Project Report

Sensory-Driven Vision-Based Control for Robotic Manipulation

Date: 2026-01-17 11:00

Generated by: Automated Reporting System

Table of Contents

1. Introduction

This project implements a sensory-driven vision-based control system for robotic manipulation using a UR5 robot in a Gazebo simulation environment. The system integrates computer vision using YOLOv5 for object detection and pose estimation, with a motion planning framework capable of executing precise pick-and-place tasks.

The primary objective is to demonstrate robust control strategies, comparing kinematic trajectory planning with Model Predictive Control (MPC) approaches. The system handles dynamic environments where objects (LEGO bricks) are identified, localized, and manipulated to build specific structures.

2. Methodology

2.1 Vision System

The vision system utilizes an RGB-D camera mounted in the simulation. A YOLOv5 model, trained on custom datasets, detects LEGO bricks and estimates their orientation. Depth data is used to compute the precise 3D coordinates of the objects relative to the robot base.

2.2 Present Controller

The system utilizes a custom Kinematic Trajectory Controller implemented in the 'ArmController' class. This controller operates by calculating the exact joint configurations required to achieve a target Cartesian pose using an analytical Inverse Kinematics (IK) solver. It interfaces with the standard ROS JointTrajectoryController to execute motion commands, ensuring precise geometric positioning.

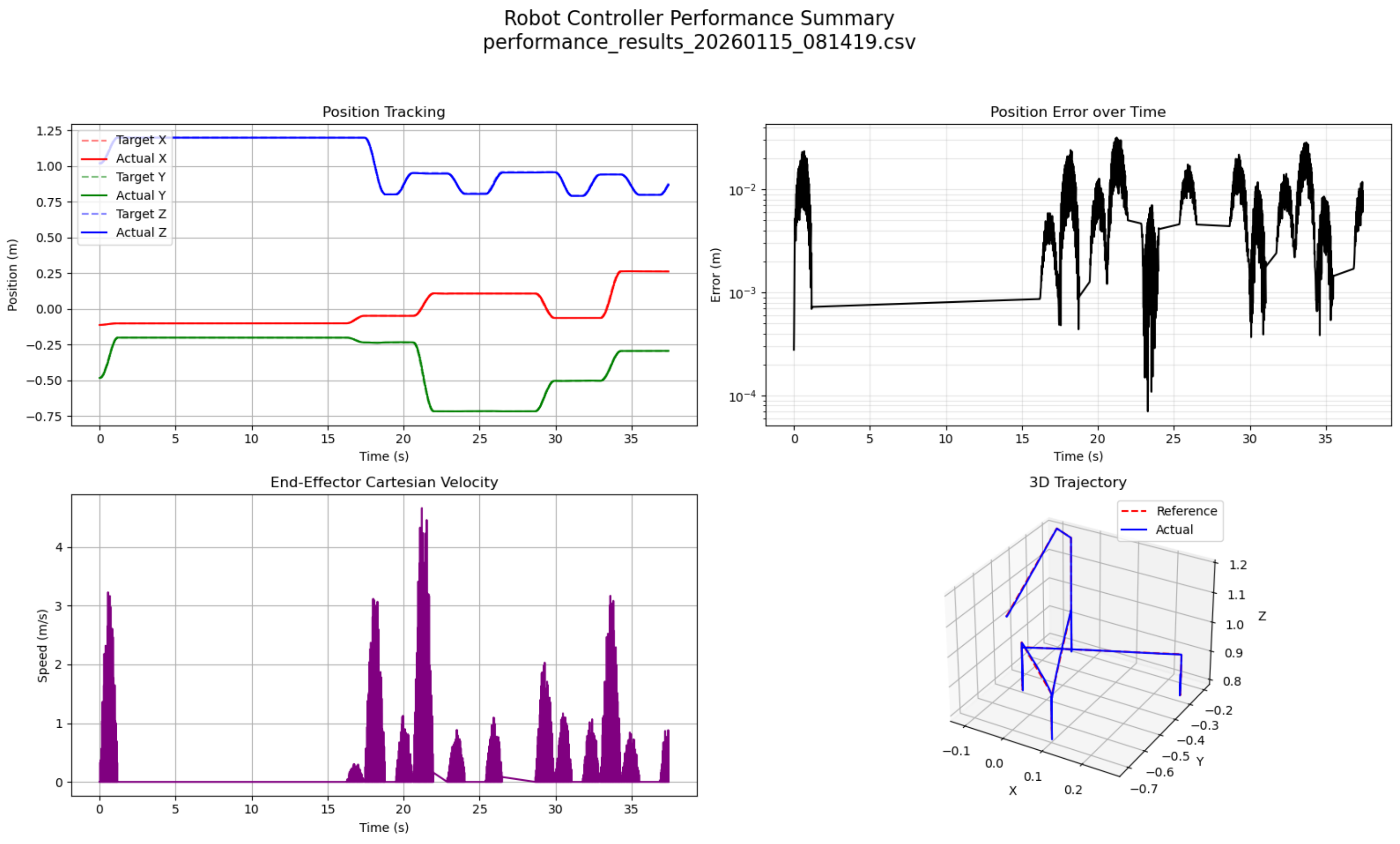
2.3 Path Planning Strategy

Path planning is performed using Linear Cartesian Interpolation. The controller computes a straight-line path in 3D space from the start to the goal position. This path is discretized into fine steps, and a cosine-based smoothing function is applied to the interpolation parameter. This technique generates a velocity profile with smooth acceleration and deceleration, minimizing jerk and ensuring stable motion during pick-and-place operations.

3. Experimental Results

The following section presents the quantitative results from the latest simulation run.

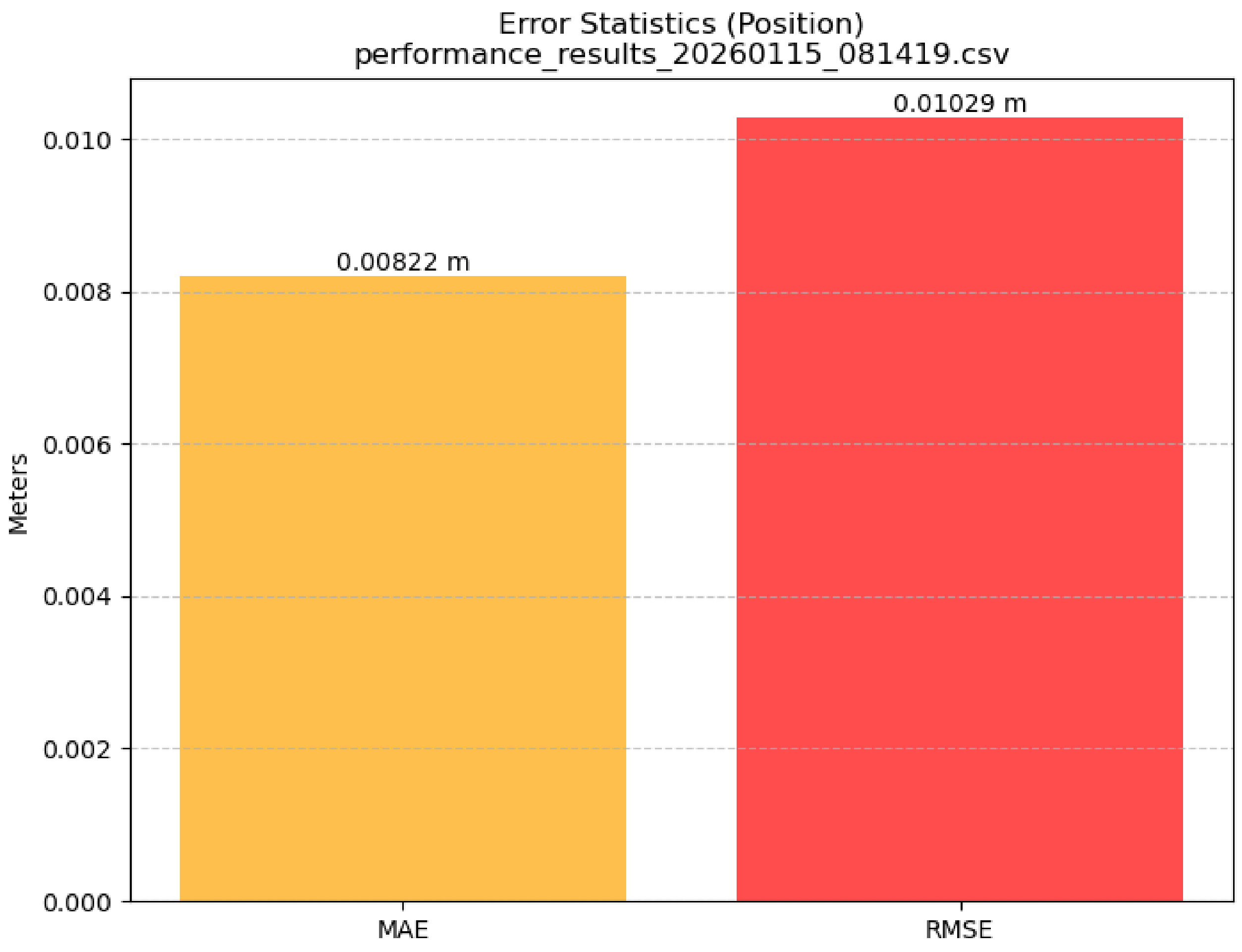
3.1 Trajectory Summary



The plot above shows the 3D trajectory of the end-effector. The alignment between the reference path (dashed red) and the actual path (solid blue) indicates the tracking accuracy of the controller.

This graph illustrates the velocity profiles of the six UR5 joints during the operation. Smooth velocity curves suggest stable control without oscillations or jerky movements.

3.3 Error Statistics



The Mean Absolute Error (MAE) and Root Mean Square Error (RMSE) provide a summary of the positioning accuracy. Lower values indicate higher precision in the manipulation task.

4. Data Logs (Sample

```
time          target_x      actual_x      error_dist
1.1620        -0.1113       -0.1113       0.0003
1.1690        -0.1113       -0.1113       0.0003
1.1750        -0.1113       -0.1113       0.0004
1.1810        -0.1113       -0.1113       0.0006
1.1870        -0.1112       -0.1113       0.0008
1.1930        -0.1112       -0.1113       0.0011
1.2000        -0.1112       -0.1113       0.0014
1.2070        -0.1112       -0.1112       0.0032
1.2140        -0.1112       -0.1112       0.0035
1.2210        -0.1112       -0.1112       0.0038
1.2280        -0.1112       -0.1112       0.0042
1.2350        -0.1111       -0.1112       0.0047
1.2410        -0.1111       -0.1112       0.0052
1.2480        -0.1111       -0.1109       0.0037
1.2540        -0.1111       -0.1109       0.0042
1.2600        -0.1110       -0.1109       0.0049
1.2680        -0.1110       -0.1109       0.0056
1.2740        -0.1110       -0.1109       0.0064
1.2820        -0.1110       -0.1106       0.0032
1.2880        -0.1109       -0.1106       0.0039
1.2940        -0.1109       -0.1106       0.0048
1.3000        -0.1109       -0.1106       0.0058
1.3060        -0.1108       -0.1106       0.0068
1.3130        -0.1108       -0.1106       0.0079
1.3200        -0.1107       -0.1106       0.0091
1.3260        -0.1107       -0.1103       0.0045
1.3330        -0.1107       -0.1103       0.0057
1.3390        -0.1106       -0.1103       0.0069
1.3450        -0.1106       -0.1103       0.0082
1.3510        -0.1105       -0.1103       0.0096
1.3580        -0.1105       -0.1103       0.0111
1.3640        -0.1104       -0.1099       0.0048
1.3710        -0.1104       -0.1099       0.0062
1.3780        -0.1103       -0.1099       0.0077
1.3870        -0.1102       -0.1099       0.0093
1.3940        -0.1102       -0.1099       0.0110
1.4000        -0.1101       -0.1099       0.0127
1.4070        -0.1101       -0.1096       0.0055
1.4130        -0.1100       -0.1096       0.0072
1.4190        -0.1099       -0.1096       0.0091
```

(Showing first 40 rows of 1891 total records)

Appendix A: Source Code

The following pages contain the complete source code used in this project.

File: motion_planning.py

```python
#!/usr/bin/python3

import os
import math
import copy
import json
import actionlib
import control_msgs.msg
from controller import ArmController
from gazebo_msgs.msg import ModelStates
import rospy
from pyquaternion import Quaternion as PyQuaternion
import numpy as np
from gazebo_ros_link_attacher.srv import SetStatic, SetStaticRequest, SetStaticResponse
from gazebo_ros_link_attacher.srv import Attach, AttachRequest, AttachResponse

PKG_PATH = os.path.dirname(os.path.abspath(__file__))

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, model_info in MODELS_INFO.items():
    pass
    # MODELS_INFO[model]["home"] = model_info["home"] + np.array([0.0, 0.10, 0.0])

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)

# Compensate for the interlocking height
INTERLOCKING_OFFSET = 0.019

SAFE_X = -0.40
SAFE_Y = -0.13
SURFACE_Z = 0.774

# Resting orientation of the end effector
DEFAULT_QUAT = PyQuaternion(axis=(0, 1, 0), angle=math.pi)
# Resting position of the end effector
DEFAULT_POS = (-0.1, -0.2, 1.2)

DEFAULT_PATH_TOLERANCE = control_msgs.msg.JointTolerance()
DEFAULT_PATH_TOLERANCE.name = "path_tolerance"
DEFAULT_PATH_TOLERANCE.velocity = 10


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} {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]


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)
    ]


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"Lego 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])


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)
```

```
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)


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 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()


if __name__ == "__main__":
    print("Initializing node of kinematics")
    rospy.init_node("send_joints")

    # Use MoveIt controller to handle obstacles
    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()

    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))

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

File: controller.py

```python
import math
import copy
import rospy
import numpy as np
import kinematics
import control_msgs.msg
import trajectory_msgs.msg
import geometry_msgs.msg
from pyquaternion import Quaternion
import pandas as pd
import datetime
import time

# Force update check
rospy.loginfo("Loading controller module...")


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)

        # State tracking for logging
        self.current_joint_state = None
        self.state_sub = rospy.Subscriber(
            f"{self.controller_topic}/state",
            control_msgs.msg.JointTrajectoryControllerState,
            self._state_cb,
        )

        # 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,
        )

        # Logging
        self.log_data = []
        rospy.on_shutdown(self.save_logs)

    def _state_cb(self, msg):
        self.current_joint_state = msg

    def save_logs(self):
        if not self.log_data:
            return
        try:
            df = pd.DataFrame(self.log_data)
            timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
            filename = f"/root/catkin_ws/performance_results_{timestamp}.csv"
            df.to_csv(filename, index=False)
            rospy.loginfo(f"Performance Data saved to {filename}")
        except Exception as e:
            rospy.logerr(f"Failed to save logs: {e}")
```

```
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
    :param x:
    :param y:
    :param z:
    :param start_quat:
    :param target_pos:
    :param target_quat:
    :param z_raise:
    :param blocking:
    :return:
    """

    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)

        # Calculate intermediate targets
        tx = sx + i_1 * dx
        ty = sy + i_1 * dy
        tz = sz + i_1 * dz + i_2 * z_raise

        t_start = time.time()
        self.send_joints(
            tx,
            ty,
            tz,
            grip,
            duration=1 / speed * 0.9,
        )
        comp_time = time.time() - t_start

        self._record_state(tx, ty, tz, comp_time)
        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 _record_state(self, tx, ty, tz, comp_time=0.0):
    if self.current_joint_state:
        try:
            ax, ay, az, _ = kinematics.get_pose(
                self.current_joint_state.actual.positions
            )

            data = {
                "time": rospy.Time.now().to_sec(),
                "target_x": tx,
                "target_y": ty,
                "target_z": tz,
                "actual_x": ax,
                "actual_y": ay,
                "actual_z": az,
                "error_dist": np.sqrt(
                    (tx - ax) ** 2 + (ty - ay) ** 2 + (tz - az) ** 2
                ),
                "computation_time": comp_time,
            }

            # Log joint velocities if available
            if (
                hasattr(self.current_joint_state.actual, "velocities")
                and len(self.current_joint_state.actual.velocities) > 0
            ):
                for i, v in enumerate(self.current_joint_state.actual.velocities):
                    data[f"v{i+1}"] = v

            self.log_data.append(data)
        except Exception:
            pass

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")
```

File: lego-vision.py

```python
#! /usr/bin/env python3

import cv2 as cv
import numpy as np
import torch
import message_filters
import rospy
import sys
import time
import os

from sensor_msgs.msg import Image
from cv_bridge import CvBridge
from rospkg import RosPack  # get abs path
from os import path  # get home path
from gazebo_msgs.msg import ModelStates
from geometry_msgs.msg import *
from pyquaternion import Quaternion as PyQuaternion

# Global variables
path_yolo = path.join(path.expanduser("~"), "yolov5")
path_vision = RosPack().get_path("vision")
path_weigths = path.join(path_vision, "weigths")
path_sounds = path.join(path_vision, "sounds")

cam_point = (-0.44, -0.5, 1.58)
height_tavolo = 0.74
dist_tavolo = None
origin = None
model = None
model_orientation = None

legoClasses = [
    "X1-Y1-Z2",
    "X1-Y2-Z1",
    "X1-Y2-Z2",
    "X1-Y2-Z2-CHAMFER",
    "X1-Y2-Z2-TWINFILLET",
    "X1-Y3-Z2",
    "X1-Y3-Z2-FILLET",
    "X1-Y4-Z1",
    "X1-Y4-Z2",
    "X2-Y2-Z2",
    "X2-Y2-Z2-FILLET",
]

argv = sys.argv
a_show = "-show" in argv

# Utility Functions


def get_dist_tavolo(depth, hsv, img_draw):
    global dist_tavolo

    # color = (120,1,190)
    # mask = get_lego_mask(color, hsv, (5, 5, 5))
    # dist_tavolo = depth[mask].max()
    # if dist_tavolo > 1: dist_tavolo -= height_tavolo
    dist_tavolo = np.nanmax(depth)


def get_origin(img):
    global origin
    origin = np.array(img.shape[1::-1]) // 2


def get_lego_distance(depth):
    return depth.min()


def get_lego_color(center, rgb):
    return rgb[center].tolist()


def get_lego_mask(color, hsv, toll=(20, 20, 255)):
    thresh = np.array(color)
    mintoll = thresh - np.array([toll[0], toll[1], min(thresh[2] - 1, toll[2])])
    maxtoll = thresh + np.array(toll)
    return cv.inRange(hsv, mintoll, maxtoll)


def getDepthAxis(height, lego):
    X, Y, Z = (int(x) for x in lego[1:8:3])
    # Z = (0.038, 0.057) X = (0.031, 0.063) Y = (0.031, 0.063, 0.095, 0.127)
    rapZ = height / 0.019 - 1
    pinZ = round(rapZ)
    rapXY = height / 0.032
    pinXY = round(rapXY)
    errZ = abs(pinZ - rapZ) + max(pinZ - 2, 0)
    errXY = abs(pinXY - rapXY) + max(pinXY - 4, 0)

    if errZ < errXY:
        return pinZ, 2, pinZ == Z  # pin, is ax Z, match
    else:
        if pinXY == Y:
            return pinXY, 1, True
        else:
            return pinXY, 0, pinXY == X


def point_distorption(point, height, origin):
    p = dist_tavolo / (dist_tavolo - height)
    point = point - origin
    return p * point + origin


def point_inverse_distortption(point, height):
    p = dist_tavolo / (dist_tavolo - height)
    point = point - origin
    return point / p + origin


def myimshow(title, img):
    def mouseCB(event, x, y, a, b):
        print(x, y, img[y, x], "\r", end="", flush=True)
        print("\033[K", end="")

    cv.imshow(title, img)
    cv.setMouseCallback(title, mouseCB)
    cv.waitKey()


# ----------------- LOCALIZATION ----------------- #


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.int32((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)

    # myimshow("asda", hsv)
    # getting lego height from camera and table
    l_dist = get_lego_distance(l_depth)
    l_height = dist_tavolo - l_dist
```

```
# masking
l_top_mask = cv.inRange(l_depth, l_dist - 0.002, l_dist + 0.002)
# cv.bitwise_xor(img_draw,img_draw,img_draw, mask=cv.inRange(depth, l_dist-0.002, l_dist+0.002))
# myimshow("hmask", l_top_mask)

# 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
# print("Model orientation: Start...", end='\r')
model_orientation.conf = 0.7
results = model_orientation(depth_image)
pandino = []
pandino = results.pandas().xyxy[0].to_dict(orient="records")

n = len(pandino)
# print("Model orientation: Finish", n)

# Adjust prediction
pinN, ax, isCorrect = getDepthAxis(l_height, nm)
if not isCorrect and ax == 2:
    if cl in (1, 2, 3) and pinN in (1, 2):  # X1-Y2-Z*, *1,2,2-CHAMFER
        cl = 1 if pinN == 1 else 2  # -> Z'pinN'
    elif cl in (7, 8) and pinN in (1, 2):  # X1-Y4-Z*, *1,2
        cl = 7 if pinN == 1 else 8  # -> Z'pinN'
    elif pinN == -1:
        nm = "{} -> {}".format(nm, "Target")
    else:
        print("[Warning] Error in classification")
elif not isCorrect:
    ax = 1
    if cl in (0, 2, 5, 8) and pinN <= 4:  # X1-Y*-Z2, *1,2,3,4
        cl = (2, 5, 8)[pinN - 2]  # -> Y'pinN'
    elif cl in (1, 7) and pinN in (2, 4):  # X1-Y*-Z1, *2,4
        cl = 1 if pinN == 2 else 7  # -> Y'pinN'
    elif cl == 9 and pinN == 1:  # X2-Y2-Z2
        cl = 2  # -> X1
        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
# ---

# 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.int32([l_box]), 0, (0, 0, 0), 2)
    return None  # filter out artifacts

if a_show:
    cv.drawContours(img_draw, np.int32([l_box]), 0, color, 2)

# silouette distorption
# 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]
# distorping closest vertex
if or_nm == "sopra":
    l_height -= 0.019
top_box[iver] = point_distorption(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.int32([top_box]), 0, (5, 5, 5), 2)
    cv.circle(img_draw, np.int32(top_box[iver]), 1, (0, 0, 255), 1, cv.LINE_AA)

# 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][:2]
    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
    ][:2]

    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.int32(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.int32(edgePin), (255, 255, 0), 2)

l_center = point_inverse_distortption(l_center, l_height)

# post rotation extra
theta = 0
if cl == 1 and ax == 1:
    theta = 1.715224 - np.pi / 2
if cl == 3 and or_nm == "sotto":
    theta = 2.359515 - np.pi
if cl == 4 and ax == 1:
    theta = 2.145295 - np.pi
if cl == 6 and or_nm == "sotto":
    theta = 2.645291 - np.pi
if cl == 10 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.int32(l_center)

    origin_from_top = origin - l_center

    endX = point_distorption(lenFrame * dirX[:2], x_to_z * dirX[2], origin_from_top)
    frameX = (center, center + np.int32(endX))

    endY = point_distorption(lenFrame * dirY[:2], x_to_z * dirY[2], origin_from_top)
    frameY = (center, center + np.int32(endY))

    endZ = point_distorption(lenFrame * dirZ[:2], x_to_z * dirZ[2], origin_from_top)
    frameZ = (center, center + np.int32(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


# image processing
def process_image(rgb, depth):

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

    get_dist_tavolo(depth, hsv, img_draw)
    get_origin(rgb)

    # results collecting localization

    # print("Model localization: Start...",end='\r')
    model.conf = 0.6
    results = model(rgb)
    pandino = results.pandas().xyxy[0].to_dict(orient="records")
    # print("Model localization: Finish  ")

    # ----
    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)

    # Play a sound if any legos were detected
    if len(msg.name) > 0:
        sound_file = path.join(
            path_sounds, "success.wav"
        )  # Assuming you have a success.wav
        if path.exists(sound_file):
            os.system(f"paplay {sound_file}")

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

    pass


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


# init node function
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


def load_models():
    global model, model_orientation

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

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


if __name__ == "__main__":

    load_models()
    try:
        start_node()
    except rospy.ROSInterruptException:
        pass
```