Cookiebot: A Gesture Based Home Robot

Authors: Jerry Yu, Rama Mannava, Seungmin Ha: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*— Cookiebot is a robotic home assistant operated by gestures. Its primary application is to help people easily transport goods around the home. Cookiebot's tasks are to 1. Be tele-operated, 2. Drive to a home charging station, 3. Drive to the user, and 4. Drive to a location a user points to. Overhead cameras will capture the user and the robot. Images will be streamed to an NVIDIA Jetson Xavier to run OpenPose and classification models to classify gestures. Gestures will be relayed via a Node.js webserver to the robot. The robot will have a Raspberry Pi and Roomba to execute commands.**

***Index Terms*—2D mapping, 3D mapping, Home assistant, Gesture recognition with ML, Image processing, Localization, Low latency ML inference, Robotics, Websockets**

1 INTRODUCTION

Home robots are currently limited to a few specific functionalities and are clunky to control with an app or remote. Robots have been limited to entertainment (Anki, Zenbo), cleaning (Roomba, Landroid), and carrying payloads (Budgee). With Cookiebot, we wanted to focus on helping people carry goods easily around their home. Other existing robots like Budgee only follow the user, but we want Cookiebot to not only go to the user, but also go anywhere in the room just by pointing at a location. The advantage of our approach is that our system has full overhead coverage of the room, the robot, and the user in order to localize the position of the robot and the user. With the overhead cameras, we can identify gestures using machine learning to create natural methods of controlling robots unlike Roomba or Anki, which require phone apps.

In order to achieve intuitive control, our system is required to have low latency and high accuracy. Our goals are to:

1. Start performing all tasks within 1.9s on average for 10 tasks to be comparable to a Google Home.
2. Accomplish all tasks end-to-end with at least a 90% percent success rate and at most a 10% false positive rate to minimize user frustration.
3. Maintain less than 1 foot drift between where the system predicts the robot is and where the robot actually is after 10 tasks to stay within easy reaching distance.

To accomplish this, our system leverages a local NVIDIA Jetson Xavier (8 Core CPU, 500 CUDA Core GPU) within the room to perform low latency convolutional neural network computation (0.03s/frame), image processing, keypoint classification using SVMs, and also to host a webserver for low latency local network websocket communication between the cameras and the robot. We are also building a robust method of creating a 2D map of the room with robot encoders and a 3D mapping from pixel blocks captured by cameras to parts of the 2D room map. This will be constantly updated during runtime to have multiple methods of maintaining user and robot localization.

2 DESIGN REQUIREMENTS

The design requirements can be divided up to two parts: software and hardware requirements. Software requirements are there to make sure the latency and user experience is reasonable. Hardware requirements, on the other hand, remind us of our physical restriction and the expectation as of how the robot will function in the real world. Overall, meeting both requirements would be essential to ensure the successful and meaningful project.

A. Software Requirements

The purpose of setting requirements for the software system is to set the maximum allowable limits of latency and gesture recognition. Before we set any quantitative requirements, we set our overall end-to-end time restraint. Google Home, which is one example of commercially successful Smart Home products, takes about 1.9s between the user input and the output. Since we want this project to be an extended and improved version of the smart home concept, we decided that meeting the 1.9s end-to-end restriction would be reasonable.

The main sources of latency would be the data transmission between the web server and the robot subsystem. The communication between the camera system and the web server, while crucial to the project, would not have much latency since they are directly connected via USB cables. For the data transmission, we agreed to set 100ms latency requirement. This would be the maximum allowable time it takes for the Raspberry Pi to send 5 sensor data packets to the web server and to receive one path data from the server. This is to ensure that we have a fluid real time control of the robot. The reason we are specifically setting the latency limit for the transmission of 5 data packets is because that is the maximum number of sensors the server would need to generate or update the map: encoder readings for each wheel and bumper readings for front, left, and right. Also, 100ms seems to be the maximum communication latency which the user won't find uncomfortable. While testing, we were able to notice the discrepancy between the robot and the data read by the web server when the latency was longer than 100ms.

In addition, we are setting the required success rate of the gesture recognition to be over 80% per frame on average. We wanted to achieve high enough success rate while also being realistic with the ML model we are using. Having success rate lower than 80% would jeopardize our goal of having intuitive and easy control of the robot. At the same time, we wouldn't need an exceptionally high success rate since with a high enough fps the subsequent frames would correct the wrong gesture recognition.

B. Hardware Requirements

Hardware requirements are in place to restrict the behavior of the robot. This is both for the successful output and for the safety of the user.

First, we are requiring the robot to arrive at the goal point within 1 ft error. This is to ensure the user would be able to reach the robot even if the robot drifts a bit. This requirement would be enforced by having a robust odometry and a correcting the localization via camera system's video feed.

In addition, the robot must complete a path within a reasonable time limit. The robot should move fast enough to assist the user, but also not too fast to jeopardize the correctness of the odometry or to potentially harm the user. For this, we are requiring the robot to complete a 10m path within 50s on average. While testing, we found this time limit to enforce a reasonable speed restriction. Specifically, the speed of the robot wasn't high enough to damage the surrounding such as an object or person placed after the initialization of the map.

We are also requiring the robot to operate for at least 2 hours without charging. This is to ensure the user can spend enough time using the robot without the need of interruption. Most assistant tasks performed by the robot would not require more than 10 minutes to complete, but in case the user wants to use the robot for several consecutive tasks or wants the robot to wait in the same location for some time, we want to make sure the robot can handle such workload without having to return to the charging station.

3 ARCHITECTURE OVERVIEW

See Appendix A at the end of the document for the system architecture diagram.

There are four main components to the overall architecture of the system: dashboard, Xavier, camera system, and the robot. The diagram clearly illustrates how each components would interact.

A. Camera System

The camera system is composed of four individual USB cameras each pointing at the front, back, and sides of the room. These cameras capture the user and robot and send the data to the Xavier via web server. Having four cameras ensures that we aren't leaving any blind spot in the room. In addition, having multiple angles helps gesture recognition by providing additional "backup" frames.

B. NVIDIA Jetson Xavier

The Xavier will be the main body for the computation as well as hosting the web server. The board processes the image received from the camera system using OpenCV and OpenPose. After getting keypoints of the image and identifying the gesture, the gesture is classified into one of the acceptable gestures. This gesture is then translated into actual commands we can send to the robot subsystem. At the same time, the Xavier is receiving data such as encoder readings and light bumper sensor readings from the robot. These data will be processed to generate a 2D map. In addition, the board will use the video feed from the camera system to translate the 2D map to 3D representation by comparing the XY-coordinate of the robot on the 2D map and the location of the robot on the camera image.

As for the web server, the Xavier will host a web server established using express.js. In addition, it will manage websockets used to communicate with the robot subsystem. Having robust server and websocket management would be critical to the project, which makes the Xavier even more important to the overall integration. During our proposal, we said that we were going to use AWS as the webserver. However, our testing of the Xavier showed that it had enough CPU bandwidth to do run both image processing with OpenPose and host a webserver. Hosting the server locally instead of AWS also reduces latency because it is less RPC calls we have to make and is in our local network.

C. Robot System

The robot subsystem is composed of two components: Raspberry Pi and Roomba which are connected via serial USB-DIN cable. The Raspberry Pi receives commands from the Xavier over the web server. These commands are translated to a series of serial input by Raspberry Pi and sent to the Roomba for execution. On top of that, the Roomba is constantly reading its sensor values and reporting them to Raspberry Pi. Then, the sensor readings are delivered to the Xavier via websockets for further processing. One thing to note is that the Raspberry Pi is running in a headless mode. In other words, all the commands need to be given to the Raspberry Pi through the web server.

D. Dashboard

Lastly, the dashboard would be the visual hub for the current state of the entire system. The Xavier relays camera view and the map view to the monitor which is connected to the board via HDMI cable. The camera view will include the current camera stream as well as the gestures and key points recognized. The map view, on the other hand, would show the estimation of the location of the robot and user on the 2D map. Using this dashboard, we will be able to debug the system more easily, and the user will be able to get a visual and intuitive status update of the system.

4 DESIGN TRADE STUDIES

We considered several different options for each of the different components of our project.

4.1 Camera System

The camera system needs to have vision of the entire room to be able to identify gestures. It also needs to view the room along multiple planes to differentiate gestures that look identical when viewed on the same 2D plane. We considered using a 3D camera or multiple webcams. After some initial testing with the webcams we decided that having multiple webcams would be able to provide the functionality we needed at a much lower price than a 3D camera would cost. Our camera setup will have two cameras for front and back, and third camera at a right angle for gesture disambiguation, and an overhead camera to aid with point gesture recognition. These four cameras will give us three planes of information, which we can use to identify gestures in 3D space, while also providing visual coverage of the entire room without blind spots.

4.2 Keypoint Recognition

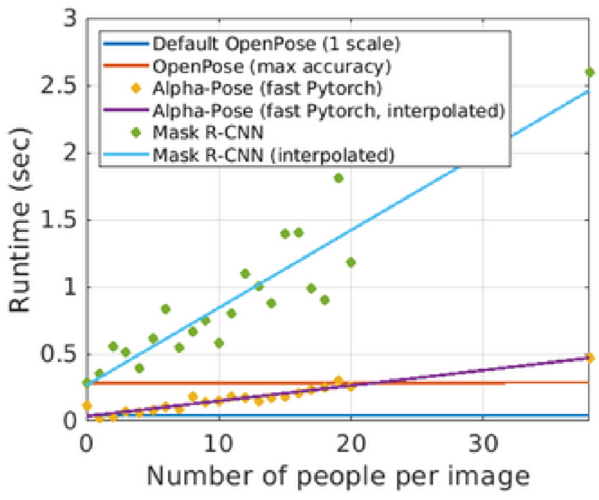


Figure 1: Runtime comparison of similar keypoint recognition algorithms. [1]

Recognizing gestures requires an algorithm that is able to identify keypoints when given an image of a person. The options we considered were OpenPose, Alphapose, and Mask R-CNNs. Fig. 2 illustrates the the relative performance of these algorithms on the same set of test images.

Given the tendency of keypoint recognition algorithms to scale linearly with the number of people in a given image, we decided to use OpenPose. For the case of only one person, AlphaPose and OpenPose both clearly perform better than the Mask R-CNN, and there isn't a large difference between the two. OpenPose gave us 0.03 seconds per frame on the NVIDIA Jetson Xavier, while also promising consistent results should we choose to support more than one person in the future. Given this clear benefit over AlphaPose and the slight difference in performance between the two for our current use case, we decided to move forward with using OpenPose as our keypoint recognition algorithm of choice.

4.3 Compute Hardware

We needed to select a hardware solution to run our chosen keypoint recognition algorithm as well as our in-house gesture recognition algorithm and the server handling robot communication. We considered using a laptop without a GPU, an NVIDIA Jetson Xavier, and a GCP instance. Fig. 3 shows the results of running OpenPose with a short test video on each of these options.

Our initial testing of OpenPose on a laptop required 20-30 seconds per frame, which was clearly unusable for any gesture recognition at all. We then tried using a GCP instance with 2 CPU cores and 5000 CUDA cores with performance results of 3.5 fps. It wasn't until our Xavier testing that we realized the importance of CPU performance and how seriously it bottlenecked our GCP test, as 4 CPU cores gave the Xavier 17 fps and activating the 4 inactive cores resulted in 27 fps. At this point we decided not to retry using a GCP instance as 27 fps is much faster than we require, and running OpenPose on the Xavier removes the latency we would have to battle to stream video to a GCP instance and stream keypoints or gestures back. CPU utilization on the Xavier at 27 fps was only around 20%, so it would also be able to handle the server required for communication with the robot. We decided to use only the Xavier for our compute hardware.

Table 1: Comparison of OpenPose Performance

Hardware	CPU Cores	CUDA Cores	FPS
CPU-Only	8	0	0.05
GCP Instance	2	5000	3.5
NVIDIA Jetson Xavier	4	500	17
NVIDIA Jetson Xavier	8	500	27

4.4 Web Server

A web server is required to bridge the gap between gesture recognition and the robot; it needs to convey the gestures that were identified to the robot so that it will be able to perform the tasks as instructed. To do this, we considered Node.js and Python as platforms, and WebSockets and REST as the method of communication. The main difference between both platform choices is their approach to asynchronous operation, as Node.js is natively asynchronous and Python is not. Given the nature of web communication and the potential for multiple concurrent requests, we decided that Node.js was the better option of the two. As for the communication itself, we had to compare providing WebSocket endpoints vs REST endpoints for the gestures and robot to both connect to. WebSockets are useful in situations where connections are anticipated to be maintained for a long time, as the overhead occurs when establishing the connection after which communication can easily proceed in both directions. On the other hand, providing REST endpoints would require reestablishing a connection with the server every time a message needed to be exchanged. This server is intended to maintain its connection with the gestures and robot indefinitely, so we opted for WebSockets to reduce the communication latency between gesture recognition and robot response.

4.5 Robot

The robot needs to be able to navigate through the room easily at a safe speed while also maintaining an adequate level of visibility to prevent any accidents with walking people. We considered using a Roomba or building our robot, but ultimately decided to use a Roomba. It is able to rotate in place so there are no concerns about turning circle for navigation in tight spaces, and has a variety of sensors that can be used for mapping and localization. Encoder data from its motors can be used to aid in localization. We decided to mount a Raspberry Pi on top of the Roomba to control it via serial port. The Raspberry Pi will communicate with the server and convert gestures into commands for the Roomba, while maintaining a map of the room and updating the Roomba's position as necessary. The Raspberry Pi is a light board that can easily be powered from a battery pack on the Roomba, allowing for untethered operation limited only by the range of its WiFi connection.

5 SYSTEM DESCRIPTION

Here are the parts of our system that we have thought about the most.

5.1 Subsystem A: Gesture Recognition

The gesture recognition system determines the user's gestures and relays the command to the webserver. It first collects images from 3 webcams placed in the front, back, and side of the room in order to maximize room coverage. We chose to use the Logitech C270 webcam since it was the cheapest USB webcam we could find and we could work with it's image format without warping or other preprocessing. Having USB webcams also allowed us to extend the cables cheaply (Amazon basics 2.0 extension 10 ft cables) and connect to a USB hub easily that our board could access with OpenCV.

OpenPose takes around 0.03s to process one image at 30 FPS, but processing 3 camera streams would bring our FPS down to 10 FPS. So, we want to build a module to determine which camera view to run OpenPose on to increase our FPS. The idea is to choose one camera as a primary camera and the others as secondaries. If we received keypoints with confidence from one camera view in previous frames, we can then classify the gesture using only those keypoints. However, we can still access the other camera views if the gesture requires multiple views or leaves the frame from one camera. In theory, we can bring the FPS to around 20 FPS by selectively choosing frames classify gestures.

OpenPose returns 25 keypoints across the body for us the classify. From experimenting with different parameters, we have found that using tracking mode gives us the best performance of 30 FPS. Tracking uses temporal information from previous frames in order to identify keypoints faster. However, it can only track one user at a time. We believe this tradeoff is worth it to reduce latency and to limit the scope of our project.

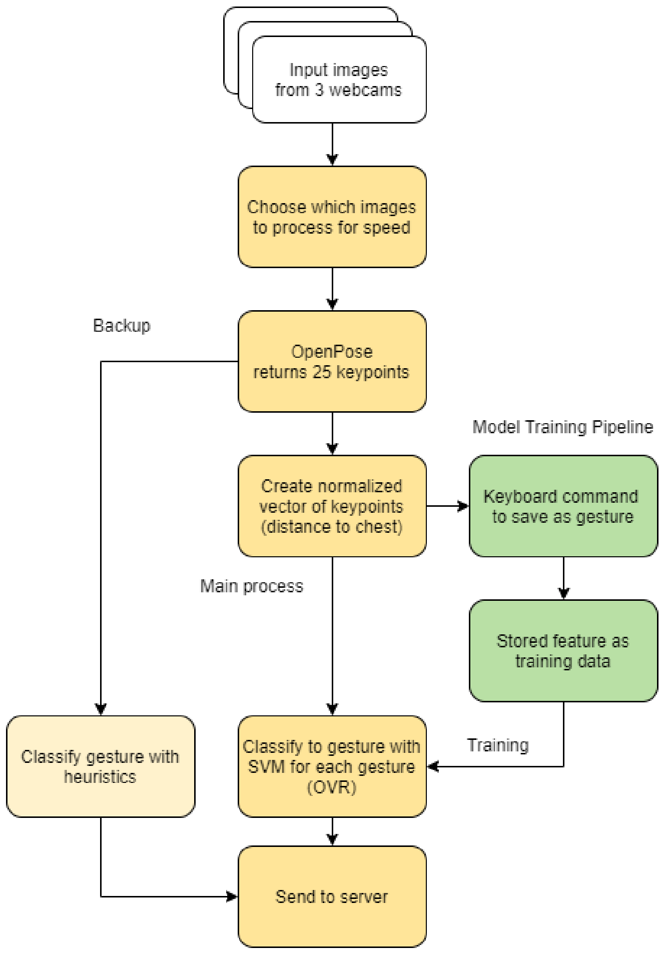

Input images from 3 webcams
Choose which images to process for speed
Backup
OpenPose returns 25 keypoints
Model Training Pipeline
Create normalized vector of keypoints (distance to chest)
Keyboard command to save as gesture
Main process
Stored feature as training data
Training
Classify gesture with heuristics
Classify to gesture with SVM for each gesture (OVR)
Send to server

Figure 2: Gesture recognition system and training pipeline

We can classify the keypoints using heuristics like left wrist is above the left shoulder, but these heuristics are not reliable for gestures for teleop and pointing when the user is not standing directly facing a camera. So, we want to use a SVM to classify each gesture by first creating a normalized feature vector for a person and gathering data to train a model. We wanted the full system to have an integrated training pipeline so we can gather data easily and collect data for edge case situations during run time. We can still use heuristics as a backup to mitigate the risk that gesture recognition fails.

5.2 Subsystem B: Getting location from point

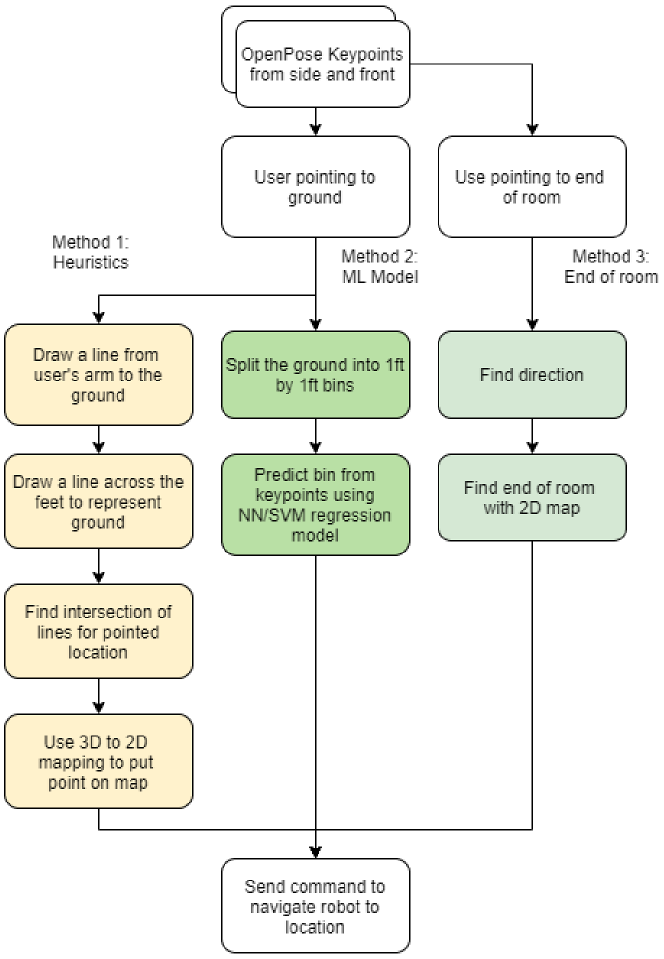

OpenPose Keypoints from side and front
User pointing to ground
Use pointing to end of room
Method 1: Heuristics
Method 2: ML Model
Method 3: End of room
Draw a line from user's arm to the ground
Split the ground into 1ft by 1ft bins
Find direction
Draw a line across the feet to represent ground
Predict bin from keypoints using NN/SVM regression model
Find end of room with 2D map
Find intersection of lines for pointed location
Use 3D to 2D mapping to put point on map
Send command to navigate robot to location

Figure 3: Pointing system

For pointing to the end of the room, we will find the direction of where the user is pointing to. From knowing the user's location on the map and the direction, we can find the point at the end of the room on the 2D map, and the robot can go to it.

For pointing to a location on the ground, we will experiment with two methods. The first method is to use heuristics to draw a line from the user's arm to the ground. We will determine where the ground is by using the location of the feet keypoints. By finding the intersection, we can find the pointed location once we convert the 3D camera view to the 2D map. The advantage of this method is that it is deterministic based on heuristics. The downsides are that we assume that we can see the feet of the user and when the user is not standing directly at the camera, more cases are needed to handle the heuristics. The solution to this problem is to use more cameras, but we are limited by our budget and OpenPose processing time.

Another method we will try to use a model to estimate the location on the floor is a ML model based approach. We will first divide the room into a grid of 1ft by 1ft of bins. We can then gather data by having the user point to a bin and mapping those keypoints to the appropriate bin. The model will predict the x and y coordinate of where the bin is on the grid, using a neural network or SVM regression model, since bins close together should have more similar keypoints.

Once we have the location on the 2D map from all the methods, we can tell the robot to plan a path towards that location.

5.3 Subsystem C: Robot

A. Components

The robot subsystem is composed of Raspberry Pi and the Roomba 671 itself. Additionally the battery pack (10,000mAH) will power the Raspberry Pi independent of the Roomba power source. The Raspberry Pi and the battery pack will be attached on top of the robot. It is important not to obtrude any sensors while attaching these components since some sensors (IR sensor, etc.) are placed on top of the Roomba. Eventually, there will be a basket attached on top as well to allow the robot to carry around a payload. A flag will also be mounted on the robot so the camera can detect it easily.

B. Odometry

Getting the XY-representation of the robot's location as precise as possible is crucial to the correct behavior of the system. To do this, we are using the encoder values of the Roomba to calculate the distance and the angle it moved each time stamp. The distance travelled is calculated using following equation:

$$\Delta d = \frac{(\Delta encoder_{right} + \Delta encoder_{left})}{2} \times r_{wheel} \times \frac{2\pi}{360} \quad (1)$$

In addition, the angle the robot has rotated can be calculated using

$$\Delta angle_{radian} = \frac{(\Delta d_{right} - \Delta d_{left})}{l_{wheelbase}} \quad (2)$$

By collecting these distance and angle travelled for each timestamp(15ms), we can approximate the XY-coordinate of the robot quite accurately.

$$\Delta x = -\Delta d \times sin(angle) \quad (3)$$

$$\Delta y = \Delta d \times cos(angle) \quad (4)$$

C. 2D mapping

For the 2D mapping, the robot will first follow the wall to collect information about the boundary of the environment. Then, based on the dimension of the room, the robot will move back and forth within the region of unknown to search for potential obstacles such as a furniture or a wall. Throughout this process, the robot will operate autonomously without any direction from the user. Therefore, we are adjusting the heading of the robot based on its sensor readings. For instance, if the robot detects an object on the right bumper, the robot will drive backward a bit and rotate to the left by a small increment. This will allow the completeness of the wall following and ensure the robot doesn't get trapped within certain region of the room. After completing this process, the robot will collect enough samples to generate a blotted-2D representation of the room. This map will be the base of any path finding and localization.

5.4 Subsystem D: 3D to 2D mapping

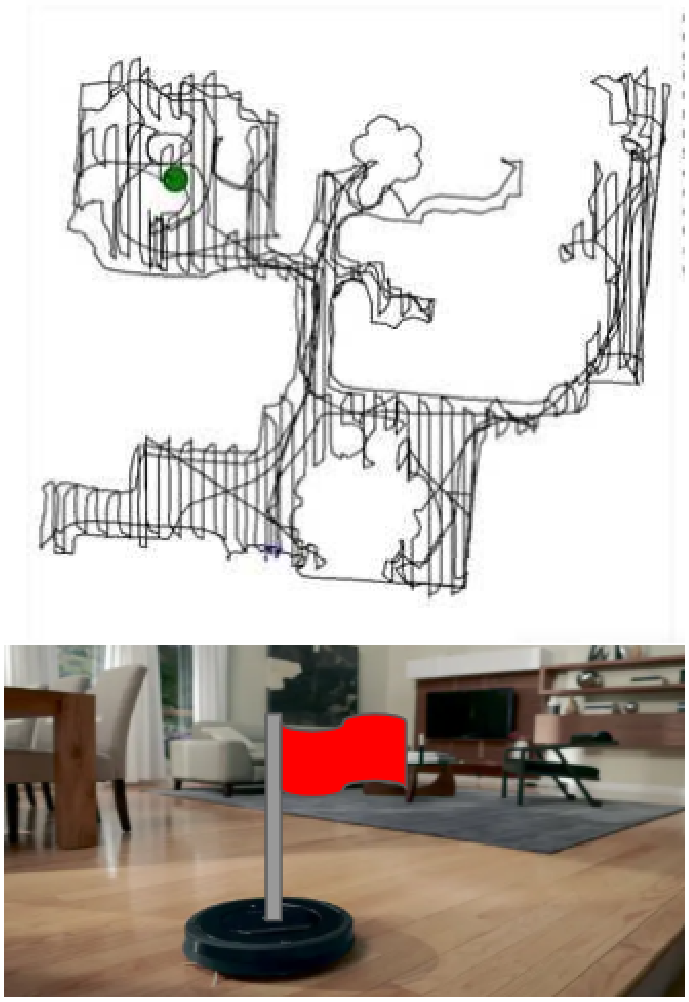


Figure 4: Initialization phase for mapping. For example, we will map pixel block at (820, 210) from the image below to position (56, -54) on the 2D map shown above.

The 3D map is generated almost at the same time as the 2D map. As the robot travels around the room, it will report its estimated XY-location to the Xavier via websockets. At the same time, the camera feed will detect the robot and report its location within its frame. These two data are then essentially connected to generate a rough one-to-one sketch of the 2D-3D map translation. Due to the resolution and fps limitation, we will group neighboring pixels within the camera image into a bin to discretize the frame. This will allow for more lenient and rational 3D mapping.

5.5 Subsystem E: Communication Interfaces

A. Camera System/Server Interface

The interface between the camera system and web server will be mostly one-directional. During the initialization phase, the server will send information about the 2D map position of the robot to the camera system via websocket in the form of JSON with fields for coordinates. This information will be used to create a mapping from camera views to 2D map positions. After initialization, JSON will be sent to the server via websocket with fields for the recognized gesture, and the estimated 2D coordinates of the user and robot based on the previously stored camera view mapping. The server will use its map to convert the gestures into a series of commands to relay to the robot, while using the position information to update its global map positions for the user and robot.

B. Server/Robot Interface

The interface between the web server and robot will be bi-directional. During the initialization phase, the robot will send information about its position periodically in the form of JSON to the server via websocket to be relayed to the camera system. After initialization, the server will send commands to the robot, which will then execute those commands and respond with estimated position updates as JSON to the server. This information will be used by the server to update its global map.

6 PROJECT MANAGEMENT

6.1 Schedule

We organized our project management to achieve three major milestones: MVP (teleop and going to home), mapping, and pointing. We have designated general tasks to achieve the milestones, and we plan to break the tasks down into more detail 2 weeks before we work on something. We felt like it was difficult to plan with small granularity ahead of schedule, and what we are learning working on the project will help us break down future tasks better. A detailed schedule will be attached at the end of the report.

Milestone 1: MVP Teleop and Driving to home
We will have setup and built the base components of our

project: attaching the camera system, running OpenPose on the Xavier, classifying easy static gestures, setting up a webserver with websockets, communicating to the Roomba via the Rpi and serial, and executing basic motor commands on the Roomba. This allows us to understand the base system before building more complicated functionality.

Milestone 2: Mapping and Localization We will work towards building a 2D map on the Roomba using encoders, synchronizing the 2D map with the 3D camera view, and figuring out a system to consolidate map changes. This allows us to build the map infrastructure and localization required for advanced tasks like going to the user or going to a point.

Milestone 3: Driving to point We believe pointing will be the most complex part of the project. We need to figure out where the user is pointing, place that onto the map, and have the robot drive to it. Additional testing is also required to figure out what method (heuristics or model) we will use to find the location a user points to.

6.2 Team Member Responsibilities

Seungmin will be in charge of the robot. He has prior experience with robotics and is a TA for Intro to Robotics. He has led the development on communication to the Roomba via the RPi and the Roomba 2D mapping with Odometry. In addition, he will work on the path planning of the robot and building the hardware for the robot.

Rama will be responsible for system devops, making sure the overall system runs smoothly. He has prior experience working with Linux systems and webservers through his research and previous internships. He has been responsible for setting up our boards, network infrastructure, webserver, and websockets. He will also be making sure the entire system falls within our performance requirements and perform optimizations if necessary. In addition, Rama will work on updating the global map.

Jerry will be responsible for the image processing and gesture recognition. He has prior experience working with machine learning and computer vision through personal projects and internships. He has been responsible for our camera setup, building data pipelines to collect data, ML models to classify gestures, and localizing the robot with the camera. In addition, Jerry work on the global map updates and system infrastructure with Rama.

We will all work together for the testing of our system and making sure we fall within our requirements.

6.3 Budget

Table 2: Bill of Materials

Component	Quantity	Cost
Roomba 671	1	243.51
Roomba DIN to USB cable	1	23.99
Raspberry Pi 4	1	45.67
RaspPi Power Source	1	9.99
Wide angle camera	1	26.99
USB Webcam	2	34.98
MicroHDMI cable	1	6.49
USB extension cable (10ft)	2	11.58
USB extension cable (25ft)	1	7.69
USB Hub	1	27.99
Total:		438.88
Budget Left:		161.12

Table 3: Sourced and scrounged parts

Component	Source
NVIDIA Jetson Xavier	Prof. Savvides
Monitor	Lab
USB mouse	Lab
USB keyboard	Personal
Ethernet cable	Personal
HDMI Cable	Personal
Battery pack	Personal
MicroUSB power source	Personal

Table 4: Tools

Tool	Purpose
OpenCV	Webcam and image processing
OpenPose	Keypoint recognition
pyserial	Communicating with Roomba
sklearn	Training SVM models
Tensorflow	Training Nueral network models
Express	Node.js webserver
AWS GPU p2.xlarge	ML model training

6.4 Risk Management

The largest risk for our project is localization of the robot. Our tasks of going back to home, going to the user, and going to the pointed location all require the robot to know where it is on the map. We are trying to mitigate the risk by using multiple methods to localize the robot. We are using data from our motor encoders to know where the robot has traveled on the 2D map. Additionally, we are going to use the camera view and our camera 3D to 2D mapping in order to get a location of the robot in the room. By having two methods to localize the robot, we can maximize the chances of localization.

Classifying the gestures incorrectly is also a risk. OpenPose can give us incorrect keypoints, which would cause an error in the classificaiton process. To address this, we are using multiple cameras to capture the user from multiple angles in the room. So, we have backup views of the user to classify gestures. Running OpenPose on more cameras decreases our total FPS, so our system can only have at most 3 cameras. We chose 3 cameras to balance the performance of our system and the cost of the hardware required with the accuracy we can get in gesture classification. in addition, we will have backup heuristics to classify the gestures if our system can not confidently recognize a gesture.

6.5 AWS Credit Usage

We have used around $3 in AWS credits with p2.xlarge instances with GPU to test OpenPose and test ML models.

References

[1] Zhe Cao et al. "OpenPose: realtime multi-person 2D pose estimation using Part Affinity Fields". In: *arXiv preprint arXiv:1812.08008*. 2018.

Appendix A

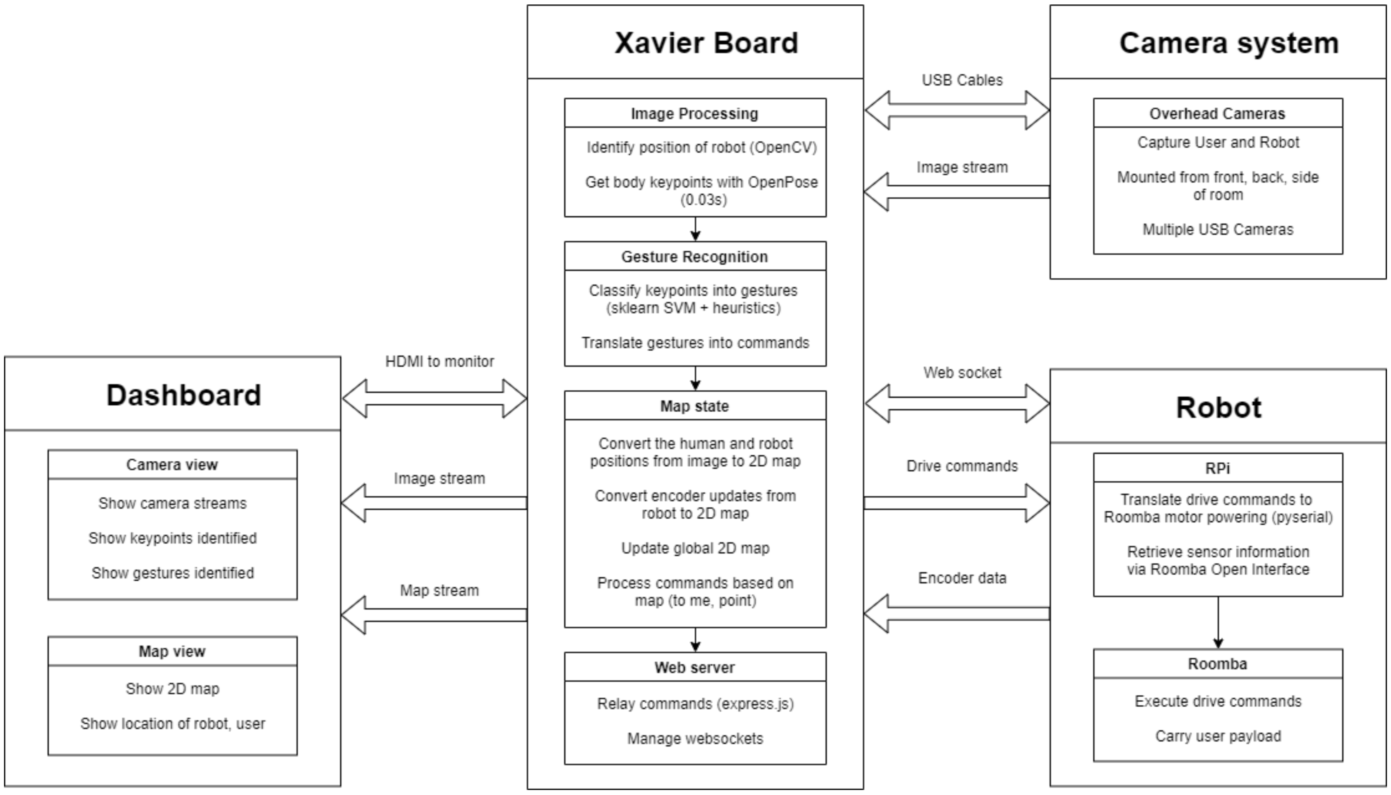
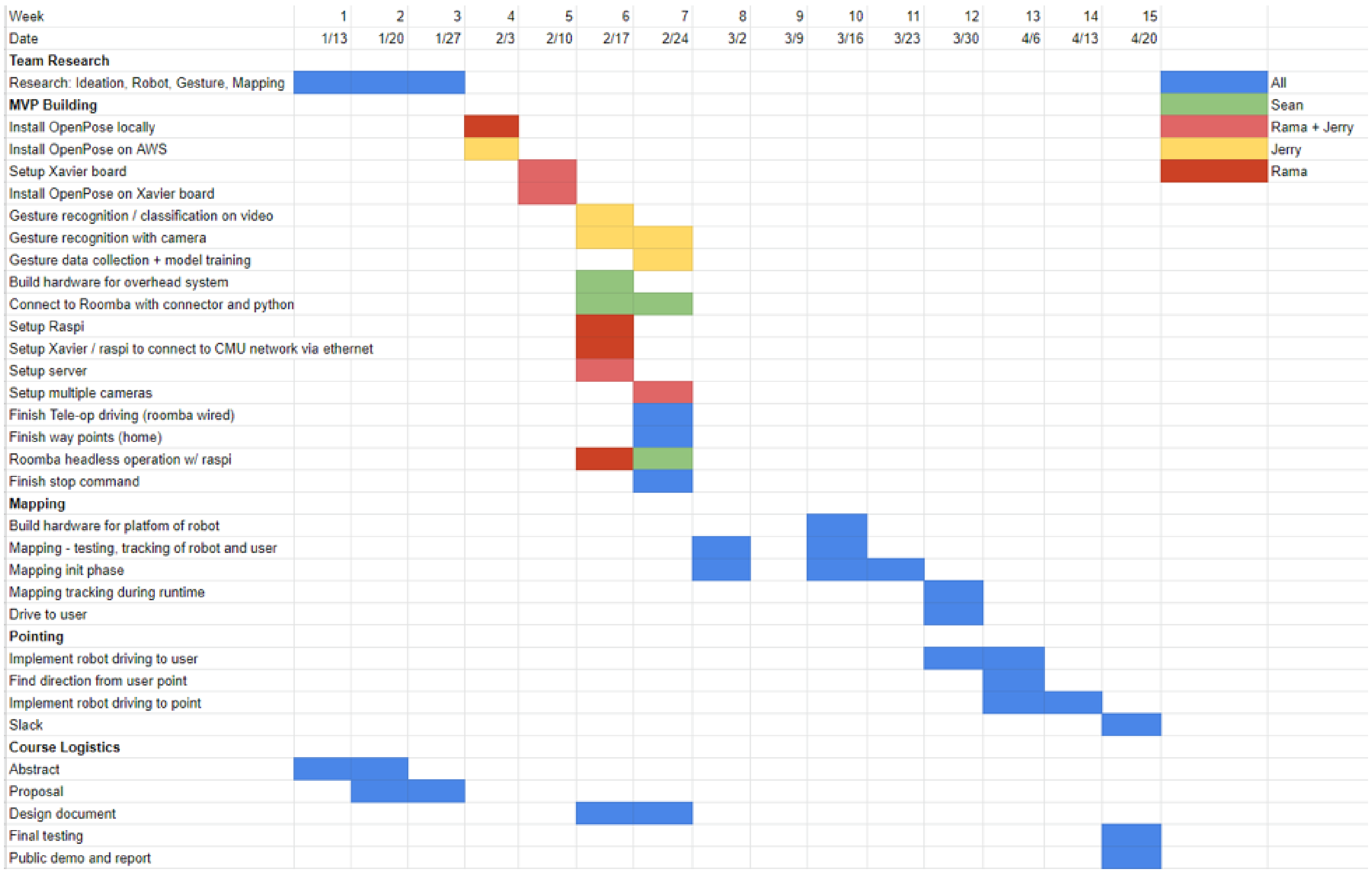

Xavier Board
Image Processing
Identify position of robot (OpenCV)
Get body keypoints with OpenPose (0.03s)
Gesture Recognition
Classify keypoints into gestures (sklearn SVM + heuristics)
Translate gestures into commands
Map state
Convert the human and robot positions from image to 2D map
Convert encoder updates from robot to 2D map
Update global 2D map
Process commands based on map (to me, point)
Web server
Relay commands (express.js)
Manage websockets
Camera system
Overhead Cameras
Capture User and Robot
Mounted from front, back, side of room
Multiple USB Cameras
USB Cables
Image stream
Web socket
Drive commands
Encoder data
Robot
RPi
Translate drive commands to Roomba motor powering (pyserial)
Retrieve sensor information via Roomba Open Interface
Roomba
Execute drive commands
Carry user payload
HDMI to monitor
Image stream
Map stream
Dashboard
Camera view
Show camera streams
Show keypoints identified
Show gestures identified
Map view
Show 2D map
Show location of robot, user

Figure 5: Overall system architecture



Week	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
Date	1/13	1/20	1/27	2/3	2/10	2/17	2/24	3/2	3/9	3/16	3/23	3/30	4/6	4/13	4/20		
Team Research																	
Research: Ideation, Robot, Gesture, Mapping	🟦	🟦	🟦													🟦	All
MVP Building																🟩	Sean
Install OpenPose locally				🟥												🟧	Rama + Jerry
Install OpenPose on AWS				🟨												🟨	Jerry
Setup Xavier board					🟧											🟥	Rama
Install OpenPose on Xavier board					🟧												
Gesture recognition / classification on video						🟨											
Gesture recognition with camera						🟨	🟨										
Gesture data collection + model training							🟨										
Build hardware for overhead system						🟩											
Connect to Roomba with connector and python						🟩	🟩										
Setup Raspi						🟥											
Setup Xavier / raspi to connect to CMU network via ethernet						🟥											
Setup server						🟧											
Setup multiple cameras							🟧										
Finish Tele-op driving (roomba wired)							🟦										
Finish way points (home)							🟦										
Roomba headless operation w/ raspi						🟥	🟩										
Finish stop command							🟦										
Mapping																	
Build hardware for platfom of robot										🟦							
Mapping - testing, tracking of robot and user								🟦		🟦							
Mapping init phase								🟦		🟦	🟦						
Mapping tracking during runtime												🟦					
Drive to user												🟦					
Pointing																	
Implement robot driving to user												🟦	🟦				
Find direction from user point													🟦				
Implement robot driving to point													🟦	🟦			
Slack															🟦		
Course Logistics																	
Abstract	🟦	🟦															
Proposal		🟦	🟦														
Design document						🟦	🟦										
Final testing															🟦		
Public demo and report															🟦		

Figure 6: Gantt chart