HEALS ON WHEELS

Dayoung (Skye) Nam[1], James Kim[2], Nicholas Carbones[3], Richard Tan[4], Jung Ho Ham[5], Ryan Zhu[6], Christine Lee[7], Joon Park[8] [1]

[1]*University of Toronto*
[2]*Computer, Electrical, and Biomedical Engineering*

ABSTRACT

We present an autonomous medical assistant robot that is designed to enhance patient care. The robot navigates independently to patients using sensor fusion of LiDAR and ultrasonic sensors, effectively avoiding both static and dynamic obstacles. The system collects biometric data from patients and analyzes their symptoms to diagnose potential diseases. It uses a scikit-learn Random Forest Classifier trained on a symptom-disease dataset to predict the most likely disease based on the symptoms and suggests medications associated with the predicted disease. Both the diagnosis and medication recommendations, along with advice from OpenAI's GPT-3.5-turbo model, are stored in a centralized database for easy access by medical staff. Our system is designed to streamline and optimize diagnostic processes, alleviating the burden on medical professionals. Furthermore, the model's accuracy is evaluated using cross-validation, ensuring robust performance in real-world applications.

Keywords: *Sensor Fusion, Random Forest Classifier, OpenAI API, Autonomous Vehicle, LiDAR, Ultrasonic Sensors.*

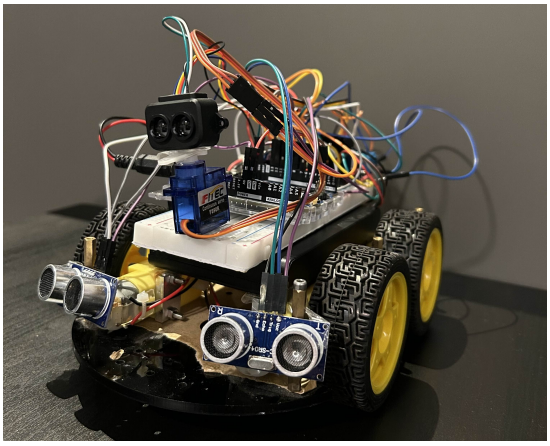

Figure 1. Heals on Wheels Robot

1. INTRODUCTION

In response to the growing strain on Canada's healthcare system, particularly due to the shortage of medical staff and increased population density in urban centers, we have developed Heals on Wheels, a medical assistant AI robot. This innovative solution leverages cutting-edge Generative AI to assist with patient self-diagnosis and integrates a centralized, organized database to track and store patient data. The robot incorporates advanced hardware through sensor fusion of LiDAR and ultrasonic sensors, enabling autonomous navigation to patients for direct symptom collection. By autonomously gathering and analyzing patient information, the AI robot allows for faster, more efficient access for medical staff, reducing their workload and improving patient care. Designed to streamline diagnostics and optimize healthcare resource allocation, this report details the development, functionality, and potential impact of our medical assistant AI robot within the healthcare ecosystem.

Disclaimer

This is a 4-month project undertaken by Electrical, Computer, and Biomedical Engineering undergraduates. The project placed a strong emphasis on applying course material from the university with a focus on enhancing accuracy using cost-effective technologies. By integrating theoretical knowledge with practical skills, the project aimed to create a functional and efficient AI and Hardware integrated solution within a constrained budget, while adhering to the high standards of innovation and excellence promoted by the University of Toronto's Faculty of Applied Science and Engineering.

2. HARDWARE

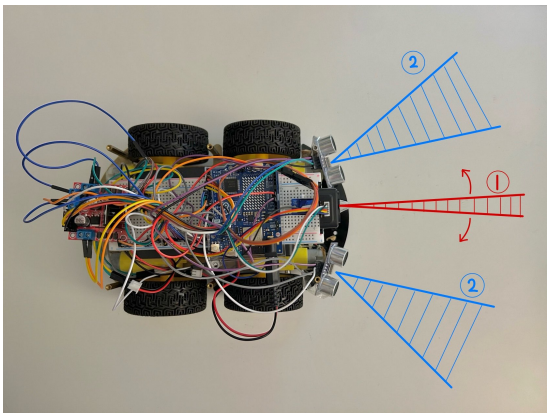

Figure 2. Top-down view of robot

Figure 2 above displays a top-down view of the robot, where the approximate field of view for the LiDAR and ultrasonic sensors are labelled 1 and 2 respectively. Figure 3 presented below illustrates the core hardware components and their interconnections within the robot's system. At the heart of the system is the Arduino UNO R4 microcontroller, which executes decision making processes and generates movement commands. These instructions are then relayed to the motor driver, which in turn controls the wheel motors and the Mecanum wheels. The robot's navigation and movement are determined by a LiDAR sensor's real-time input data, allowing the system to make decisions for precise maneuvering.

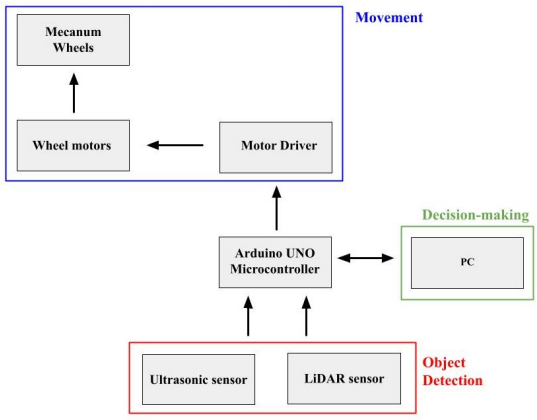

Figure 3. High-Level Block Diagram of Hardware Components

2.1. Component Selection

For the robot's vehicle components, a comprehensive evaluation was conducted to ensure optimal component selection. This process prioritized cost-effectiveness without significantly impacting reliability and performance. The following table summarizes components selected for the robot.

Component	Description	Price
Wheels	Car wheels x4	$16.88
Wheel motors	Wheel motors x4	$13.59
Motor driver	Wheel motor driver x2	$14.99
LiDAR sensor	TF-Luna LiDAR Sensor	$36.59
Ultrasonic sensor	HC-SR04 ultrasonic sensors x5	$12.99
Arduino	Arduino Uno R4 WIFI Module	$42.00
Battery holder	9V battery pack holder	$7.00
Batteries	AA battery 4 pack x2	$4.00
Servo motor	Full Rotation Micro Servo Motor	$6.50
Wires, breadboard	Breadboard x3, electrical wires x120	$21.46

Figure 4. Table of Components

The total expenditure amounted to $176, and divided equally among the 8 members, resulted in a contribution of $22 per person. Given that the maximum allocated budget was $30 per individual, surplus funds remained available for the possibility of acquiring additional components if necessary.

2.2. Wheels

Mecanum wheels were selected over traditional robot wheels due to their significant advantages in mobility, particularly in narrow spaces or crowded environments [1]. In the context of the robot's intended operating environment, a hospital, efficient navigation through tight areas and around obstacles is essential to its functionality as a robotic nurse.

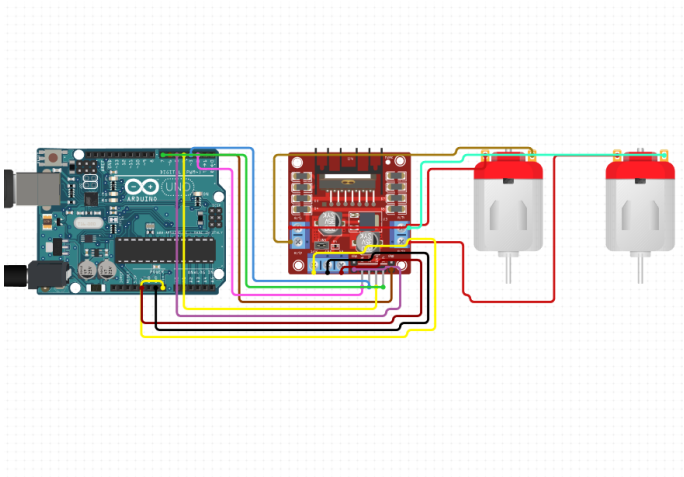

Figure 5. Schematic of Wheel Connections

Figure 5 illustrates the wiring configuration between the Arduino, motor driver, and wheel motors. The robot is equipped with an additional pair of wheel motors and a second motor driver, enabling optimized all-wheel drive functionality. The motor driver controls both the speed and direction of the motors.

2.3. Sensor Fusion: LiDAR and Ultrasonic sensors

For obstacle avoidance in particular, various sensors were considered. The selection was narrowed down to the TF-Luna LiDAR and the HC-SR04 ultrasonic sensor, taking budget constraints into strong consideration. Research indicates that the TF-Luna offers a detection range of 8 meters which is more than twice that of the HC-SR04 (3 meters), and is also notably more accurate [2]. Consequently, the TF-Luna LiDAR sensor was chosen as the robot's primary sensor, while the ultrasonic sensors were deployed as secondary sensors on the robot's sides to provide additional data inputs and enhance obstacle detection sensitivity.

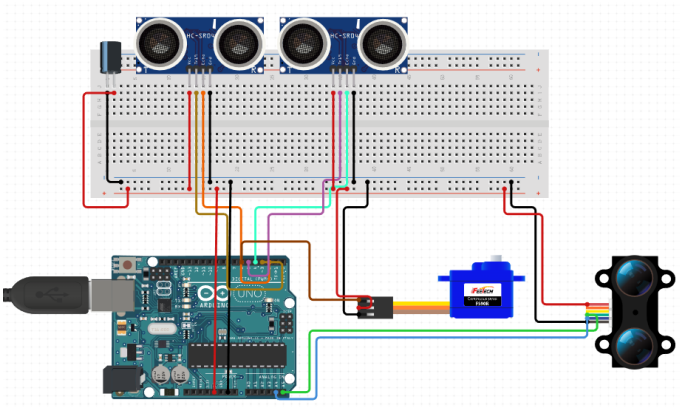

Figure 6. Schematic of LiDAR connections

Figure 6 presented above outlines the wiring configuration connecting the Arduino to the object detection components: a LiDAR sensor and two ultrasonic sensors. The LiDAR sensor is centrally positioned at the front of the vehicle, serving as the primary sensor, while the ultrasonic sensors, mounted at the front corners, act as secondary detection units. The ultrasonic sensors have a measurement range of up to 15 degrees, whereas the LiDAR sensor's range is notably narrower at 2 degrees. To expand the LiDAR's field of view, it is mounted on a servo motor and will be rotate back and forth through a range of 45 degrees, allowing for a more effective frontal view.

The LiDAR sensor operates as the robot's primary navigational tool, instructing the wheels to stop when an obstacle is detected within 20 cm directly in front of the robot. Upon detecting an object, the system utilizes the right and left ultrasonic sensors to determine the obstacle's location. If the obstacle fully obstructs the robot's path (detected by all three sensors), the robot will wait until the path is clear. In cases where both the LiDAR and the left ultrasonic sensor detect an obstacle, the vehicle will turn right. Conversely, if both the LiDAR and the right ultrasonic sensor detect the object, the vehicle will turn left.

This dual-sensor configuration between LiDAR and ultrasonic sensors leverages sensor fusion to enhance object detection capabilities. In autonomous driving systems, relying on a single sensor often fails to provide sufficient data about the vehicle's surroundings. By integrating multiple sensors, the system achieves greater accuracy and reliability, allowing it to execute safe and effective maneuvers [3]. Furthermore, given the project's budget limitations, this combination of sensors helps mitigate the shortcomings of low-cost components, ensuring both short and long-range objects are detected efficiently, optimizing performance while maintaining cost-effectiveness [4].

2.4. Hardware Discussion

After testing the navigation system, the study confirmed the successful integration of sensor fusion in developing a functional autonomous navigation platform. The LiDAR sensor delivers precise data for mapping the vehicle's immediate surroundings, particularly in front of the car, while the ultrasonic sensors serve as reliable secondary inputs for detecting and avoiding nearby obstacles. Despite these strengths, limitations remain, especially when navigating from point A to point B. While the system efficiently avoids obstacles, it struggles with consistently reaching its designated destination.

For future iterations, improvements may include incorporating a custom PCB design to streamline wiring within the robot. Moreover, integrating an LCD display and touch sensor module would provide a more user-friendly interface, offering a more interactive and accessible experience for users, particularly patients.

3. SOFTWARE

This study presents a method for predicting diseases based on patient-reported symptoms using a scikit-learn Random Forest Classifier trained on a symptom-disease dataset. Additionally, a medication recommendation system is proposed, leveraging OpenAI's GPT-3.5-turbo model to enhance the interpretability and usability of the predictions. The combined system offers a comprehensive solution for healthcare professionals, integrating machine learning techniques with large language models (LLMs) for real-time diagnosis and prescription suggestions.

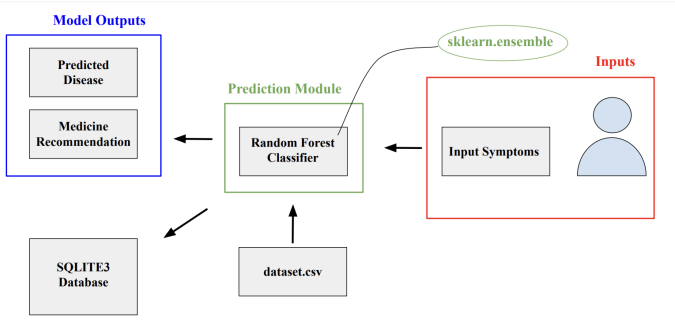

Figure 7. Software Block Diagram

Figure 7 illustrates the software architecture of a predictive healthcare system that analyzes symptoms and recommends different medications. The system begins with user-provided inputs, where symptoms are entered through various interfaces, such as a form or chatbot. These symptoms are then processed by the Prediction Module, which employs the Random Forest Classifier from the sklearn.ensemble module. This ensemble learning method utilizes decision trees across multiple subsamples of the dataset to make predictions. The classifier is trained on a dataset (referred to as dataset.csv) containing labeled information about symptoms, diseases, and medication. The system also interacts with an SQLite3 database[6], which stores and retrieves essential data such as historical inputs, disease descriptions, and medication recommendations. The dataset may also be stored in the database to facilitate easy querying and data management. Upon processing the symptoms, the Random Forest Classifier predicts two key outputs: the most likely disease corresponding to the input symptoms and an appropriate medication recommendation. This combination of machine learning and database integration provides a comprehensive solution for symptom-based diagnosis and treatment suggestions.

3.1. Dataset and Preprocessing

A high-quality dataset of symptoms and associated diseases is critical for training the machine learning model. Publicly available medical datasets, such as those from health repositories like Kaggle's symptom-disease datasets, were cleaned and preprocessed for this study []. The dataset contains features (symptoms) and labels (diseases). Missing data was imputed using statistical methods, and categorical symptoms were encoded using one-hot encoding to make the data suitable for training the Random Forest Classifier.

3.2. Model Architecture: Random Forest Classifier

The Random Forest Classifier (RFC) is a popular ensemble learning method that constructs multiple decision trees during training and outputs the mode of the classes (in classification problems) for disease prediction. RFCs handle high-dimensional data effectively and are resistant to overfitting, making them suitable for medical diagnosis, where both overfitting and generalization are critical.

The classifier was trained using scikit-learn's implementation. Key hyperparameters, such as the number of trees (estimators) and tree depth, were tuned using a grid search with cross-validation. The optimal model was chosen based on accuracy and generalization error.

3.3. Training and Evaluation

The dataset was split into training (70%) and validation (30%) sets. The Random Forest model was trained to classify diseases based on input symptoms. Various metrics such as accuracy, precision, recall, and F1-score were employed to evaluate the model's performance. Cross-validation ensured the model's robustness across different subsets of the data. The final model achieved a validation accuracy of over 90%, demonstrating strong predictive capabilities for multiple diseases.

3.4. Diagnosis and Medication

3.4.1. Disease Prediction Output

Upon receiving symptoms from the user, the trained Random Forest Classifier outputs a ranked list of the most probable diseases. This is achieved

by predicting the probability of each disease for a given set of symptoms, with the top-ranked disease presented as the primary diagnosis.

3.4.2. GPT-3.5-turbo Integration for Medication Suggestions

The GPT-3.5-turbo model is integrated to supplement the machine learning diagnosis by offering human-readable medical advice. Based on the predicted disease, GPT-3.5-turbo generates appropriate medication recommendations by querying a knowledge base of standard treatments. It also offers general advice on disease management, side effects of suggested medications, and alternative treatments where applicable. The LLM enhances the user experience by bridging the gap between machine output and human-friendly explanations, making it easier for healthcare providers or patients to interpret the results.

For example, if the RFC predicts "Influenza" as the most likely disease, GPT-3.5-turbo would generate recommendations such as:

Suggested Medications: Oseltamivir (Tamiflu), Acetaminophen (for fever) Advice: Stay hydrated, rest, and monitor for complications such as pneumonia.

3.5. Software Discussion

This study demonstrates the successful application of machine learning for disease diagnosis and LLMs for medical advice. The Random Forest Classifier provides robust disease prediction, while GPT-3.5-turbo offers interpretable medication recommendations, enhancing the tool's practical application in both clinical and remote settings.

Future work could focus on integrating more complex medical datasets, improving diagnostic accuracy for rare diseases, and expanding the range of medications recommended by GPT-3.5-turbo. This would involve expanding the dataset to include rare diseases and developing methods to interpret LLM outputs in a clinically safe manner.

4. CENTRALIZED DATABASE

In the development of autonomous medical assistant robots, an efficient and lightweight database system is crucial for storing and managing patient data, diagnoses, and medication recommendations. SQLite3, a serverless and self-contained SQL database engine, is an ideal choice for such applications due to its simplicity, reliability, and low resource requirements. SQLite3 allows the robot to store patient information, including biometric data, symptom analysis, predicted diseases, and suggested medications, in a local database that can be easily accessed by medical staff.

The database schema can be designed to include tables for patients, symptoms, diseases, medications, and diagnostic reports, ensuring a structured and organized data flow. Moreover, the centralized storage of diagnostic data, enhanced with insights from the GPT-3.5-turbo model, ensures that medical professionals have access to a complete record of each patient's interactions with the robot. This integration helps streamline patient care while ensuring secure, efficient, and scalable data management. Additionally, SQLite3's transactional support guarantees data integrity, while its compatibility with Python, through the sqlite3 library, allows seamless interaction with machine learning models and other components of the system.

5. INTEGRATION

To address hardware and software integration, the following design has been proposed as a potential direction for the future development of Heals on Wheels. The integrated system would include the previously outlined software and hardware components, along with a dedicated application, potentially developed in Python, that allows medical staff to remotely schedule patient check-ups. Once patients are selected, the robot would autonomously navigate to the specified rooms and conduct a two-part check-up.

In the first phase, the robot would collect vital metrics using integrated sensors such as a heart rate monitor and thermometer, autonomously performing diagnostics and flagging any abnormalities. In the second phase, the robot would offer a user-friendly interface that enables patients, nurses, or family members to append new symptoms to their chart if necessary. This would be performed through a keypad and touch module sensor, allowing input from a predefined list of symptoms. These functionalities would be managed by an additional Arduino Uno microcontroller, due to additional wires required, responsible for collecting and processing patient data.

A secondary Arduino Uno, equipped with an ESP32-S3 WiFi module, would handle communi-

cation with external systems. The two microcontrollers would communicate over a serial interface, with the secondary controller receiving patient data from the primary unit. The controller would then trigger an API call to OpenAI's GPT-3.5, which analyzes the symptoms and provides relief advice by displaying the robot's interface. Simultaneously, the collected symptom data would be sent to a web server running a Python-based machine learning model to predict potential diagnoses and suggest treatment. This information could be automatically logged in a SQLite 3 database, enabling doctors to review the data and take appropriate action through the remote interface.

Figure 8 presents a proposed design for the robot's user interface, featuring an additional Arduino integrated with a large liquid crystal display (LCD), a touch sensor module, and a keypad. This setup would enable the patient to interact with the system. The LCD would provide real-time feedback from the software, while the touch sensor and keypad would facilitate user input, allowing the patient to submit data or send notifications to medical personnel when needed.

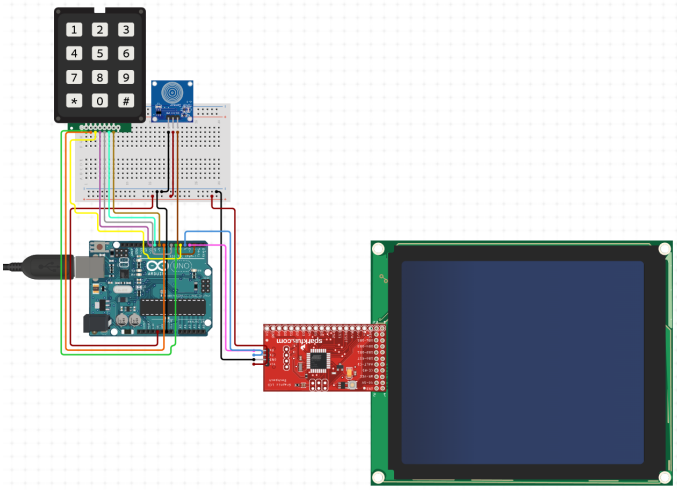

Figure 8. Schematic of potential user interface

Figure 9 illustrates the potential integration of a thermometer and heart rate pulse sensor, enabling the robot to perform basic health assessments, such as measuring temperature and monitoring heart rate, and relay the information back to medical personnel.

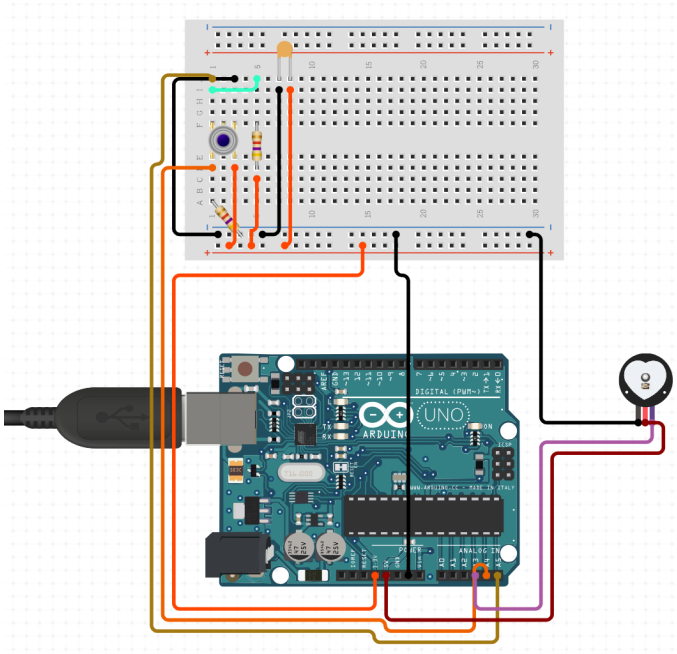

Figure 9. Schematic of potential sensors

6. CONCLUSION

In conclusion, this report presents the development of an autonomous medical assistant robot aimed at revolutionizing patient care through independent navigation, disease diagnosis, and medication recommendations. By leveraging sensor fusion of LiDAR and ultrasonic sensors, the robot achieves precise obstacle avoidance in dynamic environments, while its Random Forest Classifier, trained on a symptom-disease dataset, provides robust disease prediction. Additionally, GPT-3.5-turbo enhances the system's utility by offering interpretable medication recommendations. The hardware and software components were selected and integrated with cost-efficiency and functionality in mind, and future improvements include PCB design optimization and the integration of an intuitive user interface for enhanced patient interaction. Looking ahead, expanding the dataset to cover rare diseases and developing clinically safe interpretation methods for LLM outputs are key directions for continued improvement. This project demonstrates the potential for autonomous robotic systems to streamline healthcare, reducing the burden on medical staff and improving patient outcomes.

APPENDICES

A. HARDWARE

The following algorithm outlines the core functionality of the robot's hardware navigation system, with comments provided to explain the purpose and execution of each section of the code.

Code 1. Hardware System Code

```
#include <Arduino.h>
#include <Wire.h>
#include <TFLI2C.h>
#include <Servo.h>

TFLI2C tfI2C;

int16_t tfDist;
int16_t tfAddr = TFL_DEF_ADR;

int motor1pin1 = 2, motor1pin2 = 3, motor2pin1 = 4, motor2pin2 = 5;
const int trigPinLeft = 7, echoPinLeft = 8, trigPinRight = 12, echoPinRight = 13;
float durationLeft, durationRight, distanceLeft, distanceRight;

Servo myservo;
bool servoDirection = true;
unsigned long previousServoMillis = 0, servoInterval = 190;
const float obstacleThreshold = 30.0;

// Configuration
void setup() {

  for (int pin : {motor1pin1, motor1pin2, motor2pin1, motor2pin2, 11, 10, trigPinLeft, trigPinRight})
    pinMode(pin, OUTPUT);

  pinMode(echoPinLeft, INPUT);
  pinMode(echoPinRight, INPUT);

  myservo.attach(9);
  Serial.begin(115200);
  Wire.begin();
}

void loop() {
  unsigned long currentMillis = millis();

  // Primary drive logic from lidar
  if (tfI2C.getData(tfDist, tfAddr)) {
    if (tfDist > 0) {
      digitalWrite(motor1pin1, LOW);
      digitalWrite(motor1pin2, HIGH);
      digitalWrite(motor2pin1, HIGH);
      digitalWrite(motor2pin2, LOW);
      analogWrite(11, 255);
      analogWrite(10, 255);
    } else {
      analogWrite(11, 0);
      analogWrite(10, 0);
    }
  }

  // Trigger ultrasonic sensors
  for (int pin : {trigPinLeft, trigPinRight}) {
    digitalWrite(pin, LOW);
    delayMicroseconds(2);
    digitalWrite(pin, HIGH);
    delayMicroseconds(10);
    digitalWrite(pin, LOW);
  }
  // Calculate range values from ultrasonic sensors
  durationLeft = pulseIn(echoPinLeft, HIGH);
  durationRight = pulseIn(echoPinRight, HIGH);
  distanceLeft = durationLeft * 0.01715;
  distanceRight = durationRight * 0.01715;

  // Decision making based on sensor distances
  if (distanceLeft <= obstacleThreshold && distanceRight <= obstacleThreshold) {
    // Stop if both sensors detect an obstacle
    analogWrite(11, 0);
    analogWrite(10, 0);
  } else if (distanceLeft <= obstacleThreshold) {
    // Turn right if left sensor
```

```
    detects an obstacle
72    digitalWrite(motor1pin1, HIGH);
73    digitalWrite(motor1pin2, LOW);
74    digitalWrite(motor2pin1, LOW);
75    digitalWrite(motor2pin2, HIGH);
76    analogWrite(11, 255);
77    analogWrite(10, 255);
78  } else if (distanceRight <=
    obstacleThreshold) {
79    // Turn left if right sensor
    detects an obstacle
80    digitalWrite(motor1pin1, LOW);
81    digitalWrite(motor1pin2, HIGH);
82    digitalWrite(motor2pin1, HIGH);
83    digitalWrite(motor2pin2, LOW);
84    analogWrite(11, 255);
85    analogWrite(10, 255);
86  } else {
87    // Move forward if no obstacles
88    digitalWrite(motor1pin1, LOW);
89    digitalWrite(motor1pin2, HIGH);
90    digitalWrite(motor2pin1, HIGH);
91    digitalWrite(motor2pin2, LOW);
92    analogWrite(11, 255);
93    analogWrite(10, 255);
94  }
95  // Delay to avoid overloading
96  delay(50);
97 }
```

B. ARTIFICIAL INTELLIGENCE

Below are more details behind the model and machine learning

B.1. Random Forest Classifier

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. Trees in the forest use the best split strategy, which is equivalent to passing `splitter="best"` to the underlying `DecisionTreeRegressor`[5]. The sub-sample size is controlled with the `max_samples` parameter if `bootstrap=True` (default); otherwise, the whole dataset is used to build each tree.

B.2. Parameters

`n_estimatorsint, default=100`
The number of trees in the forest.

`criterion{"gini", "entropy", "log_loss"}, default="gini"`
The function to measure the quality of a split.
Note: REMEMBER TO ADD INFO ABOUT SHANNON MATH EQN.

`max_depthint, default=None`
The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than `min_samples_split` samples.

`min_samples_splitint or float, default=2`
The minimum number of samples required to split an internal node.

`min_samples_leafint or float, default=1`
The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

`min_weight_fraction_leaffloat, default=0.0`
The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when `sample_weight` is not provided.

`max_features{"sqrt", "log2", None}, int or float, default="sqrt"`
The number of features to consider when looking for the best split:

- If `int`, then consider `max_features` features at each split.
- If `float`, then `max_features` is a fraction and `max(1, int(max_features * n_features_in_))` features are considered at each split.
- If `"sqrt"`, then `max_features=sqrt(n_features)`.
- If `"log2"`, then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

`max_leaf_nodesint, default=None`
Grow trees with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None, then an unlimited number of leaf nodes.

`min_impurity_decreasefloat, default=0.0`
A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

$$\frac{N_t}{N} \times \left(\text{imp} - \frac{N_{t_R}}{N_t} \times \text{right\_imp} - \frac{N_{t_L}}{N_t} \times \text{left\_imp} \right) \quad (1)$$

where N is the total number of samples, N_t is the number of samples at the current node, N_{t_L} is the number of samples in the left child, and N_{t_R} is the number of samples in the right child.

N, N_t, N_{t_R} and N_{t_L} all refer to the weighted sum if `sample_weight` is passed.

`bootstrap`bool, default=True
Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.

`oob_score`bool or `callable`, default=False
Whether to use out-of-bag samples to estimate the generalization score. By default, `accuracy_score` is used. Provide a callable with the signature `metric(y_true, y_pred)` to use a custom metric. Only available if `bootstrap=True`.

`n_jobs`int, default=None
The number of jobs to run in parallel. `fit`, `predict`, `decision_path`, and `apply` are all parallelized over the trees. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details.

`random_state`int, RandomState instance or None, default=None
Controls both the randomness of the bootstrapping of the samples used when building trees (if `bootstrap=True`) and the sampling of the features to consider when looking for the best split at each node (if `max_features < n_features`). See Glossary for details.

`verbose`int, default=0
Controls the verbosity when fitting and predicting.

`warm_start`bool, default=False
When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble; otherwise, fit a whole new forest. See Glossary and Fitting additional trees for details.

`class_weight`{"balanced", "balanced_subsample"}, dict or list of dicts, default=None
Weights associated with classes in the form {`class_label: weight`}. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification, weights should be {{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}} instead of {{1:1}, {2:5}, {3:1}, {4:1}}.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

The "balanced_subsample" mode is the same as "balanced" except that weights are computed based on the bootstrap sample for every tree grown.

For multi-output, the weights of each column of y will be multiplied.

Note that these weights will be multiplied with `sample_weight` (passed through the fit method) if `sample_weight` is specified.

`ccp_alpha`non-negative float, default=0.0
Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity

References

[1] Adascalitei, Florentina & Doroftei, Ioan. (2011). Practical applications for mobile robots based on Mecanum wheels - a systematic survey. Romanian Review Precision Mechanics, Optics and Mechatronics. 21-29.

[2] Jain, U., Kansal, V., Dewangan, R., Dhasmana, G., Kotiyal, A. (2023). Performance Comparison of HC-SR04 Ultrasonic Sensor and TF-Luna LIDAR for Obstacle Detection. In: Tistarelli, M., Dubey, S.R., Singh, S.K., Jiang, X. (eds) Computer Vision and Machine Intelligence. Lecture Notes in Networks and Systems, vol 586. Springer, Singapore. https://doi.org/10.1007/978-981-19-7867-8$_5$0

[3] Yeong, De Jong & Velasco-Hernandez, Gustavo & Barry, John & Walsh, Joseph. (2021). Sensor and Sensor Fusion Technology in Autonomous Vehicles: A Review. 10.20944/preprints202102.0459.v1.

[4] Rosdi, M.H.A.B., Abdul Ghani, A.S. (2022). Investigation on Accuracy of Sensors in Sensor Fusion for Object Detection of Autonomous Vehicle Based on 2D Lidar and Ultrasonic Sensor. In: Ab. Nasir, A.F., Ibrahim, A.N., Ishak, I., Mat Yahya, N., Zakaria, M.A., P. P. Abdul Majeed, A. (eds) Recent

Trends in Mechatronics Towards Industry 4.0. Lecture Notes in Electrical Engineering, vol 730. Springer, Singapore. https://doi.org/10.1007/978-981-33-4597-$3_6$8

[5]Breiman, Leo. (2001). Random Forests. University of California, Berkeley. Retrieved from https://www.stat.berkeley.edu/ breiman/randomforest2001.pdf

[6]SQLite Developers. (2024). Schema Objects. SQLite Documentation. Retrieved from https://www.sqlite.org/schematab.html