HEV Simulation Program

Kenny Leung

As Part of the MEC E 509 Project Course

Summary

This report describes a simulation program written in the MEC E 509 project course for use in the University of Alberta Hybrid Electric Vehicle Project.

The data from the simulations will be valuable in selecting components for the vehicle, in deciding on a test program, and later in programming control strategies for operating the vehicle. It would be impractical, and in some cases, impossible to run a large battery of tests on the actual prototype. Different components can be tried out in the program as long as their attributes are known; many load cases can be simulated, and only the critical points considered in testing; new control programs can be immediately tested, and only reasonable ones can be used in the actual vehicle.

This project also has the secondary purpose of showing the benefits of using object oriented programming techniques in mechanical simulations. Programming objects remember their state, and are used as "black boxes", much as systems such as transmissions or batteries are used. The user only worries about the inputs and outputs of such objects, and usually does not care about what is inside making the object work. Since the NeXT computer is currently the only computer featuring a fully object oriented operating system, it was chosen for the implementation of this simulation.

Even though the user does not need to worry about the inner workings of the objects in the program, they must still be detailed enough inside to provide an accurate simulation. For the purposes of this project, the faithfulness of simulation was limited to the major component level. For example, in the transmission, only a set of gear ratios are kept as opposed to having a set of gears with different tooth numbers.

In order to provide an accurate simulation, the program must include all major components: the car, controller, engine, motor, batteries, transmission, wheels, and gas tank.

These requirements have been met by the program. The program includes the necessary components, and takes into account air resistance, rolling resistance, and road grade in the simulation. The parameters of each of the components can be set interactively in an easy to use graphical interface. It also takes into account efficiencies of the various components.

HEV Simulation Program

Kenny Leung
As Part of the MEC E 509 Project Course

1.0 Introduction

The purpose of writing a simulation program is to perform tests on a computer that would be impractical or impossible to do in real life. For the University of Alberta Hybrid Electric Vehicle Project, a simulation program would be valuable in component selection, in development of a testing program, and in development of control strategies.

The requirements of such a program are: it must simulate all major components; it must allow for the different load cycles, including grade; and it must simulate both an ideal and a limited car.

The simulation program contains all major components including the car, engine, motor, batteries, and transmission. Each component is implemented as an object under the object oriented programming paradigm. A programming object is well suited in mechanical simulation because they can be made to behave as a real object does, without the programmer having to worry about the details inside them.

The next section, Justification and Requirements, detail specific purposes that the simulation program will be used under, and what is required of a simulation program.

Section 3.0, Program Overview, describes the program features, and the design and implementation, including the reasons for choosing to use objected programming techniques, the NeXT computer, and Objective-C as the platform for implementing the simulation. It includes an example of the simulation code.

Some output examples are presented in Section 5, and complete operating instructions are included in the appendix.

2.0 Justification and Requirements

2.1 The need for a Simulation Program

In the course of building and testing a hybrid electric vehicle, it would be inconvenient or impossible to perform all of the necessary tests on the vehicle itself. Running tests on a scale model would be impractical because of the time and effort required to build one. Also properly instrumenting either the prototype or a model for these tests might be expensive as well as time consuming. Thus, the obvious choice is to write a simulation program which will allow tests in arbitrary cycles with speed and efficiency. The three areas where the simulation would have the most impact are in initial selection of components, development of a test program, development of control strategies.

2.1.1 In Component Selection

The problem of component selection usually revolves around the size or capacity of a component. For example, which engine has the right amount of power, or how many batteries should we use? The simulation program can be used to determine the minimum power required to run the vehicle through a typical cycle, and thus help in the selection of power units. The program can also calculate the energy required in a given cycle for a particular car, and thus give an estimate for the battery capacity required.

Determining the proper transmission gear ratios and final drive ratio to match desired performance is difficult at best. Using a simulation lets the designer try out any combination of gear ratios to find the right set.

2.1.2 In Developing a Test Program

Since a computer can run a simulation in a fraction of the time it takes to run a real test, many scenarios may be tested in a short time. Then, only the critical points may be selected, and actual tests run at that point. The data from the tests may then be compared to the simulation data to determine its accuracy. The test data may also be put back into the simulation to gain a more accurate simulation which may provide insight into other points to test.

2.1.3 In Development of Control Strategies

In order to try out a control strategy on the car, one would have to write the control program, download it to the controller, and then drive the car. Again this can be very costly in both money and time. Developing the control strategy on the computer simulation allows it to be run and fine tuned immediately. Only after the appropriate strategy is found would actual coding for the controller start, and that code could be ported from the controller section in the simulation.

Some control strategies that might be tested for are:

- optimum shift points for the transmission

- optimum speeds and load situations for shifting from electric to internal combustion drive, or vice versa
- the costs of keeping an electric drive spinning at all times versus stopping it and starting it when needed
- the stopping conditions under which caliper brakes would have to be applied in conjunction with regenerative braking in order to obtain the best compromise between safety and energy recovery

2.2 What is Required of the Simulation Program

The program must contain all the major components critical for performance. The parameters for each component must be easily changeable in an independent manner to simulate the replacement of real components in a real car. The simulation of components should be performed in appropriate detail to provide an accurate simulation. Also, the mathematical modelling of the various physical processes must be appropriate to the situation.

The program should be able to independently load arbitrary test cycles. For example, one might want to run the same vehicle through EPA city and highway cycles. The user should be able to set initial conditions for a particular run so that conditions such as passing maneouvers can be easily simulated.

The program should simulate both an ideal car and a limited car to provide a reference point for the performance of the car.

The program should have an easy to use interface to facilitate its use, as many cycles are likely to be run.

The program should be easily extensible, both in altering existing components, and in adding components.

Output should consist of graphs of speed, power, acceleration, and energy used in the cycle.

3.0 Program Overview

3.1 Program Features

The simulation is designed to be easy to use, with direct, asynchronous entry of data, so one may input only the necessary information and then run the simulation. The program conforms to the user interface standards of the NeXTstep operating system.

The program provides full simulation of all of the major components in a hybrid electrical vehicle, including engine, motor, batteries, and transmission.

3.1.1 Independent Loading of Simulation Attributes

Three attributes of a simulation can be independently loaded to provide for flexibility in running tests: cars, cycles, and environments. For instance, one might have a single car definition and want to run it on different cycles with different road conditions. Having independent modules for each of these attributes prevents the duplication of attributes and allows for greater ease of use.

3.1.2 Ease of Use

The simulation program can be launched either by double-clicking on its icon, or by double-clicking on one of its files. Also, files may be either from within the program, or by double-clicking them in the Workspace. A window with the various simulation components is presented, and an inspector is provided for each component to change its attributes. Clicking on a component will automatically bring up its inspector.

3.1.3 Simulation Detail

The level of detail to which the vehicle is simulated is only down to the major component level. For example, the engine is represented only by a speed-torque curve, mass, and efficiency. Smaller components such as cylinders, spark plugs, and a fuel injection system are ignored, although they could be integrated into the existing program if desired. The degree of simulation of each component is described in detail in Section 3.3.4.1.

3.2 Design and Implementation

In order to promote design efficiency and maintainability, the simulation is written in Objective-C, a version of the C programming language with object oriented extensions. Objective-C is only used with the NeXTstep operating system.

Object oriented programming is a technique in which data are encapsulated with sections of code that operate on those data only. The only way to gain access to change the data is through *messages* which are sent to the object. Because of the restrictions of this structure, objects within an object oriented program can be made to behave as objects would in real life.

An analogy is often made between program objects and people. A person who wants a service will make a request to another person. In the same way, an object which needs an action performed requests this action from another object. The sender of the request does not know, care, or have control over what the receiver of the request does to carry out its function. The only information that is available to the sender is an answer to the request.

This type of functionality makes object oriented programming ideal for mechanical simulations. Different components of a machine can be represented by objects, which only interact through a single interface. For example, in this program the transmission acts as a black box with the input attached to the engine and the output attached to the wheel shaft. All the engine has to do is deliver power to the input of the transmission. The transmission object handles conversion of torque and shifting of gears in a fashion which is unknown to the engine.

Another advantage is that, just as mechanical components can be replaced in the real world, an object can be replaced in an object oriented program as long as it has the same interface as the original object. This makes it extremely easy to try out different components without affecting the integrity of the rest of the program. For instance, one might replace an electric motor object with one that supports regenerative braking. In a well-designed object oriented program, it would be possible to replace the electric motor without making *any* changes to the rest of the code.

3.3 Background in Object Oriented Programming

In order to understand the structure of the simulation of the program, it is first necessary to understand the basics of object oriented programming.

3.3.1 Classes and Instances

The distinction is made between object *classes* and *instances* of those classes. A class is the definition of an object whereas instances are the actual objects based on those definitions. A class can be thought of as a cookie cutter, and instances as cookies. Instances are differentiated from each other by the values in their *instance variables*. This is usually a difficult concept to grasp, so we'll take an example.

Let's say that we want to programmatically define several gears, all with different radii and numbers of teeth. Since gears all act in the same fashion except for their radii and tooth count, we will define a Gear class with `radius` and `tooth_count` as instance variables. Then, to define all of the different gears, one simply has to ask for new instances of Gear, and set their `radius` and `tooth_count` to the different values.

3.3.2 Methods

As mentioned in Section 3.2, the only effect an object may have on another object is to send it *messages* which invoke its *methods*. A clear distinction is made between the two because the same message sent to different objects may invoke different methods. So a method is the implementation of an action while a message is the invocation of that action.

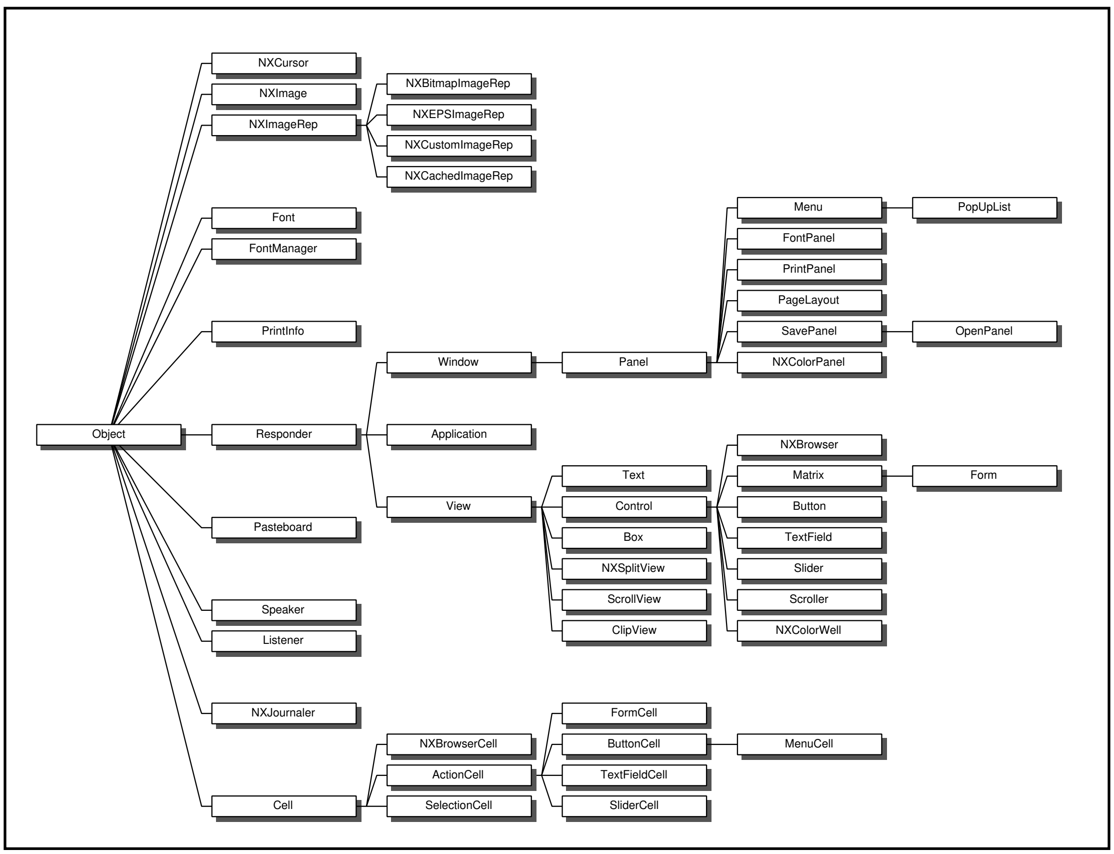
3.3.3 Inheritance and the Object Hierarchy

Classes in object oriented programming build upon each other. New classes are created by *subclassing* from old classes. The new class inherits all of the methods and instance variables of its *parent* or *superclass*, and extensions are made by adding new methods or instance variables, or *overriding* old methods.

Lets say, for example, that we have an object for a four cylinder engine with all of its ancillary parts. In order to create an object for an eight cylinder engine, we subclass the four cylinder engine. Then, we add four new instance variables with which to reference the extra cylinders. We also have to override old methods which affect cylinders to include all eight.

Creating subclasses of subclasses leads to a class hierarchy, with some root class at the top. Below is an example of an object hierarchy, the set of Application Kit objects included with the NeXTstep operating system.

FIGURE 1. The NeXTstep Application Kit Class Hierarchy



3.3.4 The Structure of the HEV Simulation Program

The entire vehicle simulation is composed of ten objects representing the car components, a test cycle, and a road environment. Three other objects act as support for file handling, data entry, and data display.

The parameters described for each of the objects below are all changeable inside the program interface. Therefore, they are the portions of the vehicles that can be readily tested. In order to test more complex ideas such as control strategies and braking, the code will have to be changed directly. Where applicable, suggestions for such changes are presented.

NOTE: If a parameter for a particular component is not described below, it should be assumed that it is not included in the simulation, and will require a code change to implement.

3.3.4.1 The Objects

• AppDelegate

This object acts as a delegate for the Application Kit's Application object. That is, the application object passes some of its functions on to this object. It handles opening and closing files and initializing defaults.

• Battery

The battery accepts requests for power from the electric motor and delivers output based on its own parameters and current charge level.

The parameters that the battery uses in the simulation are voltage-charge and current-charge curves, peak current draw, and efficiency. The same efficiency is used for both charging and discharging.

• Car

The Car object is the one which initiates the actual simulation.

The parameters that it simulates are mass, frontal area, and drag coefficient. Using these parameters and information from the cycle, and wheels, the car calculates aerodynamic drag, rolling resistance, inertial acceleration, and grade acceleration.

• Controller

The controller accepts requests for power from the car during the simulation, and forwards those requests to either the electric motor or internal combustion engine based on its control strategy.

The controller currently only supports a shift ceiling and a shift floor for switching between the two power units at different speeds. Different control strategies based on load, or running both power units in conjunction requires changing the code manually.

• Cycle

The cycle contains a time-speed map which the car tries to match during a simulation. It keeps track of the current time in the simulation and parcels out the proper speed and acceleration to the car as needed.

• DataView

This is an auto-ranging graphing object which is used throughout the program to display data.

• Engine

The engine contains a speed-torque curve and information on stall speed and redline. It receives power requests from the controller and delivers output to the transmission.

• GasTank

The gas tank receives energy requests from the engine, and adjusts its level automatically. The amount of fuel used is based on the energy density and mass density characteristics entered for the fuel.

• Inspector

This is the largest object in the program, but its function is secondary. It passes data between the user interface objects and the internal components of the simulation.

• Motor

The electric motor contains two speed-torque curves, one for low speeds, and one for high speeds. It receives power requests from the controller and delivers either the power requested or the maximum power available to the transmission. It automatically adjusts to each curve according to its rotational speed. Since there are no brakes in this simulation, the electric motor delivers as much negative power as necessary.

• Road

The road (or environment) object contains information on wind speed and temperature. It also contains a slope map which can be used as a grade component in a simulation cycle. The road keeps track of the cars current position on it, and automatically returns the grade for the proper position.

Currently, the wind speed stays constant throughout the cycle. The wind speed may be made variable through a code change.

• Transmission

The transmission takes torque inputs from the internal combustion and electric motor, multiplies them by the current gear and final drive ratio, and sends torque information onto the wheels. The transmission contains information on five gear ratios, shift points between gears, and the current gear. It also simulates a condition of zero power output when shifting gears.

• Wheels

The wheels contain information on their diameter and rolling resistance. They convert input torque to road force, which the car uses to adjust its current velocity.

3.3.4.2 An Example of Simulation Code

As explained earlier in this section, component parameters are encapsulated within objects along with the code that affects them. Therefore, changing any part of the simulation only requires changing the method of the object that performs the simulation function. The following piece of code is the powerRequired: method that is sent to from the car to the controller in requesting power during a single step in the cycle. A line-by line explanation of the code follows.

Programmers familiar with C will recognize the syntax except for the message syntax, which is composed of the receiver of the message followed by the action, with both enclosed by square brackets.

FIGURE 2. Example Code From Controller Object

```
- powerRequired:(float)power
{
float enginePower;
float motorPower;

    if ( [car currentVelocity] * 3.6 > shiftCeiling )
        runningMode = ENGINE;
    if ( [car currentVelocity] * 3.6 < shiftFloor )
        runningMode = MOTOR;

    if ( runningMode == MOTOR )
        {
        motorPower = power;
        enginePower = 0;
        }
    if ( runningMode == ENGINE )
        {
        motorPower = 0;
        enginePower = power;
        }
    if ( runningMode == BOTH )
        {
        // Not implemented yet.
        }
    if ( power < 0 )
        {
        motorPower = power;
        enginePower = 0;
        }
    [engine powerRequired:enginePower];
    [motor powerRequired:motorPower];

    return self;
}
```

```
- powerRequired:(float)power
{
```

This is the method declaration. It states that there is one argument to the method, a floating point number called power. Methods must be enclosed with braces.

```
float enginePower;
float motorPower;
```

The declaration of two local floating point variables, the power that will be asked of each drive unit.

```
    if ( [car currentVelocity] * 3.6 > shiftCeiling )
```

```
            runningMode = ENGINE;
```

If the car's velocity is beyond the shift ceiling, then change the mode to be the internal combustion engine only. The factor of 3.6 is just a conversion from m/s(car) to km/h(controller parameter). runningMode is an instance variable for the controller, and ENGINE is a previously defined constant.

```
        if ( [car currentVelocity] * 3.6 < shiftFloor )
            runningMode = MOTOR;
```

If the car's velocity is less than the shift floor, then change into electric motor only mode.

```
        if ( runningMode == MOTOR )
            {
            motorPower = power;
            enginePower = 0;
            }
```

Here is where we actually determine which drive unit is requested for the power. All it says is that, if the running mode is motor only, then all the power should come from the electric motor.

```
        if ( runningMode == ENGINE )
            {
            motorPower = 0;
            enginePower = power;
            }
```

This is the case if the mode is engine only. It says that all of the power should come from the engine.

```
        if ( runningMode == BOTH )
            {
            // Not implemented yet.
            }
```

This is a section we could add in if we wanted to have a mode where both are running. Of course, we would have to make sure that the controller switched into this mode at some point using code similar to section above which depended on the vehicle speed. One could also take the amount of power asked for into account.

```
        if ( power < 0 )
            {
            motorPower = power;
            enginePower = 0;
            }
```

This is what we do to implement braking and regenerative braking. If the power asked for is negative, then the motor gets the power request no matter what mode the controller is in.

```
        [engine powerRequired:enginePower];
        [motor powerRequired:motorPower];
```

These are the actual messages which are sent to the engine and the motor requesting power. They continue the message chain in the simulation.

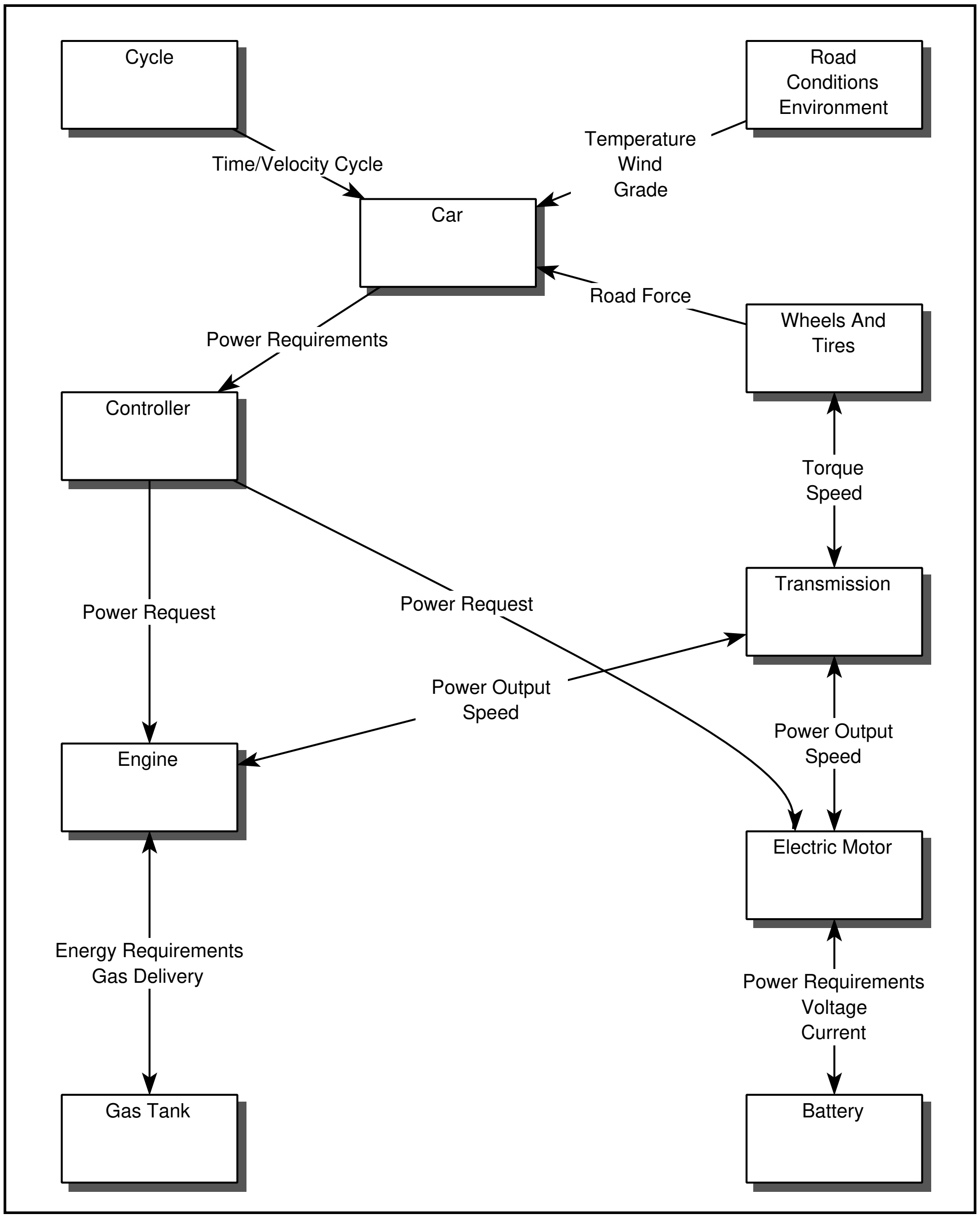
```
        return self;
    }
```

All methods must have a return value and a closing bracket.

3.3.4.3 How The Objects Interact

The following figure shows how the various components are connected within the program, and the information that each object depends on another for.

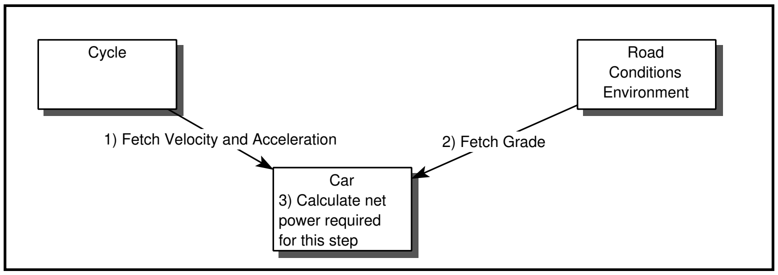
FIGURE 3. Component Interaction in HEV Simulation



3.3.4.3.1 Arbitrary Cycle Mode Operation

The following figure shows the messages that are passed during one cycle of the simulation in arbitrary mode. Since the arbitrary mode does not take into account whether or not the car can achieve the desired performance, very few messages have to be passed, and most of the components are left out.

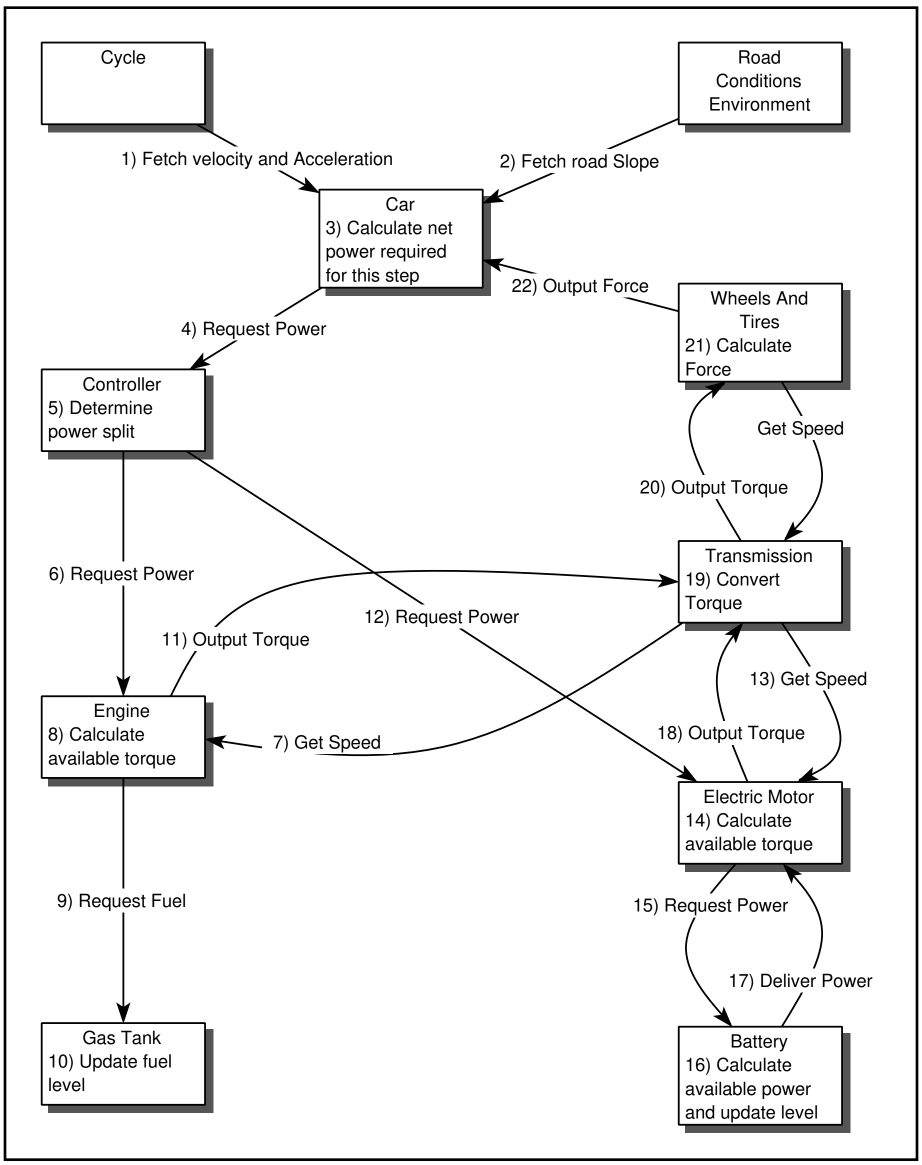
FIGURE 4. Messages Passed In Arbitrary Cycle Mode



3.3.4.3.2 Full Cycle Mode Operation

The following figure shows the messages that are passed during one cycle of the simulation in full cycle mode. The object interactions are complex, and the messages must occur in the numbered sequence given.

FIGURE 5. Messages Passed In Full Cycle Mode



4.0 Theoretical Considerations and Calculations

The math required in this simulation is not very complex. It should be kept in mind that the power of this simulation comes from the very real way that the objects interact as the simulation is run. During a simulation run, it is only necessary to determine the power required at each step in the cycle from the speed, acceleration, and grade parameters. The car itself performs these calculations, and then makes power requests of the controller object which then passes the requests down to other components until the power reaches the road.

The following sections detail the calculations made in each object during a simulation. In discussing the theoretical considerations, I have described the operation of the object as it performs its part of the simulation.

4.1 Calculations Made in the Car Object

The car object calculates the force required to cross the next time step in the cycle, and multiplies it by the current speed to obtain a required power. The power request is then passed on. The forces taken into account in this calculation are aerodynamic drag, rolling resistance, force needed to climb a grade, and force needed for acceleration.

• Aerodynamic drag is calculated using the following formula:

$$F_d = C_d \frac{\rho v^2}{2} A \qquad \textbf{(EQ 1)}$$

where

F_d = Drag Force
C_d = Drag Coefficient (determined experimentally)
ρ = Air Density
v = Current Velocity
A = Frontal Area of the car

• Rolling resistance is calculated using the following formula:

$$F_r = MgC_r \qquad \textbf{(EQ 2)}$$

where

F_r = Rolling Resistance
M = Mass of the Vehicle
g = Gravitational Constant
C_r = Coefficient of Rolling Resistance (determined experimentally)

• The force needed to climb a grade is calculated using the following formula:

$$F_g = Mg\sin\theta \qquad \textbf{(EQ 3)}$$

where

F_g = Force needed to climb grade
M = Mass of the Vehicle
g = Gravitational Constant
θ = Road Angle

• The force needed to accelerate the vehicle is given by the following formula:

$$F_i = Ma \quad \textbf{(EQ 4)}$$

where

F_i = Inertial Force
M = Mass of the Vehicle
a = Required Acceleration

The power required to move the car is then

$$P = V(F_d + F_r + F_g + F_i) \quad \textbf{(EQ 5)}$$

4.2 Calculations Made in the Engine Object

The engine object is not simulated in much detail. The only parameters that it contains are the specifications for a speed-torque curve, stall and redline speeds, and an efficiency. A fifth order polynomial was arbitrarily chosen to provide a close fit to any curve in this situation. Should this prove to be insufficient, a higher order polynomial can be substituted.

The engine receives power requests from the controller, asks for the shaft rotational speed from the transmission (returned in rad/s), and then evaluates the torque needed at that speed to accommodate the power request. The torque is found by the simple formula

$$T = \frac{P}{\omega} \quad \textbf{(EQ 6)}$$

where

T = Torque (Nm)
P = Requested Power (W)
ω = Angular Velocity (rad/s)

After converting the rotational speed of the shaft from rad/s to rpm, the maximum available torque is calculated by the formula

$$y = C_0 + C_1 x + C_2 x^2 + C_3 x^3 + C_4 x^4 + C_5 x^5 \quad \textbf{(EQ 7)}$$

where

y = Torque (Nm)
$C_0 \ldots C_5$ = Coefficients entered by the user (determined experimentally)
x = rpm

NOTE: This formula is used in evaluating all stored curves in the simulation.

The minimum of the required torque and the available torque is then passed on to the transmission.

In drawing fuel from gas tank, the engine has to calculate how much energy it is using. The energy is found from the formula

$$E = \frac{T\omega t}{\eta} \quad \textbf{(EQ 8)}$$

where

E = Energy Used (J)
T = Torque output (Nm)
ω = Angular Velocity (rad/s)
η = Thermal Efficiency

A request of energy is then made of the gas tank.

4.3 Calculations Made in the Motor Object

The math in the motor object is very similar to that used in the engine object. One major difference is that there are two torque-speed curves in the motor object to accommodate two phase electric motors which can run at either high torque and low speed or low torque and high speed, with a step transition. After determining the rotational speed of the shaft, the motor checks to see if it is in the low or high range, and uses the proper curve to check for maximum available torque. Other than this feature, the calculation of the output torque is exactly the same as in the engine. However, once the motor has determined the torque output, it still has to find out if the batteries can deliver the power. Whereas the request from the engine to the gas tank was for energy, the request from the motor to the batteries consist of a power and a time. The motor has to work backwards to find the required power using the equation

$$P = T\omega \qquad \textbf{(EQ 9)}$$

where

P = Power Required (W)

T = Output Torque available from motor (Nm)

ω = Angular Velocity (rad/s)

After the batteries have returned the available power, then the motor has to again convert that power to the final output torque to pass on to the transmission.

If the motor object is sent a request for negative power, it calculates the power then sent to the batteries by

$$P_b = P\eta_{regen} \qquad \textbf{(EQ 10)}$$

where

P_b = Energy sent to the batteries(J)

P = Power Requested (W)

η_{regen} = Regeneration Efficiency

Since there are no brakes in this simulation, the motor is considered to be able to give any negative power needed.

4.4 Calculations Made in the Gas Tank Object

The Gas tank object contains information on the current amount of fuel in L and on the fuel density in kJ/L. When asked for energy input by the engine, the gas tank calculates the fuel used by the formula

$$V = \frac{E}{1000\rho_e} \qquad \textbf{(EQ 11)}$$

where

V = Volume of fuel used (L)

E = Energy Requested (J)

ρ_e = Energy Density (kJ/L)

The current fuel level is then decreased by V.

4.5 Calculations Made in the Battery Object

The battery consists of a voltage-charge curve and a current-charge curve. As in the engine and electric motor, the curves are fifth order polynomials evaluated with (EQ 7). The battery assumes a voltage source, so it first evaluates the maximum available voltage for the current charge level. Then it determines the current drawn using the following equation:

$$I_r = \frac{P}{V} \qquad \text{(EQ 12)}$$

where

I_r = Current Requested(amps)
P = Power Requested
V = Maximum Available Voltage

The power returned to the electric motor is then

$$P = MIN(I_r, I_{max})\,V \qquad \text{(EQ 13)}$$

where

P = Power Returned
I_r = Current Requested
I_{max} = Maximum Available Current
V = Maximum Available Voltage

4.6 Calculations Made in the Transmission Object

The operation of the transmission object, as in the operation of a real transmission, involves torque multiplication. The torque relation between the input and output shafts is given by the equation

$$T_{out} = T_{in} K_G K_{FD} \qquad \text{(EQ 14)}$$

where

T_{out} = Torque Output
T_{in} = Torque Input
K_G = Gear Ration of the current gear
K_{FD} = Final Drive Ratio

4.7 Calculations Made in the Wheel Object

The wheel object must convert the torque given it to a force on the road. This is simply

$$F = \frac{T}{\left(\frac{d}{2}\right)} \qquad \text{(EQ 15)}$$

where

F = Road Force
T = Input Torque
d = Wheel Diameter

5.0 Output Examples

FIGURE 6. Velocity-Time Graphs for Ideal and Limited Vehicles

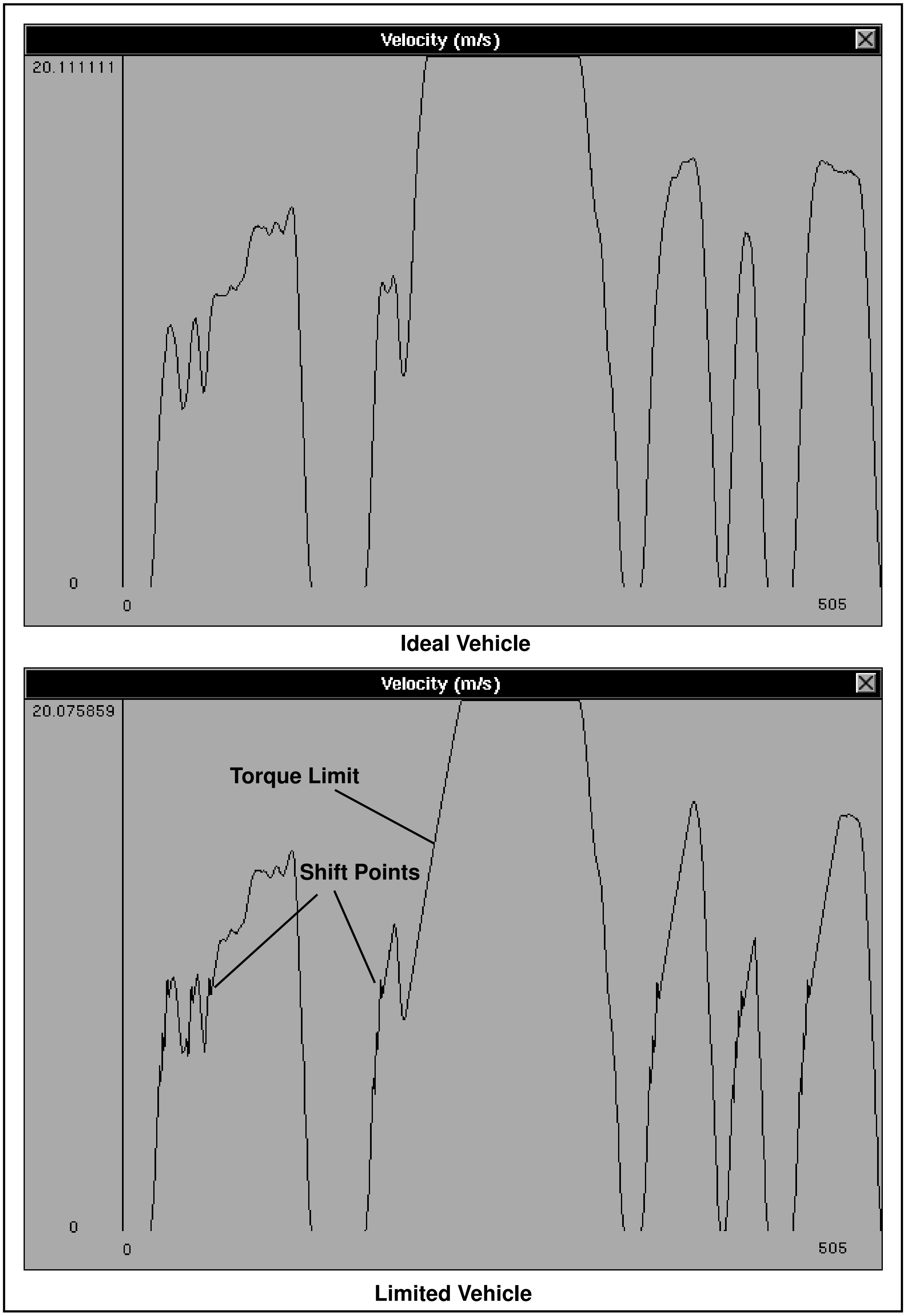
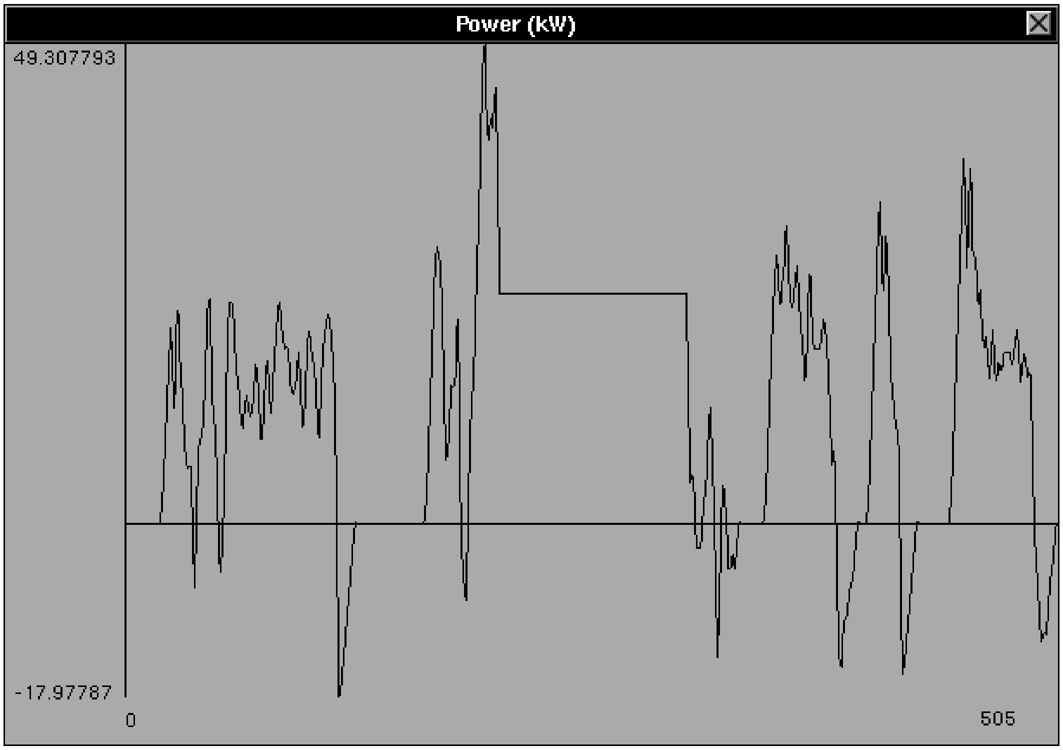
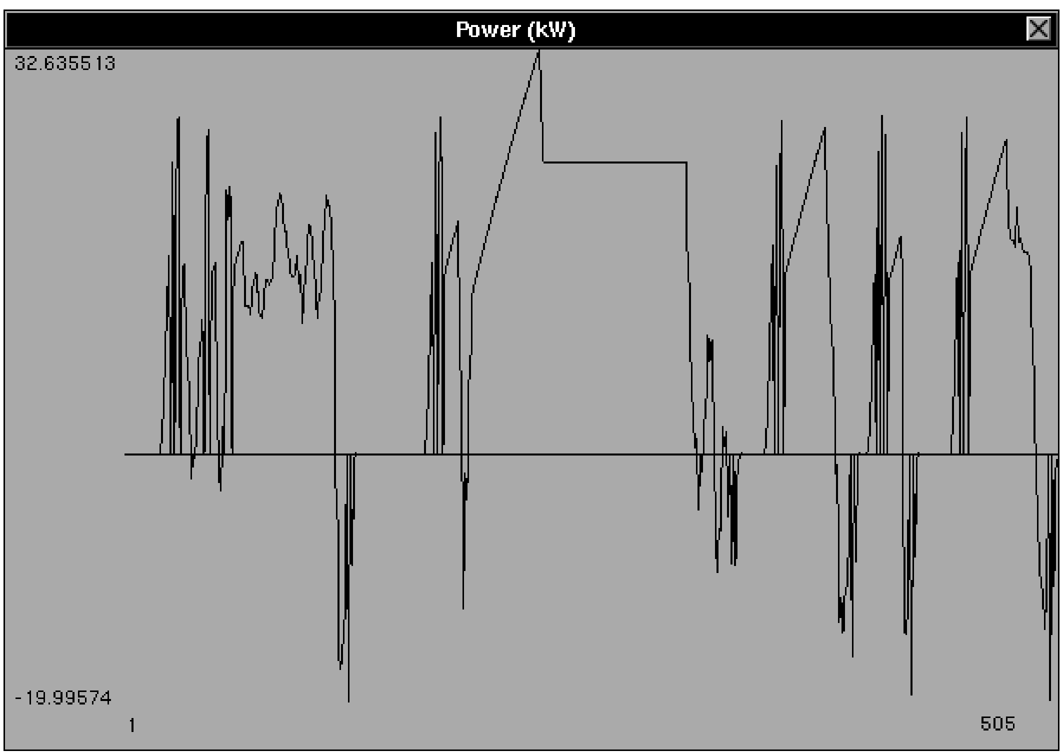


FIGURE 7. Power-Time Graphs for Ideal and Limited Vehicles



Ideal Vehicle



Limited Vehicle

The plots in Figure 6 and Figure 7 were generated with a vehicle weighing 1744kg with a drag coefficient of 0.35, and an electric motor with a flat torque curve of 100Nm. Only the electric motor was used in this run.

Figure 6 shows the velocity-time signatures of vehicles that are identical except that one is ideal and the other one is limited by its power components. One can see the shift points in the plot for the limited vehicle. Also, there are flat, sloping parts of the plot that indicate that the limited vehicle had reached a torque limit. As the plots in Figure 7 indicate, the maximum power reached by the limited vehicle was only 32.6kW compared to 49.3kW needed to match the cycle.

This run required 2.27kWh of energy from the batteries. By adding 50kg of mass to the vehicle, the required energy changes to 2.31kWh. As an example of what would be neccessary to compensate for such a weight gain, changing the drag coefficient to 0.23 reduced the energy used to 2.26kW.

6.0 Conclusion

The requirements set out in Section 2.2 have been met. The program produced simulates all of the essential performance-related components of a hybrid electric vehicle, and the interactions between the components are accurate. At this point, the accuracy of the simulation as a whole is unknown, and the values produced by the program will have to be compared against real test data from the prototype.

There are still some desireable features which could be implemented:

• Plotting of Other parameters

Other values may provide useful information from a test cycle: engine power and torque, motor power and torque, current gear, battery current and voltage output, velocity differential between cycle and simulated

• Additional Information

Duty cycle information for the electric motor

• Saving of Graphs for future analysis

Graphs could be saved and then loaded into another program which would allow for overlaying of graphs for comparison.

• Report Generation

To record various tests, it would be helpful to have a ready-to-print report generated at the end of each run.

• Additional Parameters for a more accurate simulation

Include a speed/efficiency curve for the engine

Appendix A Operating Instructions

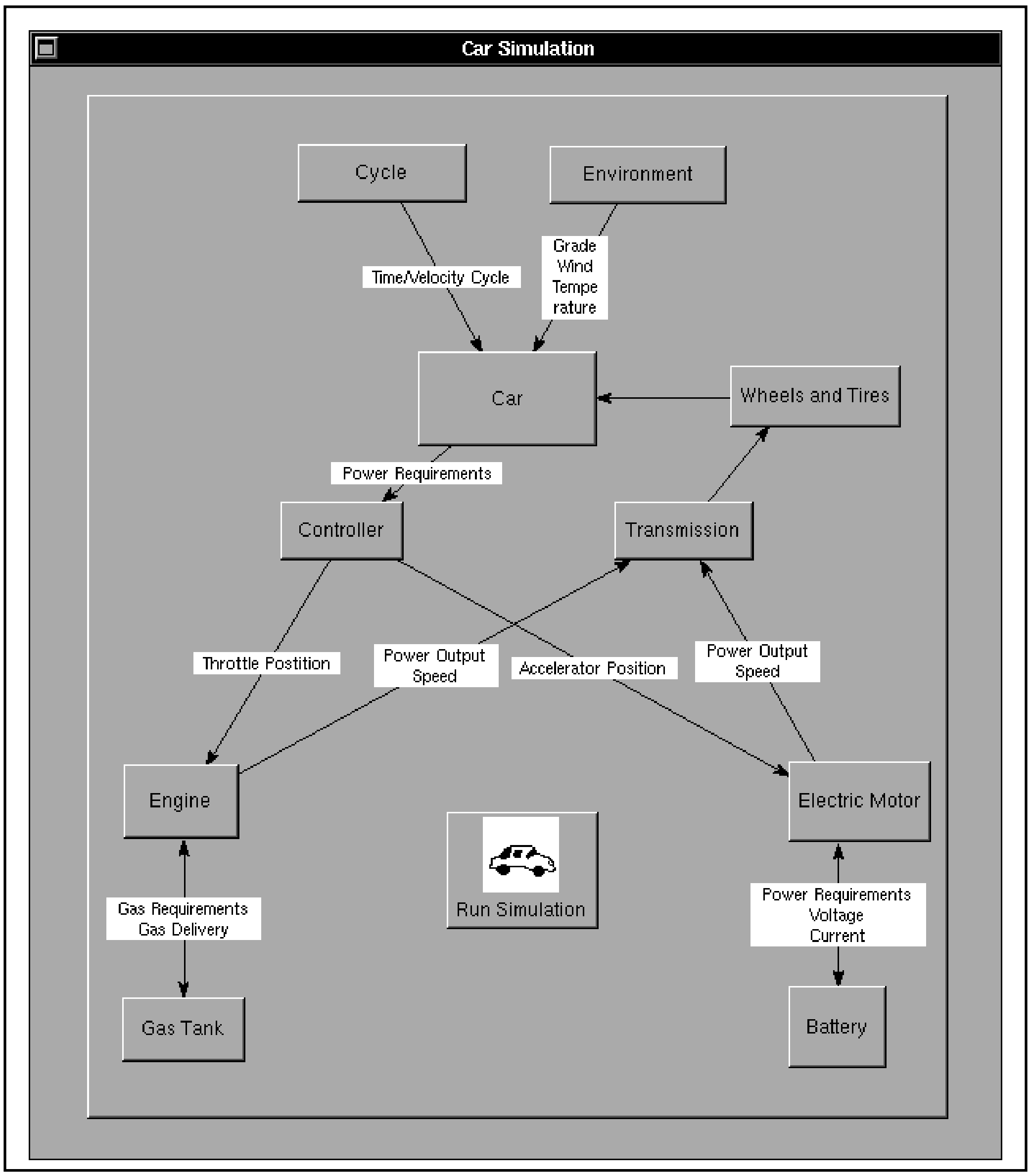
A.1 Starting the Program

The program may be started by double-clicking on its icon or double-clicking one of its files.

A.2 Entering Parameters

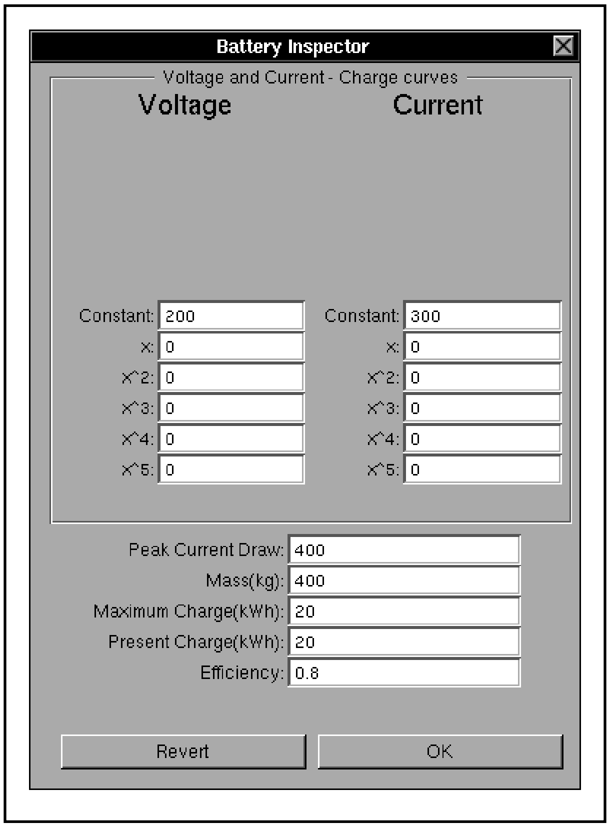
Parameter entry is accomplished through an inspector, which adjusts itself to the proper component depending on which one is selected. The user selects the component he wishes to inspect by pressing the button with its name in the main window.

FIGURE A.1. The Main Window



New values do not take effect until the “OK” button is pressed. If a mistake is made in entering values, pushing the revert button will recall the old values from the component.

FIGURE A.2. The Inspector Panel



A.3 Running the Simulation

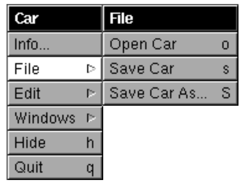
The simulation is run by pushing the “Run Simulation” button in the main window. Graphs of the results will automatically display themselves. A simulation may be run again and again.

FIGURE A.3. The Run Simulation Button

A.4 Saving and Loading Files

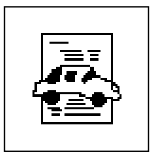
There are three types of files that may be loaded, but only the .car files can be saved from within the program.

A.4.1 The .car File

The .car file can be saved from the 

menu. Selecting “Save Car” will save all of the current component’s parameters. Selecting “Save Car As” will allow for saving the current car under a different name. A car file may either be opened from this menu or by double-clicking it in the workspace. The name of the current car file can be found in the title bar of the inspector panel when the car itself is being inspected.

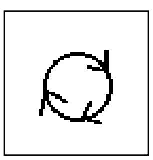
FIGURE A.4. The .car File Icon



A.4.2 The .cycle File

Cycle files can only be loaded into the program. The control for doing this can be found in the cycle inspector. The name of the cycle file appears in the title bar of the inspector window when the cycle inspector is active. Again, the file may be loaded either from within the program or by double-clicking it in the workspace.

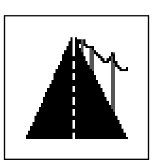
FIGURE A.5. The .cycle File Icon



A.4.3 The .environment File

The handling of environment files is very much like the handling of cycle files. Environment files can only be loaded within the program from the environment inspector, with the name of the file appearing in the title bar of the inspector panel. The file can also be opened from the Workspace by double-clicking on its icon.

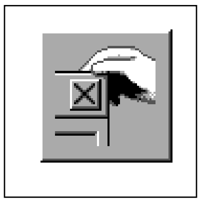
FIGURE A.6. The .environment File Icon



A.5 Printing

The simulation program does not currently support printing. In order to print a graph, use the bundled program “/NextDeveloper/Demos/Grab” to grab a window into a TIFF file to print. The instructions for using grab are on-line, and any program on the NeXT that opens TIFF files will also print them.

FIGURE A.7. The Grab Program Icon



Appendix B File Specifications

B.1 The ".car" File

Car files are stored in the NeXT typedstream format, which is used to archive objects. They may only be created and read by this program.

B.2 The ".cycle" File

Cycle files consist of ordered pairs representing time and speed at a point in the cycle. The abscissa is the time, the ordinate is the speed, and the two are separated by a comma. Below is an example of a cycle file. Each ordered pair must end in a newline.

```
0,0
1,2
3,5
5,6
```

NOTE: DOS files contain a combination carriage-return newline at the end of lines. The carriage-returns must be stripped out before the simulation program can read them.

B.3 The ".environment" File

Environment files consist of two numbers at the beginning representing the temperature in degrees Celsius and the temperature in m/s. They are followed by ordered pairs representing a distance and a slope for that distance. Positive slopes are considered to be uphill and negative slopes are considered to be downhill. Co-ordinates must be separated by a comma, and ordered pairs must be separated by a newline. Below is an example of an environment file.

```
20
20
200,.06
500,.02
500,0
500,-.01
```

NOTE: DOS files contain a combination carriage-return newline at the end of lines. The carriage-returns must be stripped out before the simulation program can read them.

Appendix C Known Bugs

C.1 Some Values Don't Reset

The current gear, gas level, and battery level have to be manually reset after each run

C.2 Accelerating from a standstill

In order to overcome some problems with requests for power, both the engine and motor will produce their full torque from a standing start, even though the cycle may not require it.

C.3 Inspector Title

The title of the inspector panel changes while loading and saving files even though that is not the proper inspector for operation. Further changes of the inspector are correct.

C.4 Saving Car Files

Recently changed parameters will not be saved unless "OK" is pushed before the save.

C.5 Validity Checking

There is almost no validity checking for any input, so improper input may lead to program crashes.

Appendix D Program Listings