Dynax: A Differentiable Dynamic Energy Simulator for Inverse Inference, Optimal Control and End-to-End Learning

Yangyang Fu, Zheng O’Neill
J. Mike Walker ’66 Department of Mechanical Engineering, Texas A&M University, College Station, TX, USA

Abstract
This paper presents a differentiable simulator for dynamic energy systems to support gradient-based tasks, such as inverse inference, optimal control and end-to-end learning. The differentiable simulator is built on an open-source auto-differentiation (AD) platform specifically designed for deep learning communities. With the inherent differentiability and parallelization across hardware accelerators, this differentiable simulator can benefit the digital twin era by leveraging modern computation hardware accelerators to perform real-time inference and learning-based control. The source code are presented at https://github.com/yangyangfu/dynax.

Highlights
- A differentiable simulator with inherent differentiability and parallelization for dynamic energy systems is implemented
- Gradient-based simulation tasks such as inverse inference, optimal control and end-to-end learning can benefit from this simulator

Introduction
Dynamic energy systems, such as building energy systems, encompass mechanical engineering, electrical engineering, and thermal transfer principles. These systems can be mathematically represented by a set of differential algebraic equations (DAEs) or ordinary differential equations (ODEs). Solving these equations on a computer relies on numerical methods, including integration and iterative techniques. These simulators were primarily developed for forward simulation applications, wherein the system of equations is solved incrementally through time, generating time-series data for a specific duration. This forward simulation capability has proven to be invaluable in both academic and industrial settings for conducting energy analyses and evaluating control strategies (Fu et al. (2019, 2020, 2021)).

However, as the world is moving fast towards the digital twin era with the support of advanced modern hardware accelerators such as multi-core CPUs, GPUs and TPUs, these simulators start to expose various limitations especially when it comes to optimization- and control-related applications. First, with a primal focus on forward simulation, these simulators may not fully and efficiently align with the requirement of digital twin applications, which often demand real-time feedback and bidirectional interactions between the virtual and physical systems. For example, digital twins are often employed for control and optimization purposes, where the goal is to adapt the system behavior based on real-time data and analysis, and thus require continuous model updates. Forward simulation often assumes a fixed model structure and do not automatically adapt to evolving system parameters. This limitation can hinder the accuracy and effectiveness of the digital twin (Negri et al. (2017)).

Second, traditional simulators may struggle to keep up with the increased computation demands of digital twins as they have limited support for parallel computation in multi-core accelerators. At the very beginning, these simulators were designed as a single-core or single-thread CPU-bound computer program. The acceleration of CPU-bound application has to rely on CPU performance that has been stagnant in the past years. For a single simulation task, recently, developers and engineers have started to leverage parallelism schemes to speed up the simulations on platforms with multi-core processors. For example, EnergyPlus since version 9.6 has a speedup up to 50 times for specific cases due to the utilization of various techniques such as code improvement, accuracy trade-offs, and parallelism (US Department of Energy (2023)). Parallelization in limited functional blocks such as heat balance and solar shading calculation brought promising speedups. Modelica can also parallelize models for simulation on multi-core platforms by utilizing special model structures (Sjölund et al. (2013); Elmqvist et al. (2015)). For example, Modelica compiler such as OpenModelica can recursively decompose the system equations into independent subsystems of equations and computing the subsystems in parallel using multi-cores, which relies on the fact that the original models have independent sub-models (Sjölund et al. (2013)). This is not always true for building energy systems where most equations are strongly coupled as algebraic equations. Breaking the algebraic equations for parallelization is still a research topic in the domain. What’s more, if the system is well-structured such as a set of discretized partial differential equations, then Modelica can run the simulation on GPUs (Elmqvist et al. (2015)). This type of parallelism implementation serves as patches to existing software framework, which may not be able to fully harness
the modern hardware accelerator.

Third, these simulators are expensive to generate derivatives for gradient-based optimization applications, such as inverse simulation, design optimization and optimal control etc. For example, when calibrating an EnergyPlus model, the gradients of the loss function (i.e., modeling errors) with respect to the tunable parameters are typically obtained through numerical differentiation using finite difference approximations, that is, evaluating the forward simulation multiple times with small perturbations to the relevant parameters to approximate the relevant gradients. This strategy is often impractical due to (1) the approximation inaccuracy caused by truncation errors and round-off errors; (2) the high computational burden of finite differencing when computing the gradient with respect to a large number of model parameters; and (3) the instability of numerical gradients over long time horizons (Baydin et al. (2018)). Due to the above limitations, derivative-free methods are typically preferred for EnergyPlus-based optimization such as OpenStudio Analysis Framework (Ball et al. (2020)) and JEPplus (Zhang and Korolija (2010)). Modelica community has developed an extension to support optimization modeling, known as Optimica (Ákesson (2008)). They obtain derivatives through numerical differentiation and/or symbolic derivatives leveraging built-in symbolic processing functionality. Optimica can also formulate the optimization problem on an existing auto-differentiation (AD) platform such as CasADi to leverage the advantages of obtaining derivatives through AD. However, CasADi is not a general-purpose AD tool (Andersson et al. (2019)), and primarily designed for optimization rather than modeling.

Forth, these simulators are not easy to be integrated with existing machine learning tools, especially with the development of physics-informed neural network or neural ODEs. For example, despite the rapid development of neural network models for building energy systems, it is difficult to incorporate these models into these simulators, or vice versa. Authors in (Thummerer et al. (2021)) leveraged the Functional Mock-up Interface to allow the integration of Modelica models as a Function Mock-up Unit into a neural network topology. The need of specialized software "bridges" may be a significant hindrance to the adoption of machine learning methods in practice.

One of the potential solutions to the above limitations is the **Differentiable Simulator** in a general-purpose AD platform, which has been a growing researching topic since 2010s. Differentiable simulators can be categorized into three types: differentiable physics simulator, differentiable neural simulator, and physics-informed differentiable simulator. While differentiable neural simulator are unique, such as neural networks for building load predictions, neural models for dynamic controls, etc, differentiable physics simulators are mostly tailed for specific domains, and most fall into robotics-related. Qiao et al. developed a differentiable simulator for soft articulated bodies in Pytorch (Qiao et al. (2021)). Deluca investigate the control of a differentiable medical ventilator environment and robotics in JAX (Gradu et al. (2021)). DiffTaichi was evaluated on 10 different physical simulators covering large-scale continuum and small-scale rigid body simulations using Taichi (Hu et al. (2019)). Degrave et al. developed a differentiable physics engine for robotics in the Theano framework (Degrave et al. (2019)). Julia community has active development for a physics-informed simulator, known as JuliaSim (Rackauckas et al. (2022)), which can formulate a physical system as a neural ODE problem, and solve it using their differentiable equation solvers. For example, JuliaSim provides acausal modeling and simulation blending with machine learning elements (Ma et al. (2021)). Similarly to Modelica, JuliaSim generates a DAE representation of the physical system models (e.g., building cooling systems), then transforms it to an ODE problem, which could be solved by various numerical integrators. To speed up the simulation, the component of a physical system can also be represented as a learnable ODE structure, trained from its physics-based counterpart. This allows to represent part of the system as physics-based ODEs, and the other as neural ODEs, which could then be solved together by selected numerical integrators. This functionality was mainly demonstrated for forward simulation.

By incorporating embedded physics, differentiable simulators can overcome the limitations present in traditional physics-based simulators:

- Differentiable simulators inherently support gradient-based tasks, such as gradient-based model identification, design optimization, and optimal control, etc. General gradient-based methods, such as policy gradient methods (e.g., Proximal Policy Optimization) and model predictive control (MPC), can benefit from differentiable simulators. For example, differentiable simulators can differentiate through the system dynamics, which allows the use of deterministic policies instead of stochastic policies that are typically used when non-differentiable dynamics are present (Belbute-Peres et al. (2018)). This can reduce sample and computational complexity since deterministic policies often require fewer samples to estimate gradients and are computationally simpler than stochastic policies.

- Differentiable simulators can be readily integrated with machine learning tools, enabling seamless interaction with neural networks. For instance, these simulators can function as a layer within a neural network, facilitating end-to-end learning of both environment parameters and control policies, as gradients can be easily obtained (Toussaint et al. (2018)). This seamless integration is particularly useful for tasks such as model identification and parametric programming, including explicit MPC. By incorporating differentiable simulators into neural network designs, they also support advanced modeling techniques like neural ODEs.
With physics-based dynamic models, differentiable simulators can be more data efficient for various tasks such as parameter estimation, control and optimization (Raissi et al. (2019)). Compared with learning a complex neural network model, learning a physics-based model typically reduces the amount of training data. What’s more, due to the inherent physics such as thermodynamics, the learnt differentiable models are usually explainable, and thus can make it easier to interpret the model’s behavior and predictions.

Differentiable simulators inherently support probabilistic programming by providing gradients of the model with respect to model parameters. The gradients can improve the efficiency and convergence of inference algorithm such as Hamiltonian Monte Carlo (Davidiwicz et al. (2021)), facilitate end-to-end learning of probabilistic models by jointly optimizing model parameters and latent variables (Belbute-Peres et al. (2018)), construct guided proposal distributions by using the gradients to inform the sampling process, improving the efficiency of sampling-based inference methods like importance sampling or Metropolis-Hastings (Liu and Wang (2016); Atchade (2011)), and enhance variational inference by allowing more efficient approximation of the true posterior distribution (Middleton et al. (2020)).

Structure
In this paper, we proposed an end-to-end differentiable dynamics simulator for the learning and control of dynamic energy systems. The framework was developed in Python, utilizing the latest advancements of the JAX library (Bradbury et al. (2018)), which is an AD and accelerated linear algebra platform specifically designed for deep learning applications. JAX provides composable transformations of Python+NumPy programs, enabling users to easily perform AD, vectorization, parallelization and just-in-time compilation using built-in primitives such as grad, vmap, pmap, and jit, respectively. JAX runtime has been shown ~100 times faster than pure Python (Frostig et al. (2018)), and can be ~6 times faster than Pytorch on the same task (Hu et al. (2019))). The rest of this paper is organized as follows: we first introduce the proposed differentiable dynamic simulator in JAX, then demonstrate its capability of performing an inverse inference from given data, integrating with MPC designs, and supporting end-to-end learning of both environment dynamics and optimal control policy. The conclusion and future work are finally discussed at the end of the paper.

Differentiable Simulator in JAX
In this paper, we present a new numerical library, Dynax, built on JAX to realize physics-based differentiable dynamic system modeling, including forward simulation, inverse inference and optimal control. The source code are also present at https://github.com/yangyangfu/dynax. The major modules in the library are summarized as follows:

- **System**: this module defines the ODE/DAE representation of the dynamic systems. Three basic classes are implemented, including traditional ODE system and index-1 DAE system, and neural ODE system. The module mainly leverages the neural network library Flax (Heek et al. (2023)) to perform parameterized modeling using physics.

- **Solver**: this module implements common numerical solvers in JAX for the ODE/DAE systems, including numerical integrators such as implicit/explicit Euler method, and iterative methods such as Newton method. These solver utilizes JAX syntax to support backpropagation of gradients especially when model rollouts are required in the loss function calculation. For instance, in a typical MPC approach, the dynamic system needs to predict an N-step future response and associated losses. To optimize the control inputs, the gradient of the loss function must be backpropagated to the initial step, forming the basis for solving the optimization problem.

- **Controller**: this module implements various controllers, including traditional control agent such as PID, and optimal control agents such as Differentiable Dynamic Programming, MPC, and model-based deep reinforcement learning (DRL). All these optimal control can benefit from a differentiable simulator.

- **Problem**: this module defines the problem to be solved, including forward simulation, inverse inference and optimal control. Forward simulation requires the definition of a system, its numerical solver, and other simulation settings such as start/end time. Inverse inference requires a predefined forward problem, and parameters that need gradients for inference based on given measurement data. Control problem requires a forward simulation problem, and a controller such as PID, MPC or DRL from the controller module.

- **DataLoader**: the data required for a dynamic system typically include system disturbances (i.e., weather for buildings), and measurement data for inverse inference of model parameters, etc. With a standard data loader, the data required for a specific problem can be standardized.

Inverse Inference
This section describes the capability of supporting inverse simulation using the differentiable dynamic models. A typical application is the model calibration of dynamic systems. We formulated the following inverse simulation problem to identify the unknown system model parameters from given time series data.
We set up an inverse simulation problem by obtaining the unknown building properties such as resistance and capacitance against given measurement data, which is a very typical model calibration problem in building energy domain. The building temperature dynamics is modeled as a RC thermal network as shown in Figure 1, with unknown $R_s$, $C_s$ and initial states. To perform state estimation for unobservable states such as interior wall temperature, we incorporate a continuous-time Kalman Filter in the RC model formulation, which eventually lead to a set of ODEs. The measurement data was generated from a one-month EnergyPlus simulation.

The inverse problem minimized the future N-step modeling errors by searching Kalman Filter parameters and RCs. 75% of the data was used for training, and the rest for testing. For training, we randomly selected a batch of data (64 samples) for each parameter update. A common optimizer known as AdamW with exponentially decayed learning rate was used for optimization. We trained the problem for 100,000 episodes with a fixed learning rate of $1e-03$, and the inverse simulation took only $7.02 \pm 0.12$ seconds on a MacBook with 2.6 GHz 6-core Intel Core i7.

Figure 2 plots the final calibration results, which shows excellent matching: the mean absolute error and the coefficient of variation of the root mean squared error for the testing data are 0.41 °C and 2.6%, respectively, and those for the training data are 0.42 °C and 2.7%, respectively. This indicates that our approach is very efficient in solving the inverse problem for RCs. The model outputs are outside air temperature $T_{oa}$, external solar load $q_{sol,e}$, internal convective heat gain $q_{conv,i}$, and HVAC heat rate $q_{hvac,i}$. The model input is the zone air temperature $T_z$.

\[ g(y) = \sum \omega_i ||\hat{y} - y||^2 + \omega_j R(p, p, \bar{p}) \]

\[ s.t. \quad \hat{y} = f(x, \dot{x}, u, d, p) \]

\[ R(p, p, \bar{p}) = \text{Relu}(p - \bar{p}) + \text{Relu}(p - \bar{p}) \]

where the originally constrained optimization problem is formulated as an unconstrained optimization problem by introducing penalty function for violating given box-bound constraints. $p$ is the tunable parameters, $g$ is the loss function including two terms: the sum of squared modeling errors and the penalty term $R$ for constraining the tunable parameters within desired lower and upper bounds, $\bar{p}$ and $\bar{p}$ respectively. $\hat{y}$ and $y$ are the predicted and measured system outputs, and $\omega_i$ is a user-specific weights for balancing these two terms in the objective. With normalized data, $\omega_i$ can be set to 1 for both terms. The differentiable simulator is represented as $f$, which calculates the outputs $\hat{y}$ by solving the system models based on system states $x$, state derivatives $\dot{x}$, control inputs $u$, disturbance inputs $d$ and model parameters $p$. The regulation term is modeled as a Relu function that penalize the objective function when the searching is out of given bound. The gradient of the loss function $g(y)$ with respect to the tunable parameters $p$ can be obtained through the chain rule as follows.

\[
\frac{\partial g(\hat{y})}{\partial p} = \frac{\partial g(\hat{y})}{\partial \hat{y}} \frac{\partial f}{\partial p}
\]

which is "effortless" to obtain in the given differentiable simulator by backpropagating the loss along trajectory rollouts. The training data is prepared to have the following format: at time $t$, the sample is represented as $[(x^t, \dot{x}^t, u^t), (x^{t+1}, \dot{x}^{t+1}, d^{t+1})]$, which include the state, control actions and disturbances over the future N-steps. If the given data has $M$ samples, then after preparation, the dataset would have $M - N$ samples.

\[
\min_{u_t} J(x_t, u_t) = \sum_{k=1}^{N-1} \ell(x_k, u_k)
\]

\[
= \sum_{k=1}^{N-1} \omega_i \eta \ell(x_k, u_k) + \omega_j R_k(T_k, T, \bar{p}) + \omega_j R_k(u_k, u, \bar{p})
\]

\[
x_{t+1} = K(T_k)
\]

\[
T_{k+1} = f(x_k, \dot{x}_k, u_k, d_k, p_k)
\]
where \( t \) is the current time step for the MPC, \( k \) is the future time step until the prediction horizon \( N \). \( \eta \) is the system cooling efficiency as a constant, \( c \) is the energy price, \( \Delta t \) is the control interval (i.e., 15 minutes in this research), and \( T \) is the zone temperature prediction from the differentiable simulator. The system states in this problem include zone temperature, interior and exterior wall temperature, and a Kalman Filter \( K \) was used to estimate the interior and exterior wall temperature at the beginning of MPC. The sampling time and control interval was set to 15 minutes. We added a random Gaussian noise \( N(\mu, \sigma^2) \) to the measured zone temperature, representing the modeling error by the differentiable simulator. The above optimization problem is then solved by using the Newton Conjugate Gradient method (Wright et al. (1999)). We performed a one-week simulation with different prediction horizons, and the control performance were summarized as in Table 1.

With fine-tuning techniques, such as gradient clipping, used for long trajectory rollouts when the MPC prediction horizon is long, the MPC performance is quite stable as shown in Figure 3. Due to the relatively large thermal mass, the MPC with 24 hours ahead of planning \((N = 96)\) tends to use more energy during nights and less energy during days, to take full advantages of the TOU pricing structure.

**End-to-end Learning**

Differentiable physics-based simulators can benefit the learning-based control agents such as DRL to perform end-to-end learning by allowing gradients to flow through the entire learning pipeline, as the agent can optimize its policy, value function and the environment model jointly. This section demonstrates the end-to-end learning of a DDQN agent (Van Hasselt et al. (2016)) and a differentiable physics-based simulator for the aforementioned building. The end-to-end learning follows the general model-based DRL framework, known as, Dyna-Q, as presented in (Sutton and Barto (2018)).

The control intervals and objectives are the same as in MPC. The DRL observation space is designed as shown in Table 2, which contains historical measurements, current measurements and future disturbances through predictions. The time index is represented by the elapsing seconds of a day, starting from 0 second and ending at 86400 seconds. Key measurements at the current time step and in the past such as zone air temperature \( T_o \), outdoor air temperature \( T_e \), solar radiation \( q_s \) and power consumption \( P \) are assumed available by installed sensors. Predictions of near-future disturbances such as outdoor air temperature, solar radiation, energy prices are assumed to be perfect. In this paper, \( k \) in Table 2 is chosen as 4, which means one hour in the time span and leads to 26 observations in total for each time step. The control space of the HVAC cooling rate is equidistantly discretised into 100 intervals within \([0, 1]\), which represents the percentage of the nominal cooling rate for the system.

![Figure 3: MPC performance with N = 96](image)

Table 1: Control performance for MPC with different prediction horizons

<table>
<thead>
<tr>
<th>N</th>
<th>Cost [$]</th>
<th>Energy [kWh]</th>
<th>Max Violation [°C]</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>38.92</td>
<td>424.70</td>
<td>1.66</td>
</tr>
<tr>
<td>48</td>
<td>34.91</td>
<td>577.14</td>
<td>1.66</td>
</tr>
<tr>
<td>96</td>
<td>31.71</td>
<td>715.05</td>
<td>1.30</td>
</tr>
</tbody>
</table>

Table 2: DRL observation space design

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_o )</td>
<td>([0, 1252])</td>
<td>Zone temperature at time ( t ) [°C]</td>
</tr>
<tr>
<td>( T_e )</td>
<td>([0, 40])</td>
<td>Outdoor air temperature at time ( t ) [°C]</td>
</tr>
<tr>
<td>( q_s )</td>
<td>([0, 1000])</td>
<td>Solar radiation at time ( t ) [W/m²]</td>
</tr>
<tr>
<td>( P )</td>
<td>([0, 1500])</td>
<td>Power at time ( t ) [W]</td>
</tr>
<tr>
<td>( p_k )</td>
<td>([0, 1])</td>
<td>Energy price at next ( k ) steps from ( t ) [$/kWh]</td>
</tr>
<tr>
<td>( T_o(k+1) )</td>
<td>([0, 1252])</td>
<td>Zone air temperature at ( k+1 ) steps from ( t ) [°C]</td>
</tr>
<tr>
<td>( q_s(k+1) )</td>
<td>([0, 1000])</td>
<td>Solar radiation at ( k+1 ) steps from ( t ) [W/m²]</td>
</tr>
<tr>
<td>( P(k+1) )</td>
<td>([0, 1500])</td>
<td>Power at previous ( k ) steps from ( t ) [W]</td>
</tr>
</tbody>
</table>

Table 3: Hyper-parameters for DDQN agent

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>learning rate</td>
<td>1e-05</td>
</tr>
<tr>
<td>minibatch size</td>
<td>64</td>
</tr>
<tr>
<td>experience replay buffer size</td>
<td>20000</td>
</tr>
<tr>
<td>neural network size</td>
<td>[26, 256, 256, 256, 101]</td>
</tr>
</tbody>
</table>

The typical DDQN settings are shown in Table 3. The model used for planning in the model-based DDQN is the same differentiable model as in the reverse inference problem, except that the parameters are identified through the interactions with the environment through the end-to-end learning process. The planning step is set to 5 so that the Q-network is updated 5 more times with the simulated experience and only once with the real experience. To demonstrate the capability of the end-to-end learning process, we compared the performance of a model-free DDQN control and our model-based DDQN control. The model-free DDQN uses the same settings as in Table 3. Figure 4 shows the learning process of both the model-free DDQN and the differentiable simulator based DDQN on the studied building zone. With all the same parameters, model-based DDQN and model-free DDQN can both learn a similar final performance, but the model-based DDQN can speed up about 25 episodes.
Conclusion and Future Work

We presented a differentiable simulator for dynamic energy systems, which supports gradient-based inverse inference, gradient-based control, and end-to-end learning. Due to the ease of obtaining gradients of any customized cost functions with respect to the model parameters, the differentiable simulator inherently benefits gradient-based tasks. Also, due to the inherent support of modern hardware accelerators in the AD framework, many tasks such as inverse simulation, end-to-end learning can be parallelized and scale up with the power of accelerators. In the reverse inference of model parameters, we showed that with batched data design, the model inference problem can be parallelized across multiple cores. In the MPC case, we showed that the differentiability of the simulator can directly support Jacobian and Hessian matrix for gradient-based optimizers such as Newton variants. In the end-to-end learning, we showed that model-based DRL can be 4 times faster than its model-free counterpart, and learning a differentiable physics-based simulator requires small amount of data.

However, this work is still at preliminary stage. The following key issues need to be addressed for large-scale deployment: 1) a modeling tool is needed to construct the system-level models and compile the equations into the ODE/DAE format automatically. Such tools are available in other communities such as Modelica and FMI/FMU, whose ODE/DAE format might be directly used by this library; 2) benchmark tests are needed to demonstrate the benefits of using accelerators. We haven’t shown the run time of each case in this paper due to the significant amount of time required to improve the code, which will be considered in the next release.

References


Heek, J., A. Levskaya, A. Oliver, M. Ritter, B. Rondipierre, A. Steiner, and M. van Zee (2023). Flax: A neural network library and ecosystem for JAX.


