

DẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH  
TRƯỜNG ĐẠI HỌC BÁCH KHOA  
KHOA KHOA HỌC - KỸ THUẬT MÁY TÍNH



## BÁO CÁO ĐỒ ÁN THIẾT KẾ LUẬN LÝ

---

Đề tài

OBJECT CLASSIFICATION WITH ROBOTIC

---

**GVHD:** Trần Thanh Bình  
**SV:** Lê Trung Trực - 2115161  
Phạm Hồng Quân - 2114554  
Nguyễn Hữu Thọ - 2114911



## Mục lục

<b>1 Giới thiệu đề tài</b>	<b>4</b>
1.1 Giới thiệu . . . . .	4
1.2 Mục tiêu . . . . .	4
<b>2 Nền tảng và phần cứng</b>	<b>4</b>
2.1 Giới thiệu về ROS . . . . .	4
2.1.1 Định nghĩa ROS . . . . .	4
2.1.2 Cài đặt ROS (version Noetic) . . . . .	4
2.1.3 Những khái niệm cơ bản trong ROS . . . . .	5
2.1.4 Catkin và Cmake . . . . .	5
2.1.5 Rviz . . . . .	5
2.2 Phần mềm mô phỏng Gazebo . . . . .	6
2.2.1 Gazebo là gì? . . . . .	6
2.2.2 Cài đặt Gazebo . . . . .	6
2.2.3 Không gian làm việc Gazebo . . . . .	6
2.3 Gói Moveit . . . . .	7
2.4 UR5 - Universal Robots . . . . .	7
2.4.1 Tìm hiểu về cánh tay UR5 . . . . .	7
2.4.2 Thông số kỹ thuật . . . . .	8
2.5 OpenCV . . . . .	8
2.5.1 OpenCV là gì? . . . . .	8
2.5.2 Ứng dụng của OpenCV . . . . .	9
2.5.3 Tính năng và các module phổ biến của OpenCV . . . . .	9
2.6 Kinect V1 sensor . . . . .	10
2.6.1 Thông số kỹ thuật . . . . .	10
2.7 Nguyên lý làm việc . . . . .	11
<b>3 Mô tả ý tưởng</b>	<b>11</b>
3.1 Ý tưởng hệ thống . . . . .	11
3.2 Ý tưởng mô phỏng . . . . .	13
3.2.1 Trường hợp mỗi loại 1 vật thể . . . . .	13
3.2.2 Trường hợp mỗi loại có nhiều vật thể . . . . .	13
3.3 Kết quả mong đợi . . . . .	13
<b>4 Spawn vật thể ngẫu nhiên</b>	<b>14</b>
<b>5 Nhận diện vật thể</b>	<b>15</b>
5.1 Phân loại vật thể . . . . .	15
5.1.1 Chuẩn bị dữ liệu . . . . .	16
5.1.2 Lựa chọn mô hình . . . . .	16
5.1.3 Huấn luyện mô hình . . . . .	18
5.1.4 Dánh giá kết quả huấn luyện mô hình . . . . .	23
5.2 Hiện thực . . . . .	26
5.2.1 Luồng thực thi: . . . . .	26
5.2.2 Các hàm chính . . . . .	26



---

<b>6 Điều khiển cánh tay UR5</b>	<b>40</b>
6.1 Lý thuyết cơ bản để điều khiển một cánh tay robot . . . . .	40
6.1.1 Ma trận xoay(Rotaion Matrix) . . . . .	40
6.1.2 Ma trận Homogeneous . . . . .	40
6.1.3 Động học tĩnh tiến(Forward Kinematics) . . . . .	41
6.1.4 Mô hình Denavit-Hartenberg . . . . .	42
6.1.5 Chuyển động đảo ngược(Inverse Kinematics) . . . . .	43
6.2 Hiện thực . . . . .	44
6.2.1 kinematics.py: . . . . .	44
6.2.2 controller.py . . . . .	47
6.2.3 motion_planning.py . . . . .	49
<b>7 Kết luận và hướng phát triển</b>	<b>56</b>
7.1 Kết luận . . . . .	56
7.1.1 Đạt được . . . . .	56
7.1.2 Hạn chế . . . . .	56
7.2 Hướng phát triển . . . . .	56
<b>8 Tài liệu tham khảo</b>	<b>57</b>



## Danh sách hình vẽ

1	Không gian làm việc của Gazebo . . . . .	6
2	Hình ảnh cánh tay robot UR5 ngoài thực tế . . . . .	7
3	Sơ đồ của Kinect v1 . . . . .	10
4	Sơ đồ cấu trúc hệ thống . . . . .	11
5	Thế giới mô phỏng trong Gazebo . . . . .	13
6	Mô phỏng nhiều vật thể hơn . . . . .	14
7	Kết quả mong đợi cho trường hợp mỗi loại 1 vật thể . . . . .	14
8	Kết quả mong đợi cho trường hợp mỗi loại có nhiều vật thể . . . . .	15
9	Các mảnh lego dùng để phân loại . . . . .	16
10	Gán nhãn bằng MakeSense.AI . . . . .	17
11	Định dạng file lable YOLOv5 . . . . .	17
12	Bắt đầu với Google Colab . . . . .	19
13	Cài đặt YOLOv5 trên Google Colab . . . . .	19
14	Ví dụ về tập dữ liệu cần chuẩn bị . . . . .	20
15	Tải tập dữ liệu lên Google Colab . . . . .	20
16	Chỉnh sửa file cấu hình thư viện huấn luyện . . . . .	21
17	Kết quả lần huấn luyện đầu tiên . . . . .	22
18	Kết quả lần huấn luyện thứ hai . . . . .	23
19	Tỷ lệ nhầm lẫn giữa các vật . . . . .	23
20	Biểu đồ kết quả huấn luyện . . . . .	24
21	Kết quả kiểm thử mô hình phân loại sau khi huấn luyện . . . . .	25
22	Dữ liệu ảnh màu chuyển từ ROS message sang OpenCV . . . . .	28
23	Dữ liệu ảnh chiều sâu chuyển từ ROS message sang OpenCV . . . . .	28
24	Các thành phần trong không gian màu HSV . . . . .	29
25	Ví dụ chuyển đổi một ảnh từ không gian màu RGB sang không gian màu HSV . . . . .	30
26	Vẽ the distorted lego bounding box và highlight đỉnh gần nhất . . . . .	35
27	Vẽ khung tọa độ và nhãn trên khung hình . . . . .	39
28	Xoay một góc $\theta$ quanh lần lượt các trục x,y,z . . . . .	40
29	Youtube: Engineering Simplified . . . . .	41
30	Structure diagram . . . . .	42



## 1 Giới thiệu đề tài

### 1.1 Giới thiệu

Cánh tay robot công nghiệp những năm gần đây đã trở thành ‘trợ thủ đắc lực’ trong các nhà máy sản xuất. Cánh tay robot có vai trò quan trọng, thay thế con người trong các hoạt động sản xuất mang tính lặp lại và trong các môi trường làm việc rủi ro cao.

Cánh tay robot công nghiệp (Cánh tay cơ khí) là một thiết bị được lập trình để hoạt động tương tự như cánh tay con người, với các khớp chuyển động theo một trực dọc và có thể xoay theo các hướng nhất định.

Cánh tay robot với nhiều lợi ích như tăng tốc độ sản xuất, cải thiện chất lượng, giảm chi phí vận hành, chi phí nhân công nên được ứng dụng rộng rãi trong nhiều lĩnh vực như y tế, sản xuất - chế biến thực phẩm, hàng không vũ trụ,...

### 1.2 Mục tiêu

Với những lợi ích của cánh tay Robot ở trên mà nhóm đã chọn đề tài "OBJECT SORTING WITH ROBOTIC ARM" để tạo một cánh tay robot mô phỏng đơn giản có tính năng nhận diện đồ vật và gấp vật đến vị trí được lập trình sẵn.

## 2 Nền tảng và phần cứng

### 2.1 Giới thiệu về ROS

#### 2.1.1 Định nghĩa ROS



Robot Operating System (ROS) là một hệ thống phần mềm chuyên dụng để lập trình và điều khiển robot, cung cấp các thư viện và công cụ để giúp các nhà phát triển phần mềm tạo ra các ứng dụng robot. ROS có những ưu điểm nổi bật như là:

1. ROS là hệ điều hành meta, mã nguồn mở, hoàn toàn miễn phí.
2. ROS có thể được lập trình bằng nhiều ngôn ngữ khác nhau như C, C++, Python, ... (source code của nhóm được viết chủ yếu bằng Python).
3. ROS Kết hợp trình điều khiển và thuật toán từ các dự án nguồn mở khác:
  - Trình mô phỏng dự án Player / Stage.
  - Thư viện xử lý hình ảnh và tầm nhìn nhân tạo từ OpenCV.
  - Thuật toán lập kế hoạch từ OpenRave.

#### 2.1.2 Cài đặt ROS (version Noetic)

Hiện nay có khá nhiều phiên bản ROS như là ROS Kinetic, ROS Melodic, ROS Dashing,... Do phần lớn thành viên trong nhóm sử dụng hệ điều hành Ubuntu phiên bản 20.04 nên nhóm quyết định sử dụng ROS Noetic vì đây là phiên bản hỗ trợ tốt nhất dành cho Ubuntu 20.04.



### 2.1.3 Những khái niệm cơ bản trong ROS

#### 1. ROS Master và Node

**ROS Master** cung cấp các dịch vụ đặt tên và đăng ký (naming and registration) cho các node trong hệ thống ROS. Nó theo dõi việc publish (truyền) và subscribe (nhận) của các node cũng như các topic và service. ROS Master được gọi bằng lệnh **roscore**. Chỉ có một ROS Master được phép chạy tại một thời điểm.

**ROS Node** thực hiện một tác vụ nào đó và luôn phải đăng ký (register) với Master khi nó được khởi tạo.

#### 2. ROS Topic và Message

**ROS Message** có thể dịch là tin nhắn hoặc dữ liệu và chúng thuộc loại **sensor\_msgs**.

**ROS Topic** là nơi mà các ROS Node có thể *publish* (truyền Message) hoặc *subscribe* (nhận Message) trong ROS.

#### 3. ROS Service

**ROS Service** cũng là một phương thức truyền-nhận dữ liệu trong ROS, nhưng **ROS Service** hoạt động theo nguyên lý Client-Server. Khi ROS Node có vai trò như Client gửi request tới ROS Node vai trò Server thì Node Server mới reply theo yêu cầu của Client.

### 2.1.4 Catkin và Cmake

#### Hệ thống xây dựng Catkin

Catkin là hệ thống xây dựng của ROS. Về cơ bản, Catkin sử dụng Cmake, có môi trường xây dựng được mô tả trong "CmakeLists.txt". Hệ thống xây dựng Catkin giúp dễ dàng sử dụng các bản dựng liên quan đến ROS, quản lý gói và sự phụ thuộc giữa các gói.

#### CMake and CMakeList.txt

Make đa nền tảng (CMake) là một nhóm công cụ đa nền tảng, mã nguồn mở được thiết kế để xây dựng, thử nghiệm và đóng gói phần mềm. Catkin, hệ thống xây dựng của ROS, sử dụng CMake theo mặc định. Môi trường xây dựng được chỉ định trong tệp "CMakeLists.txt" trong mỗi thư mục gói.

### 2.1.5 Rviz

Rviz (ROS Visualization), tên viết tắt của trực quan hóa ROS, là một công cụ trực quan hóa 3D mạnh mẽ cho ROS. Nó cho phép người dùng xem mô hình robot mô phỏng, ghi thông tin cảm biến từ các cảm biến của robot và phát lại thông tin cảm biến đã ghi. Bằng cách mô phỏng những gì robot đang nhìn thấy và làm, người dùng có thể sửa lỗi ứng dụng robot từ đầu vào cảm biến đến các hành động đã lên kế hoạch (hoặc không có kế hoạch).

Rviz hiển thị dữ liệu cảm biến 3D từ máy ảnh âm thanh nổi, tia laser, Kinects và các thiết bị 3D khác dưới dạng đám mây điểm hoặc hình ảnh chiều sâu. Dữ liệu cảm biến 2D từ webcam, camera RGB và máy đo khoảng cách laser 2D có thể được xem trong Rviz dưới dạng dữ liệu hình ảnh.

## 2.2 Phần mềm mô phỏng Gazebo

### 2.2.1 Gazebo là gì?



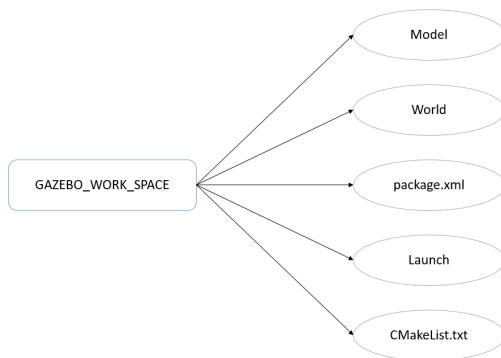
Gazebo là một công cụ mô phỏng 3D. Gazebo có thể được áp dụng cho robot design, testing AI khi ở giai đoạn đầu của các nghiên cứu. Nó được nhiều người ưa chuộng bởi:

1. Gazebo cung cấp GUI (Graphical User Interface) rất thân thiện với người dùng, với các thanh menu, có thể giúp người dùng tương tác trực tiếp trên gazebo mà không cần thông qua edit file code.
2. Gazebo là một công cụ mã nguồn mở, không tính phí.
3. Gazebo có một cộng đồng lớn.

### 2.2.2 Cài đặt Gazebo

Mô hình cánh tay gấp vật thể chạy trên hệ điều hành ROS, và ROS tương thích tốt đối với Ubuntu, nên nhóm đã quyết định cài đặt Gazebo phiên bản 11.0.0 trên Ubuntu để tiện cho việc mô phỏng.

### 2.2.3 Không gian làm việc Gazebo



Hình 1: Không gian làm việc của Gazebo

Không gian làm việc của Gazebo cần có thành phần sau:

- 2 file **package.xml** và **CMakeList.txt** để cấu hình và setup bên trong workspace.
- **Thư mục Model:** Chứa các vật thể, object có hình dạng cần sử dụng trong mô phỏng

- **Thư mục World:** Gồm các file **.world**, trong file này sẽ chứa các model nhỏ bên trong và vị trí của chúng xuất hiện trong một thế giới rộng hơn. Trong phần mô phỏng cánh tay robot gấp vật thể thì các file **.world** có thể là optional.

## 2.3 Gói Moveit

**MoveIt** là chương trình tiên tiến liên quan đến lập trình robot và đặc biệt là việc triển khai các tính năng lập kế hoạch chuyển động (Motion planning). “Nền tảng thao tác robot nguồn mở dễ sử dụng để phát triển các ứng dụng thương mại, thiết kế tạo mẫu và thuật toán đo điểm chuẩn”. [theep moveit.ros.org].

Mặc dù được sinh ra cho những chuyên gia về lập trình điều khiển robot nhưng giờ đây MoveIt được các thương hiệu lớn như Google, NASA, Microsoft,... sử dụng cho số lượng hệ thống robot ngày càng tăng, trở thành một tiêu chuẩn trên thực tế.

Về cơ bản, **MoveIt** cho phép triển khai mọi chức năng của cánh tay robot, từ lập kế hoạch chuyển động đến tránh va chạm và chức năng cảm biến, nó cũng triển khai chức năng để trừu tượng hóa hơn nữa các bộ điều khiển, chuyển từ chức năng đã được ROS cung cấp bằng cách ảo hóa khả năng điều khiển lên mức cao không dựa trên sức mạnh hay vị trí mà dựa trên chính quỹ đạo.

## 2.4 UR5 - Universal Robots

### 2.4.1 Tìm hiểu về cánh tay UR5



Hình 2: Hình ảnh cánh tay robot UR5 ngoài thực tế

Robot UR5 là sản phẩm của công ty *Universal Robot* - công ty chuyên sản xuất các robot khả lập trình. UR5 là 1 dòng cánh tay robot trong series các cánh tay robot như UR3, UR10, UR20,... Trong bài này, nhóm đã chọn robot UR5 để mô phỏng.

Cụ thể, UR5 lớn hơn một chút so với UR3 là lựa chọn lý tưởng cho việc tự động hóa các nhiệm vụ xử lý các sản phẩm nhẹ cân như nhắc lên, đặt xuống và kiểm tra. Cánh tay robot cỡ trung bình dễ lập trình, cài đặt nhanh và cũng giống như các thành viên khác trong gia đình UR.



#### 2.4.2 Thông số kỹ thuật

##### Hiệu suất

Khả năng lắp lại	$\pm 0.1mm / \pm 0.0039$ trong (4 dặm)
Nhiệt độ môi trường xung	0 – 50°C
Công suất tiêu thụ	Tối thiểu 90W, thông thường 150W, tối đa 325W

##### Thông số kỹ thuật

Trọng tải	5kg
Tầm với	850mm
Khả năng tự do hoạt động	6 khớp xoay

##### Chuyển động

Chuyển động trực cánh tay robot	Phạm vi làm việc	Tốc độ tối đa
Giá cố định	$\pm 360^\circ$	$\pm 180^\circ/\text{giây}$
Vai	$\pm 360^\circ$	$\pm 180^\circ/\text{giây}$
Khuỷu tay	$\pm 360^\circ$	$\pm 180^\circ/\text{giây}$
Cổ tay 1	$\pm 360^\circ$	$\pm 180^\circ/\text{giây}$
Cổ tay 2	$\pm 360^\circ$	$\pm 180^\circ/\text{giây}$
Cổ tay 3	$\pm 360^\circ$	$\pm 180^\circ/\text{giây}$

#### 2.5 OpenCV

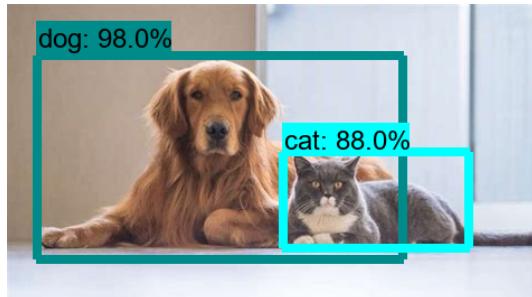


##### 2.5.1 OpenCV là gì?

OpenCV viết tắt cho Open Source Computer Vision Library. OpenCV là thư viện nguồn mở hàng đầu cho Computer Vision và Machine Learning, và hiện có thêm tính năng tăng tốc GPU cho các hoạt động theo real-time.

OpenCV được phát hành theo giấy phép BSD, do đó nó miễn phí cho cả học tập và sử dụng với mục đích thương mại. Nó có trên các giao diện C++, C, Python và Java và hỗ trợ Windows, Linux, Mac OS, iOS và Android. OpenCV được thiết kế để hỗ trợ hiệu quả về tính toán và chuyên dùng cho các ứng dụng real-time (thời gian thực). Nếu được viết trên C/C++ tối ưu, thư viện này có thể tận dụng được bộ xử lý đa lõi (multi-core processing).

### 2.5.2 Ứng dụng của OpenCV



OpenCV được sử dụng cho đa dạng nhiều mục đích và ứng dụng khác nhau như:

- Hình ảnh street view
- Kiểm tra và giám sát tự động
- Robot (đây là phần mà nhóm áp dụng vào đồ án)
- Xe hơi tự lái
- Phân tích hình ảnh trong y học
- và còn nhiều ứng dụng khác trong các thực tế ngày nay.

### 2.5.3 Tính năng và các module phổ biến của OpenCV

Theo tính năng và ứng dụng của OpenCV, có thể chia thư viện này thành các nhóm tính năng và module tương ứng như sau:

- Xử lý và hiển thị Hình ảnh/ Video/ I/O (*core, imgproc, highgui*)
- Phát hiện các vật thể (*objdetect, features2d, nonfree*)
- Geometry-based monocular hoặc stereo computer vision (*calib3d, stitching, videostab*)
- Computational photography (*photo, video, superres*)
- Machine learning & clustering (*ml, flann*)
- CUDA acceleration (*gpu*)

OpenCV có cấu trúc module, nghĩa là gói bao gồm một số thư viện liên kết tĩnh (static libraries) hoặc thư viện liên kết động (shared libraries). Xin phép liệt kê một số định nghĩa chi tiết các module phổ biến có sẵn như sau:

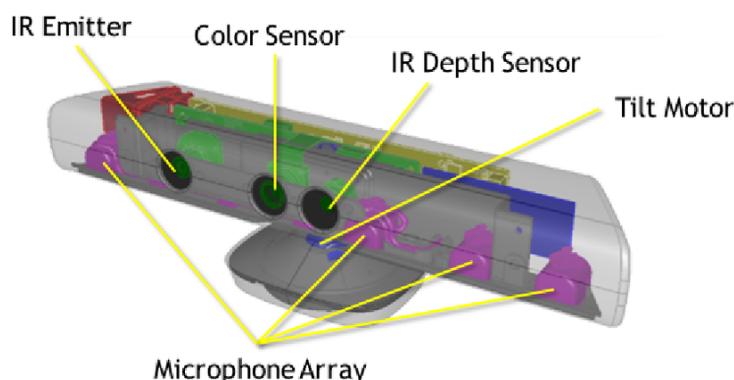
- **Core functionality** (*core*) - module nhỏ gọn để xác định cấu trúc dữ liệu cơ bản, bao gồm mảng đa chiều dày đặc và nhiều chức năng cơ bản được sử dụng bởi tất cả các module khác.
- **Image Processing** (*imgproc*) – module xử lý hình ảnh gồm cả lọc hình ảnh tuyến tính và phi tuyến (linear and non-linear image filtering), phép biến đổi hình học (chỉnh size, afin và warp phối cảnh, ánh xạ lại dựa trên bảng chung), chuyển đổi không gian màu, biểu đồ, và nhiều cái khác.

- **Video Analysis** (video) – module phân tích video bao gồm các tính năng ước tính chuyển động, tách nền, và các thuật toán theo dõi vật thể.
- **Camera Calibration and 3D Reconstruction** (calib3d) – thuật toán hình học đa chiều cơ bản, hiệu chuẩn máy ảnh single và stereo (single and stereo camera calibration), dự đoán kiểu dáng của đối tượng (object pose estimation), thuật toán thư tín âm thanh nổi (stereo correspondence algorithms) và các yếu tố tái tạo 3D.
- **2D Features Framework** (features2d) – phát hiện các đặc tính nổi bật của bộ nhận diện, bộ truy xuất thông số, thông số đối chọi.
- **Object Detection** (objdetect) – phát hiện các đối tượng và mô phỏng của các hàm được định nghĩa sẵn – predefined classes (vd: khuôn mặt, mắt, cốc, con người, xe hơi,...).
- **High-level GUI** (highgui) – giao diện dễ dùng để thực hiện việc giao tiếp UI đơn giản.
- **Video I/O** (videoio) – giao diện dễ dùng để thu và mã hóa video.
- **GPU** – Các thuật toán tăng tốc GPU từ các module OpenCV khác.
- ... và một số module hỗ trợ khác, ví dụ như FLANN và Google test wrapper, Python binding, v.v.

## 2.6 Kinect V1 sensor

### 2.6.1 Thông số kỹ thuật

Kinect là dòng thiết bị đầu vào cảm biến chuyển động do Microsoft sản xuất và phát hành lần đầu tiên vào năm 2010. Các thiết bị này thường chứa camera RGB, máy chiếu và máy dò hồng ngoại lập bản đồ độ sâu thông qua ánh sáng có cấu trúc hoặc thời gian tính toán chuyển bay, từ đó có thể được sử dụng để thực hiện nhận dạng cử chỉ theo thời gian thực và phát hiện bộ xương cơ thể, cùng với các khả năng khác.



Hình 3: Sơ đồ của Kinect v1

### Thông số kỹ thuật Kinect v1

Tính năng	Kinect v1
Cảm biến độ sâu	Ánh sáng có cầu
Dộ phân giải camera RGB	640 x 480, 30fps
Dộ phân giải camera hồng ngoại	320 x 240, 30fps
Trường ảnh RGB	62°x 48.6°
Trường nhìn ảnh chiều sâu	57°x 43°
Phạm vi đo hoạt động	0.8m - 4m

Bảng 1: Bảng thông số kỹ thuật Kinect v1

## 2.7 Nguyên lý làm việc

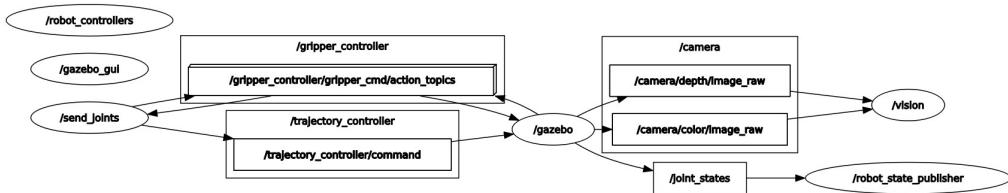
Kinect chứa ba bộ phận quan trọng phối hợp với nhau để phát hiện chuyển động của bạn và tạo ra hình ảnh vật lý của bạn trên màn hình: máy quay video VGA màu RGB, cảm biến độ sâu và micrô nhiều dải.

Máy ảnh sẽ phát hiện các thành phần màu đỏ, xanh lá cây và xanh lam cũng như các đặc điểm cơ thể và khuôn mặt. Nó có độ phân giải pixel là 640x480 và tốc độ khung hình là 30 khung hình/giây. Điều này giúp nhận dạng khuôn mặt và nhận dạng cơ thể.

Cảm biến độ sâu chứa cảm biến CMOS đơn sắc và máy chiếu hồng ngoại giúp tạo ra hình ảnh 3D khắp phòng. Nó cũng đo khoảng cách của từng điểm trên cơ thể người chơi bằng cách truyền ánh sáng cận hồng ngoại vô hình và đo "thời gian phản hồi lại" của nó sau khi phản xạ khỏi các vật thể.

## 3 Mô tả ý tưởng

### 3.1 Ý tưởng hệ thống



Hình 4: Sơ đồ cấu trúc hệ thống

Hệ thống được thiết kế sẽ có mô hình cấu trúc như sau:

- Robot controllers: Đây là các bộ điều khiển robot, chịu trách nhiệm điều khiển các chuyển động của robot.
- Gripper controller: Bộ điều khiển kẹp, chịu trách nhiệm điều khiển hoạt động của kẹp robot.
- Camera: Camera được sử dụng để thu thập thông tin về môi trường xung quanh robot.



- Gazebo: Gazebo là một môi trường mô phỏng vật lý 3D, được sử dụng để mô phỏng hoạt động của robot.
- Vision: là thành phần chịu trách nhiệm xử lý và phân tích hình ảnh từ camera.
- Trajectory controller: Bộ điều khiển quỹ đạo, chịu trách nhiệm tạo ra các quỹ đạo chuyển động cho robot.
- Joint states: là trạng thái của các khớp robot.
- Robot state publisher: Robot state publisher là thành phần chịu trách nhiệm xuất trạng thái của robot ra ngoài.

### Chức năng của các thành phần

- Robot controllers: Điều khiển các chuyển động của robot, bao gồm chuyển động của các khớp, chuyển động của kẹp, và chuyển động của toàn bộ robot.
- Gripper controller: Điều khiển hoạt động của kẹp robot, bao gồm mở kẹp, đóng kẹp, và di chuyển kẹp.
- Camera: Thu thập thông tin về môi trường xung quanh robot, bao gồm hình ảnh 2D, hình ảnh 3D, và thông tin về độ sâu.
- Gazebo: Mô phỏng hoạt động của robot trong môi trường vật lý 3D.
- Vision: Xử lý và phân tích hình ảnh từ camera, để nhận dạng các đối tượng trong môi trường, xác định vị trí của các đối tượng, và phân loại các đối tượng.
- Trajectory controller: Tạo ra các quỹ đạo chuyển động cho robot, để robot có thể di chuyển đến các vị trí mong muốn một cách an toàn và hiệu quả.
- Joint states: Xuất trạng thái của các khớp robot ra ngoài, để các thành phần khác trong hệ thống có thể truy cập và sử dụng.
- Robot state publisher: Xuất trạng thái của robot ra ngoài, để các thành phần khác trong hệ thống có thể truy cập và sử dụng.

### Tương tác giữa các bộ phận

- Robot controllers: Tương tác với nhau để phối hợp các chuyển động của robot.
- Gripper controller: Tương tác với robot controllers để nhận các lệnh điều khiển hoạt động của kẹp robot.
- Camera: Tương tác với vision để cung cấp dữ liệu hình ảnh.
- Gazebo: Tương tác với robot controllers để nhận các lệnh điều khiển chuyển động của robot, và tương tác với camera để cung cấp dữ liệu về môi trường xung quanh robot.
- Vision: Tương tác với robot controllers để cung cấp thông tin về môi trường xung quanh robot.
- Trajectory controller: Tương tác với robot controllers để cung cấp các quỹ đạo chuyển động cho robot.

- Joint states: Tương tác với robot controllers, vision, và robot state publisher để cung cấp trạng thái của các khớp robot.
- Robot state publisher: Tương tác với các thành phần khác trong hệ thống để xuất trạng thái của robot ra ngoài.

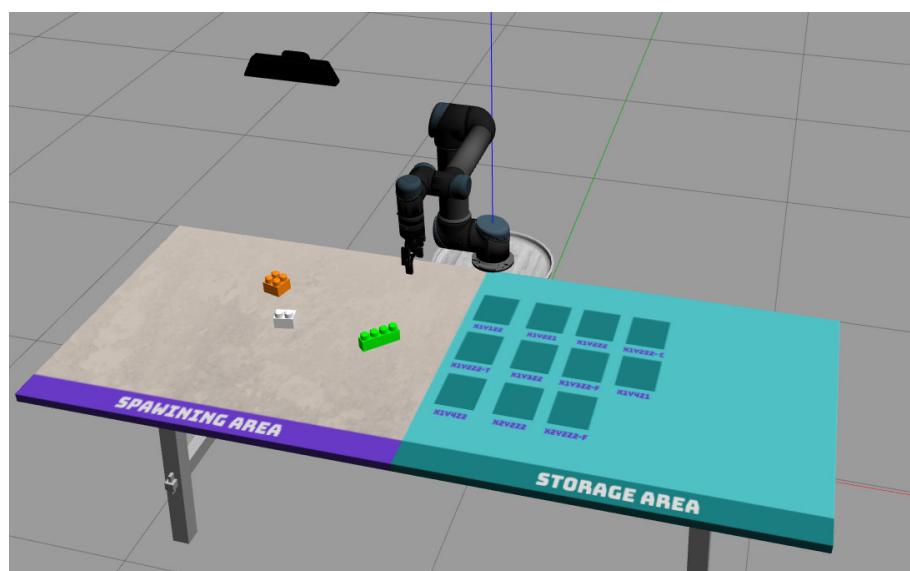
Khi robot cần di chuyển đến một vị trí mới, robot controllers sẽ gửi lệnh điều khiển chuyển động đến trajectory controller. Trajectory controller sẽ tính toán và tạo ra một quỹ đạo chuyển động cho robot. Quỹ đạo chuyển động này sẽ được gửi đến robot controllers, và robot controllers sẽ điều khiển các chuyển động của robot theo quỹ đạo đã tính toán. Trong quá trình di chuyển, camera sẽ thu thập thông tin về môi trường xung quanh robot. Vision sẽ xử lý và phân tích thông tin từ camera, để nhận dạng các đối tượng trong môi trường, xác định vị trí của các đối tượng, và phân loại các đối tượng. Thông tin này sẽ được gửi đến robot controllers, để robot controllers có thể điều chỉnh quỹ đạo chuyển động của robot cho phù hợp với môi trường xung quanh.

### 3.2 Ý tưởng mô phỏng

Trong bài này, nhóm sẽ xây dựng mô phỏng trong **Gazebo** như đã nói ở trên.

Trong thế giới mô phỏng sẽ gồm có 2 bàn dài, một bàn để đặt vật thể, cánh tay UR5 và camera ở phía trên; trong khi bàn còn lại là nơi cánh tay sẽ phân loại và gấp vật thể vào từng ô chỉ định.

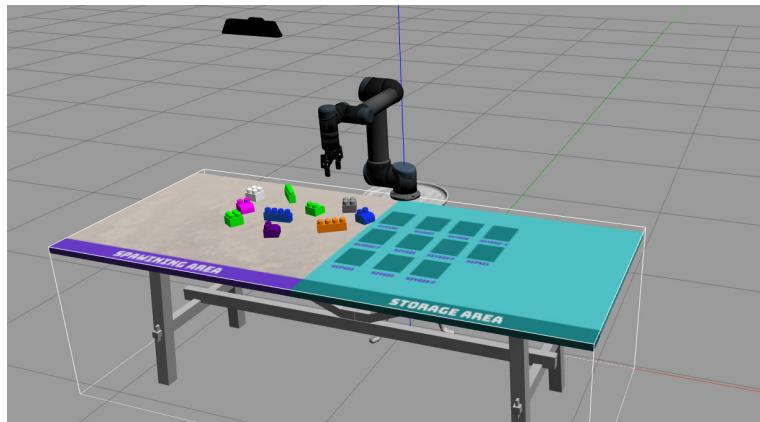
#### 3.2.1 Trường hợp mỗi loại 1 vật thể



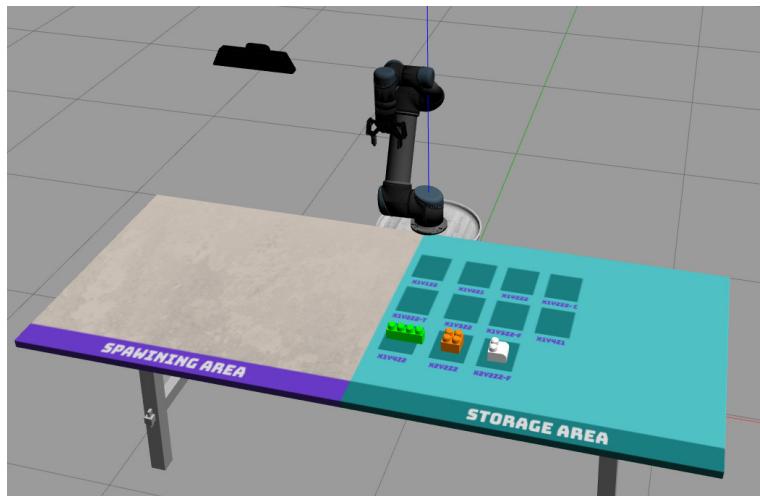
Hình 5: Thế giới mô phỏng trong Gazebo

#### 3.2.2 Trường hợp mỗi loại có nhiều vật thể

### 3.3 Kết quả mong đợi



Hình 6: Mô phỏng nhiều vật thể hơn



Hình 7: Kết quả mong đợi cho trường hợp mỗi loại 1 vật thể

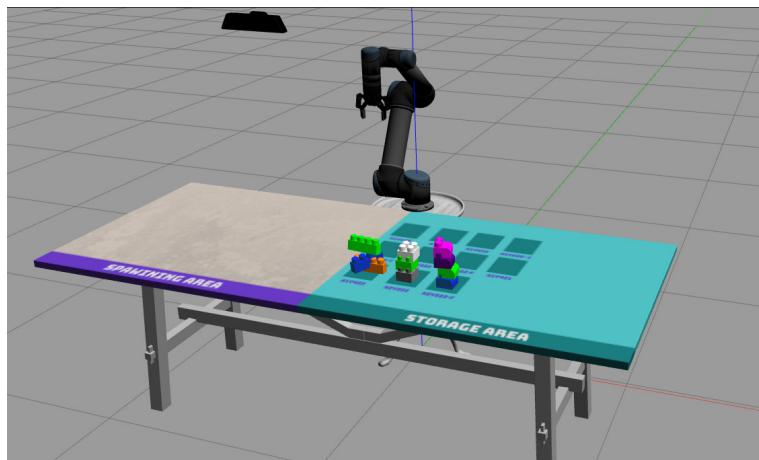
#### 4 Spawn vật thể ngẫu nhiên

Để spawn ngẫu nhiên vật thể trong môi trường mô phỏng Gazebo ta chạy lệnh sau:

```
rosrun levelManager levelManager_modified.py -level [x]
```

Trong đó, [x] là một số nguyên từ 1 đến 3, tương ứng với các cấp độ sau:

- Cấp độ 1: Spawn một khối Lego ngẫu nhiên.
- Cấp độ 2: Spawn ba khối Lego khác nhau: X1-Y4-Z2, X2-Y2-Z2, và X2-Y2-Z2-FILLET.
- Cấp độ 3: Spawn 10 khối Lego ngẫu nhiên.



Hình 8: Kết quả mong đợi cho trường hợp mỗi loại có nhiều vật thể

## 5 Nhận diện vật thể

### 5.1 Phân loại vật thể

Hệ thống phân loại vật thể: Hệ thống phân loại vật thể sử dụng mô hình phân loại để phân loại vật thể trong hình ảnh hoặc video. Hệ thống này được sử dụng trong các ứng dụng như robot tự hành, trợ lý ảo, hoặc quản lý kho hàng.

Trong thế giới thực, giả sử tình huống một công ty cần phân loại các sản phẩm khác nhau bằng camera và mang chúng đến đúng ô phân loại bằng cánh tay robot thì ta cần xây dựng một Mô hình học máy phân loại để phân loại các đối tượng thành các nhóm khác nhau. Mô hình phân loại sẽ học các đặc điểm chung của các sản phẩm trong mỗi nhóm. Sau khi được huấn luyện, mô hình có thể sử dụng các đặc điểm này để phân loại các sản phẩm.

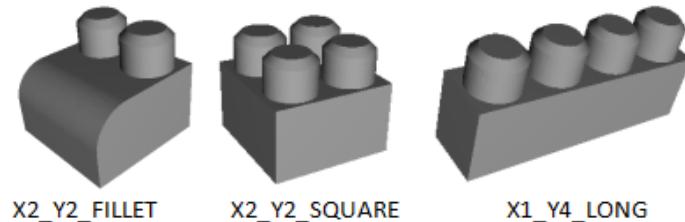
Trong trường hợp công ty cần phân loại các sản phẩm thành các nhóm khác nhau theo kích thước, hình dáng và loại sản phẩm, mô hình phân loại cần học các đặc điểm sau của sản phẩm:

- Kích thước sản phẩm: to, vừa, nhỏ,... Ví dụ, để phân loại các sản phẩm theo kích thước, mô hình phân loại có thể học các đặc điểm sau:
  - Chiều dài của sản phẩm: Dài, ngắn, vừa,...
  - Chiều rộng của sản phẩm: Rộng, hẹp, vừa,...
  - Chiều cao của sản phẩm: Cao, thấp, vừa,...
- Hình dạng sản phẩm: hình chữ nhật, hình tròn, hình tam giác,... Tương tự, để phân loại các sản phẩm theo hình dạng, mô hình phân loại có thể học các đặc điểm sau:
  - Số cạnh của sản phẩm: Có cạnh, không cạnh,...
  - Độ tròn của sản phẩm: Tròn, méo,...
  - Sự đối xứng của sản phẩm: Đối xứng, không đối xứng,...

- Loại sản phẩm: Quần áo, giày dép, đồ điện tử,... Để phân loại các sản phẩm theo loại sản phẩm, mô hình phân loại có thể học các đặc điểm sau:
  - Màu sắc của sản phẩm: Đỏ, xanh, vàng,...
  - Sự hiện diện của các đặc điểm đặc trưng: Có nhãn hiệu, có logo,...

Sau khi được huấn luyện, mô hình có thể sử dụng các đặc điểm này để phân loại các sản phẩm. Ví dụ, nếu mô hình nhìn thấy một sản phẩm có chiều dài ngắn, chiều rộng hẹp, và chiều cao vừa, mô hình có thể dự đoán rằng sản phẩm đó là nhỏ, nếu mô hình nhìn thấy một sản phẩm có số cạnh 4, độ tròn cao, và sự đối xứng cao, mô hình có thể dự đoán rằng sản phẩm đó là hình vuông,... Đặc biệt hơn mô hình có thể sử dụng các đặc điểm này để phân loại các sản phẩm mới. Hoặc công ty muốn phân loại một vài đối tượng cụ thể thì chỉ cần thu thập dữ liệu, gắn nhãn và huấn luyện mô hình chỉ liên quan đến các đối tượng đó.

Trong phạm vi đề tài, nhóm tiến hành phân loại các mảnh lego có kích thước, hình dáng khác nhau như sau:



Hình 9: Các mảnh lego dùng để phân loại

### 5.1.1 Chuẩn bị dữ liệu

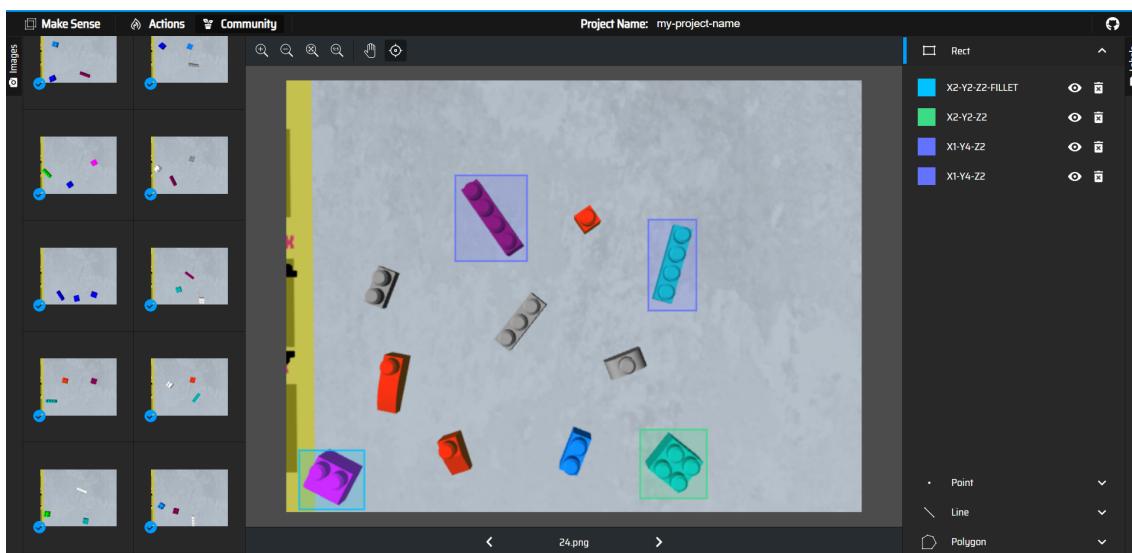
Bước đầu tiên là chuẩn bị dữ liệu để huấn luyện mô hình phân loại. Dữ liệu cần bao gồm các hình ảnh của các mảnh lego cần phân loại, cùng với nhãn phân loại tương ứng của các hình ảnh đó. Để gán nhãn cho các hình ảnh đã thu thập ta sử dụng công cụ **makesense.ai** như sau:

Sau khi đã gán nhãn phân loại cho tất cả, ta Export về máy và thu được danh sách tập tin nhãn dán và định dạng của mỗi file như sau:

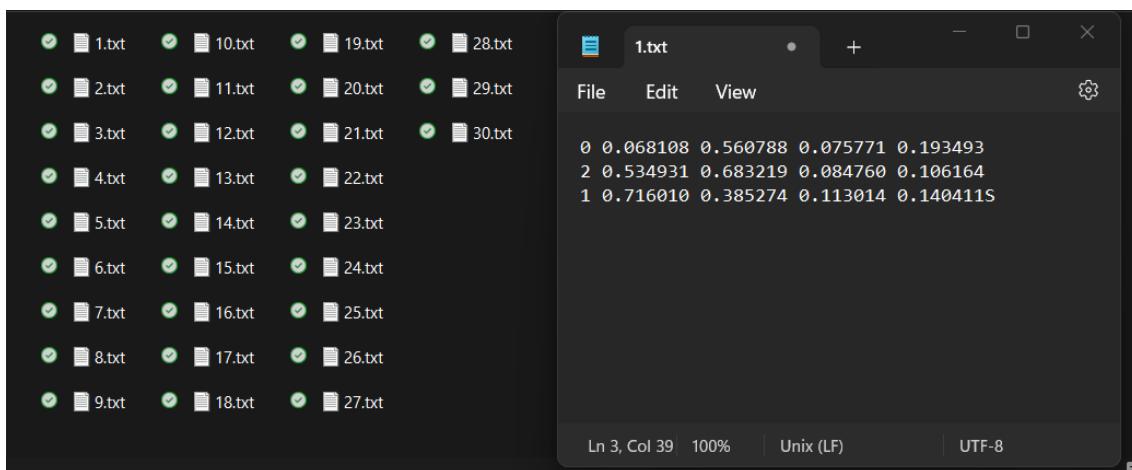
### 5.1.2 Lựa chọn mô hình

Lựa chọn mô hình phân loại vật thể phù hợp phụ thuộc vào nhiều yếu tố, bao gồm:

- **Độ phức tạp của nhiệm vụ phân loại:** Nếu nhiệm vụ phân loại đơn giản, chẳng hạn như phân loại giữa hai loại vật thể, thì các mô hình dựa trên đặc trưng có thể là đủ. Tuy nhiên, nếu nhiệm vụ phân loại phức tạp, chẳng hạn như phân loại nhiều loại vật thể trong một bối cảnh phức tạp, thì các mô hình học sâu có thể cần thiết.



Hình 10: Gán nhãn bằng MakeSense.AI



Hình 11: Định dạng file lable YOLOv5

- Kích thước của tập dữ liệu:** Nếu tập dữ liệu lớn, thì các mô hình học sâu có thể học các đặc trưng tốt hơn các mô hình dựa trên đặc trưng. Tuy nhiên, nếu tập dữ liệu nhỏ, thì các mô hình dựa trên đặc trưng có thể ít bị quá khớp hơn các mô hình học sâu.
- Thời gian và tài nguyên tính toán:** Các mô hình học sâu thường yêu cầu nhiều thời gian và tài nguyên tính toán hơn các mô hình dựa trên đặc trưng.

So sánh các mô hình huấn luyện:

Mô hình	Tính hiệu quả
YOLOV5	Hiệu quả cao, tốc độ nhanh, dễ sử dụng
SSD	Hiệu quả cao, tốc độ nhanh, áp dụng cho nhiều loại vật thể khác nhau
Faster R-CNN	Hiệu quả cao, phát hiện và phân loại các vật thể nhỏ và khó phát hiện
Mask R-CNN	Hiệu quả cao, phát hiện và phân loại các vật thể có hình dạng phức tạp

- YOLOv5 là một mô hình học sâu dựa trên CNN được thiết kế đặc biệt để



phát hiện và phân loại các vật thể trong hình ảnh. YOLOv5 đã được chứng minh là hiệu quả trong việc phát hiện và phân loại nhiều loại vật thể khác nhau, bao gồm cả các vật thể có kích thước và hình dạng tương tự nhau.

- **SSD** là một mô hình học sâu dựa trên CNN khác cũng có thể được sử dụng để phát hiện và phân loại các vật thể trong hình ảnh. SSD có hiệu quả cao và tốc độ nhanh, có thể áp dụng cho nhiều loại vật thể khác nhau.
- **Faster R-CNN** là một mô hình học sâu dựa trên CNN phức tạp hơn có thể được sử dụng để phát hiện và phân loại các vật thể nhỏ và khó phát hiện. Faster R-CNN có hiệu quả cao hơn YOLOv5 và SSD trong việc phát hiện các vật thể nhỏ, nhưng tốc độ chậm hơn.
- **Mask R-CNN** là một mô hình học sâu dựa trên CNN phức tạp nhất trong số các mô hình được liệt kê ở đây. Mask R-CNN có hiệu quả cao nhất trong việc phát hiện và phân loại các vật thể có hình dạng phức tạp. Tuy nhiên, Mask R-CNN cũng có tốc độ chậm nhất.

Trong trường hợp phân loại 3 loại vật thể lego khác nhau, YOLOv5 là một lựa chọn tốt. Một số lý do chính bao gồm:

- YOLOv5 là một mô hình mạnh mẽ và hiệu quả. Mô hình YOLOv5 đã được chứng minh là có thể đạt được độ chính xác cao trong các nhiệm vụ phân loại hình ảnh.
- YOLOv5 là một mô hình nhanh chóng. Mô hình YOLOv5 có thể xử lý hình ảnh với tốc độ nhanh, khiến nó trở nên phù hợp cho các ứng dụng thực tế đòi hỏi thời gian thực.
- YOLOv5 là một mô hình linh hoạt. Mô hình YOLOv5 có thể được sử dụng để phân loại các đối tượng khác nhau, với các kích thước và độ phức tạp khác nhau.

### 5.1.3 Huấn luyện mô hình

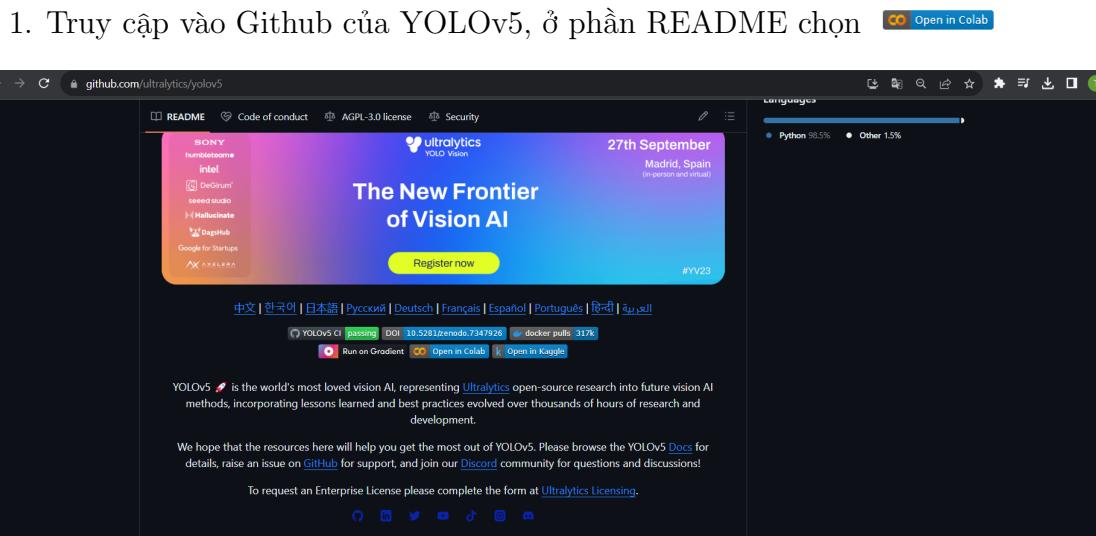
Ta có thể huấn luyện mô hình ở những nơi sau:

- Máy tính cá nhân hoặc máy chủ tại chỗ. Nếu có máy tính cá nhân hoặc máy chủ tại chỗ với cấu hình mạnh mẽ, ta có thể tự cài đặt và huấn luyện mô hình trên máy của mình.
- Các dịch vụ huấn luyện đám mây. Có nhiều dịch vụ huấn luyện đám mây cung cấp khả năng truy cập vào GPU và các tài nguyên khác cần thiết để huấn luyện mô hình. Một số dịch vụ huấn luyện đám mây phổ biến bao gồm:
  - Amazon Web Services (AWS)
  - Microsoft Azure
  - Google Cloud Platform (GCP)
  - Google Colab
- Các trung tâm dữ liệu chuyên dụng. Nếu có nhu cầu huấn luyện mô hình với quy mô lớn, ta có thể thuê các trung tâm dữ liệu chuyên dụng. Các trung tâm dữ liệu này cung cấp khả năng truy cập vào các tài nguyên mạnh mẽ và đáng tin cậy.



Mỗi lựa chọn có những ưu và nhược điểm riêng. Google Colab là một lựa chọn tuyệt vời cho người mới bắt đầu vì nó cung cấp tài nguyên miễn phí và dễ sử dụng. Tuy nhiên, Google Colab có một số hạn chế, chẳng hạn như thời gian sử dụng GPU giới hạn.

Các bước huấn luyện mô hình bằng Google Colab:



Hình 12: Bắt đầu với Google Colab

## 2. Cài đặt YOLOv5 và các thư viện cần thiết vào Google Colab.

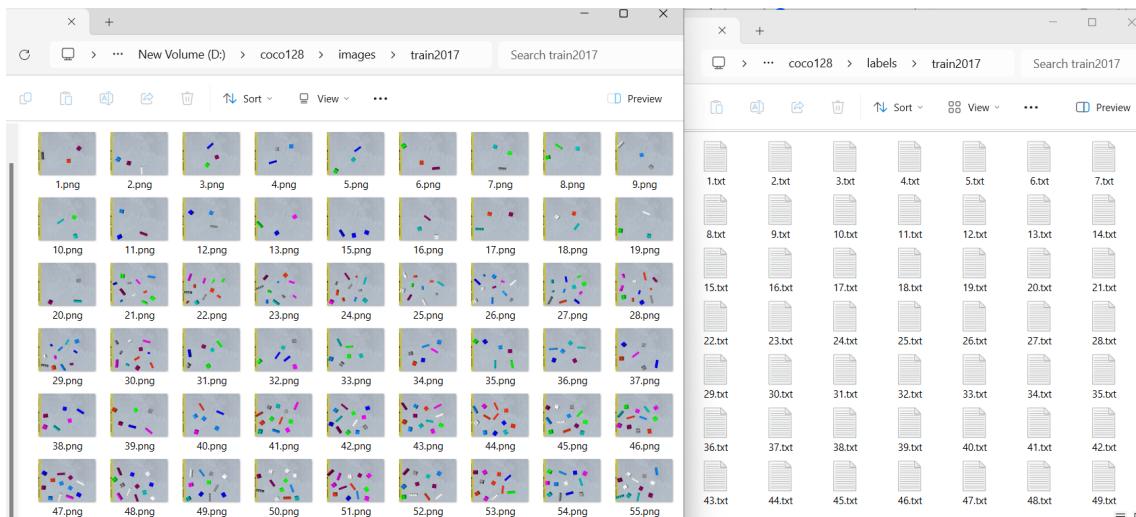
```
[1] !git clone https://github.com/ultralytics/yolov5 # clone
[2] cd yolov5
[3] !pip install -qr requirements.txt comet_ml # install
[4] import torch
[5] import utils
[6] display = utils.notebook_init() # checks

YOLOv5 🚀 v7.0-249-gf400bb Python-3.10.12 torch-2.1.0+cu118 CUDA:0 (Tesla T4, 15102MB)
Setup complete ✅ (2 CPUs, 12.7 GB RAM, 27.1/78.2 GB disk)
```

Hình 13: Cài đặt YOLOv5 trên Google Colab

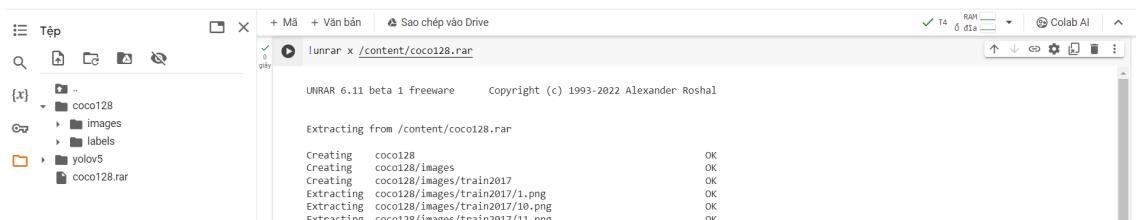
3. Dựa trên thư viện coco128 đã có sẵn đúng định dạng, ta sẽ chỉnh sửa lại cho phù hợp với mô hình và tập dữ liệu của mình để đơn giản cho quá trình huấn luyện.

Tải thư viện coco128 về sau đó thay thế ảnh, file nhãn dán thành các hình ảnh, file nhãn dán đã chuẩn bị ở 5.1.1



Hình 14: Ví dụ về tập dữ liệu cần chuẩn bị

Nén thư mục coco128 để upload lên Google Colab sau đó tiến hành giải nén ra.



Hình 15: Tải tập dữ liệu lên Google Colab

Chỉnh sửa file **coco128.yaml** tại đường dẫn `/content/yolov5/data/coco128.yaml`.

4. Tiến hành huấn luyện mô hình.

```
python train.py --img 416 --batch 16 --epochs 1000 \
--data ../custom_dataset/custom_dataset.yaml \
--cfg ../custom_dataset/custom_model.yaml \
--weights '' --name custom_model --cache
```

Trong đó:

- **img:** kích thước ảnh (độ phân giải)
- **batch:** số ảnh dùng để huấn luyện trong mỗi lượt
- **epochs:** số lượt huấn luyện cho tất cả các ảnh trong tập dữ liệu train



```
coco128.yaml x
1
2 train: /content/coco128 # train images (relative to 'path') 128 images
3 val: /content/coco128 # val images (relative to 'path') 128 images
4 test: # test images (optional)
5
6 # Classes
7 names:
8 0: X1-Y4-Z2
9 1: X2-Y2-Z2
10 2: X2-Y2-Z2-FILLET
11
12
13 # Download script/URL (optional)
14 download: https://ultralytics.com/assets/coco128.zip
15
```

Hình 16: Chỉnh sửa file cấu hình thư viện huấn luyện

- **data:** đường dẫn đến file cấu hình của tập dữ liệu
- **cfg:** đường dẫn đến file cấu hình của mô hình
- **weights:** đường dẫn đến file weight chứa độ liên kết giữa các neuron (để “là để huấn luyện từ đầu”)
- **name:** tên thư mục để lưu mô hình
- **cache:** dùng bộ nhớ đệm để huấn luyện nhanh hơn

### Lần huấn luyện đầu tiên:

```
python /content/yolov5/train.py --img 640 --batch 16 --epochs 3 --data
coco128.yaml --weights yolov5s.pt
```

Khi train YOLOv5 lần đầu với epochs=3, mô hình sẽ chỉ được huấn luyện với một lượng dữ liệu nhỏ. Điều này có thể giúp mô hình tìm hiểu các đặc điểm cơ bản của các đối tượng được phát hiện. Trong trường hợp này, mô hình được khởi tạo với weights yolov5s.pt. Vì YOLOv5s là một mô hình phát hiện vật thể mạnh mẽ, được đào tạo trên một tập dữ liệu lớn. Việc sử dụng trọng số ban đầu của YOLOv5s có thể giúp mô hình học hỏi nhanh hơn và đạt được kết quả tốt hơn.



Kết quả thu được:

```
!python /content/yolov5/train.py --img 640 --epochs 3 --data coco128.yaml --weights yolov5s.pt
④ Image sizes 640 train, 640 val
Using 2 dataloader workers
Logging results to runs/train/exp10
Starting training for 3 epochs...

Epoch GPU_mem box_loss obj_loss cls_loss Instances Size
0/2   3.49G  0.1203  0.07462  0.04154    87   640: 100% [██████████] | 4/4 [00:06<00:00, 1.64s/it]
      Class  Images Instances P       R       mAP50  mAP50-95: 100% [██████████] | 2/2 [00:07<00:00, 3.68s/it]
      all    54     333      0.00728  0.285    0.00613  0.00127

Epoch GPU_mem box_loss obj_loss cls_loss Instances Size
1/2   3.93G  0.1164  0.07684  0.0411    89   640: 100% [██████████] | 4/4 [00:01<00:00, 3.96it/s]
      Class  Images Instances P       R       mAP50  mAP50-95: 100% [██████████] | 2/2 [00:09<00:00, 4.67s/it]
      all    54     333      0.00982  0.403    0.012   0.00249

Epoch GPU_mem box_loss obj_loss cls_loss Instances Size
2/2   3.93G  0.1104  0.07282  0.04074   29   640: 100% [██████████] | 4/4 [00:00<00:00, 4.07it/s]
      Class  Images Instances P       R       mAP50  mAP50-95: 100% [██████████] | 2/2 [00:05<00:00, 2.60s/it]
      all    54     333      0.013    0.577    0.0221  0.00479

3 epochs completed in 0.009 hours.
Optimizer stripped from runs/train/exp10/weights/last.pt, 14.4MB
Optimizer stripped from runs/train/exp10/weights/best.pt, 14.4MB

Validating runs/train/exp10/weights/best.pt...
Fusing layers...
Model summary: 157 layers, 7018216 parameters, 0 gradients, 15.8 GFLOPs
      Class  Images Instances P       R       mAP50  mAP50-95: 100% [██████████] | 2/2 [00:02<00:00, 1.36s/it]
      all    54     333      0.013    0.574    0.0221  0.00477
      X1-Y4-Z2 54     120      0.0199   0.725    0.0449  0.00983
      X2-V2-Z2 54     109      0.00717   0.477    0.00943  0.00189
      X2-Y2-Z2-FILLET 54     104      0.0118   0.519    0.0121  0.00258
Results saved to runs/train/exp10
```

Hình 17: Kết quả lần huấn luyện đầu tiên

### Lần huấn luyện thứ hai:

```
!python /content/yolov5/train.py --img 640 --batch 16 --epochs 128 --data
coco128.yaml --weights /content/yolov5/runs/train/exp2/weights/last.pt --cache
```

Ta lấy kết quả huấn luyện cuối cùng *last.pt* ở lần huấn luyện đầu tiên huấn luyện tiếp với epoch lớn, mô hình sẽ được huấn luyện với lượng dữ liệu lớn hơn. Điều này có thể giúp mô hình cải thiện độ chính xác trong việc phát hiện các đối tượng, đặc biệt là các đối tượng có kích thước nhỏ hoặc xuất hiện trong các điều kiện ánh sáng phức tạp.

Nếu mô hình huấn luyện thành công ta sẽ lấy file *best.pt* của lần huấn luyện này về sử dụng.

Kết quả thu được:

```
!python /content/yolov5/train.py --img 640 --epochs 128 --data coco128.yaml --weights /content/yolov5/runs/train/exp10/weights/last.pt
```

Epoch	GPU_mem	box_loss	obj_loss	cls_loss	Instances	P	R	mAP50	mAP50-95: 100%	Size
123/127	3.93G	0.02238	0.03683	0.006179	79	640: 100% [██████████]	4/4 [00:02<00:00, 1.79it/s]			
	Class	Images	Instances			P	R	mAP50	mAP50-95: 100%	2/2 [00:01<00:00, 1.64it/s]
	all	54	333	0.993	0.994	0.995	0.797			
124/127	3.93G	0.02006	0.03626	0.0063	70	640: 100% [██████████]	4/4 [00:01<00:00, 2.86it/s]			
	Class	Images	Instances			P	R	mAP50	mAP50-95: 100%	2/2 [00:01<00:00, 1.86it/s]
	all	54	333	0.992	0.994	0.995	0.797			
125/127	3.93G	0.02013	0.03698	0.006723	73	640: 100% [██████████]	4/4 [00:01<00:00, 2.76it/s]			
	Class	Images	Instances			P	R	mAP50	mAP50-95: 100%	2/2 [00:01<00:00, 1.50it/s]
	all	54	333	0.992	0.994	0.995	0.799			
126/127	3.93G	0.0207	0.03513	0.006394	76	640: 100% [██████████]	4/4 [00:01<00:00, 2.21it/s]			
	Class	Images	Instances			P	R	mAP50	mAP50-95: 100%	2/2 [00:01<00:00, 1.01it/s]
	all	54	333	0.992	0.996	0.995	0.794			
127/127	3.93G	0.02205	0.04087	0.006877	95	640: 100% [██████████]	4/4 [00:01<00:00, 2.88it/s]			
	Class	Images	Instances			P	R	mAP50	mAP50-95: 100%	2/2 [00:01<00:00, 1.68it/s]
	all	54	333	0.992	0.997	0.995	0.798			

128 epochs completed in 0.144 hours.  
Optimizer stripped from runs/train/exp11/weights/last.pt, 14.4MB  
Optimizer stripped from runs/train/exp11/weights/best.pt, 14.4MB

Validating runs/train/exp11/weights/best.pt...  
Fusing layers...  
Model summary: 157 layers, 7018216 parameters, 0 gradients, 15.8 GFLOPS

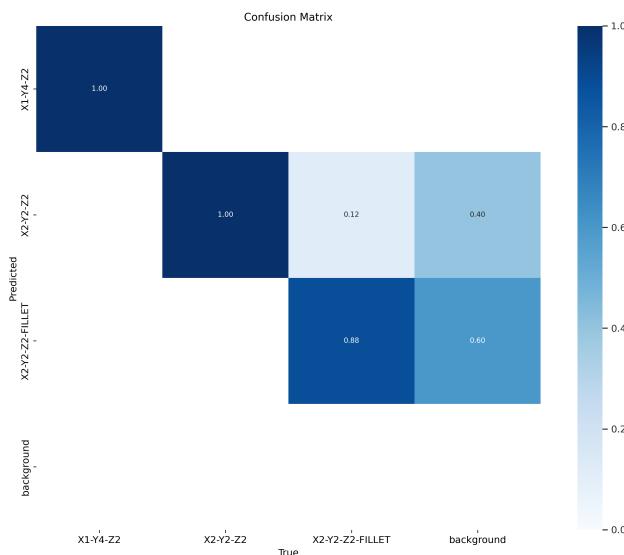
Class	Images	Instances	P	R	mAP50	mAP50-95: 100%	Size
all	54	333	0.995	0.999	0.995	0.801	
X1-Y4-Z2	54	120	0.998	1	0.995	0.814	
X2-Y2-Z2	54	109	0.986	1	0.995	0.803	
X2-Y2-Z2-FILLET	54	104	1	0.998	0.995	0.785	

Results saved to runs/train/exp11

Hình 18: Kết quả lần huấn luyện thứ hai

#### 5.1.4 Đánh giá kết quả huấn luyện mô hình

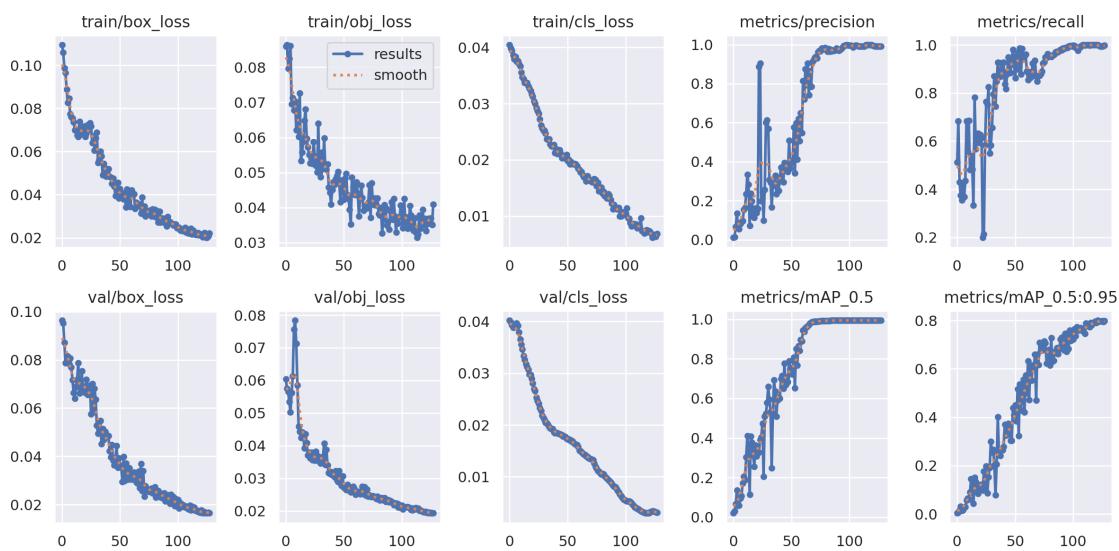
Tỷ lệ nhầm lẫn giữa các vật: Ma trận nhầm lẫn cung cấp thông tin chi tiết hơn



Hình 19: Tỷ lệ nhầm lẫn giữa các vật

về các lỗi mà mô hình đã mắc phải. Ma trận này cho thấy rằng mô hình thường nhầm lẫn các đối tượng X2-Y2-Z2-FILLET và X1-Y4-Z2.

Các biểu đồ:



Hình 20: Biểu đồ kết quả huấn luyện

Ý nghĩa các biểu đồ

- Biểu đồ "train box losS" và "val box losS" thể hiện sự giảm dần của hàm mất mát hộp khi mô hình được huấn luyện và đánh giá trên tập dữ liệu huấn luyện và đánh giá. Hàm mất mát hộp là một phép đo lỗi trong việc xác định vị trí của các đối tượng trong ảnh. Sự giảm dần của hàm mất mát này cho thấy mô hình đang học cách xác định vị trí của các đối tượng ngày càng chính xác.
- Biểu đồ "train/obj\_loss" và "val/obj\_loss" thể hiện sự giảm dần của hàm mất mát đối tượng khi mô hình được huấn luyện và đánh giá trên tập dữ liệu huấn luyện và đánh giá. Hàm mất mát đối tượng là một phép đo lỗi trong việc xác định loại đối tượng trong ảnh. Sự giảm dần của hàm mất mát này cho thấy mô hình đang học cách xác định loại đối tượng ngày càng chính xác.
- Biểu đồ "train/cis loss" và "val/cis loss" thể hiện sự giảm dần của hàm mất mát kích thước khi mô hình được huấn luyện và đánh giá trên tập dữ liệu huấn luyện và đánh giá. Hàm mất mát kích thước là một phép đo lỗi trong việc xác định kích thước của các đối tượng trong ảnh. Sự giảm dần của hàm mất mát này cho thấy mô hình đang học cách xác định kích thước của các đối tượng ngày càng chính xác.
- Biểu đồ "metrics/precision" và "metrics/recall" thể hiện độ chính xác và độ nhạy của mô hình khi được đánh giá trên tập dữ liệu đánh giá. Độ chính xác là tỷ lệ các đối tượng được mô hình dự đoán đúng. Độ nhạy là tỷ lệ các đối tượng thực sự là đối tượng được mô hình dự đoán đúng. Cả hai chỉ số này đều cao cho thấy mô hình có độ chính xác và độ tin cậy cao.

**Về độ chính xác:** Theo biểu đồ "metrics/precision", độ chính xác của mô hình tăng từ khoảng 0,75 lên khoảng 0,95 sau khi được huấn luyện 100 epoch. Điều này cho thấy mô hình học cách dự đoán các đối tượng ngày càng chính xác.

**Về độ nhạy:** Theo biểu đồ "metrics/recall", độ nhạy của mô hình tăng từ khoảng 0,8 lên khoảng 0,9 sau khi được huấn luyện 100 epoch. Điều này cho thấy mô hình học cách xác định các đối tượng thực sự là đối tượng ngày càng chính xác.

**Về hàm mất mát** Cả hai hàm mất mát hộp và mất mát đối tượng đều giảm dần sau khi được huấn luyện 100 epoch. Điều này cho thấy mô hình đang học cách xác định vị trí và loại đối tượng ngày càng chính xác.

Kiểm tra thử bằng các hình không có trong tập dữ liệu huấn luyện:



Hình 21: Kết quả kiểm thử mô hình phân loại sau khi huấn luyện

### Kết luận

Nhìn chung, kết quả sau khi huấn luyện mô hình cho thấy mô hình có độ chính xác và độ tin cậy cao. Mô hình có thể được sử dụng để phát hiện và xác định các đối tượng trong ảnh với độ chính xác cao. Tuy nhiên, vẫn có một số điểm cần lưu ý, cụ thể như sau:

- Độ chính xác của mô hình trên tập dữ liệu huấn luyện (train) cao hơn đáng kể so với tập dữ liệu thử nghiệm (val). Điều này có thể là do mô hình đã bị quá khớp (overfitting) với tập dữ liệu huấn luyện.
- Độ chính xác của mô hình trong việc dự đoán các đối tượng nhỏ (box\_loss) thấp hơn so với các đối tượng lớn. Điều này có thể là do mô hình gặp khó khăn trong việc xác định vị trí và kích thước của các đối tượng nhỏ.



## 5.2 Hiện thực

Để thực hiện chức năng phân loại và xác định vị trí của vật thể được mô phỏng bằng Gazebo sau khi đã có môi trường, mô hình cánh tay robot và spawn các mảnh lego thực hiện ở 3.3 ta chạy lệnh sau vào terminal:

```
rosrun vision lego-vision.py -show
```

Lệnh này sẽ khởi chạy node ROS có tên "vision" và chạy tập lệnh "lego-vision.py". Tùy chọn "-show" sẽ hiển thị hình ảnh đã xử lý với hộp giới hạn và nhãn văn bản.

### 5.2.1 Luồng thực thi:

1. Chương trình khởi tạo nút ROS, đăng ký vào chủ đề camera và tạo người đăng ký.
2. Hàm start\_node đợi hình ảnh RGB và độ sâu được đồng bộ hóa.
3. Khi hình ảnh có sẵn, hàm process\_CB được kích hoạt.
4. Hình ảnh được chuyển đổi sang định dạng OpenCV và được truyền cho hàm process\_image. process\_image thực hiện tất cả các xử lý hình ảnh, phát hiện vật thể và ước tính tư thế.
5. Mỗi vật thể được phát hiện được phân tích bởi process\_item trích xuất thông tin cụ thể.
6. Kết quả cuối cùng được xuất bản dưới dạng tin nhắn ModelStates chứa thông tin tư thế cho mỗi viên gạch Lego.
7. Hiển thị hình ảnh đã xử lý với hộp giới hạn và nhãn văn bản để hiển thị.

### 5.2.2 Các hàm chính

Các hàm chính trong file lego-vision.py:

- **start\_node()**: đăng ký nhận ảnh camera, chuẩn bị gửi kết quả và bắt đầu quá trình xử lý hình ảnh, đồng thời đảm bảo hệ thống luôn hoạt động.

```
def start_node():
    global pub

    print("Starting Node Vision 1.0")

    rospy.init_node('vision')

    print("Subscribing to camera images")
    #topics subscription
    rgb = message_filters.Subscriber("/camera/color/image_raw", Image)
    depth = message_filters.Subscriber("/camera/depth/image_raw", Image)

    #publisher results
    pub=rospy.Publisher("lego_detections", ModelStates, queue_size=1)

    print("Localization is starting.. ")
    print("(Waiting for images..)", end='\r'), print(end='\033[K')
```



```
#images synchronization
syncro = message_filters.TimeSynchronizer([rgb, depth], 1, reset=True)
syncro.registerCallback(process_CB)

#keep node always alive
rospy.spin()
pass
```

- Đăng ký subscriber theo dõi topic `/camera/color/image_raw`, nhận dữ liệu ảnh màu.
- Đăng ký subscriber theo dõi topic `/camera/depth/image_raw`, nhận dữ liệu ảnh chiều sâu.
- Tạo publisher với tên "lego\_detections" để gửi dữ liệu kiểu ModelStates, với queue size là 1.
- Tạo bộ xử lý đồng bộ syncro để đảm bảo dữ liệu ảnh màu và chiều sâu được nhận cùng lúc thông qua hàm process\_CB.

- **load\_models()**: Tải các mô hình YOLOv5 để phân loại.

```
def load_models():
    global model, model_orientation

    #yolo model and weights classification
    print("Loading model best.pt")
    weight = path.join(path_weights, 'best.pt')
    model = torch.hub.load(path_yolo, 'custom', path=weight, source='local')

    #yolo model and weights orientation
    print("Loading model depth.pt")
    weight = path.join(path_weights, 'depth.pt')
    model_orientation = torch.hub.load(path_yolo, 'custom', path=weight,
                                         source='local')
    pass
```

- Mô hình `best.pt` sẽ giúp phân loại các đối tượng trong ảnh và lưu trong `model`.
- Mô hình `depth.pt` sẽ giúp phân loại và xác định hướng các đối tượng trong ảnh bằng dữ liệu chiều sâu của ảnh và lưu trong `model_orientation`. Kết hợp mô hình này giúp cải thiện dự đoán chính xác hơn dựa vào các đặc trưng theo trực. Sau khi thu thập dữ liệu depth image của vật thể ta tiến hành các bước huấn luyện và đánh giá mô hình tương tự như mô hình `best.pt`.

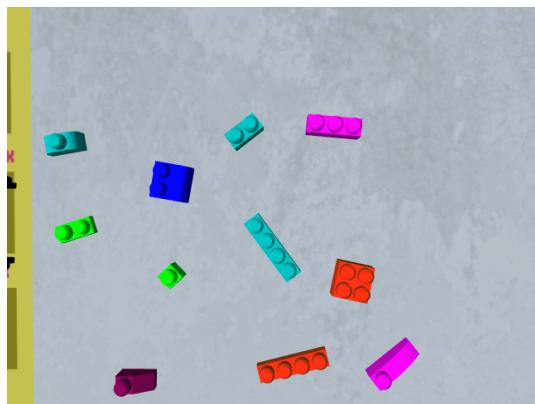
- **process\_CB(image\_rgb, image\_depth)**: Hàm gọi lại được kích hoạt khi cả hai hình ảnh camera đều có sẵn. Chuyển đổi ROS image messages sang định dạng OpenCV. Gọi hàm `process_image` để xử lý thêm.

```
def process_CB(image_rgb, image_depth):
    t_start = time.time()
    #from standard message image to opencv image
    rgb = CvBridge().imgmsg_to_cv2(image_rgb, "bgr8")
    depth = CvBridge().imgmsg_to_cv2(image_depth, "32FC1")
```

```
process_image(rgb, depth)

print("Time:", time.time() - t_start)
rospy.signal_shutdown(0)
pass
```

- "bgr8" có nghĩa là "Blue-Green-Red, 8 bit trên kênh". Điều này có nghĩa là hình ảnh được lưu trữ trong ba kênh (B, G, R) với mỗi kênh có độ chính xác 8 bit (256 giá trị có thể có).



Hình 22: Dữ liệu ảnh màu chuyển từ ROS message sang OpenCV

- "32FC1" có nghĩa là "32 bit dấu phẩy động, 1 kênh". Điều này có nghĩa là hình ảnh được lưu trữ dưới dạng hình ảnh một kênh với mỗi giá trị pixel được biểu thị dưới dạng số dấu phẩy động 32 bit, cung cấp độ chính xác cao cho thông tin độ sâu.



Hình 23: Dữ liệu ảnh chiều sâu chuyển từ ROS message sang OpenCV

- **process\_image(rgb, depth):** Thực hiện các tác vụ xử lý hình ảnh và định vị chính.

```
def process_image(rgb, depth):
```

```
img_draw = rgb.copy()
hsv = cv.cvtColor(rgb, cv.COLOR_BGR2HSV)

get_dist_tavolo(depth)
get_origin(rgb)

#results collecting localization

model.conf = 0.6      #confidence value of classification

results = model(rgb)
pandino = results.pandas().xyxy[0].to_dict(orient="records")

# ----
if depth is not None:
    imgs = (rgb, hsv, depth, img_draw)
    results = [process_item(imgs, item) for item in pandino]

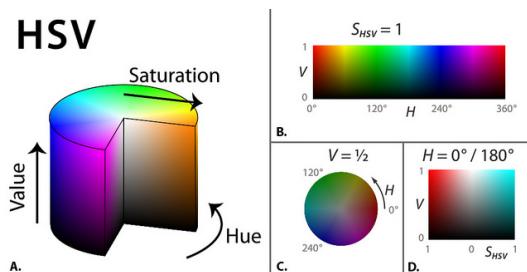
# ----

msg = ModelStates()
for point in results:
    if point is not None:
        msg.name.append(point.name)
        msg.pose.append(point.pose)
pub.publish(msg)

if a_show:
    cv.imshow("vision-results.png", img_draw)
    cv.waitKey()

pass
```

- **COLOR\_BGR2HSV** là một mã định dạng được sử dụng trong thư viện OpenCV để chuyển đổi một ảnh từ không gian màu RGB sang không gian màu HSV. Trong không gian màu HSV, mỗi pixel được biểu diễn bằng ba thành phần:

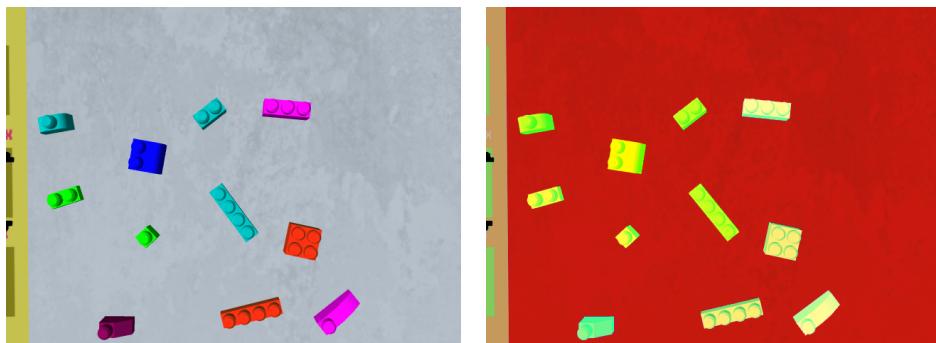


Hình 24: Các thành phần trong không gian màu HSV

- \* **H:** (Hue) Vùng màu
- \* **S:** (Saturation) Độ bão hòa màu
- \* **V (hay B):** (Value hay Bright) Độ sáng

Ví dụ, một pixel có giá trị RGB là (255, 0, 0) sẽ có giá trị HSV là (0, 255, 255). Pixel này có màu đỏ thuần túy, với độ bão hòa và độ sáng tối đa.

Ví dụ chuyển đổi một ảnh từ không gian màu RGB sang không gian màu HSV:



Hình 25: Ví dụ chuyển đổi một ảnh từ không gian màu RGB sang không gian màu HSV

- **get\_dist\_tavolo(depth):** lấy giá trị depth lớn nhất trong toàn bộ ảnh.
  - **get\_origin(img):** tìm điểm gốc (giữa ảnh) của ảnh RGB.
  - Thiết lập ngưỡng độ tin cậy (confidence) cho mô hình phân loại vật thể là 0.6.
  - Nếu có ảnh depth (chiều sâu) thì duyệt qua từng thông tin vật thể và gọi hàm riêng process\_item để xử lý chi tiết, trả về một danh sách các kết quả. Nếu kết quả hợp lệ (không phải None), thêm tên và pose (vị trí và hướng) của vật thể vào msg và publish đến topic đã tạo ở start\_node() dưới dạng tin nhắn ModelStates.
- **process\_item(imgs, item):** Xử lý một vật thể được phát hiện (được xác định bởi YOLOv5) trong hình ảnh. Trích xuất thông tin cụ thể như màu sắc, chiều cao và trực của viên gạch Lego. Vẽ hộp giới hạn và nhãn văn bản để hiển thị. Tính toán tư thế cuối cùng của viên gạch và thêm nó vào tin nhắn kết quả.

```
def process_item(imgs, item):  
  
    #images  
    rgb, hsv, depth, img_draw = imgs  
    #obtaining Yolo informations (class, coordinates, center)  
    x1, y1, x2, y2, cn, cl, nm = item.values()  
    mar = 15  
    x1, y1 = max(mar, x1), max(mar, y1)  
    x2, y2 = min(rgb.shape[1]-mar, x2), min(rgb.shape[0]-mar, y2)  
    boxMin = np.array((x1-mar, y1-mar))  
    x1, y1, x2, y2 = np.int0((x1, y1, x2, y2))  
  
    boxCenter = (y2 + y1) // 2, (x2 + x1) // 2  
    color = get_lego_color(boxCenter, rgb)  
    hsvcolor = get_lego_color(boxCenter, hsv)  
  
    sliceBox = slice(y1-mar, y2+mar), slice(x1-mar, x2+mar)  
  
    #crop img with coordinate bounding box; computing all imgs  
    l_rgb = rgb[sliceBox]  
    l_hsv = hsv[sliceBox]  
  
    if a_show: cv.rectangle(img_draw, (x1,y1),(x2,y2), color, 2)
```



```
l_depth = depth[sliceBox]

l_mask = get_lego_mask(hsvcolor, l_hsv) # filter mask by color
l_mask = np.where(l_depth < dist_tavolo, l_mask, 0)

l_depth = np.where(l_mask != 0, l_depth, dist_tavolo)

#getting lego height from camera and table
l_dist = get_lego_distance(l_depth)
l_height = dist_tavolo - l_dist

...
```

### 1. Nhập ảnh và thông tin vật thể:

**imgs:** Biến chứa bốn ảnh: ảnh RGB, ảnh HSV, ảnh depth và ảnh vẽ kết quả, được truyền vào dưới dạng một tuple.

**item:** Một dictionary chứa thông tin về một vật thể được phát hiện bởi Yolo:

- **x1, y1, x2, y2:** Tọa độ của bounding box bao quanh vật thể.
- **cn:** Độ tin cậy của việc phát hiện Yolo.
- **cl:** Loại vật thể được Yolo xác định (0, 1, 2).
- **nm:** Tên của vật thể dựa trên loại vật thể. ("X1-Y4-Z1",...)

**2. Xác định bounding box và tâm vật thể:** Thêm một khoảng biên margin vào bounding box để tránh trường hợp cạnh ảnh. Đảm bảo các tọa độ bounding box nằm trong phạm vi ảnh. Tính tọa độ trung tâm của bounding box.

**3. Xác định màu sắc vật thể:** Sử dụng hàm get\_lego\_color để lấy màu của vật thể trong cả ảnh RGB và HSV.

**4. Cắt ảnh theo bounding box:** Sử dụng slice để cắt các ảnh RGB, HSV và depth theo bounding box đã điều chỉnh.

**5. Vẽ bounding box (tùy chọn):** Nếu biến a\_show là True, vẽ bounding box trên ảnh img\_draw.

**6. Tạo vùng mask lọc vật thể:** Sử dụng hàm get\_lego\_mask trên ảnh HSV cắt lọc để lọc theo màu sắc của vật thể. Loại bỏ các điểm trong ảnh depth có giá trị vượt quá dist\_tavolo (giả sử là độ sâu của mặt bàn). Thay thế các điểm depth không được chọn bởi dist\_tavolo để đảm bảo tính toán chiều cao chính xác.

**7. Tính chiều cao của vật thể:** Sử dụng hàm get\_lego\_distance trên ảnh depth đã lọc để ước tính khoảng cách từ camera đến vật thể. Tính chiều cao của vật thể bằng cách trừ khoảng cách camera-vật thể khỏi khoảng cách camera-mặt bàn.

...



```
# model detect orientation
depth_borded = np.zeros(depth.shape, dtype=np.float32)
depth_borded[sliceBox] = l_depth

depth_image = cv.normalize(
    depth_borded, None, alpha=0, beta=255, norm_type=cv.NORM_MINMAX,
    dtype=cv.CV_8U
)
depth_image = cv.cvtColor(depth_image, cv.COLOR_GRAY2RGB).astype(np.uint8)

#yolo in order to keep te orientation
model_orientation.conf = 0.7
results = model_orientation(depth_image)
pandino = []
pandino = results.pandas().xyxy[0].to_dict(orient="records")

n = len(pandino)

# Adjust prediction
pinN, ax, isCorrect = getDepthAxis(l_height, nm)
if not isCorrect and ax == 2:
    if cl == 0 and pinN == 1: # X1-Y4-Z2
        cl = 0
    elif pinN == -1:
        nm = "{} -> {}".format(nm, "Target")
    else:
        print("[Warning] Error in classification")
elif not isCorrect:
    ax = 1
    if cl == 1 and pinN == 1: # X2-Y2-Z2
        cl = 2
        ax = 0
    else: print("[Warning] Error in classification")
nm = legoClasses[cl]

if n != 1:
    print("[Warning] Classification not found")
    or_cn, or_cl, or_nm = ['?']*3
    or_nm = ('lato', 'lato', 'sopra/sotto')[ax]
else:
    print()
    #obtaining Yolo informations (class, coordinates, center)
    or_item = pandino[0]
    or_cn, or_cl, or_nm = or_item['confidence'], or_item['class'],
        or_item['name']
    if or_nm == 'sotto': ax = 2
    if or_nm == 'lato' and ax == 2: ax = 1
#---
...

```

- 
- 1. Chuẩn bị ảnh depth cho định hướng:** Tạo một ảnh depth mới depth\_borded có kích thước bằng ảnh depth gốc, được lấp đầy bằng giá trị 0. Dùng sliceBox để cắt ảnh depth gốc theo bounding box đã điều chỉnh và sao chép vào ảnh depth mới depth\_image và chuẩn hóa ảnh depth thành dạng phù hợp với model định hướng (8 bit, RGB).

- 2. Sử dụng model Yolo để phân loại định hướng:** model\_orientation



chứa thông tin phân loại định hướng của vật thể. Thiết lập ngưỡng độ tin cậy (confidence) cho mô hình phân loại định hướng là 0.7. Kết quả trả về thông tin về các vật thể được phát hiện của model Yolo lưu trong results. Chuyển kết quả Yolo thành dạng dictionary trong pandino cho dễ xử lý .

### 3. Điều chỉnh dự đoán:

- **getDepthAxis:** Hàm tính toán trực chính của vật thể dựa trên chiều cao và loại vật thể.
- **pinN:** Trục chính được xác định bởi hàm getDepthAxis (1, 2, -1: X, Y, Z).
- **ax:** Trục được Yolo dự đoán (0, 1, 2: X, Y, Z).
- **isCorrect:** Kiểm tra tính hợp lý giữa dự đoán của Yolo và tính toán của hàm getDepthAxis.

**4. Cập nhật loại và tên vật thể:** Nếu dự đoán của Yolo không hợp lý, điều chỉnh loại vật thể dựa trên chiều cao và loại ban đầu.

**5. Kiểm tra tính hợp lý của dự đoán Yolo:** Nếu không tìm thấy vật thể hoặc Yolo trả về nhiều kết quả, in cảnh báo. or\_cn, or\_cl, or\_nm: Thông tin về vật thể được Yolo dự đoán (nếu không hợp lệ, tất cả được đặt thành '?'). ax: Điều chỉnh trực chính dựa trên tên vật thể được Yolo dự đoán.

...

```
#creating silouette top surface lego
contours, hierarchy = cv.findContours(l_top_mask, cv.RETR_TREE,
                                         cv.CHAIN_APPROX_SIMPLE)
top_center = np.zeros(2)
for cnt in contours:
    tmp_center, top_size, top_angle = cv.minAreaRect(cnt)
    top_center += np.array(tmp_center)
top_center = boxMin + top_center / len(contours)

#creating silouette lego
contours, hierarchy = cv.findContours(l_mask, cv.RETR_TREE,
                                         cv.CHAIN_APPROX_SIMPLE)
if len(contours) == 0: return None
cnt = contours[0]
l_center, l_size, l_angle = cv.minAreaRect(cnt)
l_center += boxMin
if ax != 2: l_angle = top_angle
l_box = cv.boxPoints((l_center, l_size, l_angle))

if l_size[0] <=3 or l_size[1] <=3:
    cv.drawContours(img_draw, np.int0([l_box]), 0, (0,0,0), 2)
    return None # filter out artifacts

if a_show: cv.drawContours(img_draw, np.int0([l_box]), 0, color, 2)

# silouette distortion
# get vertexs distance from origin
top_box = l_box.copy()
vertexs_norm = [(i, np.linalg.norm(vec - origin)) for vec, i in zip(l_box,
                                                               range(4))]
```



```
vertexs_norm.sort(key=lambda tup: tup[1])
# get closest vertex
iver = vertexs_norm[0][0]
vec = l_box[iver]
# distorting closest vertex
if or_nm == 'sopra': l_height -= 0.019
top_box[iver] = point_distortion(l_box[iver], l_height, origin)
v0 = top_box[iver] - vec
# adapt adiacents veretx
v1 = l_box[iver - 3] - vec # i - 3 = i+1 % 4
v2 = l_box[iver - 1] - vec

top_box[iver - 3] += np.dot(v0, v2) / np.dot(v2, v2) * v2
top_box[iver - 1] += np.dot(v0, v1) / np.dot(v1, v1) * v1

l_center = (top_box[0] + top_box[2]) / 2

if a_show:
    cv.drawContours(img_draw, np.int0([top_box]), 0, (5,5,5), 2)
    cv.circle(img_draw, np.int0(top_box[iver]), 1, (0,0,255), 1, cv.LINE_AA)

...
```

Đoạn code này xử lý việc xử lý hình bóng của một khối lego và điều chỉnh hình dạng của nó dựa trên thông tin bổ sung.

### 1. Hình bóng bề mặt trên:

- **findContours:** Tìm các đường viền của mặt nạ bề mặt trên (l\_top\_mask) bằng các hàm OpenCV.
- **minAreaRect:** Đối với mỗi đường viền, tính toán hình chữ nhật bao quanh diện tích nhỏ nhất của nó, trả về tâm, kích thước và góc.
- **top\_center:** Tổng hợp các tâm của tất cả các đường viền và chia cho số lượng của chúng để tìm tâm trung bình của bề mặt trên.

**2. Hình bóng (silhouette) lego:** Tương tự như bề mặt trên, tìm các đường viền của toàn bộ mặt nạ lego (l\_mask). Nếu không tìm thấy đường viền nào, hàm sẽ trả về None (có thể cho biết không phát hiện thấy lego). Nếu không, nó trích xuất đường viền đầu tiên, giả định nó đại diện cho thân chính của lego.

- minAreaRect: Tính toán hình chữ nhật bao quanh diện tích nhỏ nhất cho thân lego.
- **l\_center:** Lưu trữ tâm của hình chữ nhật thân lego.
- **l\_angle:** Lưu trữ góc của hình chữ nhật, có thể được điều chỉnh để khớp với góc bề mặt trên sau này.
- **l\_box:** Tạo ra 4 điểm góc của hình chữ nhật bao quanh lego dựa trên tâm, kích thước và góc của nó.

**3. Lọc các đối tượng nhỏ:** Kiểm tra xem kích thước lego (chiều rộng và chiều cao) có nhỏ hơn 3 pixel hay không. Nếu vậy, vẽ một đường viền xung quanh nó và trả về None để lọc nhiễu hoặc đối tượng nhỏ.

**4. Vẽ và biến dạng (tùy chọn):** Dựa trên cờ a\_show, nó vẽ hộp bao quanh lego trên hình ảnh img\_draw.

**5. Biến dạng hình bóng (dựa trên or\_nm):** Phần này áp dụng một biến dạng cho hình bóng dựa trên tên đối tượng (or\_nm) và chiều cao (l\_height).

- **top\_box:** Tạo một bản sao của hộp bao quanh lego.
- **vertices\_norm:** Tính toán khoảng cách của mỗi điểm góc (đỉnh) từ gốc và lưu trữ chúng trong một danh sách được sắp xếp.
- **iver:** Xác định chỉ mục của đỉnh gần nhất với gốc.
- **vec:** Lưu trữ vị trí của đỉnh gần nhất.

Biến dạng dựa trên tên đối tượng: Nếu tên đối tượng là "sopra" (nghĩa là "trên"), chiều cao lego sẽ giảm 0,019, giả sử mô phỏng một lực ấn nhẹ xuống.

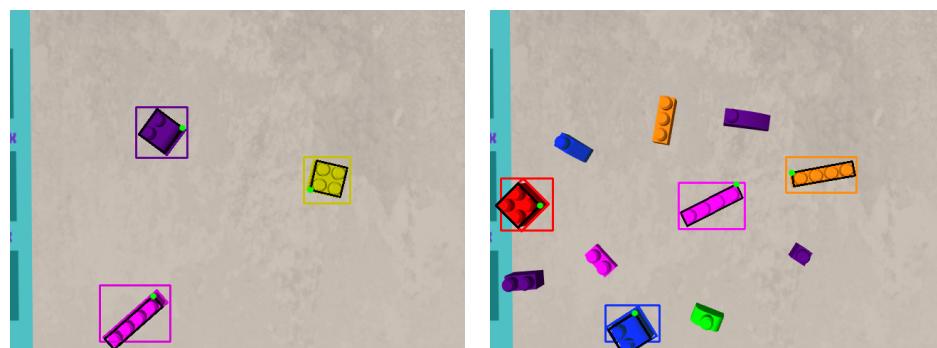
Hàm **point\_distortion()** thực hiện một biến dạng cho một điểm, theo hướng đẩy điểm đó xuống một khoảng cách nhất định. Khoảng cách này được xác định bởi tham số height. Hàm hoạt động như sau: Tính toán tỷ lệ giữa khoảng cách từ gốc đến điểm cần biến dạng (dist\_tavolo) và khoảng cách từ gốc đến điểm đó sau khi bị biến dạng (dist\_tavolo - height). Sau đó dịch chuyển điểm cần biến dạng về gốc. Cuối cùng nhân điểm đã dịch chuyển với tỷ lệ đã tính được và cộng thêm gốc.

Các đỉnh liền kề được điều chỉnh bằng cách chiếu vector v0 lên các vector tương ứng của chúng (v1 và v2). Điều này có thể nhằm duy trì hình dạng tổng thể của lego trong khi kết hợp sự biến dạng. Trong đó:

- **v0:** Vector chỉ từ vị trí ban đầu của đỉnh gần nhất đến vị trí bị biến dạng của nó.
- **v1, v2:** Vectors chỉ từ các vị trí ban đầu của các đỉnh liền kề đến gốc.

Cuối cùng:

- **l\_center:** Tính toán lại tâm của hộp bao quanh lego bị biến dạng bằng cách lấy trung bình hai góc trên.
- **a\_show:** Nếu được bật, sẽ vẽ hộp bao quanh lego bị biến dạng trên hình ảnh và tô sáng đỉnh gần nhất bằng một vòng tròn màu xanh lá.



Hình 26: Vẽ the distorted lego bounding box và highlight đỉnh gần nhất



...

```
#rotation and axis drawing
if or_nm in ('sopra', 'sotto', 'sopra/sotto', '?'): # fin x, y directions (dirX,
    dirY)
    dirZ = np.array((0,0,1))
    if or_nm == 'sotto': dirZ = np.array((0,0,-1))

    projdir = l_center - top_center
    if np.linalg.norm(projdir) < l_size[0] / 10:
        dirY = top_box[0] - top_box[1]
        dirX = top_box[0] - top_box[-1]
        if np.linalg.norm(dirY) < np.linalg.norm(dirX): dirX, dirY = dirY, dirX
        projdir = dirY * np.dot(dirY, projdir)
    edgeFar = [ver for ver in top_box if np.dot(ver - l_center, projdir) >=
        0][:]
    dirY = (edgeFar[0] + edgeFar[1]) / 2 - l_center
    dirY /= np.linalg.norm(dirY)
    dirY = np.array((*dirY, 0))
    dirX = np.cross(dirZ, dirY)

elif or_nm == "lato": # find pin direction (dirZ)
    edgePin = [ver for ver in top_box if np.dot(ver - l_center, l_center -
        top_center) >= 0][:]

    dirZ = (edgePin[0] + edgePin[1]) / 2 - l_center
    dirZ /= np.linalg.norm(dirZ)
    dirZ = np.array((*dirZ, 0))

    if cl == 10:
        if top_size[1] > top_size[0]: top_size = top_size[::-1]
        if top_size[0] / top_size[1] < 1.7: ax = 0
    if ax == 0:
        vx,vy,x,y = cv.fitLine(cnt, cv.DIST_L2,0,0.01,0.01)
        dir = np.array((vx, vy))
        vertexs_distance = [abs(np.dot(ver - l_center, dir)) for ver in edgePin]
        iverFar = np.array(vertexs_distance).argmin()

        dirY = edgePin[iverFar] - edgePin[iverFar-1]
        dirY /= np.linalg.norm(dirY)
        dirY = np.array((*dirY, 0))
        dirX = np.cross(dirZ, dirY)
        if a_show: cv.circle(img_draw, np.int0(edgePin[iverFar]), 5, (70,10,50),
            1)
        #cv.line(img_draw, np.int0(l_center),
        #np.int0(l_center+np.array([int(vx*100),int(vy*100)])),(0,0,255), 3)
    if ax == 1:
        dirY = np.array((0,0,1))
        dirX = np.cross(dirZ, dirY)

    if a_show: cv.line(img_draw, *np.int0(edgePin), (255,255,0), 2)

l_center = point_inverse_distortption(l_center, l_height)
```

...

Đoạn code trên trích xuất thông tin về hướng và trục của một vật thể trong ảnh dựa trên các điểm đánh dấu. Dưới đây là giải thích từng phần:

### 1. Xác định hướng trục Z (dirZ):



Kiểm tra or\_nm:

- Nếu bằng "sopra" (trên):  $\text{dirZ} = (0, 0, 1)$ .
- Nếu bằng "sotto" (dưới):  $\text{dirZ} = (0, 0, -1)$ .
- Nếu bằng "lato" (bên):
  - \* Tìm các điểm đánh dấu cạnh (edgePin) dựa trên vị trí tương đối so với tâm vật thể ( $l\_center$ ).
  - \* Tính  $\text{dirZ}$  là vector hướng từ tâm vật thể đến điểm đánh dấu trung bình của các cạnh.

## 2. Xác định hướng trục Y (dirY):

- Nếu or\_nm là "sopra/sotto" (trên/dưới) hoặc không xác định (?):
  - \* Kiểm tra kích thước vật thể (top\_size):
    - Nếu chiều cao (y) lớn hơn chiều rộng (x), hoán đổi vị trí hai giá trị.
    - Nếu tỉ lệ giữa chiều rộng và chiều cao nhỏ hơn 1.7, đặt  $ax = 0$ .
  - \* Nếu  $ax = 0$ :
    - Xác định vector hướng dựa trên đường biên của vật thể (cnt).
    - Tìm điểm đánh dấu xa nhất so với tâm theo hướng vừa xác định (iverFar).
    - Tính  $\text{dirY}$  là vector hướng từ điểm đánh dấu xa nhất đến điểm đánh dấu liền kề.
  - \* Nếu  $ax = 1$ : Đặt  $\text{dirY} = (0, 0, 1)$ .
- Nếu or\_nm là "lato": Tính  $\text{dirY}$  là vector vuông góc với  $\text{dirZ}$  và hướng từ tâm vật thể đến điểm đánh dấu trung bình của các cạnh.

## 3. Xác định hướng trục X (dirX):

Tính  $\text{dirX}$  là vector tích có hướng của  $\text{dirZ}$  và  $\text{dirY}$ .

```
...
# post rotation extra
theta = 0
if cl == 2 and or_nm == 'sotto': theta = 2.496793 - np.pi

rotX = PyQuaternion(axis=dirX, angle=theta)
dirY = rotX.rotate(dirY)
dirZ = rotX.rotate(dirZ)

if a_show:
    # draw frame
    lenFrame = 50
    unit_z = 0.031
    unit_x = 22 * 0.8039 / dist_tavolo
    x_to_z = lenFrame * unit_z/unit_x
    center = np.int0(l_center)

    origin_from_top = origin - l_center

    endX = point_distortion(lenFrame * dirX[:2], x_to_z * dirX[2], origin_from_top)
    frameX = (center, center + np.int0(endX))
```



```
endY = point_distortion(lenFrame * dirY[:2], x_to_z * dirY[2], origin_from_top)
frameY = (center, center + np.int0(endY))

endZ = point_distortion(lenFrame * dirZ[:2], x_to_z * dirZ[2], origin_from_top)
frameZ = (center, center + np.int0(endZ))

cv.line(img_draw, *frameX, (0,0,255), 2)
cv.line(img_draw, *frameY, (0,255,0), 2)
cv.line(img_draw, *frameZ, (255,0,0), 2)
# ---

# draw text
if or_cl != '?': or_cn = ['SIDE', 'UP', 'DOWN'][or_cl]
text = "{} {:.2f} {}".format(nm, cn, or_cn)
(text_width, text_height) = cv.getTextSize(text, cv.FONT_HERSHEY_DUPLEX, 0.4,
    1)[0]
text_offset_x = boxCenter[1] - text_width // 2
text_offset_y = y1 - text_height
box_coords = ((text_offset_x - 1, text_offset_y + 1), (text_offset_x +
    text_width + 1, text_offset_y - text_height - 1))
cv.rectangle(img_draw, box_coords[0], box_coords[1], (210,210,10), cv.FILLED)
cv.putText(img_draw, text, (text_offset_x, text_offset_y),
    cv.FONT_HERSHEY_DUPLEX, 0.4, (255, 255, 255), 1)

def getAngle(vec, ax):
    vec = np.array(vec)
    if not vec.any(): return 0
    vec = vec / np.linalg.norm(vec)
    wise = 1 if vec[-1] >= 0 else -1
    dotclamp = max(-1, min(1, np.dot(vec, np.array(ax))))
    return wise * np.arccos(dotclamp)

msg = ModelStates()
msg.name = nm
#fov = 1.047198
#rap = np.tan(fov)
#print("rap: ", rap)
xyz = np.array((l_center[0], l_center[1], l_height / 2 + height_tavolo))
xyz[:2] /= rgb.shape[1], rgb.shape[0]
xyz[:2] -= 0.5
xyz[:2] *= (-0.968, 0.691)
xyz[:2] *= dist_tavolo / 0.84
xyz[:2] += cam_point[:2]

rdirX, rdirY, rdirZ = dirX, dirY, dirZ
rdirX[0] *= -1
rdirY[0] *= -1
rdirZ[0] *= -1
qz1 = PyQuaternion(axis=(0,0,1), angle=-getAngle(dirZ[:2], (1,0)))
rdirZ = qz1.rotate(dirZ)
qy2 = PyQuaternion(axis=(0,1,0), angle=-getAngle((rdirZ[2],rdirZ[0]), (1,0)))
rdirX = qy2.rotate(qz1.rotate(rdirX))
qz3 = PyQuaternion(axis=(0,0,1), angle=-getAngle(rdirX[:2], (1,0)))

rot = qz3 * qy2 * qz1
rot = rot.inverse
msg.pose = Pose(Point(*xyz), Quaternion(x=rot.x,y=rot.y,z=rot.z,w=rot.w))

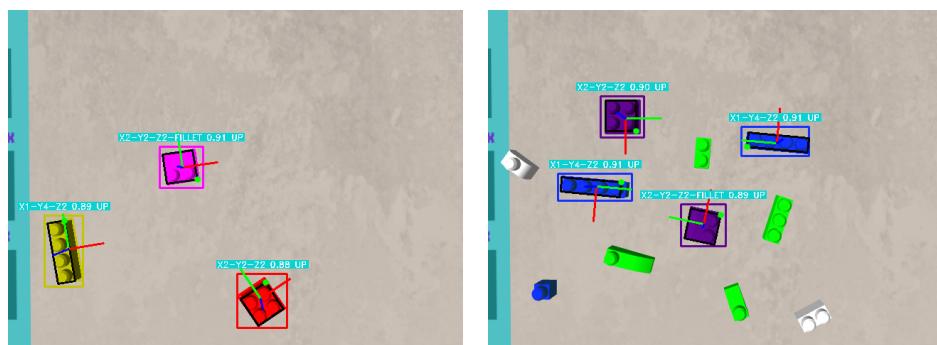
#pub.publish(msg)
#print(msg)
return msg
```

Đoạn code này thực hiện các tác vụ sau:

- **Tính toán hướng của vật thể:** Dựa vào các thông tin như loại vật thể, tên vật thể, vị trí và hướng của camera, code sẽ tính toán hướng của vật thể trên một khung hình.
- **Điều chỉnh hướng của khung tọa độ cục bộ:** Code sẽ điều chỉnh hướng của khung tọa độ cục bộ (local coordinate frame) sao cho nó trùng khớp với hướng của vật thể bằng cách sử dụng hàm `PyQuaternion.rotate()`. Hàm này sẽ thực hiện phép quay một vector theo một quaternion. Code sẽ sử dụng hàm `qz3 * qy2 * qz1`.

Trong đó, `qz3`, `qy2` và `qz1` là ba quaternion được tạo ra từ ba vector `dirX`, `dirY` và `dirZ`. Về mặt hình học, hàm `qz3 * qy2 * qz1` sẽ thực hiện phép quay vector `dirX` theo quaternion `qz3`, sau đó thực hiện phép quay vector `dirY` theo quaternion `qy2`, và cuối cùng thực hiện phép quay vector `dirZ` theo quaternion `qz1`.

- **Vẽ khung tọa độ và nhãn trên khung hình:** Sau khi tính toán hướng, code sẽ vẽ khung tọa độ và nhãn (tên vật thể, hướng) lên khung hình.
- **Tạo và gửi tin nhắn trạng thái:** Code sẽ tạo và gửi một tin nhắn trạng thái chứa thông tin về vị trí và hướng của vật thể.



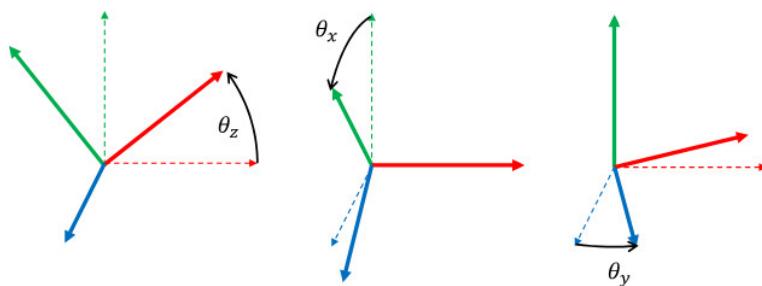
Hình 27: Vẽ khung tọa độ và nhãn trên khung hình

## 6 Điều khiển cánh tay UR5

### 6.1 Lý thuyết cơ bản để điều khiển một cánh tay robot

#### 6.1.1 Ma trận xoay(Rotaion Matrix)

Ma trận xoay là ma trận biến đổi được sử dụng để thực hiện phép quay xung quanh một trục tọa độ cố định trong không gian Euclidean. Trong môi trường 3D, giả sử hệ tọa độ xoay quay một trục cố định một góc  $\theta$ .



Hình 28: Xoay một góc  $\theta$  quanh lần lượt các trục x,y,z

Sau khi dùng các phép tính và biến đổi đơn giản, ta có ba ma trận xoay quanh các trục x,y,z như sau:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \quad R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \quad R_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

#### 6.1.2 Ma trận Homogeneous

Một ma trận homogeneous là một loại ma trận trong đó các phần tử trong hàng cuối cùng của ma trận đều là 0, ngoại trừ phần tử ở góc dưới bên phải, có giá trị là 1. Ma trận này thường được sử dụng trong các phép biến đổi tọa độ và đa phương tiện để thực hiện các phép biến đổi không đổi vị trí vật lý hoặc không gian trong không gian đa chiều. Trong trường hợp của cánh tay robot, ta quy ước một ma trận Homogeneous  $4 \times 4$ . Đối với một khớp di động (ví dụ như một khớp xoay), ma trận này có thể có dạng như sau:

$$\begin{bmatrix} R & d \\ 0 & 1 \end{bmatrix}$$

Trong đó:

- $R$  là ma trận  $3 \times 3$  biểu diễn ma trận xoay và biến đổi hướng của khớp.
- $d$  là vectơ cột  $3 \times 1$  biểu diễn vị trí của khớp trong không gian.

Ma trận này cho phép biểu diễn cả phép xoay và phép dịch chuyển của khớp. Khi kết hợp các ma trận homogeneous của các khớp liên tiếp lại, ta có thể tính toán được vị trí và hướng của tất cả các khớp trong robot, từ đó xác định được vị trí và hướng của cánh tay robot trong không gian.

### 6.1.3 Động học tịnh tiến(Forward Kinematics)

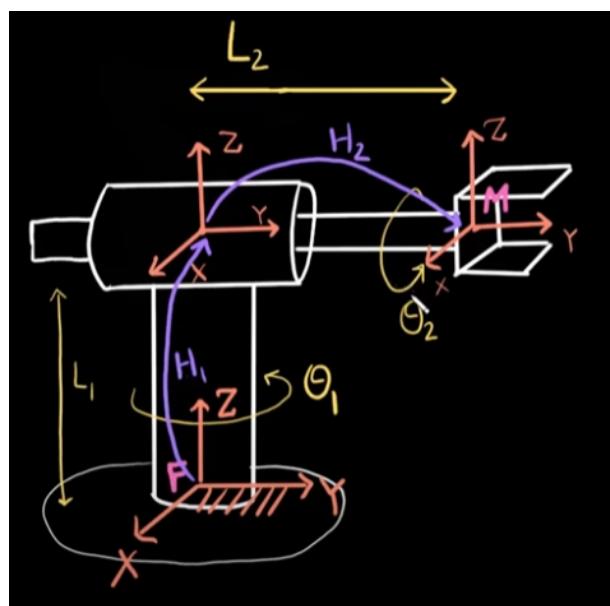
Động học tịnh tiến là một phần quan trọng trong lĩnh vực robot học và robot cơ bản. Nó liên quan đến việc xác định vị trí và hướng của các phần tử trong một cấu trúc robot khi biết các thông số và các góc quay của các khớp.

Mỗi robot có cấu trúc khác nhau, nhưng chúng thường bao gồm các khớp và liên kết giữa chúng. Forward kinematics giúp ta xác định vị trí và hướng của "công cụ" hoặc "cánh tay" của robot (có thể là bàn tay, đầu cắt, hoặc bất kỳ công cụ nào mà robot sử dụng) dựa trên thông số của các khớp trong cấu trúc robot.

Để thực hiện forward kinematics, ta thường sử dụng ma trận homogeneous để biểu diễn các phép biến đổi từ hệ thống tọa độ của một khớp đến hệ thống tọa độ của khớp tiếp theo. Các ma trận này gồm phần xoay và phần dịch chuyển, cho phép biểu diễn cả sự quay và dịch chuyển của các khớp.

Quá trình này thực hiện từ khớp đầu tiên đến khớp cuối cùng(thường gọi là end-effector), tính toán các phép biến đổi tọa độ từ gốc tọa độ của robot (thường là tọa độ của một khớp cố định như khớp cơ sở) đến vị trí và hướng của công cụ hoặc cánh tay của robot trong không gian.

Một ví dụ đơn giản sử dụng phép động học tịnh tiến:



Hình 29: Youtube: Engineering Simplified

Ta có các thông số được cho sẵn:

- $L_1, L_2$  lần lượt là khớp nối(links) của cánh tay.
- $\theta_1, \theta_2$  lần lượt là góc xoay của các khớp(joints).
- $H_1, H_2$  sẽ là ma trận Homogeneous của từng khớp.

Gọi  $H$  là ma trận biểu diễn phép biến đổi từ hệ thống tọa độ gốc đến end effector của cánh tay robot. Ta có:

$$H = H_1 \cdot H_2$$

Ta cần tìm hai ma trận  $H_1, H_2$ : Vì  $\theta_1$  xoay quanh trục Z nên ta có ma trận

$$R_z(\theta_1) = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1) & 0 \\ \sin(\theta_1) & \cos(\theta_1) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

, đồng thời ta di chuyển hệ lên  $L_1$  theo trục Z nên ta có ma trận  $d = \begin{bmatrix} 0 \\ 0 \\ L_1 \end{bmatrix}$

Sau khi ghép các ma trận lại với nhau theo công thức homogeneous trên ta có ma trận  $H_1$ :

$$H_1 = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1) & 0 & 0 \\ \sin(\theta_1) & \cos(\theta_1) & 0 & 0 \\ 0 & 0 & 1 & L_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Tương tự ta có ma trận  $H_2$ :

$$H_2 = \begin{bmatrix} \cos(\theta_2) & 0 & \sin(\theta_2) & 0 \\ 0 & 1 & 0 & L_2 \\ -\sin(\theta_2) & 0 & \cos(\theta_2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

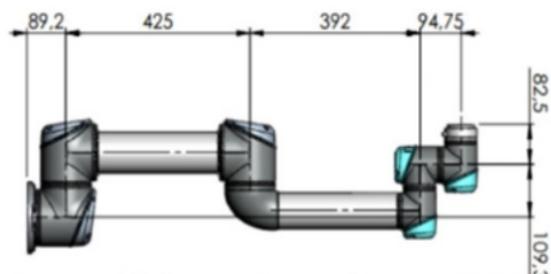
#### 6.1.4 Mô hình Denavit-Hartenberg

Mô hình D-H (Denavit-Hartenberg) là một phương pháp phổ biến để mô tả và phân tích không gian làm việc của robot trong robot kinematics. Nó sử dụng một số tham số để đặc tả vị trí và hướng của các khớp và liên kết trong robot.

Mỗi khớp của robot được đặc tả thông qua bốn tham số:

1.  $a$ : Độ dài dọc theo trục  $x$  giữa các trục  $z_i$  và  $z_{i+1}$ .
2.  $\alpha$ : Góc quay xung quanh trục  $x_i$  từ  $z_i$  đến  $z_{i+1}$ .
3.  $d$ : Độ dài dọc theo trục  $z_{i+1}$  từ  $z_i$  đến giao điểm với trục  $z_{i+1}$ .
4.  $\theta$ : Góc quay xung quanh trục  $z_{i+1}$  từ  $x_i$  đến  $x_{i+1}$ .

Bằng cách sử dụng các tham số này, ta có thể xác định một ma trận chuyển đổi dịch chuyển và xoay từ khớp  $i$  đến khớp  $i + 1$ . Khi kết hợp các ma trận này từ khớp đầu tiên đến khớp cuối cùng của robot, ta có thể tính toán được ma trận biến đổi cuối cùng, mô tả vị trí và hướng của công cụ cuối cùng của robot trong không gian toàn cầu. Dưới đây là bảng thông số DH của cánh tay UR5.



Hình 30: Structure diagram



I	$d_i$	$a_i$	$\alpha_i$	$\theta_i$
1	$d_1 = 89.459$	0	$\pi/2$	$\theta_1$
2	0	$a_2 = -425$	0	$\theta_2$
3	0	$a_3 = 392.25$	0	$\theta_3$
4	$d_4 = 109.15$	0	$\pi/2$	$\theta_4$
5	$d_5 = 94.65$	0	$-\pi/2$	$\theta_5$
6	$d_6 = 82.3$	0	0	$\theta_6$

Bảng 2: Standard DH series parameters

Mô hình D-H giúp trong việc mô tả và tính toán động học của robot, bao gồm việc tính toán đường đi, tốc độ, gia tốc và lực tác động trên robot trong quá trình thực hiện các tác vụ.

#### 6.1.5 Chuyển động đảo ngược (Inverse Kinematics)

Inverse kinematics (IK) là quá trình dùng để tính toán các giá trị góc quay của các khớp trong robot dựa trên vị trí và hướng mong muốn của end effector hoặc của một điểm nào đó trong không gian.

Forward kinematics giúp chúng ta tính toán vị trí và hướng của end effector (công cụ cuối cùng) dựa trên các góc quay của các khớp. Trong khi đó, inverse kinematics là quá trình ngược lại, nó giúp chúng ta tìm ra các góc quay của các khớp để đưa end effector đến một vị trí và hướng cụ thể.

Vấn đề phức tạp trong inverse kinematics là có thể có nhiều tổ hợp các góc  $\theta$  cho một vị trí và hướng cụ thể của end effector. Trong một số trường hợp, có thể không có tổ hợp hoặc có nhiều tổ hợp, và việc chọn giải pháp phù hợp thường đòi hỏi quyết định thông minh dựa trên ràng buộc và điều kiện cụ thể của robot.

Để tránh bài báo cáo quá dài và phức tạp, chúng tôi sẽ chỉ đưa ra một minh họa trực tiếp của UR5. Cho sẵn vị trí và góc xoay của điểm cuối(end effector), chúng ta sẽ dùng các công cụ thích hợp(Python, Matlab,...) để tìm ra các góc xoay của các khớp từ một ma trận có sẵn.

Như đã trình bày ở trên ta có công thức:

$$T = {}_1^0 T \cdot {}_2^1 T \cdot {}_3^2 T \cdot {}_4^3 T \cdot {}_5^4 T \cdot {}_6^5 T$$

Với:

- $T$ : là ma trận chuyển đổi Homogeneous từ vị trí đầu(base) đến vị trí cuối(end effector).
- ${}_i^{i-1} T$ :  $i=(1,2,3,4,5,6)$  biểu thị cho vị trí của 6 khớp của cánh tay.

Ta có ma trận biểu hiện vị trí và góc xoay mong muốn:

$$T = \begin{bmatrix} 0.1623 & -0.3938 & 0.9047 & 0.1377 \\ -0.5888 & 0.6972 & 0.4091 & -0.0699 \\ -0.7919 & -0.5991 & -0.1187 & 0.2612 \\ 0 & 0 & 0 & 1.0000 \end{bmatrix}$$

Sử dụng các phép tính và biến đổi, ta có 8 phương án thỏa mãn điều kiện:



1.0005	1.0005	0.9998	1.0001	0.9996	1.0000
1.0005	1.9562	-0.9998	2.0437	0.9996	1.0000
1.0005	0.6743	1.7373	-2.5532	-0.9996	4.1416
1.0005	2.3170	-1.7373	-0.7212	-0.9996	4.1416
0.0959	1.1462	1.0085	0.8613	1.8971	0.8826
0.0959	2.1105	-1.0085	1.9139	1.8971	0.8826
0.0959	0.8754	1.7300	-2.7310	-1.8971	4.0242
0.0959	2.5114	-1.7300	-0.9071	-1.8971	4.0242

Bảng 3: Các tổ hợp 6 góc  $\theta$  từ trái qua phải

Tùy yêu cầu của robot mà ta sẽ lọc các phương án này để cho ra phương án thỏa mãn nhất.

## 6.2 Hiện thực

Ta sẽ chia scripts Python để điều khiển cánh tay thành ba module:

- **kinematics.py**: Dùng forward kinematics để tìm trạng thái của end-effector và inverser kinematics để tính các góc xoay của các khớp với trạng thái end-effector cho sẵn.
- **controller.py**: Chứa một lớp dùng để di chuyển cánh tay.
- **motion\_planning.py**: Nhận vị trí của những miếng lego và phân loại chúng vào các vị trí cho sẵn.

Ta sẽ đi vào chi tiết từng module ở các phần sau.

### 6.2.1 kinematics.py:

Khai báo các thông số của UR5 theo mô hình DH:

```
d = [0.089159, 0.00000, 0.00000, 0.10915, 0.09465, 0.0903 + 0.1628]
a = [0.00000, -0.42500, -0.39225, 0.00000, 0.00000, 0.0000]
alph = [pi/2, 0, 0, pi/2, -pi/2, 0]
```

Tìm ma trận Homogeneous của khớp n so với khớp n-1

```
def AH(n, th):
    T_a = np.identity(4)
    T_a[0, 3] = a[n - 1]

    T_d = np.identity(4)
    T_d[2, 3] = d[n - 1]

    Rzt = arr(
        [[cos(th[n - 1]), -sin(th[n - 1]), 0, 0],
         [sin(th[n - 1]), cos(th[n - 1]), 0, 0],
         [0, 0, 1, 0],
         [0, 0, 0, 1]])

    Rxa = arr(
        [[1, 0, 0, 0],
         [0, cos(alph[n - 1]), -sin(alph[n - 1]), 0],
         [0, sin(alph[n - 1]), cos(alph[n - 1]), 0],
         [0, 0, 0, 1]])
```



```
return T_d @ Rzt @ T_a @ Rxa
```

Sử dụng forward kinematics để tìm trạng thái của end-effector

```
def forward(th):
    A_1 = AH(1, th)
    A_2 = AH(2, th)
    A_3 = AH(3, th)
    A_4 = AH(4, th)
    A_5 = AH(5, th)
    A_6 = AH(6, th)

    T_06 = A_1 @ A_2 @ A_3 @ A_4 @ A_5 @ A_6

    return T_06
```

Sử dụng inverse kinematics để tính các góc  $\theta$  từ một trạng thái end-effector cho sẵn

```
def inverse(desired_pos): # T60
    th = np.zeros((6, 8), dtype=np.float64)
    P_05 = desired_pos @ [0, 0, -d[5], 1] - [0, 0, 0, 1]

    P_05 = np.asarray(P_05).flatten()

    # **** theta1 ****

    psi = atan2(P_05[2 - 1], P_05[1 - 1])
    phi = acos(d[3] / sqrt(P_05[2 - 1] * P_05[2 - 1] + P_05[1 - 1] * P_05[1 - 1]))

    # The two solutions for theta1 correspond to the shoulder
    # being either left or right

    th[0, 0:4] = pi / 2 + psi + phi
    th[0, 4:8] = pi / 2 + psi - phi
    th = th.real

    # **** theta5 ****

    cl = [0, 4] # wrist up or down
    for c in cl:
        T_10 = linalg.inv(AH(1, th[:, c]))
        T_16 = T_10 @ desired_pos
        th[4, c:c + 2] = +acos((T_16[2, 3] - d[3]) / d[5])
        th[4, c + 2:c + 4] = -acos((T_16[2, 3] - d[3]) / d[5])

    th = th.real

    # **** theta6 ****
    # theta6 is not well-defined when sin(theta5) = 0 or when T16(1,3), T16(2,3) = 0.

    cl = [0, 2, 4, 6]
    for c in cl:
        T_10 = linalg.inv(AH(1, th[:, c]))
        T_16 = linalg.inv(T_10 @ desired_pos)
        th[5, c:c + 2] = atan2(-T_16[1, 2] / sin(th[4, c]), T_16[0, 2] / sin(th[4, c]))

    th = th.real
```



```
# **** theta3 ****

cl = [0, 2, 4, 6]
for c in cl:
    T_10 = linalg.inv(AH(1, th[:, c]))
    T_65 = AH(6, th[:, c])
    T_54 = AH(5, th[:, c])
    T_14 = T_10 @ desired_pos @ linalg.inv(T_54 @ T_65)
    P_13 = T_14 @ [0, -d[3], 0, 1] - [0, 0, 0, 1]
    t3 = cmath.acos((linalg.norm(P_13) ** 2 - a[1] ** 2 - a[2] ** 2) / (2 * a[1] *
        a[2])) # norm ?
    th[2, c] = t3.real
    th[2, c + 1] = -t3.real

# **** theta2 and theta 4 ****

cl = [0, 1, 2, 3, 4, 5, 6, 7]
for c in cl:
    T_10 = linalg.inv(AH(1, th[:, c]))
    T_65 = linalg.inv(AH(6, th[:, c]))
    T_54 = linalg.inv(AH(5, th[:, c]))
    T_14 = T_10 @ desired_pos @ T_65 @ T_54
    P_13 = T_14 @ [0, -d[3], 0, 1] - [0, 0, 0, 1]

    # theta 2

    th[1, c] = -atan2(P_13[1], -P_13[0]) + asin(a[2] * sin(th[2, c]))
    / linalg.norm(P_13)

    # theta 4

    T_32 = linalg.inv(AH(3, th[:, c]))
    T_21 = linalg.inv(AH(2, th[:, c]))
    T_34 = T_32 @ T_21 @ T_14
    th[3, c] = atan2(T_34[1, 0], T_34[0, 0])
th = th.real
return th
```

Từ vị trí và góc quay cho sẵn, biến đổi về dạng ma trận rồi dùng hàm inverse để tính các góc  $\theta$ , sau khi có được các góc, biến đổi về dạng có thể xử lý được rồi trả về một phương án

```
def get_joints(x, y, z, rot):
    # base offset
    z -= 0.771347 #-0.016300

    # create trasform matrix
    pose = np.zeros((4, 4), dtype=np.float64)
    pose[:, 3] = (x, y, z, 1)
    pose[:3, :3] = rot
    th_res = inverse(pose)

    # normalize
    th_res = (th_res + pi) % (2 * pi) - pi

    # return 5th kinematic solution
    return th_res[:, 5]
```

Từ các thông số cho sẵn, tìm ra vị trí và góc xoay của end-effector



---

```
def get_pose(joints):
    th = forward(joints)

    x, y, z = th[:3, 3]
    rot = th[:3, :3]

    z += 0.771347
    return x, y, z, rot
```

---

### 6.2.2 controller.py

Module này sẽ chứa một class "ArmController" dùng để di chuyển cánh tay đến một vị trí cho trước. Class "ArmController" có method chính "move to" dùng để di chuyển cánh tay và các methods phụ để hỗ trợ method này(publish joint\_states cho các controller\_topic)

```
def get_controller_state(controller_topic, timeout=None):
    return rospy.wait_for_message(
        f"{controller_topic}/state",
        control_msgs.msg.JointTrajectoryControllerState,
        timeout=timeout)

class ArmController:
    def __init__(self, gripper_state=0, controller_topic="/trajectory_controller"):
        self.joint_names = [
            "shoulder_pan_joint",
            "shoulder_lift_joint",
            "elbow_joint",
            "wrist_1_joint",
            "wrist_2_joint",
            "wrist_3_joint",
        ]
        self.gripper_state = gripper_state

        self.controller_topic = controller_topic
        self.default_joint_trajectory = trajectory_msgs.msg.JointTrajectory()
        self.default_joint_trajectory.joint_names = self.joint_names

        joint_states = get_controller_state(controller_topic).actual.positions
        x, y, z, rot = kinematics.get_pose(joint_states)
        self.gripper_pose = (x, y, z), Quaternion(matrix=rot)

        # Create an action client for the joint trajectory
        self.joints_pub = rospy.Publisher(
            f"{self.controller_topic}/command",
            trajectory_msgs.msg.JointTrajectory, queue_size=10)

    def move(self, dx=0, dy=0, dz=0, delta_quat=Quaternion(1, 0, 0, 0), blocking=True):
        (sx, sy, sz), start_quat = self.gripper_pose

        tx, ty, tz = sx + dx, sy + dy, sz + dz
        target_quat = start_quat * delta_quat

        self.move_to(tx, ty, tz, target_quat, blocking=blocking)

    def move_to(self, x=None, y=None, z=None, target_quat=None, z_raise=0.0,
               blocking=True):
```



```
"""
Move the end effector to target_pos with target_quat as orientation
"""

def smooth(percent_value, period=math.pi):
    return (1 - math.cos(percent_value * period)) / 2

(sx, sy, sz), start_quat = self.gripper_pose

if x is None:
    x = sx
if y is None:
    y = sy
if z is None:
    z = sz
if target_quat is None:
    target_quat = start_quat

dx, dy, dz = x - sx, y - sy, z - sz
length = math.sqrt(dx ** 2 + dy ** 2 + dz ** 2) * 300 + 80
speed = length

steps = int(length)
step = 1 / steps

for i in np.arange(0, 1 + step, step):
    i_2 = smooth(i, 2 * math.pi) # from 0 to 1 to 0
    i_1 = smooth(i) # from 0 to 1

    grip = Quaternion.slerp(start_quat, target_quat, i_1)
    self.send_joints(
        sx + i_1*dx, sy + i_1*dy, sz + i_1*dz + i_2*z_raise,
        grip,
        duration=1/speed*0.9)
    rospy.sleep(1/speed)

if blocking:
    self.wait_for_position(tol_pos=0.005, tol_vel=0.08)

self.gripper_pose = (x, y, z), target_quat

def send_joints(self, x, y, z, quat, duration=1.0): # x,y,z and orientation of lego
    block
    # Solve for the joint angles, select the 5th solution
    joint_states = kinematics.get_joints(x, y, z, quat.rotation_matrix)

    traj = copy.deepcopy(self.default_joint_trajectory)

    for _ in range(0, 2):
        pts = trajectory_msgs.msg.JointTrajectoryPoint()
        pts.positions = joint_states
        pts.velocities = [0, 0, 0, 0, 0, 0]
        pts.time_from_start = rospy.Time(duration)
        # Set the points to the trajectory
        traj.points = [pts]
        # Publish the message
        self.joints_pub.publish(traj)

    def wait_for_position(self, timeout=2, tol_pos=0.01, tol_vel=0.01):
        end = rospy.Time.now() + rospy.Duration(timeout)
        while rospy.Time.now() < end:
```



```
msg = get_controller_state(self.controller_topic, timeout=10)
v = np.sum(np.abs(msg.actual.velocities), axis=0)
if v < tol_vel:
    for actual, desired in zip(msg.actual.positions, msg.desired.positions):
        if abs(actual - desired) > tol_pos:
            break
    return
rospy.logwarn("Timeout waiting for position")
```

### 6.2.3 motion\_planning.py

Module này chịu trách nhiệm trực tiếp trong quá trình phân loại các miếng lego  
Luồng xử lí chính của cánh tay sẽ thực hiện các công đoạn sau:

1. Khai báo các lớp và thuộc tính để có thể điều khiển cánh tay.

```
rospy.init_node("send_joints")

controller = ArmController()

# Create an action client for the gripper
action_gripper = actionlib.SimpleActionClient(
    "/gripper_controller/gripper_cmd",
    control_msgs.msg.GripperCommandAction
)
print("Waiting for action of gripper controller")
action_gripper.wait_for_server()

setstatic_srv = rospy.ServiceProxy("/link_attacher_node/setstatic", SetStatic)
attach_srv = rospy.ServiceProxy("/link_attacher_node/attach", Attach)
detach_srv = rospy.ServiceProxy("/link_attacher_node/detach", Attach)
setstatic_srv.wait_for_service()
attach_srv.wait_for_service()
detach_srv.wait_for_service()
```

2. Chờ nhận được vị trí và góc của tất cả miếng lego hiện có trên bàn.

```
controller.move_to(*DEFAULT_POS, DEFAULT_QUAT)

print("Waiting for detection of the models")
rospy.sleep(0.5)
legos = get_legos_pos(vision=True)
legos.sort(reverse=True, key=lambda a: (a[1].position.x, a[1].position.y))
```

3. Gấp các miếng lego đến vị trí cho sẵn.

```
for model_name, model_pose in legos:
    open_gripper()
    try:
        model_home = MODELS_INFO[model_name]["home"]
        model_size = MODELS_INFO[model_name]["size"]
    except ValueError as e:
        print(f"Model name {model_name} was not recognized!")
        continue

    # Get actual model_name at model xyz coordinates
    try:
```



```
gazebo_model_name = get_gazebo_model_name(model_name, model_pose)
except ValueError as e:
    print(e)
    continue

# Straighten lego
straighten(model_pose, gazebo_model_name)
controller.move(dz=0.15)

"""
    Go to destination
"""
x, y, z = model_home
z += model_size[2] / 2 +0.004
print(f"Moving model {model_name} to {x} {y} {z}")

controller.move_to(x, y, target_quat=DEFAULT_QUAT * PyQuaternion(axis=[0, 0, 1],
    angle=math.pi / 2))
# Lower the object and release
controller.move_to(x, y, z)
set_model_fixed(gazebo_model_name)
open_gripper(gazebo_model_name)
controller.move(dz=0.15)

if controller.gripper_pose[0][1] > -0.3 and controller.gripper_pose[0][0] > 0:
    controller.move_to(*DEFAULT_POS, DEFAULT_QUAT)

# increment z in order to stack lego correctly
MODELS_INFO[model_name]["home"][2] += model_size[2] - INTERLOCKING_OFFSET
```

#### 4. Quay về trạng thái mặc định của cánh tay

```
print("Moving to Default Position")
controller.move_to(*DEFAULT_POS, DEFAULT_QUAT)
open_gripper()
rospy.sleep(0.4)
```

Các hàm và công đoạn phụ trợ: Tạo một dictionary chứa vị trí và kích cỡ của các miếng lego.

```
MODELS_INFO = {
    "X1-Y2-Z1": {
        "home": [0.264589, -0.293903, 0.777]
    },
    "X2-Y2-Z2": {
        "home": [0.277866, -0.724482, 0.777]
    },
    "X1-Y3-Z2": {
        "home": [0.268053, -0.513924, 0.777]
    },
    "X1-Y2-Z2": {
        "home": [0.429198, -0.293903, 0.777]
    },
    "X1-Y2-Z2-CHAMFER": {
        "home": [0.592619, -0.293903, 0.777]
    },
    "X1-Y4-Z2": {
        "home": [0.108812, -0.716057, 0.777]
    },
}
```



```
"X1-Y1-Z2": {
    "home": [0.088808, -0.295820, 0.777]
},
"X1-Y2-Z2-TWINFILLET": {
    "home": [0.103547, -0.501132, 0.777]
},
"X1-Y3-Z2-FILLET": {
    "home": [0.433739, -0.507130, 0.777]
},
"X1-Y4-Z1": {
    "home": [0.589908, -0.501033, 0.777]
},
"X2-Y2-Z2-FILLET": {
    "home": [0.442505, -0.727271, 0.777]
}
}

for model, info in MODELS_INFO.items():
    model_json_path = os.path.join(PKG_PATH, "...", "models", f"lego_{model}",
        "model.json")
    # make path absolute
    model_json_path = os.path.abspath(model_json_path)
    # check path exists
    if not os.path.exists(model_json_path):
        raise FileNotFoundError(f"Model file {model_json_path} not found")

    model_json = json.load(open(model_json_path, "r"))
    corners = np.array(model_json["corners"])

    size_x = (np.max(corners[:, 0]) - np.min(corners[:, 0]))
    size_y = (np.max(corners[:, 1]) - np.min(corners[:, 1]))
    size_z = (np.max(corners[:, 2]) - np.min(corners[:, 2]))

    #print(f"{model}: {size_x:.3f} x {size_y:.3f} x {size_z:.3f}")

    MODELS_INFO[model][ "size"] = (size_x, size_y, size_z)
```

Chờ nhận được dữ liệu vị trí và góc xoay của những miếng lego.

```
def get_legos_pos(vision=False):
    #get legos position reading vision topic
    if vision:
        legos = rospy.wait_for_message("/lego_detections", ModelStates, timeout=None)
    else:
        models = rospy.wait_for_message("/gazebo/model_states", ModelStates, timeout=None)
        legos = ModelStates()

        for name, pose in zip(models.name, models.pose):
            if "X" not in name:
                continue
            name = get_model_name(name)

            legos.name.append(name)
            legos.pose.append(pose)
    return [(lego_name, lego_pose) for lego_name, lego_pose in zip(legos.name,
        legos.pose)]
```

Dóng mở gripper theo kích cỡ lego đã biết.

```
def set_gripper(value):
```



```
goal = control_msgs.msg.GripperCommandGoal()
goal.command.position = value # From 0.0 to 0.8
goal.command.max_effort = -1 # # Do not limit the effort
action_gripper.send_goal_and_wait(goal, rospy.Duration(10))

return action_gripper.get_result()

def close_gripper(gazebo_model_name, closure=0):
    set_gripper(0.81-closure*10)
    rospy.sleep(0.5)
    # Create dynamic joint
    if gazebo_model_name is not None:
        req = AttachRequest()
        req.model_name_1 = gazebo_model_name
        req.link_name_1 = "link"
        req.model_name_2 = "robot"
        req.link_name_2 = "wrist_3_link"
        attach_srv.call(req)

def open_gripper(gazebo_model_name=None):
    set_gripper(0.0)

    # Destroy dynamic joint
    if gazebo_model_name is not None:
        req = AttachRequest()
        req.model_name_1 = gazebo_model_name
        req.link_name_1 = "link"
        req.model_name_2 = "robot"
        req.link_name_2 = "wrist_3_link"
        detach_srv.call(req)
```

Di tới vị trí của miếng lego, chỉnh sửa góc xoay của gripper cho đúng hướng của lego, nếu miếng lego đang không trong trạng thái hướng lên thì chỉnh sửa lại(tất cả được thực thi bằng hàm straighten).

```
def get_approach_quat(facing_direction, approach_angle):
    quat = DEFAULT_QUAT
    if facing_direction == (0, 0, 1):
        pitch_angle = 0
        yaw_angle = 0
    elif facing_direction == (1, 0, 0) or facing_direction == (0, 1, 0):
        pitch_angle = + 0.2
        if abs(approach_angle) < math.pi/2:
            yaw_angle = math.pi/2
        else:
            yaw_angle = -math.pi/2
    elif facing_direction == (0, 0, -1):
        pitch_angle = 0
        yaw_angle = 0
    else:
        raise ValueError(f"Invalid model state {facing_direction}")

    quat = quat * PyQuaternion(axis=(0, 1, 0), angle=pitch_angle)
    quat = quat * PyQuaternion(axis=(0, 0, 1), angle=yaw_angle)
    quat = PyQuaternion(axis=(0, 0, 1), angle=approach_angle+math.pi/2) * quat

return quat
```



```
def get_axis_facing_camera(quat):
    axis_x = np.array([1, 0, 0])
    axis_y = np.array([0, 1, 0])
    axis_z = np.array([0, 0, 1])
    new_axis_x = quat.rotate(axis_x)
    new_axis_y = quat.rotate(axis_y)
    new_axis_z = quat.rotate(axis_z)
    # get angle between new_axis and axis_z
    angle = np.arccos(np.clip(np.dot(new_axis_z, axis_z), -1.0, 1.0))
    # get if model is facing up, down or sideways
    if angle < np.pi / 3:
        return 0, 0, 1
    elif angle < np.pi / 3 * 2 * 1.2:
        if abs(new_axis_x[2]) > abs(new_axis_y[2]):
            return 1, 0, 0
        else:
            return 0, 1, 0
    #else:
    #    raise Exception(f"Invalid axis {new_axis_x}")
    else:
        return 0, 0, -1

def get_approach_angle(model_quat, facing_direction):#get gripper approach angle
    if facing_direction == (0, 0, 1):
        return model_quat.yaw_pitch_roll[0] - math.pi/2 #rotate gripper
    elif facing_direction == (1, 0, 0) or facing_direction == (0, 1, 0):
        axis_x = np.array([0, 1, 0])
        axis_y = np.array([-1, 0, 0])
        new_axis_z = model_quat.rotate(np.array([0, 0, 1])) #get z axis of lego
        # get angle between new_axis and axis_x
        dot = np.clip(np.dot(new_axis_z, axis_x), -1.0, 1.0) #sin angle between lego z
        # axis and x axis in fixed frame
        det = np.clip(np.dot(new_axis_z, axis_y), -1.0, 1.0) #cos angle between lego z
        # axis and x axis in fixed frame
        return math.atan2(det, dot) #get angle between lego z axis and x axis in fixed
        # frame
    elif facing_direction == (0, 0, -1):
        return -(model_quat.yaw_pitch_roll[0] - math.pi/2) % math.pi - math.pi
    else:
        raise ValueError(f"Invalid model state {facing_direction}")

def straighten(model_pose, gazebo_model_name):
    x = model_pose.position.x
    y = model_pose.position.y
    z = model_pose.position.z
    model_quat = PyQuaternion(
        x=model_pose.orientation.x,
        y=model_pose.orientation.y,
        z=model_pose.orientation.z,
        w=model_pose.orientation.w)

    model_size = MODELS_INFO[get_model_name(gazebo_model_name)]["size"]

    """
    Calculate approach quaternion and target quaternion
    """

    facing_direction = get_axis_facing_camera(model_quat)
    approach_angle = get_approach_angle(model_quat, facing_direction)
```



```
print(f"Leg is facing {facing_direction}")
print(f"Angle of approaching measures {approach_angle:.2f} deg")

# Calculate approach quat
approach_quat = get_approach_quat(facing_direction, approach_angle)

# Get above the object
controller.move_to(x, y, target_quat=approach_quat)

# Calculate target quat
regrip_quat = DEFAULT_QUAT
if facing_direction == (1, 0, 0) or facing_direction == (0, 1, 0): # Side
    target_quat = DEFAULT_QUAT
    pitch_angle = -math.pi/2 + 0.2

    if abs(approach_angle) < math.pi/2:
        target_quat = target_quat * PyQuaternion(axis=(0, 0, 1), angle=math.pi/2)
    else:
        target_quat = target_quat * PyQuaternion(axis=(0, 0, 1), angle=-math.pi/2)
    target_quat = PyQuaternion(axis=(0, 1, 0), angle=pitch_angle) * target_quat

    if facing_direction == (0, 1, 0):
        regrip_quat = PyQuaternion(axis=(0, 0, 1), angle=math.pi/2) * regrip_quat

elif facing_direction == (0, 0, -1):
    """
    Pre-positioning
    """
    controller.move_to(z=z, target_quat=approach_quat)
    close_gripper(gazebo_model_name, model_size[0])

    tmp_quat = PyQuaternion(axis=(0, 0, 1), angle=2*math.pi/6) * DEFAULT_QUAT
    controller.move_to(SAFE_X, SAFE_Y, z+0.05, target_quat=tmp_quat, z_raise=0.1) #
        Move to safe position
    controller.move_to(z=z)
    open_gripper(gazebo_model_name)

    approach_quat = tmp_quat * PyQuaternion(axis=(1, 0, 0), angle=math.pi/2)

    target_quat = approach_quat * PyQuaternion(axis=(0, 0, 1), angle=-math.pi) # Add a
        yaw rotation of 180 deg

    regrip_quat = tmp_quat * PyQuaternion(axis=(0, 0, 1), angle=math.pi)
else:
    target_quat = DEFAULT_QUAT
    target_quat = target_quat * PyQuaternion(axis=(0, 0, 1), angle=-math.pi/2)

"""
Grip the model
"""
if facing_direction == (0, 0, 1) or facing_direction == (0, 0, -1):
    closure = model_size[0]
    z = SURFACE_Z + model_size[2] / 2
elif facing_direction == (1, 0, 0):
    closure = model_size[1]
    z = SURFACE_Z + model_size[0] / 2
elif facing_direction == (0, 1, 0):
    closure = model_size[0]
    z = SURFACE_Z + model_size[1] / 2
controller.move_to(z=z, target_quat=approach_quat)
close_gripper(gazebo_model_name, closure)
```



```
"""
    Straighten model if needed
"""

if facing_direction != (0, 0, 1):
    z = SURFACE_Z + model_size[2]/2

    controller.move_to(z=z+0.05, target_quat=target_quat, z_raise=0.1)
    controller.move(dz=-0.05)
    open_gripper(gazebo_model_name)

    # Re grip the model
    controller.move_to(z=z, target_quat=regrip_quat, z_raise=0.1)
    close_gripper(gazebo_model_name, model_size[0])
```

Các hàm thay đổi tên cho mục đích trích xuất dữ liệu và mô phỏng trong gazebo.

```
def get_gazebo_model_name(model_name, vision_model_pose):
    """
        Get the name of the model inside gazebo. It is needed for link attacher plugin.
    """

    models = rospy.wait_for_message("/gazebo/model_states", ModelStates, timeout=None)
    epsilon = 0.05
    for gazebo_model_name, model_pose in zip(models.name, models.pose):
        if model_name not in gazebo_model_name:
            continue
        # Get everything inside a square of side epsilon centered in vision_model_pose
        ds = abs(model_pose.position.x - vision_model_pose.position.x) +
             abs(model_pose.position.y - vision_model_pose.position.y)
        if ds <= epsilon:
            return gazebo_model_name
    raise ValueError(f"Model {model_name} at position {vision_model_pose.position.x}\n{vision_model_pose.position.y} was not found!")

def get_model_name(gazebo_model_name):
    return gazebo_model_name.replace("lego_", "").split("_", maxsplit=1)[0]
```

Cố định những miếng lego vào bề mặt bàn(trong mô phỏng gazebo).

```
def set_model_fixed(model_name):
    req = AttachRequest()
    req.model_name_1 = model_name
    req.link_name_1 = "link"
    req.model_name_2 = "ground_plane"
    req.link_name_2 = "link"
    attach_srv.call(req)

    req = SetStaticRequest()
    print("{} TO HOME".format(model_name))
    req.model_name = model_name
    req.link_name = "link"
    req.set_static = True

    setstatic_srv.call(req)
```



## 7 Kết luận và hướng phát triển

### 7.1 Kết luận

#### 7.1.1 Đạt được

- Xây dựng, huấn luyện được mô hình phân loại và xác định vị trí, hướng của vật thể thông qua xử lý ảnh bằng thư viện OpenCV
- Điều khiển được cánh tay robot gấp thả đúng vị trí mong muốn.
- Khi có nhiều vật thể, sau khi xử lý cánh tay robot có thể phân loại liên tục cho đến khi hết vật thể.

#### 7.1.2 Hạn chế

- Tập dữ liệu chưa đủ lớn dẫn đến mô hình phân loại được huấn luyện chưa tối ưu (bounding box lệch).
- Cánh tay robot chỉ có thể gấp thả các vật thể tĩnh.
- Hệ thống vẫn cần có sự can thiệp của con người khi gấp lối.

### 7.2 Hướng phát triển

- Tăng cường dữ liệu huấn luyện: Dữ liệu huấn luyện là yếu tố quan trọng nhất quyết định đến hiệu quả của mô hình. Do đó, cần tăng cường dữ liệu huấn luyện bằng cách:
  - Tăng số lượng mẫu dữ liệu.
  - Tăng sự đa dạng của các mẫu dữ liệu, bao gồm các mẫu dữ liệu có độ phân giải cao, các mẫu dữ liệu có góc nhìn khác nhau, v.v.
- Sử dụng các kỹ thuật tăng cường dữ liệu (data augmentation): Các kỹ thuật tăng cường dữ liệu có thể giúp tạo ra các mẫu dữ liệu mới từ các mẫu dữ liệu hiện có, giúp tăng cường sự đa dạng và độ khó của dữ liệu huấn luyện.
- Thay đổi mô hình huấn luyện: Sử dụng các kiến trúc mạng thần kinh mới, hiệu quả hơn. Tối ưu hóa các tham số của mô hình.
- Cải thiện hệ thống điều khiển đảm bảo tính chính xác, ổn định và khả năng thích ứng với các điều kiện thực tế. Có thể dùng để đóng gói, lắp ráp.
- Tự động hóa, phân loại vật thể chuyển động là một lĩnh vực nghiên cứu đầy tiềm năng với nhiều ứng dụng thực tế. Đồng thời đảm bảo tính an toàn có khả năng tự tắt khi gặp lỗi.



## 8 Tài liệu tham khảo