Universal Differential Equations for Scientific Machine Learning

Christopher Rackauckas[1,a,b], **Yingbo Ma**[c], **Julius Martensen**[d], **Collin Warner**[a], **Kirill Zubov**[e], **Rohit Supekar**[a], **Dominic Skinner**[a], **and Ali Ramadhan**[a]

[a]Massachusetts Institute of Technology; [b]University of Maryland, Baltimore; [c]Julia Computing; [d]University of Bremen; [e]Saint Petersburg University

This manuscript was compiled on January 14, 2020

In the context of science, the well-known adage "a picture is worth a thousand words" might well be "a model is worth a thousand datasets." Scientific models, such as Newtonian physics or biological gene regulatory networks, are human-driven simplifications of complex phenomena that serve as surrogates for the countless experiments that validated the models. Recently, machine learning has been able to overcome the inaccuracies of approximate modeling by directly learning the entire set of nonlinear interactions from data. However, without any predetermined structure from the scientific basis behind the problem, machine learning approaches are flexible but data-expensive, requiring large databases of homogeneous labeled training data. A central challenge is reconciling data that is at odds with simplified models without requiring "big data".

In this work we develop a new methodology, universal differential equations (UDEs), which augments scientific models with machine-learnable structures for scientifically-based learning. We show how UDEs can be utilized to discover previously unknown governing equations, accurately extrapolate beyond the original data, and accelerate model simulation, all in a time and data-efficient manner. This advance is coupled with open-source software that allows for training UDEs which incorporate physical constraints, delayed interactions, implicitly-defined events, and intrinsic stochasticity in the model. Our examples show how a diverse set of computationally-difficult modeling issues across scientific disciplines, from automatically discovering biological mechanisms to accelerating climate simulations by 15,000x, can be handled by training UDEs.

Scientific Machine Learning | Differentiable programming | Differential Equations | Machine Learning | Deep Learning

Recent advances in machine learning have been dominated by deep learning which utilizes readily available "big data" to solve previously difficult problems such as image recognition (1–3) and natural language processing (4–6). While some areas of science have begun to generate the large amounts of data required to train deep learning models, notably bioinformatics (7–11), in many areas the expense of scientific experiments has prohibited the effectiveness of these ground breaking techniques. In these domains, such as aerospace engineering, quantitative systems pharmacology, and macroeconomics, mechanistic models which synthesize the knowledge of the scientific literature are still predominantly deployed due to the inaccuracy of deep learning techniques with small training datasets. While these types of low-parameter models are constrained to be predictive by utilizing prior structural knowledge conglomerated throughout the scientific literature, the data-driven approach of machine learning can be more flexible and allow for dropping simplifying assumptions. The purpose of this work is to bridge the gap by merging the best of both methodologies while mitigating the deficiencies.

It has recently been shown to be advantageous to merge differential equations with machine learning. Physics-Informed Neural Networks (PINNs) utilize partial differential equations in the cost functions of neural networks to incorporate prior scientific knowledge (12). While this has been shown to be a form of data-efficient machine learning for some scientific applications, the resulting model does not have the interpretability of mechanistic models. On the other end of the spectrum, machine learning practitioners have begun to make use of scientific structures as a modeling basis for machine learning. For example, neural ordinary differential equations are initial value problems of the form (13–16):

$$u' = \mathrm{NN}_\theta(u, t) \qquad [1]$$

defined by a neural network NN_θ. Because the latent function is a universal approximator (UA), it follows that NN_θ can learn to approximate any sufficiently regular differential equation. However, the resulting model is defined in a latent space without direct incorporation of known mechanisms.

The Universal Approximation Theorem (UAT) demonstrates that sufficiently large neural networks can approximate any nonlinear function with a finite set of parameters (17–19). Our approach is to directly utilize mechanistic modeling simultaneously with this aspect of machine learning in order to allow for arbitrary data-driven model extensions. The objects of this semi-mechanistic approach, which we denote as Uni-

Significance Statement

Incorporation of prior scientific knowledge into learnable structures could allow for a new form of machine learning that is data-efficient: thereby side stepping the issue of acquiring large expensive datasets by utilizing the condensed knowledge of the scientific literature. This manuscript develops a novel learning framework, the universal differential equation, which allows for machine-learning-augmented scientific models. We showcase the ability to incorporate prior scientific knowledge and train accurate neural architectures with small data. We demonstrate how the method is interpretable back to mechanistic equations and how it can accelerate climate simulations by 15,000x. The broad applications coupled with a software implementation demonstrates a viable path for small-data scientific machine learning.

CR conceived of the idea, wrote the manuscript, and was a part of the development of each of the examples. CR and YM implemented the DiffEqFlux library and the adjoint implementations. JM implemented SInDy and the Lotka-Volterra example. RS implemented the Fisher-KPP example. CW and YM implemented the climate model example. AR carried out the simulations of the 3D Bossinesq equations. DS implemented the viscoelastic fluid example. KZ implemented the HJB example. Each contributed to the text and supporting information of the respective sections.

The authors declare no conflict of interest.

[1]E-mail: crackauc@mit.edu

versal Differential Equations (UDEs), are differential equation models where part of the differential equation contains an embedded UA, such as a neural network, Chebyshev expansion, or a random forest. As a motivating example, the universal ordinary differential equation (UODE):

$$u' = f(u, t, U_\theta(u, t)) \quad [2]$$

denotes a mechanistic deterministic model f with missing portions defined by some UA U_θ. Similar generalizations to incorporate process noise, delayed interactions, and physics-based constraints are given by embedding UAs into stochastic, delay, and differential-algebraic equations respectively. In this manuscript we generalize previous adjoint approaches to allow for efficiently training any model within this extended family of possible UDEs. We showcase how UDEs lead to more data-efficient and computationally-efficient approaches in a diverse set of applications, specifically:

- We recover governing equations from much lesser data than prior methods and demonstrate the ability to accurately extrapolate from a short time series.
- We discover the differential operator and nonlinear reaction term of a biological partial differential equation (PDE) from spatiotemporal data, demonstrating the interpretability of trained UDEs.
- We automate the discovery of fast, accurate, and physically-consistent surrogates to accelerate a climate simulation by 15,000x.
- We approximate closure relations in viscoelastic fluids to accelerate the simulation of a system of 6 differential-algebraic equations by 2x, showing that this methodology is also applicable to small-scale problems.
- We derive an adaptive method for automated solving of a 100-dimensional nonlinear Hamilton-Jacobi-Bellman PDE, the first adaptive method for this class of problems that the authors are aware of.

Efficient Training of Universal Differential Equations via Differentiable Programming

Training a UDE amounts to minimizing a cost function $C(\theta)$ defined on the current solution $u_\theta(t)$. One choice is the Euclidean distance $C(\theta) = \sum_i \|u_\theta(t_i) - d_i\|$ at discrete data points (t_i, d_i). When optimized with local derivative-based methods, such as stochastic gradient decent, ADAM (20), or L-BFGS (21), this requires the calculation of $\frac{dC}{d\theta}$ which by the chain rule amounts to calculating $\frac{du}{d\theta}$. Thus the problem of efficiently training a UDE reduces to calculating gradients of the differential equation solution with respect to parameters.

In certain special cases there exist efficient methods for calculating these gradients called adjoints (22–25). The asymptotic computational cost of these methods does not grow multiplicatively with the number of state variables and parameters like numerical or forward sensitivity approaches, and thus it has been shown empirically that adjoint methods are more efficient on large parameter models (26, 27). However, given the number of different families of UDE models we wish to train, we generalize to a differentiable programming framework with reverse-mode accumulation in order to allow for deriving on-the-fly approximations for the wide range of possible differential equation types.

Given a function $f(x) = y$, the pullback at x is the function:

$$B_f^x(y) = y^T f'(x), \quad [3]$$

where $f'(x)$ is the Jacobian J. We note that $B_f^x(1) = (\nabla f)^T$ for a function f producing a scalar output, meaning the pullback of a cost function computes the gradient. A general computer program can be written as the composition of discrete steps:

$$f = f^L \circ f^{L-1} \circ \ldots \circ f^1, \quad [4]$$

and thus the vector-Jacobian product can be decomposed:

$$v^T J = (\ldots((v^T J_L)J_{L-1})\ldots)J_1, \quad [5]$$

which allows for recursively decomposing a the pullback to a primitively known set of $\mathcal{B}_{fi}^x$:

$$\mathcal{B}_f^x(A) = \mathcal{B}_{f^1}^x\left(\ldots\left(\mathcal{B}_{f^{L-1}}^{x_{L-2}}\left(\mathcal{B}_{f^L}^{x_{L-1}}(A)\right)\right)\ldots\right), \quad [6]$$

where $x_i = \left(f^i \circ f^{i-1} \circ \ldots \circ f^1\right)(x)$. Implementations of code generation for the backwards pass of an arbitrary program in a dynamic programming language can vary. For example, building a list of function compositions (a tape) is provided by libraries such as Tracker.jl (28) and PyTorch (29), while other libraries perform direct generation of backward pass source code such as Zygote.jl (30), TAF (31), and Tapenade (32).

The open-source differential equation solvers of DifferentialEquations.jl (33) were developed in a manner such that all steps of the programs have a well-defined pullback when using a Julia-based backwards pass generation system. The full set of adjoint options, which includes continuous adjoint methods and pure reverse-mode AD approaches, is described in Supplement S1. The extension library DiffEqFlux.jl handles compatibility with the Flux.jl neural network library for efficient pullback implementations of embedded deep neural networks (also known as backpropogation), allowing for common deep architectures, such as convolutional neural networks and recurrent neural networks, to be efficiently used as the basis for a UDE. Using this approach, the solvers are capable of building asymptotically-efficient gradient calculations for training ML-embedded UDEs of the classes:

- Universal Ordinary Differential Equations (UODEs)
- Universal Stochastic Differential Equations (USDEs)
- Universal Delay Differential Equations (UDDEs)
- Universal Differential-Algebraic Equations (UDAEs)
- Universal Boundary Value Problems (UBVPs)
- Universal Partial Differential Equations (UPDEs)
- Universal Hybrid (Event-Driven) Differential Equations

as well as the combinations, such as stochastic delay differential equations, jump diffusions, and stochastic partial differential equations. A combination of over 300 solver methods cover the efficient training of stiff and non-stiff versions of each of these

equations, with support for adaptivity, high-order, automatic stiffness detection, Newton-Krylov implicit handling, and GPU compatibility. Thus together, semi-mechanistic UDEs of any form can embed machine learning models and be trained using this open-source library with the most effective differential equation solvers for that class of equations.

Knowledge-Enhanced Model Reconstruction of Biological Models

Automatic reconstruction of models from observable data has been extensively studied. Many methods produce non-symbolic representations by learning latent representations (34, 35) or through dynamic mode decomposition (DMD, eDMD) (36–39). Symbolic reconstruction of equations has utilized symbolic regressions which require a prechosen basis (40, 41), or evolutionary methods to grow a basis (42, 43). However, a common thread throughout much of the literature is that added domain knowledge constrains the problem to allow for more data-efficient reconstruction (44, 45). Here we detail how a UA embedded workflow can augment existing symbolic regression frameworks to allow for reconstruction from partially known models in a more data-efficient manner.

Automated Identification of Nonlinear Interactions with Universal Ordinary Differential Equations. The SInDy algorithm (46–48) finds a sparse basis $\boldsymbol{\Xi}$ over a given candidate library $\boldsymbol{\Theta}$ minimizing the objective function $\left\|\dot{\mathbf{X}} - \boldsymbol{\Theta}\boldsymbol{\Xi}\right\|_2 + \lambda \left\|\boldsymbol{\Xi}\right\|_1$ using data for $\dot{\mathbf{X}}$ generated by interpolating the trajectory data $\mathbf{X}$. Here we describe a UDE approach to extend SInDy in a way that embeds prior structural knowledge.

As a motivating example, take the Lotka-Volterra system:

$$\begin{aligned} \dot{x} &= \alpha x - \beta xy, \\ \dot{y} &= \gamma xy - \delta y. \end{aligned} \qquad [7]$$

Assume that a scientist has a short time series from this system but knows the birth rate of the prey x and the death rate of the predator y. With only this information, a scientist can propose the knowledge-based UODE as:

$$\begin{aligned} \dot{x} &= \alpha x - U_1(x, y), \\ \dot{y} &= -\delta y + U_2(x, y), \end{aligned} \qquad [8]$$

which is a system of ordinary differential equations that incorporates the known structure but leaves room for learning unknown interactions between the the predator and prey populations. Learning the unknown interactions corresponds training the UA $U : \mathbb{R}^2 \to \mathbb{R}^2$ in this UODE.

While the SInDy method normally approximates derivatives using a spline over the data points, here we have $U_\theta(x, y)$ as an estimator of the derivative for only the missing terms of the model and we can perform a sparse regression on samples from the trained $U_\theta(x, y)$ to reconstruct only the unknown interaction equations. As described in Supplement S2, we trained $U_\theta(x, y)$ as a neural network against the simulated data for $t \in [0, 3]$ and utilized a sequentially thresholded regression (46) on the neural network outputs to reconstruct the missing dynamical equations. Using a 10-dimensional polynomial basis extended with trigonometric functions, the sparse regression yields 0 for all terms except for the missing quadratic terms, directly learning the original equations in an interpretable form. Even though the original data did not

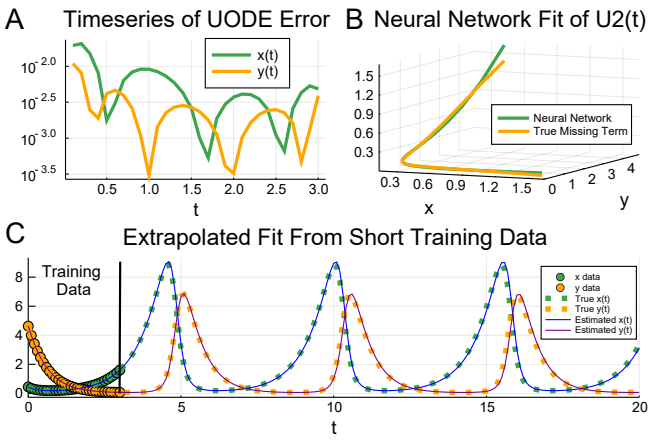

Fig. 1. Automated Lotka-Volterra equation discovery with UODE-enhanced SInDy. (A) The error in the trained UODE against $x(t)$ and $y(t)$ in green and yellow respectively. (B) The measured values of the missing term $U_2(x, y)$ throughout the time series, with the neural network approximate in green and the true value γxy in yellow. (C) The extrapolation of the knowledge-enhanced SInDy fit series. The green and yellow dots show the data that was used to fit the UODE, and the dots show the true solution of the Lotka-Volterra Equations 7 beyond the training data. The blue and purple lines show the extrapolated solution how the UODE-enhanced SInDy recovered equations.

contain a full period of the cyclic solution, the resulting fit is then able to accurately extrapolate from the short time series data as shown in Figure 1. Likewise, when attempting to learn full ODE with the original SInDy approach on the same trained data with the analytical derivative values, we were unable to recover the exact original equations from the sparse regression, indicating that the knowledge-enhanced approach increases the robustness equation discovery.

Reconstruction of Spatial Dynamics with Universal Partial Differential Equations. To demonstrate discovery of spatiotemporal equations directly from data, we consider data generated from the one-dimensional Fisher-KPP (Kolmogorov–Petrovsky–Piskunov) PDE (49):

$$\frac{\partial \rho}{\partial t} = r\rho(1 - \rho) + D\frac{\partial^2 \rho}{\partial x^2}, \qquad [9]$$

with $x \in [0, 1]$, $t \in [0, T]$, and periodic boundary condition $\rho(0, t) = \rho(1, t)$. Here ρ represents population density of a species, r is the local growth rate and D is the diffusion coefficient. Such reaction-diffusion equations appear in diverse physical, chemical and biological problems (50). To learn the generated data, we define the UPDE:

$$\rho_t = \mathrm{NN}_\theta(\rho) + \hat{D}\,\mathrm{CNN}(\rho), \qquad [10]$$

where NN_θ is a neural network representing the local growth term. The derivative operator is approximated as a convolutional neural network CNN, a learnable arbitrary representation of a stencil while treating the coefficient $\hat{D}$ as an unknown. We encode in the loss function extra constraints to ensure the learned equation is physically realizable, i.e. the derivative stencil must be conservative (the coefficients sum to zero), as described in Supplement S3. Figure 2 shows the result of training the UPDE against the simulated data, which recovers the canonical $[1, -2, 1]$ stencil of the one-dimensional Laplacian and the diffusion constant while simultaneously finding a neural representation of the unknown quadratic growth term.

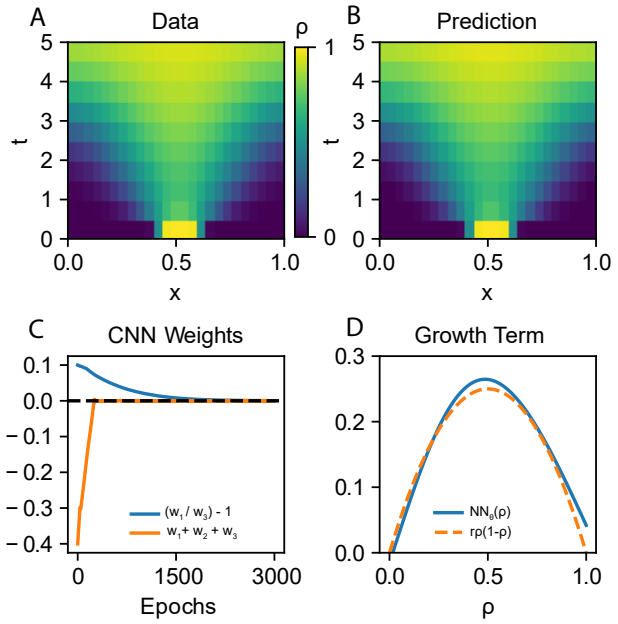

Fig. 2. Recovery of the UPDE for the Fisher-KPP equation. (A) Training data and (B) prediction of the UPDE for $\rho(x,t)$. (C) Curves for the weights of the CNN filter $[w_1, w_2, w_3]$ indicate the recovery of the $[1, -2, 1]$ stencil for the 1-dimensional Laplacian. (D) Comparison of the learned (blue) and the true growth term (orange) showcases the learned parabolic form of the missing nonlinear equation.

Thus, by utilizing the physical formulation of the model mixed with data-driven components, we arrive at a trained model which is directly interpretable and accurately describes the data generation process as a diffusion equation with a quadratic growth law. The authors are not aware of previous techniques that would give directly interpretable spatiotemporal models from a semi-blackbox fitting technique.

Accelerated Scientific Simulation with Automatically Constructed Closure Relations

Automated Discovery of Climate Model Parameterizations. As an example of directly accelerating existing scientific workflows, we focus on the Boussinesq equations (51). The Boussinesq equations are a system of 3+1-dimensional partial differential equations acquired through simplifying assumptions on the incompressible Navier-Stokes equations, represented by the system:

$$\begin{aligned} \nabla \cdot \mathbf{u} &= 0, \\ \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} &= -\nabla p + \nu\nabla^2\mathbf{u} + b\hat{z}, \\ \frac{\partial T}{\partial t} + \mathbf{u}\cdot\nabla T &= \kappa\nabla^2 T, \end{aligned} \qquad [11]$$

where $\mathbf{u} = (u, v, w)$ is the fluid velocity, p is the kinematic pressure, ν is the kinematic viscosity, κ is the thermal diffusivity, T is the temperature, and b is the fluid buoyancy. We assume that density and temperature are related by a linear equation of state so that the buoyancy b is only a function $b = \alpha g T$ where α is the thermal expansion coefficient and g is the acceleration due to gravity.

This system is commonly used in climate modeling, especially to model the ocean (51–54), in a multi-scale model that approximates these equations by averaging out the horizontal dynamics $\overline{T}(z,t) = \iint T(x,y,z,t)\,dx\,dy$ in individual boxes. The resulting approximation is a local advection-diffusion equation describing the evolution of the horizontally-averaged temperature $\overline{T}$:

$$\frac{\partial \overline{T}}{\partial t} + \frac{\partial \overline{wT}}{\partial z} = \kappa \frac{\partial^2 \overline{T}}{\partial z^2}. \qquad [12]$$

This one-dimensional approximating system is not closed since $\overline{wT}$ is unknown. Common practice closes the system by manually determining an approximating $\overline{wT}$ from ad-hoc models, physical reasoning, and scaling laws. However, we can utilize a UDE-automated approach to learn such an approximation from data. Let

$$\overline{wT} = U_\theta\left(P, \overline{T}, \frac{\partial \overline{T}}{\partial z}\right) \qquad [13]$$

where P are the physical parameters of the Boussinesq equation at different regimes of the ocean, such as the amount of surface heating or the strength of the surface winds (55). We can accurately capture the non-locality of the convection in this term by making the UDE a high-dimensional neural network. Using data from horizontal average temperatures $\overline{T}$ with known physical parameters P, we can directly reconstruct a nonlinear P-dependent parameterization by training a universal diffusion-advection partial differential equation. Supplementary Figure S1 demonstrates the accuracy of the approach using a deep UPDE with high order stabilized-explicit Runge-Kutta (ROCK) methods where the fitting is described in Supplement S4. To contrast the trained UPDE, we directly simulated the 3D Boussinesq equations under similar physical conditions and demonstrated that the neural parameterization results in around a 15,000x acceleration.

Data-Driven Nonlinear Closure Relations for Model Reduction in Non-Newtonian Viscoelastic Fluids. All continuum materials satisfy conservation equations for mass and momentum. The difference between an elastic solid an a viscous fluid comes down to the constitutive law relating the stresses and strains. In a one-dimensional system, an elastic solid satisfies $\sigma = G\gamma$, with stress σ, strain γ, and elastic modulus G, whereas a viscous fluid satisfies $\sigma = \eta\dot{\gamma}$, with viscosity η and strain rate $\dot{\gamma}$. Non-Newtonian fluids have more complex constitutive laws, for instance when stress depends on the history of deformation,

$$\sigma(t) = \int_{-\infty}^{t} G(t-s)F(\dot{\gamma}(s))\,\mathrm{d}s, \qquad [14]$$

alternatively expressed in the instantaneous form (56):

$$\begin{aligned} \sigma(t) &= \phi_1(t), \\ \frac{\mathrm{d}\phi_1}{\mathrm{d}t} &= G(0)F(\dot{\gamma}) + \phi_2, \\ \frac{\mathrm{d}\phi_2}{\mathrm{d}t} &= \frac{\mathrm{d}G(0)}{\mathrm{d}t}F(\dot{\gamma}) + \phi_3, \\ &\vdots \end{aligned} \qquad [15]$$

where the history is stored in ϕ_i. To become computationally feasible, the expansion is truncated, often in an ad-hoc manner, e.g. $\phi_n = \phi_{n+1} = \cdots = 0$, for some n. Only with a simple choice of $G(t)$ does an exact closure condition exist, e.g. the

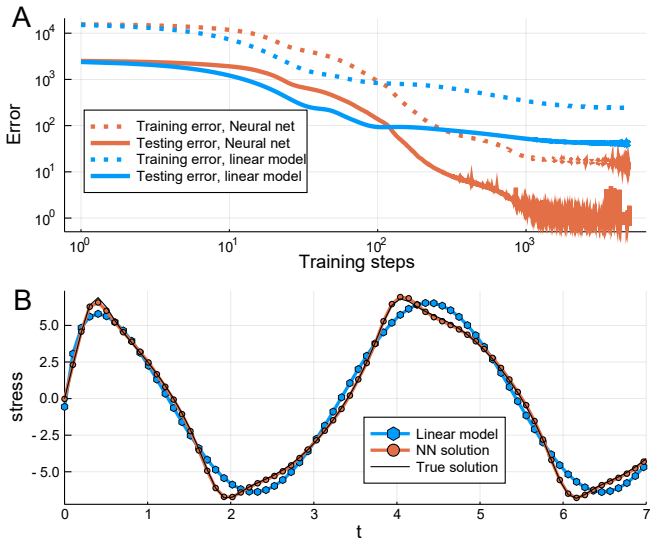

Fig. 3. Convergence of neural closure relations for a non-Newtonian Fluid. (A) Error between the approximated σ using the linear approximation Equation 6 and the neural network closure relation Equation 16 against the full FENE-P solution. The error is measured for the strain rates $\dot{\gamma} = 12\cos\omega t$ for $\omega = 1, 1.2, \ldots, 2$ and tested with the strain rate $\dot{\gamma} = 12\cos 1.5t$. (B) Predictions of stress for testing strain rate for the linear approximation and UODE solution against the exact FENE-P stress.

Oldroyd-B model. For a fully nonlinear approximation, we train a UODE according to the details in Supplement S5 to learn a latent closure relation:

$$\sigma(t) = U_0(\dot{\gamma}, \phi_1, \ldots, \phi_N), \quad [16]$$

$$\frac{d\phi_i}{dt} = U_i(\dot{\gamma}, \phi_1, \ldots, \phi_N), \quad \text{for } i = 1 \text{ to } N \quad [17]$$

from the numerical solution of the FENE-P equations, a fully non-linear constitutive law requiring a truncation condition (57). Figure 3 compares the neural network approach to a linear, Oldroyd-B like, model for σ and showcases that the nonlinear approximation improves the accuracy by more than 50x. We note that the neural network approximation accelerates the solution by 2x over the original 6-state DAE, demonstrating that the universal differential equation approach to model acceleration is not just applicable to large-scale dynamical systems like PDEs but also can be effectively employed to accelerate small scale systems.

Computationally-Efficient High-Dimensional Partial Differential Equations

It is impractical to solve high dimensional PDEs with mesh-based techniques since the number of mesh points scales exponentially with the number of dimensions. Given this difficulty, mesh-free methods based on latent functions such as neural networks have been constructed to allow for direct solving of high dimensional PDEs (58, 59). Recently, methods based on transforming partial differential equations into alternative forms, such as backwards stochastic differential equations (BSDEs), which are then approximated by neural networks have been shown to be highly efficient on important equations such as the nonlinear Black-Scholes and Hamilton-Jacobi-Bellman (HJB) equations (60–62). Here we will showcase how one of these methods, a deep BSDE method for semilinear parabolic equations (60), can be reinterpreted as a universal stochastic differential equation (USDE) to generalize the method and allow for enhancements like adaptivity, higher order integration for increased efficiency, and handling of stiff driving equations.

Consider the class of semilinear parabolic PDEs with a finite time span $t \in [0, T]$ and d-dimensional space $x \in \mathbb{R}^d$ that have the form:

$$\begin{aligned}\frac{\partial u}{\partial t}(t,x) + \frac{1}{2}\operatorname{Tr}\left(\sigma\sigma^T(t,x)\left(\operatorname{Hess}_x u\right)(t,x)\right) \\ + \nabla u(t,x)\cdot\mu(t,x) \\ + f\left(t, x, u(t,x), \sigma^T(t,x)\nabla u(t,x)\right) = 0,\end{aligned} \quad [18]$$

with a terminal condition $u(T,x) = g(x)$. Supplement S6 describes how this PDE can be solved by approximating by approximating the FBSDE:

$$\begin{aligned} dX_t &= \mu(t, X_t)dt + \sigma(t, X_t)dW_t, \\ dU_t &= f(t, X_t, U_t, U^1_{\theta_1}(t, X_t))dt + \left[U^1_{\theta_1}(t, X_t)\right]^T dW_t, \end{aligned} \quad [19]$$

where $U^1_{\theta_1}$ and $U^2_{\theta_2}$ are UAs and the loss function is given by the requiring that the terminating condition $g(X_T) = u(X_T, W_T)$ is satisfied.

Adaptive Solution of High-Dimensional Hamilton-Jacobi-Bellman Equations. A fixed time step Euler-Maryumana discretization of this USDE gives rise to the deep BSDE method (60). However, this form as a USDE generalizes the approach in a way that makes all of the methodologies of our USDE training library readily available, such as higher order methods, adaptivity, and implicit methods for stiff SDEs. As a motivating example, consider the classical linear-quadratic Gaussian (LQG) control problem in 100 dimensions:

$$dX_t = 2\sqrt{\lambda}c_t dt + \sqrt{2}dW_t, \quad [20]$$

with $t \in [0, T]$, $X_0 = x$, and with a cost function $C(c_t) = \mathbb{E}\left[\int_0^T \|c_t\|^2 dt + g(X_t)\right]$ where X_t is the state we wish to control, λ is the strength of the control, and c_t is the control process. Minimizing the control corresponds to solving the 100-dimensional HJB equation:

$$\frac{\partial u}{\partial t} + \nabla^2 u - \lambda\|\nabla u\|^2 = 0 \quad [21]$$

We solve the PDE by training the USDE using an adaptive Euler-Maruyama method (63) as described in Supplement S6. Supplementary Figure S2 showcases that this methodology accurately solves the equations, effectively extending recent algorithmic advancements to adaptive forms simply be reinterpreting the equation as a USDE. While classical methods would require an amount of memory that is exponential in the number of dimensions making classical adaptively approaches infeasible, this approach is the first the authors are aware of to generalize the high order, adaptive, highly stable software tooling to the high-dimensional PDE setting.

Discussion

While many attribute the success of deep learning to its black-box nature, the key advances in deep learning applications have come by developing new architectures which directly model the structures that are attempting to be learned. For

example, deep convolutional neural networks for image processing directly utilized the local spatial structure of images by modeling convolution stencil operations. Similarly, recurrent neural networks encode a forward time progression into a deep latent model and have excelled in natural language processing and time series prediction. Our results show that by combining known scientific models with machine learning, we can arrive at similar efficiency advancements by utilizing all known prior knowledge of the underlying problem's structure.

ACKNOWLEDGMENTS. We thank Jesse Bettencourt, Mike Innes, and Lyndon White for being instrumental in the early development of the DiffEqFlux.jl library, Tim Besard and Valentin Churavy for the help with the GPU tooling, and David Widmann and Kanav Gupta for their fundamental work across DifferentialEquations.jl. Special thanks to Viral Shah, Alan Edelman, and Steven Johnson who have been helpful in the refinement of these ideas. We thank Charlie Strauss, Steven Johnson, Nathan Urban, and Adam Gerlach for enlightening discussions and remarks on our manuscript and software.

1. BB Traore, B Kamsu-Foguem, F Tangara, Deep convolution neural network for image recognition. *Ecol. informatics* **48**, 257–268 (2018).
2. MT Islam, BMN Karim Siddique, S Rahman, T Jabid, Image recognition with deep learning in *2018 International Conference on Intelligent Informatics and Biomedical Sciences (ICIIBMS)*. Vol. 3, pp. 106–110 (2018).
3. C Chen, et al., This looks like that: deep learning for interpretable image recognition in *Advances in Neural Information Processing Systems*. pp. 8928–8939 (2019).
4. T Young, D Hazarika, S Poria, E Cambria, Recent trends in deep learning based natural language processing. *ieee Comput. intelligenCe magazine* **13**, 55–75 (2018).
5. DW Otter, JR Medina, JK Kalita, A survey of the usages of deep learning in natural language processing. *arXiv preprint arXiv:1807.10854* (2018).
6. Y Tsuruoka, Deep learning and natural language processing. *Brain nerve= Shinkei kenkyu no shinpo* **71**, 45 (2019).
7. Y Li, et al., Deep learning in bioinformatics: Introduction, application, and perspective in the big data era. *Methods* (2019).
8. B Tang, Z Pan, K Yin, A Khateeb, Recent advances of deep learning in bioinformatics and computational biology. *Front. Genet.* **10** (2019).
9. J Zou, et al., A primer on deep learning in genomics. *Nat. genetics* **51**, 12–18 (2019).
10. C Angermueller, T Pärnamaa, L Parts, O Stegle, Deep learning for computational biology. *Mol. systems biology* **12** (2016).
11. D Bacciu, PJ Lisboa, JD Martín, R Stoean, A Vellido, Bioinformatics and medicine in the era of deep learning. *arXiv preprint arXiv:1802.09791* (2018).
12. M Raissi, P Perdikaris, GE Karniadakis, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J. Comput. Phys.* **378**, 686–707 (2019).
13. M Alvarez, D Luengo, ND Lawrence, Latent force models in *Artificial Intelligence and Statistics*. pp. 9–16 (2009).
14. Y Hu, S Boker, M Neale, KL Klump, Coupled latent differential equation with moderators: Simulation and application. *Psychol. Methods* **19**, 56 (2014).
15. M Alvarez, JR Peters, ND Lawrence, B Schölkopf, Switched latent force models for movement segmentation in *Advances in neural information processing systems*. pp. 55–63 (2010).
16. TQ Chen, Y Rubanova, J Bettencourt, DK Duvenaud, Neural ordinary differential equations in *Advances in neural information processing systems*. pp. 6571–6583 (2018).
17. H Lin, S Jegelka, Resnet with one-neuron hidden layers is a universal approximator in *Advances in Neural Information Processing Systems*. pp. 6169–6178 (2018).
18. DA Winkler, TC Le, Performance of deep and shallow neural networks, the universal approximation theorem, activity cliffs, and qsar. *Mol. informatics* **36**, 1600118 (2017).
19. AN Gorban, DC Wunsch, The general approximation theorem in *1998 IEEE International Joint Conference on Neural Networks Proceedings. IEEE World Congress on Computational Intelligence (Cat. No. 98CH36227)*. (IEEE), Vol. 2, pp. 1271–1274 (1998).
20. DP Kingma, J Ba, Adam A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
21. DC Liu, J Nocedal, On the limited memory bfgs method for large scale optimization. *Math. programming* **45**, 503–528 (1989).
22. RM Errico, What is an adjoint model? *Bull. Am. Meteorol. Soc.* **78**, 2577–2592 (1997).
23. G Allaire, A review of adjoint methods for sensitivity analysis, uncertainty quantification and optimization in numerical codes. *Ingenieurs de l'Automobile* **836**, 33–36 (2015).
24. G Strang, *Computational science and engineering*. (Wellesley-Cambridge Press Wellesley) Vol. 791, (2007).
25. AC Hindmarsh, et al., SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Math. Softw. (TOMS)* **31**, 363–396 (2005).
26. B Sengupta, KJ Friston, WD Penny, Efficient gradient computation for dynamical models. *NeuroImage* **98**, 521–527 (2014).
27. C Rackauckas, et al., A comparison of automatic differentiation and continuous sensitivity analysis for derivatives of differential equation solutions. *arXiv preprint arXiv:1812.01892* (2018).
28. M Innes, et al., Fashionable modelling with flux. *CoRR* **abs/1811.01457** (2018).
29. A Paszke, et al., Pytorch: An imperative style, high-performance deep learning library in *Advances in Neural Information Processing Systems 32*, eds. H Wallach, et al. (Curran Associates, Inc.), pp. 8024–8035 (2019).
30. M Innes, et al., Zygote: A differentiable programming system to bridge machine learning and scientific computing. *arXiv preprint arXiv:1907.07587* (2019).
31. R Giering, T Kaminski, T Slawig, Generating efficient derivative code with taf: adjoint and tangent linear euler flow around an airfoil. *Futur. generation computer systems* **21**, 1345–1355 (2005).
32. L Hascoet, V Pascual, The tapenade automatic differentiation tool: Principles, model, and specification. *ACM Transactions on Math. Softw. (TOMS)* **39**, 20 (2013).
33. C Rackauckas, Q Nie, Differentialequations.jl – a performant and feature-rich ecosystem for solving differential equations in julia. *The J. Open Res. Softw.* **5** (2017) Exported from https://app.dimensions.ai on 2019/05/05.
34. Z Long, Y Lu, X Ma, B Dong, Pde-net: Learning pdes from data. *arXiv preprint arXiv:1710.09668* (2017).
35. M Raissi, P Perdikaris, GE Karniadakis, Multistep neural networks for data-driven discovery of nonlinear dynamical systems. *arXiv preprint arXiv:1801.01236* (2018).
36. PJ Schmid, Dynamic mode decomposition of numerical and experimental data. *J. fluid mechanics* **656**, 5–28 (2010).
37. MO Williams, IG Kevrekidis, CW Rowley, A data–driven approximation of the koopman operator: Extending dynamic mode decomposition. *J. Nonlinear Sci.* **25**, 1307–1346 (2015).
38. Q Li, F Dietrich, EM Bollt, IG Kevrekidis, Extended dynamic mode decomposition with dictionary learning: A data-driven adaptive spectral decomposition of the koopman operator. *Chaos: An Interdiscip. J. Nonlinear Sci.* **27**, 103111 (2017).
39. N Takeishi, Y Kawahara, T Yairi, Learning koopman invariant subspaces for dynamic mode decomposition in *Advances in Neural Information Processing Systems*. pp. 1130–1140 (2017).
40. H Schaeffer, Learning partial differential equations via data discovery and sparse optimization. *Proc. Royal Soc. A: Math. Phys. Eng. Sci.* **473**, 20160446 (2017).
41. M Quade, M Abel, K Shafi, RK Niven, BR Noack, Prediction of dynamical systems by symbolic regression. *Phys. Rev. E* **94**, 012214 (2016).
42. H Cao, L Kang, Y Chen, J Yu, Evolutionary modeling of systems of ordinary differential equations with genetic programming. *Genet. Program. Evolvable Mach.* **1**, 309–337 (2000).
43. K Raza, R Parveen, Evolutionary algorithms in genetic regulatory networks model. *CoRR* **abs/1205.1986** (2012).
44. H Schaeffer, G Tran, R Ward, Extracting sparse high-dimensional dynamics from limited data. *SIAM J. on Appl. Math.* **78**, 3279–3295 (2018).
45. R Tipireddy, P Perdikaris, P Stinis, AM Tartakovsky, A comparative study of physics-informed neural network models for learning unknown dynamics and constitutive relations. *CoRR* **abs/1904.04058** (2019).
46. SL Brunton, JL Proctor, JN Kutz, Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proc. Natl. Acad. Sci.* **113**, 3932–3937 (2016).
47. NM Mangan, SL Brunton, JL Proctor, JN Kutz, Inferring biological networks by sparse identification of nonlinear dynamics. *IEEE Transactions on Mol. Biol. Multi-Scale Commun.* **2**, 52–63 (2016).
48. NM Mangan, JN Kutz, SL Brunton, JL Proctor, Model selection for dynamical systems via sparse regression and information criteria. *Proc. Royal Soc. A: Math. Phys. Eng. Sci.* **473**, 20170009 (2017).
49. RA Fisher, The wave of advance of advantageous genes. *Annals Eugen.* **7**, 355–369 (1937).
50. P Grindrod, *The Theory and Applications of Reaction-diffusion Equations: Patterns and Waves*, Oxford applied mathematics and computing science series. (Clarendon Press), (1996).
51. B Cushman-Roisin, JM Beckers, Chapter 4 - equations governing geophysical flows in *Introduction to Geophysical Fluid Dynamics*, International Geophysics, eds. B Cushman-Roisin, JM Beckers. (Academic Press) Vol. 101, pp. 99 – 129 (2011).
52. Z Zhang, JC Moore, Chapter 11 - atmospheric dynamics in *Mathematical and Physical Fundamentals of Climate Change*, eds. Z Zhang, JC Moore. (Elsevier, Boston), pp. 347 – 405 (2015).
53. LH Kantha, CA Clayson, eds., Section 1.3 - governing equations in *Numerical Models of Oceans and Oceanic Processes*, International Geophysics. (Academic Press) Vol. 66, pp. 28–46 (2000).
54. SM Griffies, AJ Adcroft, Formulating the equations of ocean models. (2008).
55. SM Griffies, et al., Theory and Numerics of the Community Ocean Vertical Mixing (CVMix) Project, Technical report (2015).
56. F Morrison, A Morrison, *Understanding Rheology*, Raymond F. Boyer Library Collection. (Oxford University Press), (2001).
57. P Oliveira, Alternative derivation of differential constitutive equations of the oldroyd-b type. *J. Non-Newtonian Fluid Mech.* **160**, 40 – 46 (2009) Complex flows of complex fluids.
58. J Sirignano, K Spiliopoulos, Dgm: A deep learning algorithm for solving partial differential equations. *J. Comput. Phys.* **375**, 1339–1364 (2018).
59. IE Lagaris, A Likas, DI Fotiadis, Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks* **9**, 987–1000 (1998).
60. J Han, A Jentzen, W E, Solving high-dimensional partial differential equations using deep learning. *Proc. Natl. Acad. Sci.* **115**, 8505–8510 (2018).
61. Y Zang, G Bao, X Ye, H Zhou, Weak adversarial networks for high-dimensional partial differential equations. *arXiv preprint arXiv:1907.08272* (2019).
62. C Huré, H Pham, X Warin, Some machine learning schemes for high-dimensional nonlinear pdes. *arXiv preprint arXiv:1902.01599* (2019).
63. H Lamba, An adaptive timestepping algorithm for stochastic differential equations. *J. computational applied mathematics* **161**, 417–430 (2003).

PNAS
www.pnas.org

Supplementary Information for

Universal Differential Equations for Scientific Machine Learning

Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, Ali Ramadhan

Corresponding Author Christopher Rackauckas.
E-mail: crackauc@mit.edu

This PDF file includes:

Supporting Information Text

1. DiffEqFlux.jl Pullback Construction

The DiffEqFlux.jl pullback construction is not based on just one method but instead has a dispatch-based mechanism for choosing between different adjoint implementations. At a high level, the library defines the pullback on the differential equation solve function, and thus using a differential equation inside of a larger program leads to this chunk as being a single differentiable primitive that is inserted into the back pass of Flux.jl when encountered by overloading the Zygote.jl (1) and ChainRules.jl rule sets. Each layer definition defines a different handling for the back pass construction, and DiffEqFlux.jl currently has three possible layer definitions to choose from:

1. diffeq_fd: A forward-mode automatic differentiation approach which uses ForwardDiff.jl (2) on the generic code of the pure-Julia differential equation solvers.
2. diffeq_rd: A reverse-mode automatic differentiation approach which uses Tracker.jl (3) on the generic code of the pure-Julia differential equation solvers.
3. diffeq_adjoint: A adjoint-based approach which has 5 possible modes through the sensealg keyword argument.

Note that the diffeq_fd and diffeq_rd approach may soon be deprecated from the library and diffeq_adjoint will be renamed concrete_solve and dispatch between all modes. The 7 possible modes, each of which having multiple different options for internal calculations, of diffeq_adjoint/concrete_solve that are select-able by the sensealg argument are as follows. In total, these choices allow for defining pullback implementations using either AD or traditional adjoint methods in combination with the rest of the generated backwards pass.

A. Adjoint of an ODE. Before describing the modes, we first describe the adjoint of the differential equation.

We wish to solve for some cost function $G(u,p)$ evaluated throughout the differential equation, i.e.:

$$G(u,p) = G(u(p)) = \int_{t_0}^{T} g(u(t,p))dt,$$

for the ODE $u' = f(u,p,t)$. To derive this adjoint, introduce the Lagrange multiplier λ to form:

$$I(p) = G(p) - \int_{t_0}^{T} \lambda^*(u' - f(u,p,t))dt,$$

Since $u' = f(u,p,t)$, we have that:

$$\frac{dG}{dp} = \frac{dI}{dp} = \int_{t_0}^{T} (g_p + g_u s)dt - \int_{t_0}^{T} \lambda^*(s' - f_u s - f_p)dt,$$

for s_i being the sensitivity of the ith variable. After applying integration by parts to $\lambda^* s'$, we require that:

$$\lambda' = -\frac{df}{du}^* \lambda - \left(\frac{dg}{du}\right)^*,$$
$$\lambda(T) = 0,$$

to zero out a term and get:

$$\frac{dG}{dp} = \lambda^*(t_0)\frac{dG}{du}(t_0) + \int_{t_0}^{T} (g_p + \lambda^* f_p)\, dt.$$

If G is discrete, then it can be represented via the Dirac delta:

$$G(u,p) = \int_{t_0}^{T} \sum_{i=1}^{N} \|d_i - u(t_i,p)\|^2 \delta(t_i - t)dt,$$

in which case

$$g_u(t_i) = 2(d_i - u(t_i,p)),$$

at the data points (t_i, d_i). Therefore, the derivative of an ODE solution with respect to a cost function is given by solving for λ^* using an ODE for λ^T in reverse time, and then using that to calculate $\frac{dG}{dp}$. At each time point where discrete data is found, λ is then changed using a callback (discrete event handling) by the amount g_u to represent the Dirac delta portion of the integral. Lastly, we note that $\frac{dG}{du_0} = -\lambda(0)$ in this formulation.

Additionally, the ODEs

$$\mu' = -\lambda^* \frac{\partial f}{\partial p}$$

with $\mu(T) = 0$ can be appended to the system of equations to perform the quadrature for $\frac{dG}{dp}$.

B. Current Adjoint Calculation Methods. From this setup we have the following 7 possible modes for calculating the adjoint, with their pros and cons.

1. QuadratueAdjoint: a quadrature-based approach. This utilizes interpolation of the forward solution provided by DifferentialEquations.jl to calculate $u(t)$ at arbitrary time points for doing the calculations with respect to $\frac{df}{du}$ in the reverse ODE of λ. From this a continuous interpolatable $\lambda(t)$ is generated, and the integral formula for $\frac{dG}{dp}$ is calculated using the QuadGK.jl implementation of Gauss-Kronrod quadrature. While this approach is memory heavy due to requiring the interpolation of the forward and reverse passes, it can be the fastest version for cases where the number of ODE/DAE states is small and the number of parameters is large since the QuadGK quadrature can converge faster than ODE/DAE-based versions of quadrature. This method requires an ODE or a DAE.

2. InterpolatingAdjoint: a checkpointed interpolation approach. This approach solves the $\lambda(t)$ ODE in reverse using an interpolation of $u(t)$, but appends the equations for $\mu(t)$ and thus does not require saving the timeseries trajectory of $\lambda(t)$. For checkpointing, a scheme similar to that found in SUNDIALS (4) is used. Points (u_k, t_k) from the forward solution are chosen as the interval points. Whenever the backwards pass enters a new interval, the ODE is re-solved on $t \in [t_{k-1}, t_k]$ with a continuous interpolation provided by DifferentialEquations.jl. For the reverse pass, the tstops argument is set for each t_k, ensuring that no backwards integration step lies in two checkpointing intervals. This requires at most a total of two forward solutions of the ODE and the memory required to hold the interpolation of the solution between two consecutive checkpoints. Note that making the checkpoints at the start and end of the integration interval makes this equivalent to a non-checkpointed interpolated approach which replaces the quadrature with an ODE/DAE solve for memory efficiency. This method tends to be both stable and require a minimal amount of memory, and is thus the default. This method requires an ODE or a DAE.

3. BacksolveAdjoint: a checkpointed backwards solution approach. Following (5), after a forward solution, this approach solves the $u(t)$ equation in reverse along with the $\lambda(t)$ and $\mu(t)$ ODEs. Thus, since no interpolations are required, it requires $\mathcal{O}(1)$ memory. Unfortunately, many theoretical results show that backwards solution of ODEs is not guaranteed to be stable, and testing this adjoint on the universal partial differential equations like the diffusion-advection example of this paper showcases that it can be divergent and is thus not universally applicable, especially in cases of stiffness. Thus for stability we modify this approach by allowing checkpoints (u_k, t_k) at which the reverse integration is reset, i.e. $u(t_k) = u_k$, and the backsolve is then continued. The tstops argument is set in the integrator to require that each checkpoint is hit exactly for this resetting to occur. By doing so, the resulting method utilizes $\mathcal{O}(1)$ memory + the number of checkpoints required for stability, making it take the least memory approach. However, the potential divergence does lead to small errors in the gradient, and thus for highly stiff equations we have found that this is only applicable to a certain degree of tolerance like 10^{-6} given reasonable numbers of checkpoints. When applicable this can be the most efficient method for large memory problems. This method requires an ODE or a DAE.

4. ForwardSensitivity: a forward sensitivity approach. From $u' = f(u, p, t)$, the chain rule gives $\frac{d}{dt}\frac{du}{dp} = \frac{df}{du}\frac{du}{dp} + \frac{df}{dp}$ which can be appended to the original equations to give $\frac{du}{dp}$ as a time series, which can then be used to compute $\frac{dG}{dp}$. While the computational cost of the adjoint methods scales like $\mathcal{O}(N + P)$ for N differential equations and P parameters, this approach scales like $\mathcal{O}(NP)$ and is thus only applicable to models with small numbers of parameters (thus excluding neural networks). However, when the universal approximator has small numbers of parameters, this can be the most efficient approach. This method requires an ODE or a DAE.

5. ForwardDiffSensitivity: a forward-mode automatic differentiation approach, using ForwardDiff.jl (2) to calculate the forward sensitivity equations. Because it utilizes a forward-mode approach, the scaling matches that of the forward sensitivity approach and it tends to have similar performance characteristics. This method applies to any Julia-based differential equation solver.

6. TrackerAdjoint: a Tracker-driven taped-based reverse-mode discrete adjoint sensitivity. This is done by using the TrackedArray constructs of Tracker.jl (3) to build a Wengert list (or tape) of the forward execution of the ODE solver which is then reversed. This method applies to any Julia-based differential equation solver.

7. ZygoteAdjoint: a Zygote-driven source-to-source reverse-mode discrete adjoint sensitivity. This utilizes the Zygote.jl (1) system directly on the differential equation solvers to generate a source code for the reverse pass of the solver itself. Currently this is only directly applicable to a few differential equation solvers, but is under heavy development.

For each of the non-AD approaches, there are the following choices for how the Jacobian-vector products Jv (jvp) of the forward sensitivity equations and the vector-Jacobian products $v'J$ (vjp) of the adjoint sensitivity equations are computed:

1. Automatic differentiation for the jvp and vjp. In this approach, automatic differentiation is utilized for directly calculating the jvps and vjps. ForwardDiff.jl with a single dual dimension is applied at $f(u + \lambda\epsilon)$ to calculate $\frac{df}{du}\lambda$ where ϵ is a dual dimensional signifier. For the vector-Jacobian products, a forward pass at $f(u)$ is utilized and the backwards pass is seeded at λ to compute the $\lambda'\frac{df}{du}$ (and similarly for $\frac{df}{dp}$). Note that if f is a neural network, this implies that this product is computed by starting the backpropogation of the neural network with λ and the vjp is the resulting return. Tracker.jl

with arrays of tracked real values is utilized on mutating functions, and Zygote.jl is utilized for non-mutating differential equation function definitions. Thus the internal calculation of the vjp on a general UDE recurses down to primitives and embeds optimized backpropogations of the internal neural networks (and other universal approximators) for the calculation of this product when this option is used.

2. Numerical differentiation for the jvp and vjp. In this apporoach, finite differences is utilized for directly calculating the jvps and vjps. For a small but finite ϵ, $\left(f(u+\lambda\epsilon) - f(u)\right)/\epsilon$ is used to approximate $\frac{df}{du}\lambda$. For vjps, a finite difference gradient of $\lambda' f(u)$ is used.

3. Automatic differentiation for Jacobian construction. In this approach, (sparse) forward-mode automatic differentiation is utilized by a combination of ForwardDiff.jl (2) with SparseDiffTools.jl for color-vector based sparse Jacobian construction. After forming the Jacobian, the jvp or vjp is calculated.

4. Numerical differentiation for Jacobian construction. In this approach, (sparse) numerical differentiation is utilized by a combination of DiffEqDiffTools.jl with SparseDiffTools.jl for color-vector based sparse Jacobian construction. After forming the Jacobian, the jvp or vjp is calculated.

In total this gives 22 different adjoint method approaches, each with different performance characteristics and limitations. A full performance analysis which measures the optimal adjoint approach for various UDEs has been omitted from this paper, since the combinatorial nature of the options requires a considerable amount of space to showcase the performance advantages and generality disadvantages between each of the approaches. A follow-up study focusing on accurate performance measurements of the adjoint choice combinations on families of UDEs is planned.

2. Partial Reconstruction of the Lotka-Volterra Equations

On the Lotka-Volterra equations, we trained a UDE model against a trajectory of 31 points with a constant step size $\Delta t = 0.1$ starting from $x_0 = 0.44249296,\ y_0 = 4.6280594$ to recover the function $U_\theta(x, y)$. The parameters are chosen to be $\alpha = 1.3,\ \beta = 0.9,\ \gamma = 0.5,\ \delta = 1.8$. The neural network consists of an input layer, three hidden layers with 32 neurons each and a linear output layer, with all hidden layers having a swish activation function. We trained for 1000 iterations with ADAM with a learning rate $\gamma = 3 \times 10^{-2}$ and then 10000 iterations with gradient descent and a learning rate of 10^-4. The loss was chosen was the L2 loss $\mathcal{L} = \sum_i (u_\theta(t_i) - d_i)^2$. The training converged to a final loss of approximately 0.03 for the knowledge-enhanced neural network and 0.01 for the full neural network approach. From the trained neural network, data was sampled at the $\Delta t = 0.1$ and fit using the STRRidge method from the original SInDy paper with a LASSO L1 regularization term to ensure sparsity in the fit. The knowledge-enhanced neural network returned zero for all terms except non-zeros on the quadratic terms with nearly the correct coefficients that were then fixed using a post structure identification parameter estimation, while the full neural network approach resulted in the sparse regression returning either a dense basis or large numbers of terms without finding the correct piece (depending on the regularization size and the cutoff tolerances), indicating that the full fitting method was not robustly able to recover the original governing equations.

3. Model-based Learning for the Fisher-KPP Equations

To generate training data for the 1D Fisher-KPP equation 9, we take the growth rate and the diffusion coefficient to be $r = 1$ and $D = 0.01$ respectively. The equation is numerically solved in the domain $x \in [0, 1]$ and $t \in [0, T]$ using a 2nd order central difference scheme for the spatial derivatives and the time-integration is done using the Tsitouras 5/4 Runge-Kutta method. We implement periodic boundary condition $\rho(x = 0, t) = \rho(x = 1, t)$ and initial condition $\rho(x, t = 0) = \rho_0(x)$ is taken to be a localized function given by

$$\rho_0(x) = \frac{1}{2}\left(\tanh\left(\frac{x - (0.5 - \Delta/2)}{\Delta/10}\right) - \tanh\left(\frac{x - (0.5 + \Delta/2)}{\Delta/10}\right)\right), \quad [1]$$

with $\Delta = 0.2$ which represents the width of the region where $\rho \simeq 1$. The data are saved at evenly spaced points with $\Delta x = 0.04$ and $\Delta t = 0.5$.

In the UPDE Equation 10, the growth neural network $\mathrm{NN}_\theta(\rho)$ has 4 densely connected layers with 10, 20, 20 and 10 neurons each and tanh activation functions. The diffusion operator is represented by a CNN that operates on an input vector of arbitrary size. It has 1 hidden layer with a 3×1 filter $[w_1, w_2, w_3]$ without any bias. To implement periodic boundary conditions for the UPDE at each time step, the vector of values at different spatial locations $[\rho_1, \rho_2, \ldots, \rho_{N_x}]$ is padded with ρ_{N_x} to the left and ρ_1 to the right. This also ensures that the output of the CNN is of the same size as the input. The weights of both the neural networks and the diffusion coefficient are simultaneously trained to minimize the loss function

$$\mathcal{L} = \sum_i (\rho(x_i, t_i) - \rho_{\mathrm{data}}(x_i, t_i))^2 + \lambda |w_1 + w_2 + w_3|, \quad [2]$$

where λ is taken to be 10^2 (note that one could also structurally enforce $w_3 = -(w_1 + w_2)$). The second term in the loss function enforces that the differential operator that is learned is conservative—that is, the weights sum to zero. The training is done using the ADAM optimizer with learning rate 10^{-3}.

4. Reduction of the Boussinesq Equations

As a test for the diffusion-advection equation parameterization approach, data was generated from the diffusion-advection equations using the missing function $\overline{wT} = \cos(\sin(T^3)) + \sin(\cos(T^2))$ with N spatial points discretized by a finite difference method with $t \in [0, 1.5]$ with Neumann zero-flux boundary conditions. A neural network with two hidden layers of size 8 and tanh activation functions was trained against 30 data points sampled from the true PDE. The UPDE was fit by using the ADAM optimizer with learning rate 10^{-2} for 200 iterations and then ADAM with a learning rate of 10^{-3} for 1000 iterations. The resulting fit is shown in Supplementary Figure 2 which resulted in a final loss of approximately 0.007. We note that the stabilized adjoints were required for this equation, i.e. the backsolve adjoint method was unstable and results in divergence and thus cannot be used on this type of equation. The trained neural network had a forward pass that took around 0.9 seconds.

For the benchmark against the full Bossinesq equations, we utilized Oceananigans.jl (6). It was set to utilize adaptive time stepping to maximize the time step according to the CFL condition number (capped at CFL ≤ 0.3) and matched the boundary conditions, along with setting periodic boundary conditions in the horizontal dimension. The Bossinesq simulation used $128 \times 128 \times 128$ spatial points, a larger number than the parameterization, in order to accurately resolve the mean statistics of the 3-dimensional dynamics as is commonly required in practice (7). The resulting simulation took 13,737 seconds on the same computer used for the neural diffusion-advection approach, demonstrating the approximate 15,000x acceleration.

5. Automated Derivation of Closure Relations for Viscoelastic Fluids

The full FENE-P model is:

$$\boldsymbol{\sigma} + g\left(\frac{\lambda}{f(\boldsymbol{\sigma})}\boldsymbol{\sigma}\right) = \frac{\eta}{f(\boldsymbol{\sigma})}\dot{\boldsymbol{\gamma}}, \tag{3}$$

$$f(\boldsymbol{\sigma}) = \frac{L^2 + \frac{\lambda(L^2-3)}{L^2\eta}\mathrm{Tr}(\boldsymbol{\sigma})}{L^2 - 3}, \tag{4}$$

where

$$g(\boldsymbol{A}) = \frac{D\boldsymbol{A}}{Dt} - (\nabla \boldsymbol{u}^T)\boldsymbol{A} - \boldsymbol{A}(\nabla \boldsymbol{u}^T),$$

is the upper convected derivative, and L, η, λ are parameters (8). For a one dimensional strain rate, $\dot{\gamma} = \dot{\gamma}_{12} = \dot{\gamma}_{21} \neq 0$, $\dot{\gamma}_{ij} = 0$ else, the one dimensional stress required is $\sigma = \sigma_{12}$. However, σ_{11} and σ_{22} are both non-zero and store memory of the deformation (normal stresses). The Oldroyd-B model is the approximation:

$$G(t) = 2\eta\delta(t) + G_0 e^{-t/\tau}, \tag{5}$$

with the exact closure relation:

$$\sigma(t) = \eta\dot{\gamma}(t) + \phi, \tag{6}$$

$$\frac{\mathrm{d}\phi}{\mathrm{d}t} = G_0\dot{\gamma} - \phi/\tau. \tag{7}$$

As an arbitrary nonlinear extension, train a UDE model using a single additional memory field against simulated FENE-P data with parameters $\lambda = 2$, $L = 2$, $\eta = 4$. The UDE model is of the form,

$$\sigma = U_0(\phi, \dot{\gamma}) \tag{8}$$

$$\frac{\mathrm{d}\phi}{\mathrm{d}t} = U_1(\phi, \dot{\gamma}) \tag{9}$$

where U_0, U_1 are neural networks each with a single hidden layer containing 4 neurons. The hidden layer has a tanh activation function. The loss was taken as $\mathcal{L} = \sum_i (\sigma(t_i) - \sigma_{\text{FENE-P}}(t_i))^2$ for 100 evenly spaced time points in $t_i \in [0, 2\pi]$, and the system was trained using an ADAM iterator with learning rate 0.015. The fluid is assumed to be at rest before $t = 0$, making the initial stress also zero.

6. Adaptive Solving for the 100 Dimensional Hamilton-Jacobi-Bellman Equation

A. Forward-Backwards SDE Formulation. Consider the class of semilinear parabolic PDEs, in finite time $t \in [0, T]$ and d-dimensional space $x \in \mathbb{R}^d$, that have the form:

$$\begin{aligned}\frac{\partial u}{\partial t}(t,x) + \frac{1}{2}\mathrm{tr}\left(\sigma\sigma^T(t,x)\left(\mathrm{Hess}_x u\right)(t,x)\right) \\ + \nabla u(t,x)\cdot\mu(t,x) \\ + f\left(t, x, u(t,x), \sigma^T(t,x)\nabla u(t,x)\right) = 0,\end{aligned} \tag{10}$$

with a terminal condition $u(T, x) = g(x)$. In this equation, tr is the trace of a matrix, σ^T is the transpose of σ, ∇u is the gradient of u, and $\mathrm{Hess}_x u$ is the Hessian of u with respect to x. Furthermore, μ is a vector-valued function, σ is a $d \times d$

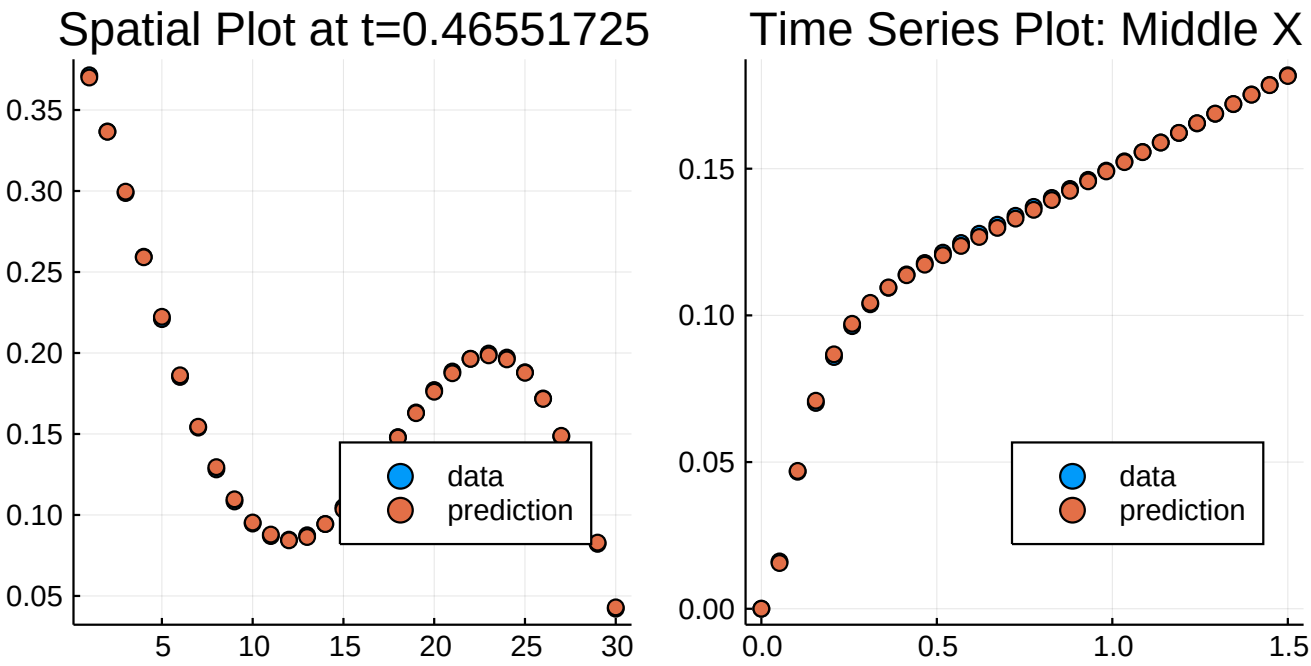

Fig. S1. Reduction of the Boussinesq equations. On the left is the comparison between the training data (blue) and the trained UPDE (ornage) over space at the 10th fitting time point, and on the right is the same comparison shown over time at spatial midpoint.

matrix-valued function and f is a nonlinear function. We assume that μ, σ, and f are known. We wish to find the solution at initial time, $t = 0$, at some starting point, $x = \zeta$.

Let W_t be a Brownian motion and take X_t to be the solution to the stochastic differential equation

$$dX_t = \mu(t, X_t)dt + \sigma(t, X_t)dW_t \tag{11}$$

with a terminal condition $u(T, x) = g(x)$. With initial condition $X(0) = \zeta$ has shown that the solution to Equation 18 satisfies the following forward-backward SDE (FBSDE) (9):

$$\begin{aligned} u(t,X_t) - u(0,\zeta) = \\ - \int_0^t f(s, X_s, u(s, X_s), \sigma^T(s, X_s)\nabla u(s, X_s))ds \\ + \int_0^t [\nabla u(s, X_s)]^T \sigma(s, X_s)dW_s, \end{aligned} \tag{12}$$

with terminating condition $g(X_T) = u(X_T, W_T)$. Notice that we can combine 11 and 12 into a system of $d + 1$ SDEs:

$$\begin{aligned} dX_t =& \mu(t, X_t)dt + \sigma(t, X_t)dW_t, \\ dU_t =& f(t, X_t, U_t, \sigma^T(t, X_t)\nabla u(t, X_t))dt \\ &+ \left[\sigma^T(t, X_t)\nabla u(t, X_t)\right]^T dW_t, \end{aligned} \tag{13}$$

where $U_t = u(t, X_t)$. Since X_0, μ, σ, and f are known from the choice of model, the remaining unknown portions are the functional $\sigma^T(t, X_t)\nabla u(t, X_t)$ and initial condition $U(0) = u(0, \zeta)$, the latter being the point estimate solution to the PDE.

To solve this problem, we approximate both unknown quantities by universal approximators:

$$\begin{aligned} \sigma^T(t, X_t)\nabla u(t, X_t) \approx U^1_{\theta_1}(t, X_t), \\ u(0, X_0) \approx U^2_{\theta_2}(X_0), \end{aligned} \tag{14}$$

Therefore we can rewrite 13 as a stochastic UDE of the form:

$$\begin{aligned} dX_t = \mu(t, X_t)dt + \sigma(t, X_t)dW_t, \\ dU_t = f(t, X_t, U_t, U^1_{\theta_1}(t, X_t))dt + \left[U^1_{\theta_1}(t, X_t)\right]^T dW_t, \end{aligned} \tag{15}$$

with initial condition $(X_0, U_0) = (X_0, U^2_{\theta_2}(X_0))$.

To be a solution of the PDE, the approximation must satisfy the terminating condition, and thus we define our loss to be the expected difference between the approximating solution and the required terminating condition:

$$l(\theta_1, \theta_2 | X_T, U_T) = \mathbb{E}\left[\|g(X_T) - U_T\|\right]. \tag{16}$$

Finding the parameters (θ_1, θ_2) which minimize this loss function thus give rise to a BSDE which solves the PDE, and thus $U^2_{\theta_2}(X_0)$ is the solution to the PDE once trained.

B. The LQG Control Problem. This PDE can be rewritten into the canonical form by setting:

$$\begin{aligned} \mu = 0, \\ \sigma = \overline{\sigma} I, \\ f = -\alpha \left\|\sigma^T(s, X_s)\nabla u(s, X_s))\right\|^2, \end{aligned} \tag{17}$$

where $\overline{\sigma} = \sqrt{2}$, T = 1 and $X_0 = (0, ..., 0) \in R^{100}$. The universal stochastic differential equation was then supplemented with a neural network as the approximator. The initial condition neural network was had 1 hidden layer of size 110, and the $\sigma^T(t, X_t)\nabla u(t, X_t)$ neural network had two layers both of size 110. For the example we chose $\lambda = 1$. This was trained with the LambaEM method of DifferentialEquations.jl (10) with relative and absolute tolerances set at $1e - 4$ using 500 training iterations and using a loss of 100 trajectories per epoch.

On this problem, for an arbitrary g, one can show with Itô's formula that:

$$u(t, x) = -\frac{1}{\lambda} \ln\left(\mathbb{E}\left[\exp\left(-\lambda g(x + \sqrt{2}W_{T-t}\right)\right]\right), \tag{18}$$

which was used to calculate the error from the true solution.

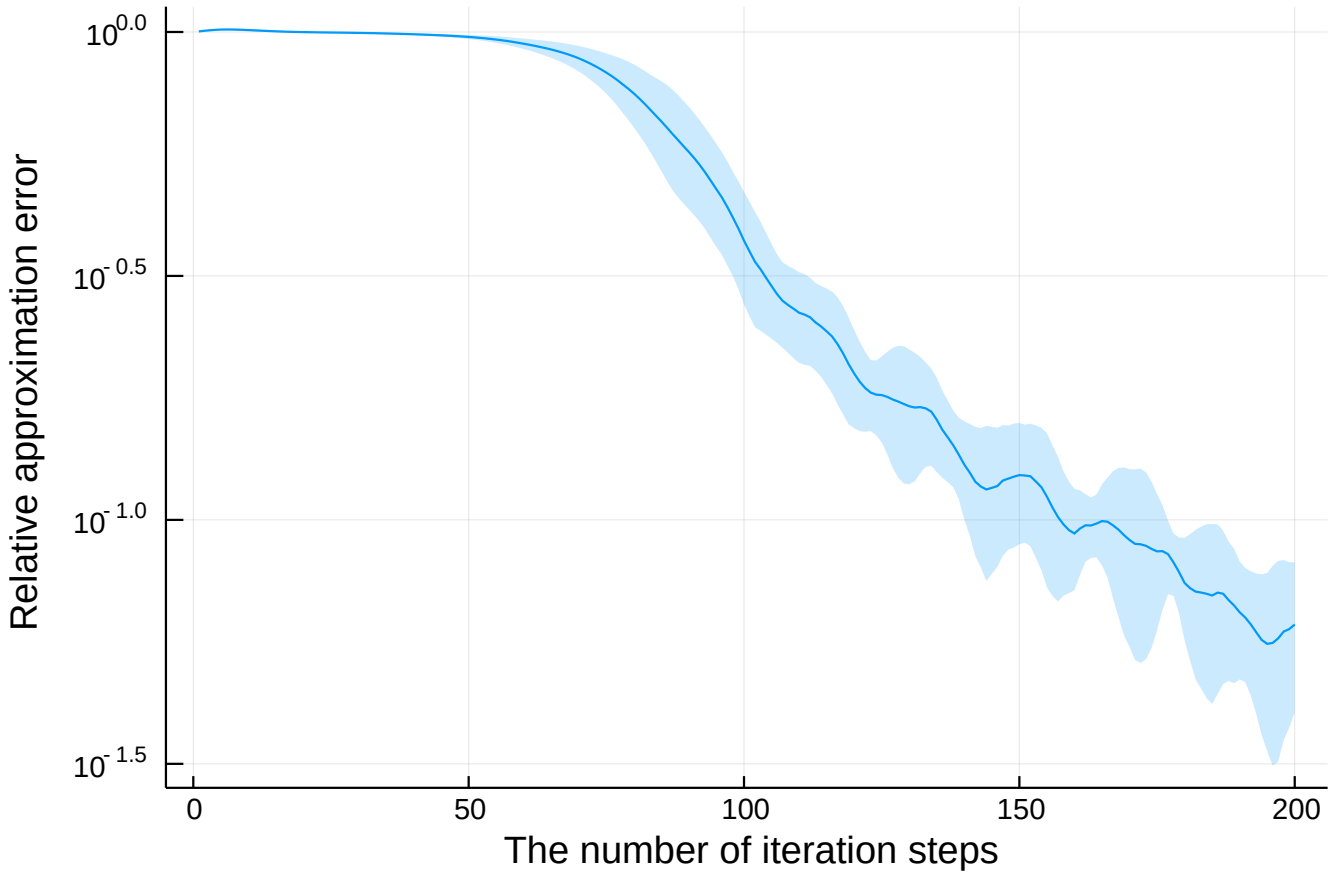

Fig. S2. Adaptive solution of the 100-dimensional Hamilton-Jacobi-Bellman equation. This demonstrates that as the universal approximators $U^1_{\theta_1}$ and $U^2_{\theta_2}$ converge to satisfy the terminating condition, $U^2_{\theta_2}$ network convergences to the solution of Equation 21.

References

1. M Innes, et al., Zygote: A differentiable programming system to bridge machine learning and scientific computing. *arXiv preprint arXiv:1907.07587* (2019).
2. J Revels, M Lubin, T Papamarkou, Forward-mode automatic differentiation in julia. *arXiv:1607.07892 [cs.MS]* (2016).
3. M Innes, et al., Fashionable modelling with flux. *CoRR* **abs/1811.01457** (2018).
4. AC Hindmarsh, et al., SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Math. Softw. (TOMS)* **31**, 363–396 (2005).
5. TQ Chen, Y Rubanova, J Bettencourt, DK Duvenaud, Neural ordinary differential equations in *Advances in neural information processing systems*. pp. 6571–6583 (2018).
6. A Ramadhan, et al., Oceananigans.jl: Fast and friendly geophysical fluid dynamics on GPUs. *The J. Open Source Softw.* **4**, 1965 (2020).
7. B Cushman-Roisin, JM Beckers, Chapter 4 - equations governing geophysical flows in *Introduction to Geophysical Fluid Dynamics*, International Geophysics, eds. B Cushman-Roisin, JM Beckers. (Academic Press) Vol. 101, pp. 99 – 129 (2011).
8. P Oliveira, Alternative derivation of differential constitutive equations of the oldroyd-b type. *J. Non-Newtonian Fluid Mech.* **160**, 40 – 46 (2009) Complex flows of complex fluids.
9. J Zhang, Backward stochastic differential equations in *Backward Stochastic Differential Equations*. (Springer), pp. 79–99 (2017).
10. C Rackauckas, Q Nie, Differentialequations.jl – a performant and feature-rich ecosystem for solving differential equations in julia. *The J. Open Res. Softw.* **5** (2017) Exported from https://app.dimensions.ai on 2019/05/05.