Battery and Electric Vehicle Modelling

Route Optimisation with the Battery in Mind

An MMSC Case Study on Mathematical Modelling

Candidate Number: 1072462

Abstract

This work will attempt to

The first part of this report will focus on the model for the battery which was the main focus of our project. The second part is the route optimisation problem in the context of electric vehicles.

The model was implemented in Python and there is a graphical user interface available that provides live insight into the model (cf. Figure 7).

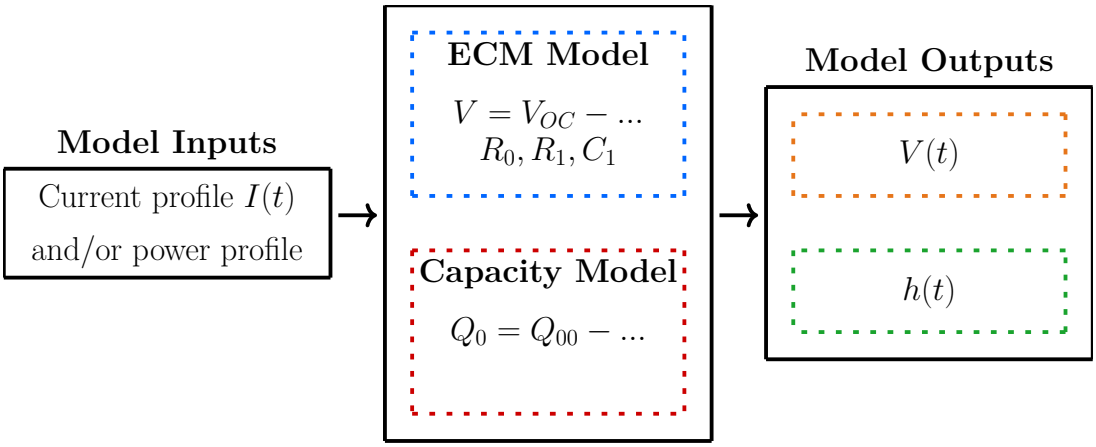


Figure 1: Schematic overview of the battery model introduced in this report. A current input $I(t)$ is converted into an output voltage $V(t)$ and corresponding state of health $h(t)$ due to battery degradation (aging). The core part of the model is an equivalent circuit model (ECM), while aging is modelled by the capacity model.

1 Introduction

Clearly, electric batteries are largely important for various industries today and demand for them is ever-growing. This includes, especially, the renewable energy sector due to the unpredictability of energy supplies such as wind and solar power where short-term storage is a necessary evil. Similar relevance may be found in the car industry where one aims for highly (space-)efficient mobile storage of energy. In many countries / regions, Electric Vehicles (EVs) still lack a well-enlarged network of charging stations, for various reasons including for example, incompatibilties between charging station suppliers. Using methods introduced in the present report, we aim to optimise the customer experience of an electric vehicle owner through a routing application that takes electric battery peculiarities and battery life-time into account.

In this report, we will consider how to model a Lithium-Ion battery, more specifically the Panasonic 18650PF for which there is some characteristic data available in the public domain (Kollmeyer 2018). To do this, we will consider an Equivalent Circuit Model (ECM), namely the Thevenin ECM, which may be found in Figure 3. The key idea here is to model the voltage output $V(t)$ given a current profile $I(t)$. This model not only depends on current and time, but also keeps track of internal quantities like the state of charge (SoC, $s \in [0, 1]$) and state of health (SoH, $h \in [0, 1]$). It can be reduced to a system of ordinary differential equations, for which, using given profiles, we find parameters R_0, R_1, C_1 and more by a fitting procedure. Later on, we also considered aging effects of the battery through usage over time, which resulted in an additional differential equation in the ODE system.

Based on the model we obtained, verified and validated on new pulse test data, the next step was to apply the model to a real-world application, namely that of electric vehicle routing. For this purpose, we obtain graph data of the road network of choice (users may enter new localitys as a text input) from OpenStreetMap (OpenStreetMap contributors 2023). In order to find the best route from A to B, the algorithm then performs a classical routing algorithm called *A-Star* to find the (geometrically) shortest path between A and B, which may not yet be a feasible or optimal path to the destination but a good start. A good initial condition is all that the remaining algorithm requires, which is a Monte-Carlo iterative method that repeatedly applies small, well-defined, perturbations to the graph route and then obtains a time or cost estimate of the modified trip using the battery (and car) model we introduced. If the

modified route yields an improvement in the metric of choice, it is accepted as the new state and the process repeats until a satisfactory route was found.

This report will focus on explicit formulation of the problem and model, numerical simulation of an electric car on a given real-world route and path optimisation through the Metropolis-Hastings method.

2 Problem and Model Formulation

2.1 The Isolated Battery

In order to model a Lithium-Ion battery in and of itself, we consider the following (physical) quantities: Let $s \in [0,1]$ denote the *state of charge* (SoC) of the battery, $h \in [0,1]$ the *state of health* (SoH), $Q \in \mathbb{R}^+$ the charge, $Q_{00} \in \mathbb{R}^+$ the maximum possible charge at the time of production (in Coulombs), $V \in \mathbb{R}$ the voltage across the battery (in Volts) with $I \in \mathbb{R}$ the corresponding current (in Amperes) where $I > 0$ corresponds to discharging the battery. Then, per common definition, $s := \frac{Q}{Q_0}$ is the amount of charge currently present in the battery as compared to $Q_0 \in \mathbb{R}^+$ the current maximum capacity, which itself is dependent on the state of health, as given by $Q_0 := hQ_{00}$. Further let $T \in [-273.15, \infty)$ denote the temperature of the battery (in degrees Celsius) and let $t \in \mathbb{R}$ represent time (in seconds). Let $c \in \mathbb{R}^+$ denote the cycle, so the number of full discharges of the battery.

From the definition of current $I := \frac{\mathrm{d}Q}{\mathrm{d}t}$, we further have that for a single cycle,

$$s = \frac{Q}{Q_0} = 1 - \frac{1}{Q_0} \int_0^t I(\tau)\,\mathrm{d}\tau\,,$$

under the assumption that Q_0, and therefore h, stays constant during that cycle.

2.2 The Equivalent Circuit Model

As mentioned earlier, we want to model the battery as an electrical component which exerts specific behaviour in an electrical circuit. In electrical engineering, such models of more complicated components are frequently represented by equivalent circuits which only consist of basic components, mostly resistors, diodes, transistors, capacitors and inductors.

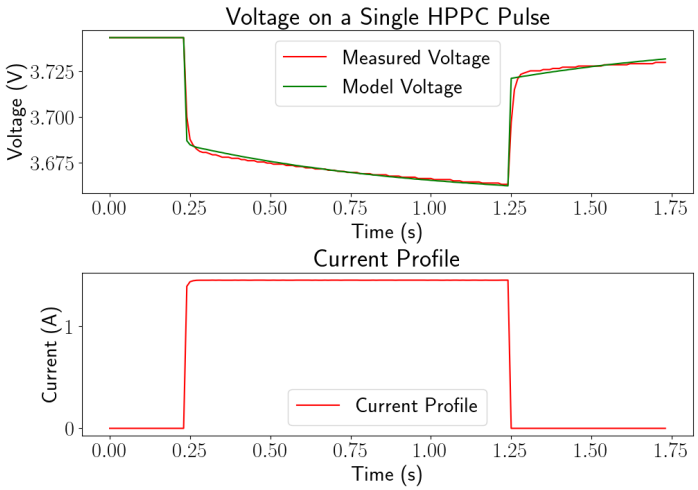


Figure 2: One of the sample HPPC (Hybrid Pulse Power Characterization) pulses we used for parameter finding. We want to model the output voltage $V(t)$ given a current input profile $I(t)$. Each sample is characterised at a specific state of charge s, state of health h and temperature T, this one was taken at $s \approx 1$, $h \approx 1$ and $T = 10$ °C.

Empirically, we know that a battery's voltage output looks roughly as given in Figure 2, which is part of a standardised cycle characterisation procedure taking the battery from "fully charged" ($s = 1$) to "completely empty" ($s = 0$). Data is due to Kollmeyer 2018. As one can see from the voltage curve, the battery exerts a "time-relaxation" behaviour on high-frequency changes in the current. One way of modelling this is through an RC circuit, confer Figure 3: the parallel circuit of R_1 and C_1 is responsible for an exponential-looking behaviour in the voltage curve given (near-)jumps in the current. Additional ohmic impedance is modelled through R_0. The "heart" of the equivalent circuit model is the open-circuit voltage V_{OC} which behaves like a voltage source but strongly depends on parameters s, h and T. We modelled it accordingly, using a polynomial ansatz including cross-terms

$$V_{OC}(s,h,T) = c_1 s + c_2 h + c_3 T + c_4 sh + c_5 sT + c_6 hT + c_7 shT + \mathcal{O}(s^2, h^2, T^2, \ldots)\,,$$

and similar polynomials are chosen for R_0, R_1 and C_1, completing the model. Using the abovementioned dataset (Kollmeyer 2018), these fits were obtained numerically using a least-squares fitting procedure. An example of the dependency on the state of charge s can be found in Figure 4.

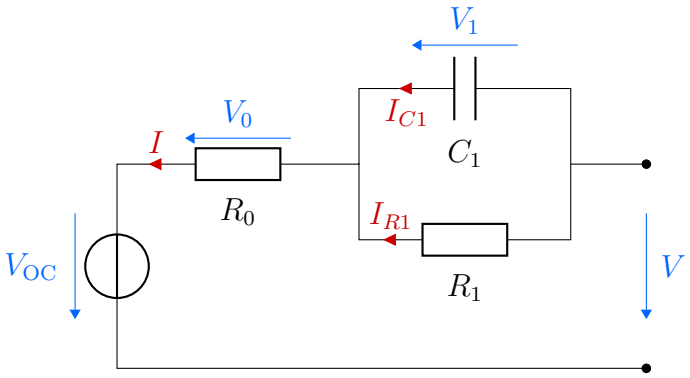
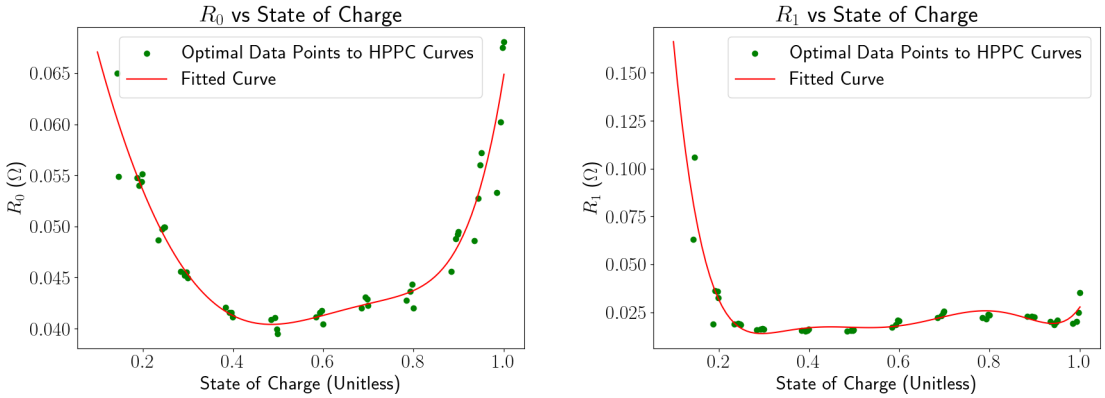


Figure 3: The Thevenin equivalent circuit model (ECM) with parameters $R_0 \in \mathbb{R}^+$, $R_1 \in \mathbb{R}^+$ and $C_1 \in \mathbb{R}^+$ and $V_{\text{OC}} \in \mathbb{R}^+$ the *open circuit voltage* which behaves according to a function $V_{\text{OC}}(s, h, T)$ dependent on s, h and T.

Kirchhoff's law then tells us that the currents $I_{R1} \in \mathbb{R}$ and $I_{C1} \in \mathbb{R}$ add up to the total current $I = I_{R1} + I_{C1}$, and that the voltages $V_0 \in \mathbb{R}$, $V_1 \in \mathbb{R}$ and V_{OC} sum up to $V = V_0 + V_1 + V_{\text{OC}}$. The capacitor behaves according to $I_{C1} = C_1 \frac{\mathrm{d}V_1}{\mathrm{d}t}$, while the resistors follow Ohm's law $V_0 = R_0 I$ and $V_1 = R_1 I_{R1}$. The resulting circuit then behaves approximately as a real Lithium-Ion battery would in many relevant situations.



(a) Fit of R_0 as a function of the SoC (s). **(b)** Fit of R_1 as a function of the SoC (s).

Figure 4: Dependencies of the Thevenin equivalent circuit (Figure 3) model's parameters R_0 and R_1 on the state of charge with a polynomial fit through the data points. Data is again due to Kollmeyer 2018.

Let us consider an application of this model in an electric vehicle next.

2.3 Battery in an Electric Vehicle (EV)

In order to model a road network, let us first consider the definition of an undirected graph $G = (V_G, E)$. We take an undirected graph for simplicity, assuming that cars may go in arbitrary direction along each edge.

2.1 Definition: Undirected Graph

A graph $G = (V_G, E)$ with vertices V_G and edges $E \subseteq V_G \times V_G$ is undirected if and only if $(v_i, v_j) \in E \Rightarrow (v_j, v_i) \in E \quad \forall\, v_i, v_j \in V_G$.

Modelling a vehicle on a road network requires a few more definitions. On a graph (V_G, E) with edges $E = \{AB, AC, ...\} \subseteq V_G \times V_G$ and vertices $V_G = \{A, B, ...\}$, let $d_{AB} \in \mathbb{R}^+$ denote the distance between two vertices $A \in V_G$ and $B \in V_G$ (in meters), $x = x_{AB} \in [0, d_{AB}]$ the progress (current location) on the route from vertex A to B (in meters), $v := \frac{\mathrm{d}x}{\mathrm{d}t}$ denote the current velocity with $v_{\max,\mathrm{AB}} \in \mathbb{R}^+$ the maximum allowed velocity on AB (in meters per second). Then let $T_{\mathrm{env}}(x) \in [-273.15, \infty)$ denote the temperature of the environment (in degrees Celsius) at location x. Note that this graph-based approach to the problem does not require any more geometric information about the network than edge lengths $\{d_{ij}\}_{i,j}$, while it may of course be motivated by spherical coordinates on the earth. Consider for example Figure 5 or Figure 9 for an actual route in Ireland.

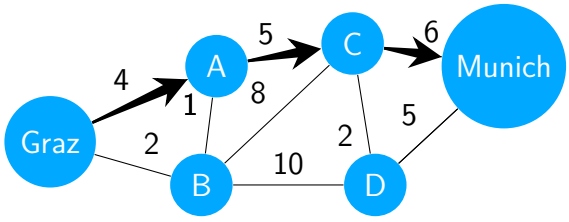


Figure 5: Exemplary route from Graz to Munich on a road network (V_G, E), where edge weights correspond to distances in between the physical places represented by nodes.

The model of our car still lacks a component realising electrical into mechanical power (a motor), which we model very in a very ordinary fashion. Let $P \in \mathbb{R}$, $P := I \cdot V$ denote the (electrical) power the car draws from the battery (in Watts) so $P > 0$ corresponds to discharging the battery. This power is to be realised into a mechanical component $P_{\mathrm{motor}} \in \mathbb{R}^+$ driving the car forwards, heating for the battery $P_{\mathrm{heat}} \in \mathbb{R}^+$, $P_{\mathrm{heat}} = c(T - T_{\mathrm{env}})$, with $c \in \mathbb{R}^+$ the heat conduction constant describing the relation between the heater and battery, and power dissipation $P_{\mathrm{diss}} \in \mathbb{R}^+$. While driving,

$P = P_{\text{motor}} + P_{\text{heat}} + P_{\text{diss}}$. The acceleration of the car $a \in \mathbb{R}$ (in meters per second squared), where $a := \frac{\mathrm{d}v}{\mathrm{d}t} = \frac{\mathrm{d}^2x}{\mathrm{d}t}$ is decomposed into $a_m \in \mathbb{R}$, which directly impacts $P_{\text{motor}}(a_m)$, and the deceleration due to friction (air, etc.) $a_f(v) \in \mathbb{R}^-$, so that in total $a = a_m + a_f$.

Most electric cars have a range from approximately 160 up to 650 km (*EV Database, Range of electric vehicles* 2023), making route changes through charging stations a necessity for longer trips. The resulting routing problem we want to solve then inspires the following definitions of charging stations. On the graph (V_G, E) there exists a set of EV charging stations $V_{\text{charge}} \subseteq V_G$ where $P_{C,\text{charge}}$ denotes the possible charging power (in Watts) at the charging station vertex $C \in V_{\text{charge}}$ with $K_C \in \mathbb{R}^+$ the occuring costs per energy unit (in Euros per Watt second) and $t_C \in \mathbb{R}^+$ the charging time per charging station B (in seconds)[1].

2.4 Battery Aging

A central aspect considered in our project was to study the effects of battery degredation, or aging. One possible model for this would be to express Q_0 in terms of the original capacity at the time of production Q_{00} along with multiple degradation effects that we consider:

$$Q_0(t, c, s, I) = Q_{00} - \frac{F_{\text{acycle}}(c)}{F_{\text{current}}(I)} - F_{\text{cal}} \quad \in [0, Q_{00}]\,. \tag{1}$$

By $F_{\text{acycle}}(c) \in \mathbb{R}$ we denote the **Current Agnostic Cycle Degradation Factor** which models cycle aging at a single current. It models the aging effects resulting only from discharge usage of the battery, at an idealised single current. $F_{\text{current}}(I) \in \mathbb{R}$ represents the **Current Scaling Factor**, a value $0 < F_{\text{current}} \leq 1$ that incorporates behaviour where higher currents are worse for cycle aging (Perez et al. 2018). This term is highly relevant for scenarios where one can choose the amount of current drawn from the battery and then optimise accordingly if this behaviour is well-known. Finally, $F_{\text{cal}}(s, t) \in \mathbb{R}$, the **Calendar Degradation Factor**, ages the battery over time and accounts for suboptimal storage in terms of the state of charge. Storage at the right state of charge and environment is one of the most crucial aspects for battery lifetime.

[1] Charging station & street data was obtained from OpenStreetMap and subsequently invoking `osmfilter england-latest.o5m keep="amenity=charging_station"`. Daily dumps of OSM's public map data were obtained from `https://download.geofabrik.de/`.

The state of health h, the ratio between Q_0 and Q_{00} is then given by

$$h = \frac{Q_0}{Q_{00}} = 1 - \frac{F_{\text{acycle}}(c) + F_{\text{cal}} F_{\text{current}}(I)}{F_{\text{current}}(I) Q_{00}} \quad \in [0, 1] \,,$$

directly following from Equation 1 and yielding a simple, yet effective approximation for aging of the battery. In our model, we used

$$\begin{aligned} F_{\text{acycle}} &= c \left(\alpha_1 + \alpha_2 \exp\{\alpha_3 c - \alpha_4\}\right) \,, \\ F_{\text{current}} &= \alpha_5 - \alpha_6 \exp\{\alpha_7 I - \alpha_8\} \,, \\ F_{\text{cal}} &= \int \left(\alpha_9(T) \left\{ \cosh(\alpha_{10} s - \alpha_{11}) + \alpha_{10} t \left| \sinh(\alpha_{10} s - \alpha_{11}) \right| \left| \frac{\mathrm{d}s}{\mathrm{d}t} \right| \right\} \right) \mathrm{d}t \,, \end{aligned}$$

in part following Perez et al. 2018, but largely our own work with $\alpha_{1,\dots,11} \in \mathbb{R}$ various parameters obtained through least-squares fitting procedures. A plot of the degredation function is given in Figure 6, using the given expressions for F_{acycle}, F_{current} and F_{cal}.

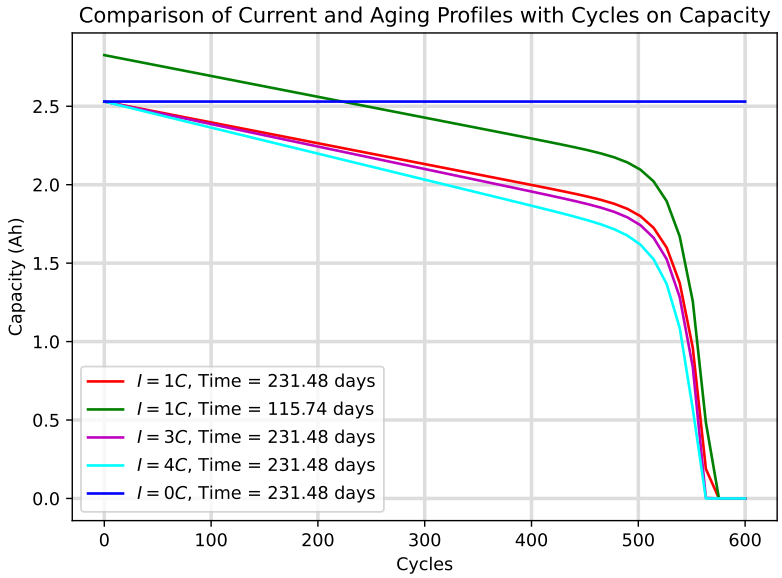


Figure 6: Capacity over discharging cycles behaviour according to the degradation model of the battery, according to Equation 1 with $Q_{00} = 2.9$ A h. Each line corresponds to different discharge currents I, as multiples of the capacity, and a different battery aging time.

Using the car and road network models introduced above, in general terms the problem may be stated as follows:

2.5 A Variational Optimisation Problem

Given source and destination vertices $A, Z \in V_G$ on the graph (V_G, E), **which connected set of edges** $E_R \subseteq E$ connecting A to Z, set of visited **charging stations** $V_C \subseteq V_{\text{charge}}$ along E_R and **charging times** $\{t_C\}_{C \in V_C}$ visited on the route E_R, and **driving behaviour** $a_m(x, v, t, s, h, T_{\text{env}}, ...)$, $a_m \in \mathcal{C}^1(\mathbb{P})$ with $\mathbb{P}$ the parameter space, **minimises**

1. the total travel time $t_{\text{total}} := \int_{E_R} \frac{1}{v} \, \mathrm{d}x + \sum_{C \in V_C} t_C$,
2. the total cost of travel $K := \sum_{C \in V_C} P_{C,\text{charge}} t_C K_C$,
3. or $-N$ where N is the highest possible number of repetitions (commutes from A to Z) with the same battery (requiring $h > 0$).

In other words, we aim to minimise the functional $F \in \mathcal{C}(\mathbb{P})^*$, so $F : \mathcal{C}(\mathbb{P}) \mapsto \mathbb{R}$ where either $F[a_m, E_R, V_C, t_C] = t_{\text{total}}$, $F[a_m, E_R, V_C, t_C] = K$ or $F[a_m, E_R, V_C, t_C] = -N$.

Obviously, solving the stated problem in its full form and with all the given details is a task exceeding the scope of this project, although we were able to make some progress in the direction of it. The intention here is to state the model in high generality and be clear about the simplifications that were made within the scope of this report in order to be able to move ahead.

There are endless possibilities to simplify, for example one could take $V_G = \{A, B\}$ and $E = \{AB\}$ with some d_{AB}, $v_{\text{max,AB}} = \infty$ and $V_C = \{\}$, so only looking at the minimisation of t_{total}. This is equivalent to ignoring the graph-algorithmic component of the problem, as it is only a single line without a charger. Another potential simplifcation would be to take $h = 1$ to ignore battery aging / degradation effects. Or even more radically, $s = \text{const.}$ to only model very short-term (on the order of seconds) effects of this model.

In the remaining sections of the report, we will assume the following:

- $T = T_{\text{env}} = \text{const.}$ and therefore $P_{\text{heat}} = 0$ to ignore the motor heating component. We neglected temperature changes in the environment and a respective battery heating strategy $P_{\text{heat}}(x, v, t, s, h, T_{\text{env}}, ...)$, $P_{\text{heat}} \in \mathcal{C}(\mathbb{P})$.
- $P_{\text{diss}} = 0$ as this is an easily implementable extension to the existing model.
- $v_{\text{max},ij} = \infty \, \forall \, i, j$ so there is no speed limit.

3 Numerical Simulation of a Car

Now that the model is defined, it remains to be implemented. The resulting ODE system can be solved using a known ODE solver such as `ode15s` in MATLAB. While this is the more accurate approach, it is harder to generalise it to a graph where environmental conditions change along different edges of the network. But given the problem settings, it is very well possible. We will however not focus on it within the scope of this report. Our solver uses a simple Forward Euler discretisation.

For a first-order (nonlinear) ODE system $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x})$, a solution may be approximated by the Forward Euler scheme

$$\frac{1}{\Delta t}\left(\boldsymbol{x}_{n+1} - \boldsymbol{x}_n\right) = \boldsymbol{f}(\boldsymbol{x}_n), \qquad \text{so} \qquad \boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \Delta t \cdot \boldsymbol{f}(\boldsymbol{x}_n)\,, \tag{2}$$

to get the approximant $\boldsymbol{x}^n$ at the n-th timestep with parameter Δt and some initial condition $\boldsymbol{x}_0$. Our code used this scheme to solve the ODE system resulting from the equivalent cicuit model. More information on the code can be found in Appendix A. Along with the numerical simulator and optimiser, we implemented a graphical user interface (cf. Figure 7) for the problem, capable of loading any graph.

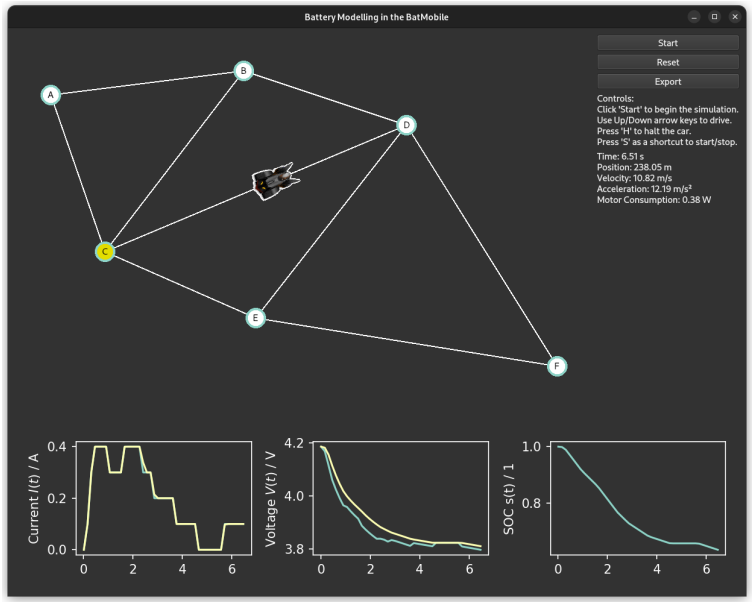


Figure 7: Screenshot of the Qt6 user interface of the car simulation software accompanying this report (confer Appendix A) running on the graph depicted in Figure 5. Using the arrow keys, one can control the battery mobile (*batmobile*) along edges of the road network.

Pressing [↑] accelerates the car and the resulting battery model changes can be observed in real-time using the charts at the bottom. [↓] decelerates (in this scenario, we used current-control, so $I(t)$ is decreased).

The simulator then iteratively updates the state $\boldsymbol{x}^n$ comprised of position x, velocity v, acceleration a, state of charge s, currents I and I_{R1}, voltage V, current edge $(v_i, v_j) \in E$ to name the most central components, in time using Equation 2. In the implementation, we assume that power output $P = IV$ of the battery is directly proportional to mechanical acceleration a_m using a factor. We model friction by

$$a_f = a_{\text{drag}} v^2 + a_{\text{friction}}\,, \quad a_{\text{drag}}, a_{\text{friction}} \in \mathbb{R}^-$$

which then results in $a = a_m + a_f$. Integrating results in velocity and position.

Next up, we will consider the route finding approach and optimisation procedure.

4 Metropolis-Hastings and A-Star

As mentioned in the introduction, our algorithm consists of two main parts. The first step is to find the geometrically shortest path using the A^*-algorithm from computer science. The second step is to perturb this path in a Monte-Carlo Markov Chain (MCMC) optimisation routine that also takes the battery model into account, and finds good driving behaviour along a perturbed route E_R along charging stations V_C.

In order to find good / optimal driving behaviour a_m along the route, we consider the following function basis expansion:

$$a_m(x, s, t) = \sum_{k=0}^{n} a_{k,1} T_k(x) + a_{k,2} T_k(s) + a_{k,3} T_k(t)\,, \text{ for each edge } (v_i, v_j) \tag{3}$$

where $\{T_k\}_{k \in \mathbb{N}_0}$ may be any function basis, for instance the Chebyshev polynomials defined by $T_k(\cos\theta) = \cos(k\theta)$. We assume that the driving behaviour along an edge does not need to be complex in order to be effective, so low expansion orders might suffice for a relatively good result (e.g. $n = 1, 2, 3$). The problem of finding the optimal driving behaviour is then reduced to finding the coefficients $\{a_{k,\{1,2,3\}}\}_{k \in \mathbb{N}_0}$ which is already much more straightforward than minimising a functional. An approach of this form can generally be regarded as a spectral method.

4.1 Shortest Path Finding

As a first step, we aim to find a good initial condition to the MCMC route optimisation routine. This is the shortest path, which we will find using the A^* (A-Star) search algorithm.

The graph data was retrieved using OSMnx (Boeing 2017) which itself is built on top of NetworkX (Hagberg, Schult and Swart 2008). The edges retrieved from the OpenStreetMap API also contain data on the highest allowed velocity $v_{\max}$, but we did not use it in our solver as this would need to be a carefully considered constraint to the optimisation routine.

A^* is based on Dijkstra's algorithm which is already highly efficient for any graph but can be made even more effective by incorporating a distance heuristic into the algorithm. This distance heuristic, for our purposes of a road network, is given by the airline distance between two vertices v_i and $v_j \in V_G$. In general, it could be any mapping $H : V_G \times V_G \mapsto \mathbb{R}^+$ that very roughly approximates the true shortest path length between v_i and v_j (Cornell Optimization 2021).

The A^* algorithm then leverages this distance heuristic in order to not need to search through all vertices (as Dijkstra's algorithm would) but only a subset of them which are in the direction of the target node. This facilitates an incredible speedup on street networks which are normally huge.

The A^*-search algorithm (Cornell Optimization 2021)

```
open_list, closed_list = empty list of possible positions
adjacent_val = 0
next_position = null
push start_position to open_list
while open_list is not empty, do
  current_node = pop(open_list)
  for each adjacent position of current_node (ap_current)
        with value (av_current), do
    if (adjacent_val < av_current) or (ap_current ∈ closed_list), do
      adjacent_val = av_current
      next_position = ap_current
    push ap_current to closed_list
  push next_position to open_list
```

The truly magical aspect of this algorithm is that in the context of a geometric network, it still returns *the* optimal shortest path in between two vertices, despite taking much

less runtime than Dijkstra's algorithm (Cornell Optimization 2021). For the optimiser accompanying this report, we used NetworkX's implementation of A^*.

4.2 Monte-Carlo Optimisation

With this initial route in mind, we will consider small perturbations to each route E_R. The state space we are optimising in here is vast, motivating the use of a Monte-Carlo method. So for our purposes here, we use the Metropolis-Hastings Monte-Carlo Markov Chain method due to Metropolis et al. 1953 and Hastings 1970.

The algorithm can then be outlined as follows, with F^n the metric we are optimising for, for instance the total travel time $F^n = t_{\text{total}} := \int_{E_R^n} \frac{1}{v}\,\mathrm{d}x + \sum_{C \in V_C^n} t_C$.

The Metropolis-Hastings algorithm (Metropolis et al. 1953; Hastings 1970)

until convergence, repeat
  sample a candidate $\boldsymbol{x}^*$.
  set $\boldsymbol{x}^{n+1} = \boldsymbol{x}^*$ with acceptance probability
    $p_{\text{accept}} = \min\left(1, \mathrm{e}^{-\beta(F^{n+1}-F^n)}\right)$, with $\beta \in \mathbb{R}^+$ a transition factor.
  Otherwise, let $\boldsymbol{x}^{n+1} = \boldsymbol{x}^n$.

In order to sample a route candidate $\boldsymbol{x}^*$, we take the old route E_R^n and perturb it to get E_R^*. We define four types of perturbations: Type-1 which turns one edge into two (taking a small detour with a connected vertex). So the route length increases. Type-2 which replaces a node by an adjacent node that still connects the two neighbouring vertices. The route length stays the same. Type-3 is a special case where the destination becomes a part of the perturbation which shortens the route to this node. Finally, Type-4 was added to allow for much larger changes using a NetworkX subroutine yielding a simple path between two vertices which were neighbours in the previous route.

The key idea here is to perturb charging stations in- and out of the route and compare results after performing the numerical simulation described in section 3. This simulation done at every MCMC iteration is generally very expensive, but experiments showed that we can use a fairly large time-step Δt and therefore speeding up the simulation by a great deal, without sacrificing all too much accuracy.

So the Monte-Carlo method explores the state-space to some extent, and returns the best route! Not only that, the optimal driving strategy a_m can be found by incorporating the coefficients $\{a_{k,\{1,2,3\}}\}_{k\in\mathbb{N}_0}$ into the Monte-Carlo state variable $\boldsymbol{x}^n$ and perturbing these by sampling from a probability distribution. Because this is an

iterative optimisation method, larger scales / maps (e.g. England) are not a problem! The Mathematical Institute's Linux machines are capable of handling a route between Cavan and Dublin (in Ireland) within half an hour of running time.

4.3 Special Case: Granny's House

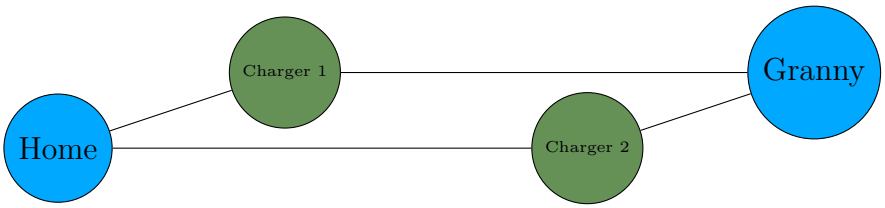


Figure 8: The graph of our exemplary problem "Granny's House", a simplified case of the problem described in subsection 2.5. The situation is that we want to visit our grandmother, who lives 115 km away from our home, in regular intervals. Along the route, there are two chargers. Our car only has a range of 200 km. Is it better to charge at the first or second charger, on the way there or on the way back?

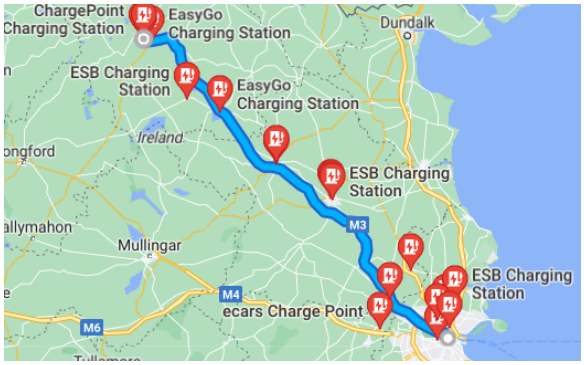


Figure 9: The unperturbed route for the "Granny's House" problem in Ireland from Dublin to Cavan, with charging stations depicted using red pins. Screenshot due to Google 2023.

4.4 Routing in Jericho

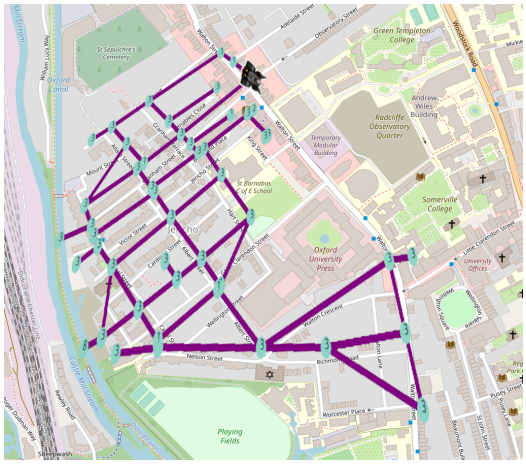
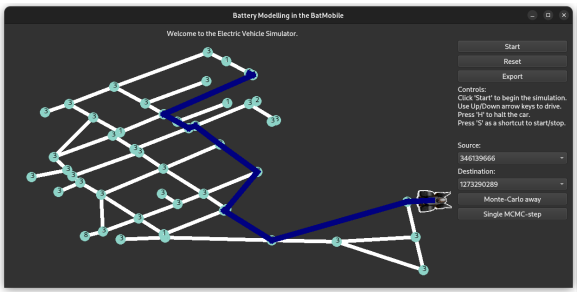
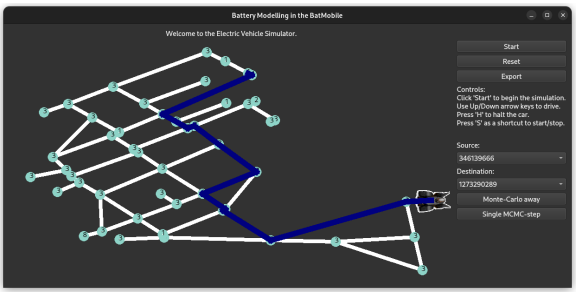


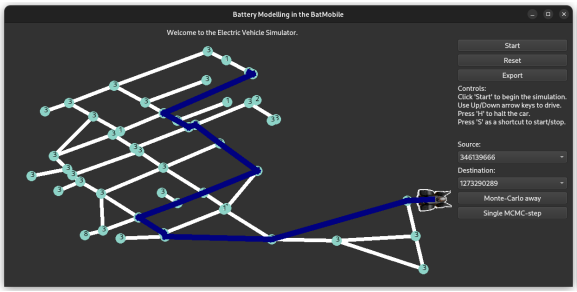
Figure 10: Overlay of the routing graph on a map of Jericho, Oxford, Oxfordshire (OpenStreetMap contributors 2023), without adjusting for the Merkator projection or non-linear roads, which leads to a slightly skewed appearance. The Mathematical Institute is adjacent to one of the shown edges.



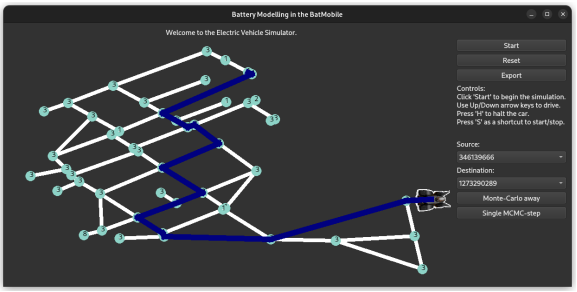
(a) Original route as returned by A^*.



(b) Perturbed route.



(c) Perturbed route.



(d) Perturbed route.

Figure 11: Routes on the extended user interface including A^* route finding along with the Monte-Carlo route perturbation-based optimisation functionality. The user can select a source and target vertex on the road graph $G = (V_G, E)$ loaded from OpenStreetMap and start the optimisation procedure. The displayed road network is that of Jericho, Oxford, Oxfordshire.

5 Conclusion

Potential improvements are, for instance, using more analytical methods to obtain solutions of the variational optimisation problem described in subsection 2.5. The Monte-Carlo Markov Chain optimisation routine could be significantly improved by further extending it with, for instance, Simulated Annealing, where the transition probabilities reduce over the runtime of the algorithm, slowly closing in on an optimum.

References

Boeing, Geoff (2017). 'OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks'. In: *Computers, Environment and Urban Systems* 65, pp. 126–139. ISSN: 0198-9715. DOI: 10.1016/j.compenvurbsys.2017.05.004.

Cornell Optimization (Dec. 2021). *A-star algorithm - Cornell University Computational Optimization Open Textbook - Optimization Wiki*. URL: https://optimization.cbe.cornell.edu/index.php?title=A-star_algorithm (visited on 01/05/2023).

EV Database, Range of electric vehicles (May 2023). Online. URL: https://ev-database.org/uk/cheatsheet/range-electric-car (visited on 01/05/2023).

Google (2023). *Google Maps Ireland*. URL: https://www.google.com/maps (visited on 08/03/2023).

Hagberg, Aric A., Daniel A. Schult and Pieter J. Swart (2008). 'Exploring Network Structure, Dynamics, and Function using NetworkX'. In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught and Jarrod Millman. Pasadena, CA USA, pp. 11–15.

Hastings, W. K. (1970). 'Monte Carlo Sampling Methods Using Markov Chains and Their Applications'. In: *Biometrika* 57, pp. 97–109. DOI: 10.1093/biomet/57.1.97.

Kollmeyer, Phillip (21st June 2018). 'Panasonic 18650PF Li-ion Battery Data'. In: 1. Publisher: Mendeley Data. DOI: 10.17632/wykht8y7tg.1. URL: https://data.mendeley.com/datasets/wykht8y7tg/1 (visited on 02/03/2023).

Metropolis, N., Arianna W. Rosenbluth, Marshall N. Rosenbluth, A. H. Teller and Edward Teller (1953). 'Equation of state calculations by fast computing machines'. In: *Journal of Chemical Physics* 21, pp. 1087–1092. DOI: 10.1063/1.1699114.

OpenStreetMap contributors (2023). *Planet dump retrieved from planet.osm.org*. URL: https://www.openstreetmap.org (visited on 08/03/2023).

Perez, Aramis, Vanessa Quintero, Francisco Jaramillo, Heraldo Rozas, Diego Jimenez, Marcos Orchard and Rodrigo Moreno (2018). 'Characterization of the degradation process of lithium-ion batteries when discharged at different current rates'. In: *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering* 232.8, pp. 1075–1089. DOI: 10.1177/0959651818774481.

A Code Structure and Setup

The code of this group project can be found on GitHub, namely in this repository. To use and sustain a Python virtual environment, install poetry, which works with the `pyproject.toml` file. After installing poetry (and subsequently after pulling, each time), run `poetry install` in the project folder. To install PyBamm as well (which has $\mathcal{O}(\frac{1}{\epsilon})$ number of dependencies), run `poetry install --with=pybamm` instead or additionally. This sadly requires Python 3.8 based on a pybamm restriction. Without pybamm, 3.11 should be fine too. Having all dependencies installed, the main interface may be launched up by executing `./main.py --locality "Jericho, Oxford"` which starts a graphical user interface. Note that you may need to install some Qt6 dependencies.

The relevant code structure is:

- The folder `simulator/` is responsible for the (numerical) simulation itself, which may be invoked without any interface at all.
  - `simulation.py` features the Simulation class with an `iterate()` method that represents a numerical integration step in time by an amount of `dt`.
  - `batgraph.py` exports a class `BatGraph` that represents a graph (a tuple of sets of edges and vertices) that the car will drive on.
  - `batmobile.py` contains the `BatMobile` class that represents our battery mobile i.e. car. **Much of the simulation takes place in this file!**
  - `battery.py` is the central file for our battery modelling project, which exports a `Battery` class, also featuring an `iterate()` method. **Most of the battery simulation takes place in this file!**
  - `optimiser.py` takes care of the optimisation part of the routing problem. It implements the Metropolis-Hastings (Monte-Carlo Markov Chain) method and defines the graph perturbations.
- The interface code is contained within the `interface/` folder.
  - `mainwindow.py` defines the general layout and actions in the user interface.
  - `batmap.py` exports the central widget that renders / animates the BatMobile car on the BatGraph.
  - `graphs.py` handles the connection of the interface and (live) plots. The plots are handled by `matplotlib` and are very intuitive to use, further almost all commands are the same as they are in MatLab.
- `main.py` creates a `MainWindow` and runs the simulator GUI.

B Graph Perturbation

The graph perturbations described above for the MCMC optimisation routine are implemented in `simulator/batgraph.py`. This is the most relevant part of the code for perturbation and subsequent optimisation:

```
def airlineDistance(self, A, B):
    """Returns airline length (heuristic) between A and B."""
    dx = self.nodes[A]["x"] - self.nodes[B]["x"]
    dy = self.nodes[A]["y"] - self.nodes[B]["y"]
    return math.hypot(dx, dy)

def findShortestPath(self, source, destination) -> tuple:
    """Determine the literal shortest path from source to target, in terms of length."""
    return tuple(shortest_paths.astar_path(self, source, destination, heuristic=self.airlineDistance))

def perturbRoute(self, route: tuple) -> tuple:
    """Return a type-1, type-2 or type-3 perturbation of a given path.
    type-1: turn one edge into two edges -> route length increases by one.
    type-2: replace one node along the route with another adjacent one -> route length stays the same.
    type-3: the destination becomes part of the perturbation (shortened route).
    type-4: take a larger detour using a simple path
    """
    i = random.randrange(0, len(route))
    j = random.choice([x for x in (i + 1, i + 2, i + 3, i + 4, i - 1, i - 2, i - 3, i - 4) if 0 <= x < len(route)])
    indexA, indexB = min(i, j), max(i, j)
    nodeA, nodeB = route[indexA], route[indexB]
    if indexB - indexA <= 2:
        # type-1-2-3 perturbation:
        sharedNeighbours = set(self.neighbors(nodeA)).intersection(self.neighbors(nodeB))
        if indexB - indexA == 2:  # type-2 perturbation
            inbetween = route[indexA + 1]
            sharedNeighbours.remove(inbetween)  # ignore the element that is already part of the route
        middle = sharedNeighbours.pop()
        if middle == route[-1]:  # type-3 perturbation, destination is perturbed in
            return route[: indexA + 1] + (middle,)  # so ignore the remaining route (which will be a loop)
        return route[: indexA + 1] + (middle,) + route[indexB:]  # choose any shared neighboudan
    else:
        # type-4 perturbation:
        generator = networkx.all_simple_paths(self, nodeA, nodeB)
        return route[:indexA] + tuple(next(generator)) + route[indexB + 1 :]
```