Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

rodeo: A Probabilistic Solver for Ordinary Differential Equations

Mohan Wu
University of Waterloo

Martin Lysy
University of Waterloo

Abstract

Parameter estimation for ordinary differential equations (ODEs) plays a fundamental role in the analysis of dynamical systems. Generally lacking closed-form solutions, ODEs are traditionally approximated using deterministic solvers. However, there is a growing body of evidence to suggest that probabilistic ODE solvers produce more reliable parameter estimates by better accounting for numerical uncertainty. Here we present **rodeo**, a Python library providing a fast, lightweight, and extensible interface to a large family of probabilistic solvers. In particular, it provides a solver scaling linearly in both the number of evaluation points and system variables. This, in combination with automatic differentiation and just-in-time (JIT) compiling techniques, is shown on several examples to compare favorably with state-of-the-art deterministic solvers.

Keywords: Differential equations, initial value problem, probabilistic solution, Kalman filter, Python, **JAX**.

1. Introduction

Parameter estimation for ordinary differential equations (ODEs) is an important problem in the natural sciences and engineering. Since most ODEs do not have-closed form solutions, they must be approximated by numerical methods. Traditionally this has been done with deterministic algorithms (e.g., Butcher 2008; Griffiths and Higham 2010; Atkinson *et al.* 2009). However, an emerging field of *probabilistic numerics* (Diaconis 1988; Skilling 1992; Hennig *et al.* 2015) indicates that probabilistic ODE solvers, which directly account for uncertainty in the numerical approximation, provide more reliable parameter estimates in ODE learning problems (Chkrebtii *et al.* 2016; Conrad *et al.* 2017).

A number of probabilistic ODE solvers that have been recently proposed are summarized in Table 1.

Table 1: Various probabilistic ODE solvers. *Method/Software:* The proposed method in the accompanying reference(s) or the name of the software library if it is publicly available. *Latent Variables:* Whether the proposed method involves a large number of latent variables which need to be integrated out using computationally intensive methods. *Implementation:* The programming language employed if an implementation is available. *Compiled:* Whether the implementation is written in compiled code. *Autodiff:* Whether the implementation enables automatic differentiation of user-defined models.

Method/Software	Latent Variables	Implementation	Compiled	Autodiff
Gaussian Process Regression (Calderhead *et al.* 2009)	Y	Matlab	N	N
Adaptive Gradient Matching (Dondelinger *et al.* 2013)	Y			
odegp (Barber and Wang 2014)	Y	Matlab	N	N
Gaussian Markov Runge-Kutta (Schober *et al.* 2014)	Y	Matlab	N	N
uqdes (Chkrebtii *et al.* 2016)	Y	Matlab	N	N
Bayesian Quadrature Filtering (Kersting and Hennig 2016)	Y			
Adams-Bashforth (Teymur *et al.* 2016)	Y			
CollocInfer (Hooker *et al.* 2016)	Y	R/Matlab	N	N
deBInfer (Boersch-Supan *et al.* 2017)	Y	R	N	N
GPmat (Ghosh *et al.* 2017)	Y	Matlab	N	N
Scalable Variational Inference (Gorbach *et al.* 2017)	Y			
Multiphase MCMC (Lazarus *et al.* 2018)	Y			
FGPGM (Wenk *et al.* 2019)	Y	Python	N	N
deGradInfer (Macdonald and Dondelinger 2020)	Y	R	N	N
simode (Dattner and Yaari 2020)	Y	R	N	N
KGode (Niu *et al.* 2021)	Y	R	N	N
MAGI (Yang *et al.* 2021)	Y	R/Python/Matlab	Y	N
ProbNum (Tronarp *et al.* 2018; Schober *et al.* 2019; Krämer and Hennig 2020; Wenger *et al.* 2021; Krämer *et al.* 2021)	N	Python	N	N
pCODE (Wang and Cao 2022)	Y	R	N	N
ProbNumDiffEq.jl (Bosch *et al.* 2021, 2022; Tronarp *et al.* 2022)	N	Julia	Y	Y

Of these methods, few have been compared directly to deterministic solvers in terms of speed and accuracy. Partly this is because many probabilistic solvers incur a high cost per time step, e.g., involving large matrix manipulations, or having a large number of latent variables to integrate out via computationally intensive methods such as Markov chain Monte Carlo (MCMC) or particle filtering (see "Latent Variables" column of Table 1). It is also because in publicly available software implementations, design considerations are often guided by simplicity of interface rather than performance optimization (see "Compiled" column of Table 1).

A notable exception to to these limitations is the Julia library **ProbNumDiffEq.jl** (Bosch *et al.* 2021, 2022; Tronarp *et al.* 2022), which is shown to outperform a number of deterministic solvers (Bosch *et al.* 2022). Moreover, **ProbNumDiffEq.jl** is seemlessly compatible with the Julia automatic differentiation (AD) engines, which allows for deployment of many gradient-based parameter inference algorithms (more on this in Section 5).

Here we present **rodeo**[1], a probabilistic ODE solver written in Python. **rodeo** provides a lightweight and extensible family of approximations to a nonlinear Bayesian filtering paradigm common to many probabilistic solvers (Tronarp *et al.* 2018). Using the **JAX** library (Bradbury *et al.* 2018), the **rodeo** algorithms are written in AD compatible, just-in-time (JIT) compiled code which reads naturally to **NumPy** users. The solver focuses on a family of fast an accurate ODE approximations based on the Kalman filter (Chkrebtii *et al.* 2016; Schober *et al.* 2019), of which a subset can be made to scale linearly in the number of system variables (Krämer *et al.* 2021). We show how to construct a likelihood approximation with low operation count, ideal for reverse-mode automatic differentiation. Several examples illustrate that **rodeo** achieves high accuracy at speeds equal or faster than commonly-used deterministic ODE solvers.

2. Probabilistic ODE Solvers via Bayesian Filtering

rodeo is designed to solve arbitrary-order multi-variable ODE systems which satisfy an initial value problem (IVP). For a univariate function $x(t)$, an ODE-IVP is of the form

$$\boldsymbol{W}\boldsymbol{x}(t) = \boldsymbol{f}(\boldsymbol{x}(t), t), \qquad \boldsymbol{x}(t_{\min}) = \boldsymbol{v}, \qquad t \in [t_{\min}, t_{\max}], \tag{1}$$

where $\boldsymbol{x}(t) = (x^{(0)}(t), x^{(1)}(t), \ldots, x^{(p-1)}(t))$ consists of $x(t) = x^{(0)}(t)$ and its first $p-1$ derivatives, $\boldsymbol{W}_{r\times p}$ is a coefficient matrix, and $\boldsymbol{f}(\boldsymbol{x}(t), t) = (f_1(\boldsymbol{x}(t), t), \ldots, f_r(\boldsymbol{x}(t), t))$ is a nonlinear function representing r equations. The ODE in (1) can always be written in the more usual "first-order" form,

$$\frac{\mathrm{d}}{\mathrm{d}t} x^{(i)}(t) = g_i(\boldsymbol{x}(t), t), \qquad i = 0, \ldots, p-2. \tag{2}$$

Equivalently, any ODE in the first-order form (2) can be written in the form of (1) with $\boldsymbol{f}(\boldsymbol{x}(t), t) = (g_0(\boldsymbol{x}(t), t), \ldots, g_{p-2}(\boldsymbol{x}(t), t))$ and $\boldsymbol{W}_{(p-1)\times p} = [\boldsymbol{0}_{(p-1)\times 1} \mid \boldsymbol{I}_{(p-1)\times(p-1)}]$. While the first-order form (2) is unique, the formulation in (1) often allows a given ODE-IVP to be specified in multiple ways, which in turn could affect the accuracy of the stochastic solver to be described below.

Unlike deterministic solvers, **rodeo** employs a probabilistic approach to solving (1) based on a widely-adopted paradigm of Bayesian nonlinear filtering (Tronarp *et al.* 2018). This approach consists of putting a Gaussian Markov process prior on $\boldsymbol{x}(t)$, and updating it with information from the ODE-IVP (1) at time points $t = t_0, \ldots, t_N$, where $t_n = n(t_{\max} - t_{\min})/N$. Specifically,

[1] https://github.com/mlysy/rodeo

let $\boldsymbol{x}_n = \boldsymbol{x}(t_n)$ and consider the general indexing notation $\boldsymbol{x}_{m:n} = (\boldsymbol{x}_m, \ldots, \boldsymbol{x}_n)$. If $\boldsymbol{x}(t)$ is the solution to (1), we would have $\boldsymbol{z}_n = \boldsymbol{W}\boldsymbol{x}_n - \boldsymbol{f}(\boldsymbol{x}_n, t_n) = \boldsymbol{0}$. Based on this observation, Tronarp *et al.* (2018) consider a state-space model in $\boldsymbol{x}_n$ and $\boldsymbol{z}_n$ of the form

$$\begin{aligned} \boldsymbol{x}_{n+1} \mid \boldsymbol{x}_n &\sim \text{Normal}(\boldsymbol{Q}\boldsymbol{x}_n, \boldsymbol{R}) \\ \boldsymbol{z}_n &\overset{\text{ind}}{\sim} \text{Normal}(\boldsymbol{W}\boldsymbol{x}_n - \boldsymbol{f}(\boldsymbol{x}_n, t_n), \boldsymbol{V}_n), \end{aligned} \tag{3}$$

where $\boldsymbol{x}_0 = \boldsymbol{v}$, $\boldsymbol{Q}$ and $\boldsymbol{R}$ are determined by the Gaussian Markov process prior, and $\boldsymbol{V}_{1:N}$ are tuning parameters. The stochastic ODE solution is then given by the posterior distribution

$$p(\boldsymbol{x}_{1:N} \mid \boldsymbol{z}_{1:N} = \boldsymbol{0}). \tag{4}$$

As $N \to \infty$, the stochastic ODE solution (4) gets arbitrarily close to the true ODE solution as the computational complexity increases (Kersting *et al.* 2020b). However, the posterior distribution (4) generally cannot be sampled from directly. Alternatives include Markov chain Monte Carlo sampling (Yang *et al.* 2021) and particle filtering (Tronarp *et al.* 2018). A less accurate but ostensibly much faster approach is to linearize (3) using Taylor expansions (Tronarp *et al.* 2018). This amounts to the surrogate model

$$\begin{aligned} \boldsymbol{x}_{n+1} \mid \boldsymbol{x}_n &\sim \text{Normal}(\boldsymbol{Q}\boldsymbol{x}_n, \boldsymbol{R}) \\ \boldsymbol{z}_n &\overset{\text{ind}}{\sim} \text{Normal}((\boldsymbol{W} + \boldsymbol{B}_n)\boldsymbol{x}_n + \boldsymbol{a}_n, \boldsymbol{V}_n), \end{aligned} \tag{5}$$

where $\boldsymbol{a}_n$, $\boldsymbol{B}_n$ and $\boldsymbol{V}_n$ are obtained sequentially by a process called *model interrogation* (Chkrebtii *et al.* 2016) (see Section 2.1). The upshot is that a random draw from the posterior $p_{\text{lin}}(\boldsymbol{x}_{0:N} \mid \boldsymbol{z}_{1:N})$ induced by the working model (5) can be easily obtained using the Kalman filtering and smoothing algorithms. Denote the posterior mean and variance of the solution process as $\boldsymbol{\mu}_{m|n} = \mathsf{E}_{\text{lin}}[\boldsymbol{x}_m \mid \boldsymbol{z}_{1:n}]$ and $\boldsymbol{\Sigma}_{m|n} = \text{var}_{\text{lin}}(\boldsymbol{x}_m \mid \boldsymbol{z}_{1:n})$. Using this notation, the steps of the Kalman ODE solver are presented in Algorithm 1. The formulation for the Kalman functions in Algorithm 1 can be found in the Appendix A.

2.1. Model Interrogations

At the heart of the Kalman ODE solver is the specification of the model interrogation in line 19 of Algorithm 1. Several proposals from the literature are summarized in Table 2. The

Table 2: Various model interrogation methods.

Reference	$\boldsymbol{a}_n$	$\boldsymbol{B}_n$	$\boldsymbol{V}_n$
Kersting and Hennig (2016)	$-\mathsf{E}[\boldsymbol{f}(\boldsymbol{x}_n, t_n) \mid \boldsymbol{z}_{1:n-1}]$	$\boldsymbol{0}$	$\text{var}(\boldsymbol{f}(\boldsymbol{x}_n, t_n) \mid \boldsymbol{z}_{1:n-1})$
Chkrebtii *et al.* (2016)	$-\boldsymbol{f}(\boldsymbol{x}_n^\star, t_n) : \boldsymbol{x}_n^\star \sim \text{Normal}(\boldsymbol{\mu}_{n\|n-1}, \boldsymbol{\Sigma}_{n\|n-1})$	$\boldsymbol{0}$	$\boldsymbol{W}\boldsymbol{\Sigma}_{n\|n-1}\boldsymbol{W}'$
Schober *et al.* (2019)	$-\boldsymbol{f}(\boldsymbol{\mu}_{n\|n-1}, t_n)$	$\boldsymbol{0}$	$\boldsymbol{0}$
Tronarp *et al.* (2018)	$-\boldsymbol{f}(\boldsymbol{\mu}_{n\|n-1}, t_n) + \boldsymbol{J}_f(\boldsymbol{\mu}_{n\|n-1}, t_n)\boldsymbol{\mu}_{n\|n-1}$	$-\boldsymbol{J}_f(\boldsymbol{\mu}_{n\|n-1}, t_n)$	$\boldsymbol{0}$
Krämer *et al.* (2021)	$-\boldsymbol{f}(\boldsymbol{\mu}_{n\|n-1}, t_n) + \boldsymbol{J}_f^*(\boldsymbol{\mu}_{n\|n-1}, t_n)\boldsymbol{\mu}_{n\|n-1}$	$-\boldsymbol{J}_f^*(\boldsymbol{\mu}_{n\|n-1}, t_n)$	$\boldsymbol{0}$

method of Kersting and Hennig (2016) is shown to be optimal in a class of Gaussian filtering approximations (Tronarp *et al.* 2018) and is therefore the most accurate. However, it requires evaluation of the intractable integrals by numerical methods, e.g., Gaussian quadrature (Kersting and Hennig 2016). The method of Chkrebtii *et al.* (2016) can be viewed as approximating these integrals by Monte Carlo (Schober *et al.* 2019). The method of Schober *et al.* (2019)

Algorithm 1 The Kalman ODE solver.

procedure `kalman_ode_sim`$(\boldsymbol{W}, \boldsymbol{f}(\boldsymbol{x}, t), \boldsymbol{v}, \boldsymbol{Q}, \boldsymbol{R})$
  $\boldsymbol{\mu}_{0|0}, \ldots, \boldsymbol{\mu}_{N|N}, \boldsymbol{\Sigma}_{0|0}, \ldots, \boldsymbol{\Sigma}_{N|N} \leftarrow$ `kalman_ode_forward`$(\boldsymbol{W}, \boldsymbol{f}(\boldsymbol{x}, t), \boldsymbol{v}, \boldsymbol{Q}, \boldsymbol{R})$
  $\boldsymbol{x}_N \sim \text{Normal}(\boldsymbol{\mu}_{N|N}, \boldsymbol{\Sigma}_{N|N})$
  for $n = N - 1 : 0$ **do**
    $\boldsymbol{x}_n \leftarrow$ `kalman_sample`$(\boldsymbol{x}_{n+1}, \boldsymbol{\mu}_{n|n}, \boldsymbol{\Sigma}_{n|n}, \boldsymbol{\mu}_{n+1|n}, \boldsymbol{\Sigma}_{n+1|n}, \boldsymbol{Q})$
  return $\boldsymbol{x}_{0:N}$

procedure `kalman_ode_mv`$(\boldsymbol{W}, \boldsymbol{f}(\boldsymbol{x}, t), \boldsymbol{v}, \boldsymbol{Q}, \boldsymbol{R})$
  $\boldsymbol{\mu}_{0|0}, \ldots, \boldsymbol{\mu}_{N|N}, \boldsymbol{\Sigma}_{0|0}, \ldots, \boldsymbol{\Sigma}_{N|N} \leftarrow$ `kalman_ode_forward`$(\boldsymbol{W}, \boldsymbol{f}(\boldsymbol{x}, t), \boldsymbol{v}, \boldsymbol{Q}, \boldsymbol{R})$
  for $n = N - 1 : 0$ **do**
    $\boldsymbol{\mu}_{n|N}, \boldsymbol{\Sigma}_{n|N} \leftarrow$ `kalman_smooth`$(\boldsymbol{\mu}_{n+1|N}, \boldsymbol{\Sigma}_{n+1|N}, \boldsymbol{\mu}_{n|n}, \boldsymbol{\Sigma}_{n|n}, \boldsymbol{\mu}_{n+1|n}, \boldsymbol{\Sigma}_{n+1|n}, \boldsymbol{Q})$
  return $\boldsymbol{\mu}_{0:N|N}, \boldsymbol{\Sigma}_{0:N|N}$

procedure `kalman_ode_forward`$(\boldsymbol{W}, \boldsymbol{f}(\boldsymbol{x}, t), \boldsymbol{v}, \boldsymbol{Q}, \boldsymbol{R})$
  $\boldsymbol{\mu}_{0|0}, \Sigma_{0|0} \leftarrow \boldsymbol{v}, \boldsymbol{0}$
  $\boldsymbol{z}_{1:N} \leftarrow \boldsymbol{0}$
  for $n = 1 : N$ **do**
    $\boldsymbol{\mu}_{n|n-1}, \boldsymbol{\Sigma}_{n|n-1} \leftarrow$ `kalman_predict`$(\boldsymbol{\mu}_{n-1|n-1}, \boldsymbol{\Sigma}_{n-1|n-1}, \boldsymbol{0}, \boldsymbol{Q}, \boldsymbol{R})$
    $\boldsymbol{a}_n, \boldsymbol{B}_n, \boldsymbol{V}_n \leftarrow$ `interrogate`$(\boldsymbol{\mu}_{n|n-1}, \boldsymbol{\Sigma}_{n|n-1}, \boldsymbol{W}, \boldsymbol{f}(\boldsymbol{x}, t))$ ▷ See Table 2
    $\boldsymbol{\mu}_{n|n}, \boldsymbol{\Sigma}_{n|n} \leftarrow$ `kalman_update`$(\boldsymbol{\mu}_{n|n-1}, \boldsymbol{\Sigma}_{n|n-1}, \boldsymbol{z}_n, \boldsymbol{a}_n, \boldsymbol{W} + \boldsymbol{B}_n, \boldsymbol{V}_n)$
  return $\boldsymbol{\mu}_{0|0}, \ldots, \boldsymbol{\mu}_{N|N}, \boldsymbol{\Sigma}_{0|0}, \ldots, \boldsymbol{\Sigma}_{N|N}$

is the simplest and fastest. Moreover, the posterior $p(\boldsymbol{x}_{0:N} \mid \boldsymbol{z}_{0:N})$ is a deterministic function of the inputs to the ODE solver, unlike the Monte Carlo method of Chkrebtii *et al.* (2016). However, it has been noted in Chkrebtii *et al.* (2016) that noise-free model interrogation with $\boldsymbol{V}_n = \boldsymbol{0}$ causes the solver's performance to deteriorate, which we have confirmed in numerical simulations not presented here. The method of Tronarp *et al.* (2018) uses a first order Taylor expansion instead of the zeroth order of Schober *et al.* (2019) – with $\boldsymbol{J}_f$ being the Jacobian of $\boldsymbol{x} \to \boldsymbol{f}(\boldsymbol{x}, t)$ – which has been shown to have better numerical stability (Tronarp *et al.* 2018). The method of Krämer *et al.* (2021) is a modified version of Tronarp *et al.* (2018) where the off block diagonal elements are removed in the Jacobian denoted by $\boldsymbol{J}_f^*$. Many experiments show that there is minimal loss in accuracy compared to the original method (Krämer *et al.* 2021) while greatly reducing computational complexity.

2.2. Gaussian Markov Process Prior

rodeo uses a simple and effective prior proposed by Schober *et al.* (2019); namely, that $x(t) = x^{(0)}(t)$ is $q-1$-times integrated Brownian motion (IBM), such that

$$x^{(q)}(t) = \sigma B(t). \tag{6}$$

This results in a q-dimensional continuous Gaussian Markov process $\boldsymbol{x}(t) = (x^{(0)}(t), x^{(1)}(t), \ldots, x^{(q-1)}(t))$ with matrices $\boldsymbol{Q}$ and $\boldsymbol{R}$ in (5) given by

$$Q_{ij} = \mathbf{1}_{i \le j} \frac{(\Delta t)^{j-i}}{(j-i)!}, \qquad R_{ij} = \sigma^2 \frac{(\Delta t)^{2q-1-i-j}}{(2q-1-i-j)(q-1-i)!(q-1-j)!}. \tag{7}$$

Here we have tacitly assumed that $q = p$, where $p - 1$ is the number of derivatives of $x(t)$ appearing in the ODE-IVP (1). However, it is often advantageous in practice to set $p > q$ to increase the smoothness of $x(t)$. This can be done by padding $\boldsymbol{v}$ and $\boldsymbol{W}$ in (1) with $q - p$ zeros and $q - p$ columns of zeros, respectively. For $\boldsymbol{v}$, a different method of padding is to work out the values of $x^{(k)}(a)$ for $q \leq k < p$ by taking derivatives of the ODE in (1). Thus, in what follows we continue to assume that $q = p$, explicitly describing the padding as needed in the examples of Section 5.

2.3. Multiple Variables and Blocking

For a multi-variable function $\boldsymbol{x}(t) = (x_1(t), \ldots, x_d(t))$, let $\boldsymbol{X}_k(t) = (x_k^{(0)}(t), \ldots, x_k^{(p_k-1)}(t))$ denote $x_k(t) = x_k^{(0)}(t)$ and its first $p_k - 1$ derivatives. An ODE-IVP for $\boldsymbol{x}(t)$ can then be written as

$$\boldsymbol{W}\boldsymbol{X}(t) = \boldsymbol{f}(\boldsymbol{X}(t), t), \qquad \boldsymbol{X}(t_{\min}) = \boldsymbol{v}, \qquad t \in [t_{\min}, t_{\max}], \tag{8}$$

where

$$\boldsymbol{X}(t) = (\boldsymbol{X}_1(t), \ldots, \boldsymbol{X}_d(t)). \tag{9}$$

The Kalman ODE solver 1 can be applied almost exactly for multiple ODE variables as for the univariate case. It is easy to see that Algorithm 1 scales linearly in the number of evaluation points N. However, it scales as $\mathcal{O}(d^3)$ in the number of ODE variables d, which is a significant drawback for large ODE systems.

Suppose that it were possible to rearrange the rows of $\boldsymbol{W}$ (and corresponding outputs of $f(\boldsymbol{X}(t), t)$) such that $\boldsymbol{W} = \operatorname{diag}(\boldsymbol{W}^{(1)}_{r_1 \times p_1}, \ldots, \boldsymbol{W}^{(d)}_{r_d \times p_d})$ is block diagonal. Furthermore, suppose that the outputs of the interrogation method are block diagonal, i.e., $\boldsymbol{a}_n = (\boldsymbol{a}^{(1)}_{p_1 \times 1}, \ldots, \boldsymbol{a}^{(d)}_{p_d \times 1})$, $\boldsymbol{B}_n = \operatorname{diag}(\boldsymbol{B}^{(1)}_{p_1 \times p_1}, \ldots, \boldsymbol{B}^{(d)}_{p_d \times p_d})$ and $\boldsymbol{V}_n = \operatorname{diag}(\boldsymbol{V}^{(1)}_{p_1 \times p_1}, \ldots, \boldsymbol{V}^{(d)}_{p_d \times p_d})$. This is the case for most of the interrogation methods presented in Table 2. Then by setting the Gaussian Markov prior on $\boldsymbol{X}(t)$ to consist of independent priors for each $\boldsymbol{X}_k(t)$, $k = 1, \ldots, d$, i.e., such that $\boldsymbol{Q} = \operatorname{diag}(\boldsymbol{Q}^{(1)}_{p_1 \times p_1}, \ldots, \boldsymbol{Q}^{(d)}_{p_d \times p_d})$ and $\boldsymbol{R} = \operatorname{diag}(\boldsymbol{R}^{(1)}_{p_1 \times p_1}, \ldots, \boldsymbol{R}^{(d)}_{p_d \times p_d})$, each of the steps in Algorithm 1 can be performed variable-wise, thus reducing the computational complexity from $\mathcal{O}(d^3)$ to $\mathcal{O}(d)$ (Krämer *et al.* 2021). This acceleration technique, together with the smoothing step to approximate $\boldsymbol{X}_{0:N}$ by $\boldsymbol{\mu}_{0:N|N}$, give **rodeo** a low memory footprint particularly advantageous for back-propagation. That is, while solvers with high memory cost must approximate gradients via the adjoint method (e.g., Chen *et al.* 2018; Kidger 2021), **rodeo** can simply back-propagate through the solver steps which is typically much faster (Kidger 2021).

At its core, **rodeo** uses the Kalman filter and smoother extensively. Defining $\boldsymbol{v} = (\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(d)})$, $\boldsymbol{\mu}_{m|n} = (\boldsymbol{\mu}^{(1)}_{m|n}, \ldots, \boldsymbol{\mu}^{(d)}_{m|n})$, $\boldsymbol{\Sigma}_{m|n} = \operatorname{diag}(\boldsymbol{\Sigma}^{(1)}_{m|n}, \ldots, \boldsymbol{\Sigma}^{(d)}_{m|n})$, etc., the exact steps of the **rodeo** solver are presented in Algorithm 2.

3. Parameter Inference

The parameter-dependent extension of the ODE-IVP (8) is of the form

$$\boldsymbol{W}_{\boldsymbol{\theta}}\boldsymbol{X}(t) = \boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{X}(t), t), \qquad \boldsymbol{X}(t_{\min}) = \boldsymbol{v}_{\boldsymbol{\theta}}, \qquad t \in [t_{\min}, t_{\max}]. \tag{10}$$

Algorithm 2 The **rodeo** probabilistic ODE solver. `rodeo_ode_sim()` is used to obtain a draw from $p_{\text{lin}}(\boldsymbol{X}_{0:N} \mid \boldsymbol{Z}_{0:N} = \boldsymbol{0})$, whereas `rodeo_ode_mv()` is used to obtain $\boldsymbol{\mu}_{0:N|N}$ and $\boldsymbol{\Sigma}_{0:N|N}$, where $\mu_{n|N} = E_{\text{lin}}[\boldsymbol{X}_n \mid \boldsymbol{Z}_{0:N} = \boldsymbol{0}]$ and $\boldsymbol{\Sigma}_{n|N} = \text{var}_{\text{lin}}(\boldsymbol{X}_n \mid \boldsymbol{Z}_{0:N} = \boldsymbol{0})$. Both use the same forward pass through the model interrogations in `rodeo_forward()`. The `block` versions of the Kalman algorithms are elementary extensions of the non-block versions. Thus, only the pseudocode for `kalman_block_sample()` is provided.

procedure `rodeo_ode_sim`$(\boldsymbol{W}, \boldsymbol{f}(\boldsymbol{X}, t), \boldsymbol{v}, \boldsymbol{Q}, \boldsymbol{R})$
  $(\boldsymbol{\mu}_{0|0}, \boldsymbol{\Sigma}_{0|0}), \ldots, (\boldsymbol{\mu}_{N|N}, \boldsymbol{\Sigma}_{N|N}) \leftarrow$ `rodeo_forward`$(\boldsymbol{W}, \boldsymbol{f}(\boldsymbol{X}, t), \boldsymbol{v}, \boldsymbol{Q}, \boldsymbol{R})$
  for $k = 1 : d$ **do**
    $\boldsymbol{X}_N^{(k)} \sim \text{Normal}(\boldsymbol{\mu}_{N|N}^{(k)}, \boldsymbol{\Sigma}_{N|N}^{(k)})$
  $\boldsymbol{X}_N \leftarrow (\boldsymbol{X}_N^{(1)}, \ldots, \boldsymbol{X}_N^{(d)})$
  for $n = N - 1 : 0$ **do**
    $\boldsymbol{X}_n \leftarrow$ `kalman_block_sample`$(\boldsymbol{X}_{n+1}, \boldsymbol{\mu}_{n|n}, \boldsymbol{\Sigma}_{n|n}, \boldsymbol{\mu}_{n+1|n}, \boldsymbol{\Sigma}_{n+1|n}, \boldsymbol{Q})$
  return $\boldsymbol{X}_{0:N}$

procedure `rodeo_ode_mv`$(\boldsymbol{W}, \boldsymbol{f}(\boldsymbol{X}, t), \boldsymbol{v}, \boldsymbol{Q}, \boldsymbol{R})$
  $(\boldsymbol{\mu}_{0|0}, \boldsymbol{\Sigma}_{0|0}), \ldots, (\boldsymbol{\mu}_{N|N}, \boldsymbol{\Sigma}_{N|N}) \leftarrow$ `rodeo_forward`$(\boldsymbol{W}, \boldsymbol{f}(\boldsymbol{X}, t), \boldsymbol{v}, \boldsymbol{Q}, \boldsymbol{R})$
  for $n = N - 1 : 0$ **do**
    $\boldsymbol{\mu}_{n|N}, \boldsymbol{\Sigma}_{n|N} \leftarrow$ `kalman_block_smooth`$(\boldsymbol{\mu}_{n+1|N}, \boldsymbol{\Sigma}_{n+1|N}, \boldsymbol{\mu}_{n|n}, \boldsymbol{\Sigma}_{n|n}, \boldsymbol{\mu}_{n+1|n}, \boldsymbol{\Sigma}_{n+1|n}, \boldsymbol{Q})$
  return $\boldsymbol{\mu}_{0:N|N}, \boldsymbol{\Sigma}_{0:N|N}$

procedure `rodeo_forward`$(\boldsymbol{W}, \boldsymbol{f}(\boldsymbol{X}, t), \boldsymbol{v}, \boldsymbol{Q}, \boldsymbol{R})$
  $\boldsymbol{\mu}_{0|0}, \Sigma_{0|0} \leftarrow \boldsymbol{v}, \boldsymbol{0}$
  $\boldsymbol{Z}_{1:N} \leftarrow \boldsymbol{0}$
  for $n = 1 : N$ **do**
    $\boldsymbol{\mu}_{n|n-1}, \boldsymbol{\Sigma}_{n|n-1} \leftarrow$ `kalman_block_predict`$(\boldsymbol{\mu}_{n-1|n-1}, \boldsymbol{\Sigma}_{n-1|n-1}, \boldsymbol{0}, \boldsymbol{Q}, \boldsymbol{R})$
    $\boldsymbol{a}_n, \boldsymbol{B}_n, \boldsymbol{V}_n \leftarrow$ `block_interrogate`$(\boldsymbol{\mu}_{n|n-1}, \boldsymbol{\Sigma}_{n|n-1}, \boldsymbol{W}, \boldsymbol{f}(\boldsymbol{X}, t))$
    $\boldsymbol{\mu}_{n|n}, \boldsymbol{\Sigma}_{n|n} \leftarrow$ `kalman_block_update`$(\boldsymbol{\mu}_{n|n-1}, \boldsymbol{\Sigma}_{n|n-1}, \boldsymbol{Z}_n, \boldsymbol{a}_n, \boldsymbol{W} + \boldsymbol{B}_n, \boldsymbol{V}_n)$
  return $(\boldsymbol{\mu}_{0|0}, \boldsymbol{\Sigma}_{0|0}), \ldots, (\boldsymbol{\mu}_{N|N}, \boldsymbol{\Sigma}_{N|N})$

procedure `kalman_block_sample`$(\boldsymbol{X}_{n+1}, \boldsymbol{\mu}_{n|n}, \boldsymbol{\Sigma}_{n|n}, \boldsymbol{\mu}_{n+1|n}, \boldsymbol{\Sigma}_{n+1|n}, \boldsymbol{Q})$
  for $k = 1 : d$ **do**
    $\boldsymbol{X}_n^{(k)} \leftarrow$ `kalman_sample`$(\boldsymbol{X}_{n+1}^{(k)}, \boldsymbol{\mu}_{n|n}^{(k)}, \boldsymbol{\Sigma}_{n|n}^{(k)}, \boldsymbol{\mu}_{n+1|n}^{(k)}, \boldsymbol{\Sigma}_{n+1|n}^{(k)}, \boldsymbol{Q}^{(k)})$
  return $\boldsymbol{X}_n = (\boldsymbol{X}_n^{(1)}, \ldots, \boldsymbol{X}_n^{(d)})$

The updated surrogate model of (5) is then

$$\begin{aligned} \boldsymbol{X}_{n+1} \mid \boldsymbol{X}_n &\sim \text{Normal}(\boldsymbol{Q}_{\boldsymbol{\eta}} \boldsymbol{X}_n, \boldsymbol{R}_{\boldsymbol{\eta}}) \\ \boldsymbol{Z}_n &\overset{\text{ind}}{\sim} \text{Normal}((\boldsymbol{W}_{\boldsymbol{\theta}} + \boldsymbol{B}_n)\boldsymbol{X}_n + \boldsymbol{a}_n, \boldsymbol{V}_n), \end{aligned} \tag{11}$$

where the Gaussian Markov process parameters $\boldsymbol{Q}_{\boldsymbol{\eta}}$ and $\boldsymbol{R}_{\boldsymbol{\eta}}$ depend on tuning parameters $\boldsymbol{\eta}$. The learning problem consists of estimating the unknown parameters $\boldsymbol{\theta}$ which determine $\boldsymbol{X}(t)$ in (10) from noisy observations $\boldsymbol{Y}_{0:M} = (\boldsymbol{Y}_0, \ldots, \boldsymbol{Y}_M)$, recorded at times $t = t'_0, \ldots, t'_M$

under the measurement model

$$\boldsymbol{Y}_i \overset{\text{ind}}{\sim} p(\boldsymbol{Y}_i \mid \boldsymbol{X}(t_i'), \boldsymbol{\phi}). \tag{12}$$

Parameter inference is conducted via the likelihood function for $\boldsymbol{\Theta} = (\boldsymbol{\theta}, \boldsymbol{\phi}, \boldsymbol{\eta})$,

$$\begin{aligned} \mathcal{L}(\boldsymbol{\Theta} \mid \boldsymbol{Y}_{0:M}) &\propto p(\boldsymbol{Y}_{0:M} \mid \boldsymbol{Z}_{1:N} = \boldsymbol{0}, \boldsymbol{\Theta}) \\ &= \int p(\boldsymbol{Y}_{0:M} \mid \boldsymbol{X}_{0:N}, \boldsymbol{\phi}) p(\boldsymbol{X}_{0:N} \mid \boldsymbol{Z}_{1:N} = \boldsymbol{0}, \boldsymbol{\theta}, \boldsymbol{\eta}) \, \mathrm{d}\boldsymbol{X}_{0:N}. \end{aligned} \tag{13}$$

We present four parameter inference methods within the **rodeo** framework.

3.1. Basic Method

A basic approximation to the likelihood function (13) takes the posterior mean $\boldsymbol{\mu}_{0:N|N}(\boldsymbol{\theta}, \boldsymbol{\eta}) = \mathsf{E}_{\text{lin}}[\boldsymbol{X}_{0:N} \mid \boldsymbol{Z}_{1:N}, \boldsymbol{\theta}, \boldsymbol{\eta}]$ of Algorithm 2 and simply plugs it into the measurement model (12), such that

$$\hat{\mathcal{L}}(\boldsymbol{\Theta} \mid \boldsymbol{Y}_{0:M}) = \prod_{i=0}^{M} p(\boldsymbol{Y}_i \mid \boldsymbol{X}_{n(i)} = \boldsymbol{\mu}_{n(i)|N}(\boldsymbol{\theta}, \boldsymbol{\eta}), \boldsymbol{\phi}), \tag{14}$$

where in terms of the ODE solver discretization time points $t = t_0, \ldots, t_N$, $N \geq M$, the mapping $n(\cdot)$ is such that $t_{n(i)} = t_i'$. A similar approach is proposed in Schober *et al.* (2019) but without the backward pass in Algorithm 2. The basic approximation (14) is very simple, but does not propagate the uncertainty in the probabilistic ODE solver to the calculation of the likelihood.

3.2. Fenrir

The Fenrir method (Tronarp *et al.* 2022) extends a likelihood approximation developed in Kersting *et al.* (2020a). It is applicable to Gaussian measurement models of the form

$$\boldsymbol{Y}_i \overset{\text{ind}}{\sim} \text{Normal}(\boldsymbol{D}_i^{(\phi)} \boldsymbol{X}_{n(i)}, \boldsymbol{\Omega}_i^{(\phi)}). \tag{15}$$

Fenrir begins by using the surrogate model (11) to estimate $p(\boldsymbol{X}_{0:N} \mid \boldsymbol{Z}_{1:N} = \boldsymbol{0}, \boldsymbol{\theta}, \boldsymbol{\eta})$. This results in a Gaussian non-homogeneous Markov model going backwards in time,

$$\begin{aligned} \boldsymbol{X}_N &\sim \text{Normal}(\boldsymbol{b}_N, \boldsymbol{C}_N) \\ \boldsymbol{X}_n \mid \boldsymbol{X}_{n+1} &\sim \text{Normal}(\boldsymbol{A}_n \boldsymbol{X}_n + \boldsymbol{b}_n, \boldsymbol{C}_n), \end{aligned} \tag{16}$$

where the coefficients $\boldsymbol{A}_{0:N-1}$, $\boldsymbol{b}_{0:N}$, and $\boldsymbol{C}_{0:N}$ can be derived using the Kalman filtering and smoothing recursions (Tronarp *et al.* 2022). In combination with the Gaussian measurement model (15), the integral in the likelihood function (13) can be computed analytically.

In order to benefit from the speed increase of blocking, the matrices $\boldsymbol{D}_i^{(\phi)}$ and $\boldsymbol{\Omega}_i^{(\phi)}$ in (15) must be block diagonal with d blocks of size $s \times p$ and $s \times s$ respectively where s is the number of measurements per variable. This is illustrated in Example 5.4 and the exact steps are presented in Algorithm 3.

3.3. DALTON

The Fenrir approximation can be applied to any of the model interrogations in Table 2. However, each of these model interrogation methods is data-free, in the sense that it does

Algorithm 3 The Fenrir probabilistic ODE likelihood approximation.

procedure `fenrir`($\boldsymbol{W} = \boldsymbol{W}_{\boldsymbol{\theta}}, \boldsymbol{f}(\boldsymbol{X}, t) = \boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{X}, t), \boldsymbol{v} = \boldsymbol{v}_{\boldsymbol{\theta}}, \boldsymbol{Q} = \boldsymbol{Q}_{\boldsymbol{\eta}}, \boldsymbol{R} = \boldsymbol{R}_{\boldsymbol{\eta}}, \boldsymbol{Y}_{0:M}, \boldsymbol{D}_{0:M} = \boldsymbol{D}^{(\phi)}_{0:M}, \boldsymbol{\Omega}_{0:M} = \boldsymbol{\Omega}^{(\phi)}_{0:M}$)
  $\ell \leftarrow 0$ ▷ Initialization
  $i \leftarrow M - 1$ ▷ Used to map $t_{n(i)}$ to t'_i
  $\boldsymbol{\mu}_{0|0}, \ldots, \boldsymbol{\mu}_{N|N}, \boldsymbol{\Sigma}_{0|0}, \ldots, \boldsymbol{\Sigma}_{N|N} \leftarrow$ `rodeo_forward`$(\boldsymbol{W}, \boldsymbol{f}(\boldsymbol{X}, t), \boldsymbol{v}, \boldsymbol{Q}, \boldsymbol{R})$
  ▷ Lines 6-21 compute $\log p(\boldsymbol{Y}_{0:M} \mid \boldsymbol{Z}_{1:N} = \boldsymbol{0}, \boldsymbol{\Theta})$
  $\boldsymbol{A}_N, \boldsymbol{b}_N, \boldsymbol{C}_N \leftarrow \boldsymbol{0}, \boldsymbol{\mu}_{N|N}, \boldsymbol{\Sigma}_{N|N}$
  for $k = 1 : d$ **do**
    $\boldsymbol{\mu}^{(k)}_N, \boldsymbol{\Sigma}^{(k)}_N \leftarrow$ `kalman_forecast`$(\boldsymbol{\mu}^{(k)}_{N|N}, \boldsymbol{\Sigma}^{(k)}_{N|N}, \boldsymbol{0}, \boldsymbol{D}^{(k)}_M, \boldsymbol{\Omega}^{(k)}_M)$
    $\ell \leftarrow \ell +$ `normal_logpdf`$(\boldsymbol{y}^{(k)}_N; \boldsymbol{\mu}^{(k)}_N, \boldsymbol{\Sigma}^{(k)}_N)$
    $\boldsymbol{\mu}^{(k)\prime}_{N|N}, \boldsymbol{\Sigma}^{(k)\prime}_{N|N} \leftarrow$ `kalman_update`$(\boldsymbol{\mu}^{(k)}_{N|N}, \boldsymbol{\Sigma}^{(k)}_{N|N}, \boldsymbol{Y}^{(k)}_M, \boldsymbol{0}, \boldsymbol{D}^{(k)}_M, \boldsymbol{\Omega}^{(k)}_M)$
  for $n = N - 1 : 0$ **do**
    for $k = 1 : d$ **do**
      $\boldsymbol{A}^{(k)}_n, \boldsymbol{b}^{(k)}_n, \boldsymbol{C}^{(k)}_n \leftarrow$ `kalman_cond`$(\boldsymbol{\mu}^{(k)}_{n|n}, \boldsymbol{\Sigma}^{(k)}_{n|n}, \boldsymbol{\mu}^{(k)}_{n+1|n}, \boldsymbol{\Sigma}^{(k)}_{n+1|n}, \boldsymbol{Q}^{(k)})$
      $\boldsymbol{\mu}^{(k)\prime}_{n|n+1}, \boldsymbol{\Sigma}^{(k)\prime}_{n|n+1} \leftarrow$ `kalman_predict`$(\boldsymbol{\mu}^{(k)\prime}_{n+1|n+1}, \boldsymbol{\Sigma}^{(k)\prime}_{n+1|n+1}, \boldsymbol{b}^{(k)}_n, \boldsymbol{A}^{(k)}_n, \boldsymbol{C}^{(k)})$
      if $t_n = t_{n(i)}$ **then**
        $\boldsymbol{\mu}^{(k)}_n, \boldsymbol{\Sigma}^{(k)}_n \leftarrow$ `kalman_forecast`$(\boldsymbol{\mu}^{(k)\prime}_{n|n+1}, \boldsymbol{\Sigma}^{(k)\prime}_{n|n+1}, \boldsymbol{0}, \boldsymbol{D}^{(k)}_i, \boldsymbol{\Omega}^{(k)}_i)$
        $\ell \leftarrow \ell +$ `normal_logpdf`$(\boldsymbol{Y}^{(k)}_i; \boldsymbol{\mu}^{(k)}_n, \boldsymbol{\Sigma}^{(k)}_n)$
        $\boldsymbol{\mu}^{(k)\prime}_{n|n}, \boldsymbol{\Sigma}^{(k)\prime}_{n|n} \leftarrow$ `kalman_update`$(\boldsymbol{\mu}^{(k)\prime}_{n|n+1}, \boldsymbol{\Sigma}^{(k)\prime}_{n|n+1}, \boldsymbol{y}^{(k)}_n, \boldsymbol{0}, \boldsymbol{D}^{(k)}_i, \boldsymbol{\Omega}^{(k)}_i)$
        $i \leftarrow i - 1$
      else
        $\boldsymbol{\mu}^{(k)\prime}_{n|n}, \boldsymbol{\Sigma}^{(k)\prime}_{n|n} \leftarrow \boldsymbol{\mu}^{(k)\prime}_{n|n+1}, \boldsymbol{\Sigma}^{(k)\prime}_{n|n+1}$
  return ℓ

not use any of the noisy observations $\boldsymbol{Y}_{0:M}$ in the ODE solver. In contrast, the DALTON approximation (Wu and Lysy 2023) is data-adaptive in that it uses the $\boldsymbol{Y}_{0:M}$ to approximate the ODE solution. DALTON provides two methods for computing (13) depending on whether the measurement model (12) is Gaussian or non-Gaussian. The former uses the identity

$$p(\boldsymbol{Y}_{0:M} \mid \boldsymbol{Z}_{1:N} = \boldsymbol{0}, \boldsymbol{\Theta}) = \frac{p(\boldsymbol{Y}_{0:M}, \boldsymbol{Z}_{1:N} = \boldsymbol{0} \mid \boldsymbol{\Theta})}{p(\boldsymbol{Z}_{1:N} = \boldsymbol{0} \mid \boldsymbol{\Theta})}. \tag{17}$$

The denominator $p(\boldsymbol{Z}_{1:N} = \boldsymbol{0} \mid \boldsymbol{\Theta})$ on the right-hand side estimated using a Kalman filter on the data-free surrogate model (11) (e.g., Tronarp *et al.* 2018; Schober *et al.* 2019, and see details in Algorithm 4). The numerator $p(\boldsymbol{Y}_{0:M}, \boldsymbol{Z}_{1:N} = \boldsymbol{0} \mid \boldsymbol{\Theta})$ can be computed by augmenting the surrogate model (11) at the measurement locations $n = n(i)$ via

$$\begin{bmatrix} \boldsymbol{Z}_{n(i)} \\ \boldsymbol{Y}_i \end{bmatrix} \overset{\text{ind}}{\sim} \text{Normal}\left(\left(\begin{bmatrix} \boldsymbol{W}_{\boldsymbol{\theta}} \\ \boldsymbol{D}^{(\phi)}_i \end{bmatrix} + \begin{bmatrix} \boldsymbol{B}_{n(i)} \\ \boldsymbol{0} \end{bmatrix}\right) \boldsymbol{X}_{n(i)} + \begin{bmatrix} \boldsymbol{a}_{n(i)} \\ \boldsymbol{0} \end{bmatrix}, \begin{bmatrix} \boldsymbol{V}_{n(i)} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{\Omega}^{(\phi)}_i \end{bmatrix}\right), \tag{18}$$

where the data-free interrogation coefficients $\boldsymbol{a}_{n(i)}$, $\boldsymbol{B}_{n(i)}$, and $\boldsymbol{V}_{n(i)}$ are computed exactly as in Table 2. Additionally, DALTON can also benefit from blocking by having the same requirements for $\boldsymbol{D}^{(\phi)}_i$ and $\boldsymbol{\Omega}^{(\phi)}_i$ as Fenrir. The exact computations are presented in Algorithm 4. The non-Gaussian case is somewhat more involved and detailed in the Appendix B.

Algorithm 4 DALTON probabilistic ODE likelihood approximation for Gaussian measurements.

procedure `dalton`($\boldsymbol{W} = \boldsymbol{W}_{\boldsymbol{\theta}}, \boldsymbol{f}(\boldsymbol{X}, t) = \boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{X}, t), \boldsymbol{v} = \boldsymbol{v}_{\boldsymbol{\theta}}, \boldsymbol{Q} = \boldsymbol{Q}_{\boldsymbol{\eta}}, \boldsymbol{R} = \boldsymbol{R}_{\boldsymbol{\eta}}, \boldsymbol{Y}_{0:M} = \boldsymbol{Y}_{0:M}, \boldsymbol{D}_{0:M} = \boldsymbol{D}^{(\phi)}_{0:M}, \boldsymbol{\Omega}_{0:M} = \boldsymbol{\Omega}^{(\phi)}_{0:M}$)

- $\boldsymbol{\mu}_{0|0}, \Sigma_{0|0} \leftarrow \boldsymbol{v}, \boldsymbol{0}$ ▷ Initialization
- $\boldsymbol{Z}_{1:N} \leftarrow \boldsymbol{0}$
- $\ell_z, \ell_{yz} \leftarrow 0, 0$
- $i \leftarrow 1$ ▷ Used to map t_n to t'_i
- ▷ Lines 7-14 compute $\log p(\boldsymbol{Z}_{1:N} = \boldsymbol{0} \mid \boldsymbol{\Theta})$
- **for** $n = 1 : N$ **do**
  - **for** $k = 1 : d$ **do**
    - $\boldsymbol{\mu}^{(k)}_{n|n-1}, \boldsymbol{\Sigma}^{(k)}_{n|n-1} \leftarrow$ `kalman_predict`$(\boldsymbol{\mu}^{(k)}_{n-1|n-1}, \boldsymbol{\Sigma}^{(k)}_{n-1|n-1}, \boldsymbol{0}, \boldsymbol{Q}^{(k)}, \boldsymbol{R}^{(k)})$
  - $\boldsymbol{a}_n, \boldsymbol{B}_n, \boldsymbol{V}_n \leftarrow$ `interrogate`$(\boldsymbol{\mu}_{n|n-1}, \boldsymbol{\Sigma}_{n|n-1}, \boldsymbol{W}, \boldsymbol{f}(\boldsymbol{X}, t_n))$
  - **for** $k = 1 : d$ **do**
    - $\boldsymbol{\mu}^{(k)}_n, \boldsymbol{\Sigma}^{(k)}_n \leftarrow$ `kalman_forecast`$(\boldsymbol{\mu}^{(k)}_{n|n-1}, \boldsymbol{\Sigma}^{(k)}_{n|n-1}, \boldsymbol{a}^{(k)}_n, \boldsymbol{W}^{(k)} + \boldsymbol{B}^{(k)}_n, \boldsymbol{V}^{(k)}_n)$
    - $\ell_z \leftarrow \ell_z +$ `normal_logpdf`$(\boldsymbol{Z}^{(k)}_n; \boldsymbol{\mu}^{(k)}_n, \boldsymbol{\Sigma}^{(k)}_n)$
    - $\boldsymbol{\mu}^{(k)}_{n|n}, \boldsymbol{\Sigma}^{(k)}_{n|n} \leftarrow$ `kalman_update`$(\boldsymbol{\mu}^{(k)}_{n|n-1}, \boldsymbol{\Sigma}^{(k)}_{n|n-1}, \boldsymbol{Z}^{(k)}_n, \boldsymbol{a}^{(k)}_n, \boldsymbol{W}^{(k)} + \boldsymbol{B}^{(k)}_n, \boldsymbol{V}_n)$
- ▷ Lines 16-30 compute $\log p(\boldsymbol{Y}_{0:M}, \boldsymbol{Z}_{1:N} = \boldsymbol{0} \mid \boldsymbol{\Theta})$
- **for** $k = 1 : d$ **do**
  - $\ell_{yz} \leftarrow$ `normal_logpdf`$(\boldsymbol{Y}^{(k)}_0; \boldsymbol{D}^{(k)}_0 \boldsymbol{v}^{(k)}, \boldsymbol{\Omega}^{(k)}_0)$
- **for** $n = 1 : N$ **do**
  - **for** $k = 1 : d$ **do**
    - $\boldsymbol{\mu}^{(k)}_{n|n-1}, \boldsymbol{\Sigma}^{(k)}_{n|n-1} \leftarrow$ `kalman_predict`$(\boldsymbol{\mu}^{(k)}_{n-1|n-1}, \boldsymbol{\Sigma}^{(k)}_{n-1|n-1}, \boldsymbol{0}, \boldsymbol{Q}^{(k)}, \boldsymbol{R}^{(k)})$
  - $\boldsymbol{a}_n, \boldsymbol{B}_n, \boldsymbol{V}_n \leftarrow$ `interrogate`$(\boldsymbol{\mu}_{n|n-1}, \boldsymbol{\Sigma}_{n|n-1}, \boldsymbol{W}, \boldsymbol{f}(\boldsymbol{X}, t_n))$
  - **if** $t_n = t_{n(i)}$ **then**
    - **for** $k = 1 : d$ **do**
      - $\boldsymbol{Z}^{(k)}_n \leftarrow \begin{bmatrix} \boldsymbol{Z}^{(k)}_n \\ \boldsymbol{Y}^{(k)}_i \end{bmatrix}, \qquad \boldsymbol{W}^{(k)} \leftarrow \begin{bmatrix} \boldsymbol{W}^{(k)} \\ \boldsymbol{D}^{(k)}_i \end{bmatrix}, \qquad \boldsymbol{B}^{(k)}_n \leftarrow \begin{bmatrix} \boldsymbol{B}^{(k)}_n \\ \boldsymbol{0} \end{bmatrix}$
      - $\boldsymbol{a}^{(k)}_n \leftarrow \begin{bmatrix} \boldsymbol{a}^{(k)}_n \\ \boldsymbol{0} \end{bmatrix}, \qquad \boldsymbol{V}^{(k)}_n \leftarrow \begin{bmatrix} \boldsymbol{V}^{(k)}_n & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{\Omega}^{(k)}_i \end{bmatrix}$
      - $i \leftarrow i + 1$
  - **for** $k = 1 : d$ **do**
    - $\boldsymbol{\mu}^{(k)}_n, \boldsymbol{\Sigma}^{(k)}_n \leftarrow$ `kalman_forecast`$(\boldsymbol{\mu}^{(k)}_{n|n-1}, \boldsymbol{\Sigma}^{(k)}_{n|n-1}, \boldsymbol{a}^{(k)}_n, \boldsymbol{W}^{(k)} + \boldsymbol{B}^{(k)}_n, \boldsymbol{V}^{(k)}_n)$
    - $\ell_{yz} \leftarrow \ell_{yz} +$ `normal_logpdf`$(\boldsymbol{Z}^{(k)}_n; \boldsymbol{\mu}^{(k)}_n, \boldsymbol{\Sigma}^{(k)}_n)$
    - $\boldsymbol{\mu}^{(k)}_{n|n}, \boldsymbol{\Sigma}^{(k)}_{n|n} \leftarrow$ `kalman_update`$(\boldsymbol{\mu}^{(k)}_{n|n-1}, \boldsymbol{\Sigma}^{(k)}_{n|n-1}, \boldsymbol{Z}^{(k)}_n, \boldsymbol{a}^{(k)}_n, \boldsymbol{W}^{(k)} + \boldsymbol{B}^{(k)}_n, \boldsymbol{V}^{(k)}_n)$
- **return** $\ell_{yz} - \ell_z$ ▷ Estimate of $\log p(\boldsymbol{Y}_{0:M} \mid \boldsymbol{Z}_{1:N} = \boldsymbol{0}, \boldsymbol{\Theta})$

3.4. Marginal MCMC

In principle, the Fenrir and DALTON algorithms can be applied with any of the model interrogations in Table 2. However, the interrogation method of Chkrebtii *et al.* (2016) produces the $\boldsymbol{a}_n$ stochastically, such that a direct application of Fenrir would not include the

additional uncertainty resulting from this step. To address this, Chkrebtii *et al.* (2016) adopt a Bayesian approach, whereby for a given parameter prior $\pi(\boldsymbol{\Theta})$ and arbitrary measurement model (12), they present a "marginal MCMC" algorithm for

$$\begin{aligned}\hat{p}(\boldsymbol{\Theta}, \boldsymbol{X}_{0:N} \mid \boldsymbol{Y}_{0:M}, \boldsymbol{Z}_{1:N} = \boldsymbol{0}) \propto \pi(\boldsymbol{\Theta}) \times p(\boldsymbol{Y}_{0:M} \mid \boldsymbol{X}_{0:N}, \boldsymbol{\phi}) \\ \times \int \hat{p}_{\text{lin}}(\boldsymbol{X}_{0:N}, \boldsymbol{\mathcal{I}}_{0:N} \mid \boldsymbol{Z}_{0:N} = \boldsymbol{0}, \boldsymbol{\theta}, \boldsymbol{\eta}) \, \mathrm{d}\boldsymbol{\mathcal{I}}_{0:N},\end{aligned} \tag{19}$$

where $\hat{p}_{\text{lin}}(\boldsymbol{X}_{0:N}, \boldsymbol{\mathcal{I}}_{0:N} \mid \boldsymbol{Z}_{0:N} = \boldsymbol{0}, \boldsymbol{\theta}, \boldsymbol{\eta})$ corresponds to the surrogate model (11) with possibly stochastic model interrogation terms $\boldsymbol{\mathcal{I}}_{0:N} = (\boldsymbol{a}_{0:N}, \boldsymbol{B}_{0:N}, \boldsymbol{V}_{0:N})$. The relevant Metropolis-Hastings update is described in Algorithm 5. The term "marginal" refers to the fact that $\hat{p}_{\text{lin}}(\boldsymbol{X}_{0:N}, \boldsymbol{\mathcal{I}}_{0:N} \mid \boldsymbol{Z}_{0:N} = \boldsymbol{0}, \boldsymbol{\theta}, \boldsymbol{\eta})$ appears both in the target posterior density and the Markov transition kernel, thus dropping out of the Metropolis-Hasting acceptance rate completely. For the Gaussian measurement model (15) and deterministic interrogation terms $\boldsymbol{\mathcal{I}}_{0:N}$, the marginal MCMC algorithm targets the posterior distribution of the Fenrir likelihood combined with the given prior $\pi(\boldsymbol{\Theta})$.

Algorithm 5 Marginal MCMC algorithm of Chkrebtii *et al.* (2016).

procedure `marginal_mcmc_step`($\boldsymbol{\Theta}^{(\text{curr})}, \boldsymbol{X}_{0:N}^{(\text{curr})}, \boldsymbol{Y}_{0:M}$)
  $\boldsymbol{\Theta}^{(\text{prop})} \sim q(\boldsymbol{\Theta} \mid \boldsymbol{\Theta}^{(\text{curr})})$
  $\boldsymbol{X}_{0:N}^{(\text{prop})} \leftarrow$ `rodeo_ode_sim`$(\boldsymbol{W}_{\boldsymbol{\theta}^{(\text{prop})}}, f_{\boldsymbol{\theta}^{(\text{prop})}}(\boldsymbol{X}, t), \boldsymbol{v}_{\boldsymbol{\theta}^{(\text{prop})}}, \boldsymbol{Q}_{\boldsymbol{\eta}^{(\text{prop})}}, \boldsymbol{R}_{\boldsymbol{\eta}^{(\text{prop})}})$
  $\rho = \dfrac{p(\boldsymbol{Y}_{0:M} \mid \boldsymbol{X}_{n(0:M)}^{(\text{prop})}, \boldsymbol{\eta}^{(\text{prop})}) \times \pi(\boldsymbol{\Theta}^{(\text{prop})}) / q(\boldsymbol{\Theta}^{(\text{prop})} \mid \boldsymbol{\Theta}^{(\text{curr})})}{p(\boldsymbol{Y}_{0:M} \mid \boldsymbol{X}_{n(0:M)}^{(\text{curr})}, \boldsymbol{\eta}^{(\text{curr})}) \times \pi(\boldsymbol{\Theta}^{(\text{curr})}) / q(\boldsymbol{\Theta}^{(\text{curr})} \mid \boldsymbol{\Theta}^{(\text{prop})})}$
  $v \sim \text{Uniform}(0, 1)$
  if $v < \rho$ **then**
    $\boldsymbol{\Theta}^{(\text{curr})}, \boldsymbol{X}_{0:N}^{(\text{curr})}, \rho^{(\text{curr})} \leftarrow \boldsymbol{\Theta}^{(\text{prop})}, \boldsymbol{X}_{0:N}^{(\text{prop})}, \rho^{(\text{prop})}$
  return $\boldsymbol{\Theta}^{(\text{curr})}, \boldsymbol{X}_{0:N}^{(\text{curr})}, \rho^{(\text{curr})}$

4. Implementation

rodeo is designed to serve two types of users. The first is statisticians and data scientists wishing to use probabilistic solvers to calibrate ODEs to empirical data. The second is researchers in probabilistic numerics interested in developing new probabilistic solvers within the general **rodeo** framework. Examples include adaptive step-size selection (Chkrebtii and Campbell 2019), initialization via deterministic solvers (Schober *et al.* 2019), uncertainty calibration (Conrad *et al.* 2017; Tronarp *et al.* 2018), and optimal tuning of the solution prior (Chkrebtii *et al.* 2016).

To best serve these audiences, **rodeo** is written in Python using the **JAX** library for scientific computing. **JAX** is a near drop-in replacement for **NumPy**, making use of **rodeo** straighforward for most Python users. In addition to programming convenience, **JAX** offers state-of-the-art just-in-time (JIT) compilation and automatic differentiation capabilities, of which the former offers a substantial performance increase over plain **NumPy** code, and the latter enables the use of high-performance gradient-based parameter learning algorithms (as

shown in Section 5). In particular, parameter inference using the **rodeo** likelihood approximations outlined in Section 3 can be conducted using powerful **JAX** libraries for optimization (**JAXopt** (Blondel *et al.* 2021), **Optax** (Babuschkin *et al.* 2020)) and MCMC (**BlackJAX** (Lao and Louf 2020)).

There are three main components to the **rodeo** library. The first is a general-purpose Kalman filter and smoother. The main Kalman algorithms are:

- `predict()`: Given $\boldsymbol{\mu}_{n-1|n-1}$ and $\boldsymbol{\Sigma}_{n-1|n-1}$, calculate $\boldsymbol{\mu}_{n|n-1}$ and $\boldsymbol{\Sigma}_{n|n-1}$.
- `update()`: Given $\boldsymbol{\mu}_{n|n-1}$ and $\boldsymbol{\Sigma}_{n|n-1}$, calculate $\boldsymbol{\mu}_{n|n}$ and $\boldsymbol{\Sigma}_{n|n}$.
- `smooth()`: Given $\boldsymbol{\mu}_{n+1|N}$ and $\boldsymbol{\Sigma}_{n+1|N}$, calculate $\boldsymbol{\mu}_{n|N}$ and $\boldsymbol{\Sigma}_{n|N}$.

These algorithms and a few others – e.g., sampling from and evaluating $p(\boldsymbol{X}_{0:N} \mid \boldsymbol{Z}_{0:N})$ – are detailed in Appendix A.

The second primary component of **rodeo** is the ODE solver itself. This consists of two functions, `solve_mv()` and `solve_sim()`, which return the mean and variance $(\boldsymbol{\mu}_{0:N|N}, \boldsymbol{\Sigma}_{0:N|N})$ and a random draw $\boldsymbol{X}_{0:N} \sim p(\boldsymbol{X}_{0:N} \mid \boldsymbol{Z}_{0:N})$, respectively, for the surrogate model (11). The common parameters to these functions are:

- `key`: A **JAX** pseudo-RNG key. This is needed for any **JAX** calculation which requires random samples.
- `ode_weight`, `ode_init`, `ode_fun`, `params`: The ODE-IVP model components $\boldsymbol{W}$, $\boldsymbol{v}$, $\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{X}, t)$ and $\boldsymbol{\theta}$. The ODE function has argument signature `function(X, t, **params)`, such that the parameters of the model may be passed via arbitrary key-value pairs. These same key-value pairs must then be supplied to `solve_sim()` and `solve_mv()`.
- `t_min`, `t_max`, `n_steps`: The time interval $(t_{\min}, t_{\max})$ over which the ODE solution is sought, and the number of solver discretization points N.
- `prior_weight`, `prior_var`: The weight and variance matrices $\boldsymbol{Q}$ and $\boldsymbol{R}$ of the solution prior as discretized to time intervals of size $\Delta t = (t_{\max} - t_{\min})/N$.
- `interrogate`: An interrogation function, i.e., a function with arguments `key`, $\boldsymbol{\mu}_{n|n-1}$, $\boldsymbol{\Sigma}_{n|n-1}$, $\boldsymbol{W}$, $\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{X}, t)$, and $\boldsymbol{\theta}$ which returns the interrogation variables $\boldsymbol{a}_n$, $\boldsymbol{B}_n$, and $\boldsymbol{V}_n$ used in the surrogate model (11). Note that the `key` argument is used here to implement stochastic interrogation methods such as that of Chkrebtii *et al.* (2016) in Table 2.

The third main component of **rodeo** consists of implementations of the Basic, Fenrir, DALTON, and Marginal MCMC inference algorithms described in Section 3. These implementations share many of the same arguments as `solve_sim()` and `solve_mv()`. Additional arguments are:

- `obs_data`, `obs_times`: The observed data $\boldsymbol{Y}_{0:M}$ and corresponding observation times $t'_{0:M}$. For simplicity these are rounded to the nearest values of the solution grid $t_{0:N}$.
- `obs_weight`, `obs_var`: For the Fenrir and Gaussian DALTON algorithms, the weight and variance matrices $\boldsymbol{D}^{(\phi)}_{1:M}$ and $\boldsymbol{\Omega}^{(\phi)}_{1:M}$ in the multivariate normal observation model (15).

- `obs_loglik`: For the Basic and Marginal MCMC algorithms, the log of the observation likelihood $p(\boldsymbol{Y}_{0:M} \mid \boldsymbol{X}_{n(0:M)}, \boldsymbol{\phi})$ in (12). This is a function with argument signature `function(obs_data, ode_data, **params)`, corresponding to $\boldsymbol{Y}_{0:M}$, $\boldsymbol{X}_{n(0:M)}$, and $\boldsymbol{\phi}$, respectively.

A few other algorithm-specific arguments will be discussed in the relevant examples in Section 5.

In addition, **rodeo** provides a few helper functions to set up the prior. The two most important ones are `ibm_init()` and `indep_init()`. `ibm_init()` creates the weight and variance matrices `prior_weight` and `prior_var` for the solution prior $\boldsymbol{X}(t)$, for given time interval Δt and number of required continuous derivatives per ODE variable, $p_1, \ldots, p_d$.

By default, we set all p_k, $k = 1, \ldots, d$ equal to their maximum value and return **JAX** arrays of size $d \times p \times p$, where $p = \max_{1 \leq k \leq d} p_k$. It is suggested that the ODE-IVP components `ode_weight`, `ode_init`, and `ode_fun` be zero-padded accordingly (see Section 5), and similarly stored in block format, i.e., with system variables as the leading dimension of **JAX** arrays. This typically offers a performance boost despite padding when there are lots of variables with a similar number of derivatives.

Zero-padding can be disabled in `solve_mv()` and `solve_sim()` by passing the ODE-IVP components as a single block. To disable zero-padding for the IBM prior, we can compute $\boldsymbol{Q}^{(k)}$ and $\boldsymbol{R}^{(k)}$ separately for each $k = 1, \ldots, d$, then combine them into densely-stored block-diagonal matrices using `indep_init()`.

5. Examples

The following examples illustrate the use of **rodeo** for solving and learning the parameters of various ODE-IVPs on $\boldsymbol{x}(t) = (x_1(t), \ldots, x_d(t))$. All examples are implemented with variable blocking as described above. That is, if the ODE-IVP involves the first $p_k - 1$ derivatives of $x_k(t)$, then $x_k(t)$ is given an IBM prior with $p = \max_{1 \leq k \leq d} p_k$ continuous derivatives, i.e., $q_k = p + 1$, and the ODE-IVP is padded accordingly. In the context of parameter learning, it will be convenient to introduce a problem-specific parameter transformation $\boldsymbol{\Theta} = \boldsymbol{\mathcal{G}}(\boldsymbol{\theta}, \boldsymbol{\phi}, \boldsymbol{\eta})$ such that the support of $\boldsymbol{\Theta}$ is all of $\mathbb{R}^{\dim\{\boldsymbol{\theta}\}+\dim\{\boldsymbol{\phi}\}+\dim\{\boldsymbol{\eta}\}}$.

5.1. A Second-Order Univariate ODE

Consider the second-order univariate ODE-IVP given by

$$x^{(2)}(t) = \sin(2t) - x(t), \qquad (x(0), x^{(1)}(0)) = (-1, 0), \qquad t \in [0, 10], \tag{20}$$

employed by Chkrebtii *et al.* (2016) to illustrate the benefits of using stochastic solvers. We recast (20) in the form of (8) and go through the inputs required for **rodeo**. Since (20) involves derivatives up to second order, we use a three-times integrated IBM prior on $\boldsymbol{X}(t) = (x^{(0)}(t), \ldots, x^{(3)})$. Since this problem is univariate, we have $d = 1$ blocks for which we use an extra dimension to store the blocks. Thus, using **NumPy**/**JAX** array notation, (20) is written in the form [**Check NumPy notation. Should be three dimensional for W.**] of (8) as

$$\begin{aligned} \boldsymbol{X}(t) &= \left[\left[x^{(0)}(t) \quad x^{(1)}(t) \quad x^{(2)}(t) \quad x^{(3)}(t)\right]\right], & \boldsymbol{f}(\boldsymbol{X}(t), t) &= \left[\left[\sin(2t) - x^{(0)}(t)\right]\right], \\ \boldsymbol{W} &= \left[\left[0 \quad 0 \quad 1 \quad 0\right]\right], & \boldsymbol{v} = \boldsymbol{X}(0) &= \left[\left[-1 \quad 0 \quad 1 \quad 0\right]\right]. \end{aligned} \tag{21}$$

These variables correspond to `ode_fun`, `ode_weight`, and `ode_init` respectively. The time interval in (20) determine the function arguments `t_min = 0` and `t_max = 10`. We use `n_steps` to represent the number of solver discretization points. We want `n_steps` to be small so that the run time is fast but large enough to give an accurate approximation. A suitable number satisfying these conditions is `n_steps = 80`. The effect of increasing `n_steps` will be examined shortly. We use the IBM process as our Gaussian prior which require step size Δt and scale parameter σ. The former is determined by `t_min`, `t_max`, and `n_steps` to be $\Delta t = 10/80$. For the latter, our experiments indicate a remarkable insensitivity to the choice of σ in the Gaussian prior over several orders of magnitude, thus for simplicity we set $\sigma = 0.1$. We use `ibm_init` to compute `prior_weight` and `prior_var` for the IBM prior. Finally, we set `interrogate` to use the Chkrebtii *et al.* (2016) interrogation method for this example. The code below provides a tutorial for solving this problem.

Listing 1: **rodeo** solver for the higher order ODE (1).

```
# --- 0. Import libraries and module -------------------------------------


import jax
import jax.numpy as jnp
import rodeo
import rodeo.prior
import rodeo.interrogate

# --- 1. Define the ODE-IVP ----------------------------------------------


def higher_fun(x, t, **params):
    """
    Higher-order ODE of Chkrebtii et al in **rodeo** format.
    Args:
        x: JAX array of shape `(1,4)` corrresponding to
           `X = (x, x^(1), x^(2), x^(3))`.
        t: Scalar time variable.

    Returns:
        JAX array of shape `(1,1)` corresponding to `f(x,t)`.

    """
    return jnp.array([[jnp.sin(2 * t) - x[0, 0]]])


W = jnp.array([[[0., 0., 1., 0.]]])  # LHS matrix of ODE
x0 = jnp.array([[-1., 0., 1., 0.]])  # initial value for the IVP

# Time interval on which a solution is sought.
t_min = 0.
t_max = 10.

# --- 3.  Define the prior process ---------------------------------------

n_vars = 1                          # number of variables in the ODE
n_deriv = 4  # max number of derivatives
sigma = jnp.array([.1] * n_vars)  # IBM process scale factor


# --- 4.  Evaluate the ODE solution --------------------------------------

n_steps = 80                  # number of evaluations steps
dt = (t_max - t_min) / n_steps  # step size

# generate the Kalman parameters corresponding to the prior
```

```
prior_Q, prior_R = rodeo.prior.ibm_init(
    dt=dt,
    n_deriv=n_deriv,
    sigma=sigma
)

key = jax.random.PRNGKey(0)  # JAX pseudo-RNG key


# deterministic ODE solver: posterior mean
Xt, _ = rodeo.solve_mv(
    key=key,
    # define ode
    ode_fun=higher_fun,
    ode_weight=W,
    ode_init=x0,
    t_min=t_min,
    t_max=t_max,
    # solver parameters
    n_steps=n_steps,
    interrogate=rodeo.interrogate.interrogate_chkrebtii,
    prior_weight=prior_Q,
    prior_var=prior_R
)

# probabilistic ODE solver: draw from posterior
Xt = rodeo.solve_sim(
    key=key,
    # define ode
    ode_fun=higher_fun,
    ode_weight=W,
    ode_init=x0,
    t_min=t_min,
    t_max=t_max,
    # solver parameters
    n_steps=n_steps,
    interrogate=rodeo.interrogate.interrogate_chkrebtii,
    prior_weight=prior_Q,
    prior_var=prior_R
)
```

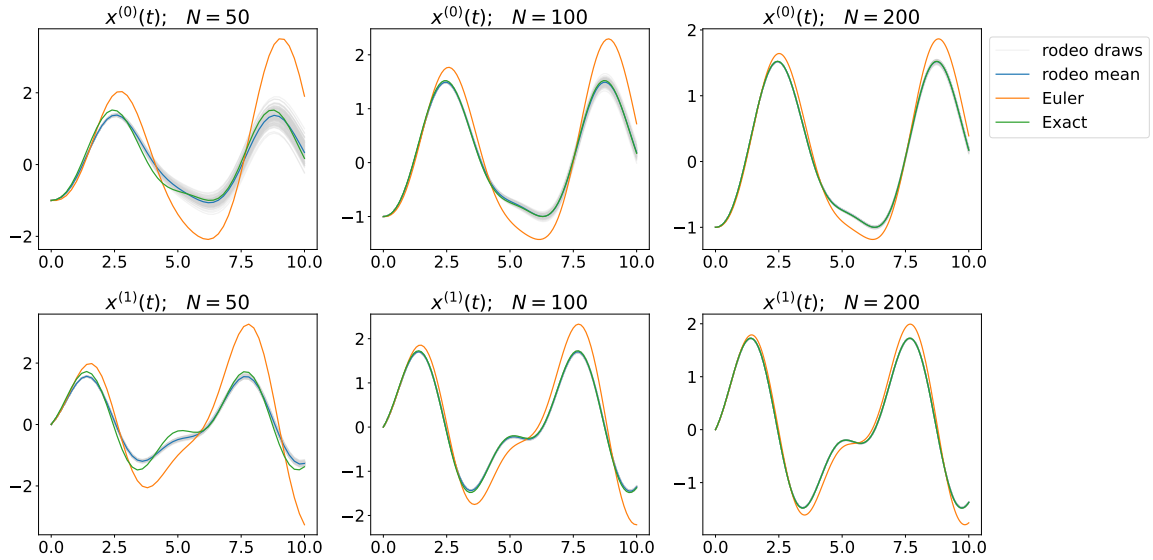

Figure 1: **rodeo**, Euler and the exact solution at $N = 50, 100, 200$ evaluation points for $t \in [0, 10]$. 100 draws and the analytical posterior mean were produced from **rodeo**.

For this problem, a closed-form solution is given by

$$x^{(0)}(t) = \tfrac{1}{3}\big(2\sin(t) - 3\cos(t) - \sin(2t)\big). \tag{22}$$

In Figure 1, we use the exact solution to benchmark **rodeo** at different numbers of evaluation points $N = 50, 100, 200$. For each N, 100 independent draws were simulated from `solve_sim()`, and the posterior mean was computed using `solve_mv()`. Figure 1 also displays the ODE solution as approximated by a simple Euler scheme:

$$\begin{aligned} x^{(1)}(t + \Delta t) &= x^{(1)}(t) + \big(\sin(2t) - x^{(0)}(t)\big)\Delta t \\ x^{(0)}(t + \Delta t) &= x^{(0)}(t) + x^{(1)}(t)\Delta t. \end{aligned} \tag{23}$$

As N increases, **rodeo** approaches the true solution. **rodeo** is more accurate than Euler for any number of evaluation points, most apparently at $N = 50$. Figure 1 also indicates that **rodeo** is able to capture the uncertainty in the solution posterior at low N, where the solver is less accurate.

5.2. Parameter Inference via Laplace Approximation

The FitzHugh-Nagumo (FN) model (Sherwood 2013) is a two-state ODE on $\boldsymbol{x}(t) = \big(V(t), R(t)\big)$, in which $V(t)$ describes the evolution of the neuronal membrane voltage and $R(t)$ describes the activation and deactivation of neuronal channels. The FN ODE is given by

$$\begin{aligned} V^{(1)}(t) &= c\Big(V(t) - \frac{V(t)^3}{3} + R(t)\Big), \\ R^{(1)}(t) &= -\frac{V(t) - a + bR(t)}{c}. \end{aligned} \tag{24}$$

In the following examples, we use **rodeo** to learn the unknown parameters $\boldsymbol{\theta} = (a, b, c, V(0), R(0))$ with $a, b, c > 0$ from data $\boldsymbol{Y}_{0:M} = (\boldsymbol{Y}_0, \ldots, \boldsymbol{Y}_M)$ following the measurement model

$$\boldsymbol{Y}_i \overset{\text{ind}}{\sim} \text{Normal}(\boldsymbol{x}(t_i), \phi^2 \cdot \boldsymbol{I}_{2\times 2}). \tag{25}$$

We begin by simulating the data with $M = 40$, $t_i = i$, $\phi = 0.2$, and true parameters $\boldsymbol{\theta} = (0.2, 0.2, 3, -1, 1)$. This is accomplished using a very small step size Δt with the interrogation method of Krämer *et al.* (2021) in Table 2, which we found to achieve the highest accuracy as a function of Δt. The simulated ODE trajectory and noisy observations are displayed in Figure 2.

```
# --- 0. Import libraries and modules -----------------------------------


import jax
import jax.numpy as jnp
import jaxopt
import blackjax
import rodeo
import rodeo.prior
import rodeo.interrogate
import rodeo.inference

# --- 1. Define the ODE-IVP ---------------------------------------------


def fitz_fun(X, t, **params):
    "FitzHugh-Nagumo ODE in rodeo format."
    a, b, c = params["theta"]
    V, R = X[:, 0]
    return jnp.array(
        [[c * (V - V * V * V / 3 + R)],
         [-1 / c * (V - a + b * R)]]
    )


def fitz_init(x0, theta):
    "FitzHugh-Nagumo initial values in rodeo format."
    x0 = x0[:, None]
    return jnp.hstack([
        x0,
        fitz_fun(X=x0, t=0., theta=theta),
        jnp.zeros_like(x0)
    ])


W = jnp.array([[[0., 1., 0.]], [[0., 1., 0.]]])  # LHS matrix of ODE
x0 = jnp.array([-1., 1.])  # initial value for the ODE-IVP
theta = jnp.array([.2, .2, 3])  # ODE parameters
X0 = fitz_init(x0, theta)  # initial value in rodeo format

# Time interval on which a solution is sought.
t_min = 0.
t_max = 40.

# --- Define the prior process ------------------------------------------

n_vars = 2
n_deriv = 3

# IBM process scale factor
sigma = jnp.array([.1] * n_vars)
```

```
# --- data simulation -----------------------------------------------------


dt_obs = 1.  # interobservation time
n_steps_obs = int((t_max - t_min) / dt_obs)
# observation times
obs_times = jnp.linspace(t_min, t_max,
                         num=n_steps_obs + 1)

# number of simulation steps per observation
n_res = 100
n_steps = n_steps_obs * n_res
# simulation times
sim_times = jnp.linspace(t_min, t_max,
                         num=n_steps + 1)

# prior parameters
dt_sim = (t_max - t_min) / n_steps  # grid size for simulation
prior_Q, prior_R = rodeo.prior.ibm_init(
    dt=dt_sim,
    n_deriv=n_deriv,
    sigma=sigma
)

# Produce a Pseudo-RNG key
key = jax.random.PRNGKey(0)


# calculate ODE via deterministic output
key, subkey = jax.random.split(key)
Xt, _ = rodeo.solve_mv(
    key=subkey,
    # define ode
    ode_fun=fitz_fun,
    ode_weight=W,
    ode_init=X0,
    t_min=t_min,
    t_max=t_max,
    theta=theta,  # ODE parameters added here
    # solver parameters
    n_steps=n_steps,
    interrogate=rodeo.interrogate.interrogate_kramer,
    prior_weight=prior_Q,
    prior_var=prior_R
)

# generate observations
noise_sd = 0.2  # Standard deviation in noise model
key, subkey = jax.random.split(key)
eps = jax.random.normal(
    key=subkey,
    shape=(obs_times.size, 2)
)
# 0th order derivatives at observed timepoints
obs_ind = jnp.searchsorted(sim_times, obs_times)
x = Xt[obs_ind, :, 0]
Y = x + noise_sd * eps
```

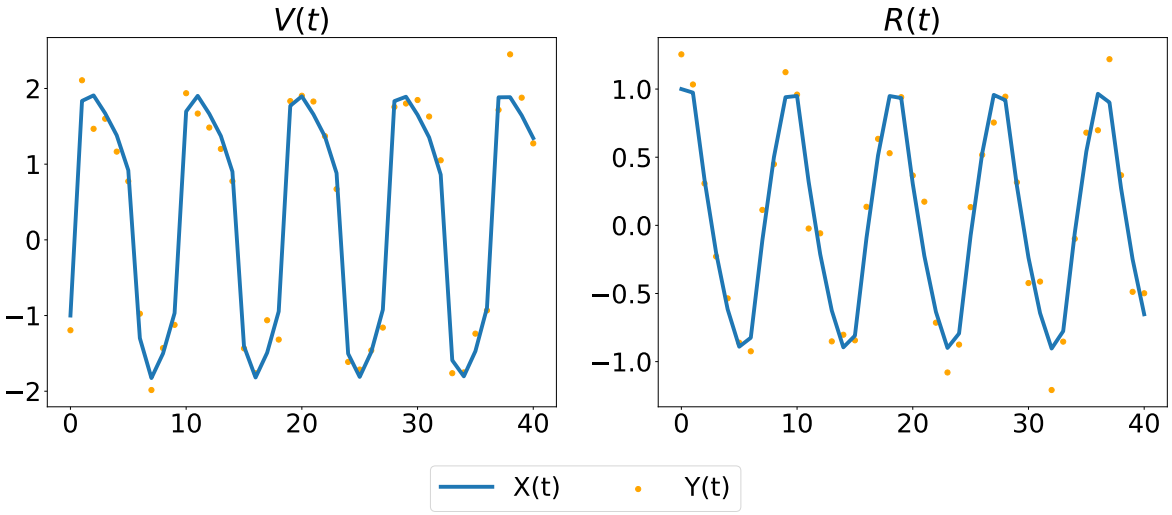

Figure 2: Simulated ODE and noisy observations of the FN model.

The Laplace Approximation

We now turn to the problem of parameter estimation. In the Bayesian context, this is achieved by postulating a prior distribution $p(\boldsymbol{\Theta})$ on $\boldsymbol{\Theta} = \boldsymbol{\mathcal{G}}(\boldsymbol{\theta}, \boldsymbol{\phi}, \boldsymbol{\eta})$, and then combining it with the stochastic solver's likelihood function $\mathcal{L}(\boldsymbol{\Theta} \mid \boldsymbol{Y}_{0:M}) \propto p(\boldsymbol{Y}_{0:M} \mid \boldsymbol{Z}_{1:N} = \boldsymbol{0}, \boldsymbol{\Theta})$ of (13) to obtain the posterior distribution

$$p(\boldsymbol{\Theta} \mid \boldsymbol{Y}_{0:M}) \propto \pi(\boldsymbol{\Theta}) \times \mathcal{L}(\boldsymbol{\Theta} \mid \boldsymbol{Y}_{0:M}). \tag{26}$$

For the Basic, Fenrir, and DALTON algorithms described in Section 3, the high-dimensional latent ODE variables $\boldsymbol{X}_{0:N}$ can be approximately integrated out to produce a closed-form likelihood approximation $\hat{\mathcal{L}}(\boldsymbol{\Theta} \mid \boldsymbol{Y}_{0:M})$ to form the corresponding posterior approximation $\hat{p}(\boldsymbol{\Theta} \mid \boldsymbol{Y}_{0:M})$. While this posterior can be readily sampled from using MCMC techniques (as we shall do momentarily) Bayesian parameter estimation can also be achieved by way of a Laplace approximation (e.g., Gelman *et al.* 2013). Namely, $\hat{p}(\boldsymbol{\Theta} \mid \boldsymbol{Y}_{0:M})$ is approximated by a multivariate normal distribution,

$$\boldsymbol{\Theta} \mid \boldsymbol{Y}_{0:M} \approx \text{Normal}(\hat{\boldsymbol{\Theta}}, \hat{\boldsymbol{V}}), \tag{27}$$

where

$$\hat{\boldsymbol{\Theta}} = \arg\max_{\boldsymbol{\Theta}} \log \hat{p}(\boldsymbol{\Theta} \mid \boldsymbol{Y}_{0:M}), \quad \hat{\boldsymbol{V}} = -\left[\frac{\partial^2}{\partial \boldsymbol{\Theta} \partial \boldsymbol{\Theta}'} \log \hat{p}(\hat{\boldsymbol{\Theta}} \mid \boldsymbol{Y}_{0:M})\right]^{-1}. \tag{28}$$

One can then obtain posteriors on the original scale $(\boldsymbol{\theta}, \boldsymbol{\phi}, \boldsymbol{\eta})$ by first sampling $\boldsymbol{\Theta}^{(1)}, \ldots, \boldsymbol{\Theta}^{(B)}$ from the normal distribution (27), then computing $(\boldsymbol{\theta}^{(b)}, \boldsymbol{\phi}^{(b)}, \boldsymbol{\eta}^{(b)}) = \boldsymbol{\mathcal{G}}^{-1}(\boldsymbol{\Theta}^{(b)})$.

The Laplace approximation is a popular tool for Bayesian machine learning applications (e.g., MacKay 1992; Gianniotis 2019), being typically much faster than MCMC (as shown below). Our Python implementation of **rodeo** uses the **JAX** library (Bradbury *et al.* 2018) for automatic differentiation and just-in-time (JIT) compilation. This produces very fast implementations of $\log \hat{p}(\boldsymbol{\Theta} \mid \boldsymbol{Y}_{0:M})$, along with its gradient and Hessian.

The code below shows how to implement the Laplace approximation – also called the Bayesian normal approximation – for estimating the parameter posterior $p(\boldsymbol{\theta} \mid \boldsymbol{Y}_{0:M})$ of the FN model using the Basic ODE likelihood approximation described in Section 3.1. We take the measurement model parameter ϕ to be known, such that the learning problem is only for the ODE model parameters $\boldsymbol{\theta} = (a, b, c, V(0), R(0))$ and the Gaussian Markov process tuning parameters $\boldsymbol{\eta} = (\sigma_1, \sigma_2)$. The parameter transformation is

$$\boldsymbol{\Theta} = \big(\boldsymbol{\Psi}_{\boldsymbol{\theta}} = (\log a, \log b, \log c, V(0), R(0)), \boldsymbol{\Psi}_{\boldsymbol{\eta}} = (\log \sigma_1, \log \sigma_2)\big). \tag{29}$$

The prior $\pi(\boldsymbol{\Theta})$ consists of independent Normal$(0, 10^2)$ distributions on each element of $\boldsymbol{\Psi}_{\boldsymbol{\theta}}$ and a flat prior on the tuning parameters $\pi(\boldsymbol{\eta}) \propto 1$.

The posterior mode $(\hat{\boldsymbol{\Psi}}_{\boldsymbol{\theta}}, \hat{\boldsymbol{\Psi}}_{\boldsymbol{\eta}})$ is obtained using the Newton-CG optimization algorithm (Nocedal and Wright 2006) as implemented in the **JAXopt** library (Blondel *et al.* 2021). As for the corresponding estimate of the posterior variance of $\boldsymbol{\Psi}_{\boldsymbol{\theta}}$, we use only the Hessian of $\log \hat{p}(\boldsymbol{\Psi}_{\boldsymbol{\theta}}, \boldsymbol{\Psi}_{\boldsymbol{\eta}} \mid \boldsymbol{Y}_{0:M})$ in terms of the model parameters $\boldsymbol{\Psi}_{\boldsymbol{\theta}}$. This is due to the insensitivity of the stochastic solver to the values of $\boldsymbol{\eta}$ beyond a rough order of magnitude, which had the effect of unduly inflating the posterior variance estimate of $\boldsymbol{\Psi}_{\boldsymbol{\theta}}$ when the Hessian was taken with respect to both $\boldsymbol{\Psi}_{\boldsymbol{\theta}}$ and $\boldsymbol{\Psi}_{\boldsymbol{\eta}}$. This approach has been previously advocated in Tronarp *et al.* (2022).

Remark. For the flat prior $\pi(\boldsymbol{\Theta}) \propto 1$, the posterior mean and variance estimates $\hat{\boldsymbol{\Theta}}$ and $\hat{\boldsymbol{V}}$ in (28) correspond to the maximum likelihood estimate (MLE) and inverse of the observed Fisher information for the approximate likelihood $\hat{\mathcal{L}}(\boldsymbol{\Theta} \mid \boldsymbol{Y}_{0:M})$. The code below can thus be used for Frequentist inference as well, e.g., with the standard errors of $\hat{\boldsymbol{\Theta}}$ given by the square root of the diagonal elements of $\hat{\boldsymbol{V}}$.

```
# --- parameter inference: basic + laplace ------------------------------

def fitz_logprior(upars):
    "Logprior on unconstrained model parameters."
    n_theta = 5  # number of ODE + IV parameters
    lpi = jax.scipy.stats.norm.logpdf(
        x=upars[:n_theta],
        loc=0.,
        scale=10.
    )
    return jnp.sum(lpi)


def fitz_loglik(obs_data, ode_data, **params):
    """
    Loglikelihood for measurement model.

    Args:
        obs_data (ndarray(n_obs, n_vars)): Observations data.
        ode_data (ndarray(n_obs, n_vars, n_deriv)): ODE solution.
    """
    ll = jax.scipy.stats.norm.logpdf(
        x=obs_data,
        loc=ode_data[:, :, 0],
        scale=noise_sd
    )
    return jnp.sum(ll)


def constrain_pars(upars, dt):
```

```
    """
    Convert unconstrained optimization parameters into rodeo inputs.

    Args:
        upars : Parameters vector on unconstrainted scale.
        dt : Discretization grid size.

    Returns:
        tuple with elements:
        - theta : ODE parameters.
        - X0 : Initial values in rodeo format.
        - Q, R : Prior matrices.
    """
    theta = jnp.exp(upars[:3])
    x0 = upars[3:5]
    X0 = fitz_init(x0, theta)
    sigma = upars[5:]
    Q, R = rodeo.prior.ibm_init(
        dt=dt,
        n_deriv=n_deriv,
        sigma=sigma
    )
    return theta, X0, Q, R


def fitz_laplace(key, neglogpost, n_samples, upars_init):
    """
    Sample from the Laplace approximation to the parameter posterior for the FN model.

    Args:
        key : PRNG key.
        neglogpost: Function specifying the negative log-posterior distribution
                    in terms of the unconstrained parameter vector ``upars``.
        upars_init: Initial value to the optimization algorithm over ``neglogpost()``.
        n_samples : Number of posterior samples to draw.

    Returns:
        JAX array of shape ``(n_samples, 5)`` of posterior samples from ``(theta, x0)``.
    """
    n_theta = 5  # number of ODE + IV parameters
    # find mode of neglogpost()
    solver = jaxopt.ScipyMinimize(
        fun=neglogpost,
        method="Newton-CG",
        jit=True
    )
    opt_res = solver.run(upars_init)
    upars_mean = opt_res.params
    # variance estimate
    upars_fisher = jax.jacfwd(jax.jacrev(neglogpost))(upars_mean)
    # unconstrained ode+iv parameter variance estimate
    uode_var = jax.scipy.linalg.inv(upars_fisher[:n_theta, :n_theta])
    uode_mean = upars_mean[:n_theta]
    # sample from Laplace approximation
    uode_sample = jax.random.multivariate_normal(
        key=key,
        mean=uode_mean,
        cov=uode_var,
        shape=(n_samples,)
    )
    # convert back to original scale
    ode_sample = uode_sample.at[:, :3].set(jnp.exp(uode_sample[:, :3]))
    return ode_sample


def neglogpost_basic(upars):
```

```
    "Negative logposterior for basic approximation."
    # solve ODE
    theta, X0, prior_Q, prior_R = constrain_pars(upars, dt_sim)
    # basic loglikelihood
    ll = rodeo.inference.basic(
        key=key,  # immaterial, since not used
        # ode specification
        ode_fun=fitz_fun,
        ode_weight=W,
        ode_init=X0,
        t_min=t_min,
        t_max=t_max,
        theta=theta,
        # solver parameters
        n_steps=n_steps,
        interrogate=rodeo.interrogate.interrogate_kramer,
        prior_weight=prior_Q,
        prior_var=prior_R,
        # observations
        obs_data=Y,
        obs_times=obs_times,
        obs_loglik=fitz_loglik
    )
    return -(ll + fitz_logprior(upars))
```

5.3. MCMC with BlackJAX

For the Basic, Fenrir, and DALTON methods, we can also use MCMC to sample from the closed-form posterior approximation $\hat{p}(\boldsymbol{\Theta} \mid \boldsymbol{Y}_{0:M})$, when the Laplace approximation is deemed to be insufficiently accurate. This can be easily and efficiently accomplished using the **BlackJAX** package(Lao and Louf 2020). Like **rodeo**, **BlackJAX** is written with **JAX** and offers many built-in MCMC algorithms to choose from. Here we use the Hybrid Monte Carlo (HMC) algorithm (Duane *et al.* 1987), using **JAX** to conveniently compute the required gradient of $\log \hat{p}(\boldsymbol{\Theta} \mid \boldsymbol{Y}_{0:M})$ via automatic differentiation.

Continuing with the example of the FN model introduced in Section 5.2, the code below shows the basic usage of **BlackJAX** with HMC to sample from the parameter posterior $p(\boldsymbol{\theta} \mid \boldsymbol{Y}_{0:M})$ using the Basic ODE likelihood approximation. We set use five HMC leapfrog steps and tune the remaining HMC parameters (step size and mass matrix) via the window adaption algorithm described in Stan Development Team (2023).

```
# --- parameter inference: basic + hmc ----------------------------------


# blackjax uses the loglikelihood function instead of the negative
def logpost_basic(upars): return -neglogpost_basic(upars)


def fitz_hmc(key, logpost, n_samples, upars_init):
    """
    Sample from the parameter posterior via the Hamiltonian Monte Carlo
    method for the FN model.

    Args:
        key : PRNG key.
        logpost : Function specifying the log-posterior distribution
                  n terms of the unconstrained parameter vector ``upars``.
        upars_init : Initial value to the optimization algorithm over ``logpost()``.
        n_samples : Number of posterior samples to draw.

    Returns:
```

```
        JAX array of shape ``(n_samples, 5)`` of posterior samples from ``(theta, x0)``.
    """
    key, *subkeys = jax.random.split(key, num=3)
    # number of times HMC run the symplectic integrator to build the trajectory
    num_steps = 5
    # Stan window adaptation algorithm to start with reasonable parameters
    warmup = blackjax.window_adaptation(
        blackjax.hmc, logpost, num_integration_steps=num_steps)
    (initial_state, parameters), _ = warmup.run(subkeys[0], upars_init)
    kernel = blackjax.hmc(logpost, **parameters).step

    # blackjax recommended api: inference loop for sampling
    def inference_loop(key, kernel, initial_state, n_samples):

        def one_step(state, rng_key):
            state, _ = kernel(rng_key, state)
            return state, state

        keys = jax.random.split(key, n_samples)
        _, states = jax.lax.scan(one_step, initial_state, keys)
        return states

    uode_sample = inference_loop(
        subkeys[1], kernel, initial_state, n_samples).position[:, :5]
    # convert back to original scale
    ode_sample = uode_sample.at[:, :3].set(jnp.exp(uode_sample[:, :3]))
    return ode_sample
```

5.4. Efficient Blocking for Gaussian Likelihoods

In the examples above, the Gaussian measurement model (25) was implemented by hand in the function `fitz_loglik()`. In contrast, the Fenrir and Gaussian DALTON algorithms are specifically designed for Gaussian noise models. As discussed in Sections 3.2 and 3.3, these algorithms can benefit from blocking when the parameters $\boldsymbol{D}_i^{(\phi)}$ and $\boldsymbol{\Omega}_i^{(\phi)}$ of the noise model model (15) are block diagonal. For the FN example, we have $d = 2$ ODE variables, $p = 2$ derivatives in the IBM prior, and $s = 1$ measurements per ODE variable, such that in **NumPy**/**JAX** array notation, the terms of (15) corresponding to the measurement model (25) are given by

$$\begin{aligned} \boldsymbol{Y}_i &= \left[\left[Y_i^{(V)}\right] \quad \left[Y_i^{(R)}\right]\right], \\ \boldsymbol{D}_i^{(\phi)} &= \left[\left[\left[1 \quad 0 \quad 0\right]\right], \left[\left[1 \quad 0 \quad 0\right]\right]\right], \\ \boldsymbol{\Omega}_i^{(\phi)} &= \left[\left[\left[\phi^2\right]\right], \left[\left[\phi^2\right]\right]\right]. \end{aligned} \tag{30}$$

The code below shows how to construct the ODE likelihood approximation for the FN model using Fenrir as described in Section 3.2, which could then be used represent the corresponding posterior $\hat{p}(\boldsymbol{\Theta} \mid \boldsymbol{Y}_{0:M})$ via the Laplace approximation or via MCMC, as shown in the examples above.

```
# --- parameter inference: fenrir + laplace ----------------------------------


# gaussian measurement model specification in blocked form
n_meas = 1  # number of measurements per variable in obs_data_i
obs_data = jnp.expand_dims(Y, axis=-1)
obs_weight = jnp.zeros((len(obs_data), n_vars, n_meas, n_deriv))
obs_weight = obs_weight.at[:].set(jnp.array([[[1., 0., 0.]], [[1., 0., 0.]]]))
obs_var = jnp.zeros((len(obs_data), n_vars, n_meas, n_meas))
obs_var = obs_var.at[:].set(noise_sd**2 * jnp.array([[[1.]], [[1.]]]))
```

```
def neglogpost_fenrir(upars):
    "Negative logposterior for fenrir approximation."
    theta, X0, prior_Q, prior_R = constrain_pars(upars, dt_sim)
    # fenrir loglikelihood
    ll = rodeo.inference.fenrir(
        key=key,  # immaterial, since not used
        # ode specification
        ode_fun=fitz_fun,
        ode_weight=W,
        ode_init=X0,
        t_min=t_min,
        t_max=t_max,
        theta=theta,
        # solver
        n_steps=n_steps,
        interrogate=rodeo.interrogate.interrogate_kramer,
        prior_weight=prior_Q,
        prior_var=prior_R,
        # gaussian measurement model
        obs_data=obs_data,
        obs_times=obs_times,
        obs_weight=obs_weight,
        obs_var=obs_var
    )
    return -(ll + fitz_logprior(upars))
```

5.5. Marginal MCMC

Unlike other parameter inference algorithms, the marginal MCMC method described in Algorithm 5 is implemented using an object-oriented programming paradigm to simplify the usage of its many interrelated components. Thus, our `MarginalMCMC` base class expects the user to define the following methods:

- `obs_loglik`: The log of the observation likelihood $p(\boldsymbol{Y}_{0:M} \mid \boldsymbol{X}_{n(0:M)}, \boldsymbol{\phi})$ as described in Section 4.
- `logprior`:
- `parse_pars`:

[**Previous version**] Our last parameter inference method is the Chkrebtii algorithm of Chkrebtii *et al.* (2016). Unlike the other methods, we adopt an object-orientated programming paradigm which allow for better organization of related methods and variables. We design a `MarginalMCMC` base class which implements Algorithm 5 that users could inherit from when implementing their own applications. In addition to the log prior, `logprior`, and the observation loglikelihood, `obs_loglik`, Chkbretii also requires the method, `parse_pars`, which parses $\boldsymbol{\Theta} = (\boldsymbol{\theta}, \boldsymbol{\phi}, \boldsymbol{\eta})$ in a specific order as described in the code below. The parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ are to be stored in key-value pairs which can be used by `solve_sim` and `obs_loglik` while $\boldsymbol{\eta}$ is used to compute the Markov prior process described in Section 2.2.

The function `fitz_chmcmc` shows how to approximate the parameter posterior $p(\boldsymbol{\theta} \mid \boldsymbol{Y}_{0:M})$ of the FN model using the Chkrebtii method described in Section 3.4. We start by initializing `fitz_mcmc` class we defined for the FN model. Next, `fitz_ch.initialize` is `CHKREBTII_ODE`$(\boldsymbol{\theta}, \boldsymbol{\eta})$ in Algorithm 5 which returns a sample $\boldsymbol{X}_{0:N} \sim p_{\text{lin}}(\boldsymbol{X}_{0:N} \mid \boldsymbol{Z}_{1:N} =$

$\mathbf{0}, \boldsymbol{\theta}, \boldsymbol{\eta}$). We define the parameter $\boldsymbol{\gamma}$ (tuned during the burning stage to obtain an overall acceptance rate of 25%) to be used in the proposal distribution. We designed `MarginalMCMC` to have similar API as **BlackJAX**, namely, the method `MarginalMCMC.step` mimics `kernel` in Section 5.3. In other words, `MarginalMCMC.step` is one iteration of the for-loop in Algorithm 5.

```
# --- parameter inference: chkrebtii --------------------------------------

class fitz_mcmc(rodeo.inference.MarginalMCMC):

    def logprior(self, upars):
        return fitz_logprior(upars)

    def obs_loglik(self, obs_data, ode_data, **params):
        return fitz_loglik(obs_data, ode_data, **params)

    def parse_pars(self, upars, dt):
        r"""
        Parse the parameters from ``upars`` such that the output contains
        a tuple which contains the elements:

            1. The weight matrix defining the ODE; W.
            2. The initial value to the IVP (i.e., ``ode_init``).
            3. The Gaussian Prior parameters (i.e., ``prior_weight``, ``prior_var``).
            4. A dictionary containing the other parameters
            (i.e., model parameter theta, measurement parameter phi).

        The order should be ``(ode_init, prior_weight, prior_var, dictionary)``.
        """
        theta, X0, prior_Q, prior_R, = constrain_pars(upars, dt)
        params = {
            "theta": theta
        }
        return W, X0, prior_Q, prior_R, params


def fitz_chmcmc(key, n_samples, upars_init):
    r"""
    Sample via the Chkrebtii algorithm.

    Args:
        key : PRNG key.
        n_samples : Number of samples to return.
        upars_init : Initial parameter to be optimized.


    Return:
        JAX array of shape ``(n_samples, 5)`` of posterior
        samples from ``(theta, x0)``.
    """
    key, *subkeys = jax.random.split(key, num=3)
    # initial the fitz_mcmc class
    fitz_ch = fitz_mcmc(
        ode_fun=fitz_fun,
        obs_data=Y,
        obs_times=obs_times,
        t_min=t_min,
        t_max=t_max,
        n_steps=n_steps)

    # compute initial state for initial parameters
    initial_state = fitz_ch.initialize(subkeys[0], upars_init)

    # proposal parameter
    scale = jnp.array(
```

```
        [0.01, 0.1, 0.01, 0.01, 0.01, 0.01, 0.01])

    # inference loop similar to blackjax api
    def inference_loop(key, initial_state, n_samples):

        def one_step(state, key):
            state, sample = fitz_ch.step(key, state, scale=scale)
            return state, sample

        keys = jax.jax.random.split(key, n_samples)
        _, samples = jax.lax.scan(one_step, initial_state, keys)
        return samples

    uode_sample = inference_loop(
        subkeys[1], initial_state, n_samples)["Theta"]
    # convert back to original scale
    ode_sample = uode_sample.at[:, :3].set(jnp.exp(uode_sample[:, :3]))
    return ode_sample
```

Figure 3 displays the parameter posteriors for the Laplace approximation and MCMC at different solver step sizes $\Delta t = T/N$ for the methods discussed. Also included for comparison are the Laplace posteriors for the basic approximation method using an Euler solver and a highly accurate deterministic solver for the ODE-IVP (10); namely, the Runge-Kutta 8 solver with Dormand-Prince step size adaptation (Dormand and Prince 1980) as implemented in the **diffrax** library (Kidger 2021). We shall assume that the output of Runge-Kutta is the true ODE solution and therefore its parameter posteriors as the ground truth. For parameter c, the Euler posteriors do not get close to the true posterior before $\Delta t = 0.02$. In contrast, the Basic approximation and Fenrir posteriors are very close to the true posterior as of $\Delta t = 0.05$. The Chkrebtii marginal MCMC method covers the true parameter values at $\Delta t \leq 0.05$ whereas the **BlackJAX** MCMC with the Basic algorithm does so at all step sizes.

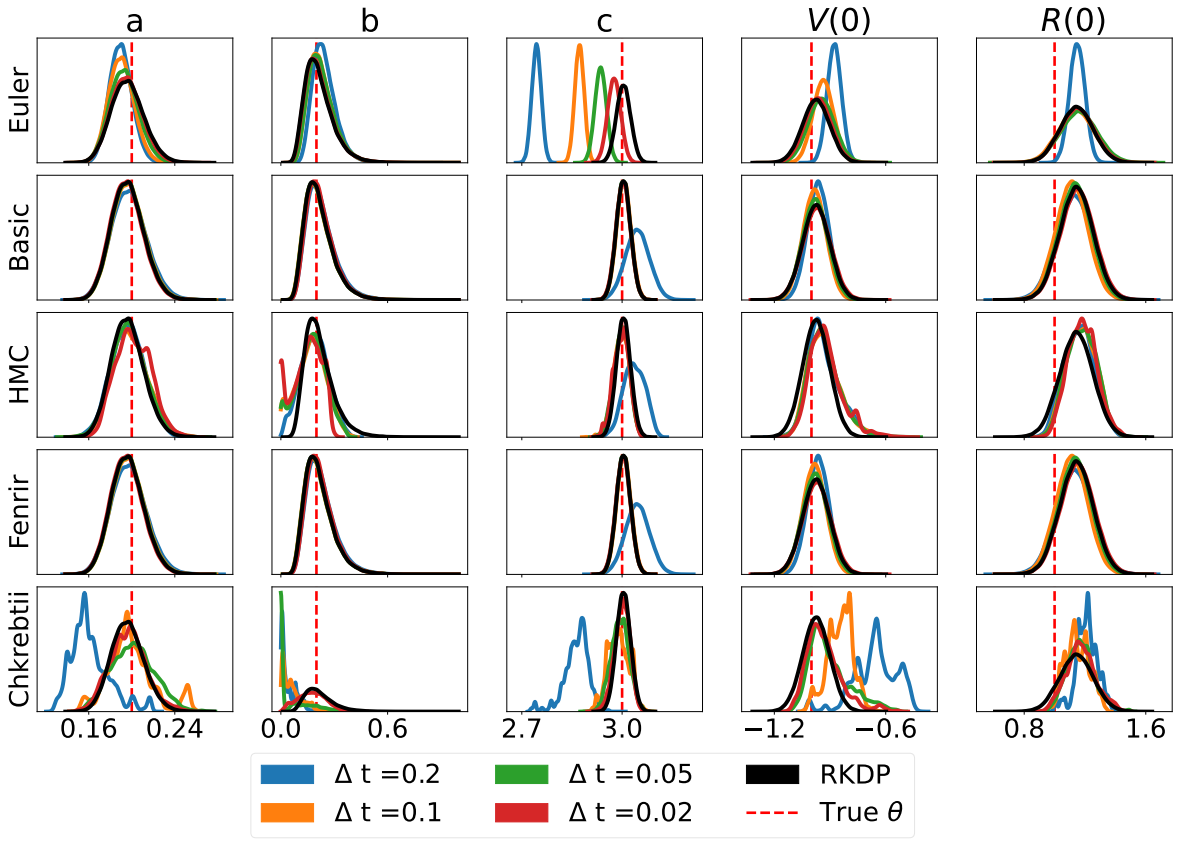

Figure 3: Parameter posteriors for the FN model using the Laplace approximation for the Euler ODE solver, the Basic ODE likelihood approximation, and Fenrir. Also included are the Chkrebtii marginal MCMC method and **BlackJAX** HMC with the Basic likelihood approximation.

5.6. An ODE System with Unobserved Components

The Hes1 model (Yang *et al.* 2021) is a three-variable dynamic system $\boldsymbol{x}(t) = (P(t), M(t), H(t))$ that describes the oscillation of the Hes1-protein (P) and Hes1-mRNA (M) with an Hes1-interacting (H) factor acting as a stabilizer. The log-scale Hes1 model is

$$\begin{aligned} \frac{\mathrm{d}\log P(t)}{\mathrm{d}t} &= -aH(t) + bM(t)/P(t) - c, \\ \frac{\mathrm{d}\log M(t)}{\mathrm{d}t} &= -d + \frac{e}{1 + P(t)^2 M(t)}, \\ \frac{\mathrm{d}\log H(t)}{\mathrm{d}t} &= -aP(t) + \frac{f}{1 + P(t)^2 H(t)} - g. \end{aligned} \tag{31}$$

It contains 10 parameters $\boldsymbol{\theta} = (a, b, c, d, e, f, g, P(0), M(0), H(0))$ with $\boldsymbol{\theta} > \mathbf{0}$. In this case, the measurements consist of noise observations of only $\log P(t)$ and $\log M(t)$, with $\log H(t)$ completely unobserved. Moreover, $\log P(t)$ and $\log M(t)$ measured at different times; each

are measured at 15-minute intervals for a four-hour period, but $\log P(t)$ measurements start at time $t = 0$ whereas $\log M(t)$ measurements start at time $t = 7.5$ minutes. Following Yang *et al.* (2021), the measurement error model is

$$Y_i^{(K)} \overset{\text{ind}}{\sim} \text{Normal}(\log K(t_i^{(K)}), \phi^2), \qquad K \in \{P, M\}, \tag{32}$$

where $\phi = 0.15$. Data is simulated in the same way as Section **??** for Y_i with true parameter values $\boldsymbol{\theta} = (0.022, 0.3, 0.031, 0.028, 0.5, 20, 0.3, 1.439, 2.037, 17.904)$. In this experiment, we compare the Laplace approximation using the Basic and Gaussian DALTON algorithms. Inference for the Basic approximation was carried out largely the same way as described in Section 5.2.

Efficient Blocking with Unobserved Components

We define $\boldsymbol{Y}_i, \boldsymbol{D}_i^{(\phi)}$ and $\boldsymbol{\Omega}_i^{(\phi)}$ in a similar manner to Section 5.4. The caveat here is that at each observation time point t_i' we have $\log H(t_i')$ unobserved and either $\log P(t_i')$ or $\log M(t_i')$ unobserved. For each unobserved component K, the observation $Y_i^{(K)}$ will be zero and the matrices $\boldsymbol{D}_i^{(\phi)}$ and $\boldsymbol{\Omega}_i^{(\phi)}$ will have zeros in the row corresponding to the unobserved component. This ensures Algorithm 3 and 4 adds zero whenever there is an unobserved component (i.e., the log density of Normal$(0; 0, 0)$). The code below shows how to construct the ODE likelihood approximation for the Hes1 model using DALTON as described in Section 3.3.

```
# --- 0. Import libraries and modules -------------------------------------


import jax
import jax.numpy as jnp
import rodeo
import rodeo.prior
import rodeo.interrogate
import rodeo.inference

# --- 1. Define the ODE-IVP -----------------------------------------------


def hes1_fun(X, t, **params):
    "Hes1 on the log-scale in rodeo format."
    P, M, H = jnp.exp(X[:, 0])
    a, b, c, d, e, f, g = params["theta"]
    logP = -a * H + b * M / P - c
    logM = -d + e / (1 + P * P) / M
    logH = -a * P + f / (1 + P * P) / H - g
    return jnp.array([[logP], [logM], [logH]])


def hes1_init(x0, theta):
    "Hes1 initial values in rodeo format."
    x0 = x0[:, None]
    return jnp.hstack([
        x0,
        hes1_fun(X=x0, t=0., theta=theta),
        jnp.zeros_like(x0)
    ])


# initial value for the ODE-IVP on log-scale
x0 = jnp.log(jnp.array([1.439, 2.037, 17.904]))
W = jnp.array([[[0., 1., 0.]], [[0., 1., 0.]], [
              [0., 1., 0.]]])  # LHS matrix of ODE
```

```
theta = jnp.array([0.022, 0.3, 0.031, 0.028, 0.5, 20, 0.3])  # ODE parameters

# Time interval on which a solution is sought.
t_min = 0.
t_max = 240.

# --- Define the prior process ------------------------------------------

n_vars = 3
n_deriv = 3

# IBM process scale factor
sigma = jnp.array([.1] * n_vars)

# simulation times
dt_sim = 0.75
n_steps = int((t_max - t_min) / dt_sim)


# --- parameter inference: dalton ---------------------------------


def hes1_logprior(upars):
    "Logprior on unconstrained model parameters."
    n_theta = 10  # number of ODE + IV parameters
    lpi = jax.scipy.stats.norm.logpdf(
        x=upars[:n_theta],
        loc=0.,
        scale=10.
    )
    return jnp.sum(lpi)


def constrain_pars(upars, dt):
    """
    Convert unconstrained optimization parameters into rodeo inputs.

    Args:
        upars: Parameters vector on unconstrainted scale.
        dt: Discretization grid size.

    Returns:
        tuple with elements:
        - theta : ODE parameters.
        - X0 : Initial values in rodeo format.
        - Q, R : Prior matrices.
    """
    theta = jnp.exp(upars[:7])
    x0 = upars[7:10]
    X0 = hes1_init(x0, theta)
    sigma = upars[10:]
    Q, R = rodeo.prior.ibm_init(
        dt=dt,
        n_deriv=n_deriv,
        sigma=sigma
    )
    return theta, X0, Q, R


# Produce a Pseudo-RNG key
key = jax.random.PRNGKey(0)

# gaussian measurement model specification in blocked form
obs_data = jnp.load("hes1data.npy")  # load the data
noise_sd = 0.15  # Standard deviation in noise model
n_steps_obs = obs_data.shape[0]
```

```
# observation times
obs_times = jnp.linspace(t_min, t_max,
                         num=n_steps_obs)
n_meas = 1  # number of measurements per variable in Y_i
obs_weight = jnp.zeros((n_steps_obs, n_vars, n_meas, n_deriv))
obs_weight = obs_weight.at[::2, 0].set(jnp.array([[[1., 0., 0.]]]))
obs_weight = obs_weight.at[1::2, 1].set(jnp.array([[[1., 0., 0.]]]))
obs_var = jnp.zeros((n_steps_obs, n_vars, n_meas, n_meas))
obs_var = obs_var.at[::2, 0].set(noise_sd**2 * jnp.array([[[1.]]]))
obs_var = obs_var.at[1::2, 1].set(noise_sd**2 * jnp.array([[[1.]]]))


def neglogpost_dalton(upars):
    "Negative logposterior for dalton approximation."
    theta, X0, prior_Q, prior_R = constrain_pars(upars, dt_sim)
    # Gaussian DALTON loglikelihood
    ll = rodeo.inference.dalton(
        key=key,  # immaterial, since not used
        # ode specification
        ode_fun=hes1_fun,
        ode_weight=W,
        ode_init=X0,
        t_min=t_min,
        t_max=t_max,
        theta=theta,
        # solver
        n_steps=n_steps,
        interrogate=rodeo.interrogate.interrogate_kramer,
        prior_weight=prior_Q,
        prior_var=prior_R,
        # gaussian measurement model
        obs_data=obs_data,
        obs_times=obs_times,
        obs_weight=obs_weight,
        obs_var=obs_var

    )
    return -(ll + hes1_logprior(upars))
```

Figure 4 displays the parameter posteriors of the Laplace approximation using the Basic and Gaussian DALTON algorithms. We were unable to obtain parameter estimates for solver step size $\Delta t > 2.0$ minutes due to numerical instability. However, the Basic and DALTON posteriors for $\Delta t \leq 0.75$ minutes are almost indistinguishable from the true posterior.

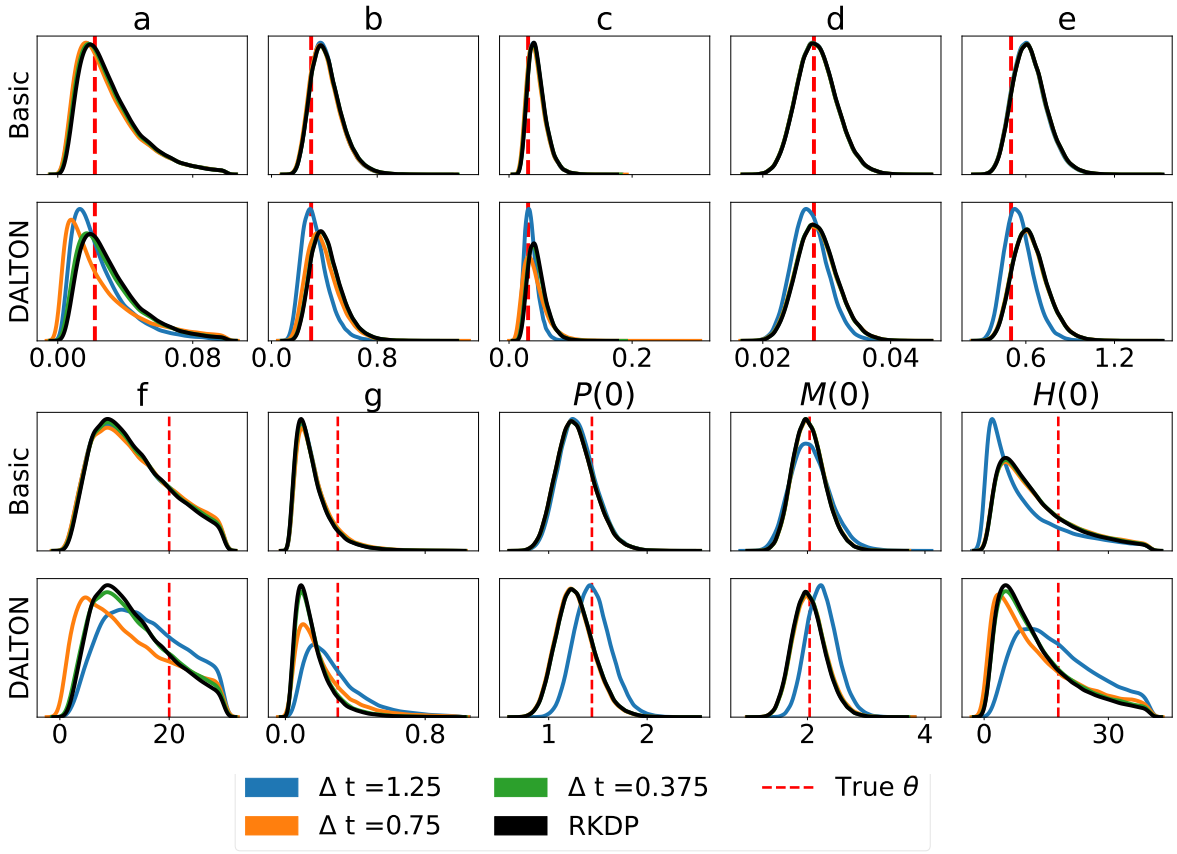

Figure 4: The parameter posteriors of the Basic and DALTON approximations for the Hes1 model.

5.7. Non-Gaussian DALTON

The SEIRAH model is a six-compartment epidemiological model used describe Covid-19 dynamics by Prague *et al.* (2020). The six compartments of a given population are: S susceptible, E latent, I ascertained infectious, R removed (both recovered and deceased), A unascertained infectious, and H hospitalized. The evolution of these quantities according to the SEIRAH model is given by

$$\begin{aligned}
\frac{\mathrm{d}S(t)}{\mathrm{d}t} &= -\frac{bS(t)(I(t)+\alpha A(t))}{N}, & \frac{\mathrm{d}E(t)}{\mathrm{d}t} &= \frac{bS(t)(I(t)+\alpha A(t))}{N} - \frac{E(t)}{D_e}, \\
\frac{\mathrm{d}I(t)}{\mathrm{d}t} &= \frac{rE(t)}{D_e} - \frac{I(t)}{D_q} - \frac{I(t)}{D_I}, & \frac{\mathrm{d}R(t)}{\mathrm{d}t} &= \frac{I(t)+A(t)}{D_I} + \frac{H_t}{D_h}, \\
\frac{\mathrm{d}A(t)}{\mathrm{d}t} &= \frac{(1-r)E(t)}{D_e} - \frac{A(t)}{D_I}, & \frac{\mathrm{d}H(t)}{\mathrm{d}t} &= \frac{I(t)}{D_q} - \frac{H(t)}{D_h},
\end{aligned} \tag{33}$$

where $N = S(t) + E(t) + I(t) + R(t) + A(t) + H(t)$ is fixed and so is $D_h = 30$ (Prague *et al.* 2020). Again we only have (indirect) observations of two of the six compartments. That is,

let

$$\tilde{I}(t) = \frac{rE(t)}{D_e}, \quad \tilde{H}(t) = \frac{I(t)}{D_q} \tag{34}$$

denote the new cases entering their respective compartments. The measurement model of Prague *et al.* (2020) is

$$Y_i^{(\tilde{K})} \overset{\text{ind}}{\sim} \text{Poisson}(\tilde{K}(t_i)), \qquad K \in \{I, H\}, \tag{35}$$

where $t_i = i$ days with $i = 0, \ldots, 60$. The unknown parameters are $\boldsymbol{\theta} = (b, r, \alpha, D_e, D_I, D_q, E(0), I(0))$ with $\boldsymbol{\theta} > 0$. Data was simulated with true parameter values $\boldsymbol{\theta} = (2.23, 0.034, 0.55, 5.1, 2.3, 1.13, 15492, 21752)$ and remaining initial values $S_0^{(0)} = 63884630$, $R_0^{(0)} = 0$, $A_0^{(0)} = 618013$, and $H_0^{(0)} = 13388$ (as reported by Prague *et al.* (2020)). Again, inference for the Basic approximation was carried out largely the same way as described in Section 5.2.

The code below shows how to construct the ODE likelihood approximation for the SEIRAH model using non-Gaussian DALTON in Algorithm 7 as described in Appendix B. The unobserved components in (35) is directly encoded in `obs_loglik_i`.

```
# --- 0. Import libraries and modules -------------------------------------


import jax
import jax.numpy as jnp
import rodeo
import rodeo.prior
import rodeo.interrogate
import rodeo.inference

# --- 1. Define the ODE-IVP -----------------------------------------------


def seirah_fun(X, t, **params):
    "SEIRAH ODE in rodeo format."
    S, E, I, R, A, H = X[:, 0]
    N = S + E + I + R + A + H
    b, r, alpha, D_e, D_I, D_q = params["theta"]
    D_h = 30
    dS = -b * S * (I + alpha * A) / N
    dE = b * S * (I + alpha * A) / N - E / D_e
    dI = r * E / D_e - I / D_q - I / D_I
    dR = (I + A) / D_I + H / D_h
    dA = (1 - r) * E / D_e - A / D_I
    dH = I / D_q - H / D_h
    return jnp.array([[dS], [dE], [dI], [dR], [dA], [dH]])


def seirah_init(x0EI, theta):
    "SEIRAH initial values in rodeo format."
    x0 = jnp.array([63884630., None, None, 0., 618013., 13388.])
    # only two of the initial values are part of the inference
    x0 = x0.at[1:3].set(x0EI)[:, None]
    return jnp.hstack([
        x0,
        seirah_fun(X=x0, t=0., theta=theta),
        jnp.zeros_like(x0)
    ])


n_vars = 6
W = jnp.repeat(jnp.array([[[0, 1, 0]]]), repeats=n_vars, axis=0)
theta = jnp.array([2.23, 0.034, 0.55, 5.1, 2.3, 1.13])  # ODE parameters
```

```
# Time interval on which a solution is sought.
t_min = 0.
t_max = 60.

# --- Define the prior process ------------------------------------------

n_deriv = 3

# IBM process scale factor
sigma = jnp.array([.1] * n_vars)

# simulation times
dt_sim = 0.02
n_steps = int((t_max - t_min) / dt_sim)

# --- parameter inference: daltonng -------------------------


def seirah_logprior(upars):
    "Logprior on unconstrained model parameters."
    n_theta = 8  # number of ODE + IV parameters
    lpi = jax.scipy.stats.norm.logpdf(
        x=upars[:n_theta],
        loc=0.,
        scale=10.
    )
    return jnp.sum(lpi)


def constrain_pars(upars, dt):
    """
    Convert unconstrained optimization parameters into rodeo inputs.

    Args:
        upars : Parameters vector on unconstrainted scale.
        dt : Discretization grid size.

    Returns:
        tuple with elements:
        - theta : ODE parameters.
        - X0 : Initial values in rodeo format.
        - Q, R : Prior matrices.
    """
    theta = jnp.exp(upars[:6])
    x0 = jnp.exp(upars[6:8])
    X0 = seirah_init(x0, theta)
    sigma = upars[8:]
    Q, R = rodeo.prior.ibm_init(
        dt=dt,
        n_deriv=n_deriv,
        sigma=sigma
    )
    return theta, X0, Q, R


# Produce a Pseudo-RNG key
key = jax.random.PRNGKey(0)

# non-gaussian measurement model
obs_data = jnp.load("seirahdata.npy")  # load the data
n_steps_obs = obs_data.shape[0]
# observation times
obs_times = jnp.linspace(t_min, t_max,
                         num=n_steps_obs)
```

```
def obs_loglik_i(obs_data_i, ode_data_i, ind, **params):
    """
    Likelihood function for the SEIRAH model in non-Gaussian DALTON format.

    Args:
        obs_data_i : Observations at index ``ind``.
        ode_data_i :  ODE solution at index ``ind``.
        ind : Index to determine the observation loglikelihood function.

    Returns:
        Loglikelihood of ``obs_data_i`` at index ``ind``.
    """
    b, r, alpha, D_e, D_I, D_q = params["theta"]
    I_in = r * ode_data_i[1, 0] / D_e
    H_in = ode_data_i[2, 0] / D_q
    Xin = jnp.array([I_in, H_in])
    return jnp.sum(jax.scipy.stats.poisson.logpmf(obs_data_i.flatten(), Xin))


def neglogpost_daltonng(upars):
    "Negative logposterior for non-Gaussian DALTON approximation."
    theta, X0, prior_Q, prior_R = constrain_pars(upars, dt_sim)
    # non-Gaussian DALTON loglikelihood
    ll = rodeo.inference.daltonng(
        key=key,  # immaterial, since not used
        # ode specification
        ode_fun=seirah_fun,
        ode_weight=W,
        ode_init=X0,
        t_min=t_min,
        t_max=t_max,
        theta=theta,
        # solver
        n_steps=n_steps,
        interrogate=rodeo.interrogate.interrogate_kramer,
        prior_weight=prior_Q,
        prior_var=prior_R,
        # non-gaussian measurement model
        obs_data=obs_data,
        obs_times=obs_times,
        obs_loglik_i=obs_loglik_i
    )
    return -(ll + seirah_logprior(upars))
```

Figure 5 displays the Basic and non-Gaussian DALTON posteriors for various step sizes Δt. All posteriors cover the true value of $\boldsymbol{\theta}$ and they are indistinguishable from the true posterior at $\Delta t = 0.1$.

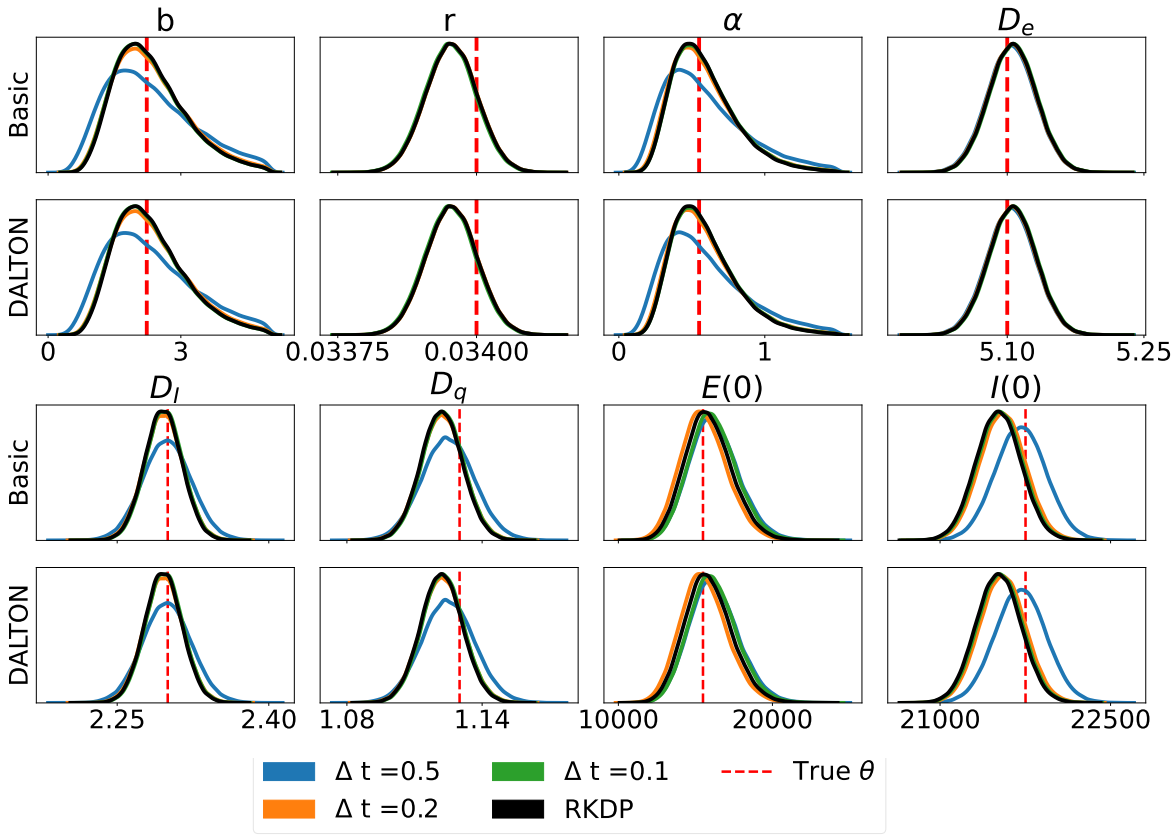

Figure 5: The parameter posteriors of the Basic and non-Gaussian DALTON methods for the SEIRAH model.

5.8. Speed Comparisons

Table 3 displays the speed of **rodeo** relative to different ODE solvers for the four examples above.

Table 3: Speed of **rodeo** relative to other ODE solvers.

Model	d	$d \cdot p$	N	LSODA	RK45	**rodeo** no blocking
Chkrebtii	1	4	30	4.38	3.32	0.98
FN	2	6	250	11.32	1.23	3.68
Hes1	3	9	120	4.62	1.93	3.95
SEIRAH	6	18	80	2.80	1.18	5.04

For each model we specify the number of system variables d, the number $d \cdot p$ of state variables in the **rodeo** Kalman filter, the number of **rodeo** evaluation points N required to obtain a Laplace posterior indistinguishable from that of the true ODE solution. We use the interrogation of Krämer *et al.* (2021) for the four examples. The deterministic ODE solvers under

comparison are LSODA and the adaptive Runge-Kutta (RK45) algorithms discussed at the beginning of Section 5 as implemented in **SciPy** and **diffrax**, respectively.

For the univariate example, blocking is slightly slower than no blocking because there is some overhead when computing over the extra dimension. However, problems are typically multivariate in practice. Thus we emphasize the results of the last three examples which indicate that 15% to 2 times faster than RK45 and 3-11 times faster than LSODA. We also compare timings to the $\mathcal{O}(d^3)$ implementation of **rodeo** which does not perform variable blocking. Our findings indicate that blocking is 3-5 times faster.

6. Discussion

We present **rodeo**, a fast and accurate probabilistic ODE solver. Its low memory cost is ideally suited for back-propagation, leading to a marked speed advantage relative to state-of-the-art deterministic solver on several ODE parameter learning problems. One limitation of the **rodeo** solver is the determination of the appropriate step size, which could be done adaptively as described in Schober *et al.* (2019).

References

Atkinson K, Han W, Stewart DE (2009). *Numerical Solution of Ordinary Differential Equations*. John Wiley & Sons. ISBN 978-0-470-04294-6.

Babuschkin I, Baumli K, Bell A, Bhupatiraju S, Bruce J, Buchlovsky P, Budden D, Cai T, Clark A, Danihelka I, Fantacci C, Godwin J, Jones C, Hemsley R, Hennigan T, Hessel M, Hou S, Kapturowski S, Keck T, Kemaev I, King M, Kunesch M, Martens L, Merzic H, Mikulik V, Norman T, Quan J, Papamakarios G, Ring R, Ruiz F, Sanchez A, Schneider R, Sezener E, Spencer S, Srinivasan S, Wang L, Stokowiec W, Viola F (2020). "The DeepMind JAX Ecosystem." URL http://github.com/deepmind.

Barber D, Wang Y (2014). "Gaussian processes for Bayesian estimation in ordinary Differential Equations." In EP Xing, T Jebara (eds.), *Proceedings of the 31st international conference on machine learning*, volume 32 of *Proceedings of machine learning research*, pp. 1485–1493. PMLR, Bejing, China. URL https://proceedings.mlr.press/v32/barber14.html.

Blondel M, Berthet Q, Cuturi M, Frostig R, Hoyer S, Llinares-López F, Pedregosa F, Vert JP (2021). "Efficient and Modular Implicit Differentiation." *arXiv preprint arXiv:2105.15183*.

Boersch-Supan PH, Ryan SJ, Johnson LR (2017). "deBInfer: Bayesian inference for dynamical models of biological systems in R." *Methods in Ecology and Evolution*, **8**(4), 511–518. doi:https://doi.org/10.1111/2041-210X.12679. https://besjournals.onlinelibrary.wiley.com/doi/pdf/10.1111/2041-210X.12679, URL https://besjournals.onlinelibrary.wiley.com/doi/abs/10.1111/2041-210X.12679.

Bosch N, Hennig P, Tronarp F (2021). "Calibrated Adaptive Probabilistic ODE Solvers." In A Banerjee, K Fukumizu (eds.), *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics*, volume 130 of *Proceedings of Machine Learning Research*, pp. 3466–3474. PMLR. URL https://proceedings.mlr.press/v130/bosch21a.html.

Bosch N, Tronarp F, Hennig P (2022). "Pick-and-Mix Information Operators for Probabilistic ODE Solvers." In G Camps-Valls, FJR Ruiz, I Valera (eds.), *Proceedings of The 25th International Conference on Artificial Intelligence and Statistics*, volume 151 of *Proceedings of Machine Learning Research*, pp. 10015–10027. PMLR. URL https://proceedings.mlr.press/v151/bosch22a.html.

Bradbury J, Frostig R, Hawkins P, Johnson MJ, Leary C, Maclaurin D, Necula G, Paszke A, VanderPlas J, Wanderman-Milne S, Zhang Q (2018). "JAX: composable transformations of Python+NumPy programs." URL http://github.com/google/jax.

Butcher JC (2008). *Numerical Methods for Ordinary Differential Equations*. John Wiley & Sons.

Calderhead B, Girolami M, Lawrence ND (2009). "Accelerating Bayesian inference over nonlinear differential equations with Gaussian processes." In *Advances in neural information processing systems*, pp. 217–224.

Chen TQ, Rubanova Y, Bettencourt J, Duvenaud DK (2018). "Neural Ordinary Differential Equations." In *Advances in Neural Information Processing Systems*, pp. 6571–6583.

Chkrebtii O, Campbell D (2019). "Adaptive step-size selection for state-space probabilistic differential equation solvers." *Statistics and Computing*, **29**. doi:10.1007/s11222-019-09899-5.

Chkrebtii OA, Campbell DA, Calderhead B, Girolami MA (2016). "Bayesian solution uncertainty quantification for differential equations." *Bayesian Analysis*, **11**(4), 1239–1267. ISSN 1936-0975. doi:10.1214/16-BA1017. URL https://projecteuclid.org/euclid.ba/1473276259.

Conrad PR, Girolami M, Särkkä S, Stuart A, Zygalakis K (2017). "Statistical analysis of differential equations: introducing probability measures on numerical solutions." *Statistics and Computing*, **27**(4), 1065–1082. ISSN 0960-3174, 1573-1375. doi:10.1007/s11222-016-9671-0. URL http://link.springer.com/10.1007/s11222-016-9671-0.

Dattner I, Yaari R (2020). *simode: Statistical Inference for Systems of Ordinary Differential Equations using Separable Integral-Matching*. R package version 1.2.0, URL https://CRAN.R-project.org/package=simode.

Diaconis P (1988). "Bayesian numerical analysis." In J Berger, S Gupta (eds.), *Statistical Decision Theory and Related Topics IV*, volume 1, pp. 163–175. Springer-Verlag, New York.

Dondelinger F, Husmeier D, Rogers S, Filippone M (2013). "ODE parameter inference using adaptive gradient matching with Gaussian processes." In CM Carvalho, P Ravikumar (eds.), *Proceedings of the sixteenth international conference on artificial intelligence and statistics*, volume 31 of *Proceedings of machine learning research*, pp. 216–228. PMLR, Scottsdale, Arizona, USA. URL https://proceedings.mlr.press/v31/dondelinger13a.html.

Dormand JR, Prince PJ (1980). "A family of embedded Runge-Kutta formulae." *Journal of Computational and Applied Mathematics*, **6**(1), 19–26. Publisher: Elsevier.

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

Gelman A, Carlin JB, Stern HS, Dunson DB, Vehtari A, Rubin DB (2013). *Bayesian Data Analysis*. 3rd edition. Chapman & Hall, New York, NY. ISBN 978-0-429-11307-9.

Ghosh S, Dasmahapatra S, Maharatna K (2017). "Fast approximate Bayesian computation for estimating parameters in differential equations." *Statistics and Computing*, **27**(1), 19–38. ISSN 0960-3174, 1573-1375. doi:10.1007/s11222-016-9643-4. URL http://link.springer.com/10.1007/s11222-016-9643-4.

Gianniotis N (2019). "Mixed Variational Inference." In *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. IEEE, Budapest, Hungary. ISBN 978-1-72811-985-4. doi:10.1109/IJCNN.2019.8852348. URL https://ieeexplore.ieee.org/document/8852348/.

Gorbach NS, Bauer S, Buhmann JM (2017). "Scalable variational inference for dynamical systems." In *Proceedings of the 31st international conference on neural information processing systems*, NIPS'17, pp. 4809–4818. Curran Associates Inc., Red Hook, NY, USA. ISBN 978-1-5108-6096-4.

Griffiths DF, Higham DJ (2010). *Numerical Methods for Ordinary Differential Equations: Initial Value Problems*. Springer Science & Business Media.

Hennig P, Osborne MA, Girolami M (2015). "Probabilistic numerics and uncertainty in computations." *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, **471**(2179), 20150142. ISSN 1364-5021, 1471-2946. doi:10.1098/rspa.2015.0142. URL https://royalsocietypublishing.org/doi/10.1098/rspa.2015.0142.

Hooker G, Ramsay JO, Xiao L (2016). "CollocInfer: Collocation Inference in Differential Equation Models." *Journal of Statistical Software*, **75**(2), 1–52. doi:10.18637/jss.v075.i02. URL https://www.jstatsoft.org/index.php/jss/article/view/v075i02.

Kersting H, Hennig P (2016). "Active uncertainty calibration in Bayesian ODE solvers." In *32nd conference on uncertainty in artificial intelligence (UAI 2016)*, pp. 309–318.

Kersting H, Krämer N, Schiegg M, Daniel C, Tiemann M, Hennig P (2020a). "Differentiable Likelihoods for Fast Inversion of Likelihood-Free Dynamical Systems." In HD III, A Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 5198–5208. PMLR. URL https://proceedings.mlr.press/v119/kersting20a.html.

Kersting H, Sullivan T, Hennig P (2020b). "Convergence rates of Gaussian ODE filters." *Statistics and Computing*, **30**, 1–26. doi:10.1007/s11222-020-09972-4.

Kidger P (2021). *On Neural Differential Equations*. Ph.D. thesis, University of Oxford. URL https://arxiv.org/pdf/2202.02435.pdf.

Krämer N, Bosch N, Schmidt J, Hennig P (2021). "Probabilistic ODE solutions in millions of dimensions." URL https://proceedings.mlr.press/v162/kramer22b/kramer22b.pdf.

Krämer N, Hennig P (2020). "Stable Implementation of Probabilistic ODE Solvers." doi:10.48550/ARXIV.2012.10106. URL https://arxiv.org/abs/2012.10106.

Lao J, Louf R (2020). "Blackjax: A sampling library for JAX." URL http://github.com/blackjax-devs/blackjax.

Lazarus A, Husmeier D, Papamarkou T (2018). "Multiphase MCMC sampling for parameter inference in nonlinear ordinary differential equations." In A Storkey, F Perez-Cruz (eds.), *Proceedings of the twenty-first international conference on artificial intelligence and statistics*, volume 84 of *Proceedings of machine learning research*, pp. 1252–1260. PMLR. URL https://proceedings.mlr.press/v84/lazarus18a.html.

Macdonald B, Dondelinger F (2020). *deGradInfer: Parameter Inference for Systems of Differential Equation*. R package version 1.0.1, URL https://CRAN.R-project.org/package=deGradInfer.

MacKay DJC (1992). "A Practical Bayesian Framework for Backpropagation Networks." *Neural Computation*, **4**(3), 448–472. ISSN 0899-7667, 1530-888X. doi:10.1162/neco.1992.4.3.448. URL https://direct.mit.edu/neco/article/4/3/448-472/5654.

Niu M, Wandy J, Daly R, Rogers S, Husmeier D (2021). "R package for statistical inference in dynamical systems using kernel based gradient matching: KGode." *Computational Statistics*, **36**(1), 715–747. doi:10.1007/s00180-020-01014-. URL https://ideas.repec.org/a/spr/compst/v36y2021i1d10.1007_s00180-020-01014-x.html.

Nocedal J, Wright SJ (2006). *Numerical Optimization*. 2e edition. Springer, New York, NY, USA.

Prague M, Wittkop L, Collin A, Dutartre D, Clairon Q, Moireau P, Thiébaut R, Hejblum BP (2020). "Multi-Level Modeling of Early COVID-19 Epidemic Dynamics in French Regions and Estimation of the Lockdown Impact on Infection Rate." *Preprint*, Epidemiology. doi:10.1101/2020.04.21.20073536.

Schober M, Duvenaud DK, Hennig P (2014). "Probabilistic ODE solvers with Runge-Kutta means." In *Advances in neural information processing systems*, pp. 739–747.

Schober M, Särkkä S, Hennig P (2019). "A probabilistic model for the numerical solution of initial value problems." *Statistics and Computing*, **29**(1), 99–122. ISSN 0960-3174, 1573-1375. doi:10.1007/s11222-017-9798-7. URL http://link.springer.com/10.1007/s11222-017-9798-7.

Sherwood WE (2013). *FitzHugh–Nagumo Model*, pp. 1–11. Springer New York, New York, NY. ISBN 978-1-4614-7320-6. doi:10.1007/978-1-4614-7320-6_147-1. URL https://doi.org/10.1007/978-1-4614-7320-6_147-1.

Skilling J (1992). "Bayesian solution of ordinary differential equations." In CR Smith, GJ Erickson, PO Neudorfer (eds.), *Maximum Entropy and Bayesian Methods: Seattle, 1991*,

Fundamental Theories of Physics, pp. 23–37. Springer Netherlands, Dordrecht. ISBN 978-94-017-2219-3. doi:10.1007/978-94-017-2219-3_2. URL https://doi.org/10.1007/978-94-017-2219-3_2.

Stan Development Team (2023). *Stan Modeling Language Users Guide and Reference Manual*. URL https://mc-stan.org.

Teymur O, Zygalakis K, Calderhead B (2016). "Probabilistic linear multistep methods." In *Advances in Neural Information Processing Systems 29*, pp. 4321–4328. Curran Associates, Inc.

Tronarp F, Bosch N, Hennig P (2022). "Fenrir: Physics-Enhanced Regression for Initial Value Problems." In K Chaudhuri, S Jegelka, L Song, C Szepesvari, G Niu, S Sabato (eds.), *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 21776–21794. PMLR. URL https://proceedings.mlr.press/v162/tronarp22a.html.

Tronarp F, Kersting H, Särkkä S, Hennig P (2018). "Probabilistic solutions to ordinary differential equations as non-linear Bayesian filtering: a new perspective." *arXiv:1810.03440 [stat]*. ArXiv: 1810.03440, URL http://arxiv.org/abs/1810.03440.

Wang H, Cao J (2022). *pCODE: Estimation of an Ordinary Differential Equation Model by Parameter Cascade Method*. R package version 0.9.4, URL https://CRAN.R-project.org/package=pCODE.

Wenger J, Krämer N, Pförtner M, Schmidt J, Bosch N, Effenberger N, Zenn J, Gessner A, Karvonen T, Briol FX, Mahsereci M, Hennig P (2021). "ProbNum: Probabilistic Numerics in Python." URL https://arxiv.org/abs/2112.02100.

Wenk P, Gotovos A, Bauer S, Gorbach NS, Krause A, Buhmann JM (2019). "Fast Gaussian process based gradient matching for parameter identification in systems of nonlinear ODEs." In K Chaudhuri, M Sugiyama (eds.), *Proceedings of the twenty-second international conference on artificial intelligence and statistics*, volume 89 of *Proceedings of machine learning research*, pp. 1351–1360. PMLR. URL https://proceedings.mlr.press/v89/wenk19a.html.

Wu M, Lysy M (2023). "Data-Adaptive Probabilistic Likelihood Approximation for Ordinary Differential Equations." 2306.05566.

Yang S, Wong SWK, Kou SC (2021). "Inference of dynamic systems from noisy and sparse data via manifold-constrained Gaussian processes." *Proceedings of the National Academy of Sciences*, **118**(15), e2020397118. ISSN 0027-8424, 1091-6490. doi:10.1073/pnas.2020397118. URL http://www.pnas.org/lookup/doi/10.1073/pnas.2020397118.

A. Kalman Functions

The Kalman recursions used in Algorithms 1, 2, 3, 4 and 7 are formulated in terms of the general Gaussian state space model

$$\begin{aligned} \boldsymbol{X}_n &= \boldsymbol{Q}_n \boldsymbol{X}_{n-1} + \boldsymbol{c}_n + \boldsymbol{R}_n^{1/2} \boldsymbol{\epsilon}_n \\ \boldsymbol{Z}_n &= \boldsymbol{W}_n \boldsymbol{X}_n + \boldsymbol{a}_n + \boldsymbol{V}_n^{1/2} \boldsymbol{\eta}_n \end{aligned}, \qquad \boldsymbol{\epsilon}_n, \boldsymbol{\eta}_n \overset{\text{ind}}{\sim} \text{Normal}(\boldsymbol{0}, \boldsymbol{I}). \tag{36}$$

Throughout, we use the notation $\boldsymbol{\mu}_{n|m} = \mathsf{E}[\boldsymbol{X}_n \mid \boldsymbol{Z}_{0:m}]$ and $\boldsymbol{\Sigma}_{n|m} = \mathrm{var}(\boldsymbol{X}_n \mid \boldsymbol{Z}_{0:m})$.

Algorithm 6 Standard Kalman filtering and smoothing functions.

procedure `kalman_predict`$(\boldsymbol{\mu}_{n-1|n-1}, \boldsymbol{\Sigma}_{n-1|n-1}, \boldsymbol{c}_n, \boldsymbol{Q}_n, \boldsymbol{R}_n)$
  $\boldsymbol{\mu}_{n|n-1} \leftarrow \boldsymbol{Q}_n \boldsymbol{\mu}_{n-1|n-1} + \boldsymbol{c}_n$
  $\boldsymbol{\Sigma}_{n|n-1} \leftarrow \boldsymbol{Q}_n \boldsymbol{\Sigma}_{n-1|n-1} \boldsymbol{Q}_n' + \boldsymbol{R}_n$
  return $\boldsymbol{\mu}_{n|n-1}, \boldsymbol{\Sigma}_{n|n-1}$
procedure `kalman_update`$(\boldsymbol{\mu}_{n|n-1}, \boldsymbol{\Sigma}_{n|n-1}, \boldsymbol{Z}_n, \boldsymbol{a}_n, \boldsymbol{W}_n, \boldsymbol{V}_n)$
  $\boldsymbol{A}_n \leftarrow \boldsymbol{\Sigma}_{n|n-1} \boldsymbol{W}_n' [\boldsymbol{W}_n \boldsymbol{\Sigma}_{n|n-1} \boldsymbol{W}_n' + \boldsymbol{V}_n]^{-1}$
  $\boldsymbol{\mu}_{n|n} \leftarrow \boldsymbol{\mu}_{n|n-1} + \boldsymbol{A}_n (\boldsymbol{Z}_n - \boldsymbol{W}_n \boldsymbol{\mu}_{n|n-1} - \boldsymbol{a}_n)$
  $\boldsymbol{\Sigma}_{n|n} \leftarrow \boldsymbol{\Sigma}_{n|n-1} - \boldsymbol{A}_n \boldsymbol{W}_n \boldsymbol{\Sigma}_{n|n-1}$
  return $\boldsymbol{\mu}_{n|n}, \boldsymbol{\Sigma}_{n|n}$
procedure `kalman_smooth`$(\boldsymbol{\mu}_{n+1|N}, \boldsymbol{\Sigma}_{n+1|N}, \boldsymbol{\mu}_{n|n}, \boldsymbol{\Sigma}_{n|n}, \boldsymbol{\mu}_{n+1|n}, \boldsymbol{\Sigma}_{n+1|n}, \boldsymbol{Q}_{n+1})$
  $\boldsymbol{A}_n \leftarrow \boldsymbol{\Sigma}_{n|n} \boldsymbol{Q}_{n+1}' \boldsymbol{\Sigma}_{n+1|n}^{-1}$
  $\boldsymbol{\mu}_{n|N} \leftarrow \boldsymbol{\mu}_{n|n} + \boldsymbol{A}_n (\boldsymbol{\mu}_{n+1|N} - \boldsymbol{\mu}_{n+1|n})$
  $\boldsymbol{\Sigma}_{n|N} \leftarrow \boldsymbol{\Sigma}_{n|n} + \boldsymbol{A}_n (\boldsymbol{\Sigma}_{n+1|N} - \boldsymbol{\Sigma}_{n+1|n}) \boldsymbol{A}_n'$
  return $\boldsymbol{\mu}_{n|N}, \boldsymbol{\Sigma}_{n|N}$
procedure `kalman_sample`$(\boldsymbol{X}_{n+1}, \boldsymbol{\mu}_{n|n}, \boldsymbol{\Sigma}_{n|n}, \boldsymbol{\mu}_{n+1|n}, \boldsymbol{\Sigma}_{n+1|n}, \boldsymbol{Q}_{n+1})$
  $\boldsymbol{A}_n \leftarrow \boldsymbol{\Sigma}_{n|n} \boldsymbol{Q}_{n+1}' \boldsymbol{\Sigma}_{n+1|n}^{-1}$
  $\tilde{\boldsymbol{\mu}}_{n|N} \leftarrow \boldsymbol{\mu}_{n|n} + \boldsymbol{A}_n (\boldsymbol{X}_{n+1} - \boldsymbol{\mu}_{n+1|n})$ ▷ $\tilde{\boldsymbol{\mu}}_{n|N} = \mathsf{E}[\boldsymbol{X}_n \mid \boldsymbol{X}_{n+1:N}, \boldsymbol{Z}_{0:N}]$
  $\tilde{\boldsymbol{\Sigma}}_{n|N} \leftarrow \boldsymbol{\Sigma}_{n|n} - \boldsymbol{A}_n \boldsymbol{Q}_{n+1} \boldsymbol{\Sigma}_{n|n}$ ▷ $\tilde{\boldsymbol{\Sigma}}_{n|N} = \mathrm{var}[\boldsymbol{X}_n \mid \boldsymbol{X}_{n+1:N}, \boldsymbol{Z}_{0:N}]$
  $\boldsymbol{X}_n \sim \mathrm{Normal}(\tilde{\boldsymbol{\mu}}_{n|N}, \tilde{\boldsymbol{\Sigma}}_{n|N})$
  return $\boldsymbol{X}_n$
procedure `kalman_forecast`$(\boldsymbol{\mu}_{n|n-1}, \boldsymbol{\Sigma}_{n|n-1}, \boldsymbol{a}_n, \boldsymbol{W}_n, \boldsymbol{V}_n)$
  $\boldsymbol{\lambda}_{n|n-1} \leftarrow \boldsymbol{W}_n \boldsymbol{\mu}_{n|n-1} + \boldsymbol{a}_n$ ▷ $\boldsymbol{\lambda}_{n|n-1} = \mathsf{E}[\boldsymbol{Z}_n \mid \boldsymbol{Z}_{0:n-1}]$
  $\boldsymbol{\Omega}_{n|n-1} \leftarrow \boldsymbol{W}_n \boldsymbol{\mu}_{n|n-1} \boldsymbol{W}_n' + \boldsymbol{V}_n$ ▷ $\boldsymbol{\Omega}_{n|n-1} = \mathrm{var}[\boldsymbol{Z}_n \mid \boldsymbol{Z}_{0:n-1}]$
  return $\boldsymbol{\lambda}_{n|n-1}, \boldsymbol{\Omega}_{n|n-1}$
procedure `kalman_cond`$(\boldsymbol{\mu}_{n|n}, \boldsymbol{\Sigma}_{n|n}, \boldsymbol{\mu}_{n+1|n}, \boldsymbol{\Sigma}_{n+1|n}, \boldsymbol{Q}_{n+1})$
  $\boldsymbol{A}_n \leftarrow \boldsymbol{\Sigma}_{n|n} \boldsymbol{Q}_{n+1}' \boldsymbol{\Sigma}_{n+1|n}^{-1}$
  $\boldsymbol{b}_n \leftarrow \boldsymbol{\mu}_{n|n} - \boldsymbol{A}_n \boldsymbol{\mu}_{n+1|n}$
  $\boldsymbol{C}_n \leftarrow \boldsymbol{\Sigma}_{n|n} - \boldsymbol{A}_n \boldsymbol{Q}_n \boldsymbol{\Sigma}_{n|n}$
  return $\boldsymbol{A}_n, \boldsymbol{b}_n, \boldsymbol{C}_n$ ▷ $\boldsymbol{X}_n \mid \boldsymbol{X}_{n+1} \sim \mathrm{Normal}(\boldsymbol{A}_n \boldsymbol{X}_{n+1} + \boldsymbol{b}_n, \boldsymbol{C}_n)$

B. DALTON for Non-Gaussian Measurements

In the case of the general measurement model (12), which we write as

$$\boldsymbol{Y}_i \overset{\text{ind}}{\sim} \exp\{g_i(\boldsymbol{Y}_i \mid \boldsymbol{s}_{n(i)}, \boldsymbol{\phi})\}, \tag{37}$$

where $\boldsymbol{s}_{n(i)} = \boldsymbol{D}_i \boldsymbol{X}_{n(i)}$ and $\boldsymbol{u}_{n(i)}$ denote observed and unobserved components of $\boldsymbol{x}(t)$ at time $t = t_i'$, such that $\frac{\partial}{\partial \boldsymbol{u}_{n(i)}} p(\boldsymbol{Y}_i \mid \boldsymbol{x}_{n(i)}, \boldsymbol{\phi}) = \boldsymbol{0}$ and $\boldsymbol{D}_i$ is a mask matrix of zeros and ones. In

order to compute the likelihood (13), omitting the dependence on $\boldsymbol{\Theta}$ we consider the identity,

$$\begin{aligned} p(\boldsymbol{Y}_{0:M} \mid \boldsymbol{Z}_{1:N} = \boldsymbol{0}) &= \frac{p(\boldsymbol{Y}_{0:M}, \boldsymbol{X}_{0:N} \mid \boldsymbol{Z}_{1:N} = \boldsymbol{0})}{p(\boldsymbol{X}_{0:N} \mid \boldsymbol{Y}_{0:M}, \boldsymbol{Z}_{1:N} = \boldsymbol{0})} \\ &= \frac{p(\boldsymbol{X}_{0:N} \mid \boldsymbol{Z}_{1:N} = \boldsymbol{0}) \times \prod_{i=0}^{M} \exp\{g_i(\boldsymbol{Y}_i \mid \boldsymbol{s}_{n(i)})\}}{p(\boldsymbol{X}_{0:N} \mid \boldsymbol{Y}_{0:M}, \boldsymbol{Z}_{1:N} = \boldsymbol{0})}, \end{aligned} \tag{38}$$

which holds for any value of $\boldsymbol{X}_{0:N}$. The numerator of (38) $p(\boldsymbol{X}_{0:N} \mid \boldsymbol{Z}_{1:N} = \boldsymbol{0})$ can be approximated using the Kalman smoothing algorithm applied to the data-free surrogate model (11), whereas the product term is obtained via straightforward calculation of (37). As for the denominator of (38), we propose to approximate it by a multivariate normal distribution which after simplifications gives the model

$$\begin{aligned} \boldsymbol{X}_{n+1} \mid \boldsymbol{X}_n &\sim \text{Normal}(\boldsymbol{Q}_{\boldsymbol{\eta}} \boldsymbol{X}_n, \boldsymbol{R}_{\boldsymbol{\eta}}) \\ \boldsymbol{Z}_n &\overset{\text{ind}}{\sim} \text{Normal}(\dot{\boldsymbol{x}}_n - \boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}_n, t_n), \boldsymbol{V}) \\ \hat{\boldsymbol{Y}}_i &\overset{\text{ind}}{\sim} \text{Normal}(\boldsymbol{s}_{n(i)}, -[\nabla^2 h_i(\hat{\boldsymbol{s}}_{n(i)})]^{-1}). \end{aligned} \tag{39}$$

where $h_i(\boldsymbol{s}) = g_i(\boldsymbol{Y}_i \mid \boldsymbol{s}, \boldsymbol{\phi})$ is a short-hand notation, ∇h_i and $\nabla^2 h_i$ are the gradient and Hessian of h_i respectively, and $\hat{\boldsymbol{Y}}_i = \hat{\boldsymbol{s}}_{n(i)} - \nabla^2 h_i(\hat{\boldsymbol{s}}_{n(i)})^{-1} \nabla h_i(\hat{\boldsymbol{s}}_{n(i)})$. We may now augment the surrogate model (11) exactly as in Gaussian setting of Section 3.3 to obtain an estimate of $p_{\text{lin}}(\boldsymbol{X}_{0:N} \mid \hat{\boldsymbol{Y}}_{0:M}, \boldsymbol{Z}_{1:N} = \boldsymbol{0})$ by the Kalman smoother. To complete the algorithm, there remains the choice $\boldsymbol{X}_{0:N}$ to plug into (38), and the choice of $\hat{\boldsymbol{s}}_{n(i)}$ in (39). For the latter, we use $\hat{\boldsymbol{s}}_{n(i)} = \mathsf{E}_{\text{lin}}[\boldsymbol{s}_{n(i)} \mid \hat{\boldsymbol{Y}}_{0:i-1}, \boldsymbol{Z}_{0:n(i)-1} = \boldsymbol{0}]$, the predicted mean of the surrogate model obtained from (39). For the former, we use $\boldsymbol{X}_{0:N} = \mathsf{E}_{\text{lin}}[\boldsymbol{X}_{0:N} \mid \hat{\boldsymbol{Y}}_{0:M}, \boldsymbol{Z}_{0:N} = \boldsymbol{0}]$, the Kalman smoother mean of $p_{\text{lin}}(\boldsymbol{X}_{0:N} \mid \hat{\boldsymbol{Y}}_{0:M}, \boldsymbol{Z}_{0:N} = \boldsymbol{0})$. For full details, please see Wu and Lysy (2023).

Affiliation:

Mohan Wu, Martin Lysy
Department of Statistics and Actuarial Science
University of Waterloo
E-mail: mlysy@uwaterloo.ca

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd

Algorithm 7 DALTON probabilistic ODE likelihood approximation for non-Gaussian measurements.

1: **procedure** `daltonng`($\boldsymbol{W} = \boldsymbol{W}_{\boldsymbol{\theta}}, \boldsymbol{f}(\boldsymbol{X}, t) = \boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{X}, t), \boldsymbol{v} = \boldsymbol{v}_{\boldsymbol{\theta}}, \boldsymbol{Q} = \boldsymbol{Q}_{\boldsymbol{\eta}}, \boldsymbol{R} = \boldsymbol{R}_{\boldsymbol{\eta}}, \boldsymbol{Y}_{0:M} = \boldsymbol{Y}_{0:M}, \boldsymbol{g}_{0:M}(\boldsymbol{Y}, \boldsymbol{X}, \boldsymbol{\phi}) = \boldsymbol{g}_{0:M}(\boldsymbol{Y}, \boldsymbol{X}, \boldsymbol{\phi})$)
2: $\quad \boldsymbol{\mu}_{0|0}, \Sigma_{0|0} \leftarrow \boldsymbol{v}, \boldsymbol{0}$ ▷ Initialization
3: $\quad \boldsymbol{Z}_{1:N} \leftarrow \boldsymbol{0}$
4: $\quad \ell_{xz}, \ell_{xyz}, \ell_y \leftarrow 0, 0, 0$
5: $\quad i \leftarrow 0$ ▷ Used to map t_n to t'_i
6: ▷ Lines 7-30 compute $\log p(\boldsymbol{X}_{0:N} \mid \hat{\boldsymbol{Y}}_{0:M}, \boldsymbol{Z}_{1:N} = \boldsymbol{0}, \boldsymbol{\Theta})$
7: $\quad$ **for** $n = 1 : N$ **do**
8: $\quad\quad$ **for** $k = 1 : d$ **do**
9: $\quad\quad\quad \boldsymbol{\mu}^{(k)}_{n|n-1}, \boldsymbol{\Sigma}^{(k)}_{n|n-1} \leftarrow$ `kalman_predict`$(\boldsymbol{\mu}^{(k)}_{n-1|n-1}, \boldsymbol{\Sigma}^{(k)}_{n-1|n-1}, \boldsymbol{0}, \boldsymbol{Q}^{(k)}, \boldsymbol{R}^{(k)})$
10: $\quad\quad \boldsymbol{a}_n, \boldsymbol{B}_n, \boldsymbol{V}_n \leftarrow$ `interrogate`$(\boldsymbol{\mu}_{n|n-1}, \boldsymbol{\Sigma}_{n|n-1}, \boldsymbol{W}, \boldsymbol{f}(\boldsymbol{X}, t_n))$
11: $\quad\quad$ **if** $t_n = t_{n(i)}$ **then**
12: $\quad\quad\quad \boldsymbol{g}_{(1)} \leftarrow \frac{\partial}{\partial \boldsymbol{s}_{n(i)}} g_i(\boldsymbol{Y}_i, \boldsymbol{D}_i \boldsymbol{\mu}_{n|n-1}, \boldsymbol{\phi})$
13: $\quad\quad\quad \boldsymbol{g}_{(2)} \leftarrow \frac{\partial^2}{\partial \boldsymbol{s}_{n(i)} \partial \boldsymbol{s}'_{n(i)}} g_i(\boldsymbol{Y}_i, \boldsymbol{D}_i \boldsymbol{\mu}_{n|n-1}, \boldsymbol{\phi})$
14: $\quad\quad\quad$ **for** $k = 1 : d$ **do**
15: $\quad\quad\quad\quad \hat{\boldsymbol{Y}}^{(k)}_i \leftarrow \boldsymbol{D}^{(k)}_i \boldsymbol{\mu}^{(k)}_{n|n-1} - \boldsymbol{g}^{-1(k)}_{(2)} \boldsymbol{g}^{(k)}_{(1)}$ ▷ Compute pseudo-observations
16: $\quad\quad\quad\quad \boldsymbol{Z}^{(k)}_n \leftarrow \begin{bmatrix} \boldsymbol{Z}^{(k)}_n \\ \hat{\boldsymbol{Y}}^{(k)}_i \end{bmatrix}, \qquad \boldsymbol{W}^{(k)} \leftarrow \begin{bmatrix} \boldsymbol{W} \\ \boldsymbol{D}^{(k)}_i \end{bmatrix}, \qquad \boldsymbol{B}^{(k)}_n \leftarrow \begin{bmatrix} \boldsymbol{B}^{(k)}_n \\ \boldsymbol{0} \end{bmatrix}$
17: $\quad\quad\quad\quad \boldsymbol{a}^{(k)}_n \leftarrow \begin{bmatrix} \boldsymbol{a}^{(k)}_n \\ \boldsymbol{0} \end{bmatrix}, \qquad \boldsymbol{V}^{(k)}_n \leftarrow \begin{bmatrix} \boldsymbol{V}^{(k)}_n & \boldsymbol{0} \\ \boldsymbol{0} & -\boldsymbol{g}^{-1(k)}_{(2)} \end{bmatrix}$
18: $\quad\quad\quad i \leftarrow i + 1$
19: $\quad\quad$ **for** $k = 1 : d$ **do**
20: $\quad\quad\quad \boldsymbol{\mu}^{(k)}_{n|n}, \boldsymbol{\Sigma}^{(k)}_{n|n} \leftarrow$ `kalman_update`$(\boldsymbol{\mu}^{(k)}_{n|n-1}, \boldsymbol{\Sigma}^{(k)}_{n|n-1}, \boldsymbol{Z}^{(k)}_n, \boldsymbol{a}^{(k)}_n, \boldsymbol{W}^{(k)} + \boldsymbol{B}^{(k)}_n, \boldsymbol{V}^{(k)}_n)$
21: $\quad$ **for** $n = N - 1 : 1$ **do**
22: $\quad\quad$ **for** $k = 1 : d$ **do**
23: $\quad\quad\quad \boldsymbol{\mu}^{(k)}_{n|N}, \boldsymbol{\Sigma}^{(k)}_{n|N} \leftarrow$ `kalman_smooth`$(\boldsymbol{\mu}^{(k)}_{n+1|N}, \boldsymbol{\Sigma}^{(k)}_{n+1|N}, \boldsymbol{\mu}^{(k)}_{n|n}, \boldsymbol{\Sigma}^{(k)}_{n|n}, \boldsymbol{\mu}^{(k)}_{n+1|n}, \boldsymbol{\Sigma}^{(k)}_{n+1|n}, \boldsymbol{Q}^{(k)})$
24: $\quad\quad\quad \boldsymbol{\mu}^{(k)}_n, \boldsymbol{\Sigma}^{(k)}_n \leftarrow$ `kalman_sample`$(\boldsymbol{\mu}^{(k)}_{n+1|N}, \boldsymbol{\mu}^{(k)}_{n|n}, \boldsymbol{\Sigma}^{(k)}_{n|n}, \boldsymbol{\mu}^{(k)}_{n+1|n}, \boldsymbol{\Sigma}^{(k)}_{n+1|n}, \boldsymbol{Q}^{(k)})$
25: $\quad\quad\quad \ell_{xyz} \leftarrow \ell_{xyz} +$ `normal_logpdf`$(\boldsymbol{\mu}^{(k)}_{n|N}; \boldsymbol{\mu}^{(k)}_n, \boldsymbol{\Sigma}^{(k)}_n)$
26: $\quad$ **for** $k = 1 : d$ **do**
27: $\quad\quad \ell_{xyz} \leftarrow \ell_{xyz} +$ `normal_logpdf`$(\boldsymbol{\mu}^{(k)}_{N|N}; \boldsymbol{\mu}^{(k)}_{N|N}, \boldsymbol{\Sigma}^{(k)}_{N|N})$

Algorithm 8 Non-Gaussian DALTON (continued)

▷ Lines 31-45 compute $\log p(\boldsymbol{X}_{0:N} \mid \boldsymbol{Z}_{1:N} = \boldsymbol{0}, \boldsymbol{\Theta})$
$\boldsymbol{Z}_{1:N} \leftarrow \boldsymbol{0}$ ▷ Reset $\boldsymbol{Z}_{1:N} = \boldsymbol{0}$
$\boldsymbol{\mu}'_{0:N|N} \leftarrow \boldsymbol{\mu}_{0:N|N}$ ▷ Store $\boldsymbol{\mu}_{0:N|N} = \mathsf{E}[\boldsymbol{X}_{0:N} \mid \boldsymbol{Z}_{1:N}, \hat{\boldsymbol{Y}}_{0:N}]$ for later
for $n = 1 : N$ **do**
  for $k = 1 : d$ **do**
    $\boldsymbol{\mu}^{(k)}_{n|n-1}, \boldsymbol{\Sigma}^{(k)}_{n|n-1} \leftarrow \texttt{kalman\_predict}(\boldsymbol{\mu}^{(k)}_{n-1|n-1}, \boldsymbol{\Sigma}^{(k)}_{n-1|n-1}, \boldsymbol{0}, \boldsymbol{Q}^{(k)}, \boldsymbol{R}^{(k)})$
  $\boldsymbol{a}_n, \boldsymbol{B}_n, \boldsymbol{V}_n \leftarrow \texttt{interrogate}(\boldsymbol{\mu}_{n|n-1}, \boldsymbol{\Sigma}_{n|n-1}, \boldsymbol{W}, \boldsymbol{f}(\boldsymbol{X}, t_n, \boldsymbol{\phi}))$
  for $k = 1 : d$ **do**
    $\boldsymbol{\mu}^{(k)}_{n|n}, \boldsymbol{\Sigma}^{(k)}_{n|n} \leftarrow \texttt{kalman\_update}(\boldsymbol{\mu}^{(k)}_{n|n-1}, \boldsymbol{\Sigma}^{(k)}_{n|n-1}, \boldsymbol{Z}^{(k)}_n, \boldsymbol{a}^{(k)}_n, \boldsymbol{W}^{(k)} + \boldsymbol{B}^{(k)}_n, \boldsymbol{V}^{(k)}_n)$
for $n = N - 1 : 1$ **do**
  for $k = 1 : d$ **do**
    $\boldsymbol{\mu}^{(k)}_n, \boldsymbol{\Sigma}^{(k)}_n \leftarrow \texttt{kalman\_sample}({\boldsymbol{\mu}'_{n+1|N}}^{(k)}, \boldsymbol{\mu}^{(k)}_{n|n}, \boldsymbol{\Sigma}^{(k)}_{n|n}, \boldsymbol{\mu}^{(k)}_{n+1|n}, \boldsymbol{\Sigma}^{(k)}_{n+1|n}, \boldsymbol{Q}^{(k)})$
    $\ell_{xz} \leftarrow \ell_{xz} + \texttt{normal\_logpdf}({\boldsymbol{\mu}'_{n|N}}^{(k)}; \boldsymbol{\mu}^{(k)}_n, \boldsymbol{\Sigma}^{(k)}_n)$
for $k = 1 : d$ **do**
  $\ell_{xz} \leftarrow \ell_{xz} + \texttt{normal\_logpdf}({\boldsymbol{\mu}'_{N|N}}^{(k)}; \boldsymbol{\mu}^{(k)}_{N|N}, \boldsymbol{\Sigma}^{(k)}_{N|N})$
▷ Lines 46-47 compute $\log p(\boldsymbol{Y}_{0:M} \mid \boldsymbol{X}_{0:N}, \boldsymbol{\phi})$
for $i = 0 : M$ **do**
  $\ell_y \leftarrow \ell_y + g_i(\boldsymbol{Y}_i, \boldsymbol{D}_i \boldsymbol{\mu}'_{n(i)|N}, \boldsymbol{\phi})$

return $\ell_{xz} - \ell_y - \ell_{xyz}$ ▷ Estimate of $\log p(\boldsymbol{Y}_{0:M} \mid \boldsymbol{Z}_{1:N} = \boldsymbol{0}, \boldsymbol{\Theta})$