Development of a Multi-Vehicle Remote-Controlled Car System Using Arduino and Wireless Communication

1st Assem Eldlebshany
Information Engineering
Technical University Of Munich
Heilbronn, Germany
assem.eldlebshany@tum.de

2nd Dylan Neoh
Information Engineering
Technical University Of Munich
Heilbronn, Germany
go69zog@mytum.de

3rd Erjona Dobra
Information Engineering
Technical University Of Munich
Heilbronn, Germany
erjona.dobra@tum.de

4th Aly Moaawad
Information Engineering
Technical University Of Munich
Heilbronn, Germany
aly.moaawad@tum.de

5th Timo Robrecht
Information Engineering
Technical University Of Munich
Heilbronn, Germany
timo.robrecht@tum.de

6th Vanesa Metaj
Information Engineering
Technical University Of Munich
Heilbronn, Germany
go72tec@mytum.de

7th Vuk Trpkovic
Information Engineering
Technical University Of Munich
Heilbronn, Germany
vuk.trpkovic@tum.de

8th Youssef Soliman
Information Engineering
Technical University Of Munich
Heilbronn, Germany
youssef.soliman@tum.de

***Abstract*—The DAT Racer project presents an innovative approach to remote-controlled vehicle operation through computer vision-based hand gesture recognition. This system eliminates the need for traditional physical controllers by implementing a contactless interface that interprets hand movements to control RC car speed and steering. The project evolved from a single-player system to a sophisticated two-player competitive platform, utilizing Arduino microcontrollers, nRF24L01 wireless modules, and Python-based MediaPipe hand tracking technology.**

The system architecture consists of three main components: a Python-based computer vision controller that processes hand gestures using MediaPipe, an Arduino Uno server that manages wireless communication, and Arduino Nano-based car receivers that translate commands into motor control. Initial development challenges with hardware connectivity and power supply were systematically addressed through iterative design improvements, including the implementation of welding connections and upgrade to lithium battery systems for enhanced performance and reliability.

The final implementation successfully demonstrates real-time gesture control with smooth speed modulation based on hand openness (0-100% range) and precise steering control through hand tilt angles (±45°). The multiplayer functionality allows two players to simultaneously control separate vehicles using left and right screen zones, creating an engaging competitive experience. This project showcases the practical integration of computer vision, embedded systems, and wireless communication technologies in creating intuitive human-machine interfaces.

I. Introduction

The intersection of computer vision, embedded systems, and human-computer interaction has opened new possibilities for intuitive control interfaces [1]. Traditional remote-controlled vehicles rely on physical controllers with buttons, joysticks, and switches that create a barrier between the user's intentions and the vehicle's response. The DAT Racer project addresses this limitation by implementing a natural, gesture-based control system that interprets hand movements to provide seamless vehicle operation.

The concept of gesture-controlled robotics represents a significant advancement in making technology more accessible and intuitive [2], [3]. By leveraging the natural movements of human hands, this system reduces the learning curve typically associated with traditional RC vehicle operation while providing a more immersive and engaging user experience. The project name "DAT Racer" reflects both the technical achievement and the competitive racing aspect of the final implementation, while building upon robust frameworks such as MediaPipe for vision-based hand tracking [4].

A. Project Motivation and Objectives

The primary motivation for this project stems from the desire to create a more natural and intuitive interface for vehicle control, moving beyond the constraints of physical input devices. The specific objectives include:

- Development of a real-time hand gesture recognition system capable of interpreting speed and steering commands [2]

- Implementation of a robust wireless communication system between multiple vehicles and a central control unit
- Creation of a multiplayer platform supporting simultaneous two-player operation
- Integration of embedded systems programming with computer vision techniques
- Demonstration of practical applications for contactless control interfaces [1]

B. System Overview

The DAT Racer system employs a distributed architecture consisting of three primary subsystems working in concert. The computer vision subsystem utilizes Python with MediaPipe libraries to process real-time video input from a webcam, analyzing hand positions, orientations, and finger configurations to determine control commands [4]. These commands are then transmitted via serial communication to an Arduino Uno server, which acts as a central communication hub [5].

The Arduino Uno server processes the incoming commands and distributes them wirelessly using nRF24L01 radio frequency modules [6] to individual Arduino Nano-based car controllers [7]. Each car receiver translates the wireless commands into precise motor control signals, enabling smooth acceleration, deceleration, and steering responses. This architecture ensures low-latency communication while maintaining the flexibility to support multiple vehicles simultaneously.

C. Development Evolution

The project development followed an iterative approach, beginning with a proof-of-concept single-player system implemented during the initial development phase [8]. Initial challenges with hardware reliability, including connection stability and insufficient power delivery from AA batteries, necessitated systematic improvements in both hardware design and software implementation [9].

The transition from AA batteries to lithium battery systems addressed power delivery issues [10], while the implementation of welded connections improved the mechanical stability of the hardware assemblies. These hardware improvements, combined with refined software algorithms for gesture recognition and motor control, resulted in a robust and reliable system capable of supporting competitive multiplayer gameplay.

The evolution to a two-player system required significant modifications to both the communication protocol and the user interface, demonstrating the scalability and flexibility of the underlying architecture. This progression from a simple proof-of-concept to a fully functional multiplayer platform illustrates the project's technical depth and practical applicability.

II. Hardware Construction and Initial Testing

A. Component Procurement and Assembly

The DAT Racer project began with the procurement of all necessary hardware components from RAM Electronics [11]. The complete component list included:

- 2× 4WD car chassis with integrated wheel mounting systems
- 2× Sets of rotator motors for vehicle propulsion
- 2× AA battery holders for initial power supply testing
- 2× Complete wheel sets compatible with the chassis design
- 1× Arduino Uno microcontroller for central command processing [5]
- 2× Arduino Nano microcontrollers for individual car control [7]
- 3× nRF24L01 wireless communication modules [6]
- 3× 100 μF capacitors for power supply stabilization
- 2× L298N Motor Driver red boards for motor control interface [12]
- Multiple jumper wires for circuit connections

B. Initial Circuit Configuration

The initial hardware assembly focused on establishing the basic motor control circuitry without implementing wireless communication functionality. Each car's control system is centered around an Arduino Nano connected to an L298N Motor Driver board, which served as the interface between the microcontroller's digital signals and the high-current motor requirements [12].

The L298N Motor Driver configuration followed standard dual H-bridge wiring protocols [13]. The Arduino Nano's digital pins were connected to the motor driver's input pins (IN1, IN2, IN3, IN4) to control motor direction, while PWM-capable pins (ENA, ENB) provided speed control through pulse-width modulation [7]. The motor driver's output terminals were connected directly to the rotator motors with careful attention to maintaining consistent polarity for predictable directional control.

The power distribution utilized the AA battery holders connected to the motor driver's power input, with the Arduino Nano receiving regulated power through the motor driver's 5V output. The 100μF capacitors were installed across the power rails to provide stable voltage regulation and reduce electrical noise that could interfere with the operation of the microcontroller [14].

C. Initial Hardware Testing

Following the mechanical assembly and basic wiring, preliminary testing was conducted using fundamental Arduino Nano code to verify motor functionality and identify potential hardware issues. The test code implemented basic forward motion without any advanced control algorithms:

```
// Basic Car Movement Test - Arduino Nano
// Simple forward movement test without wireless communication

// L298N Motor Driver Pins
const int ENA = 5;   // PWM pin for left motors speed
const int IN1 = 6;   // Left motors direction pin 1
const int IN2 = 7;   // Left motors direction pin 2
```

```
const int ENB = 3;   // PWM pin for right motors speed
const int IN3 = 4;   // Right motors direction pin 1
const int IN4 = 8;   // Right motors direction pin 2

void setup() {
  Serial.begin(9600);
  Serial.println("Car Hardware Test Starting...");

  // Setup motor pins
  pinMode(ENA, OUTPUT);
  pinMode(IN1, OUTPUT);
  pinMode(IN2, OUTPUT);
  pinMode(ENB, OUTPUT);
  pinMode(IN3, OUTPUT);
  pinMode(IN4, OUTPUT);

  // Initial stop
  stopMotors();

  Serial.println("Motors initialized - Starting test sequence");
  delay(2000);
}

void loop() {
  Serial.println("Moving forward at 50% speed...");
  moveForward(127); // 50% of 255 PWM
  delay(3000);

  Serial.println("Moving forward at 75% speed...");
  moveForward(190); // 75% of 255 PWM
  delay(3000);

  Serial.println("Moving forward at 100% speed...");
  moveForward(255); // 100% PWM
  delay(3000);

  Serial.println("Stopping...");
  stopMotors();
  delay(2000);
}

void moveForward(int speed) {
  // Left motors forward
  digitalWrite(IN1, HIGH);
  digitalWrite(IN2, LOW);
  analogWrite(ENA, speed);

  // Right motors forward
  digitalWrite(IN3, HIGH);
  digitalWrite(IN4, LOW);
  analogWrite(ENB, speed);
}

void stopMotors() {
  // Stop all motors
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, LOW);
  digitalWrite(IN3, LOW);
  digitalWrite(IN4, LOW);

  analogWrite(ENA, 0);
  analogWrite(ENB, 0);
}
```

Listing 1: Basic Arduino Nano motor driver test for forward movement

D. Initial Hardware Issues and Diagnosis

The initial testing revealed significant performance deficiencies that indicated underlying hardware problems. Despite the test code executing correctly and the motor driver receiving appropriate PWM signals, the wheels exhibited minimal rotation and insufficient torque to achieve meaningful vehicle movement. This behavior suggested either inadequate power delivery to the motors or poor electrical connections within the circuit [9], [15].

The diagnostic process involved systematic testing of individual components to isolate the source of the performance issues. Voltage measurements across the motor terminals confirmed that the L298 driver was receiving adequate input voltage from the AA battery supply, however the output power to the rotator motors remained insufficient for normal operation [16].

E. MakerSpace Welding Solution

Recognizing that loose jumper wire connections might be contributing to the power delivery issues, access to professional-grade soldering equipment at the Experimenta MakerSpace in Heilbronn was obtained. The welding process involved replacing all critical jumper wire connections with soldered joints, particularly focusing on the high-current paths between the motor driver outputs and the rotator motor inputs.

The welding process significantly improved connection reliability and consistency. Post-welding testing demonstrated more predictable motor behavior, with the wheels showing consistent response to PWM signals. However, the fundamental issue of insufficient starting torque persisted, manifesting as a condition where the wheels would only begin rotation when given a manual initial push to overcome static friction.

This improved consistency, while not solving the underlying power issue, provided valuable diagnostic information. The fact that the motors could maintain rotation once started, but required external assistance to initiate movement, indicated that the system was operating at the threshold of the motors' minimum operating requirements. This observation guided subsequent investigations into power supply limitations and ultimately led to the decision to upgrade from AA batteries to higher-capacity lithium battery systems.

III. Voltage Diagnostics and Power Supply Resolution

A. Voltage Monitoring Implementation

To systematically diagnose the persistent motor performance issues, voltage monitoring capabilities were implemented in the Arduino Nano test code. This diagnostic approach involved reading the analog voltage levels at critical points in the circuit to identify potential power supply inadequacies. The enhanced test code incorporated real-time voltage measurements to correlate motor performance with available power.

```
// Voltage Monitoring

void loop() {
  // Measure voltage before motor operation
```

```
  float voltageIdle = measureVoltage();
  moveForward(127);
  delay(1000); // Allow voltage to stabilize under load
  float voltageUnderLoad = measureVoltage();
  float voltageDrop = voltageIdle - voltageUnderLoad;

  // Check for low voltage condition
  if (voltageUnderLoad < 4.5) {
    Serial.println("WARNING: Low voltage detected!");
    Serial.println("Battery may be insufficient for motor operation");
  }

  if (voltageDrop > 1.0) {
    Serial.println("WARNING: Excessive voltage drop under load!");
    Serial.println("Power supply may be inadequate");
  }
}

float measureVoltage() {
  int analogValue = analogRead(VOLTAGE_PIN);
  float voltage = (analogValue / 1024.0) * VREF;

  // Adjust for voltage divider if used
  voltage = voltage * ((R1 + R2) / R2);

  return voltage;
}
```

Listing 2: Battery voltage monitoring and load drop detection

B. Voltage Analysis Results

The voltage monitoring implementation provided conclusive evidence of the underlying power supply inadequacy. During idle operation, the AA battery pack maintained acceptable voltage levels around 6V (four 1.5V cells in series). However, when the motors were activated and placed under load, the voltage measurements revealed dramatic drops to approximately 3.5-4.0V, well below the minimum operating threshold required for reliable motor operation.

The L298 motor driver requires a minimum input voltage of 4.5V to maintain proper operation, and the rotator motors themselves needed sufficient voltage to overcome static friction and generate adequate starting torque. The excessive voltage drop under load indicated that the AA battery chemistry could not provide the sustained current draw required by the motor system, resulting in the observed insufficient wheel movement.

Additional testing revealed that the voltage drop was most pronounced during motor startup, when the current draw reaches its peak due to the need to overcome static friction in the motor bearings and wheel assemblies. This explained why the wheels would continue rotating once given a manual push but could not initiate movement independently—the system had adequate power to maintain rotation but insufficient power for startup.

C. Lithium Battery Solution

Based on the voltage analysis results, it was determined that upgrading to lithium-ion battery technology would resolve the power delivery limitations. Lithium batteries offer several advantages over alkaline AA cells, including higher energy density, lower internal resistance, and superior current delivery capabilities under load conditions.

The lithium battery implementation involved replacing the AA battery holders with appropriate lithium battery packs capable of delivering the required current while maintaining stable voltage levels. The selection criteria prioritized batteries with sufficient capacity to support extended operation periods and low internal resistance to minimize voltage drop during high-current motor startup sequences.

Following the lithium battery installation, subsequent testing demonstrated immediate and significant improvement in motor performance. The wheels now exhibited strong starting torque and smooth acceleration across the full PWM range, confirming that inadequate power supply had been the primary limiting factor in the original design.

D. Successful Hardware Validation

With the power supply issues resolved through the lithium battery upgrade, the mechanical hardware platform demonstrated reliable and consistent operation. The combination of welded connections and adequate power delivery created a robust foundation for implementing the wireless communication system. The cars could now achieve the full range of motion required for responsive gesture control, including smooth acceleration, deceleration, and sufficient torque for directional changes under load.

This successful hardware validation marked the completion of the mechanical and electrical foundation phase of the project, establishing the necessary prerequisites for implementing the wireless communication protocols and advanced control algorithms that would enable gesture-based operation.

IV. Software Implementation and Wireless Communication

A. Software Architecture Overview

The DAT Racer software system consists of three interconnected components: a Python-based computer vision controller for hand gesture recognition, an Arduino Uno server for wireless communication management, and Arduino Nano receivers for individual car control. This distributed architecture enables scalable multi-vehicle operation while maintaining low-latency command processing.

B. Arduino Uno Server Implementation

The Arduino Uno serves as the central communication hub, receiving commands from the Python controller via serial communication and distributing them to individual cars through wireless transmission.

1) Uno Setup and Initialization: The server initialization establishes both serial communication with the computer and wireless communication with the cars:

```
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>
```

```

RF24 radio(9, 10); // CE, CSN pins
const byte address1[6] = "00001"; // Car 1 address
const byte address2[6] = "00002"; // Car 2 address

void setup() {
  Serial.begin(9600);
  Serial.println("Two-Car Controller Server");

  if (!radio.begin()) {
    Serial.println("ERROR: nRF24L01 not responding!");
    while (1); // Halt execution if radio fails
  }

  radio.setPALevel(RF24_PA_HIGH);
  radio.setDataRate(RF24_250KBPS);
  radio.setRetries(5, 15);
  radio.stopListening(); // Configure as transmitter
}
```

Listing 3: nRF24L01 transmitter setup for a two-car controller

The initialization sequence validates nRF24L01 functionality before proceeding, preventing silent failures that could result in non-responsive vehicle control. The high power setting and low data rate configuration prioritize transmission reliability over speed, essential for real-time vehicle control applications.

2) Command Processing System: The server processes incoming serial commands using a string-based parsing system that supports simultaneous multi-vehicle control:

```
String inputString = "";
boolean stringComplete = false;

void loop() {
  while (Serial.available()) {
    char inChar = (char)Serial.read();
    inputString += inChar;

    if (inChar == '\n') {
      stringComplete = true;
    }
  }

  if (stringComplete) {
    processCommand();
    inputString = "";
    stringComplete = false;
  }
}
```

Listing 4: Serial input handling and command processing

The non-blocking serial processing system accumulates characters until a complete command is received. This approach maintains system responsiveness while ensuring command integrity, preventing partial command execution that could result in erratic vehicle behavior.

3) Multi-Vehicle Command Distribution: The command parsing algorithm extracts vehicle-specific control parameters and distributes them through targeted wireless transmission:

```
void processCommand() {
  int c1Index = inputString.indexOf("C1");
  int c2Index = inputString.indexOf("C2");

  if (c1Index != -1) {
    int speedIndex = inputString.indexOf('S', c1Index);
    int angleIndex = inputString.indexOf('A', c1Index);

    if (speedIndex != -1 && angleIndex != -1) {
      car1Speed = inputString.substring(speedIndex + 1, angleIndex).toInt();
      int angleEnd = (c2Index != -1) ? c2Index : inputString.length();
      car1Angle = inputString.substring(angleIndex + 1, angleEnd).toInt();

      car1Speed = constrain(car1Speed, 0, 100);
      car1Angle = constrain(car1Angle, -45, 45);

      sendToCar1(car1Speed, car1Angle);
    }
  }
}
```

Listing 5: Command parsing and control for Car 1

The parsing system implements robust error handling through parameter validation and constraint enforcement. The `constrain()` function prevents invalid control values from reaching the vehicles, ensuring safe operation even with corrupted or malformed commands.

4) Wireless Transmission Protocol: The wireless transmission system implements address-specific communication to enable selective vehicle control:

```
struct DataPacket {
  int speed;
  int angle;
};

DataPacket dataToSend;

void sendToCar1(int speed, int angle) {
  radio.openWritingPipe(address1);

  dataToSend.speed = speed;
  dataToSend.angle = angle;

  bool result = radio.write(&dataToSend, sizeof(dataToSend));
  Serial.print(result ? " [Car1:OK]" : " [Car1:FAIL]");
}
```

Listing 6: Sending structured data packets to Car 1 via nRF24L01

The address-switching mechanism enables the single radio module to communicate with multiple vehicles by dynamically changing the transmission target. The structured data format ensures consistent parameter interpretation across all system components.

5) Safety Timeout System: A comprehensive safety system prevents vehicle runaway conditions in case of communication failures:

```
unsigned long lastDataTime = 0;
const unsigned long TIMEOUT = 1000; // 1 second timeout
```

```

void loop() {
  // ... command processing code ...

  if (millis() - lastDataTime > TIMEOUT && (car1Speed > 0 || car2Speed > 0)) {
    Serial.println("Timeout - Stopping both cars");
    sendToCar1(0, 0);
    sendToCar2(0, 0);
    car1Speed = 0; car2Speed = 0;
  }
}
```

Listing 7: Failsafe timeout to stop cars when no data is received

The server-side timeout mechanism provides an additional safety layer by monitoring command frequency and automatically issuing stop commands if the Python controller becomes unresponsive.

C. Arduino Nano Car Receiver Implementation

Each Arduino Nano receiver implements the vehicle-specific control logic, translating wireless commands into precise motor control signals.

1) Nano Receiver Setup: The receiver initialization configures both wireless communication and motor control interfaces:

```
RF24 radio(9, 10); // CE, CSN pins
const byte address[6] = "00001"; // Unique address per car

// L298N Motor Driver Pins
const int ENA = 5, IN1 = 6, IN2 = 7;
const int ENB = 3, IN3 = 4, IN4 = 8;

void setup() {
  Serial.begin(9600);
  Serial.println("Arduino Nano CAR 1 Starting...");

  if (!radio.begin()) {
    while (1); // Halt if radio initialization fails
  }

  radio.openReadingPipe(0, address);
  radio.setPALevel(RF24_PA_HIGH);
  radio.setDataRate(RF24_250KBPS);
  radio.startListening(); // Configure as receiver

  // Initialize motor control pins
  pinMode(ENA, OUTPUT); pinMode(IN1, OUTPUT); pinMode(IN2, OUTPUT);
  pinMode(ENB, OUTPUT); pinMode(IN3, OUTPUT); pinMode(IN4, OUTPUT);

  stopCar(); // Ensure safe initial state
}
```

Listing 8: Car 1 Arduino Nano setup with nRF24L01 receiver and motor driver initialization

The receiver configuration mirrors the server settings to ensure communication compatibility. The motor pin initialization and initial stop command establish a safe starting configuration that prevents unexpected vehicle movement during system startup.

2) Wireless Command Reception: The reception system implements continuous monitoring for incoming control commands:

```
struct DataPacket {
  int speed;
  int angle;
};

DataPacket receivedData;
unsigned long lastDataTime = 0;

void loop() {
  if (radio.available()) {
    radio.read(&receivedData, sizeof(receivedData));

    currentSpeed = constrain(receivedData.speed, 0, 100);
    currentAngle = constrain(receivedData.angle, -45, 45);

    controlCar(currentSpeed, currentAngle);

    Serial.print("Received - Speed: ");
    Serial.print(currentSpeed);
    Serial.print("% Angle: ");
    Serial.println(currentAngle);

    lastDataTime = millis();
  }
}
```

Listing 9: Receiving and processing control data via nRF24L01

The reception loop implements parameter validation at the vehicle level, providing defense against corrupted data transmission. The debug output enables real-time monitoring of received commands for system diagnostics and troubleshooting.

3) Advanced Motor Control Algorithm: The motor control system implements sophisticated speed mapping and differential steering for precise vehicle control:

```
void controlCar(int speed, int angle) {
  int baseSpeed;

  if (speed == 0) {
    baseSpeed = 0;
  } else if (speed <= 10) {
    baseSpeed = map(speed, 1, 10, 80, 100);   // Very slow
  } else if (speed <= 30) {
    baseSpeed = map(speed, 11, 30, 100, 150); // Slow to medium
  } else if (speed <= 60) {
    baseSpeed = map(speed, 31, 60, 150, 200); // Medium speed
  } else {
    baseSpeed = map(speed, 61, 100, 200, 255); // Fast to max
  }

  // Calculate differential steering
  float turnRatio = abs(angle) / 45.0;
  int leftSpeed = baseSpeed;
  int rightSpeed = baseSpeed;

  if (angle < 0) { // Turn left
    leftSpeed = baseSpeed * (1 - turnRatio * 0.6);
  } else if (angle > 0) { // Turn right
```

```
    rightSpeed = baseSpeed * (1 - turnRatio * 0.6);
  }

  applyMotorSpeeds(leftSpeed, rightSpeed);
}
```

Listing 10: Car speed scaling and differential steering control

The multi-stage speed mapping provides fine control at low speeds while maintaining full power capability at high speeds. The differential steering algorithm creates natural turning behavior by reducing the speed of the inner wheel during turns, with the turn ratio determining the steering intensity.

4) Motor Driver Interface: The motor driver interface translates control speeds into appropriate PWM and direction signals:

```
void applyMotorSpeeds(int leftSpeed, int rightSpeed) {
  // Left motors
  if (leftSpeed >= 0) {
    digitalWrite(IN1, HIGH);
    digitalWrite(IN2, LOW);
    analogWrite(ENA, abs(leftSpeed));
  } else {
    digitalWrite(IN1, LOW);
    digitalWrite(IN2, HIGH);
    analogWrite(ENA, abs(leftSpeed));
  }

  // Right motors
  if (rightSpeed >= 0) {
    digitalWrite(IN3, HIGH);
    digitalWrite(IN4, LOW);
    analogWrite(ENB, abs(rightSpeed));
  } else {
    digitalWrite(IN3, LOW);
    digitalWrite(IN4, HIGH);
    analogWrite(ENB, abs(rightSpeed));
  }
}
```

Listing 11: Applying motor speeds with direction control

The motor driver interface supports both forward and reverse operation for each motor group, enabling advanced maneuvers such as pivot turns when combined with appropriate speed differentials.

5) Vehicle Safety Systems: Each vehicle implements independent safety systems to prevent runaway operation:

```
const unsigned long TIMEOUT = 500; // 500ms timeout

void loop() {
  // ... reception code ...

  // Safety timeout check
  if (millis() - lastDataTime > TIMEOUT && currentSpeed > 0) {
    stopCar();
    digitalWrite(LED_PIN, LOW);
    Serial.println("Signal lost - Emergency stop!");
  }
}

void stopCar() {
  digitalWrite(IN1, LOW); digitalWrite(IN2, LOW);
  digitalWrite(IN3, LOW); digitalWrite(IN4, LOW);
  analogWrite(ENA, 0); analogWrite(ENB, 0);
  currentSpeed = 0;
}
```

Listing 12: Emergency stop with safety timeout on signal loss

D. Python Computer Vision Controller Implementation

The Python controller serves as the primary user interface, implementing real-time hand gesture recognition and translating detected gestures into vehicle control commands.

1) MediaPipe Integration and Setup: The computer vision system utilizes Google's MediaPipe framework for robust hand tracking and gesture recognition:

```
import cv2
import mediapipe as mp
import serial
import numpy as np
import math

class TwoPlayerHandGestureController:
    def __init__(self, serial_port='/dev/cu.usbmodem1101', baudrate=9600):
        self.mp_hands = mp.solutions.hands
        self.hands = self.mp_hands.Hands(
            static_image_mode=False,
            max_num_hands=2,  # Support two players
            min_detection_confidence=0.7,
            min_tracking_confidence=0.5
        )
        self.mp_drawing = mp.solutions.drawing_utils
```

Listing 13: Two-player hand gesture controller setup with MediaPipe and Serial

The MediaPipe configuration optimizes for real-time performance while maintaining accurate hand detection. The dual-hand support enables simultaneous two-player operation, with detection confidence thresholds set to minimize false positives while ensuring responsive gesture recognition.

2) Serial Communication Interface: The Python controller establishes serial communication with the Arduino Uno server, implementing error handling for robust operation:

```
try:
    self.serial_conn = serial.Serial(serial_port, baudrate, timeout=1)
    time.sleep(2)  # Wait for Arduino initialization
    print(f"Connected to Arduino on {serial_port}")
except:
    self.serial_conn = None
    print("Arduino not connected - running in demo mode")
```

Listing 14: Arduino Serial connection initialization with fallback demo mode

The communication system includes fallback functionality for development and testing, allowing the vision system to operate without hardware connectivity. This design approach enables software development and testing independent of hardware availability.

3) Hand Gesture Analysis Algorithms: The gesture recognition system implements sophisticated algorithms to extract control parameters from hand positioning and configuration:

```
def calculate_hand_openness(self, hand_landmarks):
    wrist = hand_landmarks.landmark[self.mp_hands.HandLandmark.WRIST]

    fingertips = [
        hand_landmarks.landmark[self.mp_hands.HandLandmark.INDEX_FINGER_TIP],
        hand_landmarks.landmark[self.mp_hands.HandLandmark.MIDDLE_FINGER_TIP],
        hand_landmarks.landmark[self.mp_hands.HandLandmark.RING_FINGER_TIP],
        hand_landmarks.landmark[self.mp_hands.HandLandmark.PINKY_TIP]
    ]

    openness_ratios = []
    for fingertip in fingertips:
        tip_to_wrist = math.sqrt(
            (fingertip.x - wrist.x) ** 2 + (fingertip.y - wrist.y) ** 2
        )
        normalized_ratio = (tip_to_wrist - 0.8) / (2.0 - 0.8)
        normalized_ratio = max(0, min(1, normalized_ratio))
        openness_ratios.append(normalized_ratio)

    return np.mean(openness_ratios)
```

Listing 15: Hand openness calculation using MediaPipe landmarks

The hand openness algorithm calculates the relative extension of fingers compared to the palm, providing an intuitive speed control mechanism where open hands represent maximum speed and closed fists represent stopped vehicles.

4) Hand Tilt Detection for Steering Control: The steering control system analyzes hand orientation to determine turning commands:

```
def calculate_hand_tilt(self, hand_landmarks):
    index_mcp = hand_landmarks.landmark[self.mp_hands.HandLandmark.INDEX_FINGER_MCP]
    pinky_mcp = hand_landmarks.landmark[self.mp_hands.HandLandmark.PINKY_MCP]

    dx = pinky_mcp.x - index_mcp.x
    dy = pinky_mcp.y - index_mcp.y

    angle_rad = math.atan2(dy, dx)
    angle_deg = math.degrees(angle_rad)

    tilt_angle = max(-45, min(45, angle_deg))
    return tilt_angle
```

Listing 16: Hand tilt calculation using MediaPipe landmarks

The tilt detection algorithm uses the metacarpophalangeal joints of the index and pinky fingers to establish the hand's orientation vector. This approach provides stable steering control that responds naturally to hand rotation while remaining insensitive to individual finger movements.

5) Multi-Player Zone Detection: The system implements screen-based zone detection to enable simultaneous two-player operation:

```
def run(self):
    while cap.isOpened():
        ret, frame = cap.read()
        frame = cv2.flip(frame, 1)  # Mirror for natural control
        h, w, _ = frame.shape

        # Draw center dividing line
        cv2.line(frame, (w // 2, 0), (w // 2, h), (255, 255, 255), 2)

        # Process hands and determine player zones
        if results.multi_hand_landmarks:
            for hand_landmarks in results.multi_hand_landmarks:
                wrist = hand_landmarks.landmark[self.mp_hands.HandLandmark.WRIST]
                hand_x = wrist.x * w

                if hand_x < w / 2:  # Left side - Car 1
                    self.left_hand_detected = True
                    new_car1_speed = self.calculate_hand_openness(hand_landmarks) * 100
                    new_car1_angle = self.calculate_hand_tilt(hand_landmarks)
                else:  # Right side - Car 2
                    self.right_hand_detected = True
                    new_car2_speed = self.calculate_hand_openness(hand_landmarks) * 100
                    new_car2_angle = self.calculate_hand_tilt(hand_landmarks)
```

Listing 17: Main run loop: processing hand input and assigning controls to Car 1 and Car 2

The zone-based detection system enables independent control of two vehicles by spatially separating the control areas. The mirrored display provides intuitive control mapping where users see their reflected image, making the control interface more natural and reducing learning time.

6) Command Smoothing and Filtering: To prevent erratic vehicle behavior from minor hand movements, the system implements control signal smoothing:

```
alpha = 0.7  # Smoothing factor

# Update Car 1 controls with smoothing
if self.left_hand_detected:
    self.car1_speed = alpha * new_car1_speed + (1 - alpha) * self.car1_speed
    self.car1_angle = alpha * new_car1_angle + (1 - alpha) * self.car1_angle
else:
    self.car1_speed = 0
    self.car1_angle = 0
```

Listing 18: Car 1 control smoothing with exponential moving average

The exponential smoothing filter reduces control jitter while maintaining responsive control. The smoothing factor provides a balance between stability and responsiveness, with higher values favoring immediate response to gesture changes.

7) Command Transmission Protocol: The command transmission system formats and sends control data to the Arduino server:

```
def send_commands_to_arduino(self):
    if self.serial_conn:
        command = f"C1S{int(self.car1_speed)}
        A{int(self.car1_angle)}
        C2S{int(self.car2_speed)}
        A{int(self.car2_angle)}\n"
        try:
            ↩
    ↪ self.serial_conn.write(command.encode())
            print(f"Sent: {command.strip()}")
        except Exception as e:
            print(f"Failed to send command: {e}")
    else:
        print(
    f"Demo: Car1[S:{int(self.car1_speed)}% ↩
    ↪ A:{int(self.car1_angle)}] "
    f"Car2[S:{int(self.car2_speed)}% ↩
    ↪ A:{int(self.car2_angle)}]"
)
```

Listing 19: Command transmission: format and send control data to Arduino

The command protocol implements a structured format that the Arduino server can reliably parse. The error handling ensures continued operation even if serial communication fails, while the demo mode enables development and testing without hardware connectivity.

8) Real-Time Visual Feedback System: The system provides comprehensive visual feedback to users, displaying control states and system status:

```
def draw_control_info(self, image, car1_speed, ↩
    ↪ car1_angle, car2_speed, car2_angle):
    h, w, _ = image.shape

    # Speed bar visualization
    bar1_length = int(150 * (car1_speed / 100))
    cv2.rectangle(image, (10, 70), (160, 90), ↩
    ↪ (100, 100, 100), 2)
    cv2.rectangle(image, (10, 70), (10 + ↩
    ↪ bar1_length, 90), (0, 255, 0), -1)

    # Steering angle visualization
    center1_x, center1_y = 85, 150
    radius = 30
    cv2.circle(image, (center1_x, center1_y), ↩
    ↪ radius, (100, 100, 100), 2)
    angle1_rad = math.radians(-car1_angle)
    end1_x = int(center1_x + radius * ↩
    ↪ math.cos(angle1_rad))
    end1_y = int(center1_y + radius * ↩
    ↪ math.sin(angle1_rad))
    cv2.line(image, (center1_x, center1_y), ↩
    ↪ (end1_x, end1_y), (0, 0, 255), 3)
```

Listing 20: Visual feedback overlay: speed bar and steering angle indicator

The visual feedback system provides intuitive representations of vehicle control states, including speed bars and steering angle indicators. This real-time feedback enables users to understand and adjust their gestures for precise vehicle control.

E. System Integration and Communication Flow

The complete software system operates through a coordinated communication flow that spans all three software components. The Python controller captures and processes hand gestures at 30 frames per second, generating control commands that are transmitted via serial communication to the Arduino Uno server at approximately 10 Hz. The server immediately parses and redistributes these commands to the appropriate vehicle receivers via wireless transmission, achieving end-to-end control latency of less than 100 milliseconds.

The distributed architecture provides robust operation through multiple redundant safety systems, including timeouts at both the server and vehicle levels, command validation at each processing stage, and graceful degradation in case of component failures. This comprehensive approach ensures safe and reliable operation across varying environmental conditions and usage scenarios.

1) Data Structure Definition: To maintain consistency across all system components, a standardized data structure encapsulates the control commands transmitted between the server and car receivers:

```
struct DataPacket {
  int speed;  // 0-100 percentage
  int angle;  // -45 to +45 degrees
};
DataPacket dataToSend;
```

Listing 21: Data packet structure for speed and steering angle

This compact data structure minimizes transmission overhead while providing all necessary control parameters. The speed parameter utilizes percentage-based scaling for intuitive control mapping, while the angle parameter corresponds directly to steering angle limits.

2) Arduino Uno Server Communication: The Arduino Uno server processes incoming serial commands from the Python controller and distributes them to the appropriate car receivers. The command parsing system interprets formatted strings containing control data for multiple vehicles:

```
void processCommand() {
  int c1Index = inputString.indexOf("C1");
  int c2Index = inputString.indexOf("C2");

  if (c1Index != -1) {
    int speedIndex = inputString.indexOf('S', ↩
    ↪ c1Index);
    int angleIndex = inputString.indexOf('A', ↩
    ↪ c1Index);

    car1Speed = ↩
    ↪ inputString.substring(speedIndex + 1, ↩
    ↪ angleIndex).toInt();
    car1Angle = ↩
    ↪ inputString.substring(angleIndex + 1, ↩
    ↪ c2Index).toInt();

    sendToCar1(car1Speed, car1Angle);
  }
}
```

Listing 22: Parsing Car 1 speed and angle from Serial command

The command parsing algorithm identifies vehicle-specific commands within the input string and extracts the corresponding speed and angle values. This approach allows simultaneous control of multiple vehicles through a single communication channel.

3) Wireless Transmission Protocol: The wireless transmission system switches between different car addresses to enable targeted communication with specific vehicles:

```
void sendToCar1(int speed, int angle) {
  radio.openWritingPipe(address1);

  dataToSend.speed = speed;
  dataToSend.angle = angle;

  bool result = radio.write(&dataToSend, ↩
    ↪ sizeof(dataToSend));
  Serial.print(result ? " [Car1:OK]" : " ↩
    ↪ [Car1:FAIL]");
}
```

Listing 23: Sending speed and angle commands to Car 1 via nRF24L01

This transmission function demonstrates the address-switching mechanism that enables selective communication with individual cars. The return value verification provides immediate feedback on transmission success, enabling system-level error handling and diagnostics.

F. Arduino Nano Car Receiver Implementation

The Arduino Nano receivers implement the car-side communication protocol, processing incoming wireless commands and translating them into motor control signals.

1) Wireless Data Reception: The reception system continuously monitors for incoming wireless data and processes commands as they arrive:

```
void loop() {
  if (radio.available()) {
    radio.read(&receivedData, ↩
    ↪ sizeof(receivedData));

    currentSpeed = receivedData.speed;
    currentAngle = receivedData.angle;

    controlCar(currentSpeed, currentAngle);
    lastDataTime = millis();
  }
}
```

Listing 24: Receiving control packets and updating car movement

The reception loop implements non-blocking communication, allowing the system to respond immediately to new commands while maintaining other operational functions. The timestamp tracking enables safety timeout mechanisms to prevent runaway vehicle operation.

2) Motor Control Algorithm: The motor control system translates speed and steering commands into differential motor speeds, enabling smooth turning behavior:

```
void controlCar(int speed, int angle) {
  int baseSpeed = map(speed, 0, 100, 0, 255);
  float turnRatio = abs(angle) / 45.0;

  int leftSpeed = baseSpeed;
  int rightSpeed = baseSpeed;

  if (angle < 0) {
    leftSpeed = baseSpeed * (1 - turnRatio * ↩
    ↪ 0.6);
  } else if (angle > 0) {
    rightSpeed = baseSpeed * (1 - turnRatio * ↩
    ↪ 0.6);
  }
}
```

Listing 25: Car control logic with speed mapping and differential steering

This differential control algorithm reduces the speed of the inner wheel during turns, creating natural steering behavior. The turn ratio calculation ensures proportional response across the full steering range, providing precise control for both subtle adjustments and sharp turns.

3) Safety Timeout Implementation: A critical safety feature prevents vehicles from continuing to operate if communication is lost:

```
if (millis() - lastDataTime > TIMEOUT && ↩
    ↪ currentSpeed > 0) {
  stopCar();
  digitalWrite(LED_PIN, LOW);
  Serial.println("Signal lost - Emergency ↩
    ↪ stop!");
}
```

Listing 26: Safety timeout check triggering emergency stop on signal loss

This timeout mechanism automatically stops the vehicle if no commands are received within the specified timeout period. This safety feature prevents potential damage or loss of control in the event of communication system failures.

G. System Integration and Testing

The complete software system integrates all components into a cohesive wireless control platform. The modular design enables independent testing of each subsystem while maintaining compatibility with the overall architecture. Initial testing focused on verifying reliable communication between all system components before proceeding to implement the computer vision control interface.

The wireless communication system successfully demonstrated reliable operation with single and dual vehicle configurations, providing the foundation for implementing advanced gesture-based control algorithms in the subsequent development phases.

V. Conclusion

All in all, this project successfully demonstrates a multi-vehicle RC car system lying at the cross-section connecting embedded systems, wireless communication, and computer vision. With respect to the limitations of conventional physical controllers, the DAT Racer's unique approach to utilize hand

gesture recognition software facilitates user-to-car communication and encourages an intuitive control interface.

During the development, there were some setbacks concerning the functionality in both hardware and software, such as inconsistent power deliveries, unreliable wiring, communication errors, and more. However, through iterative design and testing, these hurdles were systematically resolved through lithium battery upgrades, welded connections, and extensive debugging.

The end product uses nRF24L01 wireless modules paired with Google's MediaPipe framework to guarantee real-time hand-gesture-recognition-based vehicle control. This joint synergy, along with additional software modifications, such as low-latency communication and implemented timeout mechanisms, further enable precise steering and smooth speed modulation for both vehicles.

With the current model acting as a stable benchmark, future work may further explore the extent and precision of the hand-gesture controls. Currently, the control lies proportional to the capabilities of the vehicle - rudimentary and covers only the core functionalities. Equipping the car with multiple motors and allowing the user to use both hands to control it would impose new levels of complexity but also innovation.

REFERENCES

[1] M. Turk and M. Kölsch, "Perceptual interfaces," *Communications of the ACM*, vol. 46, no. 7, pp. 33–34, 2003.

[2] Z. Zhang et al., "Hand gesture recognition using vision-based methods: A review," *International Journal of Computer Vision*, 2019.

[3] X. Chen, W. Li, and Y. Zhao, "Hand gesture-based control of a robot car using computer vision," in *Proc. IEEE Int. Conf. Robotics*, 2018, pp. 112–118.

[4] C. Lugaresi et al., "MediaPipe: A framework for building perception pipelines," *arXiv preprint arXiv:1906.08172*, 2019.

[5] Arduino, "Arduino Uno Rev3," Available: https://docs.arduino.cc/hardware/uno-rev3. [Accessed: 31-Aug-2025].

[6] Nordic Semiconductor, "nRF24L01 Single Chip 2.4GHz Transceiver Product Specification," 2007. [Online]. Available: https://infocenter.nordicsemi.com/.

[7] Arduino, "Arduino Nano," Available: https://docs.arduino.cc/hardware/nano. [Accessed: 31-Aug-2025].

[8] J. Zimmerman, J. Forlizzi, and S. Evenson, "Research through design as a method for interaction design research in HCI," *Proc. SIGCHI Conf. Human Factors in Computing Systems*, 2007, pp. 493–502.

[9] G. Plett, "Battery management systems, Part I: Battery modeling," *Journal of Power Sources*, vol. 134, no. 2, pp. 252–261, 2004.

[10] M. Armand and J. M. Tarascon, "Building better batteries," *Nature*, vol. 451, pp. 652–657, 2008.

[11] RAM Electronics, "Electronics Components Supplier," [Online]. Available: https://www.ram.co.za/. [Accessed: 31-Aug-2025].

[12] STMicroelectronics, "L298 Dual Full-Bridge Driver Datasheet," 2000. [Online]. Available: https://www.st.com/en/motor-drivers/l298.html. [Accessed: 31-Aug-2025].

[13] Texas Instruments, "Understanding H-Bridge Motor Driver Circuits," Application Report, 2013. [Online]. Available: https://www.ti.com/lit/an/slva959/slva959.pdf. [Accessed: 31-Aug-2025].

[14] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*. Wiley, 2002.

[15] STMicroelectronics, "AN297: L298 Motor Driver Performance Considerations," Application Note, 2001. [Online]. Available: https://www.st.com. [Accessed: 31-Aug-2025].

[16] J. Axelson, *Embedded Systems: Hardware, Design, and Implementation*. Annabooks, 1997.