Project Report: Rover Bot for Autonomous Maneuvering using GPS-GROUP 27

TA ASSIGNED- Pallav sir

Team Members

- Aman Raj (2023EEB1183)
- Ansh Singh (2023EEB1186)
- Jaskirat Singh (2023EEB1212)
- Vipul Kumar Singh (2023EEB1254)
- Yashraj Omar (2023EEB1258)

Objective

To develop a GPS-guided autonomous rover that can:
- Navigate to a set of GPS coordinates.
- Calculate orientation and heading using a magnetometer.
- Maneuver around obstacles using programmed logic.

Project Development Timeline

Week 1: Planning and Initial Testing

- Finalized the project stages:
  1. Component testing
  2. Circuit diagram and schematic
  3. Pseudocode creation
- Successfully tested:
  - ESP32 module
  - NEO-6M GPS Module
- Created initial circuit diagram and added magnetometer for angle/heading calculation.

Week 2: Coding and Hardware Integration

- Converted pseudocode into working Arduino code.
- Began hardware assembly based on the schematic.
- Challenges:
  - Faulty motor driver (only 2 output pins functional).
  - ESP32 powering issues.
  - GPS not returning precise latitude-longitude values.

Week 3: Troubleshooting GPS Issues

- Completed hardware setup.
- Major issue: GPS not returning correct latitude and longitude.
- Attempted:
  - Switching from ESP32 Wrover to ESP32 Wroom.
  - Consulting seniors and experts.
  - Considering hardcoded GPS coordinates as a fallback.

Solution: Borrowed Magnetometer ,GPS Module NEO-6M and ESP32 Wroom module(works better with GPS)from a senior only for video purposes.

Working Principle

The Rover Bot is designed to autonomously navigate from one location to another using GPS coordinates. The working of the bot involves multiple stages, each handling a different function — sensing, computation, decision-making, and motion control.

1. Initialization:

- The ESP32 microcontroller initializes all necessary components: the GPS module (NEO-6M), the magnetometer (e.g., HMC5883L), and the motor driver.
- It also sets up the UART communication for the GPS and I2C communication for the magnetometer.

2. Receiving GPS Coordinates:

- The NEO-6M GPS module continuously receives satellite signals and outputs location data in NMEA format.
- The ESP32 parses this data and extracts the latitude and longitude of the current position of the rover.
- A target destination is either predefined in the code or updated externally.

3. Calculating Distance and Bearing:

- Using the current and destination GPS coordinates, the ESP32 computes the distance to the target and the bearing (direction) using spherical trigonometry formulas.
- This bearing is the angle the bot should maintain with respect to the Earth's north to

reach the destination.

4. Determining Heading Using Magnetometer:

- The magnetometer provides the current orientation (heading) of the rover with respect to magnetic north.
- This heading is compared with the calculated bearing to determine how much the bot needs to turn.

5. Decision-Making and Motor Control:

- Based on the heading error (difference between the current heading and the desired bearing), the ESP32 decides:
  - To move straight (if aligned),
  - Turn left (if the bot needs to correct leftward), or
  - Turn right.
- It sends appropriate PWM and logic signals to the motor driver (e.g., L298N), which controls the two DC motors accordingly.
- One motor is slowed down or reversed to facilitate turning.

6. Continuous Loop:

- The bot continuously updates its GPS position and heading.
- The comparison and correction loop continues until the rover reaches the vicinity of the target destination (within an acceptable error threshold, typically a few meters).

7. Error Handling and Fail-Safes:

- If GPS data is unavailable or unreliable, the bot can either halt or rely on hardcoded backup paths.
- In future versions, obstacle avoidance can be added using ultrasonic sensors to detect nearby barriers.

This sequence ensures that the rover navigates autonomously and adapts its direction in real-time as it progresses toward the goal.

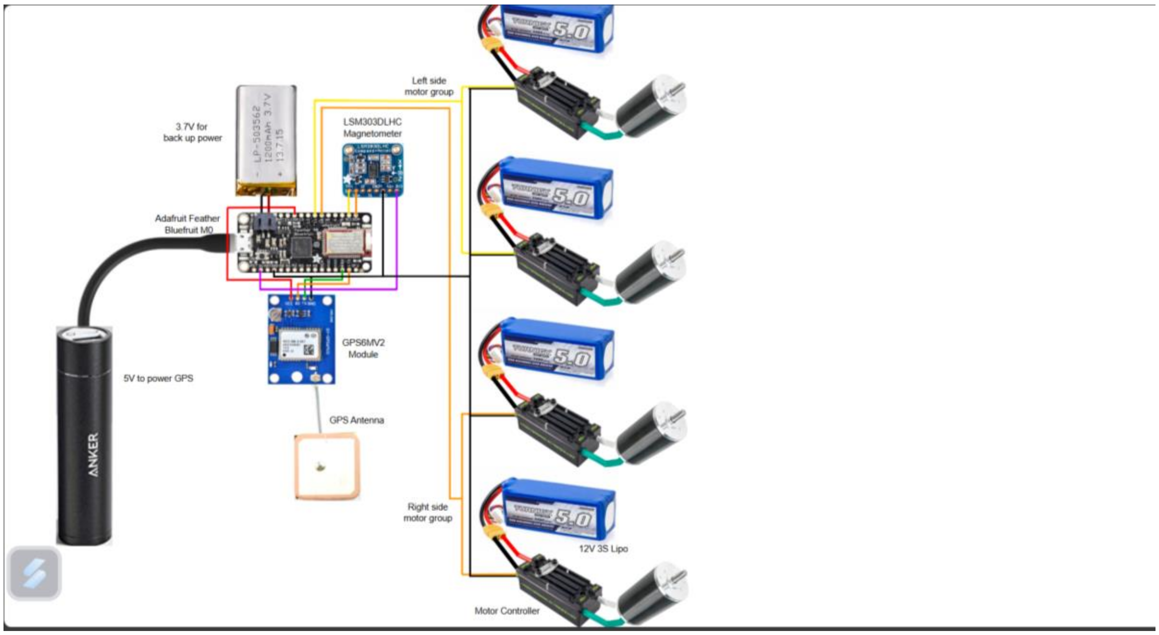
Circuit Diagram Explanation

The circuit includes the following components:

- ESP32 Board: Central control unit.
- NEO-6M GPS Module: Connected to ESP32 via UART for receiving GPS data.
- Magnetometer: I2C-connected to ESP32 for heading calculation.
- Motor Driver (L298N or similar): Drives two DC motors.
- Power Supply: Either via USB/laptop or dedicated Li-ion battery.

Key connections:

- GPS TX → ESP32 RX
- GPS RX → ESP32 TX
- Magnetometer SCL/SDA → ESP32 GPIOs with I2C support
- Motor driver inputs → ESP32 digital pins
- Motors → Motor driver outputs



OUR CODE (NEXT PAGE)

PART-1

```cpp
#include <TinyGPS++.h>
#include <math.h>

// Define the RX and TX pins for Serial 2
#define RXD2 16
#define TXD2 17

#define GPS_BAUD 9600

// The TinyGPS++ object
TinyGPSPlus gps;

// Create an instance of the HardwareSerial class for Serial 2
HardwareSerial gpsSerial(2);

void setup() {
  // Serial Monitor
  Serial.begin(115200);

  // Start Serial 2 with the defined RX and TX pins and a baud rate of 9600
  gpsSerial.begin(GPS_BAUD, SERIAL_8N1, RXD2, TXD2);
  Serial.println("Serial 2 started at 9600 baud rate");
}

void loop() {
  // This sketch displays information every time a new sentence is correctly encoded.
  unsigned long start = millis();

  while (millis() - start < 1000) {
    while (gpsSerial.available() > 0) {
      gps.encode(gpsSerial.read());
    }
```

```cpp
    if (gps.location.isUpdated()) {
      Serial.print("LAT: ");
      Serial.println(gps.location.lat(), 6);
      Serial.print("LONG: ");
      Serial.println(gps.location.lng(), 6);
      Serial.print("SPEED (km/h) = ");
      Serial.println(gps.speed.kmph());
      Serial.print("ALT (min)= ");
      Serial.println(gps.altitude.meters());
      Serial.print("HDOP = ");
      Serial.println(gps.hdop.value() / 100.0);
      Serial.print("Satellites = ");
      Serial.println(gps.satellites.value());
      Serial.print("Time in UTC: ");
      Serial.println(String(gps.date.year()) + "/" + String(gps.date.month()) + "/" + String(gps.date.day()) + "," + String(gps.time.hour()) +
      Serial.println("");
    }
  }
}





// Function to convert degrees to radians
float degreesToRadians(float degrees) {
  return degrees * (M_PI / 180.0);
}

// Function to calculate the distance between two waypoints using the Haversine formula
float haversine(float lat1, float lon1, float lat2, float lon2) {
  // Radius of Earth in meters
```

ESP32 Wrover Module

Code_for_rover.ino

```cpp
float haversine(float lat1, float lon1, float lat2, float lon2) {
  // Radius of Earth in meters
  const float R = 6371000.0; // Earth's radius in meters

  // Convert latitude and longitude from degrees to radians
  lat1 = degreesToRadians(lat1);
  lon1 = degreesToRadians(lon1);
  lat2 = degreesToRadians(lat2);
  lon2 = degreesToRadians(lon2);

  // Differences in coordinates
  float dlat = lat2 - lat1;
  float dlon = lon2 - lon1;

  // Haversine formula
  float a = sin(dlat / 2) * sin(dlat / 2) + cos(lat1) * cos(lat2) * sin(dlon / 2) * sin(dlon / 2);
  float c = 2 * atan2(sqrt(a), sqrt(1 - a));

  // Distance in meters
  float distance = R * c;

  return distance;
}

float calculateBearing(float lat1, float lon1, float lat2, float lon2) {
  // Convert latitude and longitude from degrees to radians
  lat1 = degreesToRadians(lat1);
  lon1 = degreesToRadians(lon1);
  lat2 = degreesToRadians(lat2);
  lon2 = degreesToRadians(lon2);

  // Difference in longitudes
```

Code_for_rover.ino

```cpp
  float dlon = lon2 - lon1;

  // Calculate the bearing
  float x = sin(dlon) * cos(lat2);
  float y = cos(lat1) * sin(lat2) - sin(lat1) * cos(lat2) * cos(dlon);
  float bearing = atan2(x, y);

  // Convert bearing from radians to degrees
  bearing = radiansToDegrees(bearing);

  // Normalize the bearing to a value between 0 and 360 degrees
  if (bearing < 0) {
    bearing += 360.0;
  }

  return bearing;
}

#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_HMC5883_U.h>
#include <math.h>

float normalizeAngle(float angle) {
  while (angle > 180.0) angle -= 360.0;
  while (angle < -180.0) angle += 360.0;
  return angle;
}

// To get heading from compass
float getCompassHeading() {
  sensors_event_t event;
```

PART 2

Code_for_rover.ino

```cpp
  sensors_event_t event;
  mag.getEvent(&event);

  float heading = atan2(event.magnetic.y, event.magnetic.x);
  heading = radiansToDegrees(heading);
  if (heading < 0) heading += 360.0;

  return heading;
}
void setup() {
  Serial.begin(115200);
  Serial.println("Initializing HMC5883L...");

  if (!mag.begin()) {
    Serial.println("No HMC5883L found. Check wiring!");
    while (1);
  }

  Serial.println("Sensor initialized.");
}
void loop() {
  // Simulated GPS coordinates (replace with actual GPS readings)
  float currentLat = 40.7128;
  float currentLon = -74.0060;
  float targetLat = 40.7138;
  float targetLon = -74.0050;

  // Calculate steering angle
  float steeringAngle = calculateBearing(currentLat, currentLon, targetLat, targetLon);

  // Get current heading from compass
```

Code_for_rover.ino

```cpp
    float currentHeading = getCompassHeading();

    // Calculate the angle error
    float error = normalizeAngle(steeringAngle - currentHeading);
  // Print values
    Serial.print("Current Heading: ");
    Serial.print(currentHeading);
    Serial.println("°");

    Serial.print("Target Bearing: ");
    Serial.print(steeringAngle);
    Serial.println("°");

    Serial.print("Heading Error: ");
    Serial.print(error);
    Serial.println("°");

  // Decide direction
    if (error > 5.0) {
      Serial.println("Turn RIGHT");
    } else if (error < -5.0) {
      Serial.println("Turn LEFT");
    } else {
      Serial.println("Go STRAIGHT");
    }

    Serial.println("------------");
    delay(1000); // Delay for readability
  }

  const int enA = 33;
      // PWM pin for left motors
```

```cpp
const int enA = 33;
    // PWM pin for left motors
const int in1 = 25;
const int in2 = 26;     // Direction pin 2 for left motors

// Define motor driver pins for the right side (Channel B)
const int enB = 32;   // PWM pin for right motors
const int in3 = 27;    // Direction pin 1 for right motors
const int in4 = 14;    // Direction pin 2 for right motors

const int baseSpeed=200;

void setup() {
  // put your setup code here, to run once:

  Serial.begin(115200);

}

const float Kp = 2.0;
// Base speed for motors (adjust between 0-255 as needed)
const int baseSpeed = 150;


void adjust_motors(float error) {
  // Calculate a correction value
  int correction = (int)(Kp * error);

  // Adjust left and right speeds based on correction.
  // If error is positive, the right motor gets extra speed (to turn right),
  // while the left motor slows down. Vice versa if error is negative.
  int leftSpeed = baseSpeed - correction;
```

```cpp
  int leftSpeed = baseSpeed - correction;
  int rightSpeed = baseSpeed + correction;

  // Constrain speeds to valid PWM range (0 to 255)
  leftSpeed = constrain(leftSpeed, 0, 255);
  rightSpeed = constrain(rightSpeed, 0, 255);

  // Set left motors to move forward
  digitalWrite(in1, HIGH);
  digitalWrite(in2, LOW);
  analogWrite(enA, leftSpeed);

  // Set right motors to move forward
  digitalWrite(in3, HIGH);
  digitalWrite(in4, LOW);
  analogWrite(enB, rightSpeed);
}
//pins:

const int enA = 9;    // PWM pin for left motors
const int in1 = 8;    // Direction pin 1 for left motors
const int in2 = 7;    // Direction pin 2 for left motors

// Define motor driver pins for the right side (Channel B)
const int enB = 10;   // PWM pin for right motors
const int in3 = 6;    // Direction pin 1 for right motors
const int in4 = 5;    // Direction pin 2 for right motors

void drive_forward() {
  // Set left motors to move forward
  digitalWrite(in1, HIGH);
```

```cpp
  digitalWrite(in2, LOW);

  // Set right motors to move forward
  digitalWrite(in3, HIGH);
  digitalWrite(in4, LOW);

  // Start the motors with the base speed
  analogWrite(enA, baseSpeed);
  analogWrite(enB, baseSpeed);
}

void stop_motors() {
  // Stop left motors
  digitalWrite(in1, LOW);
  digitalWrite(in2, LOW);
  analogWrite(enA, 0);

  // Stop right motors
  digitalWrite(in3, LOW);
  digitalWrite(in4, LOW);
  analogWrite(enB, 0);
}

void loop() {
  Serial.println("namaste");
  drive_forward();
  Serial.println("done");
  delay(5000);
  stop_motors();
  Serial.println("done2");
  delay(5000);
}
```

Code Overview

1. Initialize GPS, Magnetometer, Motors
2. Read current GPS coordinates
3. Compare with destination coordinates
4. Calculate bearing using magnetometer
5. Adjust rover direction using motors
6. Repeat until destination is reached

Applications in Industry

1. Agriculture:
   - Autonomous tractors or equipment navigation using GPS for precision farming.

2. Logistics & Warehousing:
   - Self-navigating trolleys for transporting materials within large facilities.

3. Surveillance and Patrolling:
   - Autonomous bots for border surveillance, area patrolling using predefined GPS routes.

4. Mining and Exploration:
   - GPS-guided rovers to explore hazardous zones without human involvement.

5. Disaster Management:
   - Deployment of autonomous bots in disaster-hit areas where manual intervention is risky.

Challenges and Future Scope

Challenges:

- GPS accuracy and signal inconsistency
- Power reliability for ESP32-ESP sourced power from computer instead of battery
- Hardware faults (motor driver output pins not dissapiating equal power to all motors,output pins of motor driver not working properly,etc)

Future Improvements:

- Integration of obstacle detection using ultrasonic or LiDAR sensors.
- Real-time tracking and remote monitoring of diverse remote terrains
- Improved GPS alternatives like RTK-GPS for centimeter-level accuracy.