Cookiebot: A Gesture-Based Home Robot

Authors: Seungmin Ha, Rama Mannava, Jerry Yu: 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 capture the user and the robot. Images are streamed to an NVIDIA Jetson Xavier to run OpenPose and classification models to classify gestures. Gestures are relayed via a Node.js webserver to the robot. The robot has a Raspberry Pi and Roomba to execute commands.**

***Index Terms*—Homography, Home assistant, Gesture recognition with ML, Image processing, Localization, Low latency ML inference, Robotics, Websockets**

Video demo: `https://youtu.be/AF8zmTaa17s`

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 to be comparable to a Google Home.
2. Recognize gestures with at least a 90% percent success rate and at most a 10% false positive rate to minimize user frustration.
3. Maintain less than 0.3 meters (1 foot) drift for the robot and user positions 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, and keypoint classification using SVMs. It also hosts a webserver for low latency local network websocket communication between with the robot. We also built 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 is constantly updated during runtime to have multiple methods of maintaining accurate robot and user localization.

2 DESIGN REQUIREMENTS

Our design requirements fall into two broad categories: time, and accuracy. The time requirements regulate the latency of the user experience, while the accuracy requirements are goals for the ultimate usability of the system. Overall, both sets of requirements are essential to define reasonable expectations for this project.

A. Time Requirements

The goal of the time requirements is to regulate the latency of the various sections of the system. The most important constraint is the end-to-end response time. The response time of a Google Home, one of the most commercially successful Smart Home products, takes about 1.9 seconds from user input to response. Since we want this project to be an extended and improved version of the smart home concept, we settled on an overall response time of 1.9 seconds as well.

The main sources of latency are the gesture recognition and tracking, as well as the path planning that the robot must perform before executing the given command. Although the specifics of how the time budget is allocated to these three tasks are not important, we decided to establish some reasonable guidelines to help us ensure that we would reach our overall goal of 1.9 seconds. Both the gesture recognition and tracking occur in parallel, so we allocated 500 milliseconds. As for the path planning, its complexity greatly increases with distance covered, so we decided to give it 1000 milliseconds. Altogether this leaves 400 milliseconds of leeway to account for variation in performance.

For the robot, we also required that the movement speed was above 0.05 meters per second for efficiency but lower than 0.2 meters per second for safety. We wanted the system to run for over 2 hours without recharging, so that it would be operational for a reasonable period of use.

B. Accuracy Requirements

The accuracy requirements are in place to guide the overall usability of the project, minimizing frustration from inaccurate gestures and discomfort from inaccurate localization.

We settled on a target success rate of 90% for the gesture recognition. This includes a false positive rate of under 10% and a false negative rate under 1%. With these metrics, the system would misclassify a gesture once in ten attempts, and would fail to recognize a gesture once out of one hundred attempts. We anticipate that aiming for this level of accuracy would greatly reduce user frustration, as a one-in-ten chance of having a gesture misunderstood would be a rare enough occurrence that it would not be annoying to redo the gesture, while the system would very rarely fail to respond to a gesture.

The localization accuracy requirements are twofold. The first requirement is for robot and user drift during usage. We defined drift to be the difference in estimated location and actual location in the room. We want the user and robot to always be within easy reaching distance, so we required that the drift for the robot and user stay under one foot. In this way, the worst-case scenario of the robot and user drifting in opposite directions would still result in a maximum drift of two feet, which is still within average reaching distance. The second requirement is for the map generated during the initialization phase. The robot explores the room to generate a map of the spaces that it can visit, as well as the obstacles in the room. We wanted the robot to have high map accuracy to increase the efficiency of its path planning, so we set a goal of 90%.

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: visualizations, Xavier, camera system, and the robot. The diagram clearly illustrates how the components interact.

A. Camera System

The camera system is composed of a single USB webcam. We originally wanted to use multiple webcams to capture different angles, but the COVID-19 situation caused the same 17 dollar webcams to be priced at over 100 dollars. As a result, we needed to have the user and the robot in frame of the same camera. Gestures additionally need to be done facing the camera. The camera captures the user and robot and send the data to the Xavier via low latency USB. When we moved to remote, we had to do this processing offline, but the functionality via USB is still implemented.

B. NVIDIA Jetson Xavier

The Xavier is 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 receives updates from the robot about its location. In addition, the board will use the video feed from the camera system to translate the 3D representation of the room to the 2D map 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 hosts a web server established using express.js. In addition, it manages websockets used to communicate with the robot subsystem. Having robust server and websocket management is 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 requires less RPC calls and is in our local network.

C. Robot System

The robot subsystem is has two main components: Raspberry Pi and Roomba. These 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 inputs by the 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 the 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 headless mode, requiring all commands to be given to the Raspberry Pi through the web server.

D. Visualizations

Lastly, we built visualizations to show the current state of the entire system. The Xavier relays the camera view and map view to the monitor which is connected to the board via HDMI cable. The camera view includes the current camera stream as well as the gestures and key points recognized. The map view, on the other hand, shows the estimated locations of the robot and user on the 2D map. Using these visualizations, we were also able to debug the system more easily. The user is also able to get a visual and intuitive update of the system status. We found that having live feedback was extremely useful for gestures like the point, where users could understand when the system made a mistake and could change their gesture such that the system could classify it correctly. We have also based our video demo on these visualizations.

4 DESIGN TRADE STUDIES

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

4.1 Camera System

Both the user and the robot need to be in frame for the camera. In addition, the camera needs to be stationary so we can re-use the 3D to 2D mapping generated during the initialization phase. We originally wanted to cover the entire room from all angles, but we decided to just use one webcam due to the COVID-19 situation causing camera prices to skyrocket.

From our testing, we found that using multiple cameras would give us higher reliability compared to using 3D cameras, which are much more expensive and harder to work with. Multiple webcams would give us views of the room along multiple planes to differentiate gestures that look identical when viewed on the same 2D plane. This is especially helpful for similar gestures like the point. Our final camera system uses one front-facing camera and requires the user to face the camera when gesturing.

4.2 Keypoint Recognition

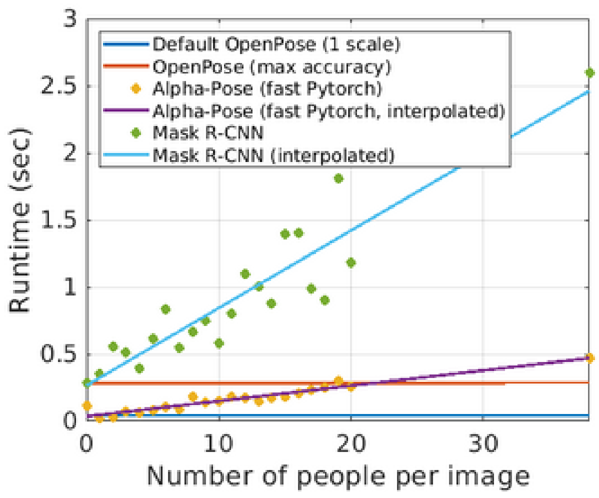


Figure 1: Runtime comparison of similar keypoint recognition algorithms. [1]

Recognizing gestures requires an algorithm that is able to identify keypoints when given an image of a person. This is a popular computer vision problem known as 2D pose estimation. 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.

Additional consideration was put into using networks with 3D pose estimation, since knowing the keypoints in 3D could ease the classification of gestures. [2] However, we were worried we would not be able to get the same level of runtime performance as OpenPose on embedded systems. The paper mentions a similar architecture to Mask R-CNN, which OpenPose strongly outperforms. OpenPose also has robust python bindings that make it easy to use, whereas 3D methods would require more configuration. 3D pose estimation is, however, an area for future work on this project.

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 remaining 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.

Note in the chart below, we did not enable OpenPose optimizations like the –tracking flag that gave us a 2x speedup on the Xavier board. We were also running an older version of OpenPose on AWS since it was the version a docker container was released at.

Table 1: Comparison of OpenPose Performance

Hardware	CPU Cores	CUDA Cores	FPS
CPU-Only	8	0	0.05
GCP Instance	2	4992	3.5
AWS p2.xlarge	4	4992	2.5
NVIDIA Jetson Xavier	4	512	17
NVIDIA Jetson Xavier	8	512	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 communicates with the server and converts 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

There are five main subsystems that comprise our project.

5.1 Subsystem A: Gesture Recognition

The gesture recognition subsystem determines the user's gestures and relays the appropriate command to the webserver. It first collects an image from the webcam containing both the robot and the user. We chose to use the Logitech C270 webcam since it was the cheapest USB webcam we could find and we could work with its 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 returns 25 keypoints across the body for us the classify, with the XY-position in the frame along with the confidence level. 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.

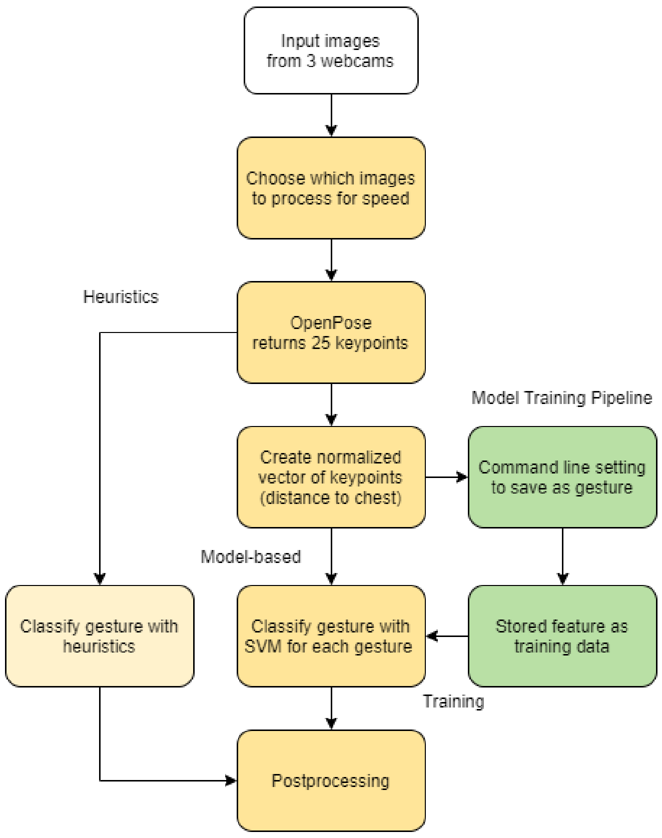

Input images from 3 webcams
Choose which images to process for speed
Heuristics
OpenPose returns 25 keypoints
Model Training Pipeline
Create normalized vector of keypoints (distance to chest)
Command line setting to save as gesture
Model-based
Classify gesture with heuristics
Classify gesture with SVM for each gesture
Stored feature as training data
Training
Postprocessing

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.

We ended up using heuristics for the to me, stop, go home, and point ready gestures. We used heuristics to determine if an arm was raised for teleop mode, and we used a SVM to classify the direction and if the gesture was for driving straight, turning left, or turning right.

In addition, we used post processing to ensure a gesture was stable before we sent it to the robot. We did this by using a sliding window and ensuring we saw a gesture for 10 frames before we sent it to a robot. In addiiton, we defined that the system needs to be at no gestures detected before a gesture is detected. This is done for all gestures except for switching between teleop.

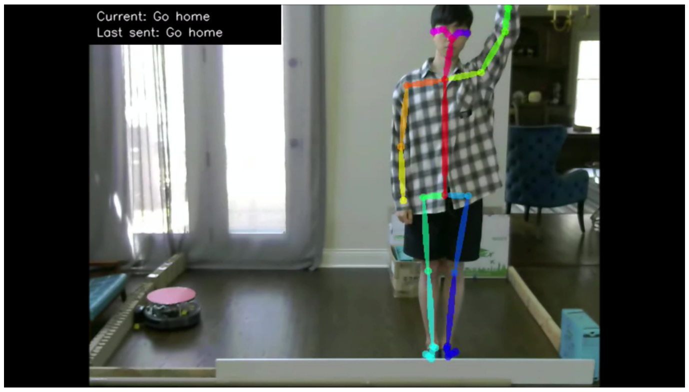


Figure 3: System recognizing go home, left hand up

5.2 Subsystem B: Getting location from point

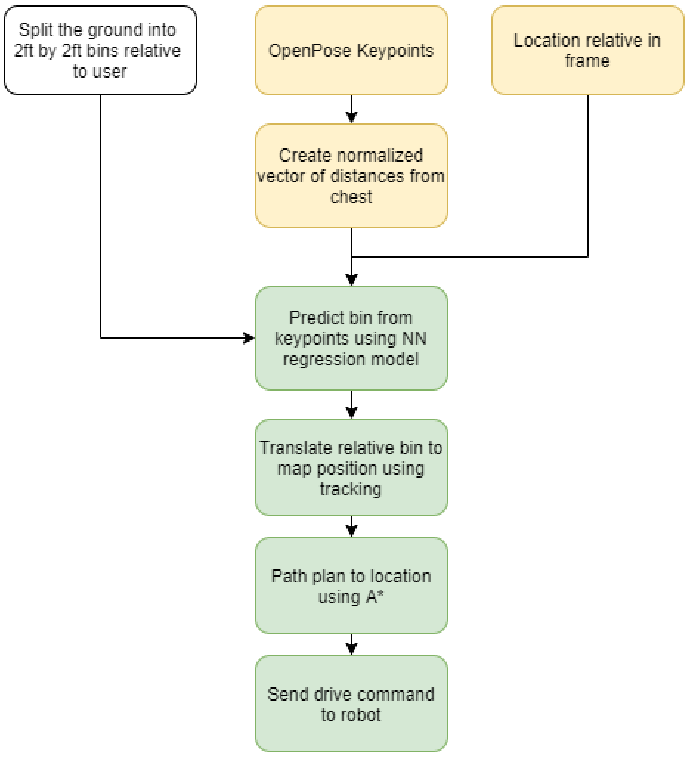

Split the ground into 2ft by 2ft bins relative to user
OpenPose Keypoints
Location relative in frame
Create normalized vector of distances from chest
Predict bin from keypoints using NN regression model
Translate relative bin to map position using tracking
Path plan to location using A*
Send drive command to robot

Figure 4: Pointing system

Pointing to a location on the ground uses a neural network. We first divide the room into a grid of 1 foot square bins. We then gather data by having the user point to a bin and mapping those keypoints to the appropriate bin. We wrote custom scripts to get video and label it. Around 1 hour of training and validation data was gathered for the point. This was converted into a dataset of 14,400 training frames and 6,000 validation frames, with training and validation data coming from different videos. The model predicts the x and y coordinate of where the bin is on the grid using a neural network regression model, since bins close together have more similar keypoints.

We explored many methods of building the network; the final architecture uses a 3 layer 64 hidden unit neural network with two regression multitask outputs of x and y for the bins. For the input of the network, we found improvements using only the arm keypoints and including the location of the user in the frame, as the same point gesture can look different in various locations of the room. We trained all the neural networks until the loss stabilized (100 epochs) with the Adam optimizer and we chose the model that performed the best.

Table 2: Point model validation performance

Architecture	Validation accuracy
Categorical NN	0.9006
Regression NN	0.9124
Regression NN with Location	0.9456

After getting the relative location, we can translate it to the 2D map using the position of the user from tracking. We can then tell the robot to plan a path towards that location using A*.

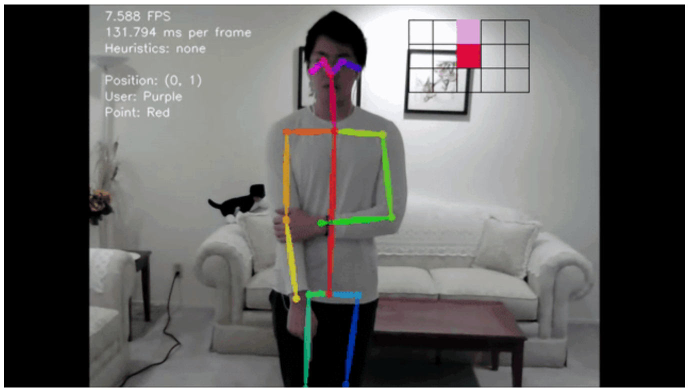


Figure 5: Visualization of pointing in front

5.3 Subsystem C: Robot

A. Components

The robot subsystem is composed of a Raspberry Pi and the Roomba 671 itself. Additionally the battery pack (10,000mAH) powers the Raspberry Pi independent of the Roomba power source. The Raspberry Pi and the battery pack are attached on top of the robot. It is important not to obstruct any sensors while attaching these components since some sensors (IR sensor, etc.) are placed on top of the Roomba. A red marking disk is 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 use the encoder values of the Roomba to calculate the distance and the angle it moves each time stamp. The distance travelled is calculated using the 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 the 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. Map Generation

For the map generation, the robot first follows the wall to collect information about the boundary of the environment. Then, based on the dimension of the room, the robot moves 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 operates autonomously without any direction from the user. This necessitates 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 correctness of the wall-following process and ensure that the robot doesn't get trapped within a 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.

D. Path finding

Based on the generated 2D map, the robot needs to calculate a path to the desired goal location. Since we generate the 2D map by occupying a 0.05m-square grid, the best approach was to use a graph search algorithm such as A* or Djikstra's algorithm. We eventually decided to use A* algorithm with the diagonal distance as a heuristic. The diagonal distance guarantees admissibility as well as consistency on a 8-connect grid representation of the space. Other variations of the A* algorithm were tested, such as weighted-A*, but there was no need to trade computation speed for optimality since the worst case computation speed was well within the requirement we set for an acceptable usability.

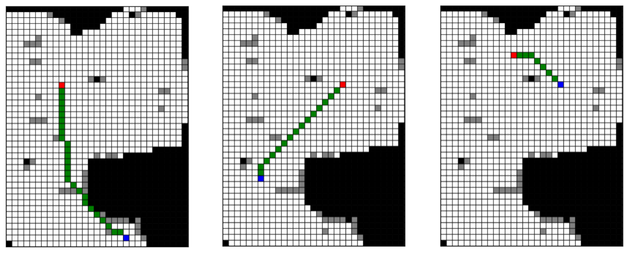


Figure 6: Three example paths generated using A* algorithm and the 2D map. The white cell represents an empty space, black cell represents an obstacle, and grey cells are the space which were visited only once during the mapping phase. The red cell represents the starting point, blue cell represents the goal, and green cells represent the path the robot will take.

5.4 Subsystem D: 3D to 2D Mapping

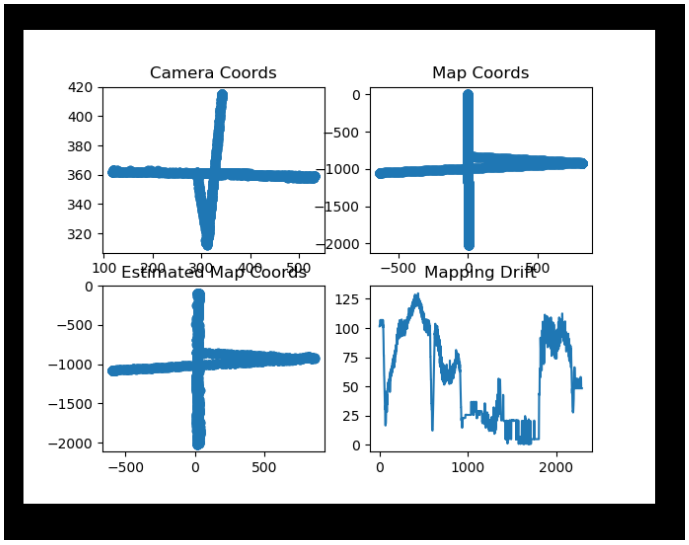


Figure 7: The results of the 3D to 2D Mapping. The first row shows the camera position of the robot as well as its position in the room according to its encoder data. The second row contains the estimation of the map positions using the generated homography, and the relevant drift in millimeters.

The 3D to 2D Mapping is generated by building a homography. This is a transformation between two planes; in this case, the planes are the camera's perspective of the floor, and the robot's 2D map view. The larger the set of corresponding coordinates used, the greater the accuracy of the homopraphy. The encoder data from the map generation can be used for this purpose, but a smaller set from simply sweeping both axes of the room satisfies accuracy requirements as well. After this mapping is generated, the websever is able to use this information to track the positions of both the robot and the user on the 2D map provided by the robot. The same mapping can be used as long as a position in the camera view is known, so both the robot and user can be localized with this same set of information.

5.5 Subsystem E: Communication Interfaces

A. Camera System/Server Interface

The interface between the camera system and web server is mostly one-directional. During the initialization phase, the server sends 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 is used to create a mapping from camera views to 2D map positions. After initialization, JSON is 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 uses 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 is bidirectional. During the initialization phase, the robot sends 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 sends commands to the robot, which then executes those commands and responds with estimated position updates as JSON to the server. This information is used by the server to update its global map.

6 VALIDATION AND METRICS

6.1 Gesture Recognition

For gesture recognition, the key metric we aimed for was classification accuracy. As mentioned in our requirements, we aimed for 90% accuracy to make our system reasonable to use. In all of our tests, we had the user face the camera and the user in frame. We also tested under different lighting conditions, ranging from having the lights on to having sunlight through the window. For the teleop gestures (left, straight, right) we had 90.8% test accuracy over 76 gestures. For the point classification, we had 83.08% test accuracy over 130 gestures. The point classification was harder than we expected, as very similar keypoints could lead to different bins and the same gesture could look different at various locations in the room. For the heuristics gestures, our test accuracy was 95.6%. In total, we achieved 90.2% test classification accuracy over 368 test gestures, reaching our goal of 90% test accuracy. We had a 8.8% misclassification or false positive rate, along with a 0.9% unrecognized or false negative rate. This met our goal of 10% misclassification and 5% unrecognized rate. We chose 5% as our unrecognized rate because we believed that false negatives were worse for the user experience, as the system does not acknowledge what a user does.

Table 3: Gesture Recognition Metrics

Gesture Type	Requirement	Test Accuracy
Teleop	90%	90.8%
Point	90%	83.8%
Heuristics	90%	95.6%
Overall	90%	90.3%

6.2 Tracking

The tracking performed using the 3D to 2D Mapping was constrained by the accuracy of location estimates over the course of system usage. Using around 3000 frames of data, we compared the server's estimate of the robot's position with its actual position from encoder data. We also used the same dataset to compare the estimated user position with the ground truth position found from markings placed on the floor of the test area. The robot had an average drift of around 6 centimeters, while the user had an average drift of 4 centimeters. The goal for drift was to keep it within 300 centimeters (1 foot) for both the robot and user, and we ultimately had much higher accuracy than we had hoped for.

Table 4: Tracking Metrics

Subject	Requirement	Drift
Robot	30 cm	6 cm
User	30 cm	4 cm

6.3 Robot

There were three main metrics we wanted to test for the robot manipulation. The first was the odometry accuracy. Throughout the operation, we heavily rely upon the odometry-generated estimated position of the robot; for instance, the path finding algorithm would be useless if the robot is not actually where we think it is. Thus, it was important to ensure that the odometry is generating a fairly accurate estimation of the robot position. For the validation, we made the robot drive to five different positions and measured the discrepancy between its actual position which was measured using a tape-measure, and its reported estimation of the current location. The result came out to be 4.8cm on average which was well within our requirement of 30cm.

The second metric we wanted to test was the map accuracy. During the mapping phase, the robot explores the room and generates a 2D map which is used in the path-finding as well as in the 2D-3D mapping. Thus, we wanted to minimize any misrepresented space on the map to ensure complete and consistent space representation. To test this, we counted misrepresented cells on the generated 2D map of a known environment. The resulting number of misrepresented cells was 56 out of 1271 cells, which meant 95.6% accuracy.

The final metric was the average speed of the robot. We wanted the robot to move fast enough to complete tasks efficiently, but also wanted to limit the speed of the robot to ensure more accurate odometry as well as to guarantee user safety. After testing with different speeds with the robot, we decided that the robot must move at speed between 5 cm/s and 20 cm/s for usability and safety. To validate, we made the robot perform 10 different tasks each requiring around 10 m of movement and measured the time it takes to complete the task. The resulting average speed was 11 cm/s which was within our desired range of speed.

Table 5: Robot manipulation Metrics

Criteria	Requirement	Test Result
Odometry accuracy	30 cm	4.8 cm
Map accuracy	90%	95.6%
Average speed	< 20 cm/s > 5 cm/s	11 cm/s

6.4 Performance

For performance, we aimed to have the entire system from user action to the robot starting to execute the command be under 1.9 seconds, the average response time for a Google Home. We first broke down our expectations for the performance based on each of our components.

For gesture recognition, we measured the latency from before the frame is captured to before the command is sent to the robot. The average latency was 56 milliseconds and the longest time was 261 milliseconds. after further time profiling, we found that this was due to outliers in openpose taking 150 milliseconds to run (usually 30 milliseconds) and the point model could take up to 100 milliseconds to run. we found the gesture recognition times by taking an average over 129 gestures.

mapping processing was the time it took to track both the robot and user in a frame. on average, it took 16 milliseconds with the longest time being 166 milliseconds from an outlier in openpose. this was tested by running the tracking on 3072 frames. as both gesture recognition and the tracking occur in parallel, the greater latency of the gesture recognition supersedes the latency of the tracking.

path planning with a* took 230 milliseconds on average, with 780 milliseconds being the time to search for a path from one corner of the room to the other corner. we tested this by taking the average time over 6 runs. although this was by far the longest step in the process, it was still within our budgeted time.

to test the rpi to server socket latency, we simulated 100 sends of location and receives of commands. the average time was 3.5 milliseconds and longest time of 13.5 milliseconds. this was faster than what we expected because we were running on a local network. we also measured the rpi execution time over 100 cycles, from receiving the command to sending the motor commands. we chose the requirement of 15 milliseconds because it is the refresh rate of the sensor data we get from the roomba. we were able to get an average time of 13.7 milliseconds and longest time over 16.2 milliseconds.

In total, we took the max of gesture recognition and mapping since we could perform them in parallel, and summed the result with the rest of the components. This led to our average response time of 0.3 seconds and longest response time of 1.07 seconds. This fits well under our goal of 1.9 seconds. We believe this response time is a result of the strong processing of the Xavier board both on the GPU and CPU fronts that allowed us to meet our performance goals.

Table 6: Performance metrics

Component	Requirement	Average	Longest
Gesture recognition	500ms	56ms	261ms
Mapping processing	100ms	16ms	166ms
Path planning	1000ms	230ms	780ms
RPi socket	100ms	3.5ms	13.5ms
RPi execution	15ms	13.7ms	16.2ms
Total time	1.9s	0.3s	1.07s

7 PROJECT MANAGEMENT

7.1 Schedule

We organized our project management to achieve three major milestones: MVP (teleop and going to home), mapping, and pointing. We designated general tasks to achieve these milestones. A detailed schedule is provided in *Appendix A.*

Milestone 1: MVP Teleop and Driving to home We set up and built the base components of our project: attached the camera system, ran OpenPose on the Xavier, classified easy static gestures, set up a webserver with websockets, communicated to the Roomba via the RPi and serial, and executed basic motor commands on the Roomba. This allowed us to understand the base system before building more complicated functionality.

Milestone 2: Mapping and Localization We worked 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 allowed 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 Pointing was the most complex part of the project. We needed to figure out where the user is pointing, place that onto the map, and have the robot plot and follow a path to it.

7.2 Team Member Responsibilities

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

Rama was 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 was responsible for setting up our boards, network infrastructure, webserver, and websockets. He also made sure the entire system fell within our performance requirements and performed optimizations when necessary. In addition, Rama worked on updating the global map via camera tracking.

Jerry was 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 was responsible for our camera setup, and building data pipelines to collect data and ML models to classify gestures. In addition, Jerry worked on system infrastructure and performance with Rama.

We all worked together for the testing of our system and made sure we fell within our requirements.

7.3 Budget

Table 7: 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 8: Tools

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

Table 9: 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

7.4 Risk Management

The largest risk for our project was localization of the robot. Our tasks of going back to home, going to the user, and going to the pointed location all required the robot to know where it was on the map. We tried to mitigate the risk by using multiple methods to localize the robot. We used data from our motor encoders to know where the robot had traveled on the 2D map. Additionally, we used 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 maximized the accuracy of the localization.

Classifying the gestures incorrectly was also a risk. OpenPose can procide incorrect keypoints, which would cause an error in the classification process. To address this, our model-based approaches had backup heuristics to classify the gestures if our system could not confidently recognize a gesture. In addition, we uses post processing on the gesture output to ensure that a gesture was not caused by noisy OpenPose output.

7.5 AWS Credit Usage

We have used around $10 in AWS credits with p2.xlarge instances with GPU to test OpenPose and test ML models. We lost access to our Xavier board briefly and needed to get OpenPose set up on AWS to run our system.

References

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

[2] Dario Pavllo et al. "3D human pose estimation in video with temporal convolutions and semi-supervised training". In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019.

Appendix A

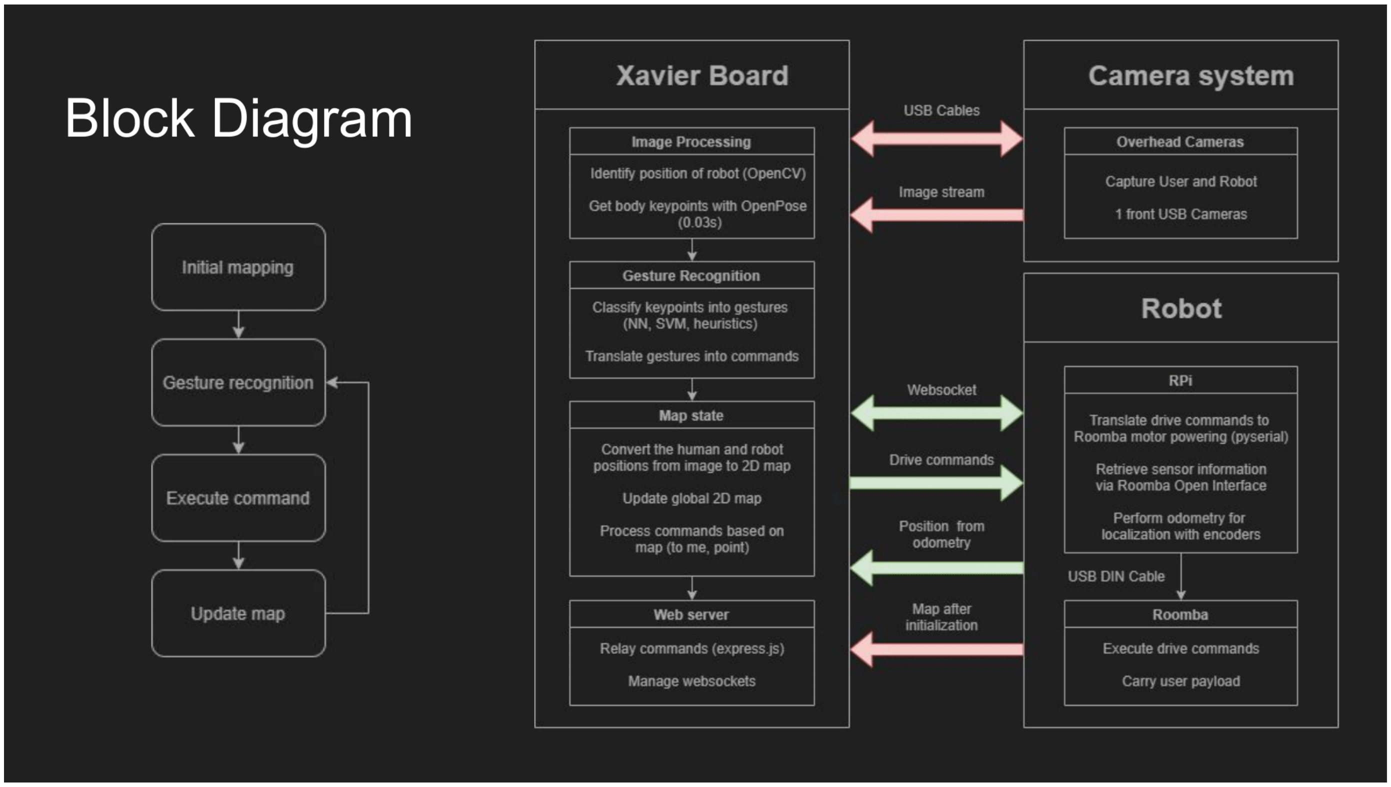
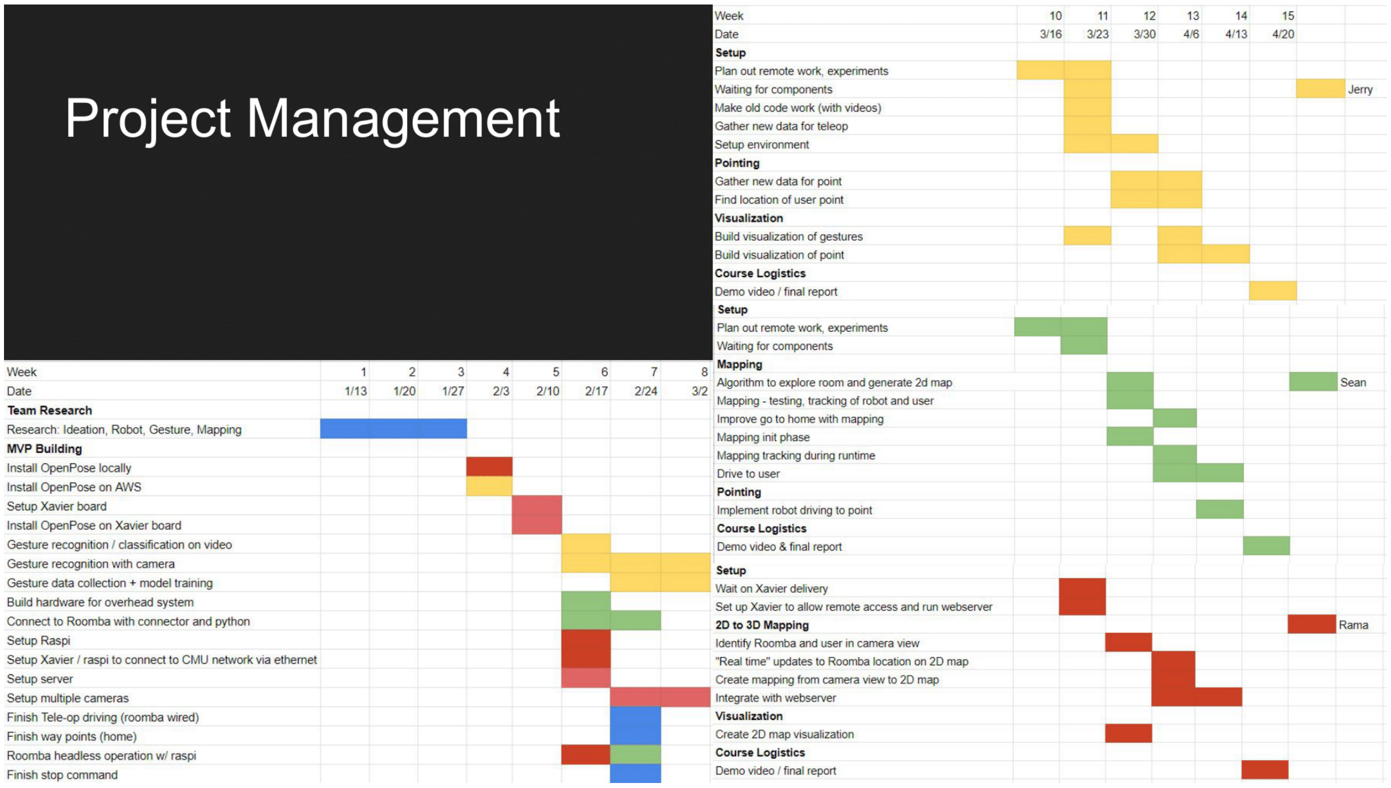

Block Diagram
Initial mapping
Gesture recognition
Execute command
Update map
Xavier Board
Image Processing
Identify position of robot (OpenCV)
Get body keypoints with OpenPose (0.03s)
Gesture Recognition
Classify keypoints into gestures (NN, SVM, heuristics)
Translate gestures into commands
Map state
Convert the human and robot positions from image to 2D map
Update global 2D map
Process commands based on map (to me, point)
Web server
Relay commands (express.js)
Manage websockets
USB Cables
Image stream
Websocket
Drive commands
Position from odometry
Map after initialization
Camera system
Overhead Cameras
Capture User and Robot
1 front USB Cameras
Robot
RPi
Translate drive commands to Roomba motor powering (pyserial)
Retrieve sensor information via Roomba Open Interface
Perform odometry for localization with encoders
USB DIN Cable
Roomba
Execute drive commands
Carry user payload

Figure 8: Overall system architecture


Project Management

Week	1	2	3	4	5	6	7	8
Date	1/13	1/20	1/27	2/3	2/10	2/17	2/24	3/2
Team Research								
Research: Ideation, Robot, Gesture, Mapping	🟦	🟦	🟦					
MVP Building								
Install OpenPose locally				🟥				
Install OpenPose on AWS				🟨				
Setup Xavier board					🟥			
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							🟦	

Week	10	11	12	13	14	15		
Date	3/16	3/23	3/30	4/6	4/13	4/20		
Setup								
Plan out remote work, experiments	🟨	🟨						
Waiting for components		🟨					🟨	Jerry
Make old code work (with videos)		🟨						
Gather new data for teleop		🟨						
Setup environment		🟨	🟨					
Pointing								
Gather new data for point			🟨	🟨				
Find location of user point			🟨	🟨				
Visualization								
Build visualization of gestures		🟨		🟨				
Build visualization of point				🟨	🟨			
Course Logistics								
Demo video / final report						🟨		
Setup								
Plan out remote work, experiments	🟩	🟩						
Waiting for components		🟩						
Mapping								
Algorithm to explore room and generate 2d map			🟩				🟩	Sean
Mapping - testing, tracking of robot and user			🟩					
Improve go to home with mapping				🟩				
Mapping init phase			🟩					
Mapping tracking during runtime				🟩				
Drive to user				🟩	🟩			
Pointing								
Implement robot driving to point					🟩			
Course Logistics								
Demo video & final report						🟩		
Setup								
Wait on Xavier delivery		🟥						
Set up Xavier to allow remote access and run webserver		🟥						
2D to 3D Mapping							🟥	Rama
Identify Roomba and user in camera view			🟥					
"Real time" updates to Roomba location on 2D map				🟥				
Create mapping from camera view to 2D map				🟥				
Integrate with webserver				🟥	🟥			
Visualization								
Create 2D map visualization			🟥					
Course Logistics								
Demo video / final report						🟥		

Figure 9: Gantt chart