Universal Differential Equations for Scientific Machine Learning: Supplemental Information

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

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

August 26, 2020

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. For any ChainRules.jl-compliant reverse-mode AD package in the Julia language, when a differential equation solve is encountered in any Julia library during the backwards pass, the adjoint method is automatically swapped in to be used for the backpropogation of the solver. The choice of the adjoint is chosen by the type of the sensealg keyword argument which are fully described below.

1.1 Backpropagation-Accelerated DAE Adjoints for Index-1 DAEs with Constraint Equations

Before describing the modes, we first describe the adjoint of the differential equation with constraints. The constrained ordinary differential equation:

$$u' = \tilde{f}(u, p, t), \tag{1}$$
$$0 = c(u, p, t), \tag{2}$$

can be rewritten in mass matrix form:

$$Mu' = f(u, p, t). \tag{3}$$

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

$$G(u, p) = \int_{t_0}^{T} g(u, p, t)dt, \tag{4}$$

To derive this adjoint, introduce the Lagrange multiplier λ to form:

$$I(p) = G(p) - \int_{t_0}^{T} \lambda^*(Mu' - f(u, p, t))dt, \tag{5}$$

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

$$\frac{dG}{dp} = \frac{dI}{dp} = \int_{t_0}^{T} (g_p + g_u s)dt - \int_{t_0}^{T} \lambda^*(Ms' - f_u s - f_p)dt, \tag{6}$$

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

$$M^*\lambda' = -\frac{df}{du}^* \lambda - \left(\frac{dg}{du}\right)^*, \tag{7}$$

$$\lambda(T) = 0, \tag{8}$$

to zero out a term and get:

$$\frac{dG}{dp} = \lambda^*(t_0)M\frac{du}{dp}(t_0) + \int_{t_0}^{T} (g_p + \lambda^* f_p)\, dt. \tag{9}$$

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

$$G(u, p) = \int_{t_0}^{T} \sum_{i=1}^{N} \|d_i - u(t_i, p)\|^2 \delta(t_i - t)dt, \tag{10}$$

in which case

$$g_u(t_i) = 2(d_i - u(t_i, p)), \tag{11}$$

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.

We have to take care of consistent initialization in the case of semi-explicit index-1 DAEs. We need to satisfy the system of equations

$$M^* \Delta\lambda^d = h^*_{u^d}\lambda^a + g^*_{u^d} \tag{12}$$
$$0 = h^*_{u^a}\lambda^a + g^*_{u^a}, \tag{13}$$

where ^{d} and ^{a} denote differential and algebraic variables, and f and g denote differential and algebraic equations respectively. Combining the above two equations, we know that we need to increment the differential part of λ by

$$-h^*_{u^d}\left(h^*_{u^a}\right)^{-1} g^*_{u^a} + g^*_{u^d} \tag{14}$$

at each callback. Additionally, the ODEs

$$\mu' = -\lambda^* \frac{\partial f}{\partial p} \tag{15}$$

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

1.2 Current Adjoint Calculation Methods

From this setup we have the following 8 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 time-series trajectory of $\lambda(t)$. For checkpointing, a scheme similar to that found in SUNDIALS [2] 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/SDE/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, SDE, or a DAE.

3. BacksolveAdjoint: a checkpointed backwards solution approach. Following [3], 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, SDE, 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 [4] to calculate the forward sensitivity equations, i.e. an AD-generated implementation of forward-mode "discretize-then-optimize". Because it utilizes a forward-mode approach, the scaling matches that of the forward sensitivity approach and it tends to have similar perfor-

mance characteristics. This method applies to any Julia-based differential equation solver.

6. TrackerAdjoint: a Tracker-driven taped-based reverse-mode discrete adjoint sensitivity, i.e. an AD-generated implementation of reverse-mode "discretize-then-optimize". This is done by using the TrackedArray constructs of Tracker.jl [5] 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, i.e. an AD-generated implementation of reverse-mode "discretize-then-optimize". 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.

8. ReverseDiffAdjoint: A ReverseDiff.jl taped-based reverse-mode discrete adjoint sensitivity, i.e. an AD-generated implementation of reverse-mode "discretize-then-optimize". In contrast to TrackerAdjoint, this methodology can be substantially faster due to its ability to precompile the tape but only supports calculations on the CPU.

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. Three methods are allowed to be chosen for performing the internal vjp calculations:

   (a) Zygote.jl source-to-source transformation based vjps. Note that only non-mutating differential equation function definitions are supported in this mode. This mode is the most efficient in the presence of neural networks.

   (b) ReverseDiff.jl tape-based vjps. This allows for JIT-compilation of the tape for accelerated computation. This is the fastest vjp choice in the presence of heavy scalar operations like in chemical reaction networks, but is not compatible with GPU acceleration.

(c) Tracker.jl with arrays of tracked real values is utilized on mutating functions.

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 approach, 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 [4] 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 48 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 Benchmarks

2.1 ODE Solve Benchmarks

The three ODE benchmarks utilized the Lorenz equations (LRNZ) weather prediction model from [6] and the standard ODE IVP Testset [7, 8]:

$$\frac{dx}{dt} = \sigma(y - x) \tag{16}$$

$$\frac{dy}{dt} = x(\rho - z) - y \tag{17}$$

$$\frac{dz}{dt} = xy - \beta z \tag{18}$$

The 28 ODE benchmarks utilized the Pleiades equation (PLEI) celestial mechanics simulation from [6] and the standard ODE IVP Testset [7, 8]:

$$x_i'' = \sum_{j \neq i} m_j (x_j - x_i)/r_{ij} \tag{19}$$

$$y_i'' = \sum_{j \neq i} m_j (y_j - y_i)/r_{ij} \tag{20}$$

where

$$r_{ij} = \left((x_i - x_j)^2 + (y_i - y_j)^2\right)^{3/2} \tag{21}$$

on $t \in [0, 3]$ with initial conditions:

$$u(0) = [3.0, 3.0, -1.0, -3.0, 2.0, -2.0, 2.0, 3.0, -3.0, 2.0, 0, 0, \tag{22}$$

$$- 4.0, 4.0, 0, 0, 0, 0, 0, 1.75, -1.5, 0, 0, 0, -1.25, 1, 0, 0] \tag{23}$$

written in the form $u = [x_i, y_i, x_i', y_i']$.

The rest of the benchmarks were derived from a discretization a two-dimensional reaction diffusion equation, representing systems biology, combustion mechanics, spatial ecology, spatial epidemiology, and more generally physical PDE equations:

$$A_t = D\Delta A + \alpha_A(x) - \beta_A A - r_1 AB + r_2 C \tag{24}$$

$$B_t = \alpha_B - \beta_B B - r_1 AB + r_2 C \tag{25}$$

$$C_t = \alpha_C - \beta_C C + r_1 AB - r_2 C \tag{26}$$

where $\alpha_A(x) = 1$ if $x > 80$ and 0 otherwise on the domain $x \in [0, 100]$, $y \in [0, 100]$, and $t \in [0, 10]$ with zero-flux boundary conditions. The diffusion constant D was chosen as 100 and all other parameters were left at 1.0. In this parameter regime the ODE was non-stiff as indicated by solves with implicit methods not yielding performance advantages. The diffusion operator was discretized using the second order finite difference stencil on an $N \times N$ grid, where N was chosen to be 16, 32, 64, 128, 256, and 512. To ensure fairness, the torchdiffeq functions were compiled using torchscript. The code for reproducing the benchmark can be found at:

`https://gist.github.com/ChrisRackauckas/cc6ac746e2dfd285c28e0584a2bfd320`

2.2 Neural ODE Training Benchmark

The spiral neural ODE from [3] was used as the benchmark for the training of neural ODEs. The data was generated according from the form:

$$u' = Au^3 \tag{27}$$

where $A = [-0.1, 2.0; -2.0, -0.1]$ on $t \in [0, 1.5]$ where data was taken at 30 evenly spaced points. Each of the software packages trained the neural ODE

for 500 iterations using ADAM with a learning rate of 0.05. The defaults using the SciML software resulted in a final loss of 4.895287e-02 in 7.4 seconds, the optimized version (choosing BacksolveAdjoint with compiled ReverseDiff vector-jacobian products) resulted in a final loss of 2.761669e-02 in 2.7 seconds, while torchdiffeq achieved a final loss of 0.0596 in 289 seconds. To ensure fairness, the torchdiffeq functions were compiled using torchscript. Code to reproduce the benchmark can be found at:

`https://gist.github.com/ChrisRackauckas/4a4d526c15cc4170ce37da837bfc32c4`

2.3 SDE Solve Benchmark

The torchsde benchmarks were created using the geometric Brownian motion example from the torchsde README. The SDE was a 4 independent geometric Brownian motions:

$$dX_t = \mu X_t dt + \sigma X_t dW_t \tag{28}$$

where $\mu = 0.5$ and $\sigma = 1.0$. Both software solved the SDE 100 times using the SRI method [9] with fixed time step chosen to give 20 evenly spaced steps for $t \in [0, 1]$. The SciML ecosystem solvers solved the equation 100 times in 0.00115 seconds, while torchsde v0.1 took 1.86 seconds. We contacted the author who rewrote the Brownian motion portions into C++ and linked it to torchsde as v0.1.1 and this improved the timing to roughly 5 seconds, resulting in a final performance difference of approximately 1,600x. The code to reproduce the benchmarks and the torchsde author's optimization notes can be found at:

`https://gist.github.com/ChrisRackauckas/6a03e7b151c86b32d74b41af54d495c6`

3 Sparse Identification of Missing Model Terms via Universal Differential Equations

The SINDy algorithm [10, 11, 12] enables data-driven discovery of governing equations from data. Notice that to use this method, derivative data $\dot{X}$ is required. While in most publications on the subject this [10, 11, 12] information is assumed. However, for our studies we assume that only the time series information is available. Here we modify the algorithm to apply to only subsets of the equation in order to perform equation discovery specifically on the trained neural network, and in our modification the $\dot{X}$ term is replaced with $U_\theta(t)$, the output of the universal approximator, and thus is directly computable from any trained UDE.

After training the UDE, choose a set of state variables:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^T(t_1) \\ \mathbf{x}^T(t_2) \\ \vdots \\ \mathbf{x}^T(t_m) \end{bmatrix} = \begin{bmatrix} x_1(t_1) & x_2(t_1) & \cdots & x_n(t_1) \\ x_1(t_2) & x_2(t_2) & \cdots & x_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ x_1(t_m) & x_2(t_m) & \cdots & x_n(t_m) \end{bmatrix} \tag{29}$$

and compute a the action of the universal approximator on the chosen states:

$$\dot{\mathbf{X}} = \begin{bmatrix} \mathbf{x}^T(t_1) \\ \dot{\mathbf{x}}^T(t_2) \\ \vdots \\ \mathbf{x}^T(t_m) \end{bmatrix} = \begin{bmatrix} \dot{x}_1(t_1) & \dot{x}_2(t_1) & \cdots & \dot{x}_n(t_1) \\ \dot{x}_1(t_2) & \dot{x}_2(t_2) & \cdots & \dot{x}_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ \dot{x}_1(t_m) & \dot{x}_2(t_m) & \cdots & \dot{x}_n(t_m) \end{bmatrix} \tag{30}$$

Then evaluate the observations in a basis $\Theta(X)$. For example:

$$\Theta(\mathbf{X}) = \begin{bmatrix} 1 & \mathbf{X} & \mathbf{X}^{P_2} & \mathbf{X}^{P_3} & \cdots & \sin(\mathbf{X}) & \cos(\mathbf{X}) & \cdots \end{bmatrix} \tag{31}$$

where X^{P_i} stands for all P_ith order polynomial terms such as

$$\mathbf{X}^{P_2} = \begin{bmatrix} x_1^2(t_1) & x_1(t_1)\,x_2(t_1) & \cdots & x_2^2(t_1) & \cdots & x_n^2(t_1) \\ x_1^2(t_2) & x_1(t_2)\,x_2(t_2) & \cdots & x_2^2(t_2) & \cdots & x_n^2(t_2) \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ x_1^2(t_m) & x_1(t_m)\,x_2(t_m) & \cdots & x_2^2(t_m) & \cdots & x_n^2(t_m) \end{bmatrix} \tag{32}$$

Using these matrices, find this sparse basis $\boldsymbol{\Xi}$ over a given candidate library $\boldsymbol{\Theta}$ by solving the sparse regression problem $\dot{X} = \boldsymbol{\Theta}\boldsymbol{\Xi}$ with L_1 regularization, i.e. minimizing the objective function $\left\|\dot{\mathbf{X}} - \boldsymbol{\Theta}\boldsymbol{\Xi}\right\|_2 + \lambda \|\boldsymbol{\Xi}\|_1$. This method and other variants of SInDy applied to UDEs, along with specialized optimizers for the LASSO L_1 optimization problem, have been implemented by the authors and collaborators DataDrivenDiffEq.jl library for use with the DifferentialEquations.jl training framework.

3.1 Application to the 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 $\alpha = 1.3$, $\beta = 0.9$, $\gamma = 0.8$, $\delta = 1.8$. The trajectory has been perturbed with additive noise drawn from a normal distribution scaled by 10^{-3}.

The neural network consists of an input layer, one hidden layers with 32 neurons and a linear output layer. The input and hidden layer have hyberbolic tangent activation functions. We trained for 200 iterations with ADAM with a learning rate $\gamma = 10^{-2}$. We then switched to BFGS with an initial stepnorm of $\gamma = 10^{-2}$ setting the maximum iterations to 10000. Typically the training converged after 400-600 iterations in total. 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 between 10^{-2} and 10^{-5} for the knowledge-enhanced neural network.

From the trained neural network, data was sampled over the original trajectory and fitted using the SR3 method with varying threshold between 10^{-7} and 10^3 logarithmically spaced to ensure a sparse solution. A pareto optimal

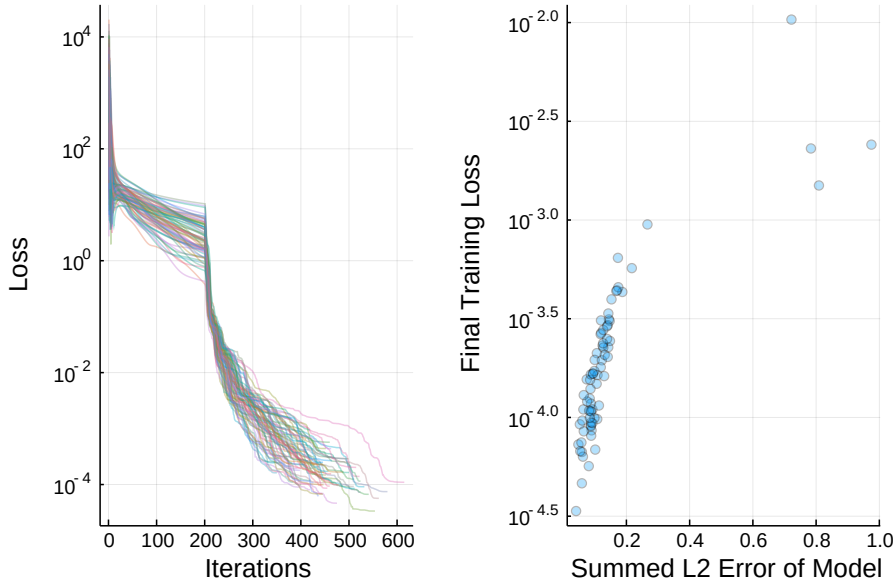

Figure 1: Loss trajectories and errors for 74 runs of the partially reconstruction of the Lotka Volterra equations under noisy data.

solution was selected via the L1 norm of the coefficients and the L2 norm of the corresponding error between the differential data and estimate. 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 an additional sparse regression over the quadratic terms. The resulting parameters extracted are $\beta \approx 0.9239$ and $\gamma \approx 0.8145$. Performing a sparse identification on the ideal, full solution and numerical derivatives computed via an interpolating spline resulted in a failure.

To ensure reproducible results, the training procedure has been evaluated 74 times given the initial trajectory. The training failed once (1.35 %) to exactly identify the original equation in symbolic terms. The final loss has a mean of 0.0004 with a standard error of 0.0012, an upper and lower bound of 0.0103 and 3.3573 10^{-5} respectively. The training converged between 241 and 615 iterations. The sum of the L2 error of the recovered model on the training data is centered around 0.1472 with a standard deviation of 0.1688. The results are shown in Figure 1.

3.2 Discovery of Robertson's Equations with Prior Conservation Laws

On Robertson's equations, we trained a UDAE model against a trajectory of 10 points on the timespan $t \in [0.0, 1.0]$ starting from $y_1 = 1.0$, $y_2 = 0.0$, and

$y_3 = 0.0$. The parameters of the generating equation were $k_1 = 0.04$, $k_2 = 3e7$, and $k_3 = 1e4$. The universal approximator was a neural network with one hidden layers of size 64. The equation was trained using the BFGS optimizer to a loss of $9e-6$.

4 Model-based Learning for the Fisher-KPP Equations

To generate training data for the 1D Fisher-KPP equation 15, 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

$$\rho_0(x) = \frac{1}{2}\left(\tanh\left(\frac{x - (0.5 - \Delta/2)}{\Delta/10}\right) - \tanh\left(\frac{x - (0.5 + \Delta/2)}{\Delta/10}\right)\right), \qquad (33)$$

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 16, 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

$$\mathcal{L} = \sum_i (\rho(x_i, t_i) - \rho_{\mathrm{data}}(x_i, t_i))^2 + \lambda |w_1 + w_2 + w_3|, \qquad (34)$$

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}.

5 Adaptive Solving for the 100 Dimensional Hamilton-Jacobi-Bellman Equation

5.1 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:

$$\begin{aligned} \frac{\partial u}{\partial t}(t,x) + \frac{1}{2}\mathrm{tr}\left(\sigma\sigma^T(t,x)\left(\mathrm{Hess}_x u\right)(t,x)\right) \\ + \nabla u(t,x)\cdot\mu(t,x) \\ + f\left(t,x,u(t,x),\sigma^T(t,x)\nabla u(t,x)\right) = 0, \end{aligned} \tag{35}$$

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 $d \times d$ 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

$$dX_t = \mu(t, X_t)dt + \sigma(t, X_t)dW_t \tag{36}$$

with a terminal condition $u(T,x) = g(x)$. With initial condition $X(0) = \zeta$ has shown that the solution to 17 satisfies the following forward-backward SDE (FBSDE) [13]:

$$\begin{aligned} u(t,&X_t) - u(0,\zeta) = \\ &- \int_0^t f(s, X_s, u(s, X_s), \sigma^T(s, X_s)\nabla u(s, X_s))ds \\ &+ \int_0^t \left[\nabla u(s, X_s)\right]^T \sigma(s, X_s)dW_s, \end{aligned} \tag{37}$$

with terminating condition $g(X_T) = u(X_T, W_T)$. Notice that we can combine 36 and 37 into a system of $d + 1$ SDEs:

$$\begin{aligned} dX_t =& \mu(t, X_t)dt + \sigma(t, X_t)dW_t, \\ dU_t =& f(t, X_t, U_t, \sigma^T(t, X_t)\nabla u(t, X_t))dt \\ &+ \left[\sigma^T(t, X_t)\nabla u(t, X_t)\right]^T dW_t, \end{aligned} \tag{38}$$

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:

$$\begin{aligned} \sigma^T(t, X_t)\nabla u(t, X_t) &\approx U^1_{\theta_1}(t, X_t), \\ u(0, X_0) &\approx U^2_{\theta_2}(X_0), \end{aligned} \tag{39}$$

Therefore we can rewrite 38 as a stochastic UDE of the form:

$$\begin{aligned} dX_t &= \mu(t, X_t)dt + \sigma(t, X_t)dW_t, \\ dU_t &= f(t, X_t, U_t, U^1_{\theta_1}(t, X_t))dt + \left[U^1_{\theta_1}(t, X_t)\right]^T dW_t, \end{aligned} \tag{40}$$

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:

$$l(\theta_1, \theta_2 | X_T, U_T) = \mathbb{E}\left[\|g(X_T) - U_T\|\right]. \tag{41}$$

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.

5.2 The LQG Control Problem

This PDE can be rewritten into the canonical form by setting:

$$\begin{aligned} \mu &= 0, \\ \sigma &= \overline{\sigma} I, \\ f &= -\alpha \left\|\sigma^T(s, X_s)\nabla u(s, X_s))\right\|^2, \end{aligned} \tag{42}$$

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 [14] 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:

$$u(t, x) = -\frac{1}{\lambda}\ln\left(\mathbb{E}\left[\exp\left(-\lambda g(x + \sqrt{2}W_{T-t}\right)\right]\right), \tag{43}$$

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

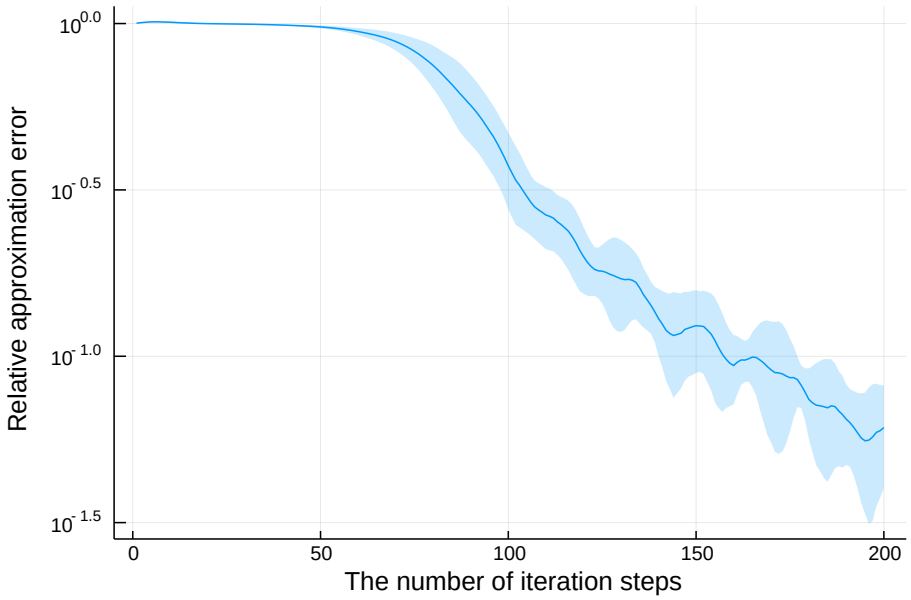

Figure 2: Adaptive solution of the 100-dimensional Hamilton-Jacobi-Bellman equation. This demonstrates that as the universal approximators $U^1_{\theta_1}$ and $U^2_{\theta_2}$ converge to satisfy the terminating condition, $U^2_{\theta_2}$ network convergences to the solution of Equation 20.

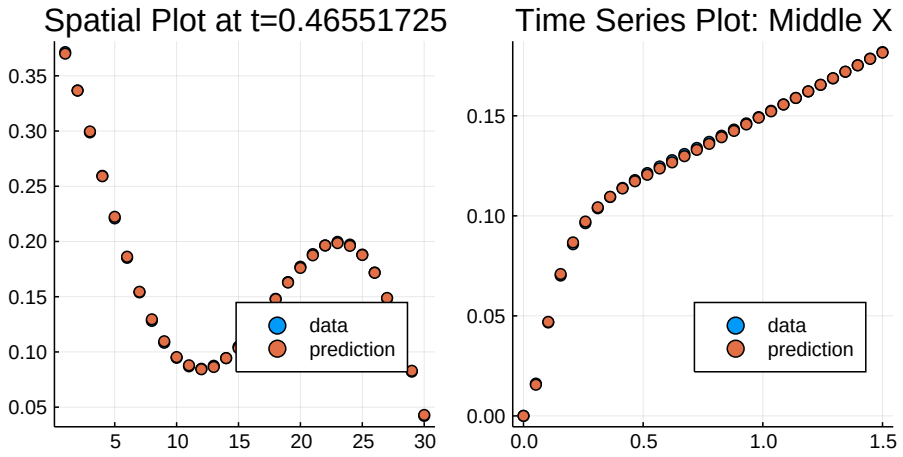

Figure 3: Reduction of the Boussinesq equations. On the left is the comparison between the training data (blue) and the trained UPDE (orange) over space at the 10th fitting time point, and on the right is the same comparison shown over time at spatial midpoint.

6 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 3 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 [15]. It was set to utilize adaptive time stepping to maximize the time step according to the CFL condition number (capped at CFL $\leq$ 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 [16]. The resulting simulation took 13,737 seconds on the same computer used for the neural diffusion-advection approach, demonstrating the approximate 15,000x acceleration.

7 Automated Derivation of Closure Relations for Viscoelastic Fluids

The full FENE-P model is:

$$\boldsymbol{\sigma} + g\left(\frac{\lambda}{f(\boldsymbol{\sigma})}\boldsymbol{\sigma}\right) = \frac{\eta}{f(\boldsymbol{\sigma})}\dot{\boldsymbol{\gamma}}, \tag{44}$$

$$f(\boldsymbol{\sigma}) = \frac{L^2 + \frac{\lambda(L^2-3)}{L^2\eta}\mathrm{Tr}(\boldsymbol{\sigma})}{L^2 - 3}, \tag{45}$$

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 [17]. 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:

$$G(t) = 2\eta\delta(t) + G_0 e^{-t/\tau}, \tag{46}$$

with the exact closure relation:

$$\sigma(t) = \eta\dot{\gamma}(t) + \phi, \tag{47}$$

$$\frac{\mathrm{d}\phi}{\mathrm{d}t} = G_0\dot{\gamma} - \phi/\tau. \tag{48}$$

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,

$$\sigma = U_0(\phi, \dot{\gamma}) \tag{49}$$

$$\frac{\mathrm{d}\phi}{\mathrm{d}t} = U_1(\phi, \dot{\gamma}) \tag{50}$$

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.

References

[1] Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckus, Elliot Saba, Viral B Shah, and Will Tebbutt. Zygote: A differentiable programming system to bridge machine learning and scientific computing. *arXiv preprint arXiv:1907.07587*, 2019.

[2] Alan C Hindmarsh, Peter N Brown, Keith E Grant, Steven L Lee, Radu Serban, Dan E Shumaker, and Carol S Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396, 2005.

[3] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in neural information processing systems*, pages 6571–6583, 2018.

[4] J. Revels, M. Lubin, and T. Papamarkou. Forward-mode automatic differentiation in julia. *arXiv:1607.07892 [cs.MS]*, 2016.

[5] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. Fashionable modelling with flux. *CoRR*, abs/1811.01457, 2018.

[6] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I (2nd Revised. Ed.): Nonstiff Problems*. Springer-Verlag, Berlin, Heidelberg, 1993.

[7] Francesca Mazzia and Cecilia Magherini. Test set for initial value problem solvers, release 2.4. Technical Report 4, Department of Mathematics, University of Bari, Italy, February 2008.

[8] Francesca Mazzia and Cecilia Magherini. *Test Set for Initial Value Problem Solvers, release 2.4*. Department of Mathematics, University of Bari and INdAM, Research Unit of Bari, February 2008.

[9] Andreas Rößler. Runge–kutta methods for the strong approximation of solutions of stochastic differential equations. *SIAM Journal on Numerical Analysis*, 48(3):922–952, 2010.

[10] Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, 2016.

[11] Niall M Mangan, Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Inferring biological networks by sparse identification of nonlinear dynamics. *IEEE Transactions on Molecular, Biological and Multi-Scale Communications*, 2(1):52–63, 2016.

[12] Niall M Mangan, J Nathan Kutz, Steven L Brunton, and Joshua L Proctor. Model selection for dynamical systems via sparse regression and information criteria. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 473(2204):20170009, 2017.

[13] Jianfeng Zhang. Backward stochastic differential equations. In *Backward Stochastic Differential Equations*, pages 79–99. Springer, 2017.

[14] Christopher Rackauckas and Qing Nie. Differentialequations.jl – a performant and feature-rich ecosystem for solving differential equations in julia. *The Journal of Open Research Software*, 5(1), 2017. Exported from https://app.dimensions.ai on 2019/05/05.

[15] Ali Ramadhan, Gregory LeClaire Wagner, Chris Hill, Jean-Michel Campin, Valentin Churavy, Tim Besard, Andre Souza, Alan Edelman, John Marshall, and Raffaele Ferrari. Oceananigans.jl: Fast and friendly geophysical fluid dynamics on GPUs. *The Journal of Open Source Software*, 4(44):1965, 2020.

[16] Benoit Cushman-Roisin and Jean-Marie Beckers. Chapter 4 - equations governing geophysical flows. In Benoit Cushman-Roisin and Jean-Marie Beckers, editors, *Introduction to Geophysical Fluid Dynamics*, volume 101 of *International Geophysics*, pages 99 – 129. Academic Press, 2011.

[17] P.J. Oliveira. Alternative derivation of differential constitutive equations of the oldroyd-b type. *Journal of Non-Newtonian Fluid Mechanics*, 160(1):40 – 46, 2009. Complex flows of complex fluids.