CAPSTONE PROJECT REPORT

LocalRobots2

Submitted by:

Logan Huber, Rose Marie Godoy, Josephine Qualls

Under the guidance of/Submitted to:

Dr. Tony Grichnik

Bradley University, Department of Computer Science

Table of Contents

1. Abstraction

In this Capstone project, we developed an autonomous quadruped robotic system designed to navigate and perform specific tasks in an unstructured environment. The project combines robotics, machine learning, and sensor integration principles to create a robot capable of obstacle detection, path planning, and real-time decision-making.

High-Level Abstraction: At a conceptual level, the robot's main objective is to autonomously navigate an environment, identify objects, and adjust its path to avoid obstacles. The system relies on data from various sensors, processed through a machine learning model, to interpret its surroundings and make navigational choices, enhancing its adaptability in unpredictable settings.

Low-Level Abstraction: Technically, the robot's functionality depends on an ROS (Robot Operating System) framework that integrates LiDAR, ultrasonic sensors, and a camera for environmental data. The machine learning algorithms, developed in Python and implemented through TensorFlow, are responsible for object detection and classification. A C++-based control system manages the motors and decision-making algorithms, ensuring precise movements and real-time adaptability. This abstraction provides insight into the robot's inner workings, from sensor data acquisition to the integration of high-level software for decision-making.

2. Introduction

2.1 Project Overview

The purpose of this Capstone project is to design, build, and program an autonomous robotic system capable of navigating a dynamic environment and performing simple tasks, such as object recognition and obstacle avoidance. We are expanding upon the predecessors of this capstone project that used locally produced robots with RasPi-based canine robots. These robots would be used in the AI and Machine learning courses to allow the students to have hands-on learning and work with a physical system. As autonomous systems are becoming increasingly

relevant across various industries, this project aims to explore the integration of advanced robotic capabilities, combining real-time sensor data processing with machine learning techniques.

Our group consists of two teams:

1. **Physical**

- Built and maintained robot chassis
- Executed, handled, and recorded the factory scripts in the Raspberry Pi's
- Check for deviations in the virtual robot's behavior executing physically
- Documented results of execution of factory settings for navigation and mapping
- Audited and logged data input streams from sensors

2. **Virtual**

- Simulate robot behaviors in a virtual environment (WeBots)
- Executed & created challenges in WeBots
- Created scripts for navigation and mapping
- Establishing scripts for the Physical team to interact with, whilst recording accuracy

2.2 Objectives

2.21 Capstone Objectives

1. Replace the locally produced robots with two RasPi-based canine robots with LIDAR
2. Develop new challenges for the AI class – beyond what is in the DOGZILLA repo
3. Implement the new challenges both physically and virtually
4. Put on an awesome show that gets people excited! (Maze traversal or pinpointing users location and navigating to them)

2.22 Robot Objectives

The main objectives for the robot are:

1. **Develop an Autonomous Navigation System:** Implement a reliable navigation system that allows the robot to explore and maneuver in an environment with minimal human intervention. (ROS or python & cpp)

2. **Implement Obstacle Detection and Avoidance:** Integrate sensors to detect and avoid obstacles, ensuring safe movement within the workspace.
3. **Integrate Machine Learning for Object Recognition:** Train and implement a machine learning model to identify and respond to specific objects in real time.
4. **Create a Robust Control System:** Design a control system that seamlessly integrates with the sensor and machine learning subsystems, enabling precise and responsive movements.
5. **Test and Optimize Performance:** Conduct iterative testing to refine the robot's algorithms, achieving high levels of reliability and adaptability.

3. Challenges and Problems

-Ran into camera display not working (Had to reset the Capture widget and use VNC)
-A lot of times if you want to test advanced features we have to stop critical Start-up functions forcefully → Usually have to reset the robot.
-Retrieving full LiDAR data (rviz2 image saving, outputting data, saving it)
-Determining if we want to use JypterNotebooks and ROS2
-The robot must have a functioning demo by Spring Break of 2025 (aka in the middle of March)

-Virtual and Physical have been → ROS not being
-Combining the efforts of Physical (Jupyter Notebooks) with virtual efforts (ROS)
-Ensuring the DOGZILLAs are project ready & able to be used within a 4-week time span for the AI and ML learning class.
-Balancing the processing power of the RASPI, base functionality, and experiments & gathering data.
-The DOGZILLA's Rasp-pi has files completely unorganized and does not have any information about where to locate them and how to ensure file integrity.

- Climbing stairs due to climbing being 2 cm or below in height and cannot distinguish what to climb upon
- A maze where 2 robots communicate with each other for the path taken due to calculations of deviation and communication server.
- A map of the first floor of Bradley University due to time constraints and the big dimension

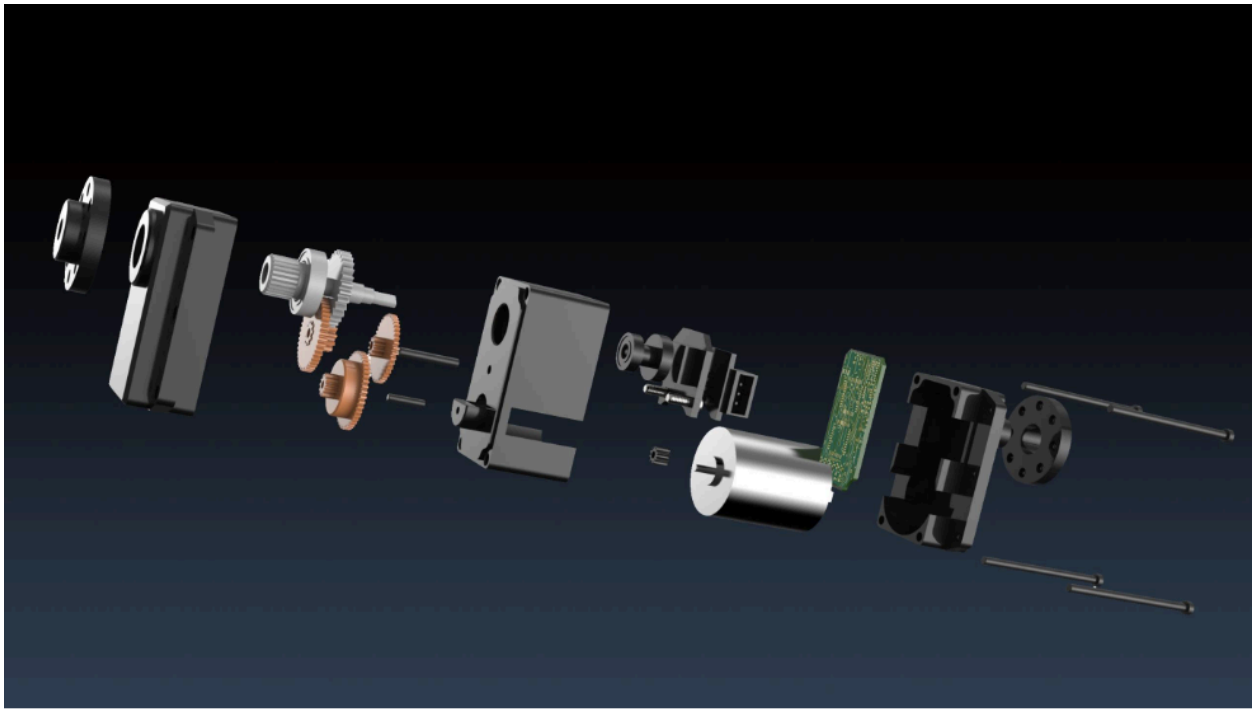
4. Robot Specifications

DOGZILLA is an advanced visual AI robot dog designed for versatility and movement featuring:

- 12 degrees of freedom with 6 high-precision servos and an aluminum alloy body
- Realistic movement capabilities, including walking and twisting like a real dog
- Omni-directional movement and six-dimensional attitude control
- Equipped with IMU and servo angle sensors for real-time posture and joint angle feedback
- Utilizes inverse kinematics algorithms for various motion gaits
- Powered by Raspberry PI 4 as the main controller, with Ubuntu 20.04 ROS2 for programming and RVIZ simulation
- Supports AI visual recognition, lidar mapping navigation, and voice control through Python programming
- Compatible with multiple remote control methods, including APP, handle, web pages, and computer keyboards
- Features advanced functions like label recognition, face detection, target tracking, and visual line patrol
- The S2 model adds lidar and intelligent voice modules for enhanced mapping navigation, obstacle avoidance, and voice commands

4.1 Hardware & Measurements

Physical Measurements → in Excel sheet in Google Drive



Servo parameters	
Model	Bus serial servo
Output torque	4.5KG•CM
Rotating speed（S/60°）	0.1 S/60°
Precision	0.01
Working voltage range	4.8V ~ 7.4V
Working temperature	-20℃ ~ +60℃
Angle range	0~360°
Weight	20±1g
Motor type	Hollow cup

4.2 Connecting to Robot

THIS SECTION IS EXTREMELY IMPORTANT, YOU WILL NEED TO KNOW VARIOUS WAYS TO CONNECT AND HOW THEY DIFFER.

Every time that you want to connect to the robot, it is **mandatory** to be connected to the robot's Wifi signal. This is true whether you connect with the phone app or wireless handle controller, using Virtual Network Computing (VNC), SSH, SCP, rdp, or Jupiter Notebook. The different ways to connect are noted in DOGZILLA's repository in sections 3 & 5, but we will go over the pros and cons of it.

The following are the Login Credentials **(IMPORTANT)**:

Login user and password-related content:

Username: **pi** Password: **yahboom**

Vnc remote login password: **yahboom**

Jupyterlab login password: **yahboom**

Factory hotspot signal: **DOGZILLA_WIFI** Password: **12345678**

4.21 Connection Method Differences

App Remote Control

Pros:

- Quick and easy
- Allows for all base functions of robot ligaments
- Allows access to camera feed
- Control individual joints
- Run all pre-programmed emotes & actions

Cons:

- Unable to access the robot terminal
- Unable to run any scripts
- Can't access data
- Unable to see errors or script progression

Handle/Controller Remote Control

Pros:

- A lot easier to handle moving the robot
- Allows to program buttons to pre or custom-made scripts
- Allows base access to robots' movement
- Run all pre-programmed emotes & actions

Cons:

- Unable to access the robot terminal
- Unable to run any scripts
- Can't access data
- Unable to see errors or script progression
- Cannot access the camera feed
- Cannot control individual joints

PC Control

Pros:

- Able to access the robot terminal
- Able to run any scripts
- Access most data
- Can access the camera feed
- Control individual joints
- A lot easier to handle moving the robot
- Allows to program buttons to pre or custom-made scripts
- Allows base access to robots' movement
- Run all pre-programmed emotes & actions

Cons:

- Takes up a lot of the robot's CPU
- Annoying to use pre or custom-made buttons
- Need to add more configurations in order to start
- File organization is extremely disorganized (especially with ROS)

5. Tools and Resources

LocalRobots Google and GitHub access codes and passwords

1. GitHub Login - **LocalRobots**
2. GitHub Password - **Webots123()**
3. Google Account Login - **localrobots0@gmail.com**
4. Google Account Password – **Webots123()!**

NOTE: Base DOGZILLA Factory File (Zipped – **RPi07_DOGZILLA**) is in the Google Drive

Dogzillas Repository (GitHub)

https://github.com/YahboomTechnology/DOGZILLA/tree/main

Dogzillas Repository (Primary with downloads)

http://www.yahboom.net/study/DOGZILLA

Backing up your Raspberry Pi

https://forums.raspberrypi.com/viewtopic.php?t=26463

Open CV and Understanding Image Processing In Python

https://docs.opencv.org/4.x/d4/da8/group__imgcodecs.html

6. Dogzilla Repository Review

During the duration of this project, we have been extensively researching and testing DOGZILLA's repository. Reading, testing, and implementing each module has led us to multiple conclusions that affected our plans to implement which methods we would use to make DOGZILLA achieve our main objectives. The sections that will be covered will go over the extremes of which will be important and/or disregarded.

Thus we have constructed a chart that has separated different sections and modulus into two categories: (Useful & Reliable for Implementation) and (Troublesome & Not Reliable for Implementation). (Reword what we have categorized these into). The following sections will not be placed in strict numerical order in every section but placed numerically in the specific category it is placed in.

6.1 Useful & Reliable for Implementation

Section 7: DOGZILLA Control Course

7.1: Basic Control (link)

Path: DOGZILLA/Samples/2_Control/1.basic_control.ipynb

This section aims to enable the robot to perform movements such as forward, backward, left, right, turn left, and turn right. For the forward and backward movements, the step is the step width of range [0,25]. Right and left translation has a step range of [0,18]. The turn left and turn right have a step range of [0,100]. A larger value means there is a higher step width and it is "faster". Clicking the button for each would perform the movement.

7.2: Gait Conditioning (link)

Path: DOGZILLA/Samples/2_control/2.speed_adjustement.ipynb

The purpose of this section is to control the speed of the robot. The move(direction, step) function allows the robot to move back and forth along specified directions, which is x to move forward or backward, y to move left or right, and z for rotational speed for left and right turns. The step parameter accepts a range of values where x has a range of [-25,25] and y has a range of [-18,18]. Positive values indicate forward or left, and negative values indicate backward or right. The turn(step) function rotates the robot with a step range of [-100,100] measured in degrees per second, where negative values turn left and positive values turn right. The pace(mode) function adjusts the movement speed based on step frequency, with speed calculated as step frequency multiplied by step width. Acceptable mode inputs are “normal”, “slow”, and “high”.

7.3: Body Control (link)

Path: DOGZILLA/Samples/2_Control/3.body_control.ipynb

This section aims to provide translation and attitude adjustment functions to allow precise control over the robot's movements and orientation. By adjusting translation parameters, the robot can move accurately in various directions or change height as needed. Attitude adjustments for roll, pitch, and yaw help the robot maintain balance, correct its posture, and align with specific orientations. The translation(direction, data) function adjusts the robot’s body position. Direction can be a single character or list, where x indicates forward/backward, y controls side-to-side movements, and z adjusts the height. Translation ranges are X: [-35,35], Y: [-18,18], and Z [75,115] in millimeters. The attitude(direction, data) function modifies the robot’s orientation, with directions for roll (r), pitch (p), and yaw (y) angles. Adjustment ranges are R: [-20,20], P[-15,15], and Y: [-11,11] in degrees.

7.4: Performing Actions (link)

Path: DOGZILLA/Samples/2_Control/4.performance.ipynb

The purpose of this section is to perform different actions. The action(action_id) function triggers specific preset actions, with an action_id range of [1,255], where 255 stops the operation and resets the robot’s posture. The perform(mode) function cycles through actions, with mode 1 enabling continuous action rotation and mode 0 stopping 1. Available actions include a variety of movements and poses like Li_Down, Wave_Hand, Stand_Up, Stretch, Crawl, and more (see list below), allowing interaction and movement for the robot.

List of actions with action ID (note: Github repo has this in Chinese, but this is in English **(link)**

ID	Actions	ID	Actions	ID	Actions
1	Lie Down	2	Stand Up	3	Crawl
4	Turn around	5	Mark Time	6	Squat
7	Turn Roll	8	Turn Pitch	9	Turn Yaw
10	3 Axis	11	Pee	12	Sit Down
13	Wave(Hand)	14	Stretch	15	Wave(Body)
16	Swing	17	Pray	18	Seek
19	Handshake				

7.5: Stabilization Mode (link)

This section describes the imu(mode) function, which ensures that the robot dog maintains a level posture by automatically adjusting its angle to keep its back horizontal. When self-stabilization is active (mode 1), manual posture adjustments are disabled; mode 0 turns it off. The stabilizing_mode button can toggle this feature on or off, with prompts indicating the status. While useful for tasks requiring steady movement or balance, self-stabilization allows the robot to stay aligned without manual adjustments. However, it may not have been used extensively in the project since it operates automatically without requiring much user input.

7.6: Servo Control (link)

Path: DOGZILLA/Samples/2_Control/6.control_motor.ipynb

The purpose of the motor control functions (check the GitHub repository for each function) is to provide precise and flexible control over the robot dog's steering gears, enabling it to perform various movements and actions. By adjusting the rotation angles of the steering gears on each leg, the robot can achieve coordinated movements. The ability to load or unload individual motors allows for fine control over speed and torque, while the reset() function ensures the robot can quickly return to its initial state. This setup is important for tasks requiring leg movements, stability, and adaptability in navigating different environments or performing complex actions.

7.7: Single Leg Control (link)

Path: DOGZILLA/Samples/2_Control/7.control_leg.ipynb

The purpose of the leg(leg_id, data) function is to provide precise control over the robot dog's leg positioning. The leg_id represents the four legs of the robot (1-4 left front, right front, right rear, and left rear respectively). The data input is a list of three numerical values representing the foot position values representing the foot position in millimeters where x is [-35,35], y is [-18,18], and z is [75,115]. Using inverse kinematics, the robot calculates the necessary movements for the steering gears to position the foot. The reset button restores the robot to its original state.

Section 8: Advanced Course

8.1: Color Recognition (link)

Path: ~DOGZILLA/Samples/3_AI_Visual/1.color_recognition.ipynb

This section obtains an image and analyzes it through the OpenCV library. This allows the robot dog to separate and process the part of the image that equals the HSV value of a chosen color from the background. Once it is run, the left image shows the original image and the right shows the image after separation. The upper part shows the chosen color as white and the rest as black. The lower part shows the chosen color as it originally appeared and the rest as black. Press close_camera to turn off the camera. Note: Choose a color that highly differs from the background color to avoid confusion with similar colors.

8.2: Color Tracking (link)

Path: ~DOGZILLA/Samples/3_AI_Visual/2.color_tracking.ipynb

This section works off the same principle as 8.1. Additionally, the robot dog adjusts its body posture to follow the color. The left side shows the camera screen. Select a color to be tracked with the button on the right, place that color in front of the camera, and then move the color around within the robot dog's range of movement to watch it track the color with the camera.

Press close_camera to turn off the camera. Note: Place the robot dog within a pure color environment to avoid color confusion.

8.3: Color Recognition Action Group (link)

Path: ~DOGZILLA/Samples/3_AI_Visual/3.color_action.ipynb

This section enables the robot to recognize specific colors and perform predefined actions when the detected color meets size and location criteria. Users can adjust global HSV values (color_Lower and color_Upper) via color buttons to identify a target color, while a daemon thread processes the camera feed, detects the color, and triggers actions defined in `G_action` when conditions are met. To execute, open the JupyterLab client, navigate to the specified file path, and run all cells; controls will appear at the bottom. The camera feed is displayed on the left panel, where the robot detects colors and performs actions when the object matches the chosen color. Ensure minimal background interference, and click "Close Camera" to stop the feed.

8.4: Face Detection (link)

Path: ~DOGZILLA/Samples/3_AI_Visual/5.face_tracking.ipynb

This feature enables the robot to detect and frame human faces using Haar feature classifiers, dynamically adjusting its posture to keep the face centered in the camera view. Grayscale camera images are processed to detect faces, and a daemon thread displays the feed while tracking face positions to guide robot movements. To execute, open the JupyterLab client, navigate to the file, and click **Run All Cells** to initialize controls. The robot detects and follows faces dynamically as they move within the camera's view. Use a solid background to reduce false detection, and click "Close Camera" to stop the feed.

8.5: Face Tracking (link)

Path: ~DOGZILLA/Samples/3_AI_Visual/5.face_tracking.ipynb

This advanced feature enhances face detection by continuously tracking human faces, and adjusting the robot's movements to keep them in view. Using optimized detection processes and a daemon thread, the robot ensures seamless tracking and displays the camera feed in real-time. To execute, open the JupyterLab client, navigate to the file, and click **Run All Cells**to activate

controls. The robot dynamically follows detected faces within its range. Minimize background noise to improve accuracy, and click "Close Camera" to stop the feed.

8.6: Watchdog Mode (link)

Path: ~DOGZILLA/Samples/3_AI_Visual/6.face_handshake.ipynb

In Watchdog Mode, the robot greets users with a handshake when it detects a face meeting specific range and size criteria. Face data such as position and size is processed via a daemon thread to trigger the handshake gesture. To use this feature, open the JupyterLab client, navigate to the file, and click **Run All Cells** to start the mode. When a face is detected, the robot performs a handshake. Ensure a clear background to avoid false triggers, and click "Close Camera" to stop the feed.

8.7: QR Code Identification (link)

Path: ~DOGZILLA/Samples/3_AI_Visual/7.QRCode.ipnyb

This section reads a QR code through the robot dog's camera. It is called the Pyzbar library, and it analyzes the image, position, and band information of the 2D code caught by the Raspberry Pi camera. The robot dog will recognize the 2D code and show the recognized data on the 2D code through its camera. When finished, click the close_camera button to turn the camera off.

8.8: QR Code Recognition Action Group (link)

Path: ~DOGZILLA/Samples/3_AI_Visual/8.QRCode_action.ipynb

This section takes the QR code reading and acts if applicable. Place the QR code in front of the camera so it can be read and decoded by the Pyzbar library. The code runs through the code_action(barcodeData) function to check for a corresponding action. If one is found, the robot dog will act.

8.11 + 12: Visual Tracking (link1, link2)

Path: cd ~/DOGZILLA/Samples/3_AI_Visual/11_12.followline/

Run: python3 follow_line.py

This program uses the robot dog's camera to track and follow a path. Open up RealVNC_Viewer and run this program through the system terminal. To set up the robot dog, press the R key to initialize its posture, set the step frequency to slow, the body height to 90, and make sure the

camera looks down and forward. Over the footage of the marker designating the path given by the camera, click and drag the mouse to make a small rectangle. The rectangle must be made over the marker and the marker alone; any other color besides the one for the marker will cause issues. The system will record the color value to keep track of what the robot will be following. There will be a second window that displays the path in black and white, with the saved color appearing as white. Press the spacebar to begin, and the robot dog will consistently update its position so it stays centered on the drawn rectangle as it moves forward. If at any point the robot dog stops on its own, press the R key to reset and then the l key to bring the path image back up. The robot dog will automatically switch to obstacle-crossing mode if the color it is following registers as horizontal and close to the bottom of the camera.
Note that the Q key exits the program, and all key inputs only work when the camera display is active. Horizontal color is determined by the y value being greater than 300 and the radius being greater than 150.

Section 14: Lidar Course (For S2)

14.1: Introduction and use of Lidar

LiDAR is a detection system that works on the principle of radar but uses light from a laser. The main benefit is its ability to gather information from all angles in regard to the robot and map out the room on a 2D scale. How this is accomplished is through individually starting up the sensor and then running a script to start it up and also representing it visually in Rvizz. There are a multitude of LiDAR scripts available with the default settings to accomplish different goals.

14.2: Obstacle Avoidance

This section goes over the xgo R200 LiDAR script which is used for obstacle avoidance. When run the script determines if a wall or a narrow passage is coming up by the different returned angles to turn away in the opposite direction. This is great for ensuring that the robot doesn't hit a wall but can also affect our ability to create nodes at intersections at an acceptable deviation from its center.

14.5: Patrol

The main benefit of this section is that we will be able to have pre-routed maps which we are then able to navigate afterward. Thus after having proper maps up it will then (to the best of its ability) set a patrol route from that map to the end objective. The main concern is whether the robot's starting position and the layout of the mapped maze are set up properly (aka 1:1 ratio with no setup defects).

6.2 Troublesome & Not Reliable for Implementation / Not Necessary

Section 6: OpenCV Basic Course

OpenCV image capabilities include geometric transformations, grayscale conversion, binarization, and noise reduction methods. However, we did not rely heavily on techniques related to color intensity manipulation, such as color histograms or brightness adjustment, as our primary objective was not to navigate a maze based on color intensity but to focus more on the basic functionality of the robot movements and will do maze in the future.

6.1: OpenCV Introduction (link)

OpenCV is an open-source API library for computer vision used in scientific research and commercial applications. Its source code is publicly available, allowing users to modify and compile specific API functions as needed. OpenCV plays a role in image processing on devices like the Raspberry Pi, supporting tasks like image reading, display, writing, quality adjustments, and pixel operations. It has applications in intrusion detection, target tracking, face detection, and recognition.

6.2: Geometric Transformation (link)

This section covers several geometric transformations in OpenCV for image processing. It begins with image scaling using cv2.resize to modify the size with options for interpolation, then illustrates clipping to extract specific image sections and panning to shift the image by a defined

number of pixels. Furthermore, it explains horizontal and vertical mirroring for image rotation via cv2.getRotationMatrix2D. It also introduced the perspective transformations using cv2.warpPerspective, enabling the conversion of quadrilateral projections. This included Python code examples and visualization techniques, often utilizing Matplotlib for display.

6.3: Picture processing and drawing text line segments (link)

This section covers image processing techniques, primarily focusing on grayscale processing, binarization, edge detection, and line segment drawing using OpenCV. Grayscale processing involves converting color images, defined by RGB values, into grayscale by equalizing RGB channels or applying methods like maximum, average, and weighted averages, catering to human visual sensitivity. Binarization converts grayscale images into black and white based on threshold values to enhance contrast. Edge detection, especially the Canny algorithm, refines image details by reducing noise and isolating prominent features, applying steps like Gaussian filtering and non-maximum suppression. OpenCV facilitates drawing geometric shapes such as lines and rectangles on images, allowing customization of color, thickness, and line types for visual demarcation in processed images.

6.4: Image beautification (link)

Image processing techniques are widely used to enhance and analyze images by manipulating color, brightness, and spatial properties. A color histogram measures the distribution of colors, independent of image objects, and is often visualized with libraries like Matplotlib or OpenCV to quantify pixel intensities. Histogram equalization, a grayscale transformation, improves contrast by stretching pixel values across the full dynamic range. For repairing damaged images, patching algorithms such as OpenCV's inpainting use nearby pixel information to fill areas marked by a mask. Brightness enhancement synchronizes RGB channels to intensify the image's brightness, while skin whitening and bilateral filtering focus on edge-preserving smoothing for aesthetic enhancement. Filtering methods, like Gaussian and median filtering, remove noise and blur using frequency domain analysis, with Gaussian filtering providing a weighted average for natural smoothing and median filtering ideal for reducing salt-and-pepper noise. These techniques, through nuanced manipulation, enable efficient image restoration, enhancement, and feature extraction.

Section 7: DOGZILLA Control Course

7.8: Reading Data (link)

Path: DOGZILLA/Samples/2_Control/8.read_data.ipynb

The purpose of the functions in this action is to monitor and retrieve real-time data from the robot, allowing for feedback on its current state and performance. The read_Motor() function retrieves the angles of 12 servos, returning a list of angles for specific servos if successful or an empty list if it fails, which helps in tracking the position of the robot's limbs or joints. The read_Battery() shops the remaining battery power as a percentage (1-100). The read_roll(), read_pitch(), and read_yaw() returns the floating point values about the robot's orientation. These functions return 0 if the reading fails. While it is important to read the data of the robot, this is not necessary for the project since the data can be visually observed directly.

7.9: PC Control (link)

Path: DOGZILLA/Samples/2_Control/9.dog_Ctrl.ipynb

This section is an introduction to using a camera with the actions outlined in 7.1, 7.2, and 7.4 and is further expanded in Section 8. The Dogzilla documentation only mentioned closing the applications or other programs that occupy the camera before running the program. However, the issue we encountered was when running the scripts, the camera could retrieve the image identifier, but not the display itself. The reason for this is that we were accessing the Raspberry Pi via SSH and Jupyter Notebook locally. We checked the camera connection by looking at the files in the /DOGZILLA/app_dogzilla/directory and running ls /dev/video* , but this only identified the connected video ports (0-16). To fix the issue, we installed VNC Viewer, connected to the Raspberry Pi, and accessed the camera through Firefox by entering the Raspberry Pi's IP address with port 8888. More detailed documentation for this camera issue can be found **here.**

7.10: Wireless Handle Control (link)

Path: ~/DOGZILLA/app_dogzilla/joystick_dogzilla.py

To run: python3 ~/DOGZILLA/app_dogzilla/joystick_dogzilla.py

The purpose of this is to enable control of the robot dog using a wireless USB handle, which interfaces with the Raspberry Pi. The wireless handle allows for various movements and actions

such as translation, adjustments in speed, rotation and other tasks. This is not used much in the project as we focused more on controlling the robot by connecting to our own computer via SSH.

Section 8: Advanced Course

8.6: Watchdog (link)

Path: ~DOGZILLA/Samples/3_AI_Visual/6.face_handshake.ipynb

This section has the robot dog perform a handshake motion when a face is registered. The camera image is shown on the left side of the screen and waits for a face to be placed in front of it. The camera draws a box around the face and checks for if the length and width of the box is greater than 60. If this is true, and the center point is inside the detection range within the middle of the picture, the robot dog will perform the handshake action. Once finished with this section, make sure to click the close_camera button to turn the camera off.

8.9: Climb (link)

Path: ~DOGZILLA/Samples/3_AI_Visual/9.obstacle_crossing.ipynb

This section has the robot dog enter a climbing and obstacle climbing mode. The robot dog's gait will have it raise its feet and climb a ladder or series of steps with a height of less than 2 cm. The gait height is controlled by the gait_type(mode) function and the input ranges of 'trot', 'walk', and 'high_walk'. Note that the length and width of the steps must be longer than the robot dog's chassis to avoid treading.

8.10: Kick Sports (link)

Path: ~DOGZILLA/Samples/3_AI_Visual/10.play_ball.ipynb

This section controls the robot dog's steering gear to perform a kicking action. The robot dog is controlled by the steering gear using both the posture and angle taken by it. Pressing the Play_Ball button activates the action, Stop makes it stop, and Close_Camera turns the camera off.

Section 10: ROS Basic Course

The entirety of this section goes over setting up packages and nodes that can be used alongside the Robot Operating System (ROS) in order to achieve feats of monitoring data, creating unique scripts, and creating packages by a system of nodes. The types of nodes are publisher (response) and subscriber (request) nodes which communicate with each other, the main concern with this is that they do not have to be stored in a uniform location. Meaning that it can be made anywhere within the distribution and won't give its file pathing to it.

What this results in its default packages we cannot modify the current nodes without spending a lot of time to find their locations. Alongside this ssh-ing or VNCing into the machine makes it nearly impossible for us to run more complex ROS scripts such as navigation and wireless transmission between multiple robots. Also, any small changes to the startup files with an incomplete custom-made package have a high probability of corrupting default scripts and data transmission nodes. It was a lot of work for little results, especially in the timeframe we were given to make this a functional course within a semester.

Section 11 & 12: ROS and OpenCV Visual Course & ROS Basic Control Course

As this is an expansion on the ROS Basic course, we for the same reasons why we didn’t want to continue with section 10 are the same. The applications for this section don't combine well with our goal to navigate a maze, especially as we would have to add a multitude of testing materials so the robot would be able to identify labels, colors, and also coordinates.

Section 14: Lidar Course (For S2)

14.3: Tracking

The biggest concern relating to this script is that it will only track the nearest object. Which in turn means that the closest object is a wall. It is not useful for our implementation as we want to track by QR Codes.

14.4: Guarding

How the lidar sensor would allow the robot to guard is that it will track the object in front of it and keep itself horizontal to where it moves. The main problem is that it will not move forward but “Block” by adjusting itself horizontally which is not the result we need

14.6-14.10: Robot Mapping and Navigation

The main problem that occurred with these sections is that at the end of our FA24 capstone, we threw out the idea of using ROS in terms of maze navigation. This is an incredible script to map and navigate if you are going in terms of ROS2 but we choose to use jupyter scripts instead. Also to visually determine if the mapping is working causes a lot of lag and burns a lot of the CPU’s resources which means that it's better to run and then test if the mapping worked.

7. Finding and Results

The DOGZILLA GitHub repository is important for understanding this capstone project. However, some parts were difficult to interpret due to documentation, code, and comments written in Chinese, along with disorganized sections. We organized and clarified the documentation to make it more understandable for future use.

Section 7: DOGZILLA Control Course

The findings demonstrate the robot's ability to perform controlled movements such as forward, backward, left, right, and turns (right and left) with speeds (normal, low, and high) based on adjustable step parameters. Testing showed that Charlotte outperformed Charlie slightly in speed across normal, low, and high modes, despite both robots showing slight directional drifts. Precise translation and attitude adjustments enabled the robot to navigate with stability, using functions for position control (translation and rotation) and maintaining posture through roll, pitch, and yaw adjustments. The IMU function proved effective for auto-stabilization, though it was minimally applied. Additionally, specific motor control functions allowed for fine-tuned movements, while leg positioning leveraged inverse kinematics for coordinated motion. Challenges with camera usage were resolved by installing VNC Viewer for remote access, improving image capture. Overall, these functions combined to allow efficient and flexible control of the robot, with real-time feedback enabling adjustments and ensuring stability.

DOGZILLA's Movement Specs:

In tests comparing the step movements of two dog robots at different modes and speeds, Charlotte was slightly faster than Charlie. Both robots showed a slight drift to the left in normal mode, drifted left halfway through in slow mode, and had a slight right drift in high mode. The test distance was 100 centimeters with a step width of 50 centimeters for both. Charlotte's average time for normal mode was 8.73 seconds, compared to Charlie's 8.96 seconds. In low mode, Charlotte averaged 28.75 seconds, while Charlie averaged 32.80 seconds. In high mode, Charlotte's average was 7.76 seconds, and Charlie's was 8.2 seconds. Frequency was calculated as distance divided by average time, then multiplied by step width, with lower speeds resulting in lower frequency.

(Calculations link)

Calculations for Charlotte: **Calculations for Charlie:**

Charlotte

Title	Normal (s)	Low (s)	High (s)
Duration:	8.95	28.72	7.75
	8.39	28.69	7.74
	9.01	29.15	7.97
	8.49	28.43	7.68
	8.83	28.75	7.64
Avg Time:	8.734	28.748	7.756
Frequency (cm)	0.2289901534	0.06957005705	0.2578648788

Charlie

Title	Normal (s)	Low (s)	High (s)
Duration:	9.25	32.27	8.18
	9.05	32.58	8.26
	8.8	32.57	8.08
	8.81	32.8	8.35
	8.88	33.8	8.15
Avg Time:	8.958	32.804	8.204
Frequency (cm):	0.2232641215	0.06096817461	0.2437835202

LiDAR Sanitization and Graphing

The location where these scripts are held is within the Google Drive of the Local Robots Capstone: CS 490 - FALL 2024 > PHYSICAL > Code & Script FA24 > Lidar Sensor Related Code.

From working with LiDAR we have determined that making this robot accessible for a month's worth of work for the Artificial Intelligence (AI) and Machine Learning (ML) course it would cause a lot of problems with setting up the environment, creating the scripts, and having the students

Maze Creation & Materials

We came up with a way for us to create a maze that first we could dynamically create, navigate, and build with the least amount of resources possible. What our team came up with was that we would be using 3D-printed polyresin molds to create pillars that can be used as corners or walls. Alongside these molds, we will be using cardboard to behave as the walls because it is cost-effective and it's able to be molded to our desire. Specifically, we can create a corner of a passage with only 2 molds and a single pizza box.

8. Current Work

During the last few weeks working on the capstone project consists of us gathering all the LiDAR data we can get for scans and determining how much information we can get from each echo's data & what can we determine from it. We primarily achieved this by using the ROS Topic Echo publisher node to give us this information and we sanitized it with the *processROS* python script. This Python script came from the data (YAML format) gathered & maintained by Logan with the combined efforts with Professor Brennan which carried the creation of the sanitization of the data (YAML format) into JSON format.

Additionally, our teams were creating ways that we could create a maze that first we could dynamically create, navigate, and build with the least amount of resources possible. This is achieved by using the 3D-printed molds and cardboard. With our reliance on the camera display, we can effectively make a maze with only corners without the need to fully create a maze with walls in between each section.

Lastly, we were finalizing what scripts and software we wanted the robot to use to achieve our goals. We concluded that using Jupyter Notebook to create & maintain the scripts so, as the factory code is highly compatible to be mixed and meshed together. The robot will be relying on the camera display to traverse the main and will be assisted by the LiDARs mapping capabilities. Thus we gave ROS the boot and will not be using it as the primary means of maze navigation and mapping.

9. Past Disregarded Ideas, plans, work, etc (Changes to our plans)

The following consists of the ideas that we have changed due to time constraints, implementation difficulties, and robots physical limitations:

- Using ROS for automating maze mapping and navigation
- Having the DOGZILLA robots communicate throughout the maze to piece segments of the map. (The lock and key dilemma)
- Traversing & Mapping Maze primarily through LiDAR Sensor
- Having Robots communicate with each other (by ROS)
- Have the maze be on multiple levels (slightly different elevations)

10. Next Steps (for Spring 2025)

need to demo on March

Physical:

The next steps for Spring 2025 involve reviewing the documentation to refresh our understanding and planning how to navigate the maze using Python scripts in Jupyter Notebooks or ROS2 with Webots. We'll need to create more maze joints and start implementing scripts to enable autonomous navigation through the maze. We will also test the code provided by the virtual team to ensure it works with the physical robot.

- Go over sections 7 and 8 to familiarize
- Link some codes together to track objects (QR code)
    - Basic & Advanced Movement
    - Camera Display & Recognition

    - QR Code Identification
    - QR Code Recognition Action Group
    - Performing Actions
    - Reading Data & Storing Data (Dictionary of Nodes & distances between them)
    - Object tracking & recording distances
- Have the robot recognize, move, and read content in QR code
    - Note it to the dictionary and store it
    - Scan around for other nodes available (those nodes are children)
    - Make a mental map of the corners
    - Have the QR codes map to the actual maze we will be creating
    - QR codes in physical need to be the same in Virtual