Electric Vehicle Simulation

Jenna Bunescu

1. Introduction

Engineering teams that work on developing electric vehicles heavily rely on simulation software as a precursor to physical prototyping. Simulations optimize the development process by reducing both time and cost, as engineers can use them to predict the behavior of components like batteries and motors under various conditions. MATLAB and Simulink are a popular choice for this purpose, but they often come with a steep learning curve, so they are not beginner-friendly.

The goal of this project was to develop a simplified electric vehicle simulation that lowers the barrier to entry for modeling vehicle performance. Instead of requiring the user to build complex code and diagrams or write equations manually, the simulation only asks for surface-level parameters such as battery capacity, internal resistance, and wheel radius. Then, the program estimates battery charge consumption and speed and temperature changes. The simulation also takes into account charging and regenerative braking.

This kind of simulation tool could be valuable for early-stage vehicle design, and it could also be used in classrooms or by hobbyists who want to explore the fundamentals of electric vehicle behavior without needing access to expensive software.

This document will describe the project, the assumptions made while designing it, and a detailed analysis of the program's functionality. The user's manual is in a different document, along with the UML diagrams.

2. Project Description

This project uses Object-Oriented Programming to help different parts of the simulation (the battery, motor, and driver input) work together in an organized and efficient way. The final product is a simple but functional simulation framework that models electric vehicle behavior based on user-defined settings. Users can adjust things like wheel radius, internal resistance, maximum speed, and more. Each of these choices affects how the car behaves, and by trying out different setups, users can get a better feel for how those variables interact.

The simulation includes a graphical user interface that updates values like speed, temperature, and battery state of charge in real time. There's also a simple animation of the car moving to help visualize what's happening. At any point, users can start a new session with either the default values or custom settings for a different vehicle setup.

When a session ends, the program creates a graph showing how the battery's charge changed over time. This gives a helpful visual summary of energy use and any energy recovered through regenerative braking. Additionally, the average speed over 100 samples is displayed in the terminal at the end of each session.

More technical details and how I built everything are explained in the Detailed Analysis section.

3. Assumptions

To maintain simplicity and usability, the following assumptions were made in both the design and implementation of the project.

- **Simplified physics:** If this simulation were to follow realistic physics, the project would have consisted of much more research than actual coding, as the physics of batteries and motors is quite difficult and beyond my understanding of mechanical, chemical, and electrical engineering. In the program, the battery discharges linearly with speed, which is not accurate in real life.

- **Constant parameters:** Some parameters, like the drag coefficient, ambient temperature, the heat transfer coefficient of the battery, the car's inertia, and the motor's regeneration efficiency, are assumed constant. I chose to keep these parameters the same across all simulation runs because implementing them would have required more time working on the project.

- **1D motion:** The vehicle moves in a straight line only, with no turning or lateral behavior. It also cannot go backward, it can only go forward. The MVP (Minimum Viable Product) for this project does not need to include backward movement, especially because it could be implemented pretty easily in the future.

- **Collision detection:** There is no collision detection and the car can drive perpetually without hitting anything. This was a deliberate simplification to focus on the electrical behavior rather than the mechanical behavior of the car. However, it would be an interesting idea to implement crash tests in the future.

- **Single-motor drive:** The model assumes propulsion from one motor; no hybrid or multi-motor configurations are considered.

- **Ideal conditions:** Environmental factors like wind, temperature, and weather are excluded. The environment is static, as seen by the fact that the drag coefficient of the road is constant, as is the temperature. Realistically, there would be temperature and terrain changes. The user can still go into the code and change the values of those variables directly, but it would be nice to implement a more variable environment in the future.

- **Discrete time system:** This program operates in discrete time rather than continuously; it updates the code at a fixed interval of time every fraction of a second. It could be argued that any computer is not continuous, but it is likely that some software can mimic continuity by updating at smaller time intervals. Since this program is not continuous, this can affect the accuracy of the results and the behavior of the car's attributes, as real-life systems are continuous.

- **Binary driver inputs:** Throttle and brake are either 0 or 1. It would have been difficult to estimate a weighted input just based on the press of a key, as the key is either being pressed or not being pressed. Some ideas included typing in the throttle and brake values or using the numbers on the keyboard, but I thought that would have been messy and unintuitive.

- **User input:** User is aware that pressing the 'W' key makes the car go forward, the 'S' key brakes the car (and regenerates the battery), and the 'C' key charges the car. The car can only charge when it is stationary, so the user needs to be aware that they must stop the car and then press 'C'.

- **Units:** Inputs use SI units. I think there is a chance that some functions use different units (like kilometers instead of meters, or seconds instead of hours) for certain attributes, since I didn't work on all the functions at the same time, and I didn't double-check for consistency, as that could have taken a long time to do. I would need to go back and check unit calculations for that, which I could do over the summer if I plan on improving this simulation.

- **State persistence:** The simulation does not save any data between runs. Once the program is closed, all data, including car position, speed, and SOC, is reset. Long-term performance evaluation or multi-session testing is not currently supported.

- **Platform dependency:** The program was developed and tested on MacOS and Windows 11. Behavior or performance on other operating systems has not been thoroughly tested and may vary depending on SFML support.

- **No full screen:** The graphics are preset on a 1280x720 screen, and there are no accurate scaling options to go from one screen size to a different size or full screen. When the user tries to go full screen, the road graphics include a blue banner, which is not supposed to be there.

4. Detailed Analysis

For the user interface, I chose SFML (Simple and Fast Multimedia Library) because I felt that having a visual element was important, and SFML is one of the easier libraries to use. Since the goal is to make the simulation intuitive and insightful, showing things visually really helps.

Once the graphics are running, the user can press the 'W' key for throttle and the 'S' key for braking. These two keys were chosen as they are widely used in video games for movement. Additionally, when the 'S' key is pressed and braking is applied, the program also applies regenerative braking to the car and re-charges the battery (however, a little slower than the charger itself). The 'C' key can also be pressed to charge the battery when the car is stationary. If the 'C' key is pressed when the car is moving, nothing will happen. I would have liked to use an interactive on-screen button for this, but I thought a key press would be simpler and sufficient. When no keys are pressed, if the car is moving, it will start decelerating based on drag.

In the blue display box, on the left, the battery's state of charge is displayed in percent, as well as the speed in m/s, and the temperature in degrees Celsius. Looking back, it would have made more sense to make speed in km/h or mi/h, as that is what car speeds are usually listed in, but the formulas I used for speed were initially in meters per second, and I never changed them.

Additionally, an alert is displayed on the screen, in red bold text, when the battery is fully charged (Battery Full!) and when its state of charge is below 20 (Battery Low!).

The first run starts with default values, but the user can choose to turn off the run by pressing the rectangular button that has "EV ON" written on it. Then, the user should not press anything on the screen after pressing it, and instead go back to the terminal to input their values. I am unsure why, but sometimes the window can crash if the user tries to interact with the screen while the terminal is prompting them for input. In the terminal, the user can either input a non-negative value for their new parameters or use '-1' to set the parameter to the default setting. I implemented error handling for this as I thought it would make things run smoother and prevent the program from crashing. I functionalized the error handling because many parameters can be changed. I also initially wanted to ask for these parameters directly on the window, and have a textbox that the user can type into, this proved to be very difficult to implement with the limited time I had, and I spent a lot of time trying to debug it, and eventually decided it would be more time-efficient to leave this functionality to the terminal.

When the terminal displays that a new EV was created, the user can go back to the window. The window will now be black, and the button in the top left corner will have "EV OFF" written on it. The user can press the button again to start the simulation of the new vehicle with the parameters they chose. Once they close the window, the terminal will output one last thing: the average speed over 100 samples between 1-second time increments. If there are not enough data points to meet those requirements, an average speed will still be calculated, with the same interval but with fewer samples. The sample count and time interval can be changed in main.cpp to include more samples over smaller time increments to provide a more accurate average speed calculation. The function averageSpeed(int sampleCount, float sampleInterval) parses the output.csv file and uses the keyword "new" to store the data encountered. This is where I used dynamic memory in the code. I wanted to use dynamic memory to create new objects, but I kept running into issues, so I dropped that idea.

Lastly, after the program ends, the user may opt to run graph.py, to generate a graph of the output.csv file data to visualize how the battery's state of charge and the car's speed changed over time throughout the session.

4.0.1 Web references

I learned how to use basic SFML graphics using the official website's guide. I found that YouTube tutorials were not as helpful as the official website, as I could more easily find exactly what I was looking for:
https://www.sfml-dev.org/tutorials/2.6/.

Other resources:
https://cplusplus.com/reference/string/string/substr/, https://cplusplus.com/reference/string/string/find/, https://en.cppreference.com/w/cpp/thread/sleep_for.

For the Physics, I used some simple formulas from the General Physics I class. Other scientific details that I didn't know about were one Google search away.