SoftwareX 15 (2021) 100789

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

PND: Physics-informed neural-network software for molecular dynamics applications

Taufeq Mohammed Razakh [a,b], Beibei Wang [a,c], Shane Jackson [a,c], Rajiv K. Kalia [a,b,c,d], Aiichiro Nakano [a,b,c,d,e], Ken-ichi Nomura [a,d,*], Priya Vashishta [a,b,c,d]

[a] Collaboratory for Advanced Computing and Simulations, University of Southern California, Los Angeles, CA 90089-0242, USA
[b] Department of Computer Science, University of Southern California, Los Angeles, CA 90089-0781, USA
[c] Department of Physics & Astronomy, University of Southern California, Los Angeles, CA 90089-0484, USA
[d] Department of Chemical Engineering & Materials Science, University of Southern California, Los Angeles, CA 90089-12111, USA
[e] Department of Quantitative & computational Biology, University of Southern California, Los Angeles, CA 90089-2910, USA

ARTICLE INFO

Article history:
Received 21 April 2021
Received in revised form 27 July 2021
Accepted 28 July 2021

Keywords:
Molecular dynamics
Machine learning
Differential equation solver

ABSTRACT

We have developed PND, a differential equation solver software based on a physics-informed neural network (PINN) for molecular dynamics simulators. Based on automatic differentiation technique provided by PyTorch, our software allows users to flexibly implement equation of motion for atoms, initial and boundary conditions, and conservation laws as loss function to train the network. PND comes with a parallel molecular dynamic engine in order to examine and optimize loss function design, and different conservation laws and boundary conditions, and hyperparameters, thereby accelerating PINN-based development for molecular applications.

Code metadata

Current Code version	1.0
Permanent link to code/repository used of this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00077
Legal Code License	MIT
Code Versioning system used	Github
Software Code Language used	C++
Compilation requirements, Operating environments & dependencies	Linux, MacOS
If available Link to developer documentation/manual	https://usccacs.github.io/PND/annotated.html
Support email for questions	razakh@usc.edu

Software metadata

Current software version	1.0
Permanent link to executables of this version	https://github.com/USCCACS/PND
Legal Software License	MIT
Computing platform/Operating System	Linux, MacOS
Installation requirements & dependencies	GNU C++14, PyTorch 1.4
Link to user manual	https://github.com/USCCACS/PND#readme
Support email for questions	razakh@usc.edu

1. Motivation and significance

Molecular dynamics (MD) simulation is a vital tool in physics, chemistry, biomedical and materials research, because it provides

* Corresponding author at: Collaboratory for Advanced Computing and Simulations, University of Southern California, Los Angeles, CA 90089-0242, USA.
E-mail address: knomura@usc.edu (Ken-ichi Nomura).

https://doi.org/10.1016/j.softx.2021.100789

atomistic-level insights into material processes [1–3]. While the size of the simulation system and attainable temporal scale are usually in trade-off, efficient algorithms based on divide-and-conquer (DC) approach [4] have enabled multi-trillion particle simulations [5]. On the other hand, extending the accessible temporal scale by atomistic simulations has been an active research topic. Various accelerated MD methods, such as hyperdynamics and parallel replica method, have successfully increased the accessible temporal scale by several orders of magnitude [6]. However, these time-accelerated algorithms often rely on specific characteristics of the potential energy surface (PES) of each material, thus limiting their applicability to broader problems. In MD simulation, Newton's equation of motion, a multi-dimensional coupled ordinary differential equation (ODE), is integrated by a numerical solver to obtain the trajectory of atoms. Time-discretization unit Δt for numerical integration is dictated by physical properties of the target system, such as pressure, temperature and vibrational frequency. A general solver that increases Δt in a material-oblivious manner has a great potential to enhance modeling capability.

Due to remarkable successes in machine learning (ML), neural network (NN) is attracting great attentions as a novel differential equation (DE) solver. Kadupitiya et al. [7] used recurrent neural networks (RNN) to predict atomic trajectories instead of directly solving equation of motions of atoms. RNN is suitable to find patterns in sequence data such as particle motion subjected to a harmonic interaction and atoms oscillate at crystal lattice positions as demonstrated in their work. Another promising approach is physics-informed neural network (PINN), a branch of deep learning that has been attracting great attention as a DE solver recently. Unlike complex network architectures like in RNN, PINN employs rather simple network architecture such as a few layers of feedforward network but augmented by physical laws. With PINN, artificial neural network is trained by minimizing the difference between the given equation of motion and the predicted atomic trajectory in phase space as the loss function. Unlike RNN, PINN does not require preexisting pattern in the target system and is more suitable for materials in liquid or gas phases.

Dissanayake et al. [8] have proposed a NN-base DE solver, in which physics laws (*i.e.* target DEs together with initial and boundary conditions) are encoded as NN training framework. The physics-informed NN (PINN) has been successfully applied to solve many DEs, including heat equation [9], Burger equation [10], Navier Stoke's equation [11], Schrödinger equation [12], Hamilton's equation of motions [13] and other applications [14–16]. Employing DC approach, a recently proposed parallel PINN [17] introduces a light-weight, coarse-grained DE solver on top of multiple fine-grained solvers in smaller time segments that are solved concurrently to achieve larger Δt with negligible runtime increase due to model training. Growing developments of PINNs, such as variational PINNs [18] and nonlocal PINN [19], hold a great promise for DE-based modeling for scientific and engineering applications.

Despite these promising developments, PINN for MD simulations is in its infancy, for which enormous developmental work is required. For example, simulated systems so far are relatively small, and applicability of PINN to practical MD simulations remain unclear due to the huge hyper-parameter space. To this end, we have developed a simulation software PND, which is a portable, efficient and easy-to-use PINN-MD simulation software based on C++, PyTorch C++ Frontend, and Message Passing Interface (MPI) library. PND comes with a scalable parallel MD engine that allows users to examine different model training scenarios, loss functions and hyperparameters, thereby facilitating rapid development of PINN-MD algorithms.

2. Software description

The core of the PND software is developed PyTorch C++ Frontend, providing users with high-performance, low latency and multithreading support, as well as Fortran and C binding capability. These features are particularly suitable for many leadership-scale high performance computing (HPC) software such as RXMD [20] and QXMD [21] to incorporate PND in their framework. Equipped with the automatic differentiation capability, PND allows users to implement standard initial and boundary conditions, as well as many forms of physics-based constrains including conservation laws for energy and linear momentum, and the principle of least action [22]. Physical law is implemented as the loss function to improve model prediction quality and achieve a faster convergence of the loss function. For example, the law of linear momentum conservation is implemented as $A(\sum_{i=1}^{N} p_{i\alpha})$ where $p_{i\alpha}$ is the momentum of ith atom in Cartesian coordinate α, A is its weight in the total loss function. These constraints may be used to guide model training as heuristics to achieve a faster convergence of the loss function. To reduce the initial barrier for users to develop and integrate PND into their MD software, we provide a scalable MD engine and demonstrate a boundary-condition problem solver with many constraints. Users can define the loss function to evaluate mean squared error (MSE), which is essentially the set of DEs governing the evolution of the system and constraints from boundary conditions. Solution of the DE is obtained by minimizing the MSE during ANN training, see Fig. 1. By inheriting the base class PND into the user's workspace (referred as ScratchPad), users can implement the laws for system in the form of a PDE through the interface of the superclass. The feed forward NN predicts atomic positions and velocities, which get passed to the MD engine to calculate terms which fit into the systems PDE such as potential energy, total momentum. Implementing the PDE takes place by overriding the loss function method of the base class, thereby easily integrating the MD Engine layer and PINN training layer in their workspace. The MSE calculation uses the sum of the mean-squared PDE residuals with automatic differentiation and the mean-squared error in initial and boundary conditions. Training of the PINN is carried out by minimizing the mismatch with respect to the NN parameters. For the given NN, t represents the time vector for which MD states need to be determined, σ the activation function and $\vec{Q}$ the predicted system states over t.

2.1. Directory structure and source code organization

In this section, we describe main classes and functions that support the core functionalities of PND. PND software is organized in three directories – ***MD_Engine***, ***Source*** and ***Example***. The header and source files to train NN and solving equations of motion are stored in ***Source*** directory. ***MD_Engine*** directory contains a scalable MD software (**pmd**), which may optionally be replaced by user's own MD engine. The ***ScratchPad*** class inherits the **PND** class and exposes member functions for users to customize the loss function. The ***ScratchPad*** source code is stored in ***Example*** directory to demonstrate techniques to perform MD simulation. An illustrative example is presented in Section 4.

Below, we provide a list of the key classes, member functions and source codes with their brief explanation.

Source/pnd.in The number of neurons in the NN and training options such as epochs and learn rate are specified in this file.

Source/pnd.cpp: This source code defines functions which the user can use to structure and train a NN for the purpose of solving DE's closely resembling an MD system over time steps. The functionality of the source code is encapsulated into the

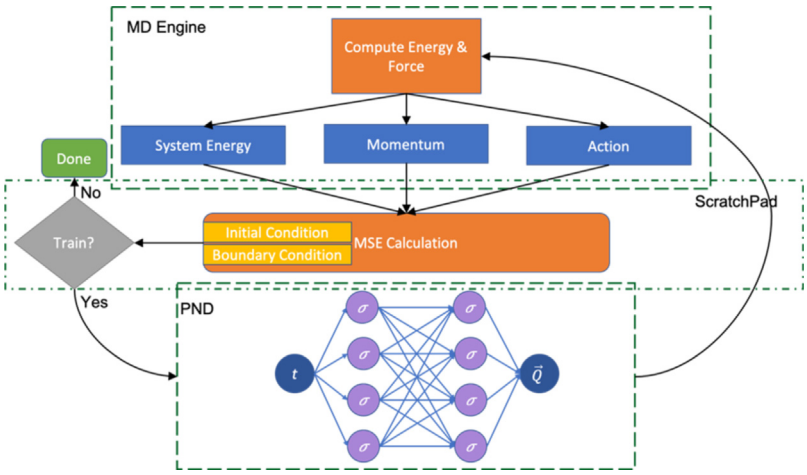

Fig. 1. Schematic of PND workflow in molecular dynamics application.

PND class. By instantiating this class, users can access instance variables to store the training parameters. Instance methods are designed to update the instance's training parameters, thus only one instance of the class may be used for all training epochs. The PND class exposes virtual functions that implements the loss function. These functions are meant to overwritten by the user allowing flexible loss function designs depending on users' needs and simulation system.

Source/pnd.hpp: A header file with declarations for functions and variables of the class that gets defined in Source/pnd.cpp.

MD_Engine/pmd.cpp: In this source code, we define the class Atom and Subsystem. Collectively, these classes provide functions and data structures to help with spatial decomposition and mapping the subsystem onto processors.

MD_Engine/pmd.hpp: Header file with declarations for classes that are defined in MD_Engine/pmd.cpp.

MD_Engine/pmd.in: Input parameters of MD system are defined in this file.

Example/ScratchPad.cpp: A sample implementation to interface with the class — PND as well as the MD engine.

CMakeLists.txt: PND employs CMake build system. User needs to add the directories containing the MD and PND code as a search path for include files of our target.

2.2. Loss function, optimizer, and training driver functions

The derivative of the loss function with respect to the network input parameters are passed to the optimizer function ***UpdateParamsNadam*** before the weights and biases are adjusted to train the network. PND provides a predefined optimizer that implements the Nestrov and Adam algorithm (NADAM) [23] to train the NN. Advanced users familiar with PyTorch may replace this function with various optimizer algorithms provided natively under the ***torch.optim*** class. Orchestrating this cycle of the loss calculation and network parameter optimization over training epochs is handled by the ***mainTrain*** function. The initial and boundary conditions as well as the systems energy from ground truth are passed down to this function as a tuple and consequently made available to the function calculating the MSE.

The loss function ***Loss*** **(*params, icfs, energy*)** takes the NN parameters (***params***), initial and boundary conditions (***icfs***) and various physical constrains, including the total energy (***energy***). Parameters of the NN are trained by minimizing a user-defined loss function, which incorporates physically-informed constraints such as initial and boundary conditions and the conservation laws for energy and momentum. The returned value is the derivative of the error defined with respect to the params. For d-dimensional configuration space for T time steps, the initial position vector is represented by $\vec{q}_0$, the final position vector is represented by $\vec{q}_T$, the initial velocity vector is represented by $\vec{v}_0$ and the final velocity vector is represented by $\vec{v}_T$. A system containing N_p atoms is represented by a vector of size $D = 2 \times d \times N_\mathrm{p}$. Output of the NN is represented by $\vec{Q}\,(params) \in \mathbb{R}^{D\times T}$ for all time steps, or $\vec{Q}(params, t)$ for the tth time step. We use a compact data layout such that the first half outputs of the NN $\vec{Q}_1(params)$ are vectors for the positions and the second half $\vec{Q}_2(params)$ are the velocities. We found that a pre-training step using a rough estimate of atom trajectory, such as linear interpolation of positions and velocities helps the model training step. The use of the pre-training step also provides a better control and check to avoid unwanted atomic positions such as overlapping atom positions before entering the main training step. Eqs (1) and (2) show an example of pre-training and main-training loss functions in a boundary-condition problem.

$$\begin{aligned} MSE^{\text{pre train}} = &\left(\vec{Q_1}\,(params, 0) - \vec{q}_0\right)^2 \\ &+ \left(\vec{Q_1}\,(params, T) - \vec{q}_T\right)^2 \\ &+ \left(\vec{Q_2}\,(params, 0) - \vec{v}_0\right)^2 \\ &+ \left(\vec{Q_2}\,(params, T) - \vec{v}_T\right)^2 \end{aligned} \tag{1}$$

$$\begin{aligned} MSE^{\text{main train}} = &\left(\vec{Q_1}\,(params, 0) - \vec{q}_0\right)^2 \\ &+ \left(\vec{Q_1}\,(params, T) - \vec{q}_T\right)^2 \\ &+ \left(\vec{Q_2}\,(params, 0) - \vec{v}_0\right)^2 \\ &+ \left(\vec{Q_2}\,(params, T) - \vec{v}_T\right)^2 \\ &+ (predictedenergy - energy)^2 \end{aligned} \tag{2}$$

3. Illustrative example

In this section, we present an illustrative example of PND use case for a simple face center cubic (FCC) crystal described by Lennard-Jones (LJ) interatomic potential [1]. We evaluate the

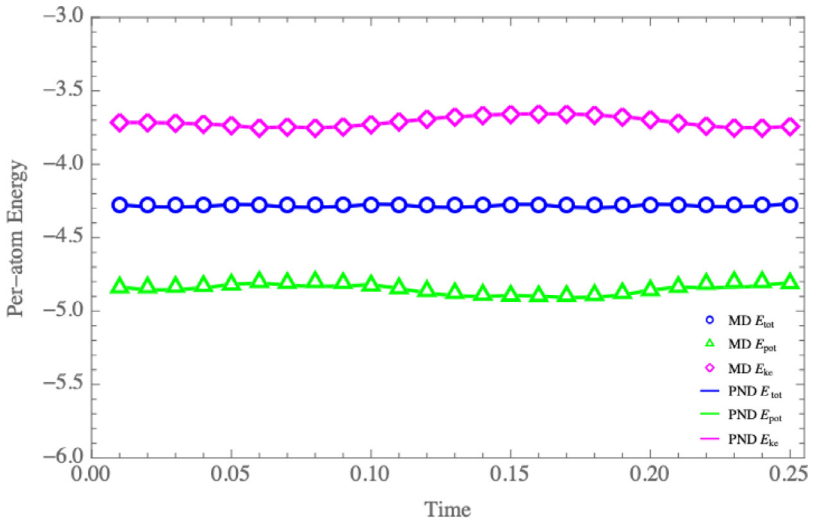

Fig. 2. Time evolution of energies by PND (solid lines) and ground-truth MD (markers) for 32 Ar atoms. Total, potential and kinetic energies are shown in blue, green and magenta, respectively. All three energies predicted by PND match well with the ground-truth MD simulation. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

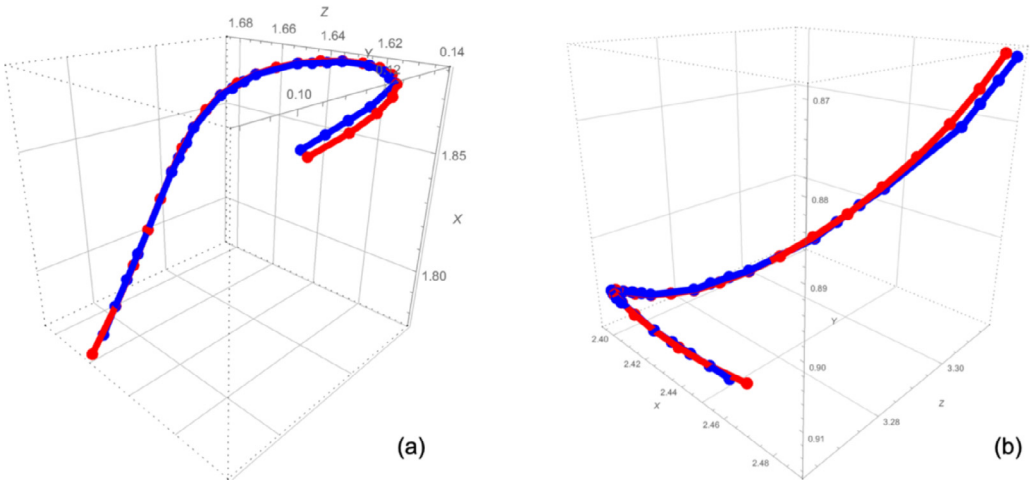

Fig. 3. (a) and (b) are trajectories of randomly selected two atoms. Red and blue trajectories are taken from ground-truth MD simulation and PND, respectively. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

PND predictive performance by comparing the energies and atom trajectories of the simulated and predicted MD systems.

The simulation was carried out for Argon atoms, starting from FCC-crystalline atomic positions with density 0.8 and random velocities corresponding to an initial temperature of 0.7 and using the time step of 0.01. All parameters are in normalized LJ unit. Fig. 2 demonstrates that PINN-MD conserves the total energy, while closely reproducing the ground-truth kinetic and potential energies obtained by a conventional ODE solver over the entire time steps. Fig. 3 shows a few typical atom trajectories, which demonstrate close match to the ground-truth trajectories. An input file of this illustrative example is provided in the source repository within the *MD_Engine* directory.

4. Impact

The design goal of PND is to facilitate development of novel PINN-based algorithms in molecular modeling software that may realize larger temporal resolution per single model training than traditional time integrator such as velocity-Verlet algorithm. The use of neural network to solve differential equations has opened up a novel research area and shown its great potential in many engineering fields. It is also expected to play an important role in materials simulation such as a novel time integration solver in MD. A number of studies [17,22,24] have proven that the prediction accuracy and algorithmic concurrency may be further enhanced by exploiting the conservation laws specific in each simulation. However, the potential of the PINN-MD approach has not been fully unleashed due to the inevitable large degrees-of-freedom in MD simulations, and more importantly, the lack of an open-source software that allows simultaneously examination the effect of physic-informed loss function, model training performance, and model prediction accuracy. The PND software satisfies the urgent demand and serves as a crucial role for the development of PINN-based molecular simulation methodology.

5. Conclusions

In summary, we have developed PND software, an opensource development platform of PINN-based ODE solvers for MD simulations. In PND, a neural network is trained to directly predict atomic position and momentum on prescribed time grids instead of performing conventional time-stepping algorithms. PND comes with the flexible Scratchpad design and a parallel MD

simulation engine, allowing PND users to implement custom loss functions, initial and boundary conditions, and conservation laws best suited for their application without requiring a third-party application. Employing PyTorch C++ Frontend for the core of the software, PND supports native multithreading, low-latency computation, GPU offloading, and Fortran and C binding capability, thus enjoying the full-fledged ML features without sacrificing performance. The system setup and input parameters are provided with the code on the Github repository https://github.com/USCCACS/PND.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was supported as part of the Computational Materials Sciences Program funded by the U.S. Department of Energy, Office of Science, Basic Energy Sciences, under Award Number DE-SC0014607.

References

[1] Rahman A. Correlations in the motion of atoms in liquid argon. Phys Rev 1964;136:A405–11. http://dx.doi.org/10.1103/PhysRev.136.A405.
[2] Wang B, Nakano A, Vashishta PD, Kalia RK. Nanoindentation on monolayer MoS2 kirigami. ACS Omega 2019;4:9952–6. http://dx.doi.org/10.1021/acsomega.9b00771.
[3] Jackson S, Nakano A, Vashishta P, Kalia RK. Electrostrictive cavitation in water induced by a SnO2 nanoparticle. ACS Omega 2019;4:22274–9. http://dx.doi.org/10.1021/acsomega.9b00979.
[4] Nakano A, Kalia RK, Nomura K, Sharma A, Vashishta P, Shimojo F, et al. A divide-and-conquer/cellular-decomposition framework for million-to-billion atom simulations of chemical reactions. Comput Mater Sci 2007;38:642–52. http://dx.doi.org/10.1016/j.commatsci.2006.04.012.
[5] Tchipev N, Seckler S, Heinen M, Vrabec J, Gratl F, Horsch M, et al. TweTriS: Twenty trillion-atom simulation. Int J High Perform Comput Appl 2019;33:838–54. http://dx.doi.org/10.1177/1094342018819741.
[6] Perez D, Uberuaga BP, Voter AF. The parallel replica dynamics method – Coming of age. Comput Mater Sci 2015;100:90–103. http://dx.doi.org/10.1016/j.commatsci.2014.12.011.
[7] Kadupitiya JCS, Fox GC, Jadhao V. Deep learning based integrators for solving Newton's equations with large timesteps. 2020, arxiv preprint arXiv:2004.06493.
[8] Dissanayake MWMG, Phan-Thien N. Neural-network-based approximations for solving partial differential equations. Commun Numer Methods Eng 1994;10:195–201. http://dx.doi.org/10.1002/cnm.1640100303.
[9] He H, Pathak J. An unsupervised learning approach to solving heat equations on chip based on Auto Encoder and Image Gradient, arxiv preprint arXiv:2007.09684.
[10] Lu F. Data-driven model reduction for stochastic Burgers equations. Entropy 2020;22. http://dx.doi.org/10.3390/e22121360.
[11] Jin X, Cai S, Li H, Karniadakis GE. NSFnets (Navier–Stokes flow nets): Physics-informed neural networks for the incompressible Navier–Stokes equations. J Comput Phys 2020;426:109951. http://dx.doi.org/10.1016/j.jcp.2020.109951.
[12] Kapetanović AL, Poljak D. Solution of the Schrödinger equation using a neural network approach. In: 2020 Int. Conf. Software, Telecommun. Comput. Networks. 2020, p. 1–5. http://dx.doi.org/10.23919/SoftCOM50211.2020.9238221.
[13] Mattheakis M, Sondak D, Protopapas P. Hamiltonian neural networks for solving differential equations, arxiv preprint arXiv:2001.11107.
[14] Lagaris IE, Likas A, Fotiadis DI. Artificial neural networks for solving ordinary and partial differential equations. IEEE Trans Neural Netw 1998;9:987–1000. http://dx.doi.org/10.1109/72.712178.
[15] Raissi M, Perdikaris P, Karniadakis GE. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. J Comput Phys 2019;378:686–707. http://dx.doi.org/10.1016/j.jcp.2018.10.045.
[16] Zhang D, Lu L, Guo L, Karniadakis GE. Quantifying total uncertainty in physics-informed neural networks for solving forward and inverse stochastic problems. J Comput Phys 2019;397:108850. http://dx.doi.org/10.1016/j.jcp.2019.07.048.
[17] Meng X, Li Z, Zhang D, Karniadakis GE. PPINN: Parareal physics-informed neural network for time-dependent PDEs. Comput Methods Appl Mech Engrg 2020;370:1–17. http://dx.doi.org/10.1016/j.cma.2020.113250.
[18] Kharazmi E, Zhang Z, Karniadakis GEM. VPINNs: Variational physics-informed neural networks for solving partial differential equations, arxiv preprint arXiv:1912.00873.
[19] Haghighat E, Bekar AC, Madenci E, Juanes R. A nonlocal physics-informed deep learning framework using the peridynamic differential operator, arxiv preprint arXiv:2006.00446.
[20] Nomura K, Kalia RK, Nakano A, Rajak P, Vashishta P. RXMD: A scalable reactive molecular dynamics simulator for optimized time-to-solution. SoftwareX 2020;11:100389. http://dx.doi.org/10.1016/j.softx.2019.100389.
[21] Shimojo F, Fukushima S, Kumazoe H, Misawa M, Ohmura S, Rajak P, et al. QXMD: An open-source program for nonadiabatic quantum molecular dynamics. SoftwareX 2019;10:100307. http://dx.doi.org/10.1016/j.softx.2019.100307.
[22] Amirikian BR, Lukashin AV. A neural network learns trajectory of motion from the least action principle. Biol Cybern 1992;66:261–4. http://dx.doi.org/10.1007/BF00198479.
[23] Dozat T. Incorporating Nesterov Momentum into Adam. ICLR Workshop; 2016.
[24] Jagtap AD, Kharazmi E, Karniadakis GE. Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems. Comput Methods Appl Mech Engrg 2020;365:113028. http://dx.doi.org/10.1016/j.cma.2020.113028.