨강-Green-파랑) 정육면체를 하나씩 인식해서 집어 옮기는 과정을 구현할 수 있습니다. 큰 흐름은 "① 색상별 중심 좌표 검출 → ② 실제 좌표(로봇 베이스 기준) 변환 → ③ 역기구학 계산 → ④ 관절값(DXL 모터) 계산 → ⑤ 로봇 이동 및 그리퍼 조작"의 반복입니다. 각 단계별로 필요한 함수와 호출 순서를 정리해 보았습니다.

---

1. 모듈·함수 구조 잡기

먼저, 아래 네 파일을 기반으로 몇 가지 모듈(함수)을 준비합니다.

- **make_contour_centroid(1)_gpt.py**: 색상 별 컨투어 검출 → 픽셀 좌표(centroid) 반환
- **pixel_to_meter_GPT.py**: 픽셀 좌표 → 실제(mm) 좌표 변환 함수
- **moving_robot.py**:
  - `i_kine(joints_init, target, DH_params, …)` – 주어진 목표 좌표로 역기구학 계산 후 관절 라디안값 반환
  - DH 매개변수, `FK_func`, `J_func` 등 상수 및 유틸함수 포함
- **motorLLC_Protocol_TP.py**:
  - `motorLLC.open()`, `motorLLC.torque_enable()`, `motorLLC.moveTo(positions, velocities)`, `motorLLC.close()` – Dynamixel 7축 제어 클래스

위 모듈을 바탕으로, 최종적으로는 새로운 파일 **main_controller.py** (예: 텀프로젝트 메인 스크립트) 하나를 만들어 전체 흐름을 자동으로 처리하도록 합니다.

---

2. 단계별 세부 구현과 함수화

아래 항목들을 참고하여 각 모듈 안에 있는 핵심 기능을 잘 분리하고, `main_controller.py` 에서 차례로 호출하도록 구성합니다.

2.1. 카메라 이미지에서 색상별 중심 좌표 검출

1. `make_contour_centroid(1)_gpt.py` 안의 무한루프 형태 코드를 함수화합니다.

python 복사 편집

```
# make_contour_centroid(1)_gpt.py import cv2 import numpy as np from pixel_to_meter_GPT import pixel_to_real_position
MIN_AREA = 500 MAX_AREA = 10000 FRAME_WIDTH, FRAME_HEIGHT = 1280, 720 morph_kernel =
cv2.getStructuringElement(cv2.MORPH_RECT, (7, 7)) def get_color_centroids(): """ 카메라로부터 프레임을 읽어서, RED, GREEN, BLUE
색상별로 컨투어의 중심 픽셀 좌표(centroid)를 하나씩 반환한다. 찾지 못한 색상은 None으로 반환. Returns: (r_centroid, g_centroid,
b_centroid) 각 centroid는 (cX, cY) 픽셀값 or None """ cap = cv2.VideoCapture(1, cv2.CAP_DSHOW)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, FRAME_WIDTH) cap.set(cv2.CAP_PROP_FRAME_HEIGHT, FRAME_HEIGHT) if not cap.isOpened(): raise
RuntimeError("Cannot open camera") r_centroid = None g_centroid = None b_centroid = None while True: ret, frame = cap.read()
if not ret: continue # Blur → Grayscale → Canny → Morphology blurred = cv2.GaussianBlur(frame, (7, 7), 0) gray =
cv2.cvtColor(blurred, cv2.COLOR_BGR2GRAY) edges = cv2.Canny(gray, 40, 180) edges_closed = cv2.morphologyEx(edges,
cv2.MORPH_CLOSE, morph_kernel, iterations=2) # Contour 검출 contours, _ = cv2.findContours(edges_closed, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE) rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB) for cnt in contours: area = cv2.contourArea(cnt)
if area < MIN_AREA or area > MAX_AREA: continue # 컨투어 내부 마스크 생성 → RGB 평균 계산 mask = np.zeros(frame.shape[:2],
dtype=np.uint8) cv2.drawContours(mask, [cnt], -1, 255, -1) r_mean, g_mean, b_mean = cv2.mean(rgb_frame, mask=mask)[:3] # 색상
분류 if r_mean > g_mean and r_mean > b_mean and r_centroid is None: label = "RED" elif g_mean > r_mean and g_mean > b_mean
and g_centroid is None: label = "GREEN" elif b_mean > r_mean and b_mean > g_mean and b_centroid is None: label = "BLUE" else:
continue # 컨투어 무게중심 M = cv2.moments(cnt) if M["m00"] == 0: continue cX = int(M["m10"] / M["m00"]) cY = int(M["m01"] /
M["m00"]) if label == "RED": r_centroid = (cX, cY) elif label == "GREEN": g_centroid = (cX, cY) else: b_centroid = (cX, cY) #
RED, GREEN, BLUE 모두 검출되면 종료 if r_centroid and g_centroid and b_centroid: break # 'q' 눌러서 강제로 빠져나올 수 있도록 예
외 처리 if cv2.waitKey(1) & 0xFF == ord('q'): break cap.release() cv2.destroyAllWindows() return r_centroid, g_centroid,
b_centroid
```

- 이 함수는 세 가지 색이 모두 검출될 때까지 루프를 돌고, (cX, cY) 픽셀 좌표 튜플을 반환합니다.

2.2. 픽셀 좌표 → 실제 좌표 변환

- **pixel_to_meter_GPT.py** 에 이미 정의된 `pixel_to_real_position(Px, Py)` 함수 사용.

python 복사 편집

```
from pixel_to_meter_GPT import pixel_to_real_position # pixel_to_real_position: (Px, Py) → (X_robot, Y_robot) 리턴 (mm 단위)
```

2.3. 역기구학 계산

- **moving_robot.py** 에 이미 존재하는 `i_kine(joints_init, target, DH_params, …)` 함수를 사용합니다.
  - `joints_init` : 현재 로봇의 관절(radian) 초기값 배열 (7×1 numpy array)
  - `target` : 4×4 유니폼 트랜스포메이션 행렬. 여기서는 우리가 관절 위치만 제어할 테니,
    - 예를 들어, 그리퍼 끝점이 가야 할 좌표 (X, Y, Z)만 지정하고 orientation(회전)은 고정시킨 뒤,
    - 아래처럼 변환 행렬을 구성합니다.

```python
import numpy as np def make_target_matrix(x_mm, y_mm, z_mm): """ 그리퍼엔드 이펙터가 도달해야 할 위치(x_mm, y_mm, z_mm)를 입력받아, orientation을 임의로 고정한 4×4 행렬을 반환한다. (예: x축 1, y축 -1, z축 -1 방향 고정; 필요에 맞게 수정.) """ # 예시: x축 정렬, y축 뒤집힘, z축 뒤집힘 (moving_robot.py 예시 코드와 동일) rot = np.array([ [ 1, 0, 0], [ 0, -1, 0], [ 0, 0, -1] ]) target = np.zeros((4, 4)) target[:3, :3] = rot target[0, 3] = x_mm target[1, 3] = y_mm target[2, 3] = z_mm target[3, 3] = 1 return target
```

  - 그런 뒤, `i_kine(joints_init, target, DH_params, error_trace=False, no_rotation=True)` 호출하면,
    - `(joints, e_trace)` or `joints` 반환.
    - 실제 구현 시 `no_rotation=True` 로 orientation 오류 트레이싱 생략하고 포지션만 맞춥니다.

2.4. 관절 라디안값 → Dynamixel 펄스(pulse) 단위 변환

- Dynamixel MX-28(예제) 기준
  - 0 ~ 1023 펄스 ⟷ 0 ~ 300°(약 5.23599 rad)
  - 1 펄스당 약 0.29°(0.00508 rad)
- 예를 들어, 라디안값 θ를 펄스로 바꾸려면

```python
import math def radian_to_pulse(theta_rad): """ MX 시리즈 (0~300° → 0~1023 펄스) 맵핑. -180° ~ +180° 범위가 아니라 0~300° 범위로 모터가 셋팅되어 있으면, theta_rad(라디안) 값을 [0, 5.23599] 사이로 클램핑한 뒤 0~1023으로 변환해야 한다. - 프로젝트 환경에 맞게 라디안→펄스 매핑을 조정. """ # 0 라디안(0°) → 512 펄스(중립), ±150° 각각 0/1023에 매핑되는 방식이라면: # 예시: θ(rad) → degree, degree → pulse = degree * (1023/300) + 512 deg = theta_rad * 180.0 / math.pi pulse = int(deg * (1023.0/300.0) + 512) # 클램핑 if pulse < 0: pulse = 0 if pulse > 1023: pulse = 1023 return pulse
```

- 만약 `moving_robot.py` 내부에 이미 이 매핑이 되어 있거나, 사용자 맞춤 모터 스케일을 적용했다면 그대로 사용하시면 됩니다.
- 최종적으로 `[theta1, ..., theta7]` 라디안 배열을 펄스 배열 `[pulse1, ..., pulse7]` 로 바꾼 뒤 `motor.moveTo(pulses, velocities)` 함수에 넣어 줍니다.
  - `velocities` 는 각 모터별 속도(pulse/s) 리스트. 일단은 동일한 속도로 설정해 놓고, 필요에 따라 조정합니다.

2.5. 그리퍼 조작

- 프로젝트에 따라 별도의 그리퍼 모터가 1축으로 구성되어 있을 수 있습니다.
- 예를 들어, DXL8번 ID를 그리퍼 전용으로 사용하거나, 서보 모터를 별도 연결했을 경우,
  - `motorLLC` 클래스에 그리퍼 축 ID를 추가
  - `gripper_open()` / `gripper_close()` 함수를 만들어 펄스값을 로우/하이로 전환
- 간단 예시:

```python
# motorLLC_Protocol_TP.py 수정 예시 class motorLLC(): def __init__(self): # 기존 DXL1~7에 추가로 DXL8 그리퍼 할당 self.gripper_id = 8 self.ids = [1,2,3,4,5,6,7] # 관절 ID # ... 생략 ... def gripper_close(self): # 그리퍼를 닫는 펄스값 (실험 → 적절한 값으로 조정) grip_close_pulse = 600 res, err = self.packetHandler.write2ByteTxRx( self.portHandler, self.gripper_id, ADDR_MX_GOAL_POSITION, grip_close_pulse ) # 에러 체크 생략 def gripper_open(self): grip_open_pulse = 400 res, err = self.packetHandler.write2ByteTxRx( self.portHandler, self.gripper_id, ADDR_MX_GOAL_POSITION, grip_open_pulse ) # 에러 체크 생략
```

- 만약 그리퍼가 전용 컨트롤러(예: 펌프형 공압 그리퍼 등)라면, 별도 I/O 제어나 직렬통신 모듈을 통해 동작한다는 점만 참고하세요.

---

3. 최종 메인 스크립트 예시 (`main_controller.py`)

아래는 전체 흐름을 묶어놓은 파이썬 스크립트 예시입니다. 실제 세부 튜닝(속도, 오프셋, 클램핑 등)은 실험하면서 조정하시기 바랍니다.

```python
```

```
# main_controller.py import time import numpy as np # (1) 색상 중심 픽셀 좌표 함수 from make_contour_centroid import
get_color_centroids # (2) 픽셀→실제 좌표 변환 from pixel_to_meter_GPT import pixel_to_real_position # (3) 역기구학 및 DH 파라미터
from moving_robot import i_kine, DH_params # (4) 모터 제어 클래스 from motorLLC_Protocol_TP import motorLLC # (5) 펄스 변환 유틸
import math def radian_to_pulse(theta_rad): deg = theta_rad * 180.0 / math.pi pulse = int(deg * (1023.0/300.0) + 512) if pulse <
0: pulse = 0 if pulse > 1023: pulse = 1023 return pulse def make_target_matrix(x_mm, y_mm, z_mm): # moving_robot.py 예시와 동일
orientation rot = np.array([[1,0,0],[0,-1,0],[0,0,-1]]) target = np.zeros((4,4)) target[:3, :3] = rot target[0, 3] = x_mm
target[1, 3] = y_mm target[2, 3] = z_mm target[3, 3] = 1 return target def main(): # 1) 모터 클래스 생성 및 초기화 motor =
motorLLC() motor.open() motor.torque_enable() # 2) 초기 관절 위치 (예: 전부 0 rad) joints_init = np.zeros((7,1)) # [[0],[0],...,
[0]] current_joints = joints_init.copy() # 3) 우선, 색상별 픽셀 중심 좌표 한 번에 모두 읽어오기 # (r_centroid, g_centroid,
b_centroid)에 픽셀 좌표 (cX, cY) 튜플이 담긴다. print("색상별 큐브 인식 대기 중...") r_pix, g_pix, b_pix = get_color_centroids()
print(f"RED 픽셀 좌표: {r_pix}, GREEN 픽셀 좌표: {g_pix}, BLUE 픽셀 좌표: {b_pix}") # 4) 픽셀→실제 좌표(mm) 변환 r_mm =
pixel_to_real_position(r_pix[0], r_pix[1]) g_mm = pixel_to_real_position(g_pix[0], g_pix[1]) b_mm =
pixel_to_real_position(b_pix[0], b_pix[1]) # r_mm, g_mm, b_mm은 (X_robot_mm, Y_robot_mm) 튜플. Z는 바닥 높이를 수동 지정. # 예: 바닥
높이(z) = 50 mm (실험 환경에 맞게 수정) z_pick = 50.0 coordinates = { "RED": (r_mm[0], r_mm[1], z_pick), "GREEN": (g_mm[0],
g_mm[1], z_pick), "BLUE": (b_mm[0], b_mm[1], z_pick) } # 5) 순서대로 각 색상 큐브를 집고 놓기 위한 목표 지점(Place 위치) 정의 # 예: 선
반 위나 박스 내부 특정 좌표 (실험 환경에 맞게 수정) place_positions = { "RED": (200.0, 100.0, z_pick), "GREEN": (200.0, 200.0,
z_pick), "BLUE": (200.0, 300.0, z_pick) } # 6) 그리퍼 닫힘/열림 테스트 (필요 시) # motor.gripper_open() # time.sleep(0.5) #
motor.gripper_close() # time.sleep(0.5) # 7) 색상 순회 (RED → GREEN → BLUE) for color in ["RED", "GREEN", "BLUE"]: print(f"{color}
큐브 처리 시작") # 7.1) Pick 위치로 IK 계산 x_pick, y_pick, z_pick = coordinates[color] target_pick = make_target_matrix(x_pick,
y_pick, z_pick) joints_result = i_kine(current_joints, target_pick, DH_params, error_trace=False, no_rotation=True) #
joints_result: 7×1 numpy array (라디안) # 다음 사이클 시 초기값으로 사용 current_joints = joints_result.copy() # 7.2) 각 관절 라디안
→ 펄스로 변환 rad_list = joints_result.flatten().tolist() pulse_list = [radian_to_pulse(rad) for rad in rad_list] # 동일 속도(예:
100)로 설정 vel_list = [100] * len(pulse_list) # 7.3) 로봇 이동 → 그리퍼 열고 내려서 집기 motor.moveTo(pulse_list, vel_list)
time.sleep(2.0) # 모터가 움직이는 시간 (속도에 따라 조정) # 7.4) 그리퍼 닫기 → 큐브 집기 motor.gripper_close() time.sleep(1.0) # 7.5)
Place 위치로 IK 계산 x_place, y_place, z_place = place_positions[color] target_place = make_target_matrix(x_place, y_place,
z_place) joints_result = i_kine(current_joints, target_place, DH_params, error_trace=False, no_rotation=True) current_joints =
joints_result.copy() # 7.6) 라디안→펄스 변환 후 이동 rad_list = joints_result.flatten().tolist() pulse_list = [radian_to_pulse(rad)
for rad in rad_list] motor.moveTo(pulse_list, vel_list) time.sleep(2.0) # 7.7) 그리퍼 열기 → 큐브 내려놓기 motor.gripper_open()
time.sleep(1.0) print(f"{color} 큐브 이동 완료\n") # (필요 시, 휴식 타임) time.sleep(1.0) # 8) 마무리: 모터 토크 오프 & 포트 닫기
motor.close() print("모든 작업 완료, 모터 포트 종료") if __name__ == "__main__": main()
```

4. 설명 및 주의사항

1. **카메라 해상도·FOV 보정**
   - `pixel_to_meter_GPT.py` 에서 가정한 `X_max`, `Y_max`, `X_offset_mm` 값은 실제 실험 환경에 맞춰 조정해야 합니다.
   - 카메라의 **실제 FOV(field of view)**와 로봇 기준 좌표계(origin) 위치 관계를 정확히 측정한 뒤, `pixel_to_real_position()` 식을 조정해야 실제 좌표 오류를 최소화할 수 있습니다.
2. **역기구학 초기 관절값**
   - 매 사이클마다 `current_joints` 를 업데이트하여, 이전 동작 관절값을 초기 입력으로 넣어 주는 이유는 IK 수렴 속도를 높이고, 불필요한 큰 궤적 변경을 줄이기 위함입니다.
   - 만약 초기 관절을 매번 전부 0으로 줘도 충분하다면, `joints_init = np.zeros((7,1))` 고정으로 두고 사용해도 되지만, 로봇이 갑자기 튕기는 현상이 생길 수 있습니다.
3. **속도 및 딜레이(time.sleep)**
   - `time.sleep(2.0)` 처럼 지연을 주는 이유는 "모터가 목표 펄스까지 이동하는 시간"을 충분히 확보하기 위함입니다.
   - 모터 속도를 높이거나 무거운 그리퍼/물체를 다룰 때는 **충분한 시간**을 확보해야 관절이 목표 위치에 안정적으로 착지할 수 있습니다.
   - 실제 테스트하면서, "로봇이 목표 위치에 거의 도달했을 때" 신호를 감지하는 센서(예: 모터 현재 위치 판독)를 추가로 활용할 수도 있습니다.
4. **그리퍼 제어**
   - 위 예시처럼 `motor.gripper_open()` / `motor.gripper_close()` 를 별도 함수로 두면 편하지만, 실제 그리퍼 메커니즘(공압, 서보, DXL 등)에 따라 제어 방법이 달라질 수 있습니다.
   - 공압 그리퍼의 경우, 별도 디지털 IO 보드나 PLC로 신호를 줘야 할 수도 있고, 서보 모터라면 PWM 신호를 별도 처리해야 합니다.
   - 화면상 예시 코드는 "DXL 8번 ID를 그리퍼"로 가정했으므로, 실제 ID나 모터 타입이 다를 경우 꼭 수정해 주세요.
5. **좌표값 정렬 및 순서 지정**
   - 예시에서는 `["RED", "GREEN", "BLUE"]` 순서로 처리했지만, 실제로 색상 인식 순서가 뒤바뀌거나 로봇이 장애물을 만날 수 있습니다.
   - 필요하다면 "가장 가까운 색상부터 먼저 처리" 같은 로직을 추가하거나, "인식→리스트로 정렬(예: x 좌표 기준 오름차순)" 후 순차적으로 처리할 수도 있습니다.
6. **오차 보정**
   - 픽셀→실제 좌표 변환 시에는 렌즈 왜곡(distortion)이 들어가면 오차가 커집니다.
   - 스테레오 카메라/캘리브레이션 라이브러리 등을 사용해 "왜곡 보정"을 미리 한 뒤, `pixel_to_real_position()` 함수의 입력으로 쓰는 것이 정확도를 높이는 방법입니다.

7. **디버깅 포인트**
    1. 색상별 컨투어가 제대로 인식되는지
    2. `pixel_to_real_position()` 변환 결과가 로봇 베이스 기준으로 합리적인지 (예: (0,0) 위치가 진짜 로봇 기준인가)
    3. `i_kine()` 함수가 수렴하지 않거나 잘못된 관절값을 반환하는지 확인 (예: unreachable target)
    4. 관절 라디안 → 펄스 변환 식이 올바른지
    5. 실제 모터가 목표 펄스까지 이동했을 때, 값이 틀어져 있는지(슬립, 과부하 등)

---

5. 요약

1. **get_color_centroids()** → 픽셀 기준 색상 중심 좌표 (cX, cY) 가져오기
2. **pixel_to_real_position(cX, cY)** → 로봇 베이스 기준 실좌표 (X_mm, Y_mm)
3. **make_target_matrix(X_mm, Y_mm, Z_mm)** → 4×4 목표 변환 행렬 생성
4. **i_kine(current_joints, target, DH_params)** → 역기구학으로 관절 라디안값 반환
5. **radian_to_pulse(θ_rad)** → Dynamixel 펄스값으로 변환
6. **motor.moveTo(pulse_list, vel_list)** → 로봇 관절 동작
7. **motor.gripper_open()/gripper_close()** → 그리퍼 집고/놓기

이렇게 함수 단위로 분리하여, `main_controller.py` 에서 위 순서대로 호출만 해 주면 "색상 인식 → IK 계산 → 동작 → 집기/놓기" 흐름이 자동으로 반복됩니다. 프로젝트 환경에 맞춰 매개변수(높이 Z, 속도, 펄스 맵핑 상수 등)를 조정하면서 한 번씩 테스트해 보시기 바랍니다.

필요한 부분 혹은 코드 내부 세부을 더 궁금하시면 언제든 물어보세요. 성공적인 텀프로젝트 되길 바랍니다.