ME 326 - Collaborative Robotics - Group Project

Website: https://sites.google.com/view/me-326-collabrobotics-group3?usp=sharing

Group 3
Hanvit Cho
Louis Conreux
Shalika Neelaveni
Tom Soulaire

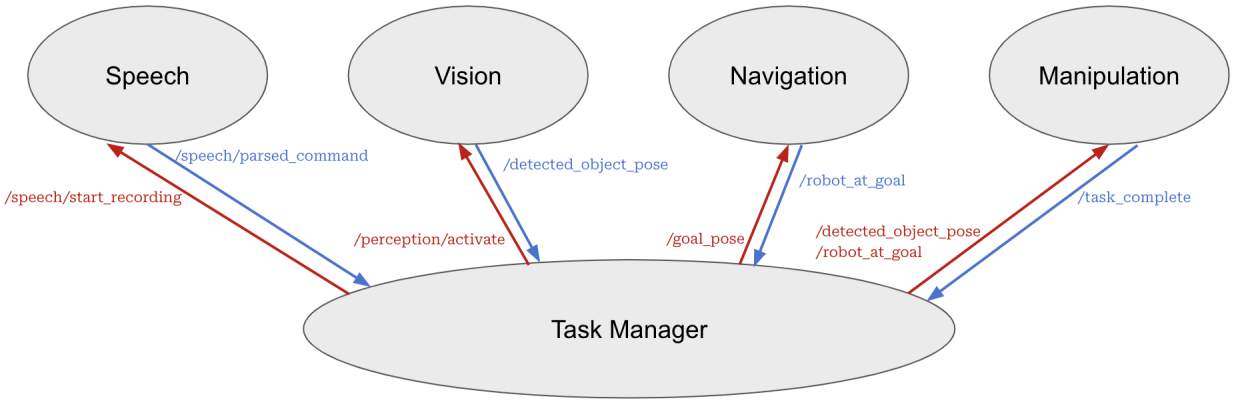
I. Contributions

Here is a breakdown of the contributions of each member:

- **Hanvit:** Focused on the manipulation node, ensuring smooth integration with the navigation node; Refined object coordinate extraction from the perception system and fine-tuned manipulator control for precise object handling.
- **Louis:** Developed the vision node, enabling object detection from camera images; Implemented the color detection module to identify objects based on color features
- **Shalika:** Led the navigation node development, ensuring the robot could successfully move to the target object; worked on integrating the perception system with navigation to improve localization accuracy.
- **Tom:** Developed the speech node and the task manager, enabling voice command processing; handled pixel-to-coordinate transformation for accurate object localization.

II. Code architecture

The code for this project can be found on Github:
https://github.com/LouConreux/ME326_Project
We chose to implement a brain-like architecture to be able to handle all of the required tasks with the same code structure. Each node is linked to a unique task manager responsible for the proper execution of the sequence of actions. The user would then have to launch a single python launcher before starting the interaction with the robot.



1. Task manager

The task manager node acts as the brain of our system, by integrating all inputs and outputs from the nodes. It decomposes high-level user instructions into a sequence of primitive actions and activates the corresponding nodes accordingly. For example, the command “fetch the banana” would be decomposed into four primitive actions: find (camera node), navigate (navigation node), pick (manipulation node), return (navigation node).
By maintaining a task queue and updating the current state of the robot, the task manager allows us to make sure information is processed in the correct order and that there is no conflict between the nodes.

2. Speech

The speech node transforms an audio user input into a JSON instruction file. It utilizes the Google Cloud Speech-To-Text API to transcribe the recorded audio file into text. This transcript is then parsed into a dictionary with four entries: task type, object, color and destination. Equivalency classes have been implemented to understand most commands from the user. For example, the retrieval task will be selected for any of the following verbs: "Retrieve", "Find", "Bring", "Fetch","Take".
Another solution could have been to use Gemini directly to parse the user audio input. For the sake of time, we chose to keep a hard-coded version of the parser.

3. Vision

The vision node utilizes the Google Cloud Vision API to interpret commands based on speech input. It processes two types of recognition tasks:

- Object-Based Recognition (when an object is specified):
    - Detects all objects in the RGB image.
    - If an object's class matches the requested object, it computes the coordinates using depth information and a pinhole camera model.
- Color-Based Recognition (when a color is specified):
    - Identifies all objects in the RGB image and classifies their colors using Otsu’s method for masking and HSV color space conversion.
    - If an object matches the requested color, it selects the first detected object of that color.
    - The system then runs the object-based recognition process.

Finally, the computed coordinates are transformed into the arm’s reference frame and published for navigation and manipulation.

4. Navigation

The navigation node gets a target object's position from the task manager. It calculates a proportional control input based on the positional error and generates velocity commands accordingly.
To ensure smooth movement, the robot's linear velocity is limited at 0.2 m/s, while its angular velocity is limited to 0.2 rad/s. The Locobot stops when it reaches within 0.2 meters of the target pose, positioning itself to face the object for efficient manipulation.

5. Manipulation

The manipulation node activates once the navigation node signals that the robot has reached its goal. It receives the target object's position, which is transferred from the camera to the arm base for final adjustments.
To ensure a precise grip, the manipulator moves directly above the object, slowly lowers toward it, and securely grasps it. Additionally, fine-tuning of the end-effector's pitch and roll is necessary to optimize the grasping action. Once the task is successfully completed, the node returns a Boolean value to indicate its completion.

III. Key algorithms

All the source code used to perform the different tasks can be found on the GitHub more precisely in:
ros/collab_ws/src/collaborative_robotics_course/locobot_autonomy/locobot_autonomy

1. Task manager

The task manager node source code is in `task_queue_manager.py`. This python file is responsible for activating the downstream nodes required by the identified task_type in the parsed audio command, and in the right order. So far, everything is hard-coded so that either Task 1 is launched when task_type == 'retrieval', Task 2 is launched if task_type == 'sequential', and Task 3 is launched if task_type == 'sort objects'. The task manager works for Task 1 and Task 2 but we did not have time to implement Task 3 with the task manager.

2. Speech

The speech node source code is in `speech.py` and the audio parsing code is inside `audio/`. To do the audio transcription, we are using Google Cloud Speech to Text API `speech_v1p1beta1`. We then parse the audio transcription result to give a general command understandable by the task manager. The parsing is hard-coded but a synonym equivalence class has been implemented to ensure a wide range of audio commands can be understood by the task manager (see details in `audio/speech_transcriber.py`).

3. Vision

The vision node source code is in `perception.py` and the vision detection code is inside `vision/`. To identify objects in an image, we are using Google Cloud Vision API `vision`. We use the bounding boxes provided by the `vision` result to retrieve the pixel coordinates of the desired object. For the color version, we also use the bounding boxes provided by the `vision` result, but refine them by using Otsu thresholding method to quickly find a mask of the identified object to compute its HSV color. We then map a color string to ranges of HSV values. The identified HSV color is then compared to those ranges to retrieve the color string. If it corresponds to the desired color, we return the object name provided by the `vision` result.

In computer vision and image processing, Otsu's method, named after Nobuyuki Otsu (大津展之, Ōtsu Nobuyuki), is used to perform automatic image thresholding. In the simplest form, the algorithm returns a single intensity threshold that clusters pixels into two classes, foreground and background.
The threshold is found by maximizing the intra-class variance defined as the weighted sum of variances of the foreground class and background class:
If *t* is the threshold used,

$$\sigma(t)^2 = \omega_f(t)\sigma_f{}^2(t) + \omega_b(t)\sigma_b{}^2(t)$$

where
$\omega_f(t)$: Frequency of pixel intensity above threshold *t*
$\omega_b(t)$: Frequency of pixel intensity below threshold *t*

4. Navigation

The navigation node source code is in `navigation.py`. The robot's motion is regulated by a feedback control system that continuously computes the error between its current position (obtained from odometry data) and the desired goal pose (published by the vision node) with a 0.2 m clearance for manipulation. The proportional control term (Kp) adjusts the robot's velocity based on the magnitude of the position error, ensuring rapid convergence to the target. Although we explored using an integral and derivative term, we found that just using a proportional controller performed best. The algorithm also employs coordinate transformations using rotation matrices to account for the robot's orientation when computing motion commands. By utilizing a non-holonomic control matrix, the system maps the desired movement of a reference point on the robot to linear and angular velocity that comply with the constraints of the differential drive mechanism. The final velocity commands are then constrained by predefined limits to ensure safe and smooth movement – in this case we ensured the linear and angular velocities did not go beyond 0.2 m/s and 0.2 rad/s, respectively. When the computed error falls below a predefined threshold, the robot stops, and a boolean signal is published to indicate that the target has been reached.

5. Manipulation

The manipulation node source code is in `manipulation.py`. The manipulation algorithm implemented in this script coordinates robotic arm movements for object grasping and placement, using a combination of object pose detection and predefined end-effector motions.

The system listens for a detected object’s pose and waits for a signal indicating that the robot's base has reached its goal before initiating grasping actions. Once the target pose is received, the algorithm executes a sequence of movements through the ArmWrapperNode, which controls the robot's gripper and arm. The gripper initially releases to ensure it is open before approaching the object. The arm is then guided to an intermediate position slightly above the object, followed by a descent to grasp it. The grasping is executed by closing the gripper, after which the object is lifted to a safe height and moved to a drop location also determined by the vision node. The arm subsequently returns to a resting position. The motion planning is based on inverse kinematics, ensuring smooth and accurate positioning. Additionally, predefined offsets in x, y, and z directions help to avoid collisions and account for slight inaccuracies while executing pick-and-place tasks.