Title

Bayesian polynomial neural networks and polynomial neural ordinary differential equations.

Journal

PLoS Computational Biology, 20(10)

Authors

Fronk, Colby

Yun, Jaewoong

Singh, Prashant

et al.

Publication Date

2024-10-01

DOI

10.1371/journal.pcbi.1012414

Peer reviewed

PLOS COMPUTATIONAL BIOLOGY

RESEARCH ARTICLE

Bayesian polynomial neural networks and polynomial neural ordinary differential equations

Colby Fronk[1]*, **Jaewoong Yun**[2,3], **Prashant Singh**[4], **Linda Petzold**[5,6]

1 Department of Chemical Engineering, University of California, Santa Barbara, California; United States of America, **2** Department of Statistics and Applied Probability, University of California, Santa Barbara, California; United States of America, **3** Department of Geography, University of California, Santa Barbara, California; United States of America, **4** Science for Life Laboratory, Department of Information Technology, Uppsala University, Uppsala, Sweden, **5** Department of Mechanical Engineering, University of California, Santa Barbara, California; United States of America, **6** Department of Computer Science, University of California, Santa Barbara, California; United States of America

* colbyfronk@ucsb.edu

Citation: Fronk C, Yun J, Singh P, Petzold L (2024) Bayesian polynomial neural networks and polynomial neural ordinary differential equations. PLoS Comput Biol 20(10): e1012414. https://doi.org/10.1371/journal.pcbi.1012414

Editor: Feng Fu, Dartmouth College, UNITED STATES OF AMERICA

Received: December 13, 2023

Accepted: August 14, 2024

Published: October 10, 2024

Data Availability Statement: The data supporting the findings of this study are available in the GitHub repository at https://github.com/colbyfronk/BayesNeuralODE/.

Funding: This research was funded in whole by the National Institute of Health (grant 2-R01-EB014877-04A1 to LRP), https://www.nih.gov/. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing interests: The authors have declared that no competing interests exist.

Abstract

Symbolic regression with polynomial neural networks and polynomial neural ordinary differential equations (ODEs) are two recent and powerful approaches for equation recovery of many science and engineering problems. However, these methods provide point estimates for the model parameters and are currently unable to accommodate noisy data. We address this challenge by developing and validating the following Bayesian inference methods: the Laplace approximation, Markov Chain Monte Carlo (MCMC) sampling methods, and variational inference. We have found the Laplace approximation to be the best method for this class of problems. Our work can be easily extended to the broader class of symbolic neural networks to which the polynomial neural network belongs.

Author summary

Polynomial neural ordinary differential equations (ODEs) are a recent approach for symbolic regression of dynamical systems governed by polynomials. However, they are limited in that they provide maximum likelihood point estimates of the model parameters. The domain expert using system identification often desires a specified level of confidence or range of parameter values that best fit the data. In this work, we use Bayesian inference to provide posterior probability distributions of the parameters in polynomial neural ODEs. To date, there are no studies that attempt to identify the best Bayesian inference method for neural ODEs and symbolic neural ODEs. To address this need, we explore and compare three different approaches for estimating the posterior distributions of weights and biases of the polynomial neural network: the Laplace approximation, Markov Chain Monte Carlo (MCMC) sampling, and variational inference. We have found the Laplace approximation to be the best method for this class of problems. We have also

developed lightweight JAX code to estimate posterior probability distributions using the Laplace approximation.

Introduction

The development of a mathematical model is critical to understanding complex chemical, biological, and mechanical processes. For example, ordinary differential equation (ODE) models are used in the field of epidemiology to describe the spread of diseases such as flu, measles, and COVID-19 and in the medical field to describe the population dynamics of CD4 T-cells in the human body during an HIV infection. Developing a mathematical model with sufficient detail is important because it can be used to identify potential methods of intervention (such as a drug) for an undesired outcome (such as the propagation of a disease). Scientists devote years to the model development cycle, which is the process of finding a model that describes a process, using data to fit parameters to the model, analyzing uncertainties in the fitted parameters, and performing additional experiments to refine and validate the model. However, these mechanistic models are powerful due to their ability to directly explain the system with known first principles such as the interaction of forces, conservation of energy in the system (thermodynamics and heat transfer), and conservation of mass (transport processes). Based on the underlying assumptions of the model, scientists know where the model can and cannot be applied to make predictions about what will happen under certain scenarios. For these reasons, mechanistic models are preferred by scientists and engineers. However, since these models entail a long development time, we need to develop new tools to accelerate and aid the model development cycle.

A relatively recent development in the system identification field is the method Sparse Identification of Nonlinear Dynamics (SINDy) [1–3], which is linear regression of time derivatives estimated from numerical differentiation methods against a list of candidate terms which the modeler believes could be in the system to determine the terms in an ODE model. SINDy has been shown to be very successful with recovering ODE equations from various fields including fluid dynamics [4], plasma physics [5], biological chemical reaction networks [6, 7], and nonlinear optical communication [8]. Like any method, SINDy is not perfect and has its flaws. For example, it has been shown that SINDy requires its training data to be observed at very close intervals of time [9].

The internet of things [10, 11] has led to an exponential growth in the amount of data being generated and stored. We have more data than can be effectively processed. For example, the emergence of robots that can speedup small-scale lab experiments in chemistry and biology [12, 13] has led to a substantially larger amount of more accurate experimental data. In the earth sciences, the growing number of satellites and in situ earth observation equipment stationed around the world [14] has led to a significant amount of data that must be processed and understood. The emergence of the GPU, along with more powerful CPUs, has allowed data-driven models such as deep learning [15] to emerge as a viable way to process and understand large amounts of data quickly.

Neural ordinary differential equations [16–25] (ODEs) are a recent deep learning approach to data-driven modeling of time-series data and dynamical systems. In Neural ODEs (NODEs), a neural network learns the right hand side of a system of ODEs. The neural ODE is integrated forward in time from an initial condition to make a prediction. In contrast to SINDy, neural ODEs have less stringent requirements on the sampling rate, number of observed data points, and can handle irregularly spaced data points [9]. A cousin of the neural

ODE is the physics-informed neural network [26–29, 29–32] (PINN), which attempts to accomplish the same thing but with a different approach to loss functions.

Neural differential equations and physics-informed neural networks are two powerful tools because a large majority of science and engineering models are described in terms of differential equations. However, these tools suffer from the same major problem as the entire family of deep learning tools—they are black-box models that are not interpretable and cannot be generalized well to regimes of conditions outside of the region it was trained on. This is an issue for scientists and engineers who need reliable models.

In response to the need for interpretability and mechanistic models, symbolic neural networks have emerged. There has been a recent explosion in the introduction of various symbolic neural network architectures [9, 33–40], which essentially embed mathematical terms within the architecture. Most of these architectures can be combined with neural differential equation or physics-informed neural network frameworks to recover interpretable symbolic equations [9] that the scientist can immediately use. This is referred to as symbolic regression with neural networks.

Most of these symbolic neural network approaches have been demonstrated on noiseless data only; however, real data is almost always noisy. Additionally, the scientist using the tool often requires uncertainty estimates for the inferred model parameters; however, most of these symbolic neural network approaches recover only point estimates for the model's parameters. Bayesian inference is one approach to handle noisy data for symbolic neural networks and symbolic neural ODEs. There has been a substantial amount of work on Bayesian neural networks [41] and some work on Bayesian neural ODEs [18, 42]. However, there is a lack of approaches attempting to find the optimal Bayesian inference method for symbolic neural networks and symbolic neural ODEs. In our work, we explore various Bayesian inference methods and provide clarity to which Bayesian methods are best suited for this class of problems. We evaluate the Laplace approximation, Markov Chain Monte Carlo (MCMC) sampling methods, and variational inference on our previously developed approach for symbolic regression with polynomial neural networks [9, 33] and polynomial neural ordinary differential equations [9]. Our code can easily be extended to the various other symbolic neural network architectures.

Methods

Neural ODEs

Neural Ordinary Differential Equations [16] are neural networks that learn an approximation to time-series data, $y(t)$, in the form of an ODE system. In many fields of science, the ODE system for which we would like to learn an approximation has the form

$$\frac{dy(t)}{dt} = f(t, y(t), \theta), \tag{1}$$

where t is time, $y(t)$ is the vector of state variables, θ is the vector of parameters, and f is the ODE model. Finding the exact system of equations for f is a very difficult and time-consuming task. With the help of the universal approximation theorem [43], a neural network (NN) is used to approximate the model f,

$$\frac{dy(t)}{dt} = f \approx NN(t, y(t), \theta). \tag{2}$$

Neural ODEs can be treated like standard ODEs. Predictions for the time series data are obtained by integrating the neural ODE from an initial condition with a discretization scheme [44–46], in exactly the same way as it is done for a standard ODE.

Learning missing terms from an ODE model with neural ODEs

When one doesn't know anything about the system's underlying equations, neural ODEs can learn the entire model:

$$\frac{dy(t)}{dt} = NN(t, y(t), \theta). \tag{3}$$

Often, parts of the model are known, f_{known}, but the modeler doesn't know all of the mechanisms and terms that describe the entire model. In this case, we can have the neural ODE learn the missing terms:

$$\frac{dy(t)}{dt} = f_{known}(t, y(t), \theta) + NN(t, y(t), \theta). \tag{4}$$

Learning the missing terms does not require significant special treatment, apart from including the known terms in the training process.

Polynomial neural ODEs

Systems in numerous fields are expressed as differential equations with the right-hand side functions f as polynomials. Examples include gene regulatory networks [47] and cell signaling networks [48] in systems biology, chemical kinetics [49], and population models in ecology [50] and epidemiology [51]. Polynomial neural ODEs are useful for this class of inverse problems in which it is known a priori that the system is described by polynomials.

Polynomial neural networks [33, 52] are neural network architectures in which the output is a polynomial transformation of the input layer. Polynomial neural networks belong to the larger class of symbolic neural network architectures. There are several different types of polynomial neural networks. For more information about these architectures, we refer the curious reader to Grigorios G. Chrysos's work. We had the most success with Ref. [33]'s π-net V1, which is shown in Fig 1. The architecture is centered around Hadamard products [53] of linear layers without activation functions, L_i:

$$L_i(x) = x * w_i + b_i \tag{5}$$

to form higher-order polynomials. The architecture must be specified upfront based on the desired polynomial degree. There are no tuning parameters in the architecture. A π-net can output any n-degree polynomial for the given state variables. The hidden layers can be larger or smaller than the input layer as long as the shape matches when the Hadamard product operation is performed.

Polynomial neural Ordinary Differential Equations [9] are polynomial neural networks embedded in the neural ODE framework [16]. Since the output of a polynomial neural ODE is a direct mapping of the input in terms of tensor and Hadamard products without nonlinear activation functions, symbolic math can be used to obtain a symbolic form of the neural network. Due to the presence of nonlinear activation functions in conventional neural networks, a symbolic equation cannot be directly obtained from conventional neural networks and conventional neural ODEs.

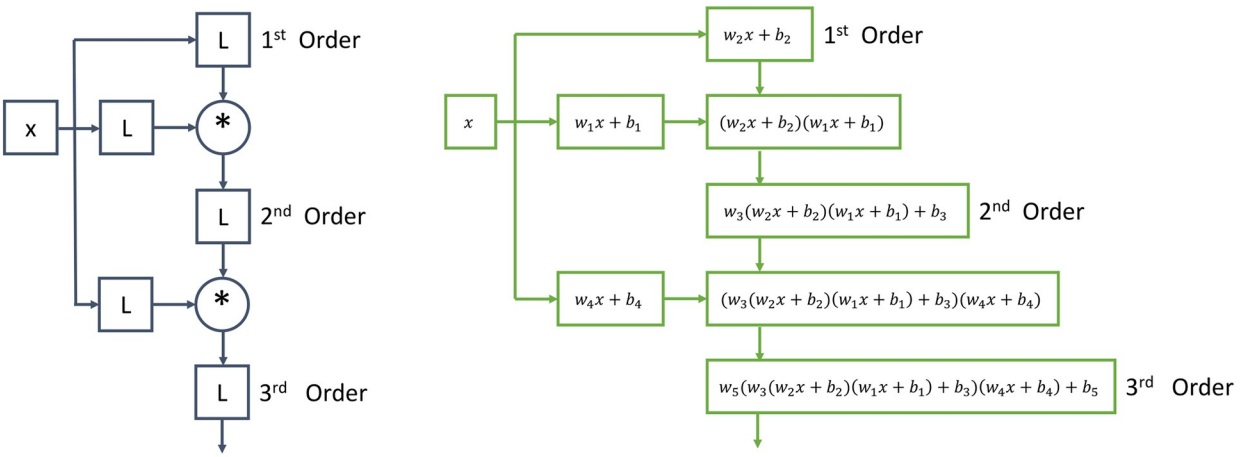

Fig 1. The neural network architecture of Ref. [33]'s π-net V1 is shown on the left. On the right, we show a worked example of what a 1-dimensional input layer with variable x symbolically looks like throughout the network architecture. The circles with the * symbol represent a layer that is the Hadamard product of the layer's inputs. The boxes labeled L represent standard linear layers without any activation functions. This neural network architecture has no standard activation functions such as tanh or ReLU, which makes it interpretable.

https://doi.org/10.1371/journal.pcbi.1012414.g001

Obtaining posterior distributions for weights and biases

We will explore and compare three different approaches for estimating the posterior distributions of weights and biases of the polynomial neural network. The approaches include the Laplace approximation, Markov Chain Monte Carlo (MCMC) sampling, and variational inference. The following text outlines each of them.

Approach #1: Laplace approximation. The Laplace approximation [54] provides Gaussian approximations of the individual posteriors. The Laplace approximation is obtained by taking the second-order Taylor expansion around the maximum a posteriori (MAP) estimate. For the polynomial neural network, approximating the log posterior over the parameters (θ), given some data (D) around a MAP estimate (θ^*), yields a normal distribution centered around θ^* with variance equal to the inverse of the Fisher information matrix ($\mathcal{I}_\theta$):

$$\theta \sim \mathcal{N}(\theta^*, \mathcal{I}_\theta^{-1}). \tag{6}$$

Under certain regularity conditions, the Fisher information matrix can be calculated via either the Hessian

$$\mathcal{I}_{\theta_{i,j}} = -\mathrm{E}\left[\frac{\partial^2}{\partial\theta_i\,\partial\theta_j}\log f(D,\theta)\right] \tag{7}$$

or the gradient

$$\mathcal{I}_{\theta_{i,j}} = \mathrm{E}\left[\left(\frac{\partial}{\partial\theta_i}\log f(D,\theta)\right)^T\left(\frac{\partial}{\partial\theta_j}\log f(D,\theta)\right)\right] \tag{8}$$

of the log-joint density function [55]. Both the gradient and Hessian are computed with the JAX [56, 57] automatic differentiation tool. As expected, we were able to obtain the same results for both methods. However, we found the calculation of the Hessian to be computationally expensive and it can only be practical for polynomial neural networks with a small

number of parameters. For this reason, we used the gradient to calculate the Fisher information.

The log-joint density function ($\log f(D, \theta)$) is defined by the log-likelihood ($\log f(D|\theta)$) and log-prior ($\log p_r(\theta)$):

$$\log f(D, \theta) \doteq \log \mathrm{f}(\mathrm{D}|\theta) + \log \mathrm{p_r}(\theta), \tag{9}$$

where $\doteq$ denotes equality up to an additive constant. When the observed noise ($y_{pred} - y_{known}$) is normally distributed with variance β^2, the log-likelihood is given by:

$$\log f(D|\theta) = -\frac{1}{2\beta^2}\sum (y_{pred} - y_{known})^2, \tag{10}$$

where y_{pred} is the predicted value by the polynomial neural network or polynomial neural ODE and y_{known} is the observed data. It is important to note that $\log f(D|\theta)$ depends on the parameters in the neural ODE (θ). The dependence on θ stems from the integration of the neural ODE with an ODE solver to obtain y_{pred}. In the case of Gaussian priors on the weights and biases with covariance α^2, the log-prior is given by:

$$\log p_r(\theta) = -\frac{1}{2}\theta^T \alpha^{-2}\theta. \tag{11}$$

We assume that we do not know β^2. We calculate it via the sample variance of $y_{pred} - y_{known}$ at the MAP point estimate.

The workflow for training Bayesian polynomial neural ODEs with the Laplace approximation is very similar to that for polynomial neural ODEs. Prior to the training process, the architecture is defined and the parameters in the network are initialized to values that yield initial coefficient values of the simplified polynomial in the range of 10^{-5} to 10^{-10}. The following steps are outlined in Algorithm 1.

The goal of the training process is to fit the neural ODE to the observed data for the state variables, y_{known}, as a function of time. The neural ODE is integrated with a differentiable ODE solver to obtain predictions for y_{known}, which we call y_{pred}. We used gradient descent [58, 59] and Adam [60] to optimize the log-joint density defined in Eq 9, with the constant term $\frac{1}{2\beta^2}$ dropped.

For the training process, we batch our observed data into N_t batch trajectories consisting of a certain number of consecutive data points in the time series (y_{known}). For each iteration (epoch) of gradient descent, we simultaneously solve N_t initial value problems corresponding to each of the batch trajectories, to obtain the predictions (y_{pred}).

In theory, one can use any differentiable discretization scheme to integrate the neural ODE forwards in time. The simpler the integration scheme, the smaller the memory and compute time costs. One can also obtain gradients for the parameters through the use of the popular continuous-time sensitivity adjoint method [16]. Direct backpropagation through complicated integration schemes have high memory costs and numerical stability issues, therefore continuous-time sensitivity adjoint method is often used for these cases. However, the adjoint method is very slow. It takes a few hours to train neural ODEs with the adjoint method, whereas it only takes a few minutes to train a neural ODE with direct backpropogation through an explicit discretization scheme. It is also important to point out that neither of these two approaches are perfect and more work needs to be done on developing differentiable ODE solvers for neural ODEs. For example, neither direct backpropogation through an explicit scheme nor the continuous-time sensitivity adjoint method can handle obtaining gradients for stiff neural ODEs [61].

Since the examples we present are for non-stiff ODEs, we do not require the adjoint or any advanced integration methods, and are able to use the fourth-order explicit Runge–Kutta–Fehlberg method [62] to solve the neural ODE. The advantage of using this method is efficient direct backpropagation through the explicit ODE scheme [9], which is computationally faster than the continuous-time sensitivity adjoint method. After the training process has converged, we have obtained the MAP estimate (θ^*) via MLE.

After obtaining θ^*, we can find the variance of the posterior by calculating the inverse of the Fisher Information Matrix. For overparameterized neural network models, the Fisher Information Matrix is often singular and cannot be inverted. In this case, an approximation to the inverse can be calculated by either the Moore–Penrose inverse [63] or by dropping the off-diagonal entries from the matrix [64]. We have had success with both of these methods for finding an approximation for the inverse of the Fisher information. For the case in which the matrix is invertible, the approximations have given similar results to the direct matrix inverse. All of our results calculate the inverse using the Moore–Penrose inverse [63]. We have prior experience using the Laplace approximation to obtain uncertainties for the output of a neural network. Based on our experience, the Moore–Penrose inverse can only be used on neural networks with less than 50,000 parameters. This is because it becomes too expensive to invert the singular Fisher information matrix.

Algorithm 1 Laplace Approximation Algorithm

1: **Input:** Training data D
2: **Step 1:** Train polynomial neural ODE on data D to find the parameters θ^*
3: **Step 2:** Calculate the posterior distribution with the Laplace approximation
4: **a.** Use the parameter estimates θ^* found in step 1 as the mean of the posterior
5: **b.** Calculate the Fisher information matrix

$$\mathcal{I}_{\theta_{i,j}} = \mathrm{E}\left[\left(\frac{\partial}{\partial\theta_i}\log f(D,\theta)\right)^T\left(\frac{\partial}{\partial\theta_j}\log f(D,\theta)\right)\right] \tag{12}$$

6: **c.** Invert the Fisher information matrix to find the covariance
7: **d.** The posterior is given by:

$$\theta \sim \mathcal{N}(\theta^*, \mathcal{I}_\theta^{-1}) \tag{13}$$

Approach #2: Markov Chain Monte Carlo. This approach for obtaining posterior distributions for the weights and biases of the polynomial neural network draws from Markov Chain Monte Carlo [65–67] (MCMC) methods for training Bayesian neural networks [68] (BNNs). The two MCMC sampling methods that we explored were Hamiltonian Monte Carlo (HMC) and The No-U-Turn-Sampler (NUTS).

Hamiltonian Monte Carlo [69, 70] (HMC) is a MCMC method that uses derivatives of the density function to generate efficient transitions. HMC starts with an initial set of parameter values. For a set number of iterations, a momentum vector is sampled and integrated following Hamiltonian dynamics [71] with the leapfrog [44] integrator with a set discretization time (ϵ) and number of steps (L). Since the leapfrog integrator incurs numerical error [44], it is corrected by use of the Metropolis–Hastings [72–75] acceptance algorithm, which helps to decide whether to accept or reject the new state predicted from Hamiltonian dynamics.

The No-U-Turn-Sampler [76] (NUTS) is an extension of HMC that automatically determines when the sampler should stop an iteration. The algorithm automatically chooses the discretization time and number of steps, which avoids the need for the user to specify these additional parameters. However, we have found this algorithm to be computationally more expensive than vanilla HMC for this class of problems.

The training process is slightly different than for the Laplace approximation. We still batched our observed data into N_t batch trajectories and simultaneously solved N_t initial value problems with the same fourth-order explicit Runge–Kutta–Fehlberg method. We used BlackJAX [77]'s sampling algorithms to do the MCMC inference. For both of these methods, we used the log-joint density defined in Eq 9. The workflow is explained in Algorithm 2.

Algorithm 2 MCMC Algorithm

Input: Initial state θ_0, number of iterations N
Step 1: Initialize $\theta = \theta_0$
for $i = 1$ to N **do**
  Step 2: Propose a new state θ' from a proposal distribution
  Step 3: Calculate the acceptance probability α
  Step 4: Accept or reject the new state based on α
  if the new state θ' is accepted **then**
    $\theta = \theta'$
  end if
  Step 5: Record the state θ
end for
Output: Collection of sampled states $\{\theta_i\}_{i=1}^{N}$ from the posterior distribution

Approach #3: Variational inference. In variational inference [78–80], we learn an approximation $q(\theta)$ to our posterior $p(\theta|D)$. Our approximation is assumed to belong to a certain family of probability density functions and the parameters of that family are optimized by minimizing the Kullback–Leibler (KL) divergence:

$$\mathrm{KL}(q(\theta)\,||\,p(\theta|D)) = \mathbb{E}_{q(\theta)}\left[\log \frac{q(\theta)}{p(\theta|D)}\right]. \tag{14}$$

We don't know the analytical form of the posterior so we cannot minimize the KL divergence directly, but we can use a trick called the Evidence Lower Bound (ELBO) [78–80]:

$$\mathrm{ELBO} = \mathbb{E}_{q(\theta)}[\log p(D|\theta)] - \mathrm{KL}(q(\theta)\,||\,p_r(\theta)). \tag{15}$$

Maximizing the ELBO is mathematically equivalent to minimizing the KL divergence. The ELBO only contains the prior $p_r(\theta)$ and likelihood $p(D|\theta)$, which we can numerically calculate.

The methodology is described in Algorithm 3. We wrote our own custom JAX code for variational inference. The neural ODEs are numerically integrated exactly the same way as was done for the Laplace approximation. We used a multivariate Gaussian distribution for the approximation $q(x)$ and found it difficult to learn a covariance matrix that remained positive semidefinite. As a result, we instead made the computation more tractable by using the mean field approximation for $q(x)$, in which $q(x)$ is the approximation consisting of independent Gaussian distributions for all of the parameters. We calculated the expectation in Eq 15 by sampling from the approximation $q(x)$. A sample size of 1000 was sufficient to keep the loss function fairly noise-free.

Algorithm 3 Variational Inference Algorithm

Input: Training data D
Step 1: Define the variational family $q(\theta)$ with parameters θ
Step 2: Initialize variational parameters θ
Step 3: Optimize the Evidence Lower Bound (ELBO)

5: **a.** Calculate the ELBO:

$$\text{ELBO} = \mathbb{E}_{q(\theta)}[\log p(D|\theta)] - \text{KL}(q(\theta)\,||\,p_r(\theta)) \tag{16}$$

6: **b.** Use gradient-based methods to update θ:

$$\theta \leftarrow \theta + \eta \nabla_\theta \mathcal{L}(\theta) \tag{17}$$

7: **Step 4:** Iterate Step 3 until convergence
8: **Step 5:** Use the optimized variational distribution $q(\theta)$ as an approximation to the posterior distribution

Obtaining posterior distributions for polynomial coefficients

The polynomial neural network is a factorized form of a polynomial. To obtain a simplified form of the polynomial we must expand the equation and combine like terms. For the case where the neural network parameters are scalar point estimates, we have already done this [9] with the use of SymPy [81]. When our parameters are Bayesian probability distributions, we must use the rules for the product and sum of probability distributions. These rules depend on the type of probability distributions that are algebraically combined, which makes it challenging to compute for even a small number of parameters (weights and biases). We explored approximating the weights and biases as independent univariate Gaussian probability density functions (PDFs), for which there are known rules [82] for the mean and variance of the product and sum of univariate Gaussian PDFs. However, this approach did not work in all cases since the weights and biases are dependent on each other.

To avoid multiplying out probability density functions of the weights and biases to obtain posterior distributions for the polynomial coefficients, we used Monte Carlo sampling. We drew random samples from the posterior distributions $w \sim P(w|D)$ and $b \sim P(b|D)$ for the weights (w) and biases (b) given the data (D). Both the Laplace approximation and variational inference give us multivariate normal distributions for the posterior distributions for the weights (w) and biases (b) of the neural network, which allowed us to use native functions in JAX, SciPy, and NumPy to directly sample from the multivariate normal posterior distributions. For MCMC and SMC ABC methods we used the samples obtained directly from these methods. For each sample, we used the approach of expanding the polynomial neural network for scalar point estimates [9]. After doing this for enough samples, we have an estimate of the posterior distribution $c \sim P(c|D)$ of the polynomial coefficients (c).

Strategies for handling large amounts of noise

Neural ODEs require initial conditions to generate predicted trajectories (y_{pred}) for the training process. When there is a large amount of observed noise in the training data, the known data points (y_{known}) cannot be used as initial conditions. When this is the case, we must use a time-series filtering or smoothing algorithm to find good initial conditions to use for the neural ODE training process. Example filtering algorithms include moving average [83] (MA), exponential moving average [84] (EMA), and Kalman filters [85]. Example smoothing algorithms include smoothing splines [86], local regression [87], kernel smoother [88], Butterworth filter [89], and exponential smoothing [90]. We applied all of these algorithms on noisy ODE time series data and found Gaussian process regression (GPR) [91] to be the most accurate approach. For brevity, we have chosen not to outline in detail the pros and cons of each of the

possible algorithms. However, it is important to note that the optimal smoothing algorithm is dependent on the data and the underlying model that describes it.

Gaussian process regression assumes a Gaussian process prior, which is specified with mean function $m(x)$ and covariance function or kernel $k(x, x')$:

$$f(x) \sim GP(m(x), k(x, x')). \tag{18}$$

The rational quadratic, Matérn, Exp-Sine-Squared kernel, and radial basis function kernels [92–94] were found to perform the best for our considered test problems and settings. We used the scikit-learn [95] Python library to perform our pre-processing with GPR. The hyper-parameters of the kernels were optimized using MLE.

Results

We will start by evaluating the methodology outlined on univariate cubic regression with a polynomial neural network. Starting with this model demonstrates that we can recover accurate Bayesian uncertainties on a standard polynomial without any ODEs. Since this problem can be posed as a Bayesian linear regression model with a closed form solution, we can directly test the accuracy of the methods and make sure they work prior to moving on to ODEs.

We then move on to the following ODE models: the Lotka-Volterra deterministic oscillator, the damped oscillator, and the Lorenz attractor. These models are common toy problems for dynamical systems and neural ODEs. The Lotka-Volterra model is a fairly easy model to identify. The damped oscillator is more difficult. In our previous work, we have shown that the dampening effect makes the vector field hard to learn [9]. Since the Lorenz attractor is chaotic and has high frequency oscillations, it is the most difficult model to learn. Since it is common in the sciences to have a partially incomplete model, we also demonstrate learning the missing terms from a partially known ODE model. For simplicity reasons, we have chosen to use the Lotka-Volterra model for learning the missing dynamics.

For each of the models outlined, we recover Bayesian posterior distributions for the model parameters and compare them to the known values. For the univariate cubic regression example, we plot the prediction along with credible intervals and confirm that the credible intervals capture the data well. For the ODE examples, we integrate the Bayesian ODE models from the known initial condition and compare it to the true trajectory. The criteria for choosing the best Bayesian inference method are: ease of use, computational cost, and accuracy.

Experiment 1: Univariate cubic regression

Prior to studying dynamical systems with neural ODEs, we tested our Bayesian polynomial neural network inference method on basic polynomials. For the test case, we used the following third order univariate function:

$$f(x) = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3. \tag{19}$$

For our experiment, we chose values of $\beta_0 = 1$, $\beta_1 = 1$, $\beta_2 = 2$, and $\beta_3 = 4$. The training data for the x-values consisted of a single data set of 200 uniformly spaced data points in the range -1.25 to 1.25. The values of $f(x)$ corresponding to the values of x were obtained by directly substituting the x-values into the function. We then added Gaussian noise with $\mu = 0$ and $\sigma^2 = 9$ to the training data. We chose this level of noise to demonstrate our methodology on data with a high level of noise. For reproducibility and comparison purposes, we used a random seed of 989 for all of the results we will show. Figs 2, 3, 4, 5 and 6 will be presented in this section of the results.

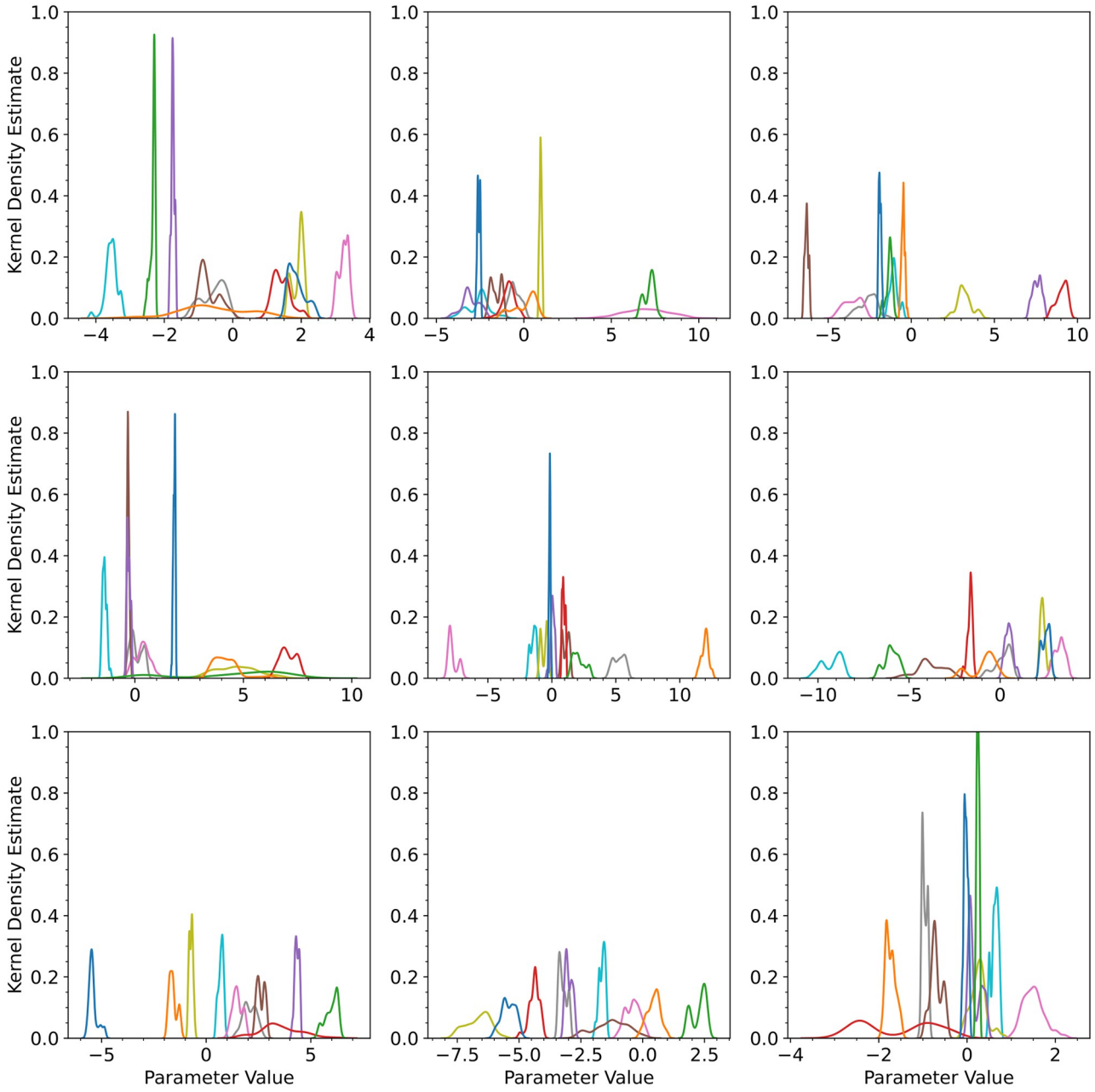

Fig 2. For the univariate cubic polynomial $f(x) = 1 + x + 2x^2 + 4x^3$, a third order Bayesian polynomial neural network was trained with the No-U-Turn-Sampler (NUTS) algorithm. The kernel density estimates for the posterior distributions of the weights and biases of the polynomial neural network are shown. The panes are sequentially ordered (left-to-right, top-to-bottom) from the first layer to the last layer in the neural network. There is no legend associated with the colors in the figure. The colors are only used to distinguish between posterior distributions of the parameters.

https://doi.org/10.1371/journal.pcbi.1012414.g002

The architecture from Ref. [9] was used for the Laplace approximation, the No-U-Turn Sampler (NUTS) method, and variational inference. The third order polynomial neural network we used had 1x10x10x10x10x1 neurons in each layer (180 total parameters). We experimented with changing the number of neurons in each hidden layer up to 200 and the results were similar. The extra parameters do not affect the posteriors significantly. For brevity, we do

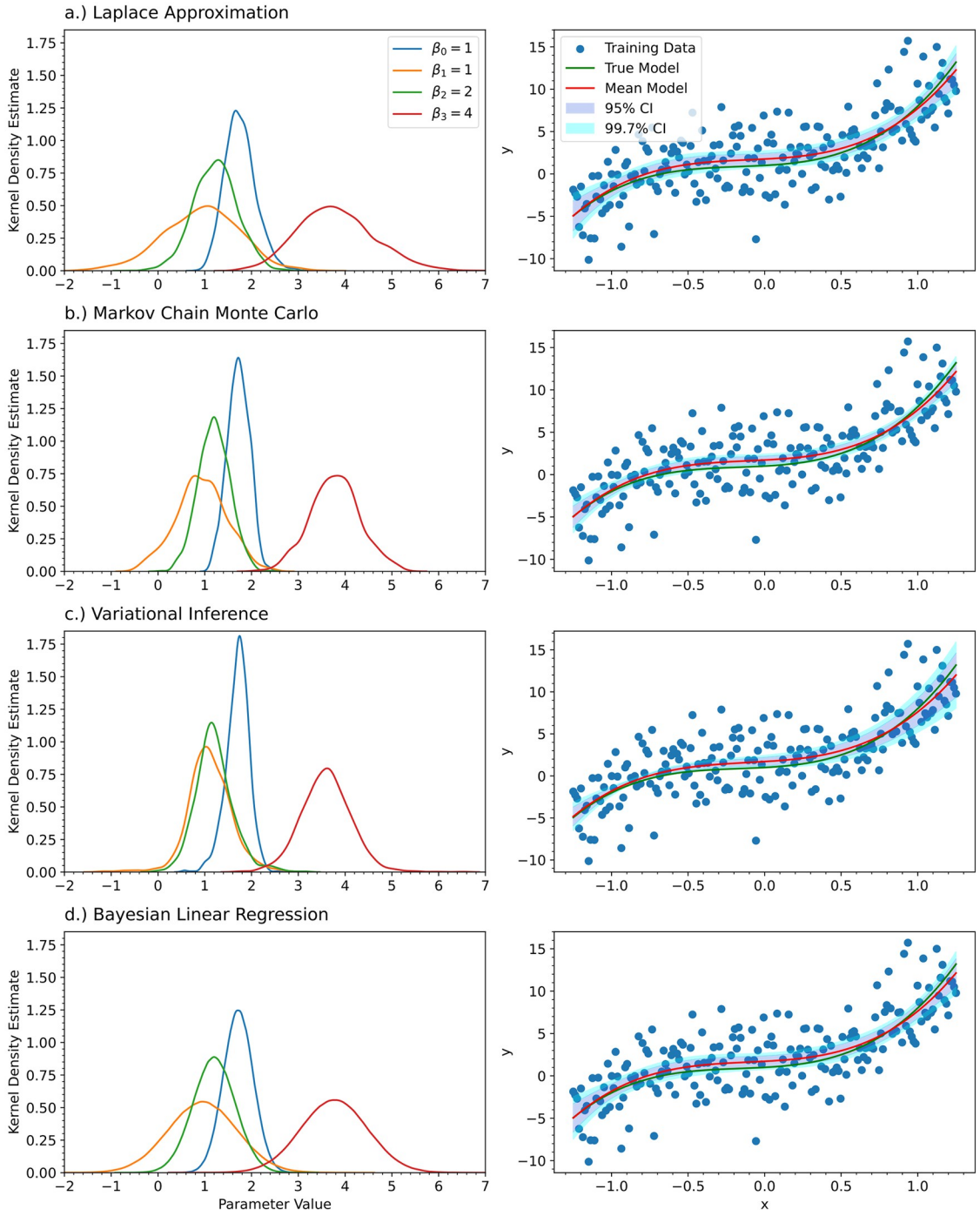

Fig 3. For the univariate cubic polynomial $f(x) = 1 + x + 2x^2 + 4x^3$, a third order Bayesian polynomial neural network was trained with a) the Laplace approximation, b) Markov Chain Monte Carlo with the No-U-Turn-Sampler (NUTS) algorithm, and c) Variational Inference. For comparision, d) Bayesian linear regression was also performed on the training data. The kernel density estimates for the posterior distributions of the polynomial coefficients are shown (left) along with their predictions and credible intervals (right). For the left column, the true value of the parameters is shown in the legend. Each of the columns share the same legend.

https://doi.org/10.1371/journal.pcbi.1012414.g003

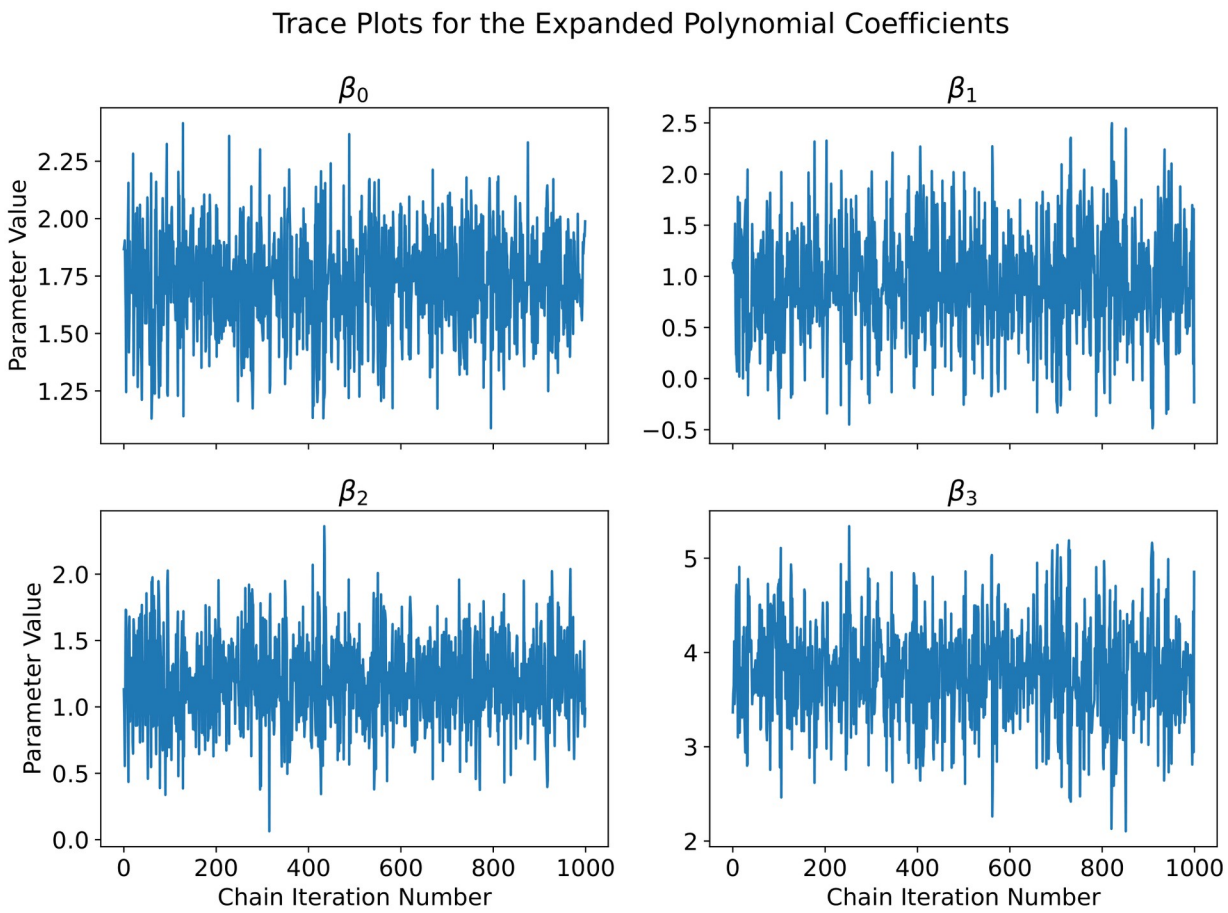

Fig 4. For the univariate cubic polynomial $f(x) = 1 + x + 2x^2 + 4x^3$, a third order Bayesian polynomial neural network was trained with Markov Chain Monte Carlo. To access the convergence and mixing of the Markov Chain, we show the trace plot for all of the expanded polynomial coefficients.

https://doi.org/10.1371/journal.pcbi.1012414.g004

not show these results. We used the Python libraries JAX [56, 57] along with Flax [96] for our neural networks.

For MCMC with NUTS, we used the Python library BlackJAX [77] to perform sampling. Since we had no prior knowledge of the weights and biases of the polynomial neural network but knew they weren't large values, we used the noninformative Gaussian prior with zero mean and standard deviation of 100000. We tested the sensitivity of the results with respect to the prior standard deviation, but found no significant effect. We omit these results for brevity. For MCMC, the warmup was set to 1000 steps and the number of steps taken following warmup was 1000. The code can also execute on a CPU within practical timeframes (a few extra minutes over GPU execution time). The GPU used was 1 core of NVIDIA GeForce RTX 3090 with 8 GB of memory. The CPU used was the Intel i7-8550U CPU 1.80GHz processor with 8 GB of memory. Since our neural network has a relatively small number of parameters, we plotted the kernel density estimates for the posterior distributions of the weights and biases of the polynomial neural network prior to expanding out the terms with Monte Carlo (see Fig 2). Most of the posterior distributions are close to being unimodal and symmetric. Some of the distributions have two bells, but the bells are very close together and rarely spaced far apart, which initially suggests that the Laplace approximation and variational inference with a

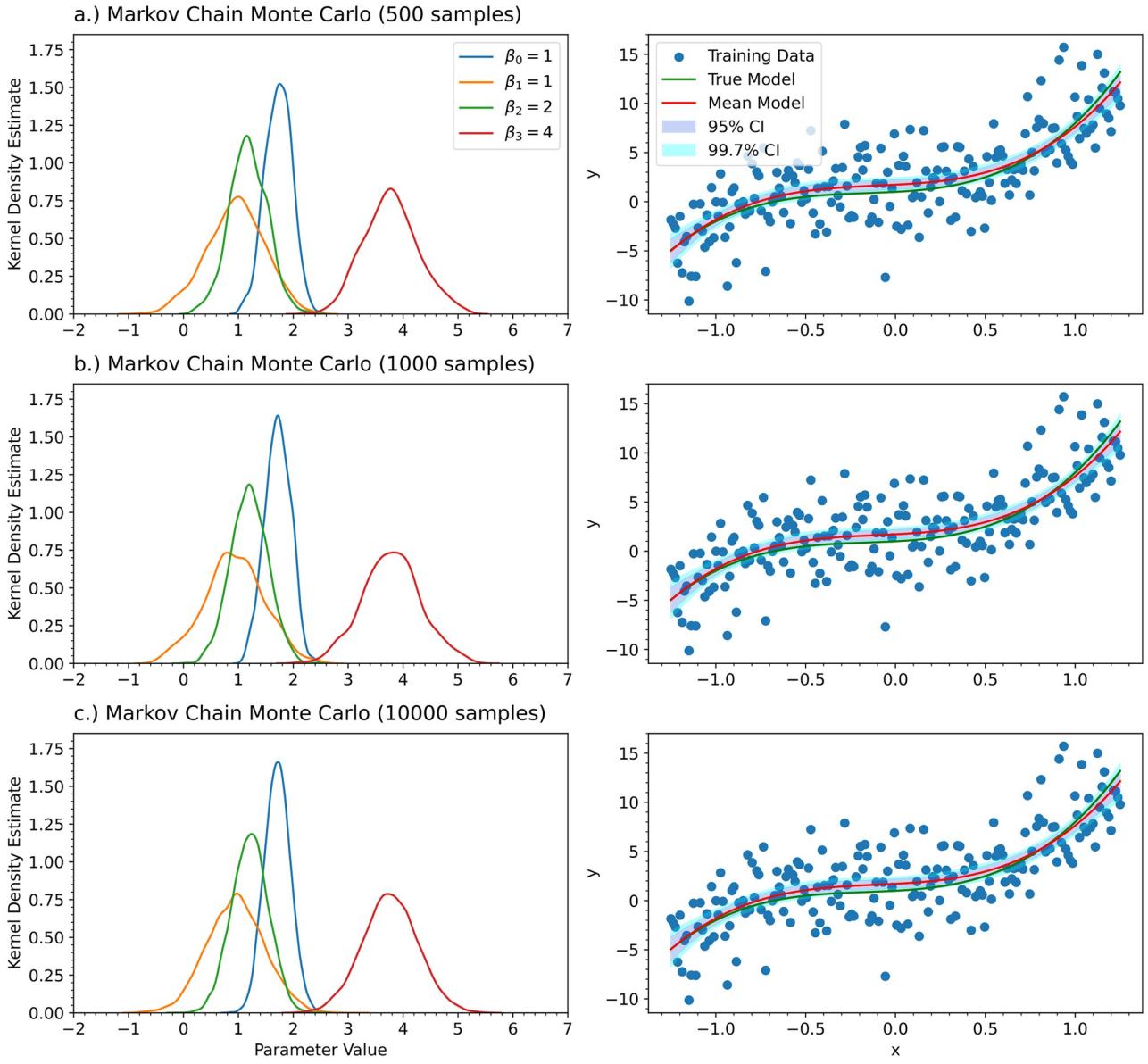

Fig 5. For the univariate cubic polynomial $f(x) = 1 + x + 2x^2 + 4x^3$, a third order Bayesian polynomial neural network was trained with Markov Chain Monte Carlo. To show sufficient Markov Chain length, we show the results of MCMC for the following sample sizes: a.) 500 samples, b) 1000 samples, and c) 10000 samples.

https://doi.org/10.1371/journal.pcbi.1012414.g005

multivariate Gaussian should work towards estimating the posteriors. The Laplace approximation approach takes significantly less time than MCMC—(1 minute vs 30 minutes). Our results from the section and the following section provide enough evidence to use the Laplace approximation.

Since this regression problem can also be posed as a Bayesian linear regression problem with a closed-form solution, we also solved it via simple Bayesian linear regression. For Bayesian linear regression, we write the model as $y = XB$. Since we have a Gaussian likelihood function and a conjugate Gaussian prior, the posterior distribution is defined by (see

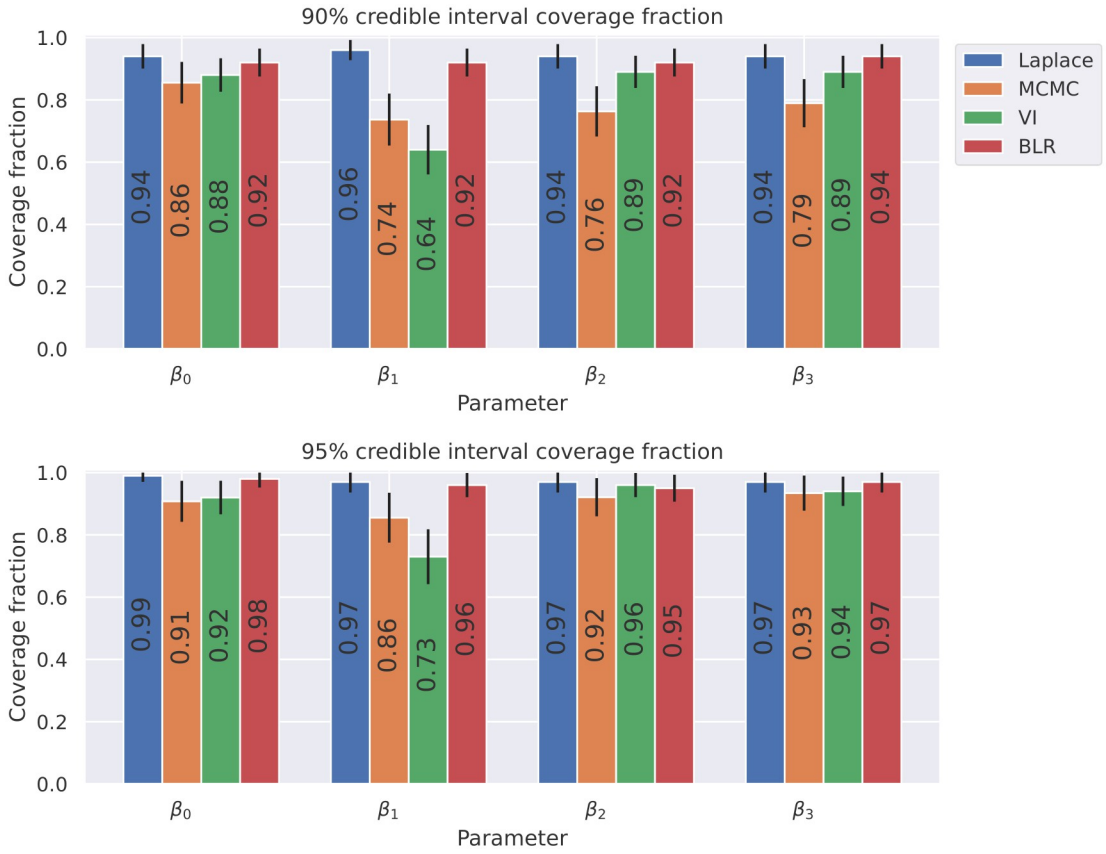

Fig 6. For the univariate cubic polynomial $f(x) = 1 + x + 2x^2 + 4x^3$, a third order Bayesian polynomial neural network was trained with the Laplace approximation, Markov Chain Monte Carlo with the No-U-Turn-Sampler (NUTS) algorithm, and Variational Inference. For comparision, Bayesian linear regression was also performed on the training data. We repeated the inference methods for 100 distinct datasets and calculated the fraction of the datasets in which the 90% and 95% credible intervals captured the true parameter value. The 90% and 95% confidence intervals for the 90% and 95% coverage fractions are also shown.

https://doi.org/10.1371/journal.pcbi.1012414.g006

Appendix):

$$p(B|D) = \mathcal{N}(\mu_B, \Sigma_B), \tag{20}$$

$$\Sigma_B = \left(\frac{X^T X}{\beta^2} + \frac{I}{\alpha^2}\right)^{-1}, \tag{21}$$

$$\mu_B = \frac{1}{\beta^2}\Sigma_B X^T y, \tag{22}$$

where the noise of the data (β^2) can be approximated by the sample variance of ($XB - y_{known}$). This approach did not use any neural networks as its intended purpose was solely method validation.

For Markov Chain Monte Carlo, we verified that our chain converged to our stationary distribution (the posterior) through the construction of a trace plot [67] as well as the calculation of the Geweke diagnostic number [67] for the expanded polynomial coefficients. The trace plot is shown in Fig 4. The trace plot shows good convergence as well as an okay level of mixing. We can quantitatively access the convergence of our Markov Chain by calculating the

Geweke diagnostic number, which is given by the following equation:

$$T = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}, \tag{23}$$

where $\bar{X}_1$ and $\bar{X}_2$ are the means of samples 1 and 2, s_1^2 and s_2^2 are the sample variances of samples 1 and 2, and n_1 and n_2 are the number of samples in samples 1 and 2 respectively [67]. Sample 1 was constructed by discarding the burn-in samples from the Markov Chain and using the first 10% of the remaining values. Sample 2 was constructed by using the remaining 50% of the values (with the burn-in values also discarded). Eq 23 is a two-sided hypothesis test that the mean of the first 10% and last 50% of the Markov Chain are the same. For each of the values of the expanded polynomial coefficients, we calculated the following Geweke diagnostic numbers: $T_{\beta_0} = -1.589$, $T_{\beta_1} = -0.942$, $T_{\beta_2} = 0.706$, and $T_{\beta_3} = 1.269$. At the confidence level of 0.05, the upper and lower tail of the t-distribution is ±1.964. This tells us that the means of the samples consisting of the first 10% and last 50% of the Markov Chain are statistically the same at the confidence level of 0.05, which is a good indicator that the Markov Chain has good convergence.

For Markov Chain Monte Carlo, we also need to justify the chosen Markov Chain length. We have repeated the MCMC experiment with sample sizes of 500, 1000, and 10000. Fig 5 shows how the results change as a function of the Markov Chain length. It can be seen that 1000 samples is sufficient. In Fig 3, we show the kernel density estimates for the posterior distributions of the coefficients of the polynomial for the Laplace approximation, MCMC with NUTS, variational inference, and Bayesian linear regression. Fig 3 also shows the model predictions corresponding to the posterior distributions found by each of the methods along with 95% and 99.7% credible intervals. Since the posterior distributions are symmetric and not skewed, all of the credible intervals shown are quantile-based. For skewed posterior distributions, Highest Posterior Density (HPD) credible intervals are more appropriate to use. The credible intervals were constructed by sampling from the posterior distribution and then evaluating the function for each of the samples obtained. The sample mean was calculated to give the mean model for the figure. The sample standard deviation was calculated and used to construct quantile-based credible intervals for the model prediction. This approach was also used for the ODE models in the other experiments.

According to Fig 3, all the methods have very similar results. MCMC predicted slightly narrower posteriors than Bayesian linear regression, whereas the Laplace approximation predicted slightly wider posteriors; however, MCMC is the most computationally expensive of the methods and scales the worst as the number of model parameters increases. Variational inference had the narrowest predictions for the posterior distributions, which resulted in a narrower credible interval for the function evaluation. Variational inference was also comparatively difficult to train. A notable amount of trial and error was required in order to guess plausible mean and covariance values to initialize the multivariate Gaussian approximation. Different initial mean and covariance matrices worked for each problem and there is no hyperparameter optimization that can be performed to speed this up. These problems will be addressed in future work. However, variational inference is still computationally cheaper than MCMC.

Fig 6, shows the coverage performance of the parameter credible intervals for each of the Bayesian inference methods. We generated 100 distinct datasets of 200 points with random seeds of 1, 2, 3, . . ., 100 respectively and then independently repeated the Bayesian inference methods with the same settings for each of the datasets. For each of the methods, we calculated the fraction of the datasets in which the 90% and 95% credible intervals for the parameters captured the true parameter value. The coverage performance is a good metric for comparing

how consistently each of the Bayesian methods were able to identify the parameters correctly. Our results show that MCMC is the least consistent method. Variational inference is slightly better than MCMC. The Laplace approximation has results the most similar to the Bayesian linear regression control experiment.

Experiment 2: Lotka-Volterra deterministic oscillator

Our first demonstration of Bayesian parameter estimates for polynomial neural ODEs is on the deterministic Lotka-Volterra ODE model [97, 98], which describes predator-prey population dynamics, such as an ecosystem of rabbits and foxes. When written as a set of first order nonlinear ODEs, the model is given by:

$$\frac{dx}{dt} = 1.5x - xy, \tag{24}$$

$$\frac{dy}{dt} = -3y + xy. \tag{25}$$

We generated our training data by integrating the initial value problem (IVP) with initial conditions $x = 1$ and $y = 1$ at $N = 100$ points uniformly spaced in time between 0 and 10. Since the Lotka-Volterra model is non-stiff, we used SciPy [99] and DOPRI5 [100], a fourth order accurate embedded method in the Runge–Kutta family of ODE solvers. We then generated 10 high-noise trajectories originating from the same initial value by adding zero-centered Gaussian noise with a standard deviation of 2 to the training data. This corresponds to a signal-to-noise ratio [101] (SNR or S/N) between 0.125 and 3.5. See Fig 7, ignoring the shaded GPR fit, to see the noisy training data. The architecture from Ref. [9] was used with 160 total parameters. Since we had no prior knowledge of the weights and biases of the polynomial neural network but knew they weren't large values, we used the noninformative Gaussian prior with zero mean and standard deviation of 100000. For MCMC, the warmup was set to 1000 steps and the number of steps taken following warmup was 1000. As discussed in the methods section, we batched our data into $N_t = 89$ trajectories of consisting of 12 consecutive data points from the time series. We simultaneously solved these batch trajectories during each epoch using our own JAX based differentiable ODE solver for the multistep fourth order explicit Runge–Kutta–Fehlberg method [62], which allows us to directly perform backpropagation through the ODE discretization scheme.

All of the Bayesian neural ODE approaches that we explored require integrating the neural ODE from starting initial conditions and comparing the prediction to the true data; however, since the data is extremely noisy, we cannot use the observed data points as initial conditions. To generate good initial guesses, we used Gaussian process regression (GPR) on the noisy data prior to the model training process. Since the Lotka-Volterra model is oscillatory, we used the Exp-Sine-Squared kernel [94] (also referred to as the periodic kernel), scaled by a constant kernel, along with the white kernel (W_k):

$$k(x_i, x_j) = c^2 \exp\left(-\frac{2\sin^2(\pi d(x_i, x_j)/p)}{l^2}\right) + W_k(\sigma^2_{GPR}). \tag{26}$$

where c^2 is the constant for the constant kernel, d is the euclidean distance function, l is the length-scale, p is the periodicity, and σ^2_{GPR} is the variance of the Gaussian noise [94]. We used MLE to obtain values for all of these unknown hyperparameters. See Fig 7 for the GPR fit on the observed data.

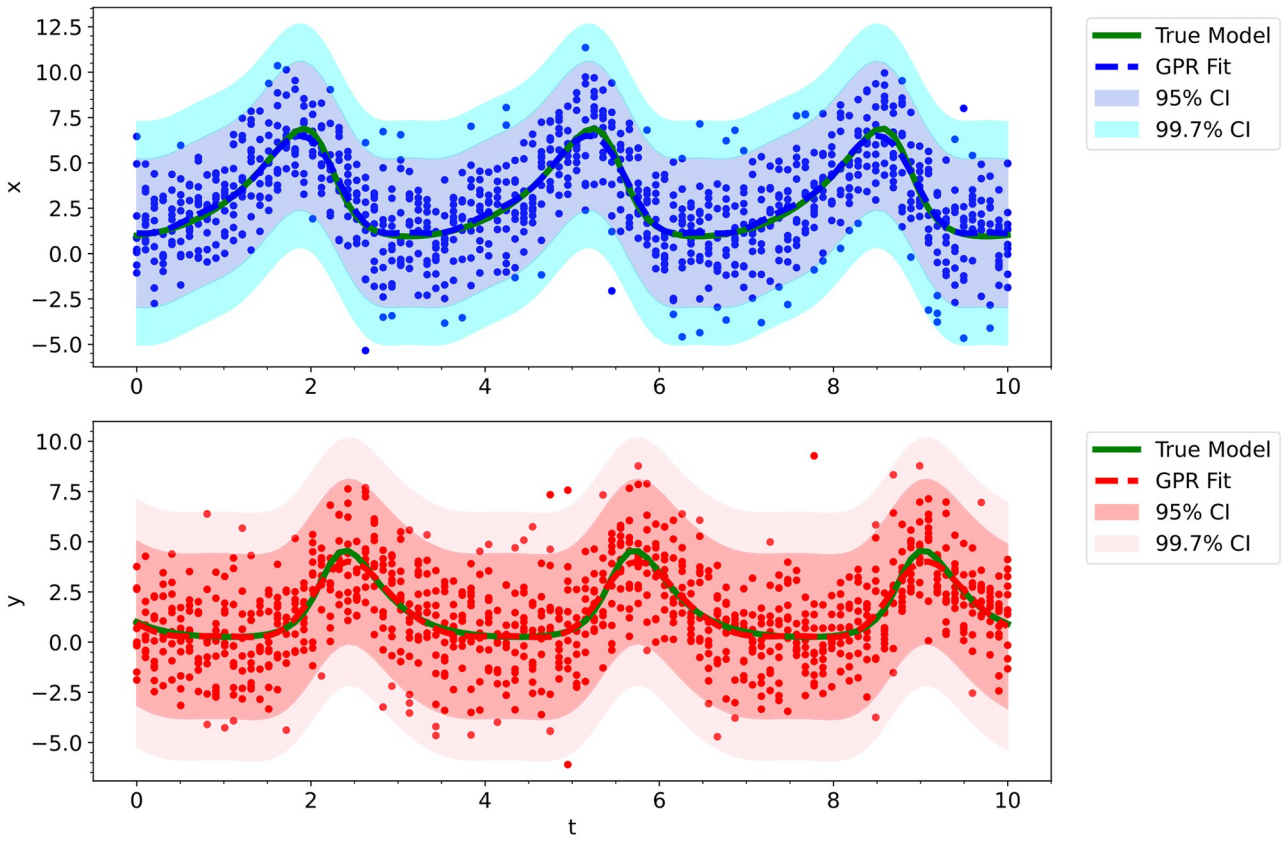

Fig 7. The fitted Gaussian process regression model trained on the noisy Lotka Volterra Oscillator data was used as initial conditions for the neural ODE's integration training trajectories.

https://doi.org/10.1371/journal.pcbi.1012414.g007

In Fig 8, we show the kernel density estimates for the posterior distributions of the ODE's parameters for the Laplace approximation, MCMC, and variational inference with a multivariate Gaussian approximation. For comparison, we also show the inferred parameters obtained by vanilla Sequential Monte Carlo Approximate Bayesian Computation [102] (SMC ABC), a standard method used for inference of parameters in ODEs, for the ODE without any neural networks. We wrote our own JAX based ABC method, but we recommend StochSS [103–105] for those who'd like to use an existing toolkit. SMC ABC had the worst performance for the true parameter values. ABC predicted really wide posterior distributions for some of parameters that were far away from the true parameter values. The Laplace approximation, Markov Chain Monte Carlo, and variational inference predicted more similar posterior distributions for the ODE parameters. As in the case for the univariate cubic polynomial, variational inference predicted very narrow posterior distributions. MCMC resulted in very jagged posterior distributions.

In Fig 9, we show the predictive performance of the inferred parameters. For the parameters obtained from each of the methods, we integrated the ODE out to a final time 5 times that of the training data's time range. The mean predicted model along with 95% and 99.97% credible intervals is shown along with the training data and true ODE model used to generate the training data. MCMC had the worst predictive performance; it predicted the oscillations to dampen over time. Variational inference had reasonable performance with only minor dampening of the oscillations over time, but its predicted posteriors weren't ideal. The Laplace

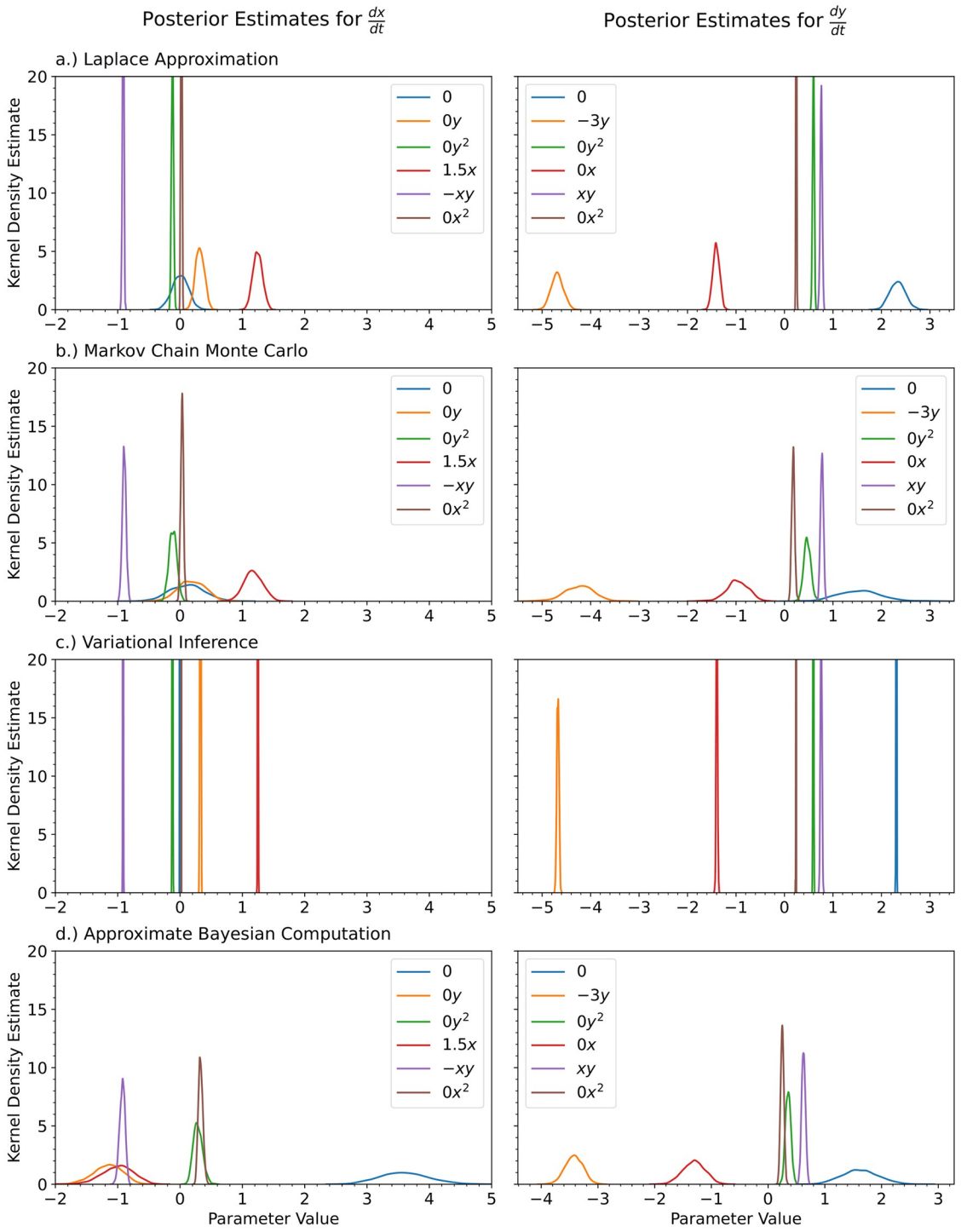

Fig 8. For the Lotka Volterra Oscillator, we show the kernel density estimates for the posterior distributions of the polynomial coefficients obtained with a.) the Laplace Approximation, b.) Markov Chain Monte Carlo, and c.) Variational Inference. For comparison purposes, we also show the case for d.) Approximate Bayesian Computation on a normal ODE. The true value of the coefficients is shown in the legend. The legend is shared for each of the columns.

https://doi.org/10.1371/journal.pcbi.1012414.g008

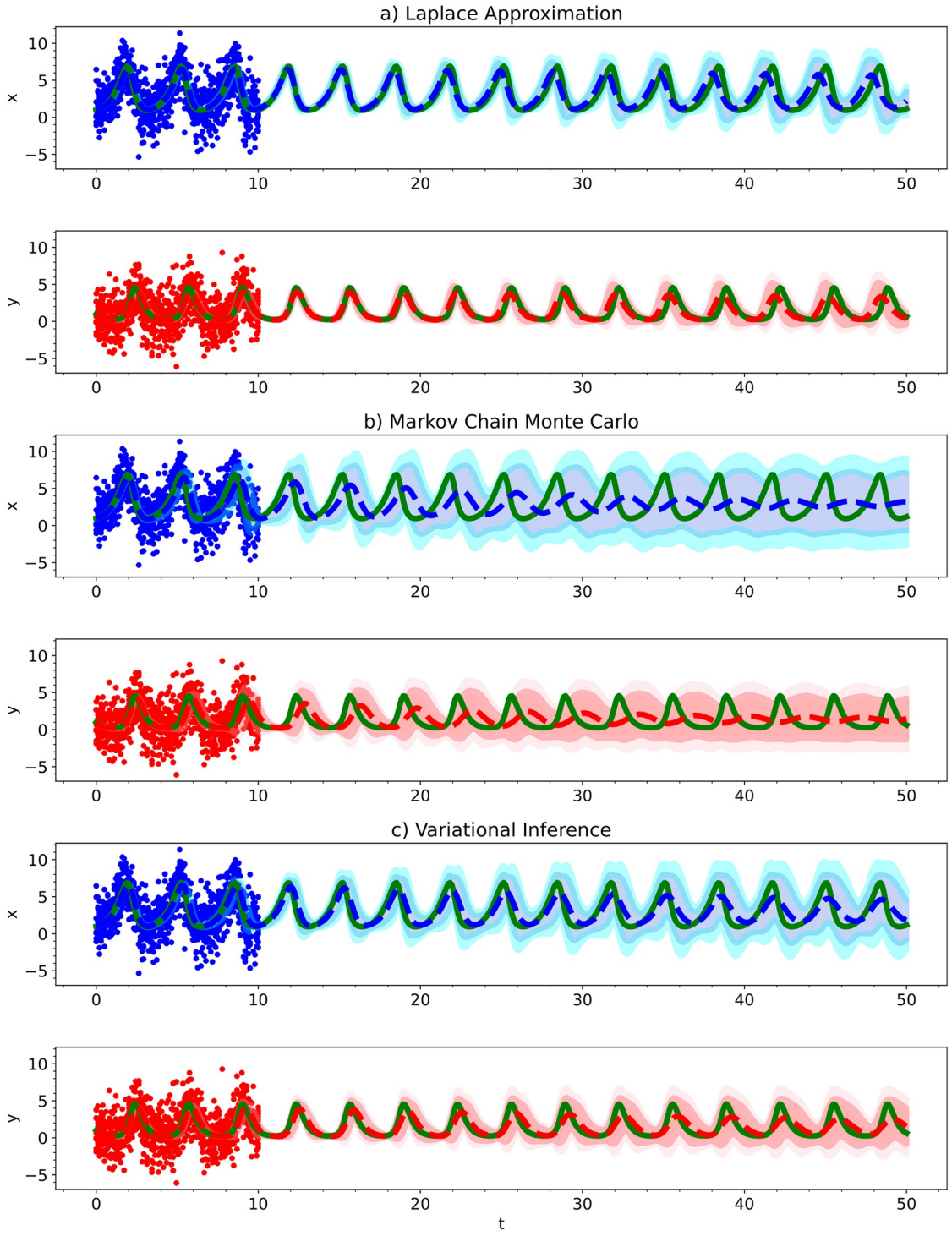

Fig 9. For the Lotka Volterra Oscillator, we show the predictive performance of a Bayesian polynomial neural ODE trained using a) the Laplace Approximation, b) Markov Chain Monte Carlo, and c) Variational Inference. The solid red and blue dots indicate the training data, solid green lines indicate the true ODE model, dashed lines indicate the predictive mean model, and shaded regions indicate 95% and 99.75% credible intervals.

https://doi.org/10.1371/journal.pcbi.1012414.g009

approximation had the best predictive performance with only minor dampening and a minor phase shift, which further highlights its performance and usability since it is also the fastest and easiest method to train.

For Markov Chain Monte Carlo, we verified that our chain converged to our stationary distribution (the posterior) through the construction of a trace plot [67] as well as the calculation of the Geweke diagnostic number [67] for the expanded polynomial coefficients. The trace plot is shown in Fig 10. The trace plot shows good convergence as well as an okay level of mixing. We can quantitatively access the convergence of our Markov Chain by calculating the Geweke diagnostic number, which is given by Eq 23. Table 1 shows the Geweke diagnostic number for each of the expanded polynomial coefficients. At the confidence level of 0.05, the upper and lower tails of the t-distribution are ±1.964. This tells us that the means of the samples consisting of the first 10% and last 50% of the Markov Chain are statistically the same at the confidence level of 0.05, which is a good indicator that the Markov Chain has converged.

Experiment 3: Damped oscillatory system

Our next example is the deterministic damped oscillatory system. This model is a popular toy model for the field of neural ODEs [16, 106]. Damped oscillations appear in many fields of biology, physics, and engineering [107, 108]. One version of the damped oscillator model is given by:

$$\frac{dx}{dt} = -0.1x^3 - 2y^3 \tag{27}$$

$$\frac{dy}{dt} = 2x^3 - 0.1y^3. \tag{28}$$

We generated our training data by integrating an initial value problem with initial conditions given by $(x_0, y_0) = (1, 1)$ over the interval $t \in [0, 25]$ for 500 points uniformly spaced in time. Since the damped oscillator is also a nonstiff ODE system, we integrated the initial value problem with the same numerical methods as was done for the Lotka-Volterra model. We then generated 10 high-noise trajectories originating from the same initial value by adding zero-centered Gaussian noise with a standard deviation of 0.6 to the training data. This corresponds to an instantaneous signal-to-noise ratio [101] ranging from 0 to 2.1. See Fig 11, ignoring the shaded GPR fit, to see the noisy training data.

The architecture from Ref. [9] was used with 660 total parameters. Since we had no prior knowledge of the weights and biases of the polynomial neural network but knew they weren't large values, we used the noninformative Gaussian prior with zero mean and standard deviation of 100000. For MCMC, the warmup was set to 1000 steps and the number of steps taken following warmup was 1000. For the training process, we created batches of trajectories consisting of 13 consecutive data points from the time series. The number of consecutive points to include was determined by trial and error and unfortunately varies from model to model. We simultaneously solve these batch trajectories during each epoch using our own JAX based differentiable ODE solver for the multistep fourth order explicit Runge–Kutta–Fehlberg method [62], which allows us to directly perform backpropagation through the ODE discretization scheme. We have previously discussed why we chose this approach in the methods section of the paper.

Prior to training our neural ODE, we used a smoothing algorithm to generate good initial values for our batch trajectories. One can use any smoothing/filtering algorithm, but we used Gaussian process regression (GPR). For this model, we had the best results with the use of a

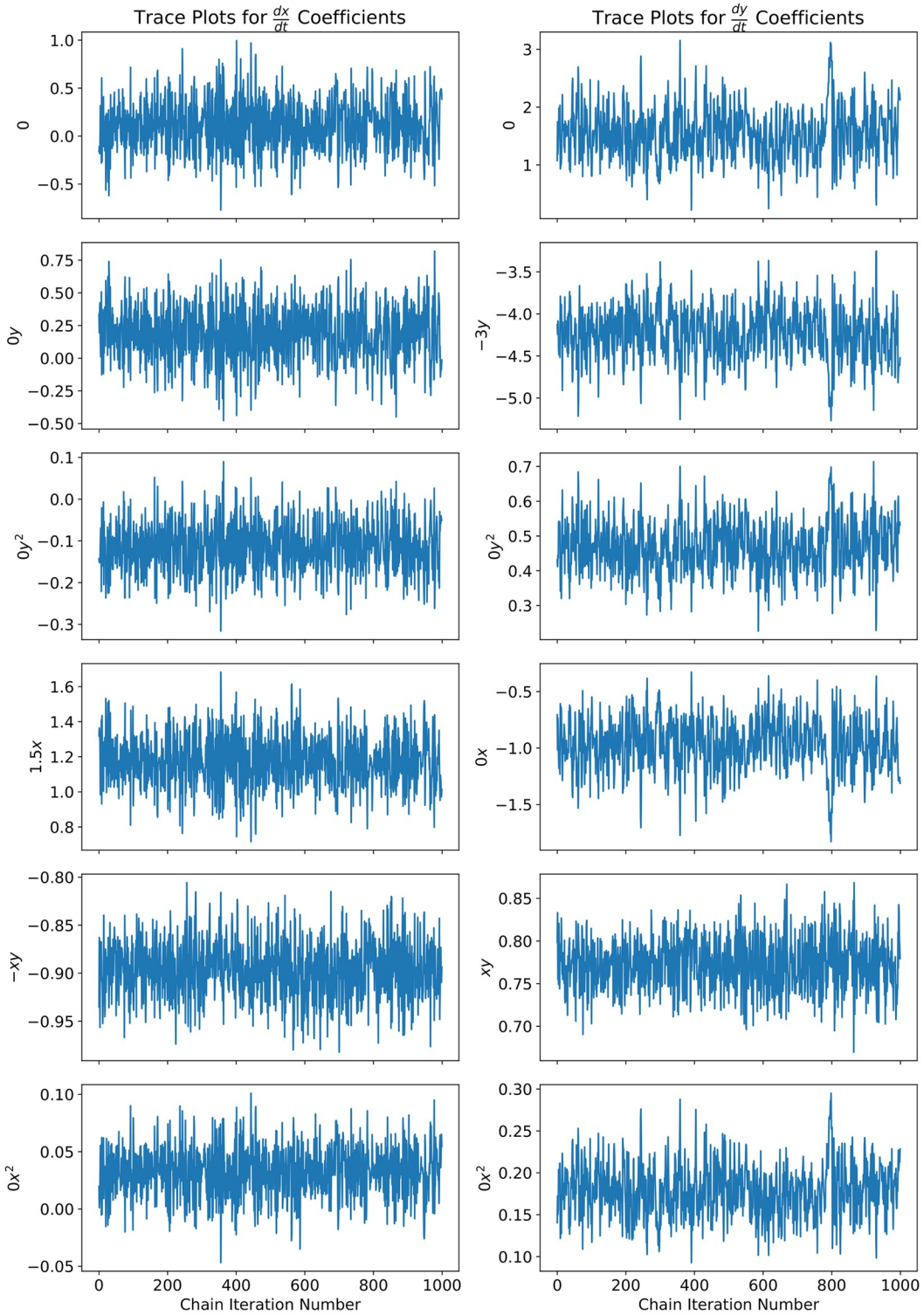

Fig 10. For the Lotka Volterra Oscillator, a second order Bayesian polynomial neural network was trained with Markov Chain Monte Carlo. To access the convergence and mixing of the Markov Chain, we show the trace plot for all of the expanded polynomial coefficients.

https://doi.org/10.1371/journal.pcbi.1012414.g010

Table 1. To access the convergence of Markov Chain Monte Carlo inference on the Lotka Volterra Oscillator, we show the Geweke diagnostic number for each of the parameters. The values were calculated with Eq 23. At the confidence level of 0.05, the upper and lower tails of the t distribution are ±1.964.

$\frac{dx}{dt}$ Terms		$\frac{dy}{dt}$ Terms	
Term	**Geweke Diagnostic**	**Term**	**Geweke Diagnostic**
0	-1.583	0	0.974
$0y$	1.620	$-3y$	-0.644
$0y^2$	-1.368	$0y^2$	0.580
$1.5x$	1.692	$0x$	-0.863
$-xy$	0.340	xy	-0.269
$0x^2$	-1.708	$0x^2$	0.831

https://doi.org/10.1371/journal.pcbi.1012414.t001

rational quadratic kernel [94] scaled by a constant kernel, along with a white kernel (W_k):

$$k(x_i, x_j) = c^2\left(1 + \frac{d(x_i, x_j)^2)}{2\alpha l^2}\right)^{-\alpha} + W_k(\sigma^2_{GPR}). \tag{29}$$

where c^2 is the constant for the constant kernel, d is the euclidean distance function, l is the length-scale, α is the scale mixture parameter, and σ^2_{GPR} is the variance of the Gaussian noise

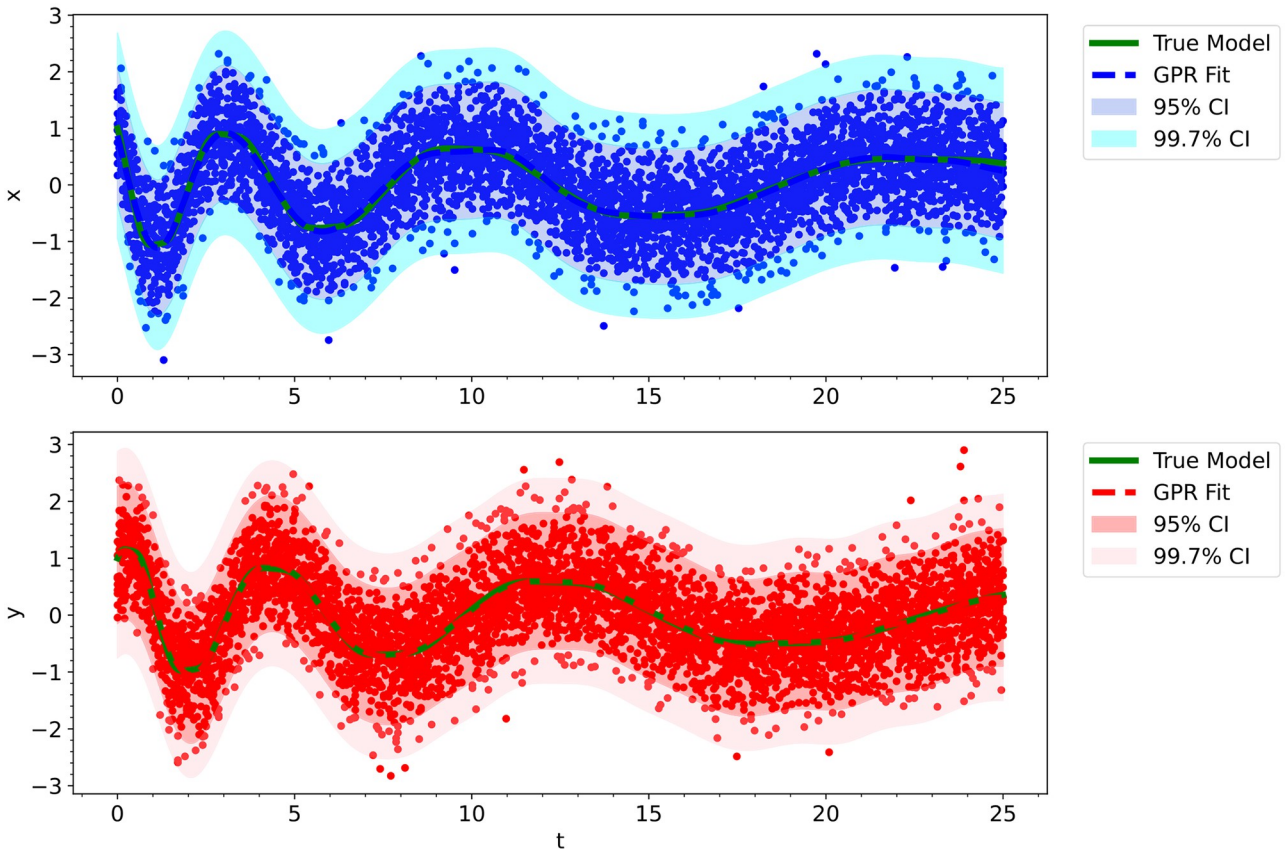

Fig 11. The fitted Gaussian process regression model trained on the noisy Damped Oscillator data was used as initial conditions for the neural ODE's integration training trajectories.

https://doi.org/10.1371/journal.pcbi.1012414.g011

[94]. We used MLE to obtain values for all of these unknown hyperparameters. See Fig 11 for the GPR fit on the observed data. As you can see in the figure, the GPR model fits the noisy data extremely well.

In Fig 12, we show the kernel density estimates for the posterior distributions of the ODE's parameters for a) the Laplace approximation, b) Markov Chain Monte Carlo, and c) variational inference with a multivariate Gaussian approximation. For comparison, we also show the inferred parameters obtained by vanilla Sequential Monte Carlo Approximate Bayesian Computation [102] (SMC ABC), a standard method used for inference of parameters in ODEs, for the ODE without any neural networks.

In Fig 13, we integrated the final Bayesian models out to a final time 5 times that of the training data's time range. The mean predicted model along with 95% and 99.97% credible intervals is shown along with the training data and true ODE model used to generate the training data.

Generally speaking, we observed the same behavior for each of these methods as we did previously for the Lotka Volterra model. Approximate Bayesian computation had the widest posterior distributions. Variational inference had the narrowest posterior distributions and the credible intervals in the trajectory prediction were too narrow to capture the true trajectory—the method is too confident about the inferred parameters. This time, Markov Chain Monte Carlo (MCMC) completely failed to learn an accurate enough model to predict the trajectory of the system beyond $t = 2$. We spent a large amount of time playing around with the best settings for MCMC for this model, but the method failed every time. The other methods did not require nearly as much time to get working results for. Given enough patience, MCMC will result in somewhat accurate results but the other methods are much easier to use. For this reason, we do not recommend using MCMC for neural ODEs. The Laplace approximation provided the most accurate parameter estimates as well as predictions for the trajectories of the system. It is also the fastest and easiest method to use. For these reasons, we recommend using the Laplace approximation over the other methods.

Experiment 4: Lorenz attractor

The Lorenz attractor [109] is an example of a deterministic chaotic system [110, 111] that came from a simplified model for atmospheric convection [112]:

$$\frac{dx}{dt} = \sigma(y - x), \tag{30}$$

$$\frac{dy}{dt} = x(r - z) - y, \tag{31}$$

$$\frac{dz}{dt} = xy - bz. \tag{32}$$

The equations describe the two-dimensional flow of a fluid with uniform depth between an upper and lower surface, given a temperature gradient. In the equations, x is proportional to the intensity of convective motion, y is proportional to the difference in temperature between the rising and falling currents of fluid, and z is proportional to the amount of non-linearity within the vertical temperature profile [109, 112]. σ is the Prandtl number, r is the Rayleigh number, and b is a geometric factor [109, 112]. Typically, $\sigma = 10$, $r = 28$, and $b = \frac{8}{3}$. For our example, we use these values for the parameters. Since the discovery of the Lorenz model, it has also been used as a simplified model for various other systems such as: chemical reactions

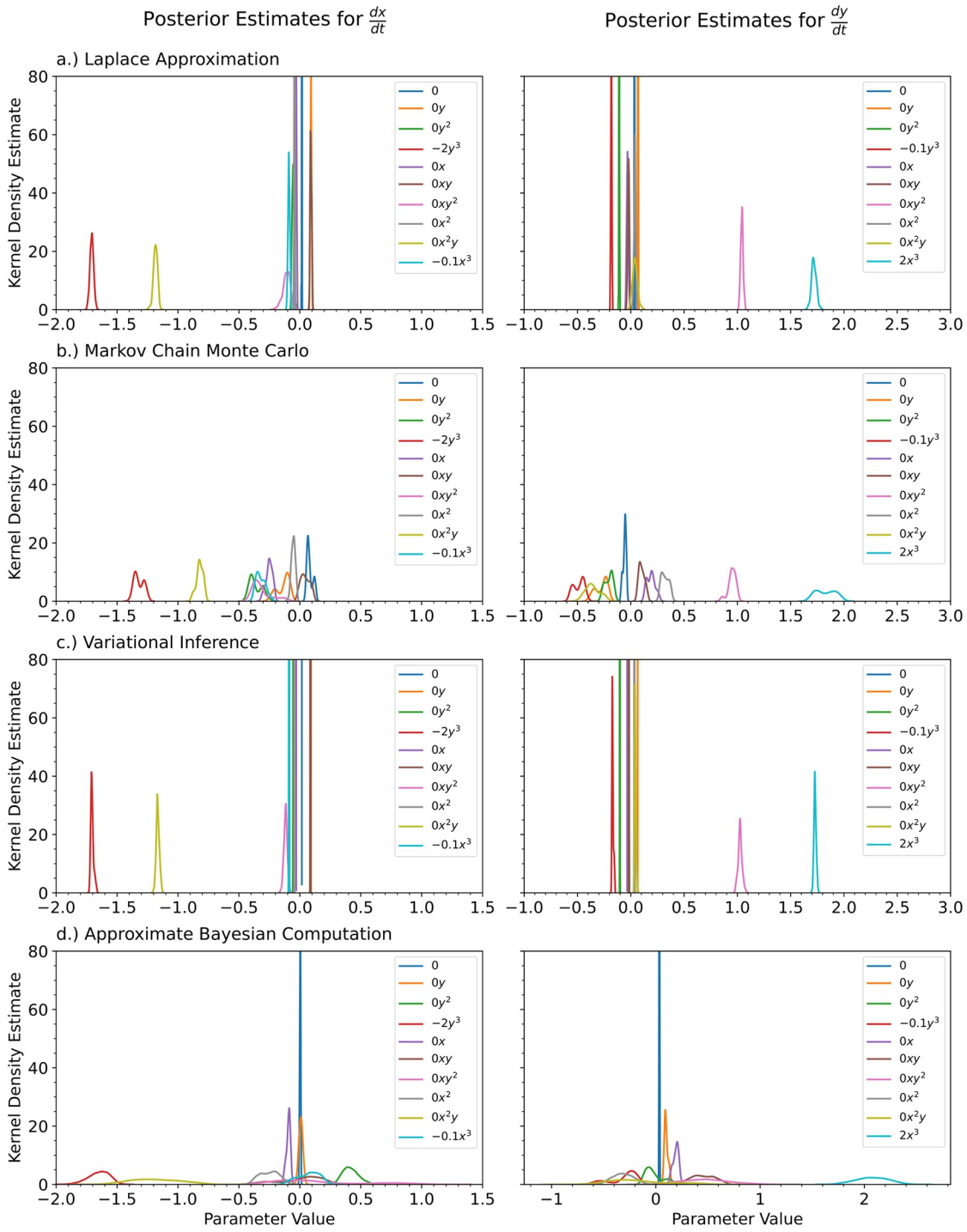

Fig 12. For the Damped Oscillator, we show the kernel density estimates for the posterior distributions of the polynomial coefficients obtained with a.) the Laplace Approximation, b.) Markov Chain Monte Carlo, and c.) Variational Inference. For comparison purposes, we also show the case for d.) Approximate Bayesian Computation on a normal ODE. The true value of the coefficients is shown in the legend. The legend is shared for each of the columns.

https://doi.org/10.1371/journal.pcbi.1012414.g012

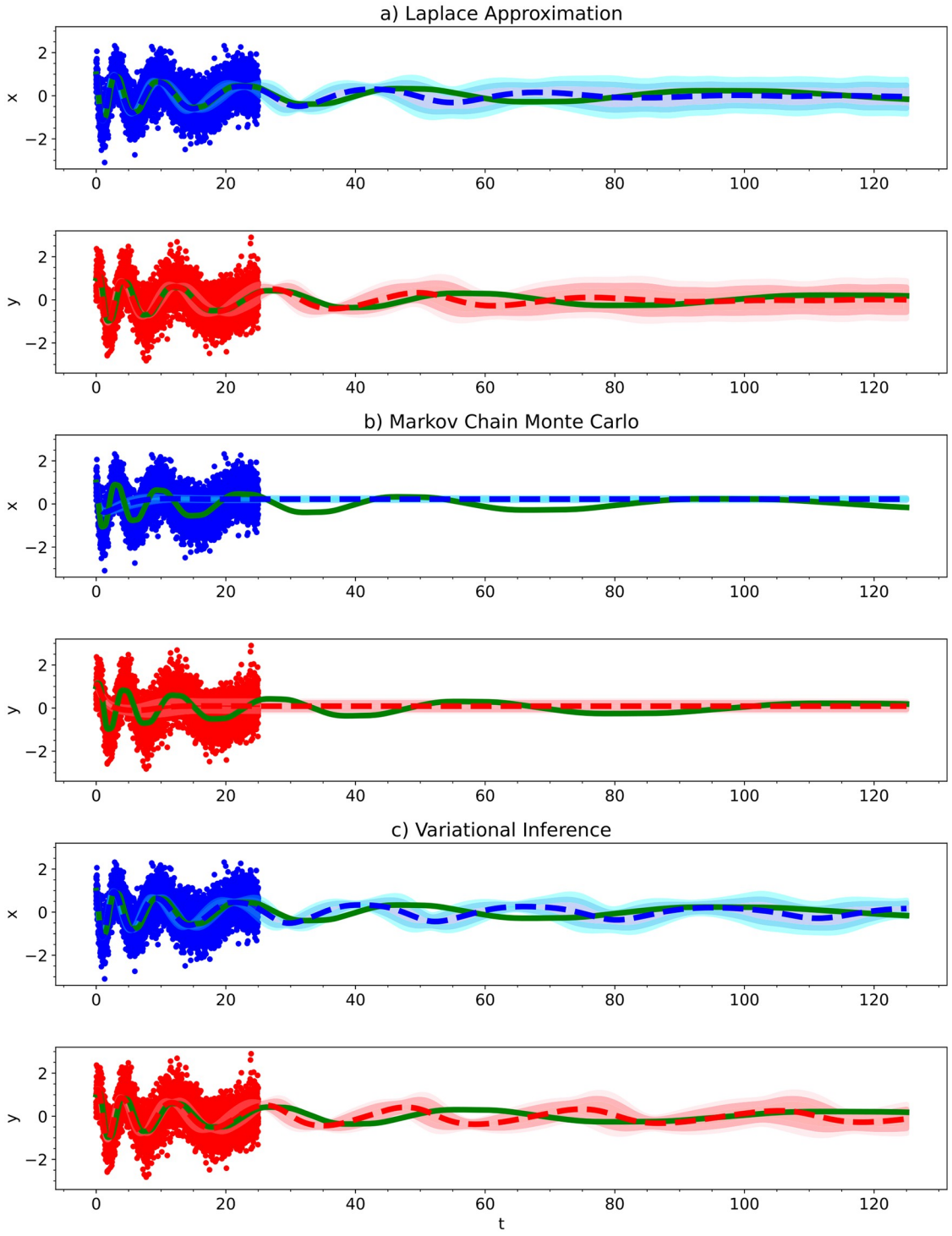

Fig 13. For the Damped Oscillator, we show the predictive performance of a Bayesian polynomial neural ODE trained using a) the Laplace Approximation, b) Markov Chain Monte Carlo, and c) Variational Inference. The solid red and blue dots indicate the training data, solid green lines indicate the true ODE model, dashed lines indicate the predictive mean model, and shaded regions indicate 95% and 99.75% credible intervals.

https://doi.org/10.1371/journal.pcbi.1012414.g013

[113], lasers [114], electric circuits [115], brushless DC motors [116], thermosyphons [117], and dynamos [117].

Due to the chaotic nature of this system and the high frequency of oscillations, we required more training data for this example than for the previous examples shown. We generated our training data from initial conditions $(x_0, y_0, z_0) = (1, 1, 1)$ over time interval $t \in [0, 30]$ for 900 points uniformly spaced in time. We then generated 10 high-noise trajectories originating from the same initial value by adding zero-centered Gaussian noise with a standard deviation of 2 to the training data. The architecture from Ref. [9] was used with 231 total parameters. Since we had no prior knowledge of the weights and biases of the polynomial neural network but knew they weren't large values, we used the noninformative Gaussian prior with zero mean and standard deviation of 100000. For MCMC, the warmup was set to 1000 steps and the number of steps taken following warmup was 1000. The training process was exactly the same as the previous two examples. This time, we batched our data into training trajectories consisting of two adjacent data points. This number was found through trial and error, but we hypothesize that the trajectory length needs to be shorter for this example due to the high frequency oscillations. For this example, we used the same GPR kernel as was used for the damped oscillator (see Eq 29).

Fig 14 shows the kernel density estimates for the various Bayesian inference methods. Since the level of noise is smaller for this example, the posterior estimates are narrower. There is also

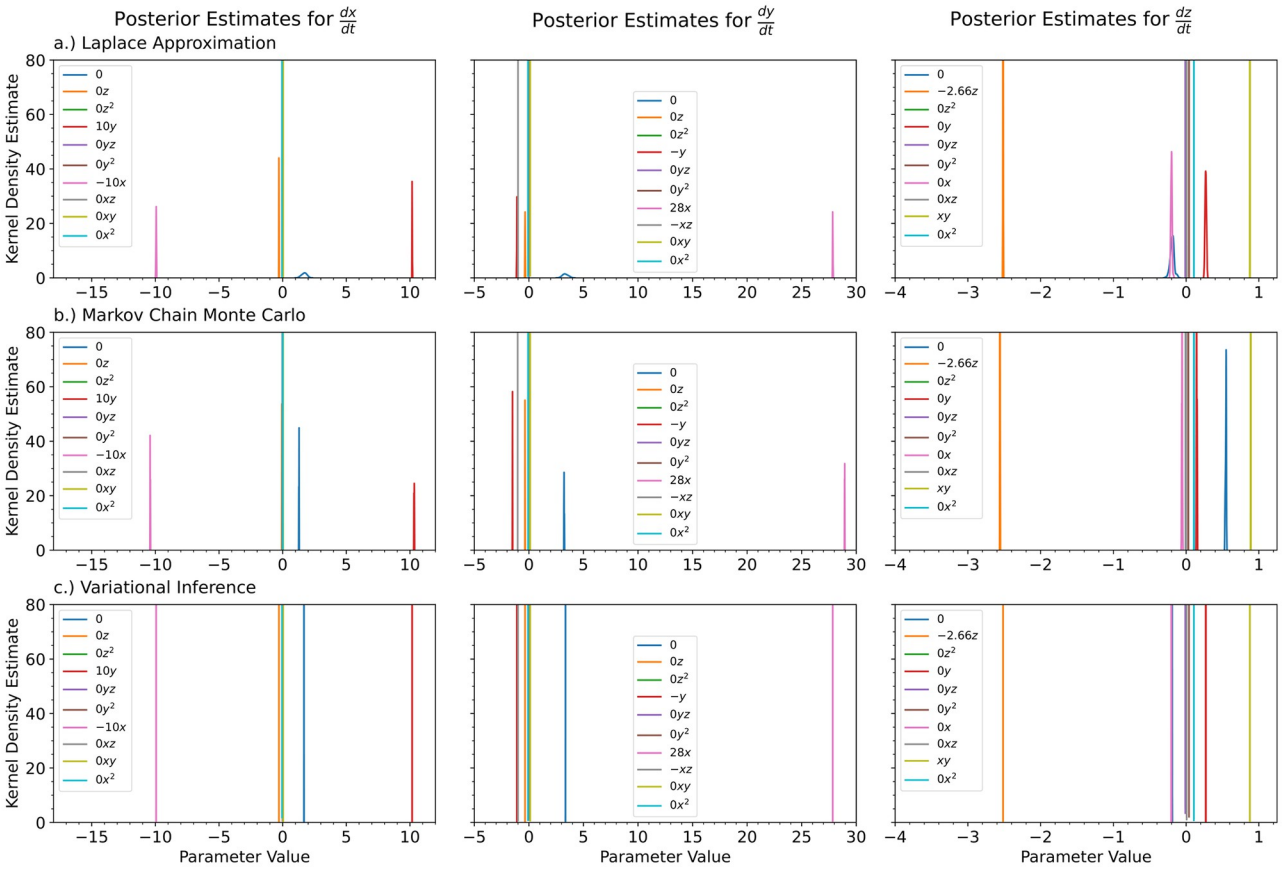

Fig 14. For the Lorenz Attractor, we show the kernel density estimates for the posterior distributions of the polynomial coefficients obtained with a.) the Laplace Approximation, b.) Markov Chain Monte Carlo, and c.) Variational Inference. The true value of the coefficients is shown in the legend. The legend is shared for each of the columns.

https://doi.org/10.1371/journal.pcbi.1012414.g014

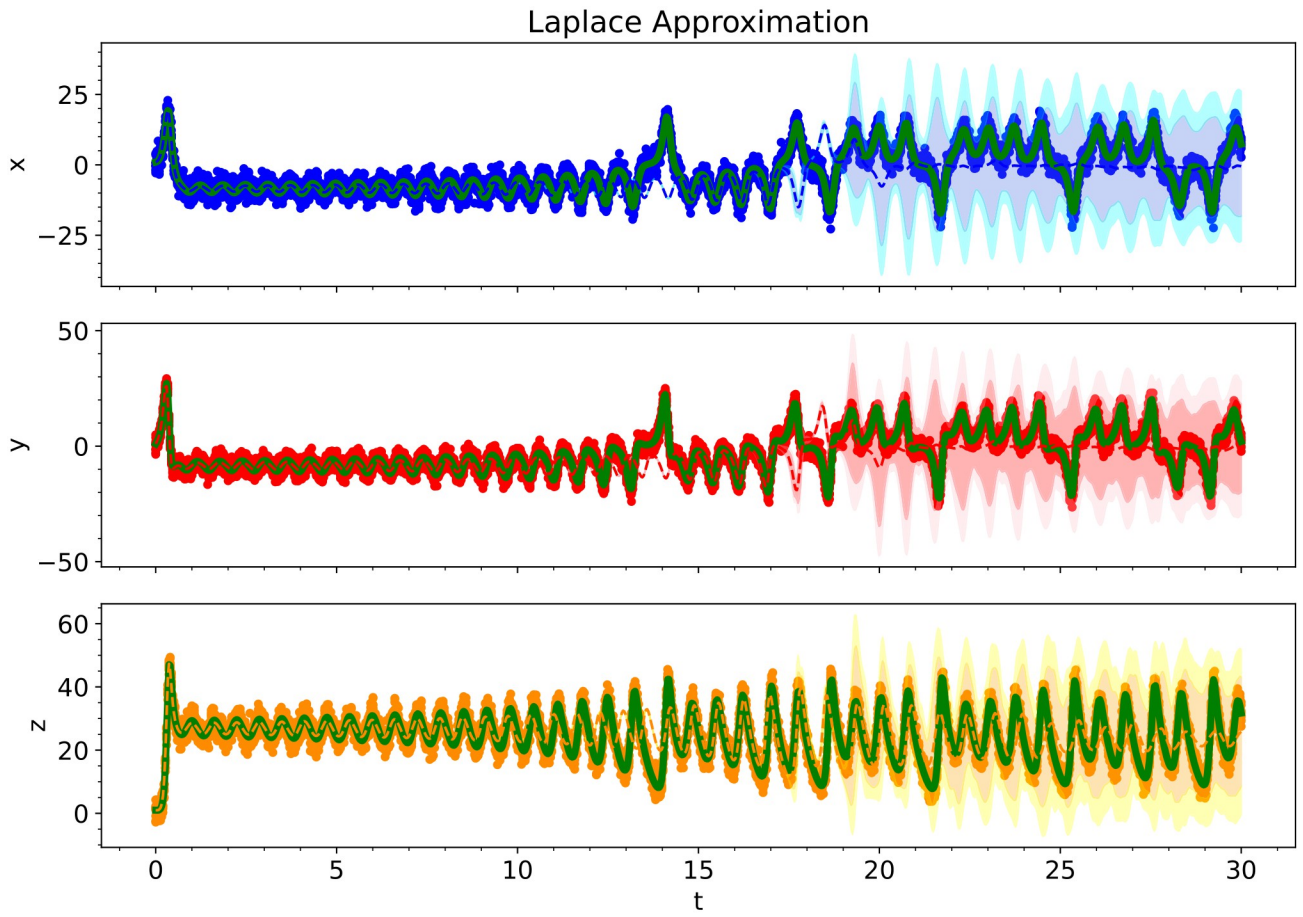

Fig 15. For the Lorenz Attractor, we show the predictive performance of a Bayesian polynomial neural ODE trained using the Laplace Approximation. The solid red, blue, and orange dots indicate the training data, solid green lines indicate the true ODE model, dashed lines indicate the predictive mean model, and shaded regions indicate 95% and 99.75% credible intervals.

https://doi.org/10.1371/journal.pcbi.1012414.g015

very little difference between the predicted Bayesian uncertainties. Figs 15, 16 and 17 show the trajectory predictions for the Laplace approximation, Markov Chain Monte Carlo, and variational inference respectively. All of the methods' 95% credible intervals were able to capture the true trajectory. In terms of accuracy, there is no clear winner for the Lorenz attractor. However, in terms of speed, the Laplace approximation is the best choice.

Example 5: Learning missing terms from a partially known ODE model

It is common for scientists to have an incomplete model of their system—one in which they are confident about certain processes undergoing the system, but there are mechanisms they aren't aware of. Rather than learn the whole system from scratch, we can incorporate the known parts of our model into the neural ODE and have the neural ODE suggest additional components of the ODE model given the observed data. Incorporating the known ODE model into the neural ODE framework is done simply by adding the output of the known equation to that of the neural ODE output—no special treatment is required aside from that (see Eq 4).

We will use the Lotka Volterra Oscillator to demonstrate the ability of polynomial neural ODEs to learn missing terms from a partially known ODE model. We also show that Bayesian

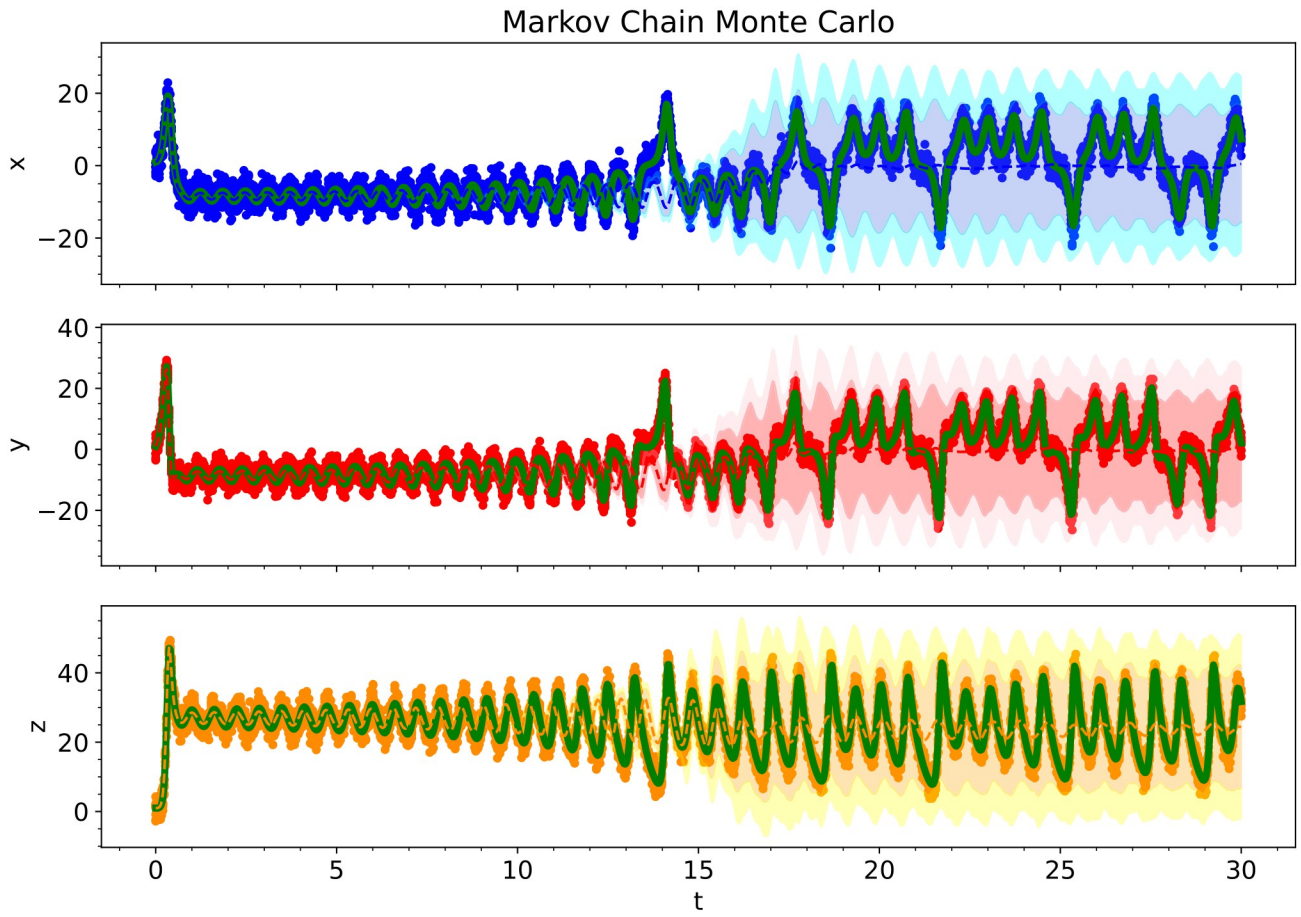

Fig 16. For the Lorenz Attractor, we show the predictive performance of a Bayesian polynomial neural ODE trained using Markov Chain Monte Carlo. The solid red, blue, and orange dots indicate the training data, solid green lines indicate the true ODE model, dashed lines indicate the predictive mean model, and shaded regions indicate 95% and 99.75% credible intervals.

https://doi.org/10.1371/journal.pcbi.1012414.g016

uncertainties can be obtained for the parameters in the terms that the neural ODE suggests including. As a reminder, the Lotka Volterra model is given by:

$$\frac{dx}{dt} = \mathbf{1.5}\,\mathbf{x} - xy, \tag{33}$$

$$\frac{dy}{dt} = -3y + \mathbf{x}\,\mathbf{y}. \tag{34}$$

For this experiment, the bold terms are the ones we do not know. The goal will be to recover these terms along with posterior distributions for the values of the parameters. We used the same training data, GPR model for the initial conditions, and training process as was previously used in the Lotka Volterra example. The only difference was including the known ODE model (see Eq 4).

Fig 18 shows the posterior distributions recovered for all of the candidate terms to include in the final ODE model. The neural ODE was able to identify the missing terms with few false terms. Most of the terms that are not in the true model are predicted to be close to zero. As was seen in the previous examples, variational inference provides very narrow posterior

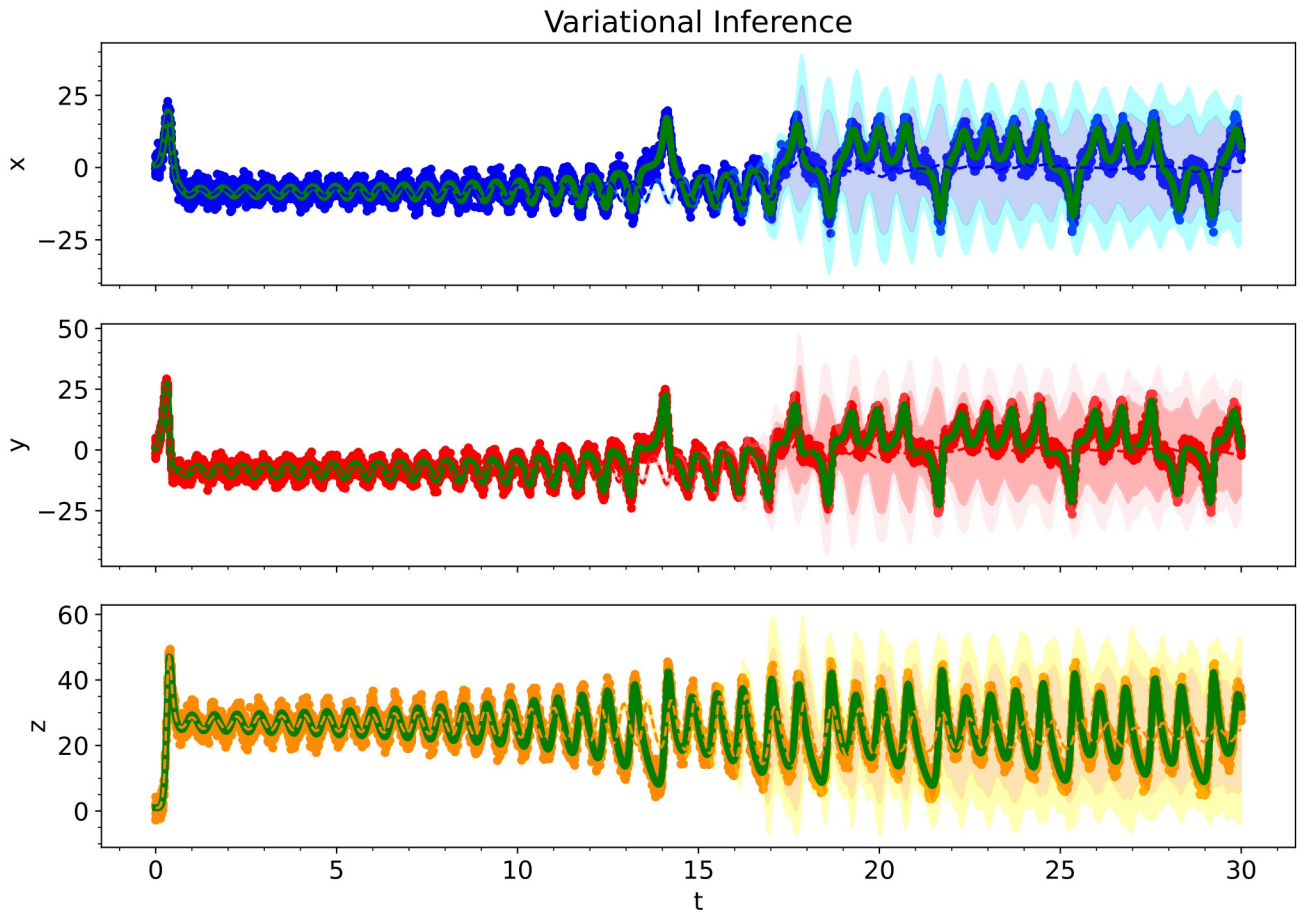

Fig 17. For the Lorenz Attractor, we show the predictive performance of a Bayesian polynomial neural ODE trained using variational inference. The solid red, blue, and orange dots indicate the training data, solid green lines indicate the true ODE model, dashed lines indicate the predictive mean model, and shaded regions indicate 95% and 99.75% credible intervals.

https://doi.org/10.1371/journal.pcbi.1012414.g017

distributions and MCMC provides results between the Laplace approximation and variational inference.

Discussion

This work addressed the problem of how to handle noisy data and recover uncertainty estimates for: (1) symbolic regression with deep polynomial neural networks and (2) polynomial neural ODEs. More broadly, we also helped to answer the question of how to handle noisy data and perform Bayesian inference on the general class of symbolic neural networks and symbolic neural ODEs.

We compared the following Bayesian inference methods: (a) the Laplace approximation, (b) Markov Chain Monte Carlo (MCMC) sampling methods, and (c) variational inference. We do not recommend using Markov Chain Monte Carlo for neural ODEs. Using MCMC for neural ODEs requires a substantial amount of patience, it is the most computationally expensive method, and we showed that the results are not encouraging. A substantial amount of development work needs to be devoted towards addressing the challenges of using MCMC for neural ODEs in an effective manner. Variational inference is also challenging to use—some

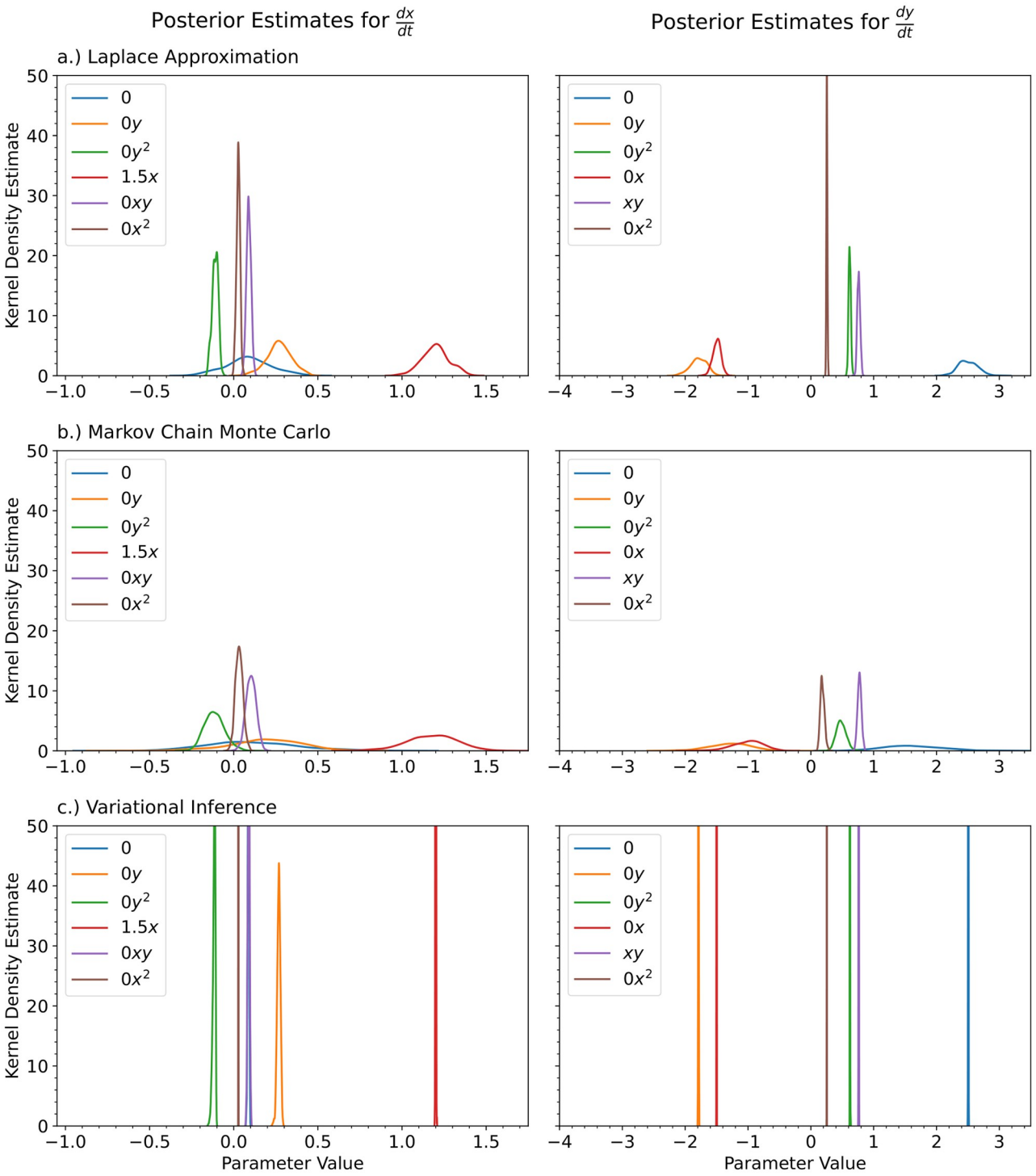

Fig 18. It is common for a domain expert to understand part of the system's underlying mechanisms, but have in incomplete model. Given an incomplete model, a neural ODE can learn the missing terms from the ODE model that best fit the observed data. We have removed two of the terms from the Lotka Volterra model and tested the neural ODE's ability to learn the missing terms. We show the kernel density estimates for the posterior distributions of the polynomial coefficients obtained with a.) the Laplace Approximation, b.) Markov Chain Monte Carlo, and c.) Variational Inference. The true value of the coefficients is shown in the legend. The legend is shared for each of the columns.

https://doi.org/10.1371/journal.pcbi.1012414.g018

time is spent deciding the mean and covariance matrix to use for initialization of the parameters. This process can be sped up by first obtaining point estimates for the parameters and using the values obtained to initialize the mean matrix. Using this approach made variational inference a viable option to implement. However, the posterior estimates generated by variational inference's posterior are consistently too narrow: it is too confident about its estimates.

The Laplace approximation is the easiest to implement and the fastest method. The main challenge associated with the Laplace approximation for neural networks is inverting the Fisher information matrix; however, most of the models in this class of problems are small enough that this is not an issue. Based on our experience, we recommend having no more than 50,000 parameters if you plan on using the Laplace approximation for a neural network and want to use the exact or pseudo inverse of the Fisher information matrix. We were initially skeptical about the Laplace approximation because it makes a Gaussian approximation for all of the parameters. However, we have shown that this approximation is not problematic when the polynomials are multiplied out. We have shown that the Laplace approximation has high accuracy. For these reasons, we recommend using the Laplace approximation for this class of problems.

It is important to point out that our paper focuses exclusively on additive noise models due to its clarity and analytical tractability. While understandable for initial exploration, it's important to point out that future work will need to address multiplicative noise. For example, biological systems and financial markets often experience noise that scales with the signal's magnitude, presenting a different set of challenges for Bayesian algorithm development.

Although our approach of using the Laplace approximation around the local minima for polynomial neural ODE models is understandable and supported by encouraging empirical evidence, particularly in scenarios where the unimodality of the posterior is apparent, it is essential to consider and acknowledge potential limitations in more complex scenarios. For instance, datasets consisting of time-series data from multiple sources or experiments, each with distinct dynamics or patterns, may present challenges. In such cases, the posterior distribution of the model parameters (weights and biases) could exhibit multimodality, with different modes corresponding to different subsets or regimes of the data. By approximating the posterior with a single Gaussian distribution centered around the MAP estimate, the Laplace approximation might overlook the multimodal nature of the posterior and the existence of multiple significant modes. Therefore, while our experiments provide preliminary evidence of the success of the Laplace approximation, there is substantial future work to be done by statisticians and researchers to explore these nuances and caveats more thoroughly. This additional research would be invaluable for a deeper understanding and more robust application of the Laplace approximation in neural networks.

Supporting information

S1 Text. Supplementary figures and tables.
(PDF)

Acknowledgments

This work has benefited from our participation in Dagstuhl Seminar 22332 "Differential Equations and Continuous-Time Deep Learning [25]". Many thanks to the organizers of this seminar: David Duvenaud (University of Toronto, CA), Markus Heinonen (Aalto University, FI), Michael Tiemann (Robert Bosch GmbH—Renningen, DE), and Max Welling (University of Amsterdam, NL). This work has also benefited from our participation in the University of

Bonn's Hausdorff School: "Inverse problems for multi-scale models." Many thanks to the organizers of this summer school: Lorenzo Contento, Jan Hasenauer, and Yannik Schälte. We'd like to thank Alexander Franks (UC Santa Barbara) for giving us the idea of using Monte Carlo for obtaining posterior distributions for the polynomial coefficients. We'd also like to thank Michael Tiemann and Katharina Ott (Robert Bosch GmbH—Renningen, DE) for recommending that we try the Laplace approximation on the polynomial neural ODEs. Lastly, we'd like to thank Patrick Kidger (Google) for recommending that we switch from PyTorch to JAX and providing resources for making the switch, as JAX has allowed us to do so much more.

Use was made of computational facilities purchased with funds from the National Science Foundation (CNS-1725797) and administered by the Center for Scientific Computing (CSC). The CSC is supported by the California NanoSystems Institute and the Materials Research Science and Engineering Center (MRSEC; NSF DMR 2308708) at UC Santa Barbara.

This work was supported in part by National Science Foundation (NSF) awards CNS-1730158, ACI-1540112, ACI-1541349, OAC-1826967, OAC-2112167, CNS-2100237, CNS-2120019, the University of California Office of the President, and the University of California San Diego's California Institute for Telecommunications and Information Technology/Qualcomm Institute. Thanks to CENIC for the 100Gbps networks. The content of the information does not necessarily reflect the position or the policy of the funding agencies, and no official endorsement should be inferred. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Author Contributions

Conceptualization: Colby Fronk, Linda Petzold.

Formal analysis: Colby Fronk.

Funding acquisition: Prashant Singh, Linda Petzold.

Investigation: Colby Fronk, Jaewoong Yun, Linda Petzold.

Methodology: Colby Fronk, Prashant Singh, Linda Petzold.

Project administration: Prashant Singh, Linda Petzold.

Resources: Colby Fronk, Linda Petzold.

Software: Colby Fronk.

Supervision: Prashant Singh, Linda Petzold.

Validation: Colby Fronk, Jaewoong Yun, Prashant Singh.

Visualization: Colby Fronk.

Writing – original draft: Colby Fronk.

Writing – review & editing: Colby Fronk, Jaewoong Yun, Prashant Singh, Linda Petzold.

References

1. Brunton SL, Proctor JL, Kutz JN. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. Proceedings of the National Academy of Sciences. 2016; 113 (15):3932–3937. https://doi.org/10.1073/pnas.1517384113 PMID: 27035946
2. Hirsh S, Barajas-Solano D, Kutz J. Sparsifying priors for Bayesian uncertainty quantification in model discovery. Royal Society open science. 2022; 9:211823. https://doi.org/10.1098/rsos.211823 PMID: 35223066

3. Kaheman K, Kutz JN, Brunton SL. SINDy-PI: a robust algorithm for parallel implicit sparse identification of nonlinear dynamics. Proceedings Mathematical, Physical, and Engineering Sciences. 2020; 476. https://doi.org/10.1098/rspa.2020.0279 PMID: 33214760
4. Rudy S, Brunton S, Proctor J, Kutz J. Data-driven discovery of partial differential equations. Science Advances. 2016; 3.
5. Alves EP, Fiuza F. Robust data-driven discovery of reduced plasma physics models from fully kinetic simulations. In: APS Division of Plasma Physics Meeting Abstracts. vol. 2020 of APS Meeting Abstracts; 2020. p. GO10.006.
6. Mangan NM, Brunton SL, Proctor JL, Kutz JN. Inferring Biological Networks by Sparse Identification of Nonlinear Dynamics. IEEE Transactions on Molecular, Biological and Multi-Scale Communications. 2016; 2(1):52–63. https://doi.org/10.1109/TMBMC.2016.2633265
7. Hoffmann M, Fröhner C, Noé F. Reactive SINDy: Discovering governing reactions from concentration data. The Journal of Chemical Physics. 2019; 150(2):025101. https://doi.org/10.1063/1.5066099 PMID: 30646700
8. Sorokina M, Sygletos S, Turitsyn S. Sparse identification for nonlinear optical communication systems: SINO method. Opt Express. 2016; 24(26):30433–30443. https://doi.org/10.1364/OE.24.030433 PMID: 28059391
9. Fronk C, Petzold L. Interpretable polynomial neural ordinary differential equations. Chaos: An Interdisciplinary Journal of Nonlinear Science. 2023; 33(4):043101. https://doi.org/10.1063/5.0130803 PMID: 37097945
10. Li S, Xu LD, Zhao S. The internet of things: a survey. Information systems frontiers. 2015; 17:243–259. https://doi.org/10.1007/s10796-014-9492-7
11. Rose K, Eldridge S, Chapin L. The internet of things: An overview. The internet society (ISOC). 2015; 80:1–50.
12. Mayr LM, Bojanic D. Novel trends in high-throughput screening. Current opinion in pharmacology. 2009; 9(5):580–588. https://doi.org/10.1016/j.coph.2009.08.004 PMID: 19775937
13. Szymański P, Markowicz M, Mikiciuk-Olasik E. Adaptation of high-throughput screening in drug discovery—toxicological screening tests. International journal of molecular sciences. 2011; 13(1):427–452. https://doi.org/10.3390/ijms13010427 PMID: 22312262
14. Balsamo G, Agusti-Panareda A, Albergel C, Arduini G, Beljaars A, Bidlot J, et al. Satellite and In Situ Observations for Advancing Global Earth Surface Modelling: A Review. Remote Sensing. 2018; 10:2038. https://doi.org/10.3390/rs10122038
15. Goodfellow I, Bengio Y, Courville A. Deep Learning. MIT Press; 2016.
16. Chen RT, Rubanova Y, Bettencourt J, Duvenaud DK. Neural ordinary differential equations. Advances in neural information processing systems. 2018; 31. https://doi.org/10.1007/978-3-030-04221-9
17. Rubanova Y, Chen RT, Duvenaud DK. Latent ordinary differential equations for irregularly-sampled time series. Advances in neural information processing systems. 2019; 32.
18. Dandekar R, Dixit V, Tarek M, Garcia-Valadez A, Rackauckas C. Bayesian Neural Ordinary Differential Equations. CoRR. 2020;abs/2012.07244.
19. Li X, Wong TKL, Chen RTQ, Duvenaud D. Scalable Gradients for Stochastic Differential Equations. In: Chiappa S, Calandra R, editors. Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics. vol. 108 of Proceedings of Machine Learning Research. PMLR; 2020. p. 3870–3882.
20. Kidger P, Morrill J, Foster J, Lyons T. Neural controlled differential equations for irregular time series. Advances in Neural Information Processing Systems. 2020; 33:6696–6707.
21. Kidger P. On neural differential equations. arXiv preprint arXiv:220202435. 2022;.
22. Morrill J, Salvi C, Kidger P, Foster J. Neural rough differential equations for long time series. In: International Conference on Machine Learning. PMLR; 2021. p. 7829–7838.
23. Jia J, Benson AR. Neural jump stochastic differential equations. Advances in Neural Information Processing Systems. 2019; 32.
24. Chen RT, Amos B, Nickel M. Learning neural event functions for ordinary differential equations. arXiv preprint arXiv:201103902. 2020;.
25. Duvenaud D, Heinonen M, Tiemann M, Welling M. Differential Equations and Continuous-Time Deep Learning. Visualization and Decision Making Design Under Uncertainty. 2023; p. 19.
26. Owhadi H. Bayesian numerical homogenization. Multiscale Modeling & Simulation. 2015; 13(3):812–828. https://doi.org/10.1137/140974596
27. Raissi M, Karniadakis G. Hidden Physics Models: Machine Learning of Nonlinear Partial Differential Equations. Journal of Computational Physics. 2017; 357.

28. Raissi M, Perdikaris P, Karniadakis GE. Numerical Gaussian processes for time-dependent and nonlinear partial differential equations. SIAM Journal on Scientific Computing. 2018; 40(1):A172–A198. https://doi.org/10.1137/17M1120762

29. Raissi M, Perdikaris P, Karniadakis GE. Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations; 2017.

30. Raissi M, Perdikaris P, Karniadakis GE. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. Journal of Computational physics. 2019; 378:686–707. https://doi.org/10.1016/j.jcp.2018.10.045

31. Cuomo S, Di Cola VS, Giampaolo F, Rozza G, Raissi M, Piccialli F. Scientific machine learning through physics–informed neural networks: Where we are and what's next. Journal of Scientific Computing. 2022; 92(3):88. https://doi.org/10.1007/s10915-022-01939-z

32. Cai S, Mao Z, Wang Z, Yin M, Karniadakis GE. Physics-informed neural networks (PINNs) for fluid mechanics: A review. Acta Mechanica Sinica. 2021; 37(12):1727–1738. https://doi.org/10.1007/s10409-021-01148-1

33. Chrysos GG, Moschoglou S, Bouritsas G, Deng J, Panagakis Y, Zafeiriou S. Deep Polynomial Neural Networks. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2022; 44(8):4021–4034. PMID: 33571091

34. Kim S, Lu P, Mukherjee S, Gilbert M, Jing L, Ceperic V, et al. Integration of Neural Network-Based Symbolic Regression in Deep Learning for Scientific Discovery. IEEE Transactions on Neural Networks and Learning Systems. 2020;PP:1–12.

35. Kubalík J, Derner E, Babuška R. Toward Physically Plausible Data-Driven Models: A Novel Neural Network Approach to Symbolic Regression. IEEE Access. 2023; 11:61481–61501. https://doi.org/10.1109/ACCESS.2023.3287397

36. Zhang M, Kim S, Lu PY, Soljačić M. Deep Learning and Symbolic Regression for Discovering Parametric Equations; 2023.

37. Abdellaoui IA, Mehrkanoon S. Symbolic regression for scientific discovery: an application to wind speed forecasting. In: 2021 IEEE Symposium Series on Computational Intelligence (SSCI); 2021. p. 01–08.

38. Su X, Ji W, An J, Ren Z, Deng S, Law CK. Kinetics Parameter Optimization via Neural Ordinary Differential Equations; 2022.

39. Ji W, Deng S. Autonomous Discovery of Unknown Reaction Pathways from Data by Chemical Reaction Neural Network. The Journal of Physical Chemistry A. 2021; 125(4):1082–1092. https://doi.org/10.1021/acs.jpca.0c09316 PMID: 33471526

40. Boddupalli N, Matchen T, Moehlis J. Symbolic regression via neural networks. Chaos: An Interdisciplinary Journal of Nonlinear Science. 2023; 33(8):083150. https://doi.org/10.1063/5.0134464 PMID: 38060788

41. Jospin LV, Laga H, Boussaid F, Buntine W, Bennamoun M. Hands-On Bayesian Neural Networks—A Tutorial for Deep Learning Users. IEEE Computational Intelligence Magazine. 2022; 17(2):29–48. https://doi.org/10.1109/MCI.2022.3155327

42. Ott K, Tiemann M, Hennig P. Uncertainty and Structure in Neural Ordinary Differential Equations. arXiv preprint arXiv:230513290. 2023;.

43. Hornik K, Stinchcombe MB, White HL. Multilayer feedforward networks are universal approximators. Neural Networks. 1989; 2:359–366. https://doi.org/10.1016/0893-6080(89)90020-8

44. Ascher UM, Petzold LR. Computer methods for ordinary differential equations and differential-algebraic equations. vol. 61. Siam; 1998.

45. Griffiths DF, Higham DJ. Numerical Methods for Ordinary Differential Equations: Initial Value Problems. Springer Undergraduate Mathematics Series. Springer London; 2010. Available from: https://books.google.com/books?id=HrrZop_3bacC.

46. Hairer E, Nørsett SP, Wanner G. Solving Ordinary Differential Equations I: Nonstiff Problems. Springer Series in Computational Mathematics. Springer Berlin Heidelberg; 2008. Available from: https://books.google.com/books?id=cfZDAAAAQBAJ.

47. Peter IS. Gene Regulatory Networks. Current Topics in Developmental Biology. Elsevier Science; 2020. Available from: https://books.google.com/books?id=ynfnDwAAQBAJ.

48. Gutkind JS. Signaling Networks and Cell Cycle Control: The Molecular Basis of Cancer and Other Diseases. Cancer Drug Discovery and Development. Humana Press; 2000. Available from: https://books.google.com/books?id=7kKuBgAAQBAJ.

49. Soustelle M. An Introduction to Chemical Kinetics. ISTE. Wiley; 2013. Available from: https://books.google.com/books?id=rkLSOZCUqqUC.

50. McCallum H. Population Parameters: Estimation for Ecological Models. Ecological Methods and Concepts. Wiley; 2008. Available from: https://books.google.com/books?id=e7gk-ocBhqcC.

51. Magal P, Auger P, Ruan S, Ballyk M, de la Parra RB, Fitzgibbon WE, et al. Structured Population Models in Biology and Epidemiology. Lecture Notes in Mathematics. Springer Berlin Heidelberg; 2008. Available from: https://books.google.com/books?id=lrxqCQAAQBAJ.

52. Fan F, Xiong J, Wang G. Universal approximation with quadratic deep networks. Neural Networks. 2020; 124:383–392. https://doi.org/10.1016/j.neunet.2020.01.007 PMID: 32062373

53. Horn RA, Horn RA, Johnson CR. Topics in Matrix Analysis. Cambridge University Press; 1994.

54. Kass RE, Tierney L, Kadane JB. Laplace's method in Bayesian analysis. Contemporary Mathematics. 1991; 115:89–99. https://doi.org/10.1090/conm/115/07

55. Gelman A, Carlin JB, Stern HS, Dunson DB, Vehtari A, Rubin DB. Bayesian Data Analysis, Third Edition. Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis; 2013. Available from: https://books.google.com/books?id=ZXL6AQAAQBAJ.

56. Bradbury J, Frostig R, Hawkins P, Johnson MJ, Leary C, Maclaurin D, et al. JAX: composable transformations of Python+NumPy programs; 2018. Available from: http://github.com/google/jax.

57. Babuschkin I, Baumli K, Bell A, Bhupatiraju S, Bruce J, Buchlovsky P, et al. The DeepMind JAX Ecosystem; 2020. Available from: http://github.com/deepmind.

58. Lemaréchal C. Cauchy and the gradient method. Doc Math Extra. 2012; 251(254):10.

59. Hadamard J. Mémoire sur le problème d'analyse relatif à l'équilibre des plaques élastiques encastrées. vol. 33. Imprimerie nationale; 1908.

60. Kingma DP, Ba J. Adam: A Method for Stochastic Optimization; 2017.

61. Kim S, Ji W, Deng S, Ma Y, Rackauckas C. Stiff neural ordinary differential equations. Chaos: An Interdisciplinary Journal of Nonlinear Science. 2021; 31(9). PMID: 34598467

62. Fehlberg E. Classical fifth-, sixth-, seventh-, and eighth-order Runge-Kutta formulas with stepsize control. National Aeronautics and Space Administration; 1968.

63. Ben-Israel A, Greville TNE. Generalized Inverses: Theory and Applications. CMS Books in Mathematics. Springer New York; 2006. Available from: https://books.google.com/books?id=abEPBwAAQBAJ.

64. Bobrovsky BZ, Mayer-Wolf E, Zakai M. Some Classes of Global Cramer-Rao Bounds. The Annals of Statistics. 1987; 15. https://doi.org/10.1214/aos/1176350602

65. Chen MH, Shao QM, Ibrahim JG. Monte Carlo Methods in Bayesian Computation. Springer Series in Statistics. Springer New York; 2012. Available from: https://books.google.com/books?id=4IrbBwAAQBAJ.

66. Liang F, Liu C, Carroll R. Advanced Markov Chain Monte Carlo Methods: Learning from Past Samples. Wiley Series in Computational Statistics. Wiley; 2011. Available from: https://books.google.com/books?id=ZmKgUO2PVpIC.

67. McElreath R. Statistical Rethinking: A Bayesian Course with Examples in R and Stan. Chapman & Hall/CRC Texts in Statistical Science. CRC Press; 2018. Available from: https://books.google.com/books?id=T3FQDwAAQBAJ.

68. Jospin LV, Laga H, Boussaid F, Buntine W, Bennamoun M. Hands-On Bayesian Neural Networks—A Tutorial for Deep Learning Users. IEEE Computational Intelligence Magazine. 2022; 17(2):29–48. https://doi.org/10.1109/MCI.2022.3155327

69. Duane S, Kennedy AD, Pendleton BJ, Roweth D. Hybrid Monte Carlo. Physics Letters B. 1987; 195 (2):216–222. https://doi.org/10.1016/0370-2693(87)91197-X

70. Neal RM. Bayesian learning for neural networks. vol. 118. Springer Science & Business Media; 2012.

71. Leimkuhler B, Reich S. Simulating Hamiltonian Dynamics. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press; 2004. Available from: https://books.google.com/books?id=tpb-tnsZi5YC.

72. Whitlock PA, Kalos M. Monte Carlo Methods. vol. 1. Wiley; 1986.

73. Tierney L. Markov chains for exploring posterior distributions. the Annals of Statistics. 1994; 22:1701–1728. https://doi.org/10.1214/aos/1176325755

74. Hastings WK. Monte Carlo sampling methods using Markov chains and their applications. Biometrika. 1970; 57(1):97–109. https://doi.org/10.1093/biomet/57.1.97

75. Chib S, Greenberg E. Understanding the Metropolis-Hastings Algorithm. The American Statistician. 1995; 49(4):327–335. https://doi.org/10.1080/00031305.1995.10476177

76. Hoffman MD, Gelman A, et al. The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. J Mach Learn Res. 2014; 15(1):1593–1623.

77. Lao J, Louf R. Blackjax: A sampling library for JAX; 2020. Available from: http://github.com/blackjax-devs/blackjax.
78. Cinelli LP, Marins MA, da Silva EAB, Netto SL. Variational Methods for Machine Learning with Applications to Deep Networks. Springer International Publishing; 2021.
79. Nakajima S, Watanabe K, Sugiyama M. Variational Bayesian Learning Theory. Cambridge University Press; 2019.
80. Šmídl V, Quinn A. The Variational Bayes Method in Signal Processing. Signals and Communication Technology. Springer Berlin Heidelberg; 2006.
81. Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, et al. SymPy: symbolic computing in Python. PeerJ Computer Science. 2017; 3:e103. https://doi.org/10.7717/peerj-cs.103
82. Bromiley PA. Products and Convolutions of Gaussian Probability Density Functions; 2013.
83. Smith SW. CHAPTER 15—Moving Average Filters. In: Smith SW, editor. Digital Signal Processing. Boston: Newnes; 2003. p. 277–284.
84. Haynes D, Corns S, Venayagamoorthy GK. An Exponential Moving Average algorithm. In: 2012 IEEE Congress on Evolutionary Computation; 2012. p. 1–8.
85. KALMAN R, BUCY R. New Results in Linear Filtering and Prediction Theory1. space. 1961; 15:150–155.
86. Wang Y. Smoothing splines: methods and applications. CRC press; 2011.
87. Cleveland WS, Loader C. Smoothing by local regression: Principles and methods. In: Statistical Theory and Computational Aspects of Smoothing: Proceedings of the COMPSTAT'94 Satellite Meeting held in Semmering, Austria, 27–28 August 1994. Springer; 1996. p. 10–49.
88. Wand MP, Jones MC. Kernel smoothing. CRC press; 1994.
89. Selesnick IW, Burrus CS. Generalized digital Butterworth filter design. IEEE Transactions on signal processing. 1998; 46(6):1688–1694. https://doi.org/10.1109/78.678493
90. Vetterli M, Kovačević J, Goyal VK. Foundations of Signal Processing. Cambridge University Press; 2014. Available from: https://books.google.com/books?id=LBZEBAAAQBAJ.
91. Roberts S, Osborne M, Ebden M, Reece S, Gibson N, Aigrain S. Gaussian processes for time-series modelling. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences. 2013; 371(1984):20110550. https://doi.org/10.1098/rsta.2011.0550 PMID: 23277607
92. Genton MG. Classes of kernels for machine learning: a statistics perspective. Journal of machine learning research. 2001; 2(Dec):299–312.
93. Kocijan J. Modelling and control of dynamic systems using Gaussian process models. Springer; 2016.
94. Duvenaud D. Automatic model construction with Gaussian processes; 2014.
95. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research. 2011; 12:2825–2830.
96. Heek J, Levskaya A, Oliver A, Ritter M, Rondepierre B, Steiner A, et al. Flax: A neural network library and ecosystem for JAX; 2023. Available from: http://github.com/google/flax.
97. Lotka A. Elements of physical biology. Williams and Wilkins Company; 1925.
98. Volterra V. Variazioni e fluttuazioni del numero d'individui in specie animali conviventi. Società anonima tipografica "Leonardo da Vinci"; 1926.
99. Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods. 2020; 17:261–272. https://doi.org/10.1038/s41592-019-0686-2 PMID: 32015543
100. Dormand JR, Prince PJ. A family of embedded Runge-Kutta formulae. Journal of Computational and Applied Mathematics. 1980; 6(1):19–26. https://doi.org/10.1016/0771-050X(80)90013-3
101. Van Etten WC. Introduction to Random Signals and Noise. Wiley; 2006. Available from: https://books.google.com/books?id=E-i59byYhBUC.
102. Sisson SA, Fan Y, Beaumont M. Handbook of Approximate Bayesian Computation. Chapman & Hall/CRC Handbooks of Modern Statistical Methods. CRC Press; 2018. Available from: https://books.google.com/books?id=9QhpDwAAQBAJ.
103. Drawert B, Hellander A, Bales B, Banerjee D, Bellesia G, Daigle BJ Jr, et al. Stochastic Simulation Service: Bridging the Gap between the Computational Expert and the Biologist. PLOS Computational Biology. 2016; 12(12):1–15. https://doi.org/10.1371/journal.pcbi.1005220
104. Jiang R, Jacob B, Geiger M, Matthew S, Rumsey B, Singh P, et al. Epidemiological modeling in StochSS Live! Bioinformatics. 2021; 37. https://doi.org/10.1093/bioinformatics/btab061 PMID: 33512399

105. Singh P, Wrede F, Hellander A. Scalable machine learning-assisted model exploration and inference using Sciope. Bioinformatics. 2020;.

106. Roesch E, Rackauckas C, Stumpf M. Collocation based training of neural ordinary differential equations. Statistical Applications in Genetics and Molecular Biology. 2021; 20. https://doi.org/10.1515/sagmb-2020-0025 PMID: 34237805

107. Janson NB. Non-linear dynamics of biological systems. Contemporary Physics. 2012; 53(2):137–168. https://doi.org/10.1080/00107514.2011.644441

108. Karnopp D, Margolis DL, Rosenberg RC. System dynamics. Wiley New York; 1990.

109. Lorenz EN. Deterministic nonperiodic flow. Journal of the Atmospheric Sciences. 1963; 20:130–141. https://doi.org/10.1175/1520-0469(1963)020%3C0130:DNF%3E2.0.CO;2

110. Ott E. Chaos in dynamical systems. Cambridge university press; 2002.

111. Hirsch MW, Smale S, Devaney RL. Differential equations, dynamical systems, and an introduction to chaos. Academic press; 2012.

112. Webster PJ, Moore AM, Loschnigg JP, Leben RR. Coupled ocean–atmosphere dynamics in the Indian Ocean during 1997–98. Nature. 1999; 401(6751):356–360. https://doi.org/10.1038/43848 PMID: 16862107

113. Poland D. Cooperative catalysis and chemical chaos: a chemical model for the Lorenz equations. Physica D: Nonlinear Phenomena. 1993; 65:86–99. https://doi.org/10.1016/0167-2789(93)90006-M

114. Haken H. Analogy between higher instabilities in fluids and lasers. Physics Letters A. 1975; 53(1):77–78. https://doi.org/10.1016/0375-9601(75)90353-9

115. Cuomo KM, Oppenheim AV. Circuit implementation of synchronized chaos with applications to communications. Phys Rev Lett. 1993; 71:65–68. https://doi.org/10.1103/PhysRevLett.71.65 PMID: 10054374

116. Hemati N. Strange attractors in brushless DC motors. IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications. 1994; 41(1):40–45. https://doi.org/10.1109/81.260218

117. Knobloch E. Chaos in the segmented disc dynamo. Physics Letters A. 1981; 82(9):439–440. https://doi.org/10.1016/0375-9601(81)90274-7