Documentation and Contribution Report

Module 1: System Requirements

Contributor: Mohammad Aswad **Mart Number:** 6116264

1. Embodiedness (The Robotic Agent)

The robot's physical embodiment dictates its configuration space (joint states) and how it can interact with the task space. It is a custom-designed wheeled mobile manipulator with the following specifications:

2.1. Locomotion & Base

- **Chassis:** The robot features a rectangular base measuring 0.5m x 0.4m x 0.1m.
- **Mass Distribution:** The base has a heavy mass of 40 kg. This acts as a critical stabilizing counterweight, preventing the robot from tipping over when its arm is fully extended in the task space.
- **Drive System:** The robot utilizes a four-wheel differential drive system. Each wheel has a radius of 0.1m and a thickness of 0.05m. The wheel dynamics are tuned with a lateral friction coefficient of 1.5 to ensure high traction and minimize slipping during locomotion.

2.2. Manipulation (Kinematic Chain)

The robot is equipped with a lifting mechanism and an articulated arm designed to perform table-height manipulation:

- **Arm Mount:** The arm assembly is mounted to the torso on a carriage using a fixed joint.
- **Arm Architecture:** The arm features multiple revolute joints including an arm base (yaw), a shoulder (pitch), an elbow (pitch), a wrist pitch and a wrist roll. This provides the necessary Degrees of Freedom (DoF) to reach the target configuration.
- **End-Effector (Gripper):** The wrist terminates in a parallel-jaw gripper. The left and right fingers operate on prismatic joints, each with a travel limit of 0.04m, allowing the robot to grasp objects up to 0.08m wide.

2.3. Sensor Suite

Following the principles of sensor classification, the robot is equipped with the following:

- **Exteroceptive, Non-Contact Sensors:**
  - An RGB-D Camera is rigidly mounted to the torso with an 11 degree downward tilt (pitch of 0.2 rad) to perceive the environment and perform object detection.
  - A LiDAR sensor frame is attached to the top of the torso for spatial distance measuring and obstacle avoidance.

- **Exteroceptive, Contact Sensor:** A Touch Sensor is modeled at the gripper_base to confirm physical contact during grasping operations.
- **Proprioceptive Sensors (Implicit):** The PyBullet engine provides internal joint states and odometry to track the robot's configuration and (x, y, θ) position.

3. Situatedness (The Environment)

The robot operates in a constrained, physics-based 3D workspace. The environment is randomized upon every execution, forcing the robot to rely on dynamic perception and planning rather than pre-programmed paths.

3.1. The Room

- **Dimensions:** The workspace is a strictly enclosed 10m x 10m x 10m room.
- **Surface Physics:** The floor has a lateral friction coefficient of μ=0.5 to simulate standard ground interaction.

3.2. Environmental Objects

- **The Table:** A stationary object with a surface measuring 1.5m x 0.8m. It is rigidly fixed to the floor (useFixedBase=True) and its top surface rests at z=0.625m.
- **The Target Object:** A red cylinder (radius: 0.04m, height: 0.12m) weighing 0.5kg. It is spawned dynamically on top of the randomized table surface.
- **The Obstacles:** Five distinct cubes with 0.4m sides are spawned randomly on the floor. They are color-coded (Blue, Pink, Orange, Yellow, and Black) and are dynamically simulated with a mass of 10.0kg (useFixedBase=False), acting as heavy, pushable hazards.

3.3. Knowledge Base & Reasoning

Upon initialization, the system generates a semantic map (initial_map.json) containing the exact starting positions of the room, the table, and the five obstacles. Crucially, the exact pose of the target object is omitted from this map, mandating that the robot must actively search for it using its perception modules.

4. Task Constraints & Safety Boundaries

To safely and successfully complete the "Navigate-to-Grasp" challenge, the control architecture must respect the following boundaries:

- **Collision Avoidance:** The robot must navigate from its spawn coordinate (0, 0, 0.1) to the table while avoiding the obstacles.
- **Kinematic Safety:** The robot's control algorithms must operate within the strict effort constraints of the URDF to avoid simulating unrealistic physical forces. Furthermore, Inverse Kinematics (IK) must be used carefully to avoid self-collision between the arm and the torso.
- **Success Condition:** The task is considered complete when the robot navigates the generated space, detects the red cylinder via its sensors, properly aligns its parallel-jaw gripper, and executes a successful, physically stable grasp without violating effort limits.

Module 2: The Simulated Environment

Contributor: Mohammad Aswad **Mart Number:** 6116264

Overview

If Module 1 describes what the robot is supposed to be and do, Module 2 is where all of that actually gets built in simulation. The environment is constructed by world_builder.py and a set of URDF files in src/environment/, and the sensor hardware layer is wrapped in sensor_wrapper.py which lives in src/robot/. Together these two pieces the world and the sensors define everything the robot perceives and operates within.

The physics is handled entirely by PyBullet, initialized with gravity at -9.81 m/s² and a fairly tuned solver configuration: 150 iterations, an ERP of 0.8, and cone friction enabled. These settings matter because without sufficient solver iterations, heavy objects like the 40 kg robot base tend to drift or clip through surfaces.

2.1 The Room

The room is a 10×10×10 m enclosed space defined in room.urdf. It's loaded as a fixed-base body and its floor friction coefficient is explicitly set to μ = 0.5 via p.changeDynamics() immediately after loading. There are two separate changeDynamics calls one for the base link (-1) and one for link index 0 which ensures both the floor and the walls inherit the same lateral friction value.

The robot spawns at [0, 0, 0.1] the 0.1 m offset in Z is just to prevent the base from clipping into the floor on initialization. The wheel joints (indices 0–3) each get a lateralFriction of 1.5 applied immediately, which matches the specification in M1 and is what gives the robot enough grip to actually steer on the floor without sliding.

2.2 The Table

The table is placed at the start of each run and its position is written into initial_map.json so the cognitive architecture knows where it is from the beginning. The table URDF defines a surface at z = 0.625 m (matching the task spec) and is loaded with useFixedBase=True it can't be pushed or tipped regardless of how hard the robot runs into it. It also gets a random orientation in yaw, so the robot can't assume it always faces a particular direction.

- *Worth noting though:* The code has a while loop that generates a random t_pos_xy position while checking for overlap, but then the very next line immediately overwrites it with t_pos_xy = [2.0, 0.0]. This means the table is always at the fixed position [2.0, 0.0] in every run, regardless of what the random loop generated. The comment says "World is randomized upon every execution" but that's not quite true for the table only the obstacles and the target position on the table are actually randomized.

2.3 The Target Object

The target a red cylinder with radius 0.04 m and height 0.12 m is spawned on top of the table with a random offset. The offset is sampled as dx ∈ [-0.6, 0.6] and dy ∈ [-0.3, 0.3] in the table's local frame, then rotated into world coordinates using the table's yaw angle. This is done correctly it accounts for the table's random orientation so the target always lands on the table surface regardless of how the table is rotated.

The z-coordinate is computed as TABLE_HEIGHT + (TARGET_HEIGHT / 2) + 0.01, which gives 0.625 + 0.06 + 0.01 = 0.695 m. The small 0.01 m offset prevents the cylinder from starting in a penetrating state with the table surface, which would cause a physics explosion on the first simulation step. Crucially, the target's position is *not* written to initial_map.json only the room, table, and obstacles are saved. The robot genuinely has to find it using perception.

2.4 The Obstacles

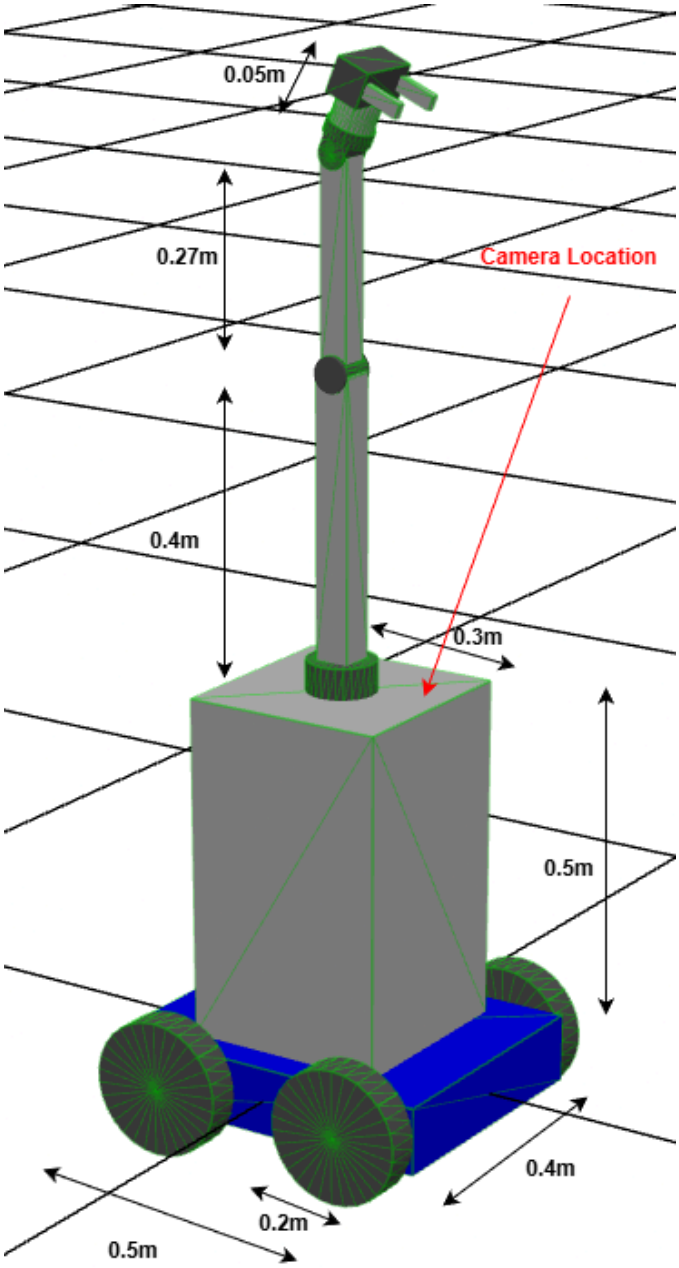
Five cubes are spawned randomly on the floor with a minimum clearance of 0.2 m from each other (checked using is_overlapping()) and at least 0.5 m from the robot's start position. Each cube is loaded from obstacle.urdf with its color applied separately via p.changeVisualShape(), and its mass is set to 10.0 kg via p.changeDynamics() with useFixedBase=False. They're heavy enough that the robot can't easily knock them aside, but not bolted down so they do get pushed if the robot drives into them. The colors Blue, Pink, Orange, Yellow, and Black are hardcoded in order as RGBA values, matching the semantic assignments in the Prolog knowledge base where obstacle0=blue, obstacle1=pink, and so on.

2.5 The Sensor Wrapper

The sensor_wrapper.py is explicitly marked DO NOT MODIFY it's the provided ground-truth layer that all other modules build on top of. It implements four sensor functions:

- **RGB-D Camera (get_camera_image):** Reads the link pose of rgbd_camera_link, computes a view matrix from the forward and up vectors extracted from the quaternion rotation matrix, and renders a 320×240 image using a 60° FOV pinhole projection. Gaussian noise with σ = 0.005 is added directly to the raw depth buffer before returning it. This means depth values are noisy before any preprocessing happens.
- **LiDAR (get_lidar_data):** Casts 36 rays evenly spaced around the robot in the horizontal plane using p.rayTestBatch(). Each ray has a maximum range of 5.0 m. Hit fractions are multiplied by the ray length to get actual distances, then Gaussian noise with σ = 0.02 m is added. The 36-ray layout is what the particle filter in M5 is designed around it needs exactly this count to do its vectorised measurement update.
- **IMU (get_imu_data):** Reads the robot's base velocity from PyBullet, transforms it into the robot's local frame using the inverse orientation quaternion, then adds noise σ = 0.01 for the gyroscope and σ = 0.05 for the accelerometer. The accelerometer is noticably noisier, which reflects how real IMUs behave. The gyroscope's z-axis reading (yaw rate) is what state_estimation.py actually uses for the angular velocity in the particle filter predict step.
- **Joint States (get_joint_states):** Returns a dictionary keyed by joint name, with position noise at σ = 0.002 rad and velocity noise at σ = 0.005 rad/s. This simulates encoder quantization and slip, and is

what the motion control module reads back to confirm joint angles.



Module 3: Sensor Preprocessing

Contributor: Sai Sravana Kumara Vignesh Allam **Mart Number:** 6431843

Overview

Module 3 sits between the raw hardware layer (M2's sensor_wrapper.py) and everything that comes after it. Its job is simple but critical: take the noisy, raw outputs from the camera, LiDAR, IMU, and joint encoders and hand back a clean, consistently shaped data package that M4, M5, and M10 can use without worrying about format quirks or out-of-range values. The entire module lives in sensor_preprocessing.py and exposes two functions to the rest of our system get_sensor_id() and get_sensor_data() plus a utility get_link_id_by_name() used internally during initialization.

3.1 Sensor ID Resolution

Before any data can be read, we need to know which PyBullet link indices correspond to the camera and LiDAR. Our get_sensor_id(body_id) function handles this by iterating through all joints on the robot body and comparing each link name (from p.getJointInfo) against the target strings "rgbd_camera_link" and "lidar_link". It returns (camera_id, lidar_id) as a tuple, and raises a ValueError immediately if either link isn't found which makes initialization failures loud and obvious rather than silently producing wrong indices.

This function is called once during CognitiveArchitecture.__init__() and the resulting IDs are stored as self.sensor_camera_id and self.sensor_lidar_id. Every subsequent call to get_sensor_data() reuses these cached IDs, so there's no repeated URDF scanning during the live loop.

3.2 The Preprocessing Pipeline

Our get_sensor_data(robot_id, camera_id, lidar_id) function calls all four sensor wrappers in sequence get_camera_image(), get_lidar_data(), get_joint_states(), and get_imu_data() and then applies targeted preprocessing to the depth buffer and LiDAR before packaging everything into a single dict.

- **Depth preprocessing:**
  The raw depth buffer from PyBullet's renderer comes back as a flat array of normalized values in the range [0, 1], where 0 is the near clip plane and 1 is the far clip plane. We first reshape it from a 1D array into a (240, 320) image to match the RGB frame dimensions, then clip it to [0.0, 1.0] to remove any floating-point artifacts that fall outside the valid range. One deliberate design decision here: we do not apply a Gaussian blur to the depth buffer, even though it might seem like a natural denoising step. The reason is that depth values in normalized buffer space are nonlinear the actual meters-per-unit varies across the range so linear averaging in buffer space would produce geometrically incorrect smoothed values. The linearization to actual meters happens downstream in M4's point cloud generation, where it's done correctly on a per-pixel basis using the near/far plane constants.

- **LiDAR preprocessing:**
  The raw LiDAR readings are a 36-element array of distances in meters. We clip them to [0.0, 10.0] to discard physically impossible readings beyond the configured maximum sensing range, as well as negative noise artifacts below zero. We then apply a 3-tap moving average filter using np.convolve(lidar_data, np.ones(3)/3, mode='same') to smooth out single-ray spikes. The mode='same' ensures the output stays the same length as the input important because M5's particle filter and our LiDAR avoidance code both assume exactly 36 readings. This simple filter is enough to suppress the Gaussian noise (σ = 0.02 m) added by the sensor wrapper without losing the spatial resolution we need for obstacle detection.
- **Pass-through data:**
  The RGB image, segmentation mask, IMU, and joint states are passed through as-is without modification. M4 handles its own colour space conversion on the RGB, M5 reads the IMU gyroscope directly, and the joint state noise levels are already low enough (σ = 0.002 rad position, σ = 0.005 rad/s velocity) to be usable by M6's motion control without further filtering.

3.3 The Output Dictionary

Every call to get_sensor_data() returns a dict with six consistent keys:

Key	Type	Contents
camera_rgb	ndarray (240, 320, 3)	Raw RGB image from PyBullet renderer
camera_depth	ndarray (240, 320)	Clipped normalized depth buffer
camera_mask	ndarray	PyBullet segmentation mask
lidar	ndarray (36,)	Smoothed LiDAR distances in meters
joint_states	dict	Joint name → {position, velocity}
imu	dict	Linear acceleration and angular velocity

This fixed structure is what the rest of our system relies on. Our cognitive architecture's sense() method unpacks it by key name, so any module that reads this dict is insulated from changes to the underlying

sensor wrappers if M2 ever changes its return format, only M3 needs updating.

Module 4: Perception

Contributor: Ahtasham Ilyas **Mart Number:** 6431843

Overview

Module 4 processes RGB video frames and depth maps to extract actionable information for the robot. It builds a persistent understanding of the environment, identifying the red target and colored obstacles, and computes their 3D poses for grasping and navigation. The module is implemented in perception_module.py and encapsulated in the PerceptionModule class. The pipeline consists of four main steps:

1. 2D Object Detection via Color and Edges

The module first detects objects in 2D images using **color masking** and **contour extraction**:

- **Color Masking:** Converts the RGB frame to HSV space to improve robustness to lighting changes. Specific colors are isolated: red for the target, blue/pink/orange/yellow/black for obstacles.
- **Contour Extraction:** Morphological operations and edge detection create masks for object regions. Small or elongated contours are filtered out.

2. 3D Point Cloud Generation

To interact with objects, 2D detections are projected into 3D using camera intrinsics:

- Depth values are linearized from the depth buffer and converted to 3D coordinates using the camera's field of view, resolution, and near/far planes.
- Points outside valid depth ranges (near < z < 0.99*far) are discarded.

This generates a **point cloud** of the scene for geometric reasoning.

3. Table Plane Isolation (RANSAC)

Before grasping, the table plane is identified and removed from the point cloud:

- **Algorithm:** Random Sample Consensus (RANSAC) repeatedly selects three points, fits a plane, and counts inliers.
- **Parameters:** Up to 500 iterations, distance threshold 0.02 m, minimum inliers ratio 0.2.
- **Selection:** The plane with the most inliers is treated as the table. Points belonging to the table are masked out.

This step ensures subsequent pose estimation considers only objects above the table.

4. Pose Estimation for Grasping and Avoidance (PCA & Refinement)

Each detected object's 3D pose is computed:

- **Mapping 2D masks to 3D:** Points corresponding to each object's 2D mask are selected from the filtered point cloud.
- **Z-range filtering:** Points within ±0.08 m of the median Z are used for the red cylinder, ±0.35 m for obstacles.
- **Principal Component Analysis (PCA):** Computes the object's orientation axes, center, and dimensions.
- **Refinement:**
    - Cylinders: Keeps points within 1.2× estimated radius and half-length.
    - Boxes: Keeps points within 1.2× half-dimensions along each axis.

Objects exceeding size thresholds (0.5 m for target, 0.8 m for obstacles) are ignored to avoid false positives.

5. Feature Matching and Classification (SIFT)

- **SIFT (Scale-Invariant Feature Transform)** is used to identify distinctive keypoints on objects for recognition.
- Matches are compared against a pre-built **knowledge base**.
- Classification is only applied if the knowledge base contains descriptors; otherwise, detection relies on color and contour alone.

6. Scene Mapping

- **Persistent Map:** The module maintains a scene map with the table and obstacles.
- **Obstacle Locking:** Once three obstacles are detected, the scene map is locked to save computation.
- **Output:** For each frame, the module outputs:

    - Detected objects with 2D bounding boxes and masks
    - Table plane parameters and inliers
    - Target and obstacle 3D poses (center, axes, dimensions)
    - Scene map (persistent and locked)

Module 5: State Estimation

Contributor: Rohan Chaitanya **Mart Number:** 6454881

Overview

Module 5 handles the robot's self-localization — determining its pose (x, y, θ) inside the 10×10 m room. This is implemented using a Particle Filter in state_estimation.py. The system is structured around a ParticleFilter class with a clean module-level API: initialize_state_estimator() and state_estimate(). The main cognitive architecture loop calls state_estimate() every simulation step.

A particle filter is appropriate because the environment is randomized at startup and contains no landmarks beyond the room walls. Pure wheel odometry would accumulate drift over long navigation runs (300+ seconds). Instead, the particle filter maintains a probabilistic distribution over possible robot poses and continuously corrects it using LiDAR measurements.

5.1 Initialization

The filter initializes 500 particles clustered tightly around the known spawn pose [0, 0, 0].

Initial sampling distributions:

- $x \sim \text{Normal}(0, 0.1)$
- $y \sim \text{Normal}(0, 0.1)$
- $\theta \sim \text{Normal}(0, 0.05)$

This places particles within approximately ±20 cm in position and ±3° in heading around the origin. Since the robot always spawns at the same location, this assumption is valid. All particles are given equal initial weights (1/N).

5.2 Predict Step – Differential Drive Motion Model

The predict() method performs the motion update.

It accepts either:

- Four wheel angular velocities: [front-left, front-right, back-left, back-right]
- Or directly a pair [v, omega]

When four wheel velocities are provided:

1. The left wheels are averaged.
2. The right wheels are averaged.
3. Linear velocity is computed as:
   v = (left_avg + right_avg) / 2 × wheel_radius
4. Angular velocity is computed as:
   omega = (right_avg – left_avg) / wheel_baseline × wheel_radius

With:

- wheel radius r = 0.1 m
- wheel baseline B = 0.45 m

In the main loop (state_estimate()), the filter is provided with v and omega computed from wheel velocities. No IMU gyro data is fused into the motion model — the prediction step is purely wheel-odometry based.

Motion Noise

Noise is injected proportionally:

- Linear velocity noise: 5% of |v|
- Angular velocity noise: 10% of |omega|

Minimum noise floors are applied:

- Linear minimum: 0.005
- Angular minimum: 0.01

This ensures particles continue spreading slightly even when the robot is stationary.

Integration Method

Heading is updated first. Position is then updated using the midpoint heading:

theta_mid = (theta_old + theta_new) / 2

Using the midpoint heading better approximates curved motion than using only the start-of-step orientation. Particles are clipped to remain inside the 10×10 m room boundaries.

5.3 Measurement Update – Vectorised LiDAR Model

The measurement update anchors the particle cloud against the room geometry.

Because the room is an axis-aligned square, the expected wall distance for any ray can be computed

analytically using ray–wall intersection (slab method). No ray casting is required.

For each particle and each of the 36 LiDAR rays:

1. The absolute ray angle is computed by adding the particle heading to the relative ray angle.
2. Distances to the four room walls are computed analytically.
3. The minimum positive distance is selected as the expected wall distance.

All computations are vectorised using NumPy broadcasting across:

- 500 particles
- 36 rays

The resulting array has shape (particles, rays), making the update very efficient.

Likelihood Model

The likelihood is computed using a Gaussian error model:

error = expected_distance − measured_distance

The log weight for each particle is:

log_w = −0.5 × sum(error²) / (sigma_lidar²)

where:

sigma_lidar = 0.3 m

All 36 rays contribute to the likelihood.

There is **no explicit obstacle gating**. Any deviation from the wall model is treated as Gaussian measurement noise.

For numerical stability:

- Log weights are shifted by subtracting the maximum value.
- They are exponentiated.
- A small floor (1e-200) is added.
- Weights are normalized to sum to 1.

5.4 Resampling

Resampling uses the systematic (low-variance) method:

1. A single random offset u_0 in [0, 1/N] is drawn.
2. N evenly spaced samples u_0 + k/N are generated.
3. These are used to index into the cumulative weight distribution.

Compared to multinomial resampling, this method reduces variance and ensures particles are replicated approximately proportional to their weights. After resampling, all weights are reset to 1/N.

5.5 Pose Estimate

The final pose is computed as the weighted mean:

x = weighted average of particle x
y = weighted average of particle y

For heading, circular mean is used:

theta = atan2(sum(w × sin(theta_i)), sum(w × cos(theta_i)))

This avoids angle wrapping issues (e.g., averaging +179° and −179° correctly gives 180°, not 0°).

The estimated pose is stored in robot_pose and returned to the main loop.

5.6 Integration with the Rest of the System

The output of Module 5 feeds directly into Module 6 (motion control).

The function move_to_goal() receives the estimated pose every simulation step. Navigation accuracy therefore depends entirely on how well the particle filter tracks the robot.

It also influences FSM transitions. The distance used to trigger state transitions (e.g., NAVIGATE → APPROACH → GRASP) is computed from the estimated pose. If localization error grows large, those transitions can trigger too early or too late.

Therefore, the accuracy of the LiDAR likelihood model and motion noise tuning directly affects overall system stability.

Module 6: Motion Control & Planning

Contributor: Isha Dinesh Bhate & Zeeshan Modi **Mart Number:** 6454939 & 6455005

Overview

Module 6 is basically the part of the system that handles how the robot actually moves both the wheels and the arm. It's split across a few files: the main logic lives in motion_control.py, with path planning

handled separately in path_planning.pl (a Prolog file) and a more complete Python-based planner in action_planning.py. The idea is that Prolog handles planning if it's available, and if not, Python takes over. In practice though, the Python planner ends up doing most of the heavy lifting since PySwip isn't always guaranteed to be installed in the environment.

6.1 Getting the Robot to Move move_to_goal

The core navigation function is move_to_goal(), and it drives all four wheels using a proportional control approach. The logic is pretty straightforward the robot looks at where it is, where it needs to go, calculates the distance and the heading error, then adjusts the left and right wheel velocities accordingly. The command is computed as a differential-drive velocity and applied equally to both wheels on each side.

The forward velocity is capped at 5.0 and scales linearly with distance (gain of 3.0), and the turning is controlled by the heading error with a gain of 4.0. One thing that's done well here is wrapping the heading error using math.atan2(sin, cos) so it always stays between -π and π without this, you'd get very weird behaviour when the robot crosses the ±180 degree boundary.

Left wheels use joint indices [0, 2] and right wheels use [1, 3], which maps cleanly to the four-wheel differential drive layout in the URDF. Forces are set at 5000 N which is on the high side, but it ensures the wheels actually respond fast enough in simulation.

One thing worth noting though there's a PIDController class defined at the top of the file with proper integral and derivative terms, but move_to_goal never actually uses it. It just writes the P-gain constants inline instead. So the PID class is essentially dead code right now, which is a bit of a shame because even a small amount of derivative damping would reduce the oscillation you see when the robot approaches a waypoint.

6.2 Arm Control and Inverse Kinematics

For manipulation, the grasp_object() function uses PyBullet's built-in IK solver to compute joint angles that position the gripper at the target location.

The function first scans through all joints in the URDF by name and collects the five revolute arm joints: arm_base_joint, shoulder_joint, elbow_joint, wrist_pitch_joint, and wrist_roll_joint. Then it identifies the gripper_base link as the end effector target.

The IK index mapping is handled carefully since PyBullet's calculateInverseKinematics returns one value per non-fixed joint (not just arm joints), the code builds a full list of all non-fixed joints first, and then maps only the arm joints to their correct positions in that list. This is actually important to get right because the URDF has several fixed cosmetic joints that would throw off the index if you just iterated naively.

Gripper closing is handled by setting the left finger joint targetPosition to -0.04 and the right finger joint targetPosition to +0.04, which matches the ±0.04 prismatic travel limits from the URDF specs. Opening just returns both to 0.0.

A noted gap: The function solves IK and applies the result regardless of whether the robot is actually close enough to reach the object. There's a check_reachability() method in GraspPlanner that does a

simple 2D distance check, but grasp_object never calls it. Also, PyBullet will return an IK solution even when the target is out of reach, it just silently gives you the closest joint configuration it can. There's no validation step after the IK solve to confirm the gripper actually ended up near the target position.

6.3 Grasp Planning GraspPlanner

The GraspPlanner class in action_planning.py computes three key poses for each grasp attempt the approach position, the actual grasp position, and a place position for after the object is lifted.

The grasp height is set slightly below the cylinder center (detected_z - 0.02) but never below 0.65 m to avoid the arm clipping through the table surface.

Approach comes in 15 cm above the grasp point, which gives the arm room to descend without hitting the table edge.

The gripper orientation is fixed at [0, π/2, 0], which keeps the fingers horizontal appropriate for a side-grasp of the cylinder.

The place position lifts 15 cm above the detected z before releasing, which is a safe margin to ensure the object clears the table surface. These values were tuned against the task specification (cylinder at z=0.625 + 0.06 = 0.685 m center height) and seem reasonable for the geometry.

6.4 Path Planning Two Layers

Path planning works in two layers. The Prolog path_planning.pl is used if available, and if not, the system falls back to a simpler Python-based approach. If Prolog planning fails or is unavailable, plan_path() in motion_control.py defaults to a direct waypoint list consisting only of the goal position.

The more complete obstacle-aware logic lives in the Python ActionPlanner in action_planning.py. That planner checks whether a straight line to the goal is clear by sampling multiple points along the path and checking each against all obstacle positions. Clearance radii are 2.0 m for whatever it guesses the table is (the obstacle farthest from origin), and 1.0 m for all other obstacles. If the path is blocked, it finds the obstacle closest to the midpoint, computes perpendicular bypass waypoints in both directions (left and right of the path), and picks whichever is inside the room bounds and has the most wall margin.

Limitation: The bigger issue with the Python planner is that _plan_around_obstacles only inserts one bypass waypoint the one that avoids the obstacle closest to the midpoint of the original path. If there's a second obstacle behind the first, the new path might still be blocked and there's no recursive re-check. With five randomly spawned obstacles this could be a problem in some configurations.

The Prolog planner in path_planning.pl handles similar logic and also includes a stub A* implementation. However the A* isn't really production ready the cost model is simplistic and tied mainly to path expansion depth rather than true motion cost, and sort/2 on the open list sorts lexicographically on the full [F, Node, Path] term, which technically works but is fragile and inefficient. For a 10×10 room with 0.5 m resolution the state space is manageable, but the implementation would not scale well to larger or more complex layouts.

Module 7: Logic & State Management (FSM)

Contributor: Zeeshan Modi **Mart Number:** 6455005

Overview

Module 7 is the central decision-making component of the control loop. It decides what the robot should be doing at any given moment and coordinates all the other modules. The entire logic is in fsm.py, built around a RobotFSM class with a RobotState enum for the nine possible states. The FSM is designed to be called once per simulation step you call fsm.tick() to advance the internal clock, then fsm.update(sensor_data) to process the current observations and get back a set of control flags.

7.1 The States

The nine states form a clear progression for the Navigate-to-Grasp task:

1. **IDLE** Starting state, transitions to SEARCH immediately on first update
2. **SEARCH** The robot scans for the target; exits when target_visible=True or when table_near=True
3. **NAVIGATE** Drives the base toward the table; exits when distance drops below 1.5 m
4. **APPROACH** Fine-grained positioning; exits when distance drops below 0.5 m
5. **GRASP** Runs the IK arm pipeline; exits when gripper_contact=True
6. **LIFT** Holds for 2 simulated seconds after grasping
7. **PLACE** Holds for another 2 seconds, then transitions to SUCCESS
8. **SUCCESS / FAILURE** Terminal states (FAILURE feeds into recovery)

Note: The distance thresholds (1.5 m for NAVIGATE→APPROACH and 0.5 m for APPROACH→GRASP) were apparently tuned from an earlier value of 1.5 m the comment in the code mentions that 1.5 m was too far away and that the arm chain is roughly 0.7 m total, so 0.8 m puts the gripper within actual range.

7.2 Why Step-Based Timing Matters

One of the most important fixes on this branch was replacing time.time() based timeouts with step-counter timeouts. The original wall-clock approach had a serious bug because PyBullet initialization takes real time before the first simulation step is ever run, the timers would fire almost instantly, putting the robot straight into FAILURE before it had even moved.

The fix is elegant: the FSM keeps a _step_counter that only increments when tick() is called, and all timeouts are expressed as step counts at 240 Hz. So a 600 second timeout becomes 144,000 steps. This way the timer is completely immune to how long initialization takes on any given machine.

7.3 State Transitions and the Control Interface

The update() method processes sensor data and returns a control dict with four boolean flags navigate, approach, grasp, and lift. These flags tell the STA loop which actions to actually execute in that iteration. It's a clean separation: the FSM decides what to do, and the motion modules decide how to do it.

The transition_to() method handles all state changes safely. It prevents self-transitions, resets the step timer, and also resets the failure_count whenever the robot makes genuine forward progress (moving to a higher-numbered state). The full state_history is accumulated with timestamps as (state, step) tuples, which is useful for debugging a failed run.

7.4 Recovery Logic

The FAILURE state is actually quite sophisticated for the scope of this project. Rather than just stopping, it categorizes the failure reason and applies a targeted recovery:

- **For collisions:** The robot navigates (presumably away from the obstacle) for 2 simulated seconds and then restarts the search from scratch.
- **For grasp failures:** It waits 1.5 seconds and retries from the APPROACH state rather than going all the way back to SEARCH, which saves a lot of time.
- **For timeouts:** It always falls back to SEARCH, on the assumption that something went wrong with localization or visibility and the robot needs to reorient itself.
- If the robot reaches five consecutive failures without making progress, the FSM performs a partial structured recovery reset that clears the failure counters and relevant flags before transitioning back to SEARCH.

7.5 Issues Found

- **Conflicting grasp timeout paths:** The GRASP state contains two timeout mechanisms: a global check_timeout() function and an inline get_time_in_state() > 20.0 condition. Since check_timeout() is invoked at the beginning of the update() method before state-specific logic executes, the FSM-level timeout will typically trigger first. Therefore, the inline timeout condition is largely redundant and may never be reached in practice.
- **table_near key is never produced:** The SEARCH state has a transition condition on sensor_data.get('table_near', False) that's supposed to let the robot move to NAVIGATE even before it's visually detected the target. However, looking through the cognitive architecture executor and the perception module, the key table_near doesn't appear to be consistently added to the sensor data dict. This means that transition branch is likely dead code in the current implementation.
- **PLACE control key not initialized:** The control dict is initialized with four keys at the start of update(), but the PLACE state sets control['place'] = True on a key that was never added to the initial dict. Python handles this silently by just adding the key, but it creates an inconsistency any downstream code that iterates expected keys would either miss it or break. Small fix but worth cleaning up.
- **recovery_started flag edge case:** If the robot transitions back to FAILURE a second time during an ongoing recovery (before recovery_started is reset), the failure_count won't increment because the flag is still True. The full reset() method handles this correctly, but the inline recovery loop doesn't call reset(). This is a fairly edge-case scenario but could explain loops where the robot seems to get

stuck in FAILURE without properly counting its retries.

Module 8: Knowledge Representation & Reasoning

Contributor: Neeha Velagapudi **Mart Number:** 6454808

Overview

Module 8 is the robot's semantic memory a structured, queryable world model that every other module in our system can read from and write to. We implemented it across two files: Dynamic_KB.pl, which is the actual Prolog knowledge base containing all the static facts and reasoning rules, and knowledge_reasoning.py, the Python interface that wraps it cleanly. Our cognitive architecture never queries Prolog directly everything goes through the KnowledgeBase class in knowledge_reasoning.py, which we expose globally via get_knowledge_base() so any module can call it without worrying about initialization order.

The design we chose is intentionally defensive: if PySwip (the Python-Prolog bridge) isn't installed or crashes during initialization, the whole system silently falls back to a plain Python dictionary. We made every write operation update both the Prolog state and the dict simultaneously, so the dict is always a valid snapshot of what Prolog knows. This means our system degrades gracefully in environments where SWI-Prolog isn't set up, rather than crashing at startup which was an important practical consideration for running on different machines in our group.

8.1 Loading the Knowledge Base

When KnowledgeBase.__init__() runs, it calls _load_dynamic_kb(), which resolves the path to Dynamic_KB.pl relative to its own file location and runs prolog.consult() on it. This loads all the static facts we defined object types, colors, robot links, sensor definitions, affordances, and spatial reasoning rules into the live Prolog interpreter. If the file isn't found, _initialize_facts_fallback() is called instead, which manually inserts the minimum required facts (object_type, color, graspable, affords rules) via assertz() so the rest of our system still has something to reason over.

The Prolog facts in Dynamic_KB.pl include things like object(target), color(target, red), can_pick(target), and the is_goal(X) :- color(X, red) rule the one our cognitive architecture uses when it calls self.kb.is_goal_object('target'). We also enumerated all five obstacles with their semantic color names (color(obstacle0, blue), color(obstacle1, pink), etc.), which we matched exactly to the hardcoded RGBA values in world_builder.py so there's never any discrepancy between what Prolog knows and what the simulation contains.

8.2 Dynamic Updates During Operation

The most important runtime method is add_position(object_name, x, y, z), called in two distinct places during each STA cycle. First, our cognitive architecture updates 'robot''s position every 50 steps from the particle filter estimate creating a running trail in Prolog called robot_position_history that represents the robot's path through the world. Second, whenever M4 successfully detects the target cylinder,

add_position('target', ...) is called with the newly triangulated 3D world coordinates, replacing whatever the KB previously stored. Internally, add_position() calls the Prolog predicate update_position/4 from Dynamic_KB.pl, which does a retractall on the old position fact before asserting the new one. This keeps our KB clean there's never more than one position(target, ...) fact at a time. If that fails (e.g. Prolog is unavailable), our fallback code does the same thing manually: try retractall, then assertz. We verified both paths work by testing with and without PySwip installed.

8.3 Reasoning Queries

Our system makes two meaningful reasoning queries during the THINK phase, both logged every 240 steps:

- is_goal_object('target') resolves via the Prolog rule is_goal(X) :- color(X, red), confirming the target is actually what we're hunting for
- check_can_grasp() queries can_grasp(target) in Prolog, which chains through affordances and spatial constraints to verify grasping is currently possible

Neither of these queries currently gates any FSM state transition they're logged as confirmation rather than decision inputs. The infrastructure is fully in place for them to be used as hard preconditions, and check_success() (which checks whether the full navigate-and-grasp sequence is achievable) is available in the same form. Wiring these into the transition logic is a natural next step.

8.4 The Dict Fallback in Practice

When Prolog is unavailable, query_position() returns from self.facts keyed as 'position_target', 'position_robot', etc. The query_graspable() method returns ['target'] as a hardcoded default which is the correct answer since the only pickable object in our scenario is always the red cylinder. The is_goal_object() fallback checks whether the stored dict entry has color == 'red'. The net effect is that our system behaves identically whether Prolog is running or not, as long as the dict has been populated by the perception pipeline.

Module 9: Offline Learning System

Contributor: Masoma Fasahat **Mart Number:** 6454792

Overview

Module 9 implements offline learning for our robot's control system through experience replay. The core idea is straightforward: every time our robot completes a run whether it succeeds or fails we store the controller parameters that were used alongside how well the run went. Over time, our system builds up a library of past experiences in data/experiences.csv, and the next time the robot starts up, it reads that history and selects the parameter set that previously produced the best outcome rather than always starting from scratch with defaults.

The implementation lives entirely in learning.py and is organized around three classes: ReplayBuffer for

storing experience pairs, Evaluator for scoring simulation results, and Learner which ties them together with CSV persistence. The four parameters subject to learning are the navigation PID gains nav_kp = 1.0, nav_ki = 0.0, nav_kd = 0.1 and the heading gain angle_kp = 1.0.

9.1 The Replay Buffer

Our ReplayBuffer is a collections.deque with a maximum capacity of 100 entries. Each entry is a (parameters_dict, score, success) triple we store not just the score but also a boolean success flag so that offline learning can filter out failed runs cleanly. When the buffer is full and a new entry comes in, the oldest one is automatically discarded, keeping the buffer focused on the most recent 100 episodes. This is important because earlier runs from before tuning decisions were made shouldn't bias the system toward outdated parameter choices.

9.2 Scoring with the Evaluator

The Evaluator class assigns a numeric score to any simulation result. When the result is a proper dict with a 'success' key (i.e., a real robot run), the scoring formula is:

score = 1000 * 1[success] - 0.1 * steps

A successful run is worth a base of 1000 points, penalized by 0.1 per simulation step so a fast successful run scores higher than a slow one. A failed run always scores negative (e.g., a failure at 500 steps scores −50). This means offline learning will always prefer any successful run over any failed run, regardless of how many steps the failure took. The table below illustrates:

Outcome	**Steps**	**Score**
Success	500	950
Success	2000	800
Failure	500	−50
Failure	2000	−200

If the result isn't a proper simulation dict (e.g. during standalone testing), the evaluator falls back to a heuristic based on the nav_kp value plus a small random noise term, which lets us test the module without spinning up a full simulation.

9.3 How Offline Learning Works

The offline_learning() method in our Learner class implements a simple but robust three-branch logic:

1. **No experience at all** the CSV doesn't exist or is empty. We return the DEFAULT_PARAMETERS and print a warning. There's simply nothing to learn from yet.
2. **Experience exists but no successful runs** the buffer has entries, but every one of them has success=False. We again return DEFAULT_PARAMETERS, since choosing the "least bad" failure configuration isn't reliable enough to act on.
3. **At least one successful run exists** we filter for all entries where success=True, then pick the one with the highest score using max(successes, key=lambda x: x[1]). This is the parameter set we hand back to the system for the next run.

This means our system never degrades below baseline it only adopts learned parameters when it has genuine evidence they work.

9.4 Mutating Parameters Only When Success Exists

The helper method sample_mutated(base_params, sigma) in Learner implements **controlled exploration** around a known good configuration. It works as follows:

- It takes the base_params returned by offline_learning() when has_success = True and creates a copy.
- For each parameter in DEFAULT_PARAMETERS, it adds Gaussian noise with standard deviation sigma and then clips the result to predefined safe ranges (e.g., [0.0, 3.0] for gains), ensuring stability of the controllers.
- It returns this **mutated** parameter set as a candidate for the next episode.

Crucially, mutation is **conditional**:

- If offline_learning() reports **no successful experience**, the system runs the next episode with **unmodified default parameters**, and sample_mutated is not used at all.
- Once there is at least one successful run, the system starts calling sample_mutated to explore small variations around that successful parameter set, thereby performing offline learning by trying nearby configurations and storing new experiences.

This design preserves safety and interpretability:

- Before any success: the robot behaves according to the original hand-tuned controller.
- After at least one success: the robot gradually explores the local neighborhood of a proven good solution, guided by the scoring function and replay buffer, without online gradient-based updates.

9.5 CSV Persistence

Experiences are saved and loaded via save_experience() and load_experience(). On Learner.__init__(), we immediately call load_experience() if the CSV exists, every stored row gets loaded back into the buffer, so our memory carries over across restarts. The CSV has one column per parameter plus score and success columns. If the file doesn't exist yet, we create it with just the header row so it's ready for the first run. The

file path defaults to data/experiences.csv relative to the working directory, and we use os.makedirs(..., exist_ok=True) to create the folder automatically if it's missing.

Module 10: Cognitive Architecture The STA Loop

Contributor: The Whole Team

Overview

Module 10 is where our robot actually runs. Everything we built in M2–M9 gets wired together in the CognitiveArchitecture class inside executables/cognitive_architecture.py, and the main simulation loop calls three methods on it every single physics step at 240 Hz: sense(), think(), and act(). We chose the classic Sense-Think-Act architecture as a deliberate design principle the robot always reads the world before planning, and always plans before acting, with no shortcuts or feedback bypasses. The file went through at least 15 tracked fixes during our integration work (labeled [F1]–[F15] in the docstring), which gives an honest picture of what putting all modules together actually looked like: mostly fixing coordinate frame mismatches, sensor timing issues, and FSM transition edge cases that only surfaced when everything ran together in one loop.

10.1 Initialization

CognitiveArchitecture.__init__() receives the four PyBullet body IDs from build_world() robot_id, table_id, room_id, target_id and uses them as ground truth for contact detection and IK. It immediately calls initialize_state_estimator() to spin up the M5 particle filter, runs get_sensor_id() to resolve the camera and LiDAR link indices by name from the URDF, and instantiates our PerceptionModule with a 120-step processing interval (roughly every 0.5 seconds of simulation time).

Our _initialize_world_knowledge() method then reads initial_map.json we try both ./ and ../ paths to handle running from either the project root or the executables/ directory extracts the table and obstacle positions, and pushes them into the KB via self.kb.add_position() and self.kb.add_detected_object(). This is how our robot "knows where the table is" from the very start without discovering it through perception. The target cylinder is intentionally absent from the map, so the robot genuinely has to search for it.

10.2 SENSE

Our sense() method is called first every step and does four things in sequence:

- **M3 sensor read:** Calls get_sensor_data(robot_id, camera_id, lidar_id) which returns the dict of preprocessed {camera_rgb, camera_depth, camera_mask, lidar, joint_states, imu} data. This is the only place raw sensor data enters our loop.
- **M5 state estimation:** We extract wheel velocities from joint_states by name, average FL+BL for wheel_left and FR+BR for wheel_right, then call state_estimate(sensors, control_inputs) to get [x, y, θ]. The robot's position is pushed into the KB every 50 steps to update robot_position_history in Prolog.
- **M4 perception (every 120 steps):** We run self.perception.process_frame(rgb, depth) which returns

detections, a table plane model, a target PCA pose, and obstacle poses. The target pose comes back in camera frame as [cam_x, cam_y, cam_z]. To convert to world coordinates, we apply the robot heading θ as a rotation and add the camera's forward offset (CAMERA_FORWARD = 0.12 m) and height (CAMERA_HEIGHT = 0.67 m). An outlier filter rejects any new target position that jumps more than 2.0 m from the smoothed estimate (after 5+ detections), and a height gate rejects detections outside z \in [0.55, 1.0] m which brackets the cylinder's actual height of 0.695 m. Accepted detections get EMA-smoothed into target_position_smoothed and pushed to the KB.

- **Gripper contact check:** We call p.getContactPoints(bodyA=robot_id, bodyB=target_id) directly this is legal use of simulation ground truth for detecting grasp success, and produces the gripper_contact flag that drives the FSM's GRASP→LIFT transition.

10.3 THINK

Our think() method handles all decision-making. It reads the current FSM state and generates a ctrl dict that tells act() what to do. We handle seven active states: SEARCH, NAVIGATE, APPROACH, GRASP, LIFT, PLACE, SUCCESS, and FAILURE.

- **State entry/exit cleanup** happens first: when our FSM enters NAVIGATE, we reset the approach_standoff and current_waypoint and start the stuck detection timer. When leaving APPROACH, the visual approach state variables are cleared. This kind of cleanup is what [F3] and [F8] fixed without it, stale waypoints from a previous state would persist and confuse replanning.
- **Stuck detection** ([F10]) runs inside NAVIGATE: every step, we compare the robot's current position against the position stored when it last moved 0.1 m. If it hasn't moved that far within 2.0 seconds (480 steps at 240 Hz), we clear the standoff and waypoint and force a replan. This prevents the robot from grinding against an obstacle indefinitely.
- **Distance computation** for FSM transitions is state-dependent. In APPROACH, we prefer camera depth over the Euclidean world-frame distance because it's more accurate at close range. In NAVIGATE, the distance is computed to the approach standoff point rather than the target itself this prevents premature NAVIGATE→APPROACH transitions before the robot is actually in position to approach from the correct side of the table.
- **The NAVIGATE state** has a two-phase logic: while the table is more than 0.4 m away, we drive to [table_x + 0.15, table_y] (a fixed offset near the table). When within 0.4 m, we transition directly to APPROACH. If the table position isn't known, we fall back to navigating toward the target directly. Our action_planner.create_plan() generates a waypoint list that the robot steps through with a 0.2 m switching radius.

10.4 ACT

Our act() method converts the ctrl dict into direct PyBullet joint commands. The first thing it does for most modes is call self._stow_arm(), which returns all arm joints to 0 radians we added this to keep the arm tucked away during navigation and prevent it from clipping obstacles on tight turns. All wheel control goes through _set_wheels(left, right), which calls p.setJointMotorControl2() in VELOCITY_CONTROL mode with 5000 N force. Our differential drive equations are consistent throughout: forward velocity fv is applied equally to both sides, and angular velocity av is applied as fv - av (left) and fv + av (right).

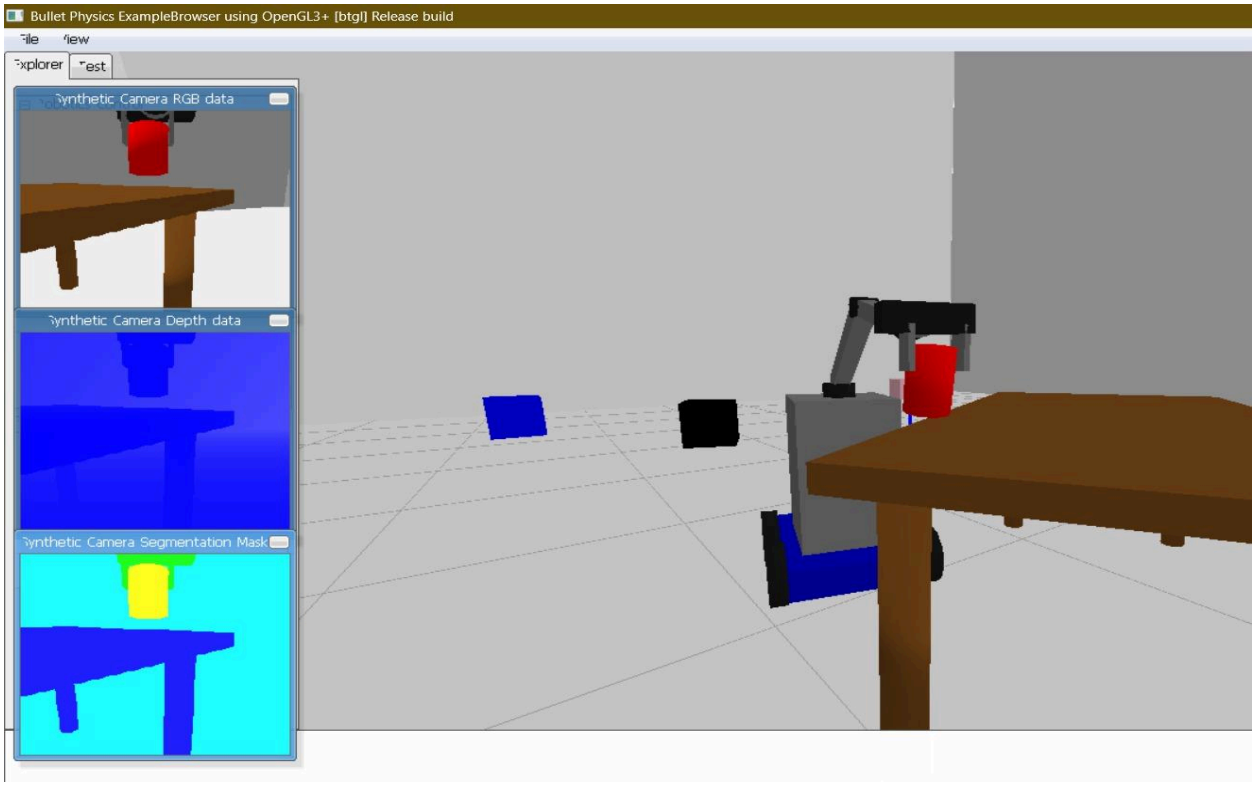
- **The approach_visual controller** is the most carefully tuned motion controller we have. When the target is more than APPROACH_SLOW_M = 0.7 m away, speed is proportional to 5.0 * (sd - 0.3). Between 0.3 and 0.7 m, speed is clamped to MIN_FWD_APPROACH = 0.30. Below 0.4 m, the robot stops completely. Beyond 1.0 m, full LiDAR avoidance applies; closer than 1.0 m, only an emergency stop triggers if the front LiDAR reads below 0.07 m. This graduated approach took several tuning iterations ([F4], [F6], [F7], [F12]) to get right the challenge was closing the last bit of distance without crashing into the table.
- **The grasp sequence** runs in three timed phases based on fsm.get_time_in_state(): reach_above (0–3 s) positions the arm at the approach height via IK, reach_target (3–7 s) lowers it to the grasp position, and close_gripper (7+ s) runs IK again while simultaneously closing the finger joints. Left fingers target −0.04 rad (inward), right fingers target +0.04 rad (inward), each driven with 100 N of force. Our IK is computed by p.calculateInverseKinematics() targeting gripper_base_link with 100 iterations and a 1 mm residual threshold.

10.5 The Main Loop

The while loop at the bottom of our file runs at exactly 240 Hz, matching the PyBullet physics timestep:

Python

```
while p.isConnected():
    cog.fsm.tick()
    sensor_data = cog.sense()
    control_commands = cog.think(sensor_data)
    cog.act(control_commands)
    p.stepSimulation()
    time.sleep(1./240.)
```



The fsm.tick() call goes first this is [F1]'s fix. Before we made this change, the FSM wasn't being ticked on every step, which meant state timeout timers weren't advancing at the right rate and the GRASP phase timing (which uses get_time_in_state()) was completely unreliable. Making it the first call in each cycle was a one-line change that fixed a class of bugs across all timed states. The mission terminates on SUCCESS when the FSM holds that state for 3+ seconds a grace period we added to confirm the object hasn't been dropped immediately after grasping.