Neural Network-Based MPC Control Law for Active Cell Balancing

ECE 5772: High Performance Embedded Programming

Luke Nuculaj
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
lukenuculaj@oakland.edu

Michael Muller
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
mwmuller@oakland.edu

***Abstract*—In the context of battery pack balancing, active cell balancing methods have been unanimously adopted by both industry and literature alike for purposes of distributing charge evenly across the pack and thereby prolonging an electric vehicle's (EV) range. Specifically, receding horizon methods – like model predictive control (MPC) – are among the most popular approaches for their ability to consider system constraints. However, they suffer from requiring extensive computation at each time step. Alternatively, this work sought to utilize deep learning to learn the MPC control law and parallelize the forward pass computation with Intel's TBB library. This work considers the case of 3, 5, 10, and 20 cells, and develops deep neural networks (DNN) for each case. The training data consisted of 10,000 randomly-generated state-of-charge (SOC) vectors and their corresponding balancing current vectors as produced by the MPC algorithm – the 20-cell network required 30,000 data points to converge to something usable. Upon deployment of the four DNNs to the Intel Atom dual-core processor, it was found that while parallel_for and parallel_pipeline parallelization techniques reduced the computation time in comparison to the sequential DNN implementation, the combination of the parallel_for with parallel_reduce was the optimal choice in terms of timing: for the 20-cell DNN, 7578.15 microseconds in comparison to the 13256.7 microseconds for the sequential approach, reducing the computation time by $\approx 42.8\%$. As for balancing performance, each of the networks successfully brought their respective battery packs to equilibrium, albeit slightly breaching the system constraints in some instances. Specifically, the largest violation of the Kirchhoff's current law constraint appeared in the 20-cell pack with the sum residing on the order of –0.1.**

I. Introduction

With the rising popularity of electric vehicles in recent years, their superiority to their gasoline-powered counterparts in areas of reduced carbon emissions and cost efficiency have rightfully catapulted EVs to the frontier of today's cutting-edge, infrastructural technology [1], [2]. At the heart of EVs lay hundreds of battery cells – typically of the lithium-ion variety [3]–[5]. Due to the inevitability of manufacturing variations, battery cells exhibit SOC imbalances with one another which – in turn – curtail both battery life and performance, reducing an electric vehicle's range [6]–[9]. To combat this, non-dissipative cell balancing techniques are frequently employed in conjunction with advanced control techniques – MPC being among the most explored and appealing control techniques by virtue of its ability to account for system constraints [6], [10], [11]. While receding horizon methods of control have shown much promise in balancing battery packs under various physical constraints, they suffer not only from their extensive computational demands, but also the frequency with which these computations need be performed. In the pursuit of designing a responsive controller, one which does not quickly produce a set of balancing currents will inevitably contribute to the gradual degradation of the battery pack.

To address the two-fold issue, we utilize deep learning to learn the MPC control law for a given pack size – even the forward propagation of a sufficiently sized DNN requires much less computation than active-set or interior point methods that are typically used in MPC, especially for larger prediction horizons. However, the forward pass of a neural network is not a trivial computation by any means, so we aim to employ parallelization techniques learned in the course (see "Section II Part B" for a detailed explanation regarding how neural networks can be decomposed into smaller sub-tasks that can be computed in parallel). Specifically, we aim to develop DNNs for the case of 3, 5, 10, and 20 cells; the DNN for each case will naturally grow as with more cells, there is a higher-dimensional MPC control law to be learned by the network. As will be explained further in later sections, the training data for each neural network will be extracted directly from the MPC algorithm as SOC vector inputs and balancing current outputs. Lastly, Intel's TBB library is employed in three diverse ways: parallel_for, parallel_pipeline, and parallel_for combined with parallel_reduce. The acceleration of the neural network computation in each case is tabulated in Section IV and compared to the sequential computation, and the performance of each of the neural networks is evaluated for its ability to balance all while abiding by system constraints.

II. Methodology

A. Mathematical Formulation of the Problem

To perform the pack balancing, the initial development of a mathematical model for each cell is paramount. The following

equation is used

$$s_{k+1}^i = s_k^i - \frac{\eta^i T_s}{3600 C^i} u_k^i, \tag{1}$$

where superscript "i" denotes the i^{th} cell in the pack, s is the SOC, $\eta \in [0, 1]$ is the Coulombic efficiency, C is the cell capacity in Amp hours, T_s is the sampling period in seconds, and u_k^i is the i^{th} cell's balancing current at time step k. With this formulation, the implication is that $u_k^i > 0$ pulls charge out of the cell, and $u_k^i < 0$ moves charge into the cell. For this work, homogeneous cells are assumed with values $\eta = 1$, $C = 4.1$, and $T_s = 1$.

Because the cell-level dynamics represented in (1) are decoupled between all cells, the state vector $x_k \in \mathbb{R}^{n\times 1}$ is the column vector of each cell's SOC (n being the number of cells in the pack), and the control vector $u_k \in \mathbb{R}^{n\times 1}$ is a column vector of each cell's balancing current, and the sparse system dynamics can be expressed as

$$x_{k+1} = Ax_k + Bu_k \tag{2}$$

$$A = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} B = \begin{bmatrix} -\frac{1}{3600\cdot 4.1} & 0 & \dots \\ 0 & -\frac{1}{3600\cdot 4.1} & \dots \\ \vdots & \vdots & \ddots \end{bmatrix},$$

where $A, B \in \mathbb{R}^{n\times n}$.

In addition to the mathematical formulation of the system dynamics, careful attention must be paid in acknowledging the physical constraints inherent to the problem. Besides the systems dynamics constraint (2), each cell's SOC must fall within the continuous range of [0,1] (3), there must be limitations placed on the current (4) to prevent damaging the pack, and the sum of the balancing currents must equate to zero (5) (Kirchhoff's current law). For clarity, the system constraints are expressed as

$$0 \le x_k \le 1 \tag{3}$$

$$-0.3 \le u_k \le 0.3 \tag{4}$$

$$\sum u_k^i = 0. \tag{5}$$

With an MPC algorithm, the optimization problem

$$\min_{u_0,\dots,u_{N-1}} x_N^T P x_N + \sum_{i=0}^{N-1} \left(x_i^T Q x_i + u_i^T R u_i \right) \tag{6}$$

can be cast onto the battery balancing problem, where N is the size of the prediction horizon, P is the block-diagonal weighting matrix on the terminal state, Q is the block-diagonal weighting matrix on the intermediate states, and R is the block-diagonal weighting matrix on the control sequence. The MPC algorithm here is subject to the system constraints outlined in (2) (3), (4), and (5). For this particular case, the diagonal elements of P and Q are set to 100, and the diagonal elements of R are set to 0.01. This specific assignment of the weights asserts a larger penalty on the system being away from equilibrium, and less of a penalty on asserting harsher balancing currents. And finally, the size of the prediction horizon $N = 15$, translating to how many time steps into the future the MPC algorithm looks into the future to produce its control move.

Computationally, this is an incredibly expensive algorithm, and as mentioned earlier, needs to be computed at each time step for ideal system control. The approach outlined here seeks to use deep neural networks to instead learn the MPC control law. Additionally, deep neural networks are easily parallelizable structures, and the techniques learned in class can be readily applied to further expedite the computation (discussed in Section C). DNNs provide the convenience of having a network to learn the MPC algorithm. Then we can leverage the redundant computation of the network architecture with parallel computing techniques.

B. Training and Parallelizing the Deep Neural Networks

Each of our DNNs was trained with outputs from the MPC algorithm for a given pack size 3, 5, 10, and 20 (see Section C for the tabulated sizes of each of the networks used in this work). With our initial training set we recognized the accuracy of our networks was poor for SOCs not used for training. This was expected since our networks learned the MPC algorithm for only a single initial SOC. Alternatively, we generated 10,000 random SOC vectors, used the MPC algorithm to fetch the associated 10,000 control vectors, and this approach covered a wider span of possible SOCs, leading to better all-around balancing performance. While this method worked for our lower sized packs, our larger packs were still having accuracy issues. For our 20-cell pack, the neural network was initially trained on 30,000 data points, and once the performance converged to a local minimum, the training set was reduced to 20,000, and the process was repeated iteratively until the training set consisted of 1,000 data points. This method of training the 20-cell network seemed to work the best in terms of converging to a state of acceptable performance.

Once the training data acquisition process had been completed and the neural networks performed well in terms of MSE, a Python script was written to extract our network weights and biases and format them into header files to be readily used in our CPP implementation.

Neural networks are often dubbed "embarrassingly parallel" as there is little to no effort required in separating the forward pass calculation into smaller sub-tasks which can be performed in parallel. To understand why, consider the formula for computing the output of a neuron k

$$z_k^j = \sigma \left(b_k^j + \sum_{i=1}^{n_{j-1}} w_i^k z_i^{j-1} \right) \tag{7}$$

where z_k^j is the output at the j^{th} layer and the k^{th} neuron, b_k^j is the bias at the j^{th} layer and the k^{th} neuron, w_i^k is the weight that connects the i^{th} neuron in the previous layer to the k^{th} neuron in the current layer, and σ is the activation function of choice. For this particular case, we opted for the ReLU activation function in the hidden layers, and a purely linear

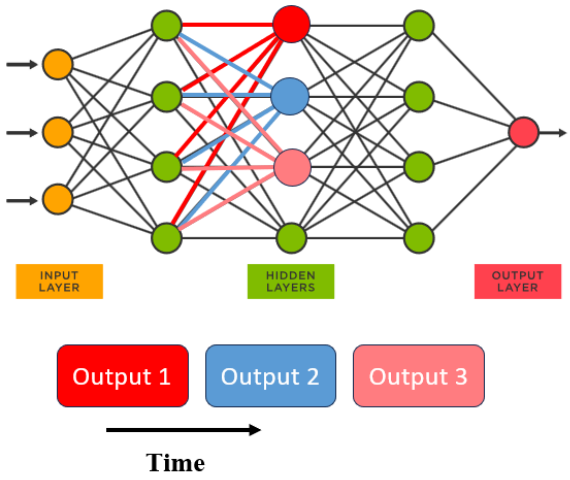

Fig. 1. Sequential flow for layer computation.

activation for the output layer for purposes of computational lightness. Consider the sequential implementation of each of the neurons' outputs shown in Fig. 1.

For a given layer j, the computation of the k^{th} neuron's output is completely independent of the other neurons in the same layer. Because of their computational independence, (7) for each neuron in a layer can be computed in parallel. Specifically, we utilize three methods sourced from Intel's TBB library to accomplish the parallelization: parallel_for, parallel_pipeline, and parallel_for combined with parallel_reduce. The visualization of the parallel_for (Fig. 2) separates the output calculations into sub-ranges, and it's over these sub-ranges that the scheduler assigns a designated worker to perform the sub-range of the output computations. The parallel_pipeline approach takes this concept further and constructs a software pipeline over a given layer, where each of the tasks queued into the pipeline are the output computations for the current layer (Fig. 3). The pipeline contained three stages: serial_in_order, parallel, and finally serial_in_order. The first stage passes in each layer to the pipeline, the second performs the node dot products in parallel and finally the third stage assigns the output to our current layer's output array.

While the aforementioned parallelization methods may seem to be the furthest that we can break up the layer computation, the dot product for each output neuron itself is an associative operation. With this fact, the addition itself need not be sequential, but can instead be reduced with the introduction of the parallel_reduce function. Specifically, the third and final parallelization technique uses parallel_for to map the output neuron computations, and parallel_reduce inside of the tasks to perform the accumulation (Fig. 4).

III. Experimental Setup

As mentioned earlier, a Python script was developed to extract the weights and biases from the trained neural networks

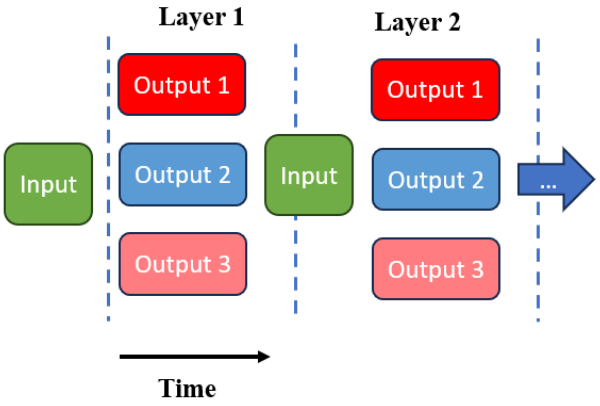

Fig. 2. parallel_for layer computation.

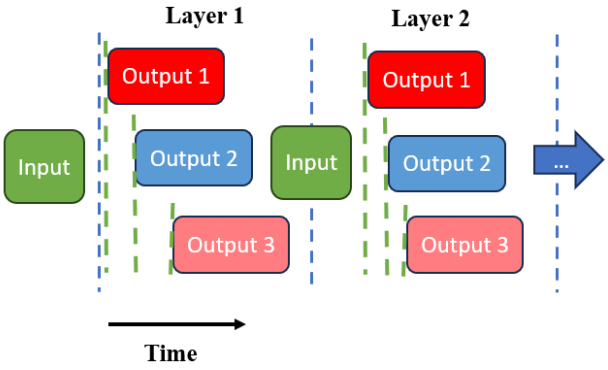

Fig. 3. parallel_pipeline layer computation.

in MATLAB for the 3, 5, 10, and 20-cell cases. These extracted weights and biases were formatted into usable CPP arrays that were stored in header files and included in the main file upon execution. Initially, a WSL environment was used to run each of the neural networks' CPP implementations in a Linux environment. Once the outputs were cross-validated and confirmed to be matching with the identical neural networks in MATLAB, the CPP code was exported to the board where successful operation of each of the neural networks was confirmed. Here, each neural net would stream its SOC and balancing current data to a text file, which was recovered and

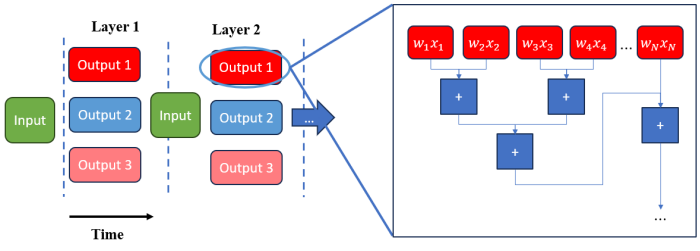

Fig. 4. parallel_for and parallel_reduce layer computation.

TABLE I
NEURAL NETWORK PARAMETERS

Pack Size (Cells)	3	5	10	20
Num. Layers	6	6	6	7
3 Cell Size	3-48-48-48-48-3			
5 Cell Size	5-64-64-64-64-5			
10 Cell Size	10-96-128-128-96-10			
20 Cell Size	20-256-256-256-256-256-20			

TABLE II
TIMING RESULTS (μS)

Cell Num.	3	5	10	20
sequential	547.35	645.7	2107.05	13256.7
for	580.4	661.7	1638.5	8424.35
pipeline	791.65	1125.95	2374.5	9758.45
for + reduce	518.95	737.25	1596.95	7578.15

exported back to MATLAB for further analysis. For the sake of clarity, information about the networks implemented in CPP are listed in Table I.

IV. RESULTS

Timing results for the four CPP implementations (sequential, parallel_for, parallel_pipeline, and parallel_for + parallel_reduce) were run on the Intel Atom and obtained over an average of 20 runs. These timing results are tabulated in Table II and are plotted in Fig. 5. While for the smaller network sizes of 3 to 5 cells the sequential approach seemed like it had the generally faster execution, it wasn't until the DNNs for 10 and 20 cells that the gap between the sequential and parallel approaches began to widen. This separation in execution times is especially apparent in the 20-cell DNN, where the sequential exhibits an average execution time of 13256.7 μs, while the parallel_for + reduce was the fastest implementation of them all with an average execution of 7578.15 μs. Provided that the Intel Atom is a dual-core processor, the near-50% reduction in computation time tracks logically. Furthermore, while the parallel_for and parallel_pipeline methods did save on time for the larger network sizes, we conclude that the parallel_for + reduce won out because it is implemented with both the isolated output neuron computation and the reductive accumulation of the dot product in mind. Given this, the for + reduce was undoubtedly the most appropriate parallelization technique for an operation of this variety.

While the timing results are clear, they are virtually meaningless without usable results. Therefore, we hone in on the balancing performance of the CPP-implemented DNNs. For the sake of brevity, we will only be considering the cases of 5 and 10 cells, but it should be noted that the cases of 3 and 20 cells yielded similar performance. As seen in Fig. 6, 7, and 8, the 5-cell network performs well in balancing the battery pack, abiding by the current constraints as well as the Kirchhoff current law constraint, with the sum for the latter residing in the vicinity of 10^{-3}.

Furthermore, the 10-cell network showcases similar quality in terms of performance (Fig. 9, 10, and 11). That being said,

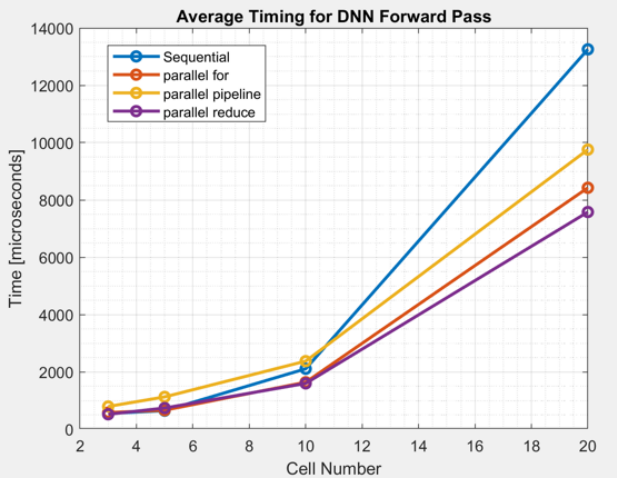

Fig. 5. Plot of timing results (μs).

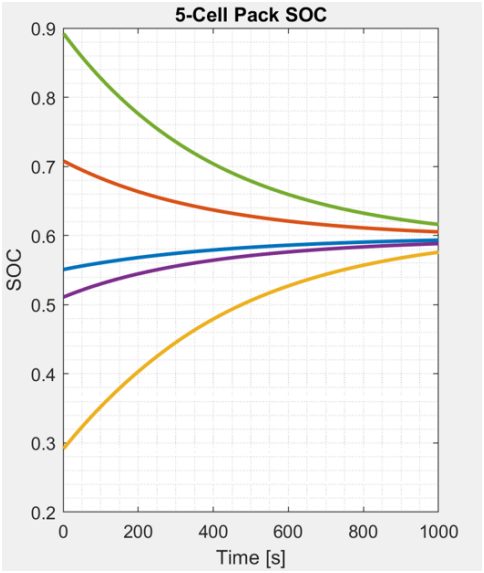

Fig. 6. 5-cell balancing SOCs.

it does slightly breach the current constraints as the balancing currents enter the vicinity of ± 0.4. Nonetheless, the sum of the balancing currents resides on the order of 10^{-2}, which is still acceptable performance and could likely be corrected with more extensive training.

V. CONCLUSION

Overall, parallelization was expected to improve our performance, and it most certainly did upon consideration of the results. With the number of nested loops of computation, it was expected to see great improvements over the sequential implementation as the size of the network increased. Our problem directly benefits from a large network size as well, specifically from wider layers as the parallel execution occurs over the layers. As the pack size increases, we need to add layers and increase the number of nodes to achieve reliable results compared to the MPC approach. With a significant

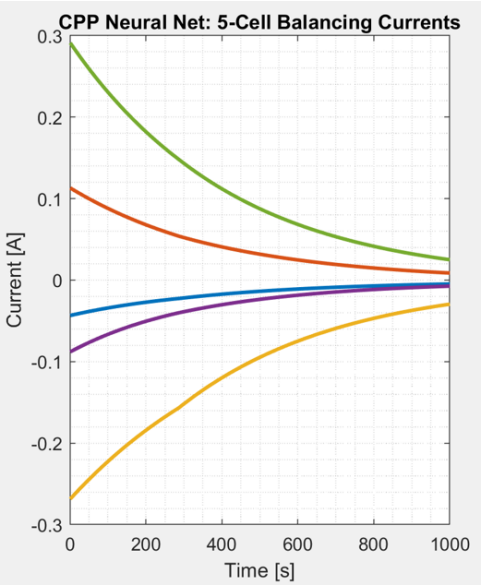

Fig. 7. 5-cell balancing currents.

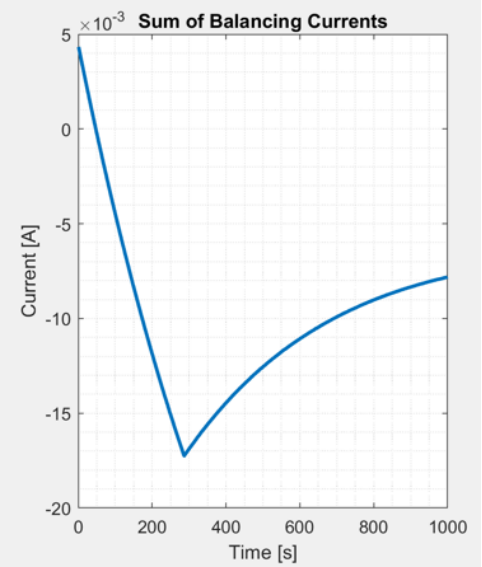

Fig. 8. 5-cell current sum.

improvement for a relative pack size of 2, we can assume this only gets better as we process larger pack sizes, e.t 100 or even 1000. Our parallelization techniques provide a necessary improvement to efficiently balance at high pack sizes. Parallel_reduce + parallel_for was surprising as the best overall method, and upon further analysis, was the most appropriate parallel technique for performing several dot products in parallel. It would be interesting to see if this continues as the pack size increases, or if we converge on the parallel_for technique being the more efficient method.

REFERENCES

[1] P. Ahmadi, "Environmental impacts and behavioral drivers of deep decarbonization for transportation through electric vehicles," *Journal of Cleaner Production*, 2019.

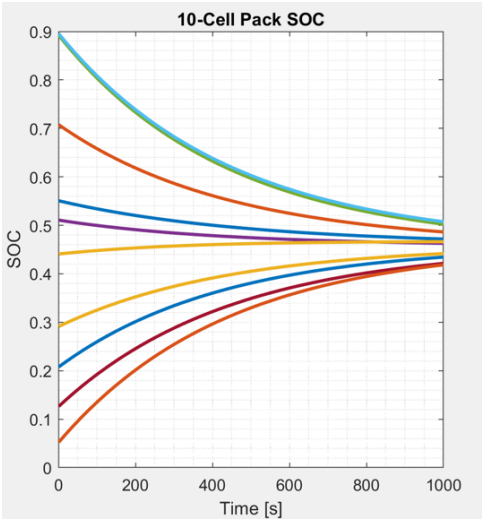

Fig. 9. 10-cell balancing SOCs.

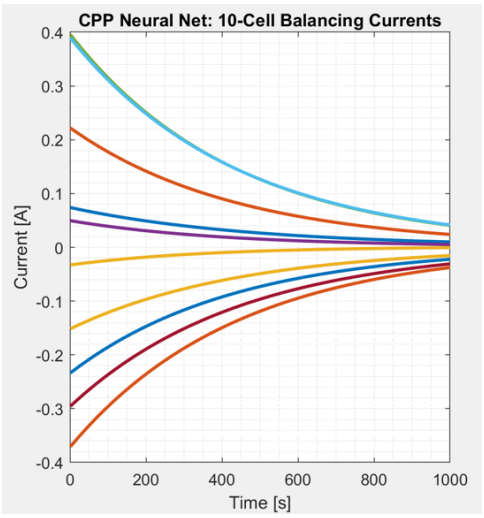

Fig. 10. 10-cell balancing currents.

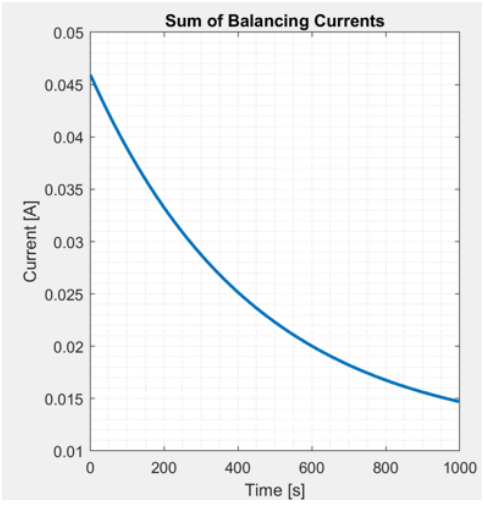

Fig. 11. 10-cell current sum.

[2] H. Hao, X. Cheng, Z. Liu, , and F. Zhao, "Electric vehicles for greenhouse gas reduction in china: A cost-effectiveness analysis," *Transportation Research Part D*, 2017.

[3] X. Chen, W. Shen, T. Vo, Z. Cao, and A. Kapoor, "An overview of lithium-ion batteries for electric vehicles," *IEEE*, pp. 230–235, 2012.

[4] H. Askari, A. Khajepour, M. B. Khamesee, and Z. L. Wang, "Embedded self-powered sensing systems for smart vehicles and intelligent transportation," *Nano Energy*, vol. 66, p. 104103, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2211285519308109

[5] M. Dendaluce Jahnke, F. Cosco, R. Novickis, J. Pérez Rastelli, and V. Gomez-Garay, "Efficient neural network implementations on parallel embedded platforms applied to real-time torque-vectoring optimization using predictions for multi-motor electric vehicles," *Electronics*, vol. 8, no. 2, 2019. [Online]. Available: https://www.mdpi.com/2079-9292/8/2/250

[6] J. Chen, A. Behal, and C. Li, "Active battery cell balancing by real time model predictive control for extending electric vehicle driving range," *IEEE Transactions on Automation Science and Engineering*, accepted June 2023.

[7] M. Einhorn, W. Roessler, and J. Fleig, "Improved performance of serially connected li-ion batteries with active cell balancing in electric vehicles," *IEEE Transactions on Vehicular Technology*, vol. 60, no. 6, pp. 2448–2457, 2011.

[8] J. Huang, D. Shi, and T. Chen, "Event-triggered state estimation with an energy harvesting sensor," *IEEE Transactions on Automatic Control*, vol. 62, no. 9, pp. 4768–4775, 2017.

[9] J. Chiasson and B. Vairamohan, "Estimating the state of charge of a battery," *IEEE Transactions on Control Systems Technology*, vol. 13, no. 3, pp. 465–470, April 2005.

[10] A. Pozzi, M. Zambelli, A. Ferrara, and D. M. Raimondo, "Balancing-aware charging strategy for series-connected lithium-ion cells: A nonlinear model predictive control approach," *IEEE Transactions on Control Systems Technology*, vol. 28, no. 5, pp. 1862–1877, 2020.

[11] F. S. Hoekstra, L. W. Ribelles, H. J. Bergveld, and M. Donkers, "Real-time range maximisation of electric vehicles through active cell balancing using model-predictive control," in *2020 American Control Conference*, Denver, CO, July 1–3, 2020, pp. 2219–2224.