

An electric vehicle model and validation using a Nissan Leaf: A Python-based object-oriented programming approach

Advances in Mechanical Engineering
2018, Vol. 10(7) 1–7
© The Author(s) 2018
DOI: 10.1177/1687814018782099
journals.sagepub.com/home/ade


Simon Howroyd  and Rob Thring

Abstract

Electric vehicles are becoming more and more prevalent, especially with major manufacturers announcing that they will be focusing on electric or hybrid vehicles in the future. This article describes an object-oriented approach to a vehicle model using Python 3. This approach allows for flexibility of vehicle design. The key parameters were input to define the specific vehicle for validation, in this case a Nissan Leaf. It is anticipated that this flexibility will lead to rapid exploratory design of vehicle variants, such as four-wheel drive, independent wheel drive and multiple electrical sources. The model had its objects individually validated before the whole vehicle was verified against common drive cycles and a real-world drive in the United Kingdom recorded using an On-board Diagnostics (OBD2) Bluetooth dongle.

Keywords

Driving modelling/simulation, electric, electric vehicles, vehicle control systems, vehicle electronics, vehicle performance, vehicle simulation/modelling, automobile, automotive control, vehicle design, vehicle dynamics, vehicle

Date received: 10 January 2018; accepted: 4 May 2018

Handling Editor: Yong Chen

Introduction

Object-oriented programming (OOP) has many benefits over other programming paradigms such as imperative, functional or procedural paradigms with respect to enhancing the functionality and usability of a vehicle model.^{1–4} Many processor architectures are natively capable of running compiled C++ code⁵ unlike a multi-paradigm environment such as MATLAB⁶ which generally requires an operating system. This opens the door to opportunities of running the vehicle model on small embedded systems in vehicle in future projects to allow for hardware-in-the-loop (HIL) testing and model predictive control (MPC).^{7–10} It is critical for advanced software to be usable by others without a deep knowledge of the language, style or structure of the software in order to enhance usability.¹¹

Furthermore, OOP gives the software for a vehicle model some key functionality benefits. Encapsulation

allows for certain portions of data to be protected from external manipulation. This encapsulation allows for greater certainty that the model is performing correctly and that data have not been manipulated by code trying to force a desired output. It also allows for greater decoupling of different data structures, leading to a modular structure to the full model which utilises composition and inheritance. Composition allows for objects to contain other objects in a ‘has-a’ methodology. For example, a powertrain *has a* motor, battery and gearbox (among others) and therefore receives a

Aeronautical and Automotive Engineering, Loughborough University, Loughborough, UK

Corresponding author:

Simon Howroyd, Aeronautical and Automotive Engineering, Loughborough University, Loughborough LE11 3TU, UK.
Email: S.Howroyd@lboro.ac.uk



reference to these other objects within the model so it can access their data in a controlled manner. Inheritance allows for an ‘is-a’ relationship to be created. For example, the battery and the motor are both electrical devices so they can inherit the ‘ElectricalDevice’ abstract base class. From a user friendliness point of view, it ensures that any access to the voltage of the motor is done in the same way as the voltage of the battery, for example. It also greatly reduces the duplication of code which can lead to excessive debugging. Inheritance goes hand in hand with polymorphism which in turn allows for reduced memory usage and quicker development time.¹²

On the road today are several different types of electric vehicles. They are often loosely categorised as *hybrids*; however, this general umbrella term is not specific enough to describe this work. Any vehicle which has a mechanical connection between an internal combustion engine (ICE) and the wheels is not considered in this model (although could be included in the future) meaning that only electrical drive is explored. However, the model does allow for an ICE to be used as a generator to convert fuel into electrical energy in vehicle as it can be easily compartmentalised in the code as a type of an *ElectricalDevice* object (utilising inheritance). Similarly, a hydrogen fuel cell may also be represented and used as a range extender or as a primary electrical supply instead of a battery, for example.

In this article, we will only consider an electric vehicle with a battery as the only energy store on the vehicle, in order to validate the model and provide a suitable foundation for future work. It is the authors’ intent to use this model to perform explorative research on different vehicle powertrain design configurations and their associated performance, such as independent wheel drive, four-wheel drive (4WD) with one motor and 4WD with one motor per axle, to name a few general ideas.

The structure of the code is given to allow for replication, a demonstration of the programming methodology used and the robustness of the code. To understand and prove the model’s limitations, a robustness study and a validation study were conducted using real-world data.

Model

The OOP language being used for the vehicle model is Python 3.^{13,14} This is a high-level interpreted language, removing the need for length and complicated compilations to run the code. Python has a large standard library, automatic memory management and is cross-compileable into C++ code for embedded systems making it ideal for this research. Furthermore, it is open-source and freely available.

Traditionally, MATLAB is used for engineering and science programming; however, this software is not ideal for low-power embedded systems.^{15,16} With a view to running HIL testing onboard an electric vehicle, it is logical to aim for the lowest power approach to reduce any impact the computer has on vehicle range. Rather than cross-compiling from MATLAB to an embedded system, which is difficult to do, a native Python-to-C++ cross-compiler may be used (e.g. Cython).

High-level classes

In Python, the vehicle is defined as an interface class as in Figure 1. This structure allows for all vehicles to have the same methods at the higher level, that is, a global interface to interact with the vehicle without needing to know the specifics of what type of vehicle it is or how it works. For simplicity, the only methods that are required by higher levels are an ability to demand a velocity and the ability to run the model and update the output at each time step. A key advantage of this method is that multiple vehicles can be spawned into the environment with the same or different parameters allowing for a study of a fleet of vehicles.

Inherited by the car class are the aerodynamics and a powertrain. The aerodynamics only required velocity at each time step since the model is two-dimensional (2D). It then outputs a drag force. The powertrain requires the velocity at the last time step and the demanded velocity at this time step; however, this method can be improved by adding future demands to anticipate a change occurring, that is, feed-forward control. The powertrain outputs a force which is summed with the aerodynamics force and the new velocity is calculated using Newton’s first and second laws.

The powertrain is a dynamic class that can be instantiated in different ways based upon the objects that get passed to its constructor. For example, a single motor object connected to the two front wheels may be passed to the powertrain to define a front wheel drive car like the Nissan Leaf. However, the user may pass the constructor four motors, each only connected to one wheel to simulate a four-wheel independent drive car.

Low-level classes

Beneath the powertrain in the model hierarchy are the motor, wheel, battery, friction brakes, transmission, road and the electronic speed controller (ESC) classes.

The motor and battery both inherit the abstract base class of an electrical device. In this article, a battery is considered but any electrical device could be looked at in the future such as an ICE driving a generator, fuel cell or a supercapacitor. Furthermore, these can be hybridised.¹⁷ Again, this allows for various methods and properties to be defined as common between all

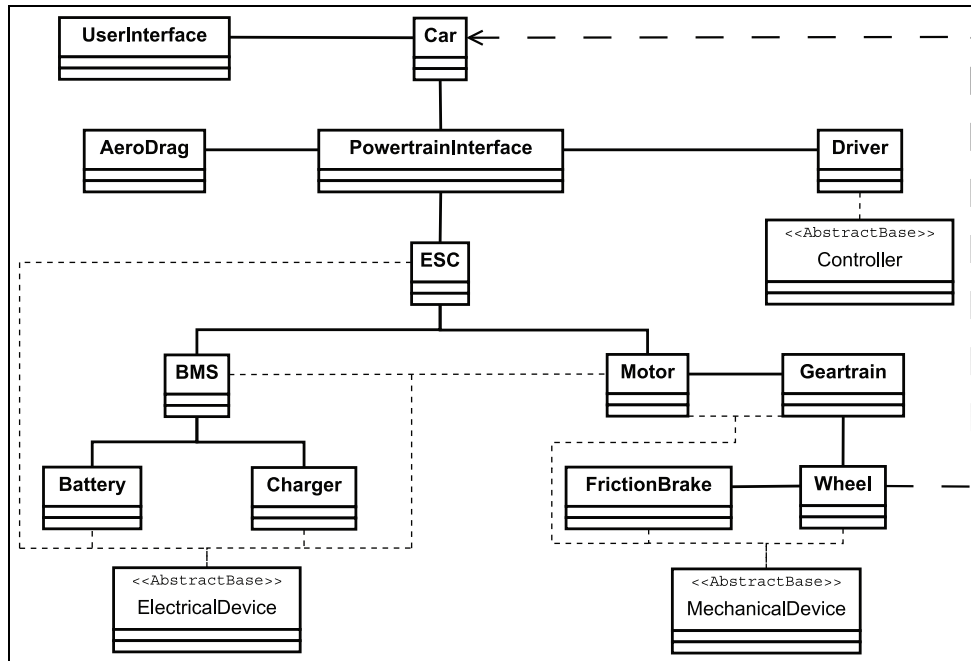


Figure 1. Software schematic showing the various objects of the model, each being independent blocks of code. Of particular note are the abstract base classes which define the linking structure between the higher level classes. This methodology is native to OOP and provides simplicity when changing the vehicle configuration to, for example, four-wheel drive or independent wheel drive.

electrical devices, such as voltage, cumulative energy and power limit. All of these data are available on each iteration allowing for any of these data to be plotted to understand device specification requirements.

Similarly, the motor, wheel, transmission and friction brake all inherit the abstract base class of a rotational, mechanical shaft. This is to permit setting torque limits and to understand how the torque is being transferred around the system. It is notable that here the motor is inheriting two base classes; this is called *multiple inheritance*.

Finally, the ESC and separate traction control module both inherit from a proportional–integral–derivative (PID) controller abstract base class. The whole speed management system is constructed in a nested logic manner to allow the motor and friction brake to have their own independent control systems with rate limits and saturation limits. The two are controlled by the ESC which attempts to reduce the error between the velocity demand from the drive cycle and the actual vehicle speed. The traction control system may have different modes, such as economy, comfort, sport and race, as is seen in many vehicles today, and intercepts the velocity demand signal before it reaches the ESC. It then may amplify or reduce the error or apply a derivative-based ramp to suit the different driving modes. It is notable that the model can bypass the cruise controller and send signals directly to the motor and brake to mimic a driver controlling the vehicle with pedals.

User interface

The user interface (UI) is the main runtime code that the user can configure to meet the needs of the simulation. Pre-defined vehicle parameters can be passed from the UI into the high-level car class constructor to instantiate an object representative of a Nissan Leaf, for example. Furthermore, these pre-defined values can be modified to create subtle or extreme changes to the object in order to perform exploratory research, for example, halving the battery size, reducing motor power or making the car 4WD. Finally, a whole custom car could be defined or iterated to produce the specification of vehicle required to meet certain requirements such as range, weight and power.

The UI is also responsible for interrogating the drive cycle that the user wishes to follow and analysing which point in time, with respect to the system clock, that the model is in. This non-deterministic approach means that faster computers will ultimately produce more data points per second of drive cycle time than the slower ones. It also means that the model runs in real time, opening the door for future HIL testing. Different drive cycles can be attached to different vehicle objects, so multiple vehicles can be simulated simultaneously while driving different cycles, or alternatively a drive cycle can be repeated indefinitely (or until the battery is fully discharged).

Outputs from the UI are typically time series datasets of any user-accessible property of any object within

the model which are typically presented as comma-separated value files or graphs.

Vehicle configuration

As has been shown, the structure of the object-oriented approach allows for linking of different classes to be changed, that is, there are fewer dependencies and therefore fewer changes to the code required to alter the setup of the model. For example, extra motor objects can be spawned, perhaps one per wheel, to simulate independent wheel drive, or the single motor can be linked to all four wheels to provide a more traditional 4WD setup. Alternatively, two motors could be used, one on each axle to provide a distributed 4WD setup. The idea in this example is that it may be preferred to design a vehicle with no motor bay to maximise the interior space, so smaller motors in the wheel hubs would be ideal. The model will assist in explorative design to assess the performance of the vehicle in this configuration.

Verification

Module

This section of the article describes the model verification, that is, the confirmation that the code has been correctly implemented and is performing as expected. This includes each of the individual objects within the model and the linking between objects to become the full model.

Figure 2(a) shows a verification of the wheel object. This test shows the stability and accuracy of the wheel under low acceleration and full power, followed by increasing brake torque for deceleration. For this test, the effects of aerodynamic drag, vehicle inertia and wheel slip were ignored as these effects require feedback from a higher level object in the overall model to calculate. For this test, the whole vehicle model was not included in order to ensure that the only effects seen in the verification are those caused by the wheel and its directly coupled components of the drivetrain.

Following the verification of the wheel model, the whole powertrain needed to be verified. This includes both powered wheels and the two non-driven wheels (which includes brakes on each wheel). It can be seen from Figure 2(b) that very little force is required by the motor for maintaining a constant speed. This is to be expected when aerodynamic drag is omitted. The speed controller is also included in this test to ensure its basic ability to control the motor and brakes.

Controller

Figure 3(a) shows the speed response to the step input of 25 and 100 km/h. Both show that there is an

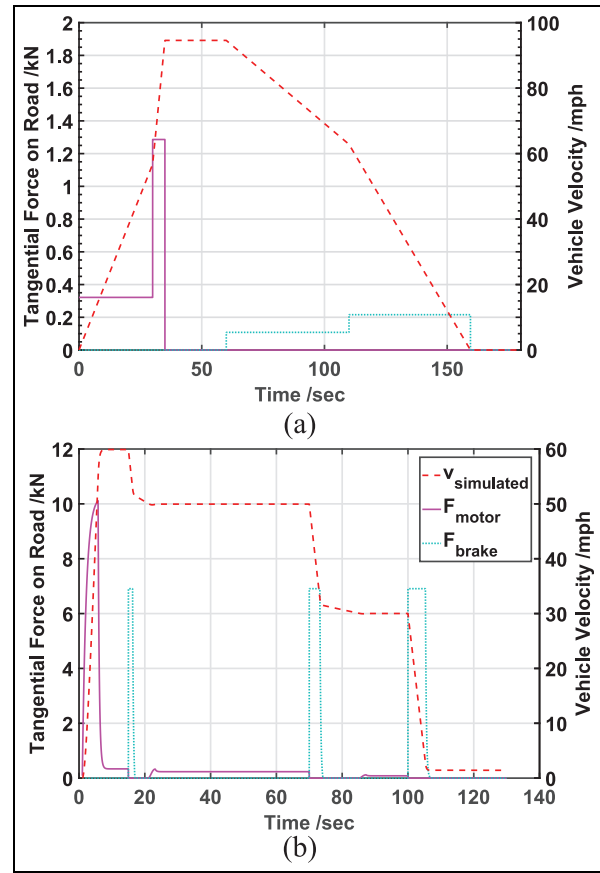


Figure 2. Verification of powertrain: (a) Wheel object verification achieved by simulating a single wheel, motor and brake without any higher level linking. The wheel does not decelerate noticeably once spooled up as it is freewheeling in space, and it only has inertia in this test. (b) Powertrain object verification achieved by simulating two driven and two non-driven wheels which all have independent brakes and no aerodynamic drag or higher level linking. The powertrain is slightly overdamped resulting in no overshoot.

overshoot of 3 km/h for no more than 1 s, which holds true for any step input of significant size. The Worldwide Harmonised Light Vehicles Test Procedure (WLTP) states a trace tolerance of ± 2 km/h and ± 1 s.¹⁸ The WLTP does not implement full acceleration step changes such as this test, so this is a more extreme case. Therefore, it is deemed to be acceptable at this stage since the controller is being developed to match the performance of the Nissan Leaf, not one optimised to meet the WLTP.

The speed control system, imitating the driver's acceleration and brake pedal positions, must be robust enough to not become unstable during normal operation. The control system has been tested using step inputs, impulses, ramps and sine waves. Figure 3(b) shows that there are minimal phase lag (< 1 s) and amplitude losses when the acceleration demand for the vehicle is constantly varying, noting that the frequency

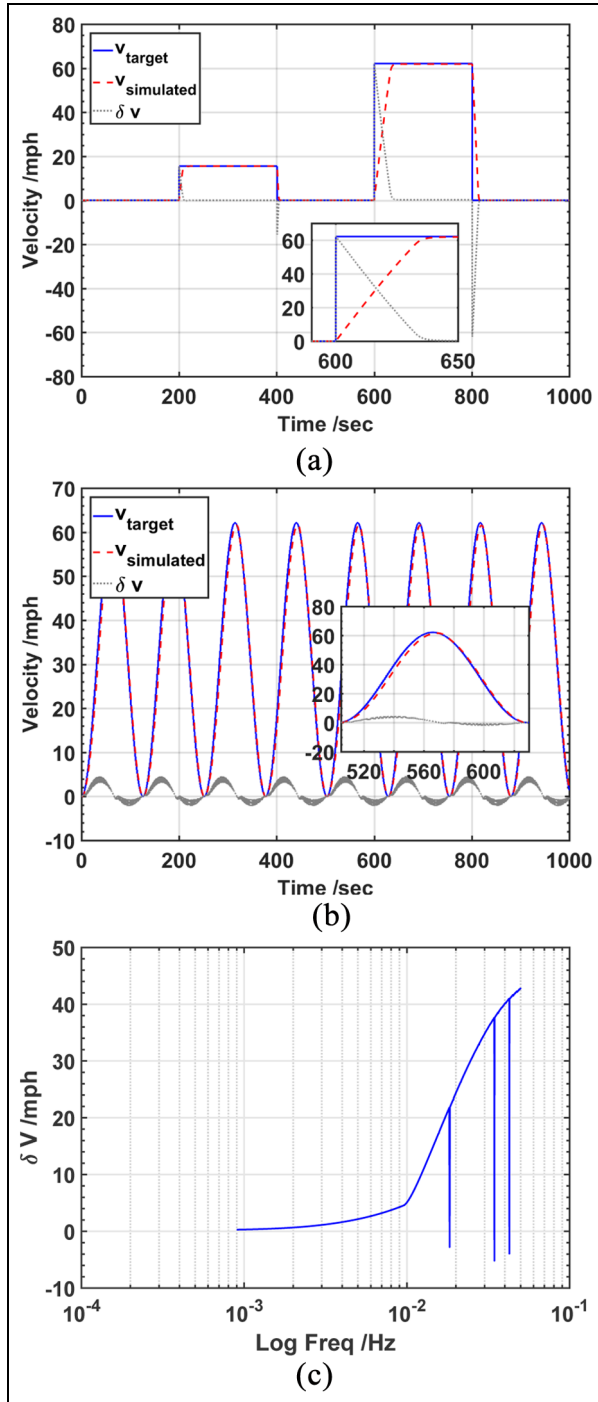


Figure 3. Controller robustness: (a) Velocity step change showing that the controller is slightly underdamped at a maximum power step input resulting in an acceptable overshoot of 3 km/h. (b) Constantly varying acceleration showing minimal phase lag (< 1 s) between the desired velocity and the actual velocity in a sinusoidal demand test. There is no wind-up or diversion over time. (c) Frequency response showing the increase of magnitude of δv as the acceleration requirements increase towards a 0 – 60 mile/h of 10 s.

Table 1. Vehicle data.

Battery V_{max}	398.4 ¹⁹	V
Battery V_{nom}	360 ¹⁹	V
Battery V_{min}	240 ¹⁹	V
Battery P_{max}	90,000 ²⁰	W
Battery mass	294 ¹⁹	kg
Battery capacity	30,000 ¹⁹	Wh
Battery η	97.05 ²¹	%
Battery utilisation	88.50 ²¹	%
Inverter V_{max}	403 ²²	VDC
Inverter V_{min}	240 ²²	VDC
Inverter I_{max}	340 ²²	A _{RMS}
Inverter $I_{max4sec}$	425 ²²	A _{RMS}
C_d	0.28 ²⁰	
C_dA	0.768	m ²
Area	2.744 ²³	m ²
Mass	1521 ²⁴	kg
Wheel rolling r	0.216 ^a	m
Brake r	0.100 ^a	m
Brake T_{max}	500	Nm
Regen R_{max}	20,000 ^a	W
Motor T_{max}	280 ²⁵	Nm
Motor V	345 ²⁵	V
Motor P_{max}	80,000 ²²	W
Motor reduction ratio	7.9377 ²⁵	
Motor rpm_{max}	10,390 ²⁵	
Motor η	96 ²⁴	%
0–60 mile/h	6.3 ^a	s

^aMeasured from the author's 2016 Nissan Leaf.

and amplitude of the sine wave have been chosen to ensure that the acceleration demand is always within the capabilities of the Nissan Leaf, that is, < 10 s. Figure 3(c) shows the full frequency response of the model.

Typical controller testing has also been conducted at high- and low-frequency sine wave inputs, impulse and various steady-state conditions. The results have shown that the simple PID controller is robust and able to take a velocity demand and control the motor/brake in order to meet this demand in a realistic way with respect to the Nissan Leaf. The results shown have been validated against the authors' Nissan Leaf; however, this is discussed in detail later.

Whole vehicle

The final phase of verification is to link all the various objects into the whole model of the Nissan Leaf. This was done using the structure shown in Figure 1 and the data from Table 1. It is well known that the Nissan Leaf meets various drive cycle standards, for example, New European Driving Cycle (NEDC) and Federal Test Procedure (FTP)-75. These drive cycles were used as inputs to the vehicle model and are shown in

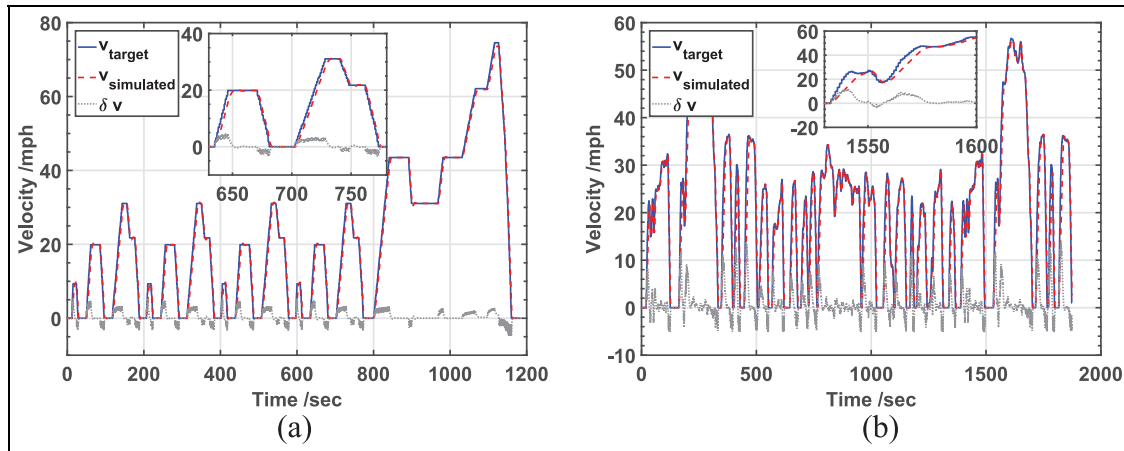


Figure 4. Whole vehicle verification using common standardised drive cycles as velocity inputs: (a) New European Driving Cycle (NEDC) and (b) Federal Test Procedure (FTP)-75 US Environmental Protection Agency Federal Test Procedure.

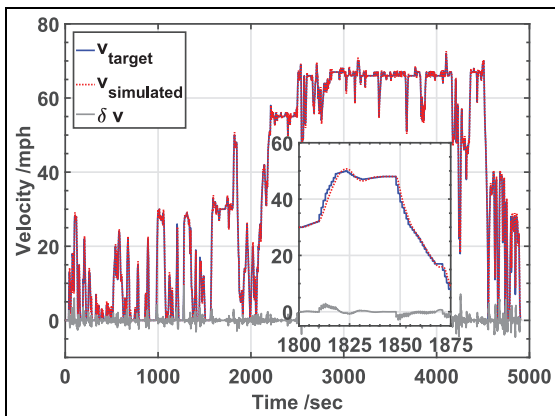


Figure 5. Real-world comparison—real-world drive from Birmingham to Loughborough, UK. The car velocity data were captured using an OBD2 dongle and then fed into the model as a velocity demand. Phase lag is < 1 s and the velocity matched to $< \pm 3$ km/h. The driving style used on this journey was neither aggressive nor modest.

Figure 4(a) and (b), respectively. The points of interest here are the aggressive acceleration zones of the FTP where δV increases to a maximum of 11 mile/h; however, this error is not sustained for more than 1s putting it within the tolerance of the standard test, which would normally be driven by a human.

Validation

Following verification, the vehicle model must be validated against real-world data to ensure that it is an accurate representation of an electric vehicle. In this article, the authors' Nissan Leaf is used to validate the model. Figure 5 shows a 44-mile (70 – km) real-world drive captured by the author using an OBD2 dongle attached to the car. This allows for the recording of

data at variable, random frequencies between 0.5 and 3 Hz which are constantly changing and not definable. The velocity data were saved and input into the model as a target, the output of which is also shown in Figure 5. Note that since the dongle does not record throttle position, the real-world velocity is being used as the target velocity to the model's speed controller. Feed-forward control has not been implemented in this article. This means that there is no ability to compare the phase lag between demand and response for the model and real car. Instead, the authors' experience as a Nissan Leaf owner has been used to ensure that the model is representative, as far as is possible with this method, by ensuring that the transient accelerations are well within the capabilities of the Nissan Leaf such that the real vehicle (or model) has an acceleration capability in excess of the rate of change of target velocity.

Nevertheless, the variance between the setpoint and modelled velocity is less than 3 km/h throughout and was only higher during sudden acceleration or deceleration events. A phase shift of under 1s is observed which is reasonable for stereotypical driving styles; however, it may not be so for race driving (which is not considered). Further studies might be conducted in the future to validate the model for more extreme driving styles.

Conclusion

This article has presented a flexible new electric vehicle model which can be easily reconfigured to any electric vehicle using the parameters in Table 1. The model was validated using a 2016 Nissan Leaf and written in Python 3.

Using the programming paradigm of object orientation, the vehicle model is configurable in its setup. For example, the Nissan Leaf, front wheel drive, single-motor setup can be extended to 4WD with a gearbox

coupling or independent motors on each wheel. This encourages some exciting future work into specification optimisation for different variants of the Nissan Leaf or other validated vehicles using this model.

Verification included an evaluation of each object within the model both independently and with linked dependencies to ensure an accurate representation of the real world. The model was then validated using a real-world drive between Birmingham and Loughborough, recorded by the means of an OBD2 Bluetooth dongle connected to the authors' Nissan Leaf. The validation was acceptable and showed a phase shift of under 1s due to the intentional lack of feed-forward control in the speed controller algorithm.


Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship and/or publication of this article: This work has been supported by EPSRC (EP/M009394/1).

ORCID iD

Simon Howroyd  <https://orcid.org/0000-0003-4390-9081>

References

- Meyer B. *Object-oriented software construction*, vol. 2. New York: Prentice Hall, 1988.
- Longuet-Higgins HC. Inside the C++ object model. *Nature* 1982; 300: 667–668.
- Joines JA and Roberts SD. Fundamentals of object-oriented simulation. In: *Proceedings of the simulation conference*, Washington, DC, 13–16 December 1998, pp.141–149. New York: IEEE.
- Kölling M. The problem of teaching object-oriented programming. Part 1 languages. *JOOP* 1999; 11: 8–15.
- Nagel W. Embedding Python in your C programs, 2005, <http://www.linuxjournal.com/article/8497>
- The Mathworks Inc. MATLAB – MathWorks, 2016, <http://www.mathworks.com/products/matlab/>
- Ramaswamy D, McGee R, Sivashankar S, et al. A case study in hardware-in-the-loop testing: development of an ECU for a hybrid electric vehicle. SAE technical paper 2004-01-0303, 2004.
- Ling KV, Wu BF and Maciejowski J. Embedded model predictive control (MPC) using a FPGA. *IFAC Proc Vol* 2008; 41: 15250–15255.
- Patil K, Molla SK and Schulze T. Hybrid vehicle model development using ASM-AMESim-Simscape co-simulation for real-time HIL applications. SAE technical paper 2012-01-0932, 2012.
- Fan C, Lin CY and Li K. HIL safety function validation of the multi-power sources electric vehicle. In: *Proceedings of the 2016 IEEE 8th international power electronics and motion control conference (IPEMC-ECCE Asia)*, Hefei, China, 22–26 May 2016, pp.2770–2774. New York: IEEE.
- Cooper S and Sahami M. Reflections on Stanford's MOOCs. *Commun ACM* 2013; 56(2): 28.
- Snyder A. Encapsulation and inheritance in object-oriented programming languages. *ACM SIGPLAN Notices* 1986; 21(11): 38–45.
- Van Rossum G and Drake FL. *Python language reference manual*. Bristol: Network Theory Limited, 2003.
- Rossum GV and Drake FL. Python tutorial. *History* 2010; 42(4): 1–122.
- Stenson LV, Turnock SR, Phillips AB, et al. Model predictive control of a hybrid autonomous underwater vehicle with experimental verification. *Proc IMechE, Part M: J Engineering for the Maritime Environment* 2014; 228(2): 166–179.
- Sanderson C. Armadillo: an open source C++ linear algebra library for fast prototyping and computationally intensive experiments. Technical Report, The University of Queensland, Brisbane, QLD, Australia, 31 September 2010. Sydney, NSW, Australia: NICTA.
- Howroyd S and Chen R. Powerpath controller for fuel cell & battery hybridisation. *Int J Hydrogen Energ* 2016; 41: 4229–4238.
- Tutuianu M, Marotta A, Steven H, et al. Development of a worldwide harmonized test procedure for light duty vehicles. Technical Report, 3 January 2014. DOI: 10.3141/2503-12.
- Nissan. First responder's guide, 2016, https://owners.nissanusa.com/content/techpub/ManualsAndGuides/LEAF/2016/2016_LEAF-first-responders-guide.pdf
- Nissan. The new Nissan Leaf: the next chapter, 2013, <http://newsroom.nissan-europe.com/eu/en-gb/media/press-releases/101643>
- Lohse-Busch H and Duoba M. Advanced Technology Vehicle Lab Benchmarking – Level 1: In Argonne National Laboratory, p.9, http://www1.eere.energy.gov/vehiclesandfuels/pdfs/merit_review_2012/veh_sys_sim/vss_030_lohsebusch_2012_o.pdf
- Sato Y, Ishikawa S, Okubo T, et al. Development of high response motor and inverter system for the Nissan LEAF electric vehicle. SAE technical paper, 2011-01-0350, 2011.
- Nissan. *Owner's Manual*. 2016. <https://owners.nissanusa.com/content/techpub/ManualsAndGuides/LEAF/2016/2016-LEAF-owner-manual.pdf>
- Hayes JG and Davis K. Simplified electric vehicle powertrain model for range and energy consumption based on EPA coast-down parameters and test validation by Argonne National Lab data on the Nissan Leaf. In: *Proceedings of the 2014 IEEE transportation electrification conference and expo (ITEC)*, Dearborn, MI, 15–18 June 2014, pp.1–6. New York: IEEE.
- MarkLines. Nissan Leaf Teardown (Part 2): main components disassembled – MarkLines Automotive Industry Portal, 2012, https://www.marklines.com/en/report/rep1104_201209