Accelerating Inference: Mitotic Stein Variational Gradient Descent for Bayesian Analysis of Dynamical Systems

a thesis presented
by
Jamie Liu
to
The Departments of Statistics and Computer Science

in partial fulfillment of the requirements
for the degree of
Bachelor of Arts (Honors)
in the subject of
Statistics and Computer Science

Harvard University
Cambridge, Massachusetts
May 2025

Thesis advisor: Professor Samuel Kou Jamie Liu

Accelerating Inference: Mitotic Stein Variational Gradient Descent for Bayesian Analysis of Dynamical Systems

Abstract

This thesis introduces mitotic Stein variational gradient descent (mSVGD), a novel enhancement to Stein variational gradient descent (SVGD) designed to improve speed, convergence behavior, and robustness of particle-based variational inference. The research focuses on addressing computational inefficiencies in manifold-constrained Gaussian Process (MAGI) inference, a framework for Bayesian inference of ordinary differential equation systems. By leveraging a structured particle expansion approach inspired by mitotic cell division, mSVGD mitigates sensitivity to hyperparameter selection, accelerates convergence, and enhances robustness against noisy and sparse data.

Empirical evaluations on the FitzHugh-Nagumo, Hes1 protein, and Lorenz models demonstrate that mSVGD achieves a $30\times$ to $50\times$ speedup over the traditional MAGI implementation that uses Hamiltonian Monte Carlo sampling, while maintaining or improving inference accuracy. The proposed method also exhibits superior consistency in convergence behavior and increased stability compared to SVGD. These results position mSVGD as a scalable and efficient alternative to both traditional MCMC-based inference techniques and standard SVGD.

This work contributes to ongoing advancements in variational inference, particularly for dynamical systems with computational constraints, and highlights the potential of mSVGD as a powerful tool for Bayesian inference in complex, real-world scientific applications.

Contents

Listing of figures

For my loving and supportive family.

Acknowledgments

I would like to express my deepest gratitude to my advisor, Professor Samuel Kou, for his invaluable guidance, insightful discussions, and unwavering support throughout the development of this thesis. His expertise, patience, and encouragement throughout the research process have been instrumental in shaping both this work and my academic growth, and greatly contributed to my decision to pursue graduate studies in statistics.

I would also like to thank Professor David Alvarez-Melis for serving as my second reader. Thank you to Dr. Alex Young for coordinating STAT 99R and providing a guiding voice throughout the thesis-writing process.

Next, I extend my sincere thanks to Professor Joesph Blitzstein. Before deciding to concentrate in statistics, I took STAT 110 on a whim, which was undoubtedly one of the best decisions in my academic career. From igniting my passion for statistics to giving me the opportunity to serve as a teaching fellow for STAT 110 and STAT 111, I don't think I can overstate how much Professor Blitzstein has contributed to my growth as an academic and as a human being. Thank you, Joe.

I must also thank Skyler Wu, who has been a peer, a friend, a mentor, and a role model to me for the past couple years. For giving me advice — about my thesis, research, and beyond — and for answering questions that I felt were too stupid to ask an advisor or professor, thank you.

Thank you to my treasured friends, who have supported me through all the different stages of my life. You know who you are.

Last but not least, I would like to thank my family. My mom, Wei, who has provided guidance, wisdom, and support, and has sacrificed so much for me. My dad, Jun, who is supportive of all of my endeavors and has helped shine a light through the unfamiliar waters of the world of academia. My sister, Jackie, who has zero interest in the content of this thesis, but has given me enthusiastic encouragement as always. Thank you and I love you.

1 Introduction

1.1 Motivation and Objectives

Broadly speaking, a dynamical system refers to a system in which a function or set of functions is used to describe the time dependence of a point in some space. For this thesis, we will mainly focus on dynamical systems that describe points in $\mathbb{R}^D$ that are governed by systems of ordinary differential equations (ODEs) over a time interval $[0, T]$.

ODE-based dynamical system models excel in cases for which modeling the time dependence of a variable is easier or more informative than directly modeling the variable itself. There have been successful applications of ODE models in many scientific disciplines for modeling natural phenomena, including but not limited to neuron spike generation (Section 6.2), mRNA and protein oscillation (Section 6.3), and chaotic weather patterns (Section 6.4).

Consider a variable $\boldsymbol{x}(t) = (x_1(t), \ldots, x_D(t)) \in \mathbb{R}^D$ governed by a system of ODEs with parameters $\boldsymbol{\theta} \in \mathbb{R}^p$. We can describe the behavior of $\boldsymbol{x}(t)$ at a time $t \in [0, T]$ with $\boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{\theta}, t)$, the vector containing the derivatives of each component of $\boldsymbol{x}(t)$ with respect to time.

$$\boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{\theta}, t) = \dot{\boldsymbol{x}}(t) = \frac{d\boldsymbol{x}(t)}{dt} = \left(\frac{dx_1(t)}{dt}, \ldots, \frac{dx_D(t)}{dt} \right)$$

In theory, given an initial condition $\boldsymbol{x}(0)$ and the parameter values $\boldsymbol{\theta}$, one could integrate over the interval $[0, T]$ to obtain the value of $\boldsymbol{x}(t)$ at any time $t \in [0, T]$ by following along the directions described by the derivatives in $\boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{\theta}, t)$. In practice, this process can be approximated via numerical integration.

As an illustrative example, consider the three-component system $\boldsymbol{x} = (T_U, T_I, V)$ for modeling HIV infection described in the simulation study by Liang, Miao, and Wu [5] in 2010, where T_U, T_I are the concentrations of uninfected and infected cells, respectively, and V is the viral load. The system obeys the ODEs

$$\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\theta}, t) = \left(\frac{dT_U}{dt}, \frac{dT_I}{dt}, \frac{dV}{dt} \right) = \begin{pmatrix} \lambda - \rho T_U - \eta(t) T_U V \\ \eta(t) T_U V - \delta T_I \\ N \delta T_I - cV \end{pmatrix} \tag{1.1}$$

where $\eta(t) = 9 \times 10^{-5} \times (1 - 0.9 \cos(\pi t / 1000))$ and $\boldsymbol{\theta} = (\lambda, \rho, \delta, N, c)$ are the associated parameters. For this example, we set the ground truth values $\boldsymbol{\theta} = (36, 0.108, 0.5, 1000, 3)$ and initial conditions $\boldsymbol{x}(0) = (600, 30, 100000)$.

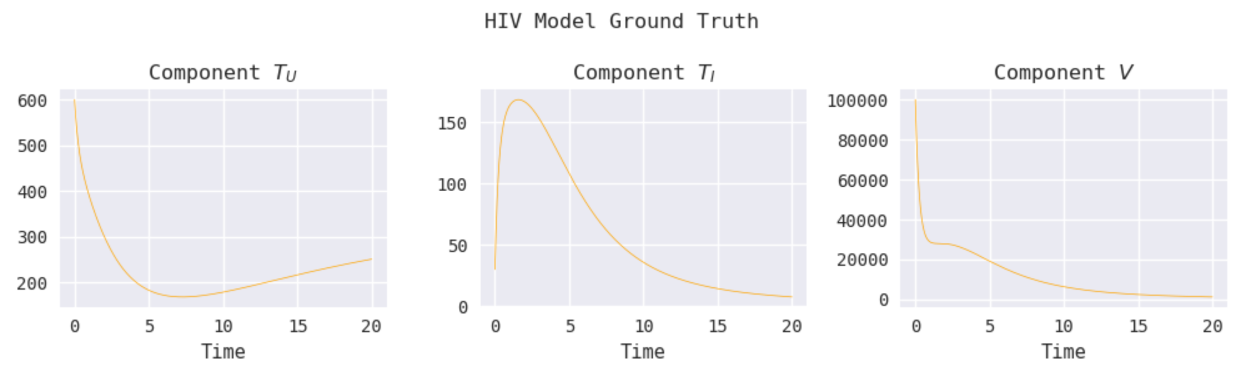


Figure 1.1.1: HIV model ground truth trajectories.

In practical applications, we generally observe some noisy data from the trajectory of $\boldsymbol{x}$ at a set of times $\boldsymbol{\tau} = (\boldsymbol{\tau}_1, \ldots, \boldsymbol{\tau}_D)$, which we denote as $\boldsymbol{y}(\boldsymbol{\tau}) = (\boldsymbol{y}_1(\boldsymbol{\tau}_1), \ldots, \boldsymbol{y}_D(\boldsymbol{\tau}_D))$. The structure of the ODE is known or assumed, but the parameters $\boldsymbol{\theta}$ are unknown. This setup naturally gives rise to several important learning tasks, of which we will focus on two.

1. *Parameter inference.* Given noisy observations $\boldsymbol{y}(\boldsymbol{\tau})$, we want to learn the parameters $\boldsymbol{\theta}$ that govern the ODEs which describe the underlying behavior of the dynamical system. The values of $\boldsymbol{\theta}$ usually have domain-specific interpretations and information about them could be crucial in practice.

2. *Trajectory recovery.* Given noisy observations $\boldsymbol{y}(\boldsymbol{\tau})$, we want to learn and reconstruct the underlying ground-truth trajectory $\boldsymbol{x}(t)$. In other words, we would like to de-noise the data and extract the true signal to learn about the behavior of the system.

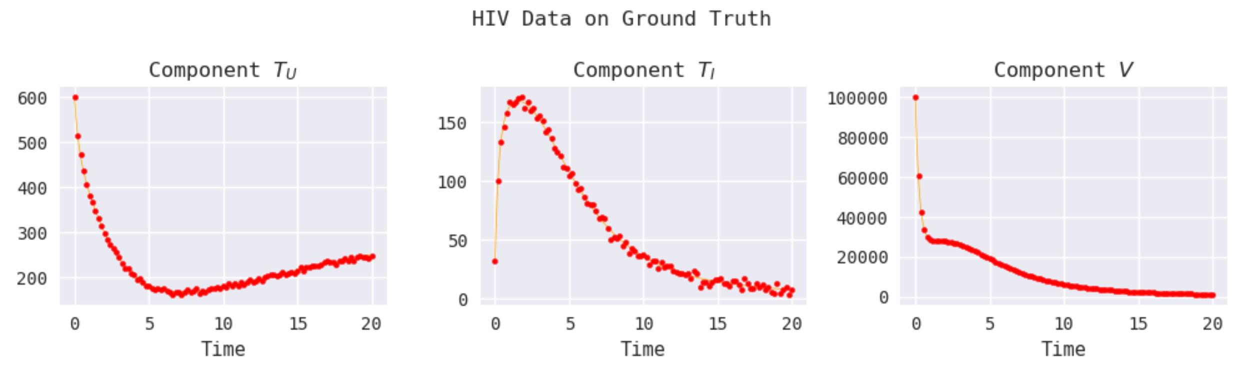


Figure 1.1.2: HIV model ground truth trajectories and dataset with NumPy seed 2025, with observations at $\tau = (0, 0.2, 0.4, \ldots, 20)$ for all components and sampling standard deviations $\boldsymbol{\sigma} = (\sqrt{10}, \sqrt{10}, 10)$.

As this is an applied statistical learning task, we have multiple objectives to consider. First, we would like our results to be accurate and interpretable. We will discuss what this means in terms of ODE models in Section 6.1. Second, and equally important, we want our methods to be computationally feasible. Achieving perfect results asymptotically may be meaningless in real-life settings with limited time, money, and computational power.

This thesis builds upon an existing method for performing inference on ODE models called MAGI [13]. The central goal is to reduce MAGI's computational cost by increasing computational speed, while endeavoring to retain accuracy and interpretability.

1.2 Overview and Contributions

In 2021, Yang, Wong, and Kou introduced Manifold-contrained Gaussian Process Inference (MAGI), which was shown to be able to perform these inference tasks better than previously existing methods [13]. We go into the details of the MAGI method in Chapter 2, then discuss computational challenges and some ideas for a faster algorithm in Chapter 3.

Chapter 4 explores the seminal work on particle-based variational inference, Stein Variational Gradient Descent (SVGD), which was presented by Liu and Wang in 2016 [8]. In Section 4.3, I propose an augmentation to SVGD, for which I have coined the name *mitotic Stein variational gradient descent* (mSVGD), that aims to improve the speed, stability, and robustness of the original method.

We bring these ideas together in Chapter 5, where we detail the Python package that I wrote to implement mSVGD for MAGI. We discuss the mathematical derivation and code design, as well as give a tutorial with package documentation, using the HIV model as a running example.

In Chapter 6, we consider three additional real-world dynamical systems to compare the performance of my package with the existing MAGI package, which is implemented in C++ with an R front-end [12]. We show that mSVGD provides equally or more accurate inference with a much faster computational speed, ranging from approximately 30× to 50× faster on average, depending on the test case. We also compare mSVGD to standard SVGD, demonstrating its much faster convergence, more consistent convergence behavior, and robustness to hyperparameter settings and variance in data.

Finally, we conclude in Chapter 7 with a discussion of main takeaways, some avenues for future work and exploration, and a few closing remarks.

1.3 Testing and Reproducibility

The testing in this thesis is done using synthetic data. For each test setting, we define a system of ODEs, pick an initial condition $\boldsymbol{x}(0)$, and set the ground-truth parameters $\boldsymbol{\theta}$. We also set the sampling standard deviation $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_D)$, which describes the noise of the observations, and decide the time points $\boldsymbol{\tau}$ at which we will make observations. We mostly follow the ground-truth settings described by the original MAGI paper [13]. Then, we use numerical integration to solve for the ground-truth trajectory $\boldsymbol{x}(t)$. For each psuedosample, we draw samples centered at $\boldsymbol{x}(\boldsymbol{\tau})$ with noise governed by $\boldsymbol{\sigma}$.

The source code of my Python package can be found on GitHub [6] and installed via pip following the tutorial in Chapter 5. The code used to generate noisy observations from the example dynamical systems is included. The existing MAGI package can be installed for R from CRAN [12]. Its source code can also be found on GitHub [11].

For accuracy and speed performance, we analyze results across $B = 100$ randomly generated pseudosamples for each test setting. Testing was conducted locally on consumer-grade hardware, using an Intel(R) i7-13700K CPU and NVIDIA GeForce RTX 4080 SUPER GPU. For the demonstrative figures included in this thesis, we use datasets generated using the NumPy seed 2025. Ground-truth values are documented alongside results and plots. The hyperparameter settings used in mSVGD testing can be found in Appendix C.

2 Manifold-constrained Gaussian Process Inference

2.1 Overview of the MAGI Framework

2.1.1 Related Works

Historically, limitations in data collection methods, computational resources, and statistical methodology prevented ordinary differential equations from being used for data fitting in practical settings, relegating them to tools for theoretical understanding. More recently, advances in technology and developments in statistical tools have allowed this to change.

Before the advent of MAGI, there existed several methods for performing inference on dynamical systems, most of which requiring computationally expensive numerical integration or Markov chain Monte Carlo (MCMC) subprocesses. The original MAGI paper discusses some of these methods, showing MAGI to outperform all of them [13]. As such, we will use only MAGI as a performance benchmark.

2.1.2 The MAGI Method

Consider a variable $\boldsymbol{x}(t) = (x_1(t), \ldots, x_D(t)) \in \mathbb{R}^D$ governed by a system of ODEs with parameters $\boldsymbol{\theta} \in \mathbb{R}^p$. We can describe the behavior of $\boldsymbol{x}(t)$ at a time $t \in [0, T]$ with $\boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{\theta}, t)$, the vector containing the derivatives of each component of $\boldsymbol{x}(t)$ with respect to time.

$$\boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{\theta}, t) = \dot{\boldsymbol{x}}(t) = \frac{d\boldsymbol{x}(t)}{dt} = \left(\frac{dx_1(t)}{dt}, \ldots, \frac{dx_D(t)}{dt}\right) \tag{2.1}$$

Manifold-constrained Gaussian Process Inference is a fully Bayesian method [13]. We view the trajectory $\boldsymbol{x}(t)$ as a realization of the stochastic process $\boldsymbol{X}(t) = (X_1(t), \ldots, X_D(t))$ and the model parameters $\boldsymbol{\theta}$ as a realization of the random vector $\boldsymbol{\Theta}$. We also give Bayesian treatment to the sampling standard deviation $\boldsymbol{\sigma}$, if it is unknown. Under the Bayesian paradigm, the primary object used for inference is the posterior distribution, which is obtained by combining the likelihood function of the data with a chosen prior distribution of the unknown random variables.

We impose a general prior distribution $\pi(\cdot)$ on $\boldsymbol{\Theta}$ and independent Gaussian process (GP) priors on each component $X_d(t)$ such that $X_d(t) \sim \mathcal{GP}(\mu_d, \mathcal{K}_d)$, $t \in [0, T]$, where $\mu_d : \mathbb{R} \to \mathbb{R}$ is a mean function and $\mathcal{K}_d : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is a positive definite covariance kernel. Then, by the definition of Gaussian process, for any finite set of time points $\boldsymbol{t}$, $X_d(\boldsymbol{t})$ has a multivariate Normal distribution with mean vector $\mu_d(\boldsymbol{t})$ and covariance matrix $\mathcal{K}_d(\boldsymbol{t}, \boldsymbol{t})$.

Let $\boldsymbol{\tau}_d$ be the set of times that component d is observed and $\boldsymbol{\tau} = (\boldsymbol{\tau}_1, \ldots, \boldsymbol{\tau}_D)$. Denote the observations by $\boldsymbol{y}(\boldsymbol{\tau}) = (\boldsymbol{y}_1(\boldsymbol{\tau}_1), \ldots, \boldsymbol{y}_D(\boldsymbol{\tau}_d))$. Let $N_d = |\boldsymbol{y}_d(\boldsymbol{\tau}_d)|$ be the number of observations in the dth component and $N = N_1 + \cdots + N_D$. If the dth component is not observed, then $N_d = 0$, $\boldsymbol{\tau}_d = \emptyset$, and $\boldsymbol{y}_d(\boldsymbol{\tau}_d) = \emptyset$.

Casting the trajectories X_d as GPs facilitates computation since GPs provide closed-form expressions for $\dot{\boldsymbol{X}}(t)$ and $\boldsymbol{X}(t)$; with a GP prior on $\boldsymbol{X}(t)$, the conditional distribution of $\dot{\boldsymbol{X}}(t)$ is also a GP with its mean function and covariance kernel fully specified. However, this is inherently incompatible with the ODE model because Equation 2.1 also fully specifies $\dot{\boldsymbol{x}}(t)$ given $\boldsymbol{x}(t)$. The key contribution of MAGI addresses this incompatibility by constraining the GPs to satisfy the ODE system described by Equation 2.1.

This constrained is performed using conditional probability. We define a random variable W that quantifies the discrepancy between the derivatives of the stochastic process $\boldsymbol{X}(t)$ and the derivatives yielded by Equation 2.1, at a given value of parameters $\boldsymbol{\theta}$.

$$W = \sup_{t \in [0,T],\, d \in \{1,\ldots,D\}} \left| \dot{X}_d(t) - \boldsymbol{f}(\boldsymbol{X}(t), \boldsymbol{\theta}, t)_d \right| \tag{2.2}$$

Here, $\dot{X}_d(t)$ is the dth component of the derivative of the GP at time t and $\boldsymbol{f}(\boldsymbol{X}(t), \boldsymbol{\theta}, t)_d$ is the dth component of the ODE system at time t. Note that we take the supremum over $t \in [0, T]$ and $d \in \{1, \ldots, D\}$ of their absolute difference, so $W \equiv 0$ if and only if $\dot{\boldsymbol{X}}$ matches $\boldsymbol{f}$ at all times t and in all dimensions d, and thus, if and only if $\boldsymbol{X}$ satisfies the desired ODE system for the given parameters $\boldsymbol{\theta}$.

In theory, we can then write the posterior density of $\boldsymbol{X}(t)$ and $\boldsymbol{\theta}$ given the observations $\boldsymbol{y}(\boldsymbol{\tau})$ and subject to the ODE constraint $W = 0$.

$$P_{\boldsymbol{\Theta}, \boldsymbol{X}(t) \mid W, \boldsymbol{Y}(\boldsymbol{\tau})}\left(\boldsymbol{\theta}, \boldsymbol{x}(t) \mid W = 0, \boldsymbol{Y}(\boldsymbol{\tau}) = \boldsymbol{y}(\boldsymbol{\tau})\right) \tag{2.3}$$

In practice, however, W and $\boldsymbol{X}(t)$ are not generally computable. Therefore, we approximate W and $\boldsymbol{X}(t)$ by considering their values on a finite discretization set. We define $\boldsymbol{I} = (t_1, \ldots, t_n)$ such that $\boldsymbol{\tau} \subseteq \boldsymbol{I}^D$ and $\boldsymbol{I} \subset [0, T]$, and define $n = |\boldsymbol{I}|$ to be the number of discretization points. We then define $W_{\boldsymbol{I}}$, the finite approximation of W.

$$W_{\boldsymbol{I}} = \max_{t \in \boldsymbol{I},\, d \in \{1, \ldots, D\}} \left| \dot{X}_d(t) - \boldsymbol{f}(\boldsymbol{X}(t), \boldsymbol{\theta}, t)_d \right| \tag{2.4}$$

Note that the only difference between W and $W_{\boldsymbol{I}}$ is that we are now taking the maximum over a finite set $t \in \boldsymbol{I}$ rather than the supremum over the continuous interval $[0, T]$. Also, note that $W_{\boldsymbol{I}}$ converges to W monotonically as $\boldsymbol{I}$ becomes dense in $[0, T]$.

Therefore, the practically computable posterior density can be written as

$$P_{\boldsymbol{\Theta}, \boldsymbol{X}(\boldsymbol{I}) \mid W_{\boldsymbol{I}}, \boldsymbol{Y}(\boldsymbol{\tau})}\left(\boldsymbol{\theta}, \boldsymbol{x}(\boldsymbol{I}) \mid W_{\boldsymbol{I}} = 0, \boldsymbol{Y}(\boldsymbol{\tau}) = \boldsymbol{y}(\boldsymbol{\tau})\right) \tag{2.5}$$

which is the joint conditional distribution of $\boldsymbol{\Theta}$ and $\boldsymbol{X}(\boldsymbol{I})$ together. Thus, we simultaneously infer both the parameters $\boldsymbol{\theta}$ and the discretized trajectory $\boldsymbol{X}(\boldsymbol{I})$ from the noisy observations $\boldsymbol{y}(\boldsymbol{\tau})$. We have excluded $\boldsymbol{\sigma}$ from this expression for convenience, but the posterior form would be the same regardless. As we proceed, we will ensure that no terms including $\boldsymbol{\sigma}$ are dropped in order to maintain a valid posterior density for $\boldsymbol{\sigma}$ in cases where it is unknown.

To derive a closed-form analytical expression, we must carefully factor the posterior density using properties of conditional probability.

Remark. We ask the reader to forgive the abuse of notation. Evaluation of density functions will be notated as probabilities for the sake of brevity, despite being statistically invalid.

First, we start with the definition of conditional probability.

$$\begin{aligned} &P_{\boldsymbol{\Theta}, \boldsymbol{X}(\boldsymbol{I}) \mid W_{\boldsymbol{I}}, \boldsymbol{Y}(\boldsymbol{\tau})}\left(\boldsymbol{\theta}, \boldsymbol{x}(\boldsymbol{I}) \mid W_{\boldsymbol{I}} = 0, \boldsymbol{Y}(\boldsymbol{\tau}) = \boldsymbol{y}(\boldsymbol{\tau})\right) \\ &\propto P(\boldsymbol{\Theta} = \boldsymbol{\theta}, \boldsymbol{X}(\boldsymbol{I}) = \boldsymbol{x}(\boldsymbol{I}), W_{\boldsymbol{I}} = 0, \boldsymbol{Y}(\boldsymbol{\tau}) = \boldsymbol{y}(\boldsymbol{\tau})) \end{aligned} \tag{2.6}$$

We then factor the expression into computable parts.

$$\begin{aligned} P(\boldsymbol{\Theta} = \boldsymbol{\theta}, \boldsymbol{X}(\boldsymbol{I}) = \boldsymbol{x}(\boldsymbol{I}), W_{\boldsymbol{I}} = 0, \boldsymbol{Y}(\boldsymbol{\tau}) = \boldsymbol{y}(\boldsymbol{\tau})) = \pi_{\boldsymbol{\Theta}}(\boldsymbol{\theta}) & \\ \times P(\boldsymbol{X}(\boldsymbol{I}) = \boldsymbol{x}(\boldsymbol{I}) \mid \boldsymbol{\Theta} = \boldsymbol{\theta}) & \quad (2.7a) \\ \times P(\boldsymbol{Y}(\boldsymbol{\tau}) = \boldsymbol{y}(\boldsymbol{\tau}) \mid \boldsymbol{X}(\boldsymbol{I}) = \boldsymbol{x}(\boldsymbol{I}), \boldsymbol{\Theta} = \boldsymbol{\theta}) & \quad (2.7b) \\ \times P(W_{\boldsymbol{I}} = 0 \mid \boldsymbol{Y}(\boldsymbol{\tau}) = \boldsymbol{y}(\boldsymbol{\tau}), \boldsymbol{X}(\boldsymbol{I}) = \boldsymbol{x}(\boldsymbol{I}), \boldsymbol{\Theta} = \boldsymbol{\theta}) & \quad (2.7c) \end{aligned}$$

Since $\boldsymbol{X}(\boldsymbol{I})$ and $\boldsymbol{\Theta}$ are independent *a priori*, the first term (2.7a) can be simplified as:

$$P(\boldsymbol{X}(\boldsymbol{I}) = \boldsymbol{x}(\boldsymbol{I}) \mid \boldsymbol{\Theta} = \boldsymbol{\theta}) = P(\boldsymbol{X}(\boldsymbol{I}) = \boldsymbol{x}(\boldsymbol{I})) \quad \text{(2.7a simp.)}$$

The second term (2.7b) corresponds to the observations $\boldsymbol{y}(\boldsymbol{\tau})$. By the model setup, $\boldsymbol{Y}(\boldsymbol{\tau})$ and $\boldsymbol{\Theta}$ are conditionally independent given $\boldsymbol{X}(\boldsymbol{I})$. Then, we can simplify:

$$P(\boldsymbol{Y}(\boldsymbol{\tau}) = \boldsymbol{y}(\boldsymbol{\tau}) \mid \boldsymbol{X}(\boldsymbol{I}) = \boldsymbol{x}(\boldsymbol{I}), \boldsymbol{\Theta} = \boldsymbol{\theta}) = P(\boldsymbol{Y}(\boldsymbol{\tau}) = \boldsymbol{y}(\boldsymbol{\tau}) \mid \boldsymbol{X}(\boldsymbol{I}) = \boldsymbol{x}(\boldsymbol{I})) \quad \text{(2.7b simp.)}$$

Following the definition of $W_{\boldsymbol{I}}$ and using conditional independence, third term (2.7c) can be simplified as follows:

$$\begin{aligned} & P(W_{\boldsymbol{I}} = 0 \mid \boldsymbol{Y}(\boldsymbol{\tau}) = \boldsymbol{y}(\boldsymbol{\tau}), \boldsymbol{X}(\boldsymbol{I}) = \boldsymbol{x}(\boldsymbol{I}), \boldsymbol{\Theta} = \boldsymbol{\theta}) \\ & = P(\dot{\boldsymbol{X}}(\boldsymbol{I}) - \boldsymbol{f}(\boldsymbol{x}(\boldsymbol{I}), \boldsymbol{\theta}, \boldsymbol{I}) = 0 \mid \boldsymbol{Y}(\boldsymbol{\tau}) = \boldsymbol{y}(\boldsymbol{\tau}), \boldsymbol{X}(\boldsymbol{I}) = \boldsymbol{x}(\boldsymbol{I}), \boldsymbol{\Theta} = \boldsymbol{\theta}) \\ & = P(\dot{\boldsymbol{X}}(\boldsymbol{I}) - \boldsymbol{f}(\boldsymbol{x}(\boldsymbol{I}), \boldsymbol{\theta}, \boldsymbol{I}) = 0 \mid \boldsymbol{X}(\boldsymbol{I}) = \boldsymbol{x}(\boldsymbol{I})) \\ & = P(\dot{\boldsymbol{X}}(\boldsymbol{I}) = \boldsymbol{f}(\boldsymbol{x}(\boldsymbol{I}), \boldsymbol{\theta}, \boldsymbol{I}) \mid \boldsymbol{X}(\boldsymbol{I}) = \boldsymbol{x}(\boldsymbol{I})) \quad \text{(2.7c simp.)} \end{aligned}$$

Since $\boldsymbol{X}(\boldsymbol{I})$ is multivariate Gaussian, the first term (2.7a) is multivariate Gaussian. Under Gaussian noise, the second term (2.7b) is the likelihood for independent Gaussians with means described by $\boldsymbol{x}(\boldsymbol{I})$ and variances described by $\boldsymbol{\sigma}^2$. Given $\boldsymbol{X}(\boldsymbol{I})$, we know $\dot{\boldsymbol{X}}(\boldsymbol{I})$ is multivariate Gaussian as long as the covariance kernel $\mathcal{K}$ is twice differentiable, so the third term (2.7c) is also multivariate Gaussian.

Theorem 2.1.2.1. *We can write the practically computable posterior density function as follows. The terms in the function are labeled with the tags of the corresponding expressions derived via the factorization in Equation 2.7.*

$$P_{\boldsymbol{\Theta},\boldsymbol{X}(\boldsymbol{I})|W_{\boldsymbol{I}},\boldsymbol{Y}(\boldsymbol{\tau})}(\boldsymbol{\theta},\boldsymbol{x}(\boldsymbol{I}) \mid W_{\boldsymbol{I}}=0,\boldsymbol{Y}(\boldsymbol{\tau})=\boldsymbol{y}(\boldsymbol{\tau}))$$

$$\propto \pi_{\boldsymbol{\Theta}}(\boldsymbol{\theta}) \times \exp\Bigg(-\frac{1}{2}\sum_{d=1}^{D} \Bigg[$$

$$+ n\log(2\pi) + \log|C_d| + \|x_d(\boldsymbol{I}) - \mu_d(\boldsymbol{I})\|^2_{C_d^{-1}} \tag{2.7a}$$

$$+ N_d \log(2\pi\sigma_d^2) + \|x_d(\boldsymbol{\tau}_d) - y_d(\boldsymbol{\tau}_d)\|^2_{\sigma_d^{-2}} \tag{2.7b}$$

$$+ n\log(2\pi) + \log|K_d| + \left\| \boldsymbol{f}^{\boldsymbol{x},\boldsymbol{\theta}}_{d,\boldsymbol{I}} - \dot{\mu}_d(\boldsymbol{I}) - m_d\{x_d(\boldsymbol{I}) - \mu_d(\boldsymbol{I})\} \right\|^2_{K_d^{-1}} \Bigg]\Bigg) \tag{2.7c}$$

where $\|\boldsymbol{v}\|^2_{A^{-1}} = \boldsymbol{v}^\top A^{-1}\boldsymbol{v}$ is the squared Mahalanobis norm induced by the positive semi-definite matrix A, $n = |\boldsymbol{I}|$ is the cardinality of $\boldsymbol{I}$, and $\boldsymbol{f}^{\boldsymbol{x},\boldsymbol{\theta}}_{d,\boldsymbol{I}}$ denotes the dth component of $\boldsymbol{f}(\boldsymbol{x}(\boldsymbol{I}),\boldsymbol{\theta},\boldsymbol{I})$. In addition, we define the multivariate Gaussian covariance matrices C_d and K_d for each component d:

$$\begin{cases} C_d = \mathcal{K}_d(\boldsymbol{I},\boldsymbol{I}) \\ m_d = {}'\mathcal{K}_d(\boldsymbol{I},\boldsymbol{I})\,\mathcal{K}_d(\boldsymbol{I},\boldsymbol{I})^{-1} \\ K_d = \mathcal{K}''_d(\boldsymbol{I},\boldsymbol{I}) - {}'\mathcal{K}_d(\boldsymbol{I},\boldsymbol{I})\,\mathcal{K}_d(\boldsymbol{I},\boldsymbol{I})^{-1}\,\mathcal{K}'_d(\boldsymbol{I},\boldsymbol{I}) \end{cases} \tag{2.8}$$

where $\mathcal{K}_d(s,t)$ is the covariance kernel of the GP prior for the dth component and the partial derivatives of $\mathcal{K}_d$ are denoted:

$$\begin{cases} {}'\mathcal{K}_d = \frac{\partial}{\partial s}\mathcal{K}_d(s,t) \\ \mathcal{K}'_d = \frac{\partial}{\partial t}\mathcal{K}_d(s,t) \\ \mathcal{K}''_d = \frac{\partial^2}{\partial s \partial t}\mathcal{K}_d(s,t) \end{cases} \tag{2.9}$$

In practice, we choose the Matern kernel for $\mathcal{K}_d$.

$$\mathcal{K}_d(s,t) = \phi_1 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu}\frac{|s-t|}{\phi_2} \right)^{\nu} B_\nu \left(\sqrt{2\nu}\frac{|s-t|}{\phi_2} \right) \tag{2.10}$$

where Γ is the Gamma function and B_ν is the modified Bessel function of the second kind. The Matern kernel is twice differentiable for degrees of freedom $\nu > 2$, so we choose $\nu = 2.01$. The two remaining hyperparameters, ϕ_1 and ϕ_2, are fit separately for each component, so each has a different kernel for its GP prior. Their meaning and specification are discussed in Section 2.2. An alternative fitting method is proposed in Section 5.2.

Prior Tempering

By Bayes' rule, the posterior density in Equation 2.3 can also be factored as:

$$\begin{aligned}
&P_{\boldsymbol{\Theta},\boldsymbol{X}(\boldsymbol{I})|W_{\boldsymbol{I}},\boldsymbol{Y}(\boldsymbol{\tau})}\left(\boldsymbol{\theta},\boldsymbol{x}(\boldsymbol{I}) \mid W_{\boldsymbol{I}}=0, \boldsymbol{Y}(\boldsymbol{\tau})=\boldsymbol{y}(\boldsymbol{\tau})\right) \\
&\propto P(\boldsymbol{y}(\boldsymbol{\tau}) \mid \boldsymbol{\Theta}=\boldsymbol{\theta}, \boldsymbol{X}(\boldsymbol{I})=\boldsymbol{x}(\boldsymbol{I}), W_{\boldsymbol{I}}=0) \times P(\boldsymbol{\theta},\boldsymbol{x}(\boldsymbol{I}) \mid W_{\boldsymbol{I}}=0) \\
&= P(\boldsymbol{y}(\boldsymbol{\tau}) \mid \boldsymbol{X}(\boldsymbol{I})=\boldsymbol{x}(\boldsymbol{I})) \times P(\boldsymbol{\theta},\boldsymbol{x}(\boldsymbol{I}) \mid W_{\boldsymbol{I}}=0) \\
&= P(\boldsymbol{y}(\boldsymbol{\tau}) \mid \boldsymbol{X}(\boldsymbol{\tau})=\boldsymbol{x}(\boldsymbol{\tau})) \times P(\boldsymbol{\theta},\boldsymbol{x}(\boldsymbol{I}) \mid W_{\boldsymbol{I}}=0)
\end{aligned} \tag{2.11}$$

As the cardinality $|\boldsymbol{I}|$ increases with more discretization points, the likelihood part $P(\boldsymbol{y}(\boldsymbol{\tau}) \mid \boldsymbol{X}(\boldsymbol{\tau})=\boldsymbol{x}(\boldsymbol{\tau}))$ stays unchanged, but the prior part $P(\boldsymbol{\theta},\boldsymbol{x}(\boldsymbol{I}) \mid W_{\boldsymbol{I}}=0)$ grows and tends to dominate the expression. To address this, a tempering hyperparameter β is introduced to counterbalance the influence of the prior on the posterior. Then, we modify the analytical form of the posterior in Theorem 2.1.2.1 as follows.

$$P^{(\beta)}_{\boldsymbol{\Theta},\boldsymbol{X}(\boldsymbol{I})|W_{\boldsymbol{I}},\boldsymbol{Y}(\boldsymbol{\tau})}(\boldsymbol{\theta},\boldsymbol{x}(\boldsymbol{I}) \mid W_{\boldsymbol{I}}=0, \boldsymbol{Y}(\boldsymbol{\tau})=\boldsymbol{y}(\boldsymbol{\tau})) \tag{2.12}$$

$$\propto P(\boldsymbol{y}(\boldsymbol{\tau}) \mid \boldsymbol{X}(\boldsymbol{\tau})=\boldsymbol{x}(\boldsymbol{\tau})) \times P(\boldsymbol{\theta},\boldsymbol{x}(\boldsymbol{I}) \mid W_{\boldsymbol{I}}=0)^{1/\beta}$$

$$\propto \pi_{\boldsymbol{\Theta}}(\boldsymbol{\theta}) \times \exp\Bigg(-\frac{1}{2}\sum_{d=1}^{D}\Bigg[\frac{1}{\beta}\left\|x_d(\boldsymbol{I})-\mu_d(\boldsymbol{I})\right\|^2_{C_d^{-1}} \tag{2.7a$^{(\beta)}$}$$

$$+\left(N_d\log(2\pi\sigma_d^2)+\left\|x_d(\boldsymbol{\tau}_d)-y_d(\boldsymbol{\tau}_d)\right\|^2_{\sigma_d^{-2}}\right) \tag{2.7b}$$

$$+\frac{1}{\beta}\left\|\boldsymbol{f}^{\boldsymbol{x},\boldsymbol{\theta}}_{d,\boldsymbol{I}}-\dot{\mu}_d(\boldsymbol{I})-m_d\{x_d(\boldsymbol{I})-\mu_d(\boldsymbol{I})\}\right\|^2_{K_d^{-1}}\Bigg]\Bigg) \tag{2.7c$^{(\beta)}$}$$

The MAGI paper [13] recommends setting $\beta = Dn/N$, where D is the number of system components, $n = |\boldsymbol{I}|$ is the number of discretization time points, and $N = \sum_{d=1}^{D} N_d$ is the total number of observations. Heuristically, this setting balances the likelihood contribution from the total number of observations with the prior contribution from the total number of discretization points across all of the system components.

Remark. Setting $\beta = 1$ reduces this tempered posterior density to the original posterior density in Theorem 2.1.2.1.

2.2 MAGI Computation

In this section, we review the computation methods outlined by the MAGI paper [13], which are implemented in its accompanying code package [11]. With the posterior distribution specified in Theorem 2.1.2.1 and its augmentation in Equation 2.12, Yang et al. propose the use of Hamiltonian Monte Carlo (HMC) [2] to obtain samples of $\boldsymbol{X}(\boldsymbol{I})$ and $\boldsymbol{\Theta}$ together, in addition to any unknown sampling standard deviations in $\boldsymbol{\sigma}$. The posterior means of the samples are used as point estimates for the trajectories and parameters.

2.2.1 Hamiltonian Monte Carlo

"The king's vehicle picks a random direction, either north or south, and drives off at a random momentum. As the vehicle goes uphill, it slows down and turns around when its declining momentum forces it to. Then it picks up speed again on the way down. After a fixed period of time, they stop the vehicle, get out, and start shaking hands and kissing babies. Then they get back in the vehicle and begin again. Amazingly, [the king's advisor] can prove mathematically that this procedure guarantees that, in the long run, the locations visited will be inversely proportional to their relative elevations, which are also inversely proportional to the population densities."

— Richard McElreath [9]

In this section, we will give a high-level overview of the Hamiltonian Monte Carlo (HMC) sampling scheme. The interested reader may refer to Ref. [2] or Ref. [9] for a more thorough introduction.

Suppose we have some unknown variables of interest $\boldsymbol{z}$. HMC is a Markov chain Monte Carlo (MCMC) method that gives the physical interpretation to the current state of $\boldsymbol{z}$ as the position of a small, frictionless particle. The log-density of $\boldsymbol{z}$ gives the "particle" a surface to glide across, allowing the algorithm to leverage Hamiltonian mechanics to simulate the movement of the particle through a potential energy landscape.

After starting $\boldsymbol{z}$ at an initial position, HMC introduces auxiliary momentum variables $\boldsymbol{p}$ for each element of $\boldsymbol{z}$. The negative log-density of $\boldsymbol{z}$ has the physical interpretation of the "potential energy" at "position" $\boldsymbol{z}$, while $K(\boldsymbol{p}) = \boldsymbol{p}^T\boldsymbol{p}/2$ is analogous to the "kinetic energy."

Particle movements are proposed by leapfrog steps, a numerical integration method used to simulate the Hamiltonian dynamics which ensures that the sampler follows a path that conserves the energy of the system. This allows for smarter and larger steps when compared to "classical" MCMC methods such as Metropolis-Hastings. In practice, HMC is only approximating the smooth path of a particle, so a Metropolis-Hastings style acceptance-rejection step is introduced at each iteration to correct for any discretization errors introduced by the leapfrog integration.

For MAGI, $\boldsymbol{z} = (\boldsymbol{X}(\boldsymbol{I}), \boldsymbol{\Theta}, \boldsymbol{\sigma})$. The leapfrog method hyperparameters are tuned automatically to achieve an acceptance rate between 60% and 90%. More details regarding the application of HMC for MAGI can be found in the Supplementary Information section of Ref. [13]. For the purposes of this thesis, the most important aspect is the initialization for $\boldsymbol{X}(\boldsymbol{I})$, $\boldsymbol{\Theta}$, and $\boldsymbol{\sigma}$, which will be discussed in detail in the following subsections.

2.2.2 Initialization for Observed Components

Setting ϕ_1, ϕ_2, σ_d for Observed Components

Each trajectory component X_d has an independent GP prior, $X_d(t) \sim \mathcal{GP}(\mu_d, \mathcal{K}_d)$. The mean function μ_d can be user-specified, with the default $\mu_d(t) = 0$. The Matern kernel $\mathcal{K}$ (Equation 2.10) has three hyperparameters: ϕ_1, ϕ_2, and ν. As discussed in Section 2.1, we set $\nu = 2.01$ so that the kernel is twice differentiable. The two remaining hyperparameters, ϕ_1 and ϕ_2, are set during the initialization stage and held fixed during sampling. ϕ_1 controls the overall variance level of the GP, while ϕ_2 controls the bandwidth for how much neighboring points of the GP affect each other.

When the observation noise level σ_d is unknown, $(\phi_1, \phi_2, \sigma_d)$ are jointly fit for each component d by maximizing GP fitting without conditioning on any ODE information.

$$\begin{aligned}(\tilde{\phi}_1, \tilde{\phi}_2, \tilde{\sigma}_d) &= \underset{\phi_1,\phi_2,\sigma_d}{\arg\max}\, P(\phi_1, \phi_2, \sigma_d \mid \boldsymbol{y}_d(\boldsymbol{I}_0)) \\ &= \underset{\phi_1,\phi_2,\sigma_d}{\arg\max}\, \pi_{\Phi_1}(\phi_1)\, \pi_{\Phi_2}(\phi_2)\, \pi_{\sigma_d}(\sigma_d)\, P(\boldsymbol{y}_d(\boldsymbol{I}_0) \mid \phi_1, \phi_2, \sigma_d)\end{aligned} \tag{2.13}$$

where $\boldsymbol{y}_d(\boldsymbol{I}_0) \mid \phi_1, \phi_2, \sigma_d \sim \mathcal{N}(0, \mathcal{K}_\phi(\boldsymbol{I}_0, \boldsymbol{I}_0) + \operatorname{diag}_{|\boldsymbol{I}_0|}(\sigma_d^2))$. The index set $\boldsymbol{I}_0$ is the smallest, evenly spaced set such that all observation time points in the component d are in $\boldsymbol{I}_0$, i.e. $\boldsymbol{\tau}_d \subseteq \boldsymbol{I}_0$. The priors $\pi_{\Phi_1}(\phi_1)$ and $\pi_{\sigma_d}(\sigma_d)$ for the GP variance parameter and observation noise standard deviation are set to be flat.

The prior $\pi_{\Phi_2}(\phi_2)$ for the GP bandwidth parameter is set to be a Gaussian distribution. The mean μ_{ϕ_2} is set to be half of the period corresponding to the frequency that is the weighted average of all of the frequencies in the Fourier transform of $\boldsymbol{y}_d$ on $\boldsymbol{I}_0$, with the values at unobserved time linearly interpolated from the observation values at $\boldsymbol{\tau}_d$. The weight on a given frequency is the squared modulus of the Fourier transform with that frequency. The standard deviation is set such that the maximum time T is three SDs away from μ_{ϕ_2}. The Gaussian prior on ϕ_2 prevents it from being too large.

In subsequent sampling of $(\boldsymbol{X}(\boldsymbol{I}), \boldsymbol{\Theta}, \boldsymbol{\sigma})$, the hyperparameters (ϕ_1, ϕ_2) are fixed at $(\tilde{\phi}_1, \tilde{\phi}_2)$, while $\tilde{\sigma}_d$ gives the initial state of σ_d for performing Bayesian inference.

If σ_d is known, then the hyperparameters ϕ_1 and ϕ_2 are fitted by themselves by maximizing a similar objective function with a fixed σ_d.

$$\begin{aligned}(\tilde{\phi}_1, \tilde{\phi}_2) &= \arg\max_{\phi_1,\phi_2} P(\phi_1, \phi_2 \mid \boldsymbol{y}_d(\boldsymbol{I}_0), \sigma_d) \\ &= \arg\max_{\phi_1,\phi_2} \pi_{\Phi_1}(\phi_1)\, \pi_{\Phi_2}(\phi_2)\, P(\boldsymbol{y}_d(\boldsymbol{I}_0) \mid \phi_1, \phi_2, \sigma_d)\end{aligned} \tag{2.14}$$

The priors for ϕ_1 and ϕ_2 are the same as previously defined.

Initialization of $\boldsymbol{X}(\boldsymbol{I})$ for Observed Components

To provide starting values for $\boldsymbol{X}_d(\boldsymbol{I})$, we use the observed values $\boldsymbol{y}_d(\boldsymbol{\tau})$ and linearly interpolate the remaining points in the discretization set $\boldsymbol{I}$.

Initialization of $\boldsymbol{\Theta}$ for When All Components Are Observed

To provide starting values for $\boldsymbol{\Theta}$ when *all* components are observed, we optimize the full posterior from Theorem 2.1.2.1 as a function of $\boldsymbol{\theta}$ alone, holding $\boldsymbol{X}(\boldsymbol{I})$ and $\boldsymbol{\sigma}$ fixed at their initial values and $\boldsymbol{\phi}$ fixed at its fitted value.

2.2.3 Initialization for Unobserved Components

Separate treatment is required for setting $\boldsymbol{\phi}$ and initializing $(\boldsymbol{X}(\boldsymbol{I}), \boldsymbol{\Theta})$ for unobserved component(s), i.e. for components $d \in \{1, \ldots, D\}$ such that $\boldsymbol{y}_d(\boldsymbol{\tau}_d) = \emptyset$. In practice, we often need more than one or two observations to perform initialization tasks, so it may be necessary to treat any component with a sufficiently small $|\boldsymbol{y}_d(\boldsymbol{\tau}_d)|$ as unobserved.

We use an optimization procedure to maximize the full posterior from Theorem 2.1.2.1 as a function of $\boldsymbol{\theta}$, the whole curve of $\boldsymbol{X}(\boldsymbol{I})$ for unobserved components, and the corresponding $\boldsymbol{\phi}$. We hold the values of $\boldsymbol{\sigma}$, $\boldsymbol{\phi}$, and $\boldsymbol{X}(\boldsymbol{I})$ for the observed components fixed at their initial values, which are computed as discussed in the previous sections. The values of $\boldsymbol{\phi}$ are fixed at the optimized values, and the optimize values of $\boldsymbol{\theta}$ and $\boldsymbol{X}(\boldsymbol{I})$ are used as starting values. Note that we do not have to fit σ_d for unobserved components, since the sampling variance does not exist for components that are not sampled.

One might notice that this fitting procedure is quite intensive, both conceptually and computationally, particularly for a large number of unobserved components and a large number of discretization points $n = |\boldsymbol{I}|$. The gradient of the Matern kernel is also quite troublesome to work with. As such, we propose a less computationally expensive initialization method for unobserved components in Section 5.2.

3 Variational Inference

3.1 Computational Challenges of MAGI and HMC

High-dimensional models are commonplace under the MAGI framework. Not only do we have to consider the number of parameters in $\boldsymbol{\theta}$ and the number of dimensions D in the dynamical system, we also need to consider the density of the discretization set $\boldsymbol{I}$, which we would like to be as fine as computationally feasible, as discussed in Section 2.1. Then, the space of inferred values in MAGI has dimensionality $|\boldsymbol{\theta}| + D \cdot |\boldsymbol{I}| + |\boldsymbol{\sigma}_{\text{unknown}}|$, which can easily be in the hundreds or thousands.

Although Hamiltonian Monte Carlo is a highly effective sampling algorithm, it has some limitations that warrant the exploration of alternate methods. Particularly in higher dimensions, HMC may run into problems that can incur significant computational cost or make an inference task completely intractable. We will discuss a handful below.

1. *Tuning Parameters.* HMC requires careful tuning of several parameters, such as the leapfrog step size, number of steps to make, and acceptance probability. Poor choices can lead to inefficient sampling or even divergence. In high dimensions, finding optimal parameters becomes more challenging due to the increased complexity of the target distribution and parameter space. For MAGI, [12], these parameters are tuned automatically during a burn-in period, which can be a costly process.

2. *Curse of Dimensionality.* As the dimensionality of the parameter space increases, the size of the space grows exponentially, making it harder for HMC to explore the entire space efficiently. Although HMC makes larger and more informed steps than classical methods such as Metropolis-Hastings, it is not immune to the curse of dimensionality, which can lead to high autocorrelation, slow exploration, inefficient sampling, or failure to converge. The consequences of these phenomena include inaccurate estimates, high variance in estimates, and longer computation time.

3. *Difficulty Exploring Parameter Space.* HMC can struggle to explore high-dimensional parameters spaces for a variety of reasons.

    - Highly correlated parameters in high dimensions can create complex geometries that are difficult for HMC to navigate, leading to inefficient sampling.
    - HMC tends to be sensitive to the initial conditions, especially in high dimensions. The process by which MAGI computes initial conditions is outlined in Section 2.1.2; this process may not yield a sufficiently intelligent starting point, particularly in high dimensions with sparse and noisy data.
    - While HMC is good at global exploration, it can sometimes get stuck in local modes, particularly in high-dimensional spaces with complex, multi-modal distributions.

4. *Numerical Stability.* The leapfrog integration of Hamiltonian dynamics is a numerical method that can become unstable in high dimensions, especially if the step size is not chosen carefully. This can lead to errors in the simulation and poor sampling.

To ensure that these problems do not negatively impact the inference from the MAGI method, careful — and more importantly, slow — computations are required while running HMC. Additionally, like most MCMC methods, HMC is a highly sequential algorithm. This induces a speed bottleneck by heavily limiting HMC's ability to leverage parallel computing methods, such as GPU acceleration. Therefore, if we would like to work with higher-dimensions, it is natural that we should desire methods that are faster and more robust to high dimensionality. To address this challenge, we turn to variational inference.

3.2 Variational Inference

"It appears to be a quite general principle that, whenever there is a randomized way of doing something, then there is a nonrandomized way that delivers better performance but requires more thought."

— Edwin Thompson Jaynes

Variational inference (VI) methods are a family of techniques for approximating probability densities and associated integrals, which are often intractable. Variational inference is widely used to approximate posterior densities for Bayesian models as an alternative to MCMC sampling methods. While MCMC provides a numerical approximation to the exact posterior using a set of samples, VI provides an analytical solution to an approximation of the posterior. VI tends to be faster and easier to scale to large data than MCMC, though the statistical properties of MCMC methods have been studied more thoroughly and are better understood. In this section, we give a brief introduction of the VI class of methods. The thorough review published by Blei, Kucukelbir, and McAuliffe in 2017 is highly recommended for the interested reader [1].

To set up a general problem, suppose we have some unknown variables of interest $\boldsymbol{z}$ and observed data $\boldsymbol{y}$. In Bayesian models, the object of interest is the posterior density $p(\boldsymbol{z} \mid \boldsymbol{y})$.

When conducting MCMC, we construct an ergodic Markov chain on $\boldsymbol{z}$ whose stationary distribution is $p(\boldsymbol{z} \mid \boldsymbol{y})$, then sample from the chain to collect samples from the stationary distribution. However, there exist problems for which MCMC methods are not feasible, such as when datasets are very large or models are too complex. In such settings, VI provides a good alternative.

Rather than using sampling, variational inference recasts learning a probability density as an optimization task. First, we define a variational family $\mathcal{Q}$, which is a set of densities over the unknown variables $\boldsymbol{z}$. Then, we try to find the distribution q^* in $\mathcal{Q}$ that is the closest to the true posterior with respect to Kullback-Leibler (KL) divergence.

$$q^*(\boldsymbol{z}) = \underset{q \in \mathcal{Q}}{\arg\min}\, \mathcal{D}_{\mathrm{KL}}(q(\boldsymbol{z}) \mid\mid p(\boldsymbol{z} \mid \boldsymbol{y})) \tag{3.1}$$

where

$$\begin{aligned} \mathcal{D}_{\mathrm{KL}}(q \mid\mid p) &= \int_z q(z) \log \frac{q(z)}{p(z)}\, dz \\ &= \mathbb{E}_{z \sim q}\left[\log \frac{q(z)}{p(z)}\right] \\ &= \mathbb{E}_{z \sim q}[\log q(z)] - \mathbb{E}_{z \sim q}[\log p(z)] \end{aligned}$$

We use the optimized member q^*, which we call the "variational distribution," to approximate the true posterior. Since we restrict the variational distribution to the variational family $\mathcal{Q}$, the complexity of the optimization is governed by the reach of the pre-defined set of candidate densities. One of the key challenges of variational inference is to choose an appropriate family $\mathcal{Q}$. Generally, it is desirable for $\mathcal{Q}$ to be flexible enough to capture a variational density close to $p(\boldsymbol{z} \mid \boldsymbol{y})$, but simple enough for the optimization to be both tractable and efficient.

Many variational inference methods require some parametric assumptions. For example, Gaussian approximation VI sets $\mathcal{Q}$ to be the family of $|\boldsymbol{z}|$-dimensional multivariate Gaussian distributions, so $q \sim \mathcal{N}_{|\boldsymbol{z}|}(\mu, \Sigma)$ for some mean vector μ and covariance matrix Σ. Then, μ and Σ are optimized to minimize the KL divergence between q and the target density p. Heuristically, the quality of this approximation depends on how "close" p is to a Gaussian distribution, thus making this a relatively restrictive choice of $\mathcal{Q}$.

Mean field approximation is another well known VI method, under which the variational distribution is assumed to factorize over some partition of the components of $\boldsymbol{z}$.

$$q(\boldsymbol{z}) = \prod_{j=1}^{p} q_j(\boldsymbol{z}_j)$$

The distribution for each partition j that minimizes KL divergence satisfies

$$\log q_j^*(z_j) = \mathbb{E}_{q_{-j}^*}[\log p(z, y)] + c$$

where the expectation is taken over the "best" distributions of the partitions other than j, which usually belong to named distribution families, $\log p(z, y)$ is the true joint density of the unknown variables and the data, and c is a normalizing constant that can usually be recovered by pattern matching. Generally, the system of q_j^* equations cannot be solved directly, but an iterative algorithm can be used to find a solution.

Often times, q_j take the forms of known, named distributions, whose expectations are known in closed form. Thus, $\mathcal{Q}$ is implicitly defined to be the product space of these distribution families. Combined with the factorization approximation, this may be fine in simpler models, but is often too restrictive in higher dimensions.

3.3 VI Challenges and Solutions

In implementing variational inference for MAGI, our primary goals are to increase inference speed while maintaining inference accuracy and to ensure that our work is accessible to an end user. As such, there are several challenges that we must consider.

Many VI algorithms, such as the examples of Gaussian approximation and mean field approximation in the previous section, require specialized mathematical derivations and optimization algorithms for each unique model. This is undesirable for our use case, since it is unreasonable to assume that an end user would have both the knowledge and willingness to take these steps. So, our VI algorithm should have the ability to be implemented behind the scenes, abstracted away from the user interface.

Naturally, the next step for us is to do the complex derivations based on the target density defined in Theorem 2.1.2.1, leaving the end user to work with only the resulting algorithm. However, the posterior density is not as defined as it appears. The ODE system defined by $\boldsymbol{f}$ is different for every model, so the target posterior is not a single distribution, nor even a single family of distributions. Thus, for many classical variational inference techniques, we often run into one of two problems when applying them to MAGI. First, the restriction on the variational family $\mathcal{Q}$ may be too strict, resulting in highly varying expressiveness and ability to capture a good approximation, depending strongly on the ODE system chosen and the induced posterior density. Second, a general optimization procedure may not exist for for a more flexible and expressive variational family $\mathcal{Q}$, making the problem intractable for our broad class of potential MAGI posteriors.

Hence, we desire a general-purpose VI algorithm that allows for a highly flexible variational family $\mathcal{Q}$ and a straight-forward algorithm that can solve the optimization problem with minimal user input. The solution that we explore in this thesis is particle-based variational inference (ParVI). As our starting point, we look to the ParVI algorithm introduced by Liu and Wang in 2016: Stein variational gradient descent [8].

4

Stein Variational Gradient Descent

4.1 Particle-based Variational Inference

Unlike most classical variational inference methods, ParVI methods usually do not make distributional assumptions nor seek to optimize a set of parameters. Instead, similar to MCMC techniques, ParVI can be viewed as a class of sampling methods that attempt to approximate the target distribution with a fixed number of samples, which are canonically called "particles." However, unlike the stochasticity that powers MCMC, ParVI deterministically updates the particles to minimize the difference between the target distribution and the distribution represented by the collection of particles, typically with respect to KL divergence.

The use of particles is more flexible and thus more accurate than the parametric approximations of classical variational inference methods. Additionally, since the particles jointly approximate the target distribution, they must "interact" with each other during the optimization process and "spread out" through the target density, thus appearing negatively correlated. So, ParVI also mitigates the sampling inefficiency that often arises in MCMC methods due to autocorrelation in Markov chain samples.

4.2 Overview of the SVGD Method

In this section, we review Stein variational gradient descent, the ground-breaking and seminal ParVI algorithm introduced by Liu and Wang in 2016 [8]. The detailed mathematics behind the technique are not necessary for understanding the methods and results of this thesis, but the resulting algorithm serves as the basis of our work.

Background: Reproducing Kernel Hilbert Space

Let x be a continuous random variable of interest, taking values in $\mathcal{X} = \mathbb{R}^d$, and let $\mathcal{K}(x, x') : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a positive definite kernel. The reproducing kernel Hilbert space (RKHS) $\mathcal{H}$, induced by the kernel $\mathcal{K}$, is the closure of the linear span

$$\left\{ f : f(x) = \sum_{i=1}^{m} a_i \mathcal{K}(x, x_i),\ a_i \in \mathbb{R},\ m \in \mathbb{N},\ x_i \in \mathcal{X} \right\} \tag{4.1}$$

equipped with the inner product

$$\langle f, g \rangle_{\mathcal{H}} = \sum_{ij} a_i b_j \mathcal{K}(x_i, x_j), \text{ for } g(x) = \sum_i b_i \mathcal{K}(x, x_i) \tag{4.2}$$

Let $\mathcal{H}^d$ denote the space of vector functions $\mathbf{f} = [f_1, \ldots, f_d]$ with $f_i \in \mathcal{H}$, equipped with inner product

$$\langle \mathbf{f}, \mathbf{g} \rangle_{\mathcal{H}^d} = \sum_{i=1}^{d} \langle f_i, g_i \rangle_{\mathcal{H}} \tag{4.3}$$

Background: Stein's Identity and Kernelized Stein Discrepancy

Let $p(x)$ be a continuously differentiable target density supported on $\mathcal{X} \subseteq \mathbb{R}^d$, and $\boldsymbol{\phi}(x) = [\phi_1(x), \ldots, \phi_d(x)]^\top$ be a smooth vector function. We define the Stein operator $\mathcal{A}_p$, which acts on the function $\boldsymbol{\phi}$.

$$\mathcal{A}_p \boldsymbol{\phi}(x) = \boldsymbol{\phi}(x) \nabla_x \log p(x)^\top + \nabla_x \boldsymbol{\phi}(x)$$

Stein's identity states that, for sufficiently regular functions $\boldsymbol{\phi}$, we have

$$\mathbb{E}_{x \sim p}[\mathcal{A}_p \boldsymbol{\phi}(x)] = 0.$$

This identity can be checked using integration by parts, assuming mild zero boundary conditions on $\boldsymbol{\phi}$, either: $\boldsymbol{\phi}(x)p(x) = 0\ \forall x \in \partial\mathcal{X}$ when $\mathcal{X}$ is compact, or

$\lim_{\|x\|\to\infty} \boldsymbol{\phi}(x)p(x) = 0$ when $\mathcal{X} = \mathbb{R}^d$. We say that the function $\boldsymbol{\phi}$ is in the Stein class of p if Stein's identity holds.

Now, let $q(x)$ be a different smooth density, also supported on $\mathcal{X}$. The expectation of $\mathcal{A}_p\boldsymbol{\phi}(x)$ under $x \sim q$, $\mathbb{E}_{x\sim q}[\mathcal{A}_p\boldsymbol{\phi}(x)]$, is no longer equal to zero for general $\boldsymbol{\phi}$. Instead, the magnitude of $\mathbb{E}_{x\sim q}[\mathcal{A}_p\boldsymbol{\phi}(x)]$ relates to how different the distributions p and q are, and can thus be used to define a discrepancy measure. *Stein discrepancy* considers the "maximum violation of Stein's identity" among functions $\boldsymbol{\phi}$ in some proper function set $\mathcal{F}$.

$$\text{SteinD}(q,p) = \max_{\boldsymbol{\phi}\in\mathcal{F}} \{[\mathbb{E}_{x\sim q}\text{tr}\,(\mathcal{A}_p\boldsymbol{\phi}(x))]^2\}$$

The choice of the function set $\mathcal{F}$ determines the discriminative power and computational tractability of Stein discrepancy. Traditionally, $\mathcal{F}$ is taken to be sets of functions with bounded Lipschitz norms, which casts a challenging functional optimization problem that is often computationally intractable.

Kernelized Stein discrepancy (KSD) [7] bypasses this difficulty by maximizing $\boldsymbol{\phi}$ in the unit ball of a reproducing kernel Hilbert space, for which the optimization has a closed form solution. KSD is defined as

$$\mathbb{S}(q,p) = \max_{\boldsymbol{\phi}\in\mathcal{H}^d} \{[\mathbb{E}_{x\sim q}\text{tr}\,(\mathcal{A}_p\boldsymbol{\phi}(x))]^2, \text{ s.t. } \|\boldsymbol{\phi}\|_{\mathcal{H}^d} \leq 1\} \tag{4.4}$$

where we assume the kernel $\mathcal{K}(x,x')$ of the RKHS $\mathcal{H}$ is in the Stein class of p as a function of x for any fixed $x' \in \mathcal{X}$.

Theorem 4.2.0.1. *The $\boldsymbol{\phi} \in \mathcal{H}^d$ that maximizes $\{[\mathbb{E}_{x\sim q}\text{tr}\,(\mathcal{A}_p\boldsymbol{\phi}(x))]^2, \text{ s.t. } \|\boldsymbol{\phi}\|_{\mathcal{H}^d} \leq 1\}$ is*

$$\boldsymbol{\phi}(x) = \frac{\boldsymbol{\phi}^*_{q,p}(x)}{\|\boldsymbol{\phi}^*_{q,p}\|_{\mathcal{H}^d}}, \text{ where } \boldsymbol{\phi}^*_{q,p}(\cdot) = \mathbb{E}_{x\sim q}[\mathcal{A}_p\mathcal{K}(x,\cdot)] \tag{4.5}$$

with the KSD evaluating to

$$\mathbb{S}(q,p) = \|\boldsymbol{\phi}^*_{q,p}\|^2_{\mathcal{H}^d} \tag{4.6}$$

Remark. See Appendix A.1 for a proof of this theorem.

It can be shown that $\mathbb{S}(q,p)$ equals zero (and equivalently $\boldsymbol{\phi}^*_{q,p}(x) \equiv 0$) if and only if $p = q$, as long as $\mathcal{K}(x,x')$ is strictly positive definite in a proper sense, which is satisfied by many commonly used kernels. Note that the radial basis function (RBF) kernel, $\mathcal{K}(x,x') = \exp(-\frac{1}{h}\|x - x'\|_2^2)$, is both positive definite and in the Stein class of smooth densities supported on $\mathcal{X} = \mathbb{R}^d$ because of its decaying property.

An important observation is that both the Stein operator and KSD only depend on the target p through the score function $\nabla_x \log p(x)$, which can be calculated from the density p up to proportionality, without requiring normalization constants.

4.2.1 Variational Inference Using Smooth Transforms

As discussed in Section 3.2, the goal of variational inference is to approximate the target distribution $p(x) = \tilde{p}(x)/Z$, where $\tilde{p}$ is an unnormalized density and Z is a normalizing constant, using a simpler variational distribution $q^*(x)$ found in a predetermined variational family $\mathcal{Q}$ by minimizing the KL divergence between q^* and p.

$$q^* = \underset{q \in \mathcal{Q}}{\arg\min} \{\mathcal{D}_{\mathrm{KL}}(q \mid\mid p) \equiv \mathbb{E}_{x \sim q}[\log q(x)] - \mathbb{E}_{x \sim q}[\log \tilde{p}(x)] + \log Z\} \tag{4.7}$$

For SVGD, we let $\mathcal{Q}$ be the set of distributions obtained by smooth transforms from a tractable reference distribution. In other words, $\mathcal{Q}$ is the set of distributions of random variables of the form $z = T(x)$, where $T : \mathcal{X} \to \mathcal{X}$ is a smooth, one-to-one transform, and x is drawn from a tractable reference distribution $x \sim q_0$.

We need to further restrict the set of transforms $\mathcal{T}$ to make the variational optimization practically solvable. One approach is to consider functions T with a certain parametric form and optimize the corresponding parameters, but this introduces the challenge of selecting an appropriate parametric family to balance accuracy, tractability, and solvability.

Instead, Stein Variational Gradient Descent iteratively constructs incremental, additive transforms that effectively perform steepest descent on $\mathcal{T}$ in an RKHS. This method does not require explicitly specifying parametric forms, nor calculating the Jacobian matrix for T. It also gives rise to a simple form that mimics classical gradient descent, making it easily implementable and interpretable, as well as scalable with all the tools already developed for gradient descent.

Stein Operator as the Derivative of KL Divergence

Consider an incremental transform formed by a small perturbation of the identity map:

$$T(x) = x + \epsilon \boldsymbol{\phi}(x) \tag{4.8}$$

where $\boldsymbol{\phi}(x)$ is a smooth function that characterizes the perturbation direction and the scalar $\epsilon \geq 0$ represents the perturbation magnitude. When ϵ is sufficiently small, the Jacobian of T is full rank (close to the identity matrix), and T is guaranteed to be a one-to-one map by the inverse function theorem.

Theorem 4.2.1.1. *Let p be the target density. Let $z = T(x) = x + \epsilon \boldsymbol{\phi}(x)$ and $q_{[T]}(z)$ the density of z when $x \sim q(x)$. Then, the gradient with respect to ϵ of the KL divergence between $q_{[T]}$ and p can be related to the Stein operator as follows.*

$$\nabla_\epsilon \mathcal{D}_{KL}(q_{[T]} \mid\mid p) \mid_{\epsilon=0} = -\mathbb{E}_{x\sim q}[\operatorname{tr}(\mathcal{A}_p \boldsymbol{\phi}(x))]$$

where $\mathcal{A}_p \boldsymbol{\phi}(x) = \nabla_x \log p(x) \boldsymbol{\phi}(x)^\top + \nabla_x \boldsymbol{\phi}(x)$ *is the Stein operator.*

Remark. See Appendix A.2 for a proof of this theorem.

To find the optimal perturbation direction that gives the steepest descent in KL divergence within the zero-centered balls of $\mathcal{H}^d$, we want the $\boldsymbol{\phi}$ that minimizes $\nabla_\epsilon \mathcal{D}_{KL}(q_{[T]} \mid\mid p) \mid_{\epsilon=0}$. This is equivalent to maximizing $-\nabla_\epsilon \mathcal{D}_{KL}(q_{[T]} \mid\mid p) \mid_{\epsilon=0}$ $= \mathbb{E}_{x\sim q}[\operatorname{tr}(\mathcal{A}_p \boldsymbol{\phi}(s))]$ in $\mathcal{H}^d$. By relating to the definition of $\mathbb{S}(q,p)$ in Equation 4.4 and the solution in Theorem 4.2.0.1, we see that this expression maximizes in the direction $\boldsymbol{\phi}^* = \mathbb{E}_{x\sim q}[\mathcal{A}_p \mathcal{K}(x,\cdot)]$.

Lemma 4.2.1.2. *Consider all the perturbation directions* $\boldsymbol{\phi}$ *in the ball* $\mathcal{B} = \{\boldsymbol{\phi} \in \mathcal{H}^d : \|\boldsymbol{\phi}\|^2_{\mathcal{H}^d} \leq \mathbb{S}(q,p)\}$ *of the vector-valued RKHS* $\mathcal{H}^d$*. The direction of steepest descent that maximizes* $-\nabla_\epsilon \mathcal{D}_{KL}(q_{[T]} \mid\mid p) \mid_{\epsilon=0}$ *is*

$$\boldsymbol{\phi}^*_{q,p}(\cdot) = \mathbb{E}_{x\sim q}[\mathcal{K}(x,\cdot)\nabla_x \log p(x) + \nabla_x \mathcal{K}(x,\cdot)]$$

with $-\nabla_\epsilon \mathcal{D}_{KL}(q_{[T]} \mid\mid p) \mid_{\epsilon=0} = \mathbb{S}(q,p)$*.*

The result of Lemma 4.2.1.2 suggests an iterative procedure that transforms an initial reference distribution q_0 to the target distribution p.

```
Algorithm 1: Stein Variational Gradient Descent
  Input: Initial reference distribution $x \sim q_0$, target distribution $p$
  Output: Resulting density $q_L$ transformed to the target $p$
1 for $\ell = 0$ to convergence do
2     Transform $x$ by $T^*_\ell(x) = x + \epsilon_\ell \cdot \boldsymbol{\phi}^*_{q_\ell,p}(x)$ ;
3     Set $q_{\ell+1}(x) \leftarrow q_{\ell_{[T^*_\ell]}}(x)$ ;
4 return $q_\ell$ ;
```

Each step of the algorithm decreases $\mathcal{D}_{\text{KL}}$ by $\epsilon_\ell \cdot \mathbb{S}(q_\ell, p)$. The constructed sequence of iteratively perturbed densities will eventually converge to the target p with sufficiently small step size $\{\epsilon_\ell\}$, under which $\boldsymbol{\phi}^*_{q_\infty,p}(x) \equiv 0$ and T^*_∞ reduces to the identity map.

However, Algorithm 1 is not practically computable. At each step, we must compute $\boldsymbol{\phi}^*_{q_\ell,p}(x)$ by evaluating an expectation over $x \sim q_\ell$, which is generally intractable. Instead, we can approximate the expectation using a set of k particles $\{x_i\}_{i=1}^k$ drawn from the initial distribution q_0, then iteratively update the particles with an empirical version of the transform by using the empirical mean of the particles in the ℓth iteration to approximate the expectation under q_ℓ.

Algorithm 2: SVGD In Practice

```
   Input: Initial particles $\{x_i\}_{i=1}^k \sim q_0$, target distribution $p$, convergence criterion $C$
   Output: A transformed set of particles $\{x_i\}_{i=1}^k$ that approximates the target $p$
1  SVGD($\{x_i\}_{i=1}^k$, $p$, $C$):
2      for $\ell = 0$ to $L - 1$ do
3          Define $\hat{\boldsymbol{\phi}}^*(x) = \frac{1}{k}\sum_{j=1}^k \left[\mathcal{K}(x_j^\ell, x)\nabla_{x_j^\ell} \log p(x_j^\ell) + \nabla_{x_j^\ell}\mathcal{K}(x_j^\ell, x)\right]$;
4          Update each particle $x_i^{\ell+1} \leftarrow x_i^\ell + \epsilon_\ell \hat{\boldsymbol{\phi}}^*(x_i^\ell)$ ;
5          if converged then
6              return $\{x_i^\ell\}_{i=1}^k$ ;
7      return $\{x_i^L\}_{i=1}^k$ ;
```

Algorithm 2 deterministically transports the set of particles to the target distribution, effectively providing a sampling method for $p(x)$. In practice, the convergence criterion C consists of a fixed L at a predetermined maximum number of iterations, along with the option of setting some early stopping criterion, e.g. when $\left|\hat{\boldsymbol{\phi}}^*(x_i^\ell)_j\right|$ is sufficiently small for all components j in all particles x_i^ℓ.

SVGD mimics a gradient dynamics at the particle level, where the two terms within the sum that appears in $\hat{\boldsymbol{\phi}}^*(x)$ play two different roles. The first term drives the particles toward the high probability areas of $p(x)$ by following a smoothed gradient direction, which is the weighted sum of the gradients of all of the points, weighted by the kernel function. The second term acts as a repulsive force that pushes the particles away from each other and prevents all the points from collapsing together into local modes of $p(x)$.

For the choice of kernel $\mathcal{K}$, the SVGD paper [8] uses the radial basis function kernel, $\mathcal{K}(x, x') = \exp(-\frac{1}{h}\left\|x - x'\right\|_2^2)$, likely because it is a simple but flexible kernel that satisfies all of the conditions imposed on the theoretical results discussed. It also has a relatively simple gradient, making it tractable to work with in practice. For the RBF bandwidth parameter h, the paper suggests updating h^ℓ at every iteration, such that $h^\ell = \text{med}_\ell^2/\log k$, where med_ℓ is the median of the pairwise distances between the particles $\{x_i^\ell\}_{i=1}^k$ at step ℓ.

This is based on the heuristic that, for each x_i, we want the contribution, i.e. weight, from its own gradient to balance out with the influence from the other points. Since $\mathcal{K}(x_i, x_i) = 1$, we set $\sum_{j\neq i}\mathcal{K}(x_i, x_j) \approx (k-1)\exp(-\frac{1}{h_\ell}\text{med}_\ell^2) = 1$.

Rearranging the expression gives us $h^\ell = \text{med}_\ell^2/\log(k-1)$. However, as discussed above, the result presented in the paper was instead $h^\ell = \text{med}_\ell^2/\log k$, possibly for brevity since

$\log k$ and $\log(k-1)$ are very close for large k. Another possibility was to avoid $\log(0)$ errors when using $k=1$, which reduces to MAP estimation. However, the $\log k$ in the denominator still causes divide-by-zero errors for $k=1$. This seems to be fixed in the accompanying code, available on GitHub [4], which uses bandwidth $h^\ell = \text{med}_\ell^2 / \log(k+1)$.

Remark. The choice of kernel $\mathcal{K}$ and the treatment of the bandwidth parameter h were by far the least rigorously justified steps in the SVGD paper, making this a ripe area for future research.

4.3 Improvements on SVGD: mSVGD

4.3.1 Motivation

The particles $\{x_i^\ell\}_{i=1}^k$ form better approximations for q_ℓ for larger values of k. To see this, let Φ_ℓ denote the nonlinear map that takes the measure of q_ℓ to that of $q_{\ell+1}$ in Algorithm 1, such that $q_{\ell+1} = \Phi_\ell(q_\ell)$. Then, the updates in Algorithm 2 can be seen as applying the same map Φ_ℓ on the empirical measure $\hat{q}_\ell$ of the particles $\{x_i^\ell\}_{i=1}^k$ to get the empirical measure $\hat{q}_{\ell+1}$ of the particles $\{x_i^{\ell+1}\}_{i=1}^k$ at the next iteration. That is, $\hat{q}_{\ell+1} = \Phi_\ell(\hat{q}_\ell)$. Since the empirical measure $\hat{q}_0$ given by the sample from the initial distribution converges to the true measure of the initial distribution q_0 as k increases, then by induction, $\hat{q}_\ell$ converges to q_ℓ when the map $\Phi_{\ell-1}$ is continuous in a proper sense.

In practice, this means that there are several crucial benefits to using a large number of particles k, many of which boil down to the central problem of expressiveness. When k is too small, the particles $\{x_i^\ell\}_{i=1}^k$ may not be expressive enough to properly approximate q_ℓ. This can lead to many issues, including the following.

- *Instability.* There may be instability while running the algorithm due to errors in approximations using the empirical mean.

- *Convergence.* It may be the case that $\{x_i^\infty\}_{i=1}^k$ cannot properly represent the target distribution $p(x)$, causing problems with non-convergence or vanishing gradients due to an insufficient sample size. This is an issue in both the computation task and in downstream inference.

- *Local modes.* In theory, SVGD can escape local modes in the target density landscape. But, the particles may get stuck when k particles are insufficient to spread throughout the full density. For larger k, the “repulsive force” discussed in the previous section spreads the particles away from each other, allowing them to “overflow” out of shallow local modes.

- *Tail approximation.* An insufficient number of particles k can prevent SVGD from properly spreading particles into tails or low probability regions. This leads to misrepresentation of extreme probabilities, causing issues with tasks such as constructing interval estimators or learning skewed distributions.

In summary, it is desirable to maximize the number of particles k used for SVGD. Unfortunately, as with most things in life, there is no free lunch. Increasing k introduces a new set of challenges, many of which are insurmountable in the standard SVGD setting.

The theoretical complexity of Algorithm 2 scales approximately polynomially with respect to k. The real-life runtime may grow even faster, depending on factors including the complexity of the target density, the implementation of the algorithm, and the hardware used for computation. Thus, naively increasing the number of particles incurs a significant computational cost that quickly becomes prohibitively expensive.

Another concern pertains to the numerical stability of the algorithm. Consider the sum in the approximation $\hat{\boldsymbol{\phi}}^*$.

$$\hat{\boldsymbol{\phi}}^*(x) = \frac{1}{k}\sum_{j=1}^{k}\left[\mathcal{K}(x_j^\ell, x)\nabla_{x_j^\ell}\log p(x_j^\ell) + \nabla_{x_j^\ell}\mathcal{K}(x_j^\ell, x)\right] \tag{4.9}$$

When k is too large and the particles are far from the target $p(x)$, this approximation can exhibit strange behavior due to both numerical instability from adding a large number of large floating point numbers and the $\nabla_{x_j^\ell}\log p(x_j^\ell)$ terms dominating the sum. The phenomena observed in testing included exploding gradients, diverging or collapsing particles, and approximation errors due to floating point imprecision. To make matters worse, the undesirable behavior varied — in both kind and severity — depending on the gradient descent optimizer being used, e.g. vanilla gradient descent vs. Adam. Some of these problems, particularly that of diverging or collapsing particles, were mitigated by setting a smaller learning rate ϵ, but this often caused the particles to slow down too quickly in later iterations. In these cases, the algorithm may not converge, or may require a significantly larger number of iterations to reach convergence, compounding the issue of prohibitive computational cost. So, when k is large, the convergence and runtime of the algorithm are highly sensitive to both the learning rate and the choice of gradient descent optimizer.

4.3.2 Proposed Solution

All the challenges associated with using a larger number of particles k can be overcome with a single solution: a smarter starting state. If the initial particles $\{x_i^0\}_{i=1}^k \sim q_0$ are already close to the target p, SVGD only requires a small number of iterations to reach convergence, diminishing the effect of increased computation per iteration. In addition, the gradient terms in the approximation in Equation 4.9 are small, eliminating most of the issues with numerical instability, as well as the need for small learning rates.

Unfortunately, since we generally do not have access to p beyond its log-density, starting SVGD near p is not a feasible solution. Usually, we are limited to a semi-informed or completely arbitrary initial distribution $\{x_i^0\}_{i=1}^k \sim q_0$. Then, how do we obtain a set of particles $\{x_i^0\}_{i=1}^k$ whose empirical measure is close to that of p? Conveniently, we already have the means to do so: SVGD itself. Our proposed solution addresses the circularity of using SVGD to solve for an initialization state of SVGD.

Instead of obtaining the full initialization set $\{x_i^0\}_{i=1}^k$, we use SVGD with early stopping to obtain a smaller set of particles $\{x_i\}_{i=1}^{k_0}$ for $k_0 \ll k$. Then, we use this smaller set to learn a starting state for the full set of k particles. Under this setup, majority of the computation occurs on a small number of particles k_0, leaving a small amount of micro-adjustment to be done on the larger set.

There are still some concerns to consider. One of the most glaring problem with using a small number of particles, as discussed in Section 4.3.1, is the risk of getting stuck in local modes. If the small set $\{x_i\}_{i=1}^{k_0}$ gets stuck in a local mode, we may still end up with a starting state quite far from p. To address this, we propose incrementally increasing the number of particles. That is, for $k_0 < k_1 < \cdots < k$, use SVGD on $\{x_i\}_{i=1}^{k_0}$ to solve for the starting state of $\{x_i\}_{i=1}^{k_1}$, then use SVGD again to solve for the starting state of $\{x_i\}_{i=1}^{k_2}$, and so on. This way, the particles are more likely to spread out of local modes earlier in the algorithm. Additionally, this process makes micro-adjustments of the particles at each level k_j, ensuring that errors in learning $\{x_i^0\}_{i=1}^k$ from $\{x_i\}_{i=1}^{k_0}$ are corrected as the algorithm progresses, rather than all at once on a very large k. Together, these effects work to minimize the amount of computation required on larger numbers of particles.

The final consideration is how to actually learn the starting state of $\{x_i\}_{i=1}^{k_{j+1}}$ from the smaller set $\{x_i\}_{i=1}^{k_j}$. One idea is to take the sample mean vector and sample covariance matrix of $\{x_i\}_{i=1}^{k_j}$, then sample a whole new set of k_{j+1} particles from a predetermined distribution, e.g. multivariate Gaussian. While seemingly reasonable, this idea has two major downsides that may lead to a bad starting state for $\{x_i\}_{i=1}^{k_{j+1}}$. First, it requires a strong distributional assumption that may be very different from the target p. Second, true randomness from i.i.d. sampling is usually much less evenly spread out than the particle spreading learned by SVGD, thus requiring more computation to correct.

Our proposed solution, mitotic Stein variational gradient descent (mSVGD), is inspired by cellular mitosis. Instead of resampling particles, we split each particle in the smaller set

$\{x_i\}_{i=1}^{k_j}$ by taking the states of the particles in the last two iterations of SVGD and concatenating them. This gives us $\{x_i\}_{i=1}^{k_{j+1}} \equiv \texttt{concat}(\{x_i^L\}_{i=1}^{k_j}, \{x_i^{L+1}\}_{i=1}^{k_j})$, effectively doubling the sample for no additional computational cost.

Algorithm 3: Mitotic SVGD

```
Input: Initial particles $\{x_i\}_{i=1}^{k_0} \sim q_0$, target distribution $p$, number of splits $M$,
       convergence criterion $C$
Output: A transformed set of particles $\{x_i\}_{i=1}^{k_M}$ that approximates the target $p$
1 mSVGD($\{x_i\}_{i=1}^{k_0}$, $p$, $M$, $C$):
2     for $m = 0$ to $M$ do
3         Run SVGD($\{x_i\}_{i=0}^{k_m}$, $p$, $C$) ;
4         if $m < M$ then
5             Set $\{x_i\}_{i=1}^{k_{m+1}} \leftarrow$ concat($\{x_i^L\}_{i=1}^{k_m}$, $\{x_i^{L+1}\}_{i=1}^{k_m}$)
6         else
7             return $\{x_i^L\}_{i=1}^{k_M}$ ;
```

If we consider all k_j particles jointly, the concatenation step can be viewed as taking two highly correlated samples of the joint distribution learned from SVGD. This negates the need for distributional assumptions by preserving the learned distribution shape. Then, the correction done by SVGD on the k_{j+1} particles must simply nullify the correlation between the split particles. Note that as the particles grow dense in the density of p, the correlations grow less significant with respect to the spread of the particles throughout p, thus requiring less computation to correct at larger k_j's.

We demonstrate the advantages of mSVGD over standard SVGD for MAGI in Section 6. Rigorous testing of mSVGD in other target distribution settings beyond MAGI is left for future work, along with formal investigations of theoretical properties and guarantees.

Also for future work, we discuss some alternative methods for obtaining the mitosis splits. First, instead of concatenating two consecutive steps, one possibility is to concatenate two more distant steps. If the gradients have not shrunk too much by the time of splitting, this may give less correlated particles. Another possibility is to clone the particles in the last state, then add some small deterministic or stochastic noise to the clone before concatenating. Lastly, one possibility is to run multiple instantiations of SVGD in parallel and concatenate the results. This method has a cascading growth in computational cost, since e.g. if we want $k = 800$ with $k_0 = 100$, we need to run $8\times$ instances of $k_0 = 100$, to get $4\times$ instances of $k_1 = 200$, to get $2\times$ instances of $k_2 = 400$, to get our final set of $k = 800$ particles, rather than just one instance at each level. However, this may be feasible if we have access to multiple computers or multiple GPUs.

5 A Python Package: mSVGD for MAGI

5.1 Mathematical Derivation

As discussed in Section 4.2, the SVGD algorithm only depends on the target p through score function $\nabla_x \log p(x)$. Thus, to apply mSVGD for MAGI, we must derive the gradient of the logarithm of the tempered posterior from Equation 2.12 with respect to each of the target variables. Although this density varies depending on the dynamical system $\boldsymbol{f}$, we can do most of the work in deriving the score function, leaving minimal effort for the end user.

Log Tempered Posterior

First, we take the natural logarithm of the tempered target density from Equation 2.12.

$$\begin{aligned}
&\log P^{(\beta)}_{\boldsymbol{\Theta},\boldsymbol{X}(\boldsymbol{I})|W_{\boldsymbol{I}},\boldsymbol{Y}(\boldsymbol{\tau})}(\boldsymbol{\theta},\boldsymbol{x}(\boldsymbol{I}) \mid W_{\boldsymbol{I}}=0, \boldsymbol{Y}(\boldsymbol{\tau})=\boldsymbol{y}(\boldsymbol{\tau})) \\
&\qquad = \text{constant} + \log \pi_{\boldsymbol{\Theta}}(\boldsymbol{\theta}) - \frac{1}{2}\sum_{d=1}^{D}\Bigg[\frac{1}{\beta}\left\|x_d(\boldsymbol{I}) - \mu_d(\boldsymbol{I})\right\|^2_{C_d^{-1}} \\
&\qquad\qquad + \left(N_d \log(2\pi\sigma_d^2) + \left\|x_d(\boldsymbol{\tau}_d) - y_d(\boldsymbol{\tau}_d)\right\|^2_{\sigma_d^{-2}}\right) \\
&\qquad\qquad + \frac{1}{\beta}\left\|\boldsymbol{f}^{\boldsymbol{x},\boldsymbol{\theta}}_{d,\boldsymbol{I}} - \dot{\mu}_d(\boldsymbol{I}) - m_d\{x_d(\boldsymbol{I}) - \mu_d(\boldsymbol{I})\}\right\|^2_{K_d^{-1}}\Bigg]
\end{aligned}$$

Gradient $\nabla_{\boldsymbol{\theta}}$

Now, we derive the gradient of the target log-density with respect to the ODE parameters $\boldsymbol{\theta}$. Note that the only terms in the log-density that depends on $\boldsymbol{\theta}$ are the prior and the K_d^{-1} norm terms. Recalling that the K_d^{-1} matrices are symmetric, we have a relatively simple gradient computation.

$$\begin{aligned} &\nabla_\theta \log P^{(\beta)}(\boldsymbol{\theta}, \boldsymbol{x}(\boldsymbol{I})) = \\ &\qquad \nabla_\theta \log \pi_{\boldsymbol{\Theta}}(\boldsymbol{\theta}) - \frac{1}{\beta}\sum_{d=1}^{D}\left(\nabla_\theta \boldsymbol{f}_{d,\boldsymbol{I}}^{\boldsymbol{x},\boldsymbol{\theta}}\right)^\top \left(K_d^{-1}\right)\left(\boldsymbol{f}_{d,\boldsymbol{I}}^{\boldsymbol{x},\boldsymbol{\theta}} - \dot{\mu}_d(\boldsymbol{I}) - m_d\{x_d(\boldsymbol{I}) - \mu_d(\boldsymbol{I})\}\right) \end{aligned} \tag{5.1}$$

The user inputs required are the gradient of the log-prior, the ODE system $\boldsymbol{f}$, and the gradient of $\boldsymbol{f}$ with respect to $\boldsymbol{\theta}$. Since the prior $\pi_{\boldsymbol{\Theta}}$ is usually set to be flat, its score $\nabla_\theta \log \pi_{\boldsymbol{\Theta}}(\boldsymbol{\theta})$ is usually just 0.

Gradient $\nabla_{\boldsymbol{x}}$

Next is the gradient of the target density with respect to the ODE trajectory $\boldsymbol{x}$. Below is the gradient with respect to a single component $\boldsymbol{x}_q$. The gradients with respect to each of the components have the same form.

$$\begin{aligned} &\nabla_{\boldsymbol{x}_q} \log P^{(\beta)}(\boldsymbol{\theta}, \boldsymbol{x}(\boldsymbol{I})) = \\ &\quad -\left[\frac{1}{\beta}\left(C_q^{-1}\right)\left(x_q(\boldsymbol{I}) - \mu_d(\boldsymbol{I})\right)\right] - \left[\frac{1}{\sigma_q^2}\left(x_q(\boldsymbol{\tau}_q) - y_q(\boldsymbol{\tau}_q)\right)\right] \\ &\quad - \frac{1}{\beta}\sum_{d=1}^{D}\left(\nabla_{\boldsymbol{x}_q}\boldsymbol{f}_{d,\boldsymbol{I}}^{\boldsymbol{x},\boldsymbol{\theta}} - m_d\nabla_{\boldsymbol{x}_q}x_d(\boldsymbol{I})\right)^\top \left(K_d^{-1}\right)\left(\boldsymbol{f}_{d,\boldsymbol{I}}^{\boldsymbol{x},\boldsymbol{\theta}} - \dot{\mu}_d(\boldsymbol{I}) - m_d\{x_d(\boldsymbol{I}) - \mu_d(\boldsymbol{I})\}\right) \end{aligned} \tag{5.2}$$

The user inputs required are the ODE system $\boldsymbol{f}$ and the gradient of $\boldsymbol{f}$ with respect to $\boldsymbol{x}$.

Gradient $\nabla_{\boldsymbol{\sigma}}$

Finally, we compute the gradient with respect to the sampling standard deviations σ_d.

$$\nabla_{\sigma_d} \log P^{(\beta)}(\boldsymbol{\theta}, \boldsymbol{x}(\boldsymbol{I})) = -\frac{N_d}{\sigma_d} + \|x_d(\boldsymbol{\tau}_d) - y_d(\boldsymbol{\tau}_d)\|_{\sigma_d^{-3}}^2 \tag{5.3}$$

where N_d is the number of observations in component d. Recall from Section 2.1.2 that the prior on σ_d is flat. There is no user input required beyond the data.

With these gradients, we can now conduct SVGD for MAGI. The required user inputs are the gradient of the log-prior of $\boldsymbol{\theta}$, the ODE system $\boldsymbol{f}$, and the gradients of $\boldsymbol{f}$ with respect to $\boldsymbol{\theta}$ and $\boldsymbol{x}$. These are the same inputs required by the current MAGI package that uses HMC [12], so our method does not increase the burden on the end user.

5.2 Improvements on MAGI Initialization

In our implementation of mSVGD for MAGI, the initialization procedure is slightly different than the setup described in Section 2.2. The existing initialization procedure involves a costly optimization step, optimizing the full posterior from Theorem 2.1.2.1 jointly over the ODE parameters $\boldsymbol{\theta}$ and the trajectory $\boldsymbol{X}_d(\boldsymbol{I})$ and Matern kernel parameters ϕ_d for any unobserved components d. Not only does this involve optimizing a complex function over many parameters, but the gradients of the Matern kernels with 2.01 degrees of freedom are difficult to manually compute, impossible to automatically differentiate, and unstable to approximate numerically in the presence of so many other variables being optimized. Hence, we desire an alternative initialization method that is faster, more stable, and easier to implement.

The basis for the new initialization method was proposed by Skyler Wu in "Are Statistical Methods Obsolete in the Era of Deep Learning?", a paper submitted in December 2024 to the JASA Special Issue on Statistics in AI.

The main idea is to compute the initialization of the ODE parameters and the unobserved components using only the ODE struture defined by $\boldsymbol{f}$, without any Gaussian process information. Thus, we do not need to fit the Matern kernel parameters $\boldsymbol{\phi}$ jointly with $\boldsymbol{X}(\boldsymbol{I})$ and $\boldsymbol{\theta}$. Below is an overview of the new fitting procedure. Step 2 will be expanded upon in the following subsection.

1. For observed components, initialize $\boldsymbol{X}_d(\boldsymbol{I})$ by using the observed values $\boldsymbol{y}_d(\boldsymbol{\tau})$ and linearly interpolating the remaining points in the discretization set $\boldsymbol{I}$.

2. Jointly initialize $\boldsymbol{\theta}$ and unobserved $\boldsymbol{X}_d(\boldsymbol{I})$ using ODE information from $\boldsymbol{f}$.

3. Initialize $\boldsymbol{\phi}$ and $\boldsymbol{\sigma}$ for the fully initialized $\boldsymbol{X}(\boldsymbol{I})$ using the procedure for observed components discussed in Section 2.2.2.

Initializing $\boldsymbol{\theta}$ and Unobserved $\boldsymbol{X}_d(\boldsymbol{I})$ Using $\boldsymbol{f}$.

Let $\hat{\boldsymbol{X}}$ be a guess for the unobserved components of $\boldsymbol{X}(\boldsymbol{I})$ and $\hat{\boldsymbol{\theta}}$ be a guess for the ODE parameters $\boldsymbol{\theta}$. Let $\tilde{X}$ be the full trajectory matrix, where the observed trajectories are filled with the interpolated values from Step 1 and the unobserved trajectories are filled with $\hat{\boldsymbol{X}}$. Note that $\tilde{X}$ contains estimates for $\boldsymbol{X}$ at every time step in the discretization set

$\boldsymbol{I}$, so we can use numerical differentiation to compute the matrix $\hat{\boldsymbol{f}}$, an estimate of the ODE system values. We use a (mostly) second order finite-difference approximation, using the time steps in $\boldsymbol{I}$. The details of this computation can be found in the `_helpers.py` file in the source code [6].

Let $\hat{\boldsymbol{f}}_{t,d}$ be the estimate of the derivative of the dth component at time t. Since the true ODE system $\boldsymbol{f}$ is known, we can define the following loss function.

$$\mathscr{L}(\hat{\boldsymbol{X}}, \hat{\boldsymbol{\theta}}) = \text{mean}\left((\hat{\boldsymbol{f}}_{t,d} - \boldsymbol{f}(\tilde{\boldsymbol{X}}, \hat{\boldsymbol{\theta}})_{t,d})^2\right) \tag{5.4}$$

This loss can be interpreted as the magnitude of the difference between the numerical derivatives given by the guess $(\hat{\boldsymbol{X}}, \hat{\boldsymbol{\theta}})$ and the actual derivatives at the guesses, implied by $\boldsymbol{f}$. It minimizes when $\tilde{\boldsymbol{X}}$ and $\hat{\boldsymbol{\theta}}$ have numerical derivatives that best match the ideal derivatives. We use automatic differentiation gradient descent to optimize $\mathscr{L}$, solving for the best initializations $\hat{\boldsymbol{X}}$ and $\hat{\boldsymbol{\theta}}$ given the ODE structure defined by $\boldsymbol{f}$.

Though this setup is mostly stable, there are edge cases of non-identifiability. There exist ODE systems $\boldsymbol{f}$ for which a certain value of some element of $\boldsymbol{\theta}$ may cause the derivatives to be zero. As a dummy example,

$$\frac{dX_1}{dt} = \theta_1\left(X_2^2 + \frac{X_3}{\theta_2}\right)$$

Then, when the guess of $\hat{\theta}_1$ is zero or near-zero and the guess of the trajectory $\hat{X}_1$ is flat, the estimated derivative agrees with the implied derivative, thus giving a small value of the loss $\mathscr{L}$. This may be a strong local minimum, or even an undesirable global minimum, resulting in completely incorrect initializations.

Also, notice that the X_3 and θ_2 are not identifiable if there is no further information about them in the derivatives of the other components. If a guess $(\hat{X}_3, \hat{\theta}_2)$ has a numerically estimated derivative that agrees with implied derivative, then for any constant c, $(c\hat{X}_3, c\hat{\theta}_2)$ does as well. Thus, the initial estimates of some of the unobserved components in $\hat{\boldsymbol{X}}$ may be translated up or down, corresponding to a proportionally scaled element of $\hat{\boldsymbol{\theta}}$.

To fix these issues, the approach that we take is to add an attraction term to the loss function, which can be used to pull elements of $\boldsymbol{\theta}$ away from undesired values and lessen the effects of translation invariance. Let $\tilde{\boldsymbol{\theta}}$ be a user-defined guess for $\boldsymbol{\theta}$ and let $\boldsymbol{\lambda}$ be a vector of confidence levels corresponding to each element of $\tilde{\boldsymbol{\theta}}$, describing how strongly to pull $\hat{\boldsymbol{\theta}}$ toward $\tilde{\boldsymbol{\theta}}$. Then, we augment $\mathscr{L}$ as follows.

$$\mathscr{L}^*(\hat{\boldsymbol{X}}, \hat{\boldsymbol{\theta}}) = \text{mean}\left((\hat{\boldsymbol{f}}_{t,d} - \boldsymbol{f}(\tilde{\boldsymbol{X}}, \hat{\boldsymbol{\theta}})_{t,d})^2\right) + \text{mean}\left(\boldsymbol{\lambda}_j(\hat{\boldsymbol{\theta}}_j - \tilde{\boldsymbol{\theta}}_j)^2\right) \tag{5.5}$$

Remark (1). Formulating $\boldsymbol{\lambda}$ as a vector allows finer grained control of the strength of attraction for each element of $\boldsymbol{\theta}$. Note that when the confidence level $\boldsymbol{\lambda}_j = 0$ for all j, the augmented loss function $\mathscr{L}^*$ reduces back to $\mathscr{L}$.

Remark (2). For gradient descent, we set the initial guess for $\hat{\boldsymbol{X}}$ to be a matrix filled with the mean of the linearly imputed observed trajectories, and the initial guess for $\hat{\boldsymbol{\theta}}$ to be $\tilde{\boldsymbol{\theta}}$.

5.3 Code Design

5.3.1 Functionality and Speed

Obviously, the top priority when designing the Python package was to write functioning code. This means correctly implementing the mSVGD algorithm for MAGI as described in the preceding chapters. Recall that in Section 4.2, there was some ambiguity about how to set the RBF kernel bandwidth h^ℓ at each iteration. For our implementation, we use $h^\ell = \text{med}_\ell^2 / \log k$ to avoid divide-by-zero errors for $k = 2$. We impose a separate MAP estimation treatment for $k = 1$, bypassing the use of the kernel entirely.

Additionally, since the overall objective is to provide a computationally faster method for MAGI, the algorithms were optimized for speed. The implementation of SVGD, even without using the mitosis option, is significantly faster than the code provided alongside the SVGD paper [4]. Unlike their code, we leverage batch matrix operations instead of much slower for-loops, as well as add an option to use GPU acceleration via PyTorch or TensorFlow. Although our implementation of SVGD is much faster than that of Liu et al. [4] and our MAGI inference is much faster than that of Wong et al. [12], it is said that code optimization is never finished. However, the same cannot be said about thesis projects, so we leave further optimization to future work.

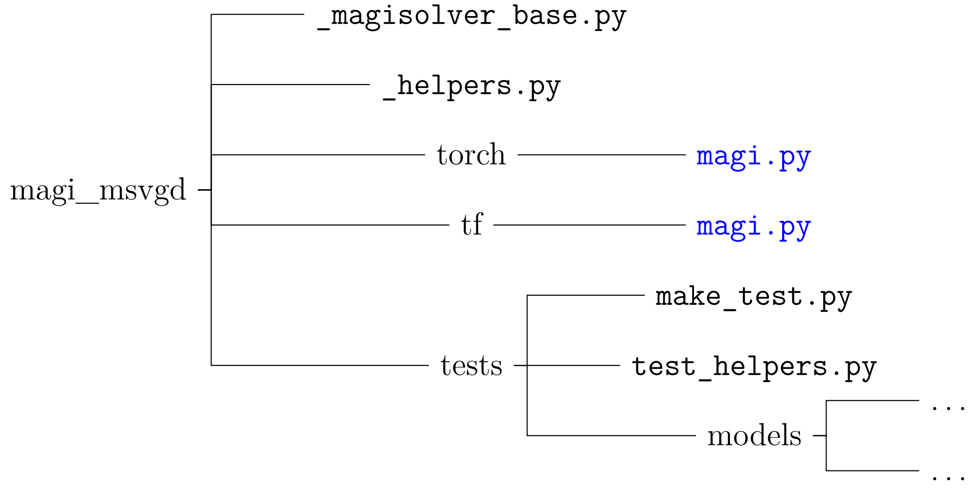
5.3.2 User-Friendliness

Since the package is intended to be used, user-friendliness is also a priority. The package code is open source and entirely written in Python, making it very readable despite a small number of unorthodox design choices for the sake of computational speed. The user interface was designed to be similar to the existing MAGI package to minimize confusion if a user wishes to use both. Despite incorporating flexible customization of MAGI and mSVGD hyperparameter settings in our package, there are some non-essential optional settings provided by the original MAGI package that we do not include. Their implementation is left to future work.

Our package also provides the option of using either a PyTorch or TensorFlow front-end to conduct computation. Implementing multiple front-ends for the same package introduces challenges with functionality, redundancy, consistency, and maintenance. Our solution to these concerns is discussed in the next subsection.

5.3.3 Organization and Structure

To ensure that the PyTorch and TensorFlow versions have the same functionality, to minimize redundant code, and to maximize ease of maintenance, we organize our code with a structured abstraction scheme.



The `_magisolver_base.py` file contains all of the package logic, with `_helpers.py` containing helper functions for MAGI initialization. These are polymorphic implementations; that is, all library-specific functions and objects, such as tensor operations or gradient descent optimizers, are abstracted away, to be specified later. This way, we have a single common back-end base that never sees PyTorch nor TensorFlow, upon which our front-ends can be built. This ensures consistency between the front-ends, minimizes code redundancy, and reduces package maintenance and code changes to editing just the back-end.

The `torch/magi.py` and `tf/magi.py` files define the library-specific functions and objects for PyTorch and TensorFlow, respectively, and load them into the skeleton provided by `_magisolver_base.py`. The user interacts with the package through these files and is thus able to use their front-end of choice with ease, without need for manual configuration.

The `tests` folder contains the source code used for making test cases, using the PyTorch front-end. `make_test.py` includes code for numerically solving the ODEs and generating data, `test_helpers.py` includes code for discretizing data and automatically checking gradients, and the `tests` folder contains the ODEs and gradients of the model used in the tutorial in the next section and the models used for testing in Chapter 6.

5.4 Package Tutorial

To install the package from GitHub via PIP, enter the following command in the Shell (Linux), Command Line (Windows), or Terminal (Mac). If using the TensorFlow front-end, Linux is strongly recommended for GPU compatibility. Note that the TensorFlow front-end is functional, though it has some minor bugs and is slower than the PyTorch front-end. Investigations and improvements are still underway. See Appendix B for a demonstration of the TensorFlow front-end.

```
pip install git+https://github.com/jamieliu2/magi_msvgd.git@main
```

In a notebook file, we first import NumPy and PyTorch for working with data objects and defining the ODE and its gradients. Then, we import the `MAGISolver` object. This is the only object required by the user to interact with the package.

```python
import numpy as np
import torch
from magi_msvgd.torch.magi import MAGISolver
```

Now, we define the ODE and its gradients. As a running example, recall the HIV model from the introduction (Section 1.1). The three-component system consists of $\boldsymbol{X} = (T_U, T_I, V)$, where T_U, T_I are the concentrations of uninfected and infected cells, respectively, and V is the viral load.

$$\boldsymbol{f}(\boldsymbol{X}, \boldsymbol{\theta}, t) = \begin{pmatrix} \lambda - \rho T_U - \eta(t) T_U V \\ \eta(t) T_U V - \delta T_I \\ N\delta T_I - cV \end{pmatrix} \tag{5.6}$$

where $\eta(t) = 9 \times 10^{-5} \times (1 - 0.9\cos(\pi t/1000))$ and $\boldsymbol{\theta} = (\lambda, \rho, \delta, N, c)$ are the associated parameters. To facilitate computation, we also derive the gradients of $\boldsymbol{f}$ with respect to $\boldsymbol{X}$ and $\boldsymbol{\theta}$.

$$\frac{\partial \boldsymbol{f}(\boldsymbol{X}, \boldsymbol{\theta}, t)}{\partial \boldsymbol{X}} = \begin{pmatrix} -\rho - \eta(t)V & 0 & -\eta(t)T_U \\ \eta(t)V & -\delta & \eta(t)T_U \\ 0 & N\delta & -c \end{pmatrix} \tag{5.7}$$

$$\frac{\partial \boldsymbol{f}(\boldsymbol{X}, \boldsymbol{\theta}, t)}{\partial \boldsymbol{\theta}} = \begin{pmatrix} 1 & -T_U & 0 & 0 & 0 \\ 0 & 0 & -T_I & 0 & 0 \\ 0 & 0 & NT_I & \delta T_I & -V \end{pmatrix} \tag{5.8}$$

When encoding the ODE in PYTHON, we must write a function that can be automatically differentiated. This is necessary for the initialization procedure described in Section 5.2.

The input `X` is a matrix with $n = |\boldsymbol{I}|$ rows and D columns. The input `theta` is a row vector with p components. The input `t` is a column vector with n components. The return value is an $n \times D$ matrix describing the ODE in all of the components at each time in $\boldsymbol{I}$.

```python
def ode(X, theta, t=None):
    n = X.shape[0]
    T_U, T_I, V = X.T
    lam, rho, delta, N, c = theta.repeat([n, 1]).T
    t = t.flatten()
    eta = 9e-5 * (1 - 0.9*torch.cos(torch.pi*t/1000))

    return torch.stack([lam - rho*T_U - eta*T_U*V,
                        eta*T_U*V - delta*T_I,
                       N*delta*T_I - c*V], axis=1)
```

Next, we encode the gradient of the ODE with respect to $\boldsymbol{X}$. This function takes the same inputs as `ode` and outputs an $n \times D \times D$ array, where the array slice `[:,i,j]` is the partial derivative of the jth system component with respect to the ith system component.

```python
def dfdx(X, theta, t=None):
    n = X.shape[0]
    T_U, T_I, V = X.T
    lam, rho, delta, N, c = theta.repeat([n, 1]).T
    t = t.flatten()
    eta = 9e-5 * (1 - 0.9*torch.cos(torch.pi*t/1000))

    zero = torch.zeros(n, device=theta.device, dtype=theta.dtype)

    return torch.stack([torch.stack([-rho - eta*V, zero, -eta*T_U], axis=1),
                        torch.stack([eta*V, -delta, eta*T_U], axis=1),
                        torch.stack([zero, N*delta, -c], axis=1)], axis=2)
```

Finally, the encoding of the gradient of the ODE with respect to $\boldsymbol{\theta}$ takes the same arguments as the other two and returns an $n \times p \times D$ array, where the array slice `[:,i,j]` is the partial derivative of the jth system component with respect to the ith parameter in $\boldsymbol{\theta}$.

```
def dfdtheta(X, theta, t=None):
    n = X.shape[0]
    T_U, T_I, V = X.T
    lam, rho, delta, N, c = theta.repeat([n, 1]).T
    t = t.flatten()
    eta = 9e-5 * (1 - 0.9*torch.cos(torch.pi*t/1000))

    zero = torch.zeros(n, device=theta.device, dtype=theta.dtype)
    one = torch.ones(n, device=theta.device, dtype=theta.dtype)

    return torch.stack([torch.stack([one, -T_U, zero, zero, zero], axis=1),
                        torch.stack([zero, zero, -T_I, zero, zero], axis=1),
                       torch.stack([zero, zero, N*T_I, delta*T_I, -V],
                        ↪  axis=1)], axis=2)
```

Remark. The encodings and dimensionalities of all of these functions are consistent with those of the existing MAGI package [12].

Included in our package is some helpful code for pre-processing tasks. Unfortunately, these are written using PyTorch and are thus inaccessible to the TensorFlow front-end. Using the helper code, we can numerically solve systems, construct observation matrices, sample dummy data, and discretize data sets. We can also use automatic differentiation to check that `dfdx` and `dfdtheta` correctly encode the gradients of `ode`.

```
from magi_msvgd.tests import test_helpers
# n = arbitrary data size, D = dimensions in X, p = dimensions in theta
test_helpers.check_gradients(ode, dfdx, dfdtheta, n=201, D=3, p=5,
↪  trials=100)
```

For each of `trials` loops, this will randomly generate an $n \times D$ matrix as a test $\boldsymbol{X}$, a p-dimensional vector as a test $\boldsymbol{\theta}$, and a scalar as a test time t. Then, at these values, it will compare the gradients given by automatic differentiation of `ode` with the gradients given by `dfdx` and `dfdtheta`. It will return a tuple `(x_score, theta_score)` containing the proportion of trials that were correct.

Remark. Correctly encoded gradients may not return perfect 1.0 scores due to floating point errors. Tolerance for "closeness" may be adjusted, but generally if the score is greater than 0.75, the encoding is probably correct.

Now, we need to generate a dataset. For this example, we set the ground truth values $\boldsymbol{\theta} = (36, 0.108, 0.5, 1000, 3)$ and initial conditions $\boldsymbol{x}(0) = (600, 30, 100000)$. We sample data with sampling standard deviations $\boldsymbol{\sigma} = (\sqrt{10}, \sqrt{10}, 10)$, which are unknown. All three components are observed at 0.2 minute increments from time 0 to time $T = 20$ minutes.

Our package provides code for generating test cases, which we will demonstrate here. Users can also use this code by creating a PYTHON file that defines a model of interest, following the template of the example models in the `tests/models` folder. Note that the `test_helpers` functions can be used without defining the whole model, and only gradient checking requires PYTORCH.

We set the random NUMPY seed 2025 for reproducibility. Note that the argument `obs_times` is a matrix that encodes the times at which to sample observations, where the 0th column contains the observation times, and the rest of the columns contain truthy/falsy values that indicate whether the corresponding component was observed at that time.

```python
from magi_msvgd.tests.make_test import ODEmodel
from magi_msvgd.tests.models import hiv
# define HIV model
HIV = ODEmodel(hiv)
# solve ODE via numerical integration
HIV.get_ode_solution(X0=np.array([600, 30, 100000]), T=20, step=1e-4)

# define observation times (0, 0.2, 0.4, ..., 20)
tau = [np.linspace(0, 20, 101), np.linspace(0, 20, 101)]
# define observation matrix, col 0=times, rows in other cols: 1 if observed,
↪  0 otherwise
obs_times = test_helpers.obs_matrix(tau)

# generate data
data = HIV.generate_sample(obs_times=obs_times, sigma=np.array([10**0.5,
↪  10**0.5, 10]), random_seed=2025)
```

Let's inspect the ground truth solution (orange lines) and the sample data (red points).

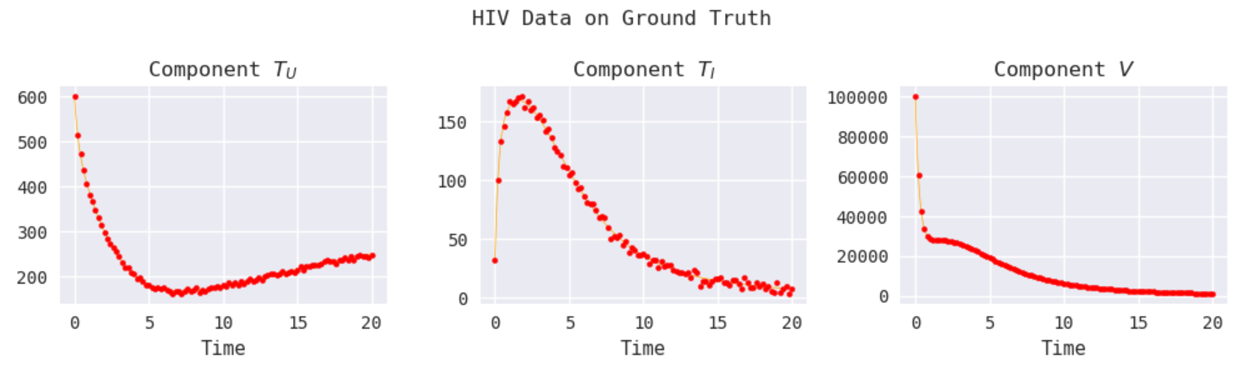


Figure 5.4.1: HIV model ground truth trajectories and dataset with NumPy seed 2025.

The last preparation step is to define the discretization set $\boldsymbol{I}$ and to construct the discretized data matrix, which is done manually by the user. This is the means by which the `MAGISolver` object interacts with the data. The discretized data matrix may be a NumPy array or a PyTorch/TensorFlow tensor, and should have dimensionality $n \times (D+1)$. The 0th column should contain the times in the discretization set $\boldsymbol{I}$, and the remainder of the columns should be filled with the data of the corresponding components. The unobserved points, whether due to discretization or being unobserved in the original dataset, should be filled with null types, e.g. `None` or `np.nan` or `torch.nan`.

For this example, we use $2\times$ discretization, adding 1 equally spaced discretization time between every pair of observed data points. So, we set $\boldsymbol{I} = (0, 0.1, 0.2, \ldots, 20)$. This gives us $n = |\boldsymbol{I}| = 201$ discretization time points. We also provide code for discretizing data.

```
# define discretization set
I = np.linspace(0, 20, 201)
data_disc = test_helpers.discretize_data(data, I)
```

Now we have all we need to use the `MAGISolver` object to conduct mSVGD for MAGI. Instantiating this object stores all of the supplied variables, computes initialization states for $(\boldsymbol{X}, \boldsymbol{\theta}, \boldsymbol{\sigma})$, computes the Matern kernel parameters $\boldsymbol{\phi}$, and pre-computes the GP matrices C_d^{-1}, m_d, K_d^{-1} for each component d. Behind the scenes, it also configures the library-specific polymorphic representations, depending on the front-end being used. The existing MAGI package uses spares matrices for the GP matrices [11], which we do not.

```
magisolver = MAGISolver(
    ode=ode,
    dfdx=dfdx,
    dfdtheta=dfdtheta,
    data=data_disc,
    theta_guess=np.array([50., 0., 0., 1000., 5.]),
    theta_conf=np.array([0., 0., 0., 0., 0.]),
    sigmas=None,
    X_guess=1
    mu=None, mu_dot=None,
    pos_X=False, pos_theta=False,
    prior_temperature=None
    bayesian_sigma=True
)
```

MAGISolver Parameters

- **ode** (*function*) – A Python function that encodes the ODE. Takes three arguments (`X`, `theta`, `t`) and returns an $n \times D$ matrix. Must be automatically differentiable.
- **dfdx** (*function*) – A Python function that encodes the gradient of the ODE with respect to $\boldsymbol{X}$. Takes three arguments (`X`, `theta`, `t`) and returns an $n \times D \times D$ array.
- **dfdtheta** (*function*) – A Python function that encodes the gradient of the ODE with respect to $\boldsymbol{\theta}$. Takes three arguments (`X`, `theta`, `t`) and returns an $n \times p \times D$ array.
- **data** (*array or tensor*) – A NumPy array or PyTorch/TensorFlow tensor containing the discretized data at time points $\boldsymbol{I}$. Unobserved values should be filled with null type values.
- **theta_guess** (*array or tensor*) – A NumPy array or PyTorch/TensorFlow tensor containing the user guess for $\boldsymbol{\theta}$. This is required for determining p and to provide a starting state for computing the initialization state.
- **theta_conf** (optional*: float or array or tensor*) – A NumPy array or PyTorch/TensorFlow tensor containing the confidence levels of `theta_guess`. If 0.0 (default), `theta_guess` does not affect the loss function to solve for $\boldsymbol{\theta}$ initialization. Otherwise, the elements of `theta_conf` scales the attraction strength of the corresponding elements of $\boldsymbol{\theta}$'s initializaiton state toward `theta_guess`.
- **sigmas** (optional*: None or array or tensor*) – A NumPy array or PyTorch/TensorFlow tensor containing sampling standard deviations. If `None`

(default), all sampling standard deviations are treated as unknown. Individual entries may be set to null type values if $\boldsymbol{\sigma}$ is partially unknown.

- **X_guess** (optional*: int*) – Number of times to recursively run the initialization procedure of unobserved components in $\boldsymbol{X}$. May yield more stable initializations and escape local modes. (default: 1). **Warning:** setting `X_guess>1` does not currently work and for the TensorFlow front-end and will cause an error, due to TensorFlow's restrictions on creating `tf.Variables`.

- **mu** (optional*: None or array or tensor*) – An $n \times D$ NumPy array or PyTorch/TensorFlow tensor containing the GP prior mean function of each component, evaluated at the times in $\boldsymbol{I}$. If `None` (default), use the flat zero prior $\mu(x) = 0\ \forall x \in \mathbb{R}$ for all components.

- **mu_dot** (optional*: None or array or tensor*) – An $n \times D$ NumPy array or PyTorch/TensorFlow tensor containing the derivative of the GP prior mean function of each component, evaluated at the times in $\boldsymbol{I}$. If `None` (default), use the flat zero prior $\dot{\mu}(x) = 0\ \forall x \in \mathbb{R}$ for all components.

- **pos_X, pos_theta** (optional*: bool*) – Flag if $\boldsymbol{X}$ or $\boldsymbol{\theta}$ is strictly positive. Setting to `True` will sample initial particles from $|q_0|$ and restrict mSVGD to positive particles. (default: `False`). **Warning:** these settings are not currently working for the TensorFlow front-end due to TensorFlow's immutable tensors, but will not cause an error.

- **prior_temperature** (optional*: float*) – The tempering factor $1/\beta$ by which to scale the contribution of the GP prior. If `None` (default), use the recommended setting $1/\beta = N/(Dn)$.

- **bayesian_sigma** (optional*: boolean*) – If `True` (default), give Bayesian treatment to unknown sampling standard deviations σ_d, leaning alongside $\boldsymbol{X}$ and $\boldsymbol{\theta}$. Otherwise, hold fixed at initialized value, which is usually a significantly worse estimate of σ_d.

Here, we can manually adjust the initialization states, if necessary. In this example, we correct the bad σ_V initialization, as recommended by Ref. [12], which will help stabilize the learning algorithm.

```
magisolver.sigmas[2] = 10
```

We can edit the initialization states of $\boldsymbol{X}, \boldsymbol{\theta}, \boldsymbol{\sigma}$ in the attributes `x_init`, `theta_init`, and `sigmas`, respectively. We can also edit the optimized GP prior hyperparameters $\boldsymbol{\phi}$ via the attribute `phis`, which is a $D \times 2$ array. However, if we change $\boldsymbol{\phi}$, we have to re-compute the GP matrices C_d^{-1}, m_d, K_d^{-1} with the `build_matrices` helper function.

```
from magi_msvgd._helpers import build_matrices
build_matrices(magisolver)
```

Now, we use the `initialize_particles` method to prepare for mSVGD. This function defines some variables necessary for the mSVGD algorithm and stores batched matrices to facilitate computation. It also samples the initial set of k_0 particles from q_0, which is a Gaussian distribution with means at the previously solved initialization states and a user-defined initial standard deviation. In this example, we use $k_0 = 400$.

```
magisolver.initialize_particles(k_0=400, dtype=torch.float64, device='cuda',
↪   init_sd=1, random_seed=2025)
```

MAGISolver Method: `initialize_particles` Parameters

- **k_0** (*int*) – The number of initial particles to use for mSVGD. If `k_0` is set to 1, automatically configure MAP estimation; mitosis can still be used normally.

- **dtype** (*type*) – The data type to use for mSVGD computation. This should be a PyTorch dtype if using the PyTorch front-end, and a TensorFlow, NumPy, or Python dtype if using the TensorFlow front-end. Note: 32-bit floats are much faster (depending on the hardware) and very stable, but sometimes 64-bit floats are needed, usually when gradients are too large to represent in 32 bits.

- **device** (optional*: None or device*) – The device to use for mSVGD computation. If `None` (default), allow front-end to choose device. For PyTorch, the default device is typically CPU. TensorFlow manages device usage automatically, regardless of `device` setting.

- **init_sd** (optional*: float*) – The standard deviation for the mSVGD initial Gaussian distribution q_0. Note: $|q_0|$ is used for $\boldsymbol{\sigma}$ and if `pos_X` or `pos_theta` is flagged. (default: 0.2).

- **random_seed** (optional*: None or int*) – The random seed to use when sampling particles from the Gaussian initial distribution q_0. The initial states of $\boldsymbol{X}$, $\boldsymbol{\theta}$, and $\boldsymbol{\sigma}$

are sampled separately, with the seed reset to `random_seed` for each. If `None` (default), sample without setting a seed.

- **mitosis** (*not for user*) – This setting allows this function to be used to re-define batched matrices during mitosis splits of the mSVGD particles. Users should leave it set to `False` (default).

Finally, we run mSVGD using the `solve` method.

```python
# define optimizer object class and parameters
optimizer = torch.optim.Adam
optimizer_kwargs = {'lr':1e-1}

# run solver
X_result, theta_result, sigma_result, trajectories =
↪  magisolver.solve(optimizer=optimizer, optimizer_kwargs=optimizer_kwargs,
                max_iter=500, mitosis_splits=1, atol=0.85, rtol=0,
                ↪  monitor_convergence=20)
```

MAGISolver Method: `solve` Parameters

- **optimizer** (*optimizer class*) – A PyTorch or TensorFlow optimizer object class to use for mSVGD, e.g. from `torch.optim` or `tf.keras.optimizers`.

- **optimizer_kwargs** (optional*: dict*) – A Python dictionary containing keyword arguments to for `optimizer`. `optimizer_kwargs` should not contain the reserved keyword "params." (default: empty `dict()`).

- **max_iter** (optional*: int*) – The maximum number of mSVGD iterations per mitosis split. (default: $10,000$).

- **mitosis_splits** (optional*: int*) – The number of mitosis splits M to perform for mSVGD, such that the final number of particles $k = k_0 \cdot 2^M$. (default: 0).

- **atol** (optional*: float*) – Absolute tolerance parameter for early stopping criterion, defined below. (default: 10^{-2}).

- **rtol** (optional*: float*) – Relative tolerance parameter for early stopping criterion, defined below. (default: 10^{-8}).

$$\left|\hat{\boldsymbol{\phi}}^*(x_i)_j\right| \leq \texttt{atol} + \texttt{rtol} \times |x_{i,j}| \text{ for all particles } i \text{ in all inferred variables } j \tag{5.9}$$

- **bandwidth** (optional*: float*) – Bandwidth parameter h for SVGD radial basis function kernel. If non-positive (default: -1), automatically tune bandwidth at each iteration, $h^\ell = \text{med}_\ell^2 / \log k$.

- **monitor_convergence** (optional*: int*) – If 0 (default), do not monitor convergence. Otherwise, `monitor_convergence` specifies how often to print $\max\left(\left|\hat{\boldsymbol{\phi}}^*(x_i)_j\right|\right)$. It also specifies how often to record the current state of the $\boldsymbol{\theta}$ components of the particles, which are returned upon completion. **Warning:** this setting may incur significant memory cost.

MAGISolver Method: `solve` Returns

- **Xs** (*tensor*) – A $k \times n \times D$ dimensional tensor containing the resulting state for the $\boldsymbol{X}$ components of the particles. The 0th axis indexes over the particles, the 1st axis indexes over the time steps in $\boldsymbol{I}$, the 2nd axis indexes over the ODE components.

- **thetas** (*tensor*) – A $k \times p$ dimensional tensor containing the resulting state for the $\boldsymbol{\theta}$ components of the particles. The 0th axis indexes over the particles, the 1st axis indexes over the entries in $\boldsymbol{\theta}$.

- **sigmas** (*tensor*) – A $k \times |\boldsymbol{\sigma}_{\text{unknown}}|$ dimensional tensor containing the resulting state for the $\boldsymbol{\sigma}$ components of the particles. The 0th axis indexes over the particles, the 1st axis indexes over the entries in $\boldsymbol{\sigma}_{\text{unknown}}$. Note that the ith column corresponds to the ith unknown σ_d, ordered by d, which may not correspond to the ith ODE system component. If all sampling standard deviations are known, `solve` still returns an empty tensor for `sigmas`.

- optional**: trajectories** (*list[tensor]*) – This is returned if an only if `solve` is configured to monitor convergence. A list of $k \times p$ tensors that represent the state of the $\boldsymbol{\theta}$ components of the particles, at iteration intervals encoded by the `monitor_convergence` parameter. If `monitor_convergence` is 0 or `False`, only the first three items are returned.

Note that we can extract mSVGD results without having to re-run `solve`, since `MAGISolver` saves the state of the particles in the `magisolver.particles` attribute.

```
X_result, theta_result, sigma_result =
↪  magisolver.from_msvgd_vector(magisolver.particles)
```

This means that if we want continue to run mSVGD on partially-transformed particles or run mSVGD with different stopping criteria at different split levels, we can simply re-call the `solve` method with the desired parameter settings. We can also force a split by running `solve` with `max_iter=1` and `mitosis_splits=1`.

At any point, we can optionally save the `MAGISolver` object by writing its byte stream to a file using the `dill` library, which extends PYTHON's `pickle` library [10].

```
pip install dill
```

```
import dill
with open('magisolver.obj', 'wb') as file:
    dill.dump(magisolver, file)
```

Then, in some later session, after importing the necessary libraries, we only need to load the byte stream to have access to the original `magisolver` object. This allows us to avoid re-running the initialization procedure and to save computed results.

```
with open('magisolver.obj', 'rb') as file:
    magisolver = dill.load(file)
```

Below, we plot the evolution the distribution of $\boldsymbol{\theta}$ as mSVGD progressed, which is contained by the information returned by `solve` in `trajectories`. We can also inspect the final learned distributions of $\boldsymbol{\theta}$, which is encoded in the returned variable `thetas`

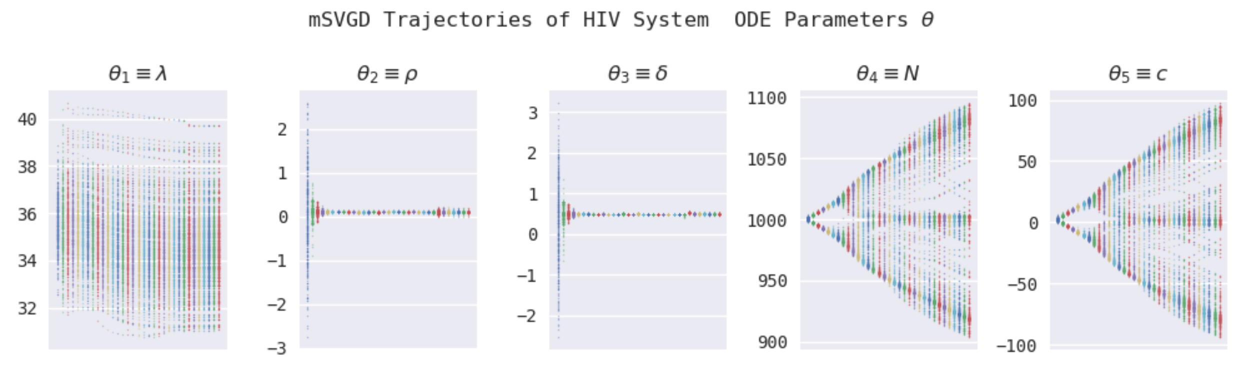


Figure 5.4.2: HIV model system parameters $\boldsymbol{\theta}$ evolving during mSVGD for dataset with NUMPY seed 2025.

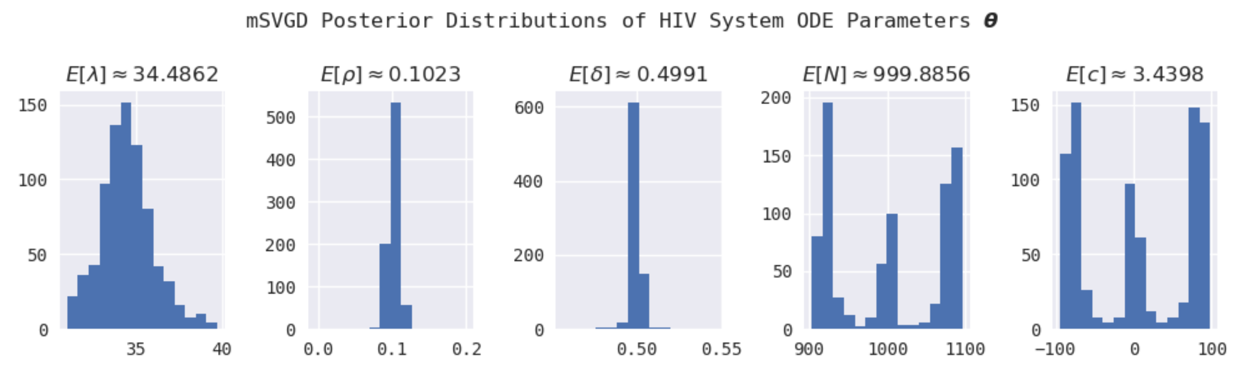


Figure 5.4.3: Posterior distribution of HIV model system parameters $\boldsymbol{\theta}$ learned from mSVGD for dataset with NumPy seed 2025. Ground truth: $\boldsymbol{\theta} = (36, 0.108, 0.5, 1000, 3)$.

Next, we plot the learned distributions of the trajectories $\boldsymbol{X}$, which we visualize using the inferred posterior mean and 95% credible interval, plotted on top of the true trajectory.

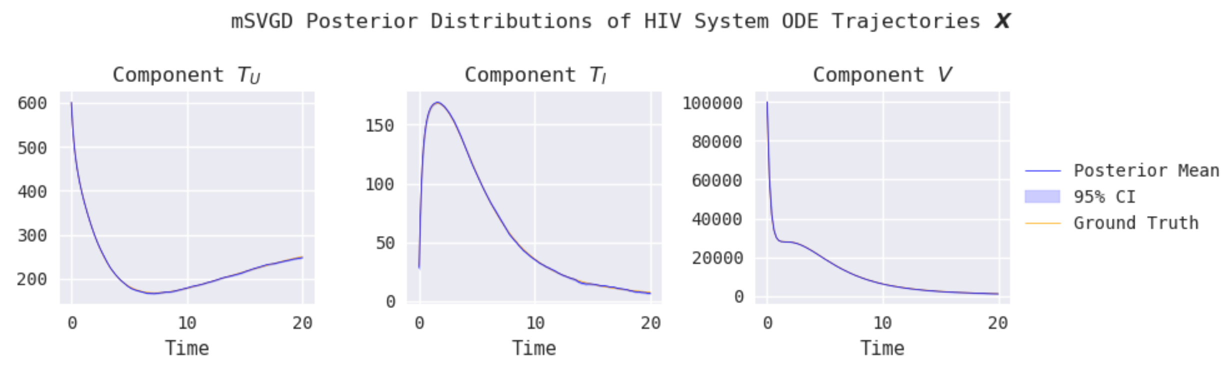


Figure 5.4.4: Posterior distribution of HIV model system trajectories $\boldsymbol{X}$ learned from mSVGD for dataset with NumPy seed 2025.

Finally, similar to $\boldsymbol{\theta}$, we also plot a histogram depicting the learned distributions of any unknown $\boldsymbol{\sigma}$, which were relatively poorly estimated in this example.

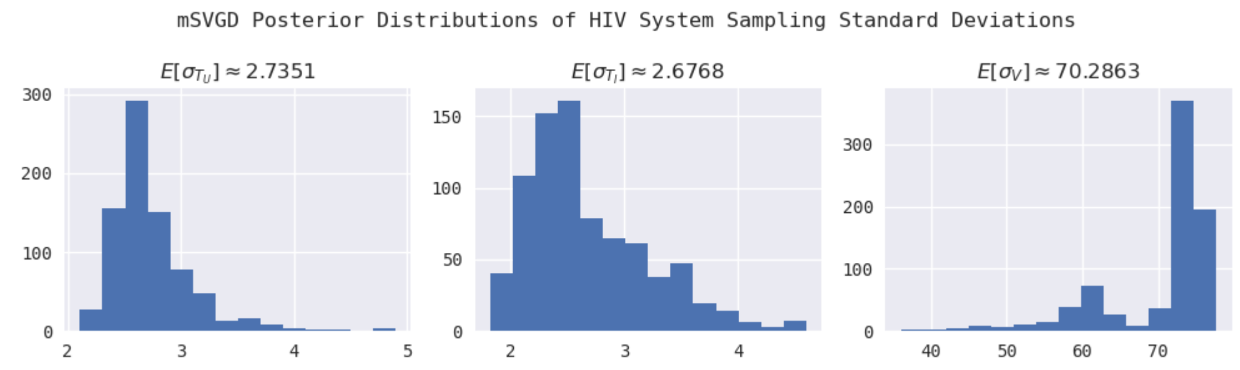


Figure 5.4.5: Posterior distribution of HIV model sampling standard deviations $\boldsymbol{\sigma}$ learned from mSVGD for dataset with NumPy seed 2025. Ground truth: $\boldsymbol{\sigma} \approx (3.162, 3.162, 10)$.

Recall that we set the ground truth values $\boldsymbol{\theta} = (36, 0.108, 0.5, 1000, 3)$. mSVGD learned posterior mean estimates $\hat{\boldsymbol{\theta}} \approx (34.486, 0.102, 0.499, 999.886, 3.440)$, accurately inferring the ODE parameters. Notice also that the plotted inferred trajectories (blue) almost exactly cover the ground truth trajectories (orange), showing that mSVGD also accurately recovered the system trajectories. In this example, the learned distributions of the parameters N and c were somewhat strange, and the standard deviations $\boldsymbol{\sigma}$ were relatively poorly inferred due to the scale of the V component, but such inaccuracies are not universal phenomena.

The initialization procedure when instantiating the `MAGISolver` object took about 20 seconds — longer than most models — the particle preparation in `initialize_particles` took about 0.01 seconds, and the mSVGD run in `solve` took about 8 seconds, giving a total runtime of less than 30 seconds. More rigorous and detailed performance results are discussed in Chapter 6.

6 Performance and Results

6.1 ODE Inference Evaluation Metrics

The following metrics are the evaluation metrics defined and used in the original MAGI paper [13]. We discuss other possible metrics in Section 7.2. I was unable to reproduce the performance numbers in the MAGI paper on the hardware available to me, so I will compare my results to the results reported in the paper, rather than re-running the experiments. For details on the mSVGD configuration and hyperparameter settings for each of the test cases, see Appendix C.

6.1.1 Speed Performance

We define the **runtime** of an ODE inference method to be the duration of real-world computational time required to run the algorithm itself. This is approximately equivalent to the "hands off" time for the user.

Runtime excludes the time required for pre-processing steps such as importing packages, defining the ODE, setting hyperparameters, and formatting data. It also excludes the time required for post-processing steps such as summarizing or visualizing results. It *does* include any time required to solve for initialization states.

Naturally, we consider a small runtime to be more desirable than a large runtime.

6.1.2 Parameter Inference Accuracy

When evaluating the accuracy of inference on $\boldsymbol{\theta}$, we consider the performance of a point estimator for each entry $\theta_j \in \boldsymbol{\theta}$. We define the evaluation metric **parameter root mean squared error** (PRMSE). After choosing a test setting — in particular, setting a ground-truth $\boldsymbol{\theta}$ — for each $\theta_j \in \boldsymbol{\theta}$, we take the root mean squared error of its point estimate from its ground-truth across $B = 100$ randomly generated datasets.

$$\text{PRMSE}(\theta_j) = \sqrt{\frac{1}{B}\sum_{i=1}^{B}(\hat{\theta}_{j,i} - \theta_j)^2}$$

6.1.3 Trajectory Recovery Accuracy

In some cases, a dynamical system may be insensitive to some of the parameters in $\boldsymbol{\theta}$, leading to poor inference of these values. In extreme cases, some parameters may not be fully identifiable given only the observed data. However, it is still possible for drastically incorrect estimates of some parameters to yield correct or reasonable trajectory inference. Therefore, it is both necessary and informative to consider trajectory recovery accuracy to account for possible parameter insensitivity and to measure how well the system components can be inferred.

Under the MAGI framework, the trajectory is estimated at a set of discrete time points $\boldsymbol{I} \subset [0, T]$ such that the observed times $\boldsymbol{\tau} \subseteq \boldsymbol{I}^D$. Given a test setting with a ground-truth $\boldsymbol{\theta}$ and an initial condition $\boldsymbol{x}(0)$, we can use numerical integration to solve for the ground-truth trajectory $\boldsymbol{x}$. Then, we can compute the **mean trajectory root mean squared error** (TRMSE) for each $x_d \in \boldsymbol{x}$ by comparing its estimates to its ground truth values at each observation time $t \in \boldsymbol{\tau}_d$, then averaging over the $B = 100$ datasets.

$$\text{MTRMSE}(x_d) = \frac{1}{B}\sum_{i=1}^{B}\sqrt{\frac{1}{|\boldsymbol{\tau}_d|}\sum_{t \in \boldsymbol{\tau}_d}(\hat{x}_d(t) - x_d(t))^2}$$

Remark. For MAGI, the point estimates used are the posterior means. Smaller PRMSE and MTRMSE are considered to be better.

6.2 FitzHugh-Nagumo Model

The FitzHugh-Nagumo (FN) equations are an ion channel model that describes spike potentials. The system consists of $\boldsymbol{X} = (V, R)$, where V defines the voltage of the neuron membrane potential and R is the recovery variable from neuron currents. $\boldsymbol{X}$ satisfies the ordinary differential equation system

$$\boldsymbol{f}(\boldsymbol{X}, \boldsymbol{\theta}, t) = \begin{pmatrix} c\left(V - \frac{V^3}{3} + R\right) \\ -\frac{1}{c}\left(V - a + bR\right) \end{pmatrix} \tag{6.1}$$

where $\boldsymbol{\theta} = (a, b, c)$ are the associated parameters. As in the the MAGI paper [13], we set the ground truth parameters $a = 0.2$, $b = 0.2$, and $c = 3$ and generate the true trajectories from initial conditions $V(0) = -1$ and $R(0) = 1$. We simulate datasets under additive Gaussian noise with known $\sigma_V = \sigma_R = 0.2$ and sample 41 evenly spaced observations of each component at times $\boldsymbol{\tau}_d = (0, 0.5, \ldots, 20)$.

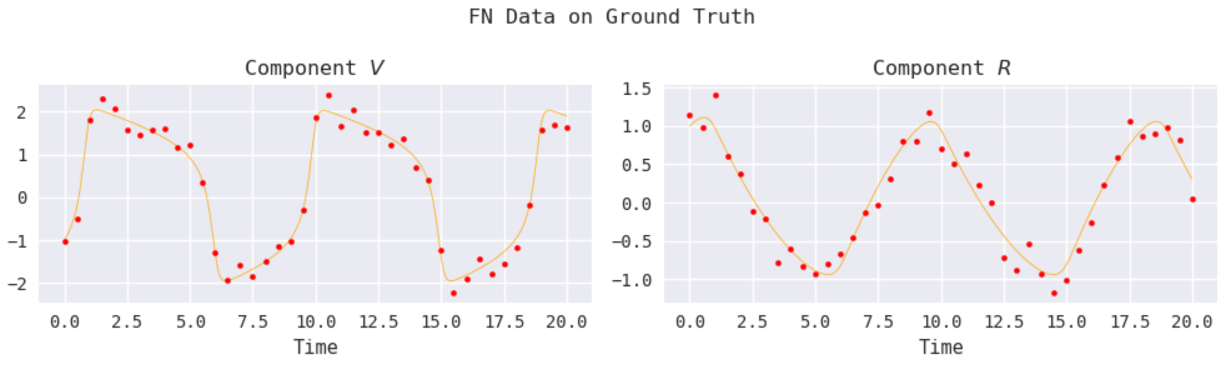


Figure 6.2.1: FN ground truth trajectories and dataset with NumPy seed 2025.

To facilitate computation, we also derive the gradients of $\boldsymbol{f}$ with respect to $\boldsymbol{X}$ and $\boldsymbol{\theta}$.

$$\frac{\partial \boldsymbol{f}(\boldsymbol{X}, \boldsymbol{\theta}, t)}{\partial \boldsymbol{X}} = \begin{pmatrix} c(1 - V^2) & c \\ -1/c & -b/c \end{pmatrix} \tag{6.2}$$

$$\frac{\partial \boldsymbol{f}(\boldsymbol{X}, \boldsymbol{\theta}, t)}{\partial \boldsymbol{\theta}} = \begin{pmatrix} 0 & 0 & V - V^3/3 + R \\ 1/c & -R/c & (V - a + bR)/c^2 \end{pmatrix} \tag{6.3}$$

6.2.1 Varying Level of Discretization

To compare mSVGD to the HMC implementation in the existing MAGI package, we consider the test settings of a varying number of discretization time points in $\boldsymbol{I}$, $n = |\boldsymbol{I}| \in \{41, 81, 161, 321\}$. We then compare the runtime, PRMSE, and MTRMSE to those reported in Ref. [13], over $B = 100$ datasets. For details on mSVGD configuration settings, see Appendix C.1.

As discussed in Section 2.1, as $\boldsymbol{I}$ grows dense in $[0, T]$, the practically-computable manifold constraint $W_{\boldsymbol{I}}$ converges to the theoretical manifold constraint W monotonically. The MAGI paper suggests increasing the denseness of $\boldsymbol{I}$ until the results stabilize, the algorithm begins to yield diminishing returns, or computation time exceeds acceptable limits [13]. The concern regarding the last point is significantly reduced by mSVGD.

		PRMSE			MTRMSE		Runtime
$\lvert\boldsymbol{I}\rvert$	Algorithm	a	b	c	V	R	(seconds)
41	HMC	0.026	0.091	0.211	0.358	0.146	50.4
41	mSVGD	0.025	0.110	0.135	0.107	0.062	**1.3**
81	HMC	0.020	0.165	0.199	0.270	0.142	100.2
81	mSVGD	0.033	0.148	0.150	0.081	0.057	**1.9**
161	HMC	0.020	0.172	0.128	0.103	0.070	187.8
161	mSVGD	0.017	0.118	0.076	0.066	0.052	**5.3**
321	HMC	0.020	0.162	0.097	0.072	0.051	356.4
321	mSVGD	0.019	0.090	0.041	0.065	0.049	**11.7**

Table 6.2.1: Results of FN model inference with varying number of discretization points $\{41, 81, 161, 321\}$ equally spaced in time, based on $B = 100$ simulated datasets for each test case. The HMC results are the numbers reported in Ref. [13].

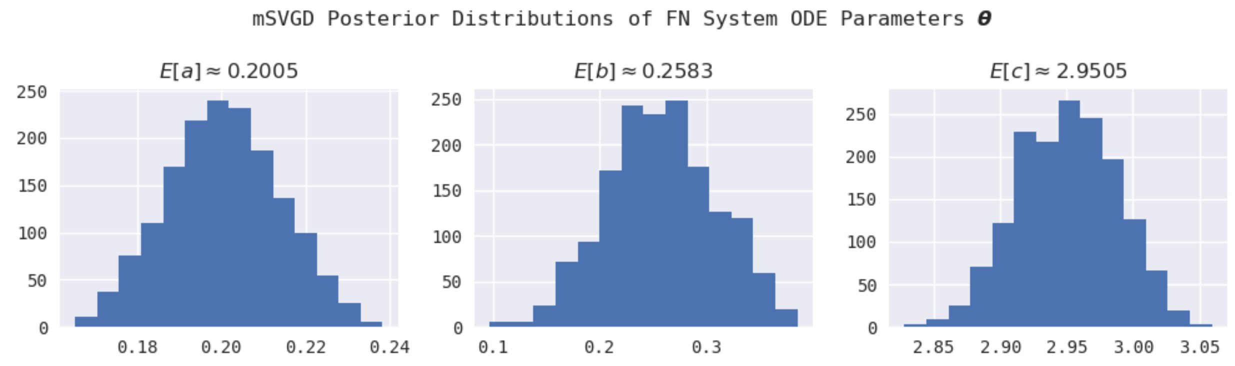


Figure 6.2.2: Posterior distribution of FN system parameters $\boldsymbol{\theta}$ learned from mSVGD for dataset with NumPy seed 2025. Ground truth: $\boldsymbol{\theta} = (0.2, 0.2, 3.0)$.

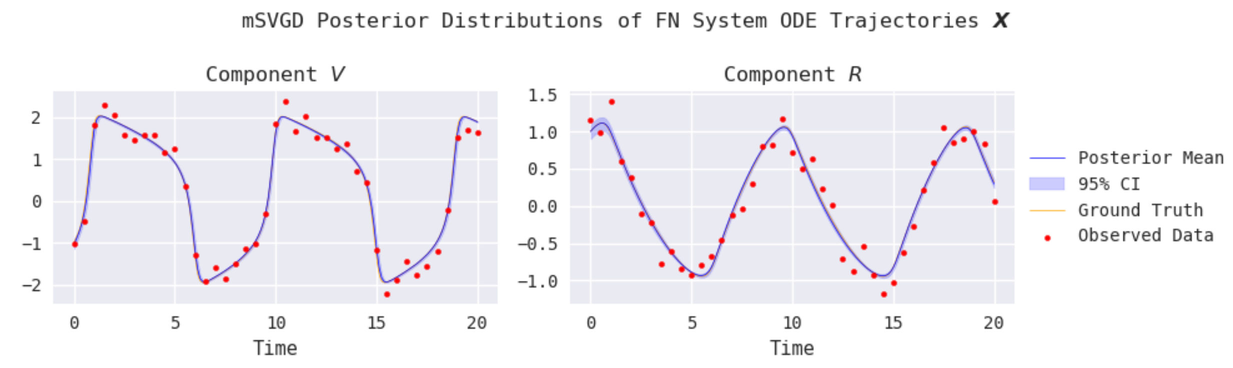


Figure 6.2.3: Posterior distribution of FN system trajectories $\boldsymbol{X}$ learned from mSVGD for dataset with NumPy seed 2025.

6.2.2 Fewer Observations

We can also compare inference on the FN model in a sparser setting, sampling $N_d = 21$ evenly spaced observations in each component, instead of the 41 observations in the previous section.

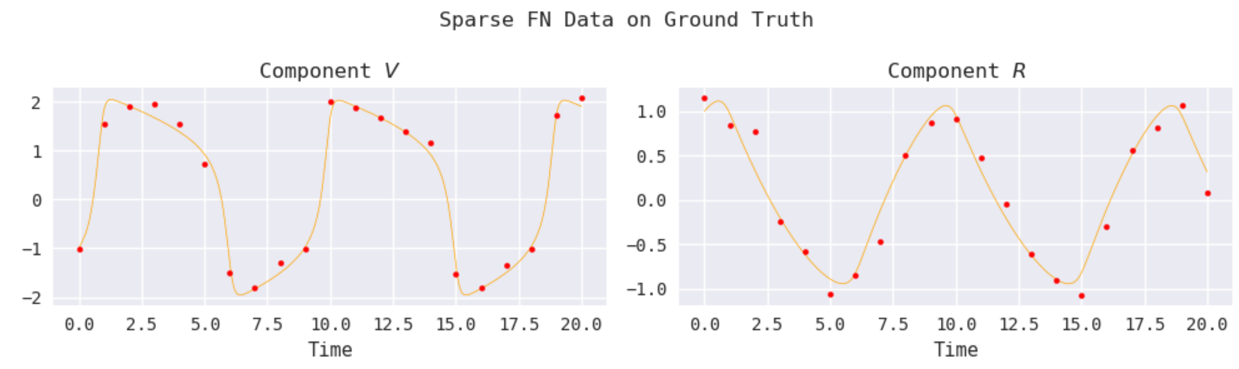


Figure 6.2.4: FN ground truth trajectories and sparse dataset with NumPy seed 2025.

This test setting was included in the MAGI paper to answer two questions: (1) how does MAGI perform when the number of observations is more sparse, and (2) how does MAGI perform if the observation time points are spaced farther apart?

$\|\boldsymbol{I}\|$	Algorithm	PRMSE a	PRMSE b	PRMSE c	MTRMSE V	MTRMSE R	Runtime (seconds)
321	HMC	0.029	0.280	0.261	0.128	0.107	348.6
321	mSVGD	0.039	0.171	0.189	0.130	0.070	**11.1**

Table 6.2.2: Results of FN model inference with sparser observations at times $\{0, 1, \ldots, 20\}$, based on $B = 100$ simulated datasets. The HMC results are the numbers reported in Ref. [13].

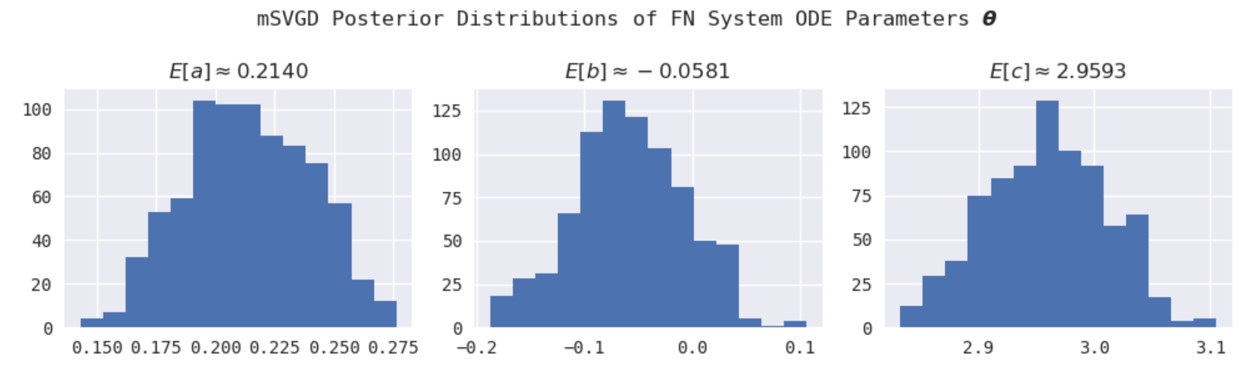


Figure 6.2.5: Posterior distribution of FN system parameters $\boldsymbol{\theta}$ learned from mSVGD for sparse dataset with NumPy seed 2025. Ground truth: $\boldsymbol{\theta} = (0.2, 0.2, 3.0)$.

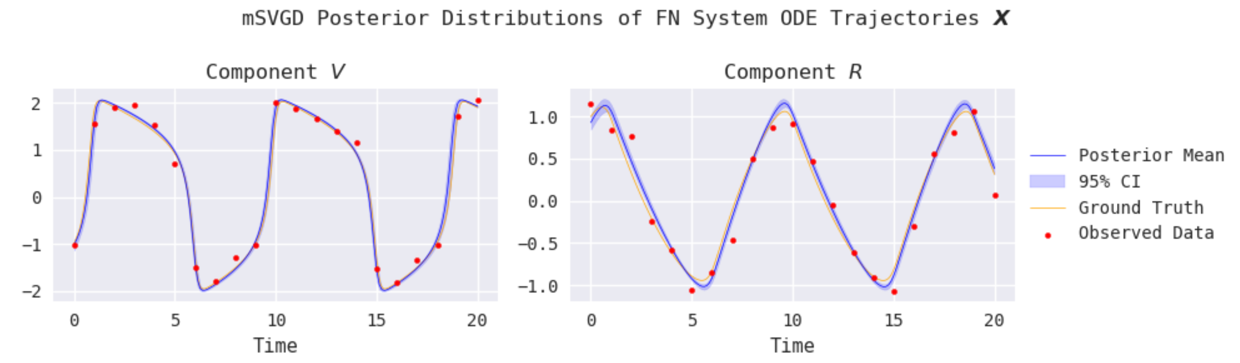


Figure 6.2.6: Posterior distribution of FN system trajectories $\boldsymbol{X}$ learned from mSVGD for sparse dataset with NumPy seed 2025.

6.2.3 Discussion of Results

While it appears from the tables of results that mSVGD provides more accurate inference than HMC, we cannot say for certain whether this is due to improper convergence by HMC or simply variance in the simulated datasets. However, we can say with certainty that mSVGD does not give worse inference than HMC, and that it runs much, much faster.

While the inference was accurate, stabilizing the computation required minor tuning, particularly for the $|\boldsymbol{I}| = 161$ and $|\boldsymbol{I}| = 321$ cases. We modified the number of mitosis splits and the maximum number of iterations per split. Empirically, mSVGD seems to give faster and better results if it is tuned to run all `max_iter` steps for earlier splits, and only rely on tolerance-based early stopping criteria in later splits. The key is to ensure that `max_iter` is large enough for the earlier splits to sufficiently converge while being small enough not to encounter diminishing returns.

Remark. Tuning is required for each test setting. After doing so, mSVGD remains stable regardless of the simulated dataset.

Overall, while there was some minimal tuning in the larger test cases, mSVGD required very little user intervention for these test cases. Given the equally or more accurate inference and significantly reduced computation time, we conclude that our mSVGD package solidly outperforms the existing HMC implementation for the FN model.

6.3 Hes1 Protein Model

The Hes1 model is a dynamical system that governs the oscillation of Hes1 mRNA (M) and Hes1 protein (P) levels in cultured cells, where it is postulated that a Hes1-interacting factor (H) contributes to a stable oscillation, a manifestation of biological rhythm. The ODEs of the three-component system $\boldsymbol{X} = (P, M, H)$ are

$$\boldsymbol{f}(\boldsymbol{X}, \boldsymbol{\theta}, t) = \begin{pmatrix} -aPH + bM - cP \\ -dM + \frac{e}{1+P^2} \\ -aPH + \frac{f}{1+P^2} - gH \end{pmatrix} \tag{6.4}$$

where $\boldsymbol{\theta} = (a, b, c, d, e, f, g)$ are the associated parameters. The main goal for this model is to infer these parameters. a and b govern the rate of protein synthesis in the presence of the interacting factor, c, d, and g are the rates of decomposition, and e and f are inhibition rates. As in the MAGI paper [13], we set $\boldsymbol{\theta} = (0.022, 0.3, 0.031, 0.028, 0.5, 20, 0.3)$. For this system, the parameters in $\boldsymbol{\theta}$ are strictly positive, so we set independent flat priors on $(0, \infty)$. For `magi_msvgd` in practice, this equates to setting the `pos_theta` flag to `True`.

P is observed every 15 minutes at times $\{0, 15, \dots, 240\}$, M is observed every 15 minutes at times $\{7.5, 22.5, \dots 232.5\}$, and H is never observed. Note that P and M are asynchronous and never observed at the same time. The initial condition is set to be $P(0) = 1.438575$, $M(0) = 2.037488$, and $H(0) = 17.90385$. The simulation noise for both P and M are set to be multiplicative following a log-Normal distribution with known standard deviations $\sigma_d = 0.15$.

Since the components in $\boldsymbol{X}$ are strictly positive and subject to multiplicative log-Normal noise, we apply a log-transformation so that the resulting noise is additive Gaussian with standard deviation $\sigma_d = 0.15$. Define the transformed $\tilde{\boldsymbol{X}}$ with components

$$\tilde{P} = \log P, \qquad \tilde{M} = \log M, \qquad \tilde{H} = \log H$$

with initial conditions $\tilde{P}(0) = 0.36365304$, $\tilde{M}(0) = 0.71171768$, and $\tilde{H}(0) = 2.88501577$. Then, applying the chain rule $\frac{d(\log u)}{dt} = \frac{du}{dt} \cdot \frac{1}{u}$, we work with the transformed system, which satisfies the ODEs

$$\boldsymbol{f}(\tilde{\boldsymbol{X}}, \boldsymbol{\theta}, t) = \begin{pmatrix} -aH + bM/P - c \\ -d + \frac{e}{(1+P^2)M} \\ -aP + \frac{f}{(1+P^2)H} - g \end{pmatrix} \tag{6.5}$$

Below, we plot the ground truth trajectories and an example dataset generated with the NumPy seed 2025. Note that while we work entirely with the log-transformed trajectories and data, both are exponentiated for plotting to visualize on the original scale.

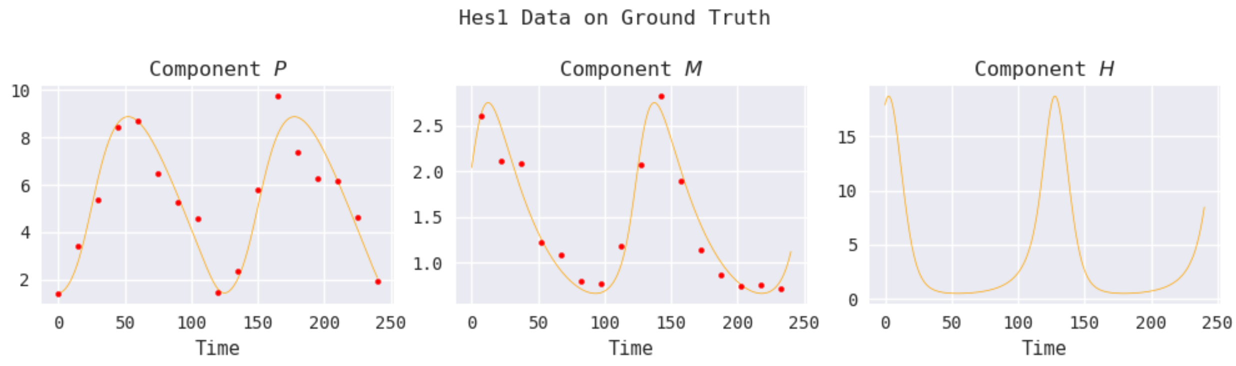


Figure 6.3.1: Hes1 ground truth trajectories and dataset with NumPy seed 2025.

To facilitate computation, we also derive the gradients of the transformed $\boldsymbol{f}$ with respect to $\tilde{X}$ and $\boldsymbol{\theta}$.

$$\frac{\partial \boldsymbol{f}(\tilde{\boldsymbol{X}}, \boldsymbol{\theta}, t)}{\partial \tilde{\boldsymbol{X}}} = \begin{pmatrix} -bM/P & bM/P & -aH \\ -\frac{2eP^2}{(1+P^2)^2 M} & -\frac{e}{(1+P^2)M} & 0 \\ -aP - \frac{2fP^2}{(1+P^2)^2 H} & 0 & -\frac{f}{(1+P^2)H} \end{pmatrix} \tag{6.6}$$

$$\frac{\partial \boldsymbol{f}(\tilde{\boldsymbol{X}}, \boldsymbol{\theta}, t)}{\partial \boldsymbol{\theta}} = \begin{pmatrix} -H & M/P & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & \frac{1}{(1+P^2)^2 M} & 0 & 0 \\ -P & 0 & 0 & 0 & 0 & \frac{1}{(1+P^2)^2 H} & -1 \end{pmatrix} \tag{6.7}$$

6.3.1 Experiment Setting

The Hes1 model is a much more challenging test setting than the FN model. The larger number of components in $\boldsymbol{X}$, the larger number of parameters in $\boldsymbol{\theta}$, the asynchronous nature of P and M, the sparser observations, and the completely unobserved H make this an inherently more difficult inference problem. Additionally, the log-transformation, the multiplicative noise, the more complex ODEs and gradients, and the fact that the components of $\boldsymbol{X}$ and components of $\boldsymbol{\theta}$ are all on very different numerical scales makes computation less stable.

To compare mSVGD to the HMC implementation in the existing MAGI package, we consider the test setting used in Ref. [13]. We set the discretization $\boldsymbol{I} = \boldsymbol{\tau}_1 \cup \boldsymbol{\tau}_2$ $= \{0, 7.5, 15, 22.5, \ldots, 232.5, 240\}$. We then compare the runtime, PRMSE, and MTRMSE to those reported in Ref. [13], over $B = 100$ datasets. For details on mSVGD configuration settings, see Appendix C.2.

		PRMSE						
$\lvert\boldsymbol{I}\rvert$	Algorithm	a	b	c	d	e	f	g
36	HMC	0.003	0.059	0.007	0.003	0.090	6.936	0.162
36	mSVGD	0.001	0.036	0.005	0.003	0.094	0.539	0.028

		MTRMSE			Runtime
$\lvert\boldsymbol{I}\rvert$	Algorithm	P	M	H	(seconds)
36	HMC	0.97	0.21	2.57	372.0
36	mSVGD	0.37	0.12	1.11	**11.2**

Table 6.3.1: Results of Hes1 model inference based on $B = 100$ simulated datasets. The HMC results are the numbers reported in Ref. [13].

6.3.2 Discussion of Results

Again, mSVGD provides inference that is more accurate than that of HMC at a significantly reduced runtime. mSVGD required slightly more tuning for the Hes1 model

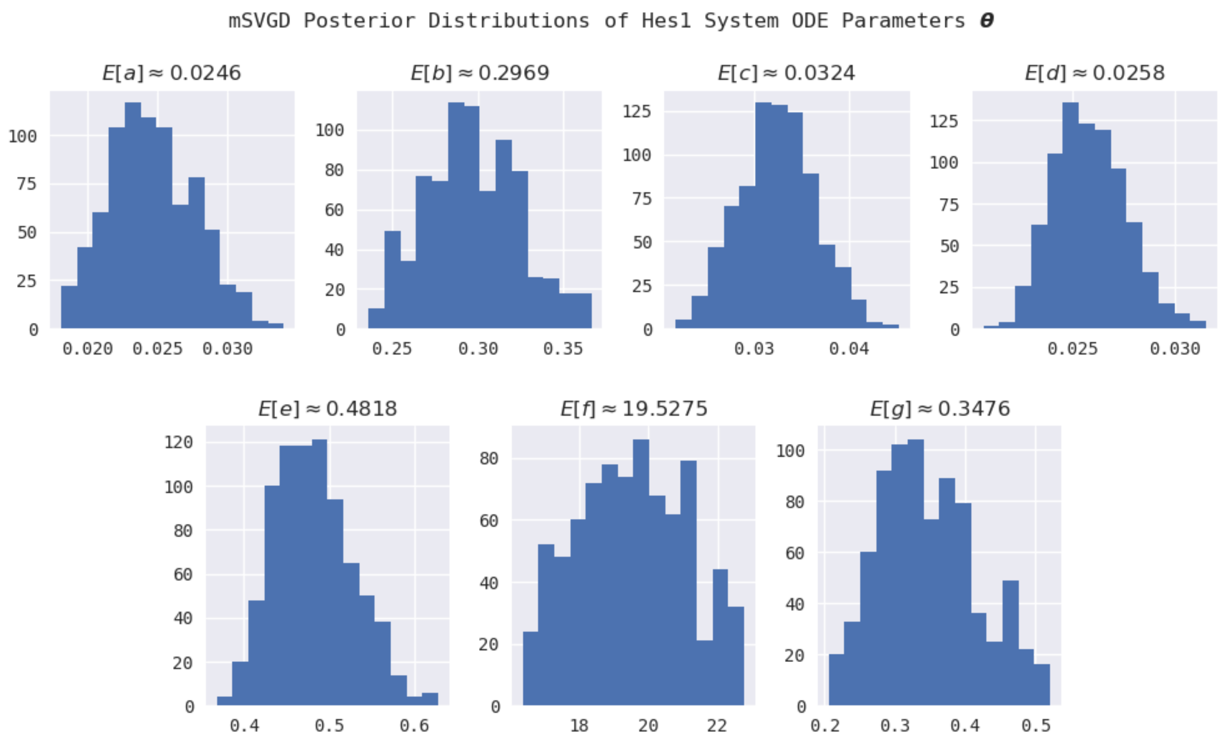


Figure 6.3.2: Posterior distribution of Hes1 system parameters $\boldsymbol{\theta}$ learned from mSVGD for dataset with NumPy seed 2025. Ground truth: $\boldsymbol{\theta} = (0.022, 0.3, 0.031, 0.028,\ 0.5, 20, 0.3)$.

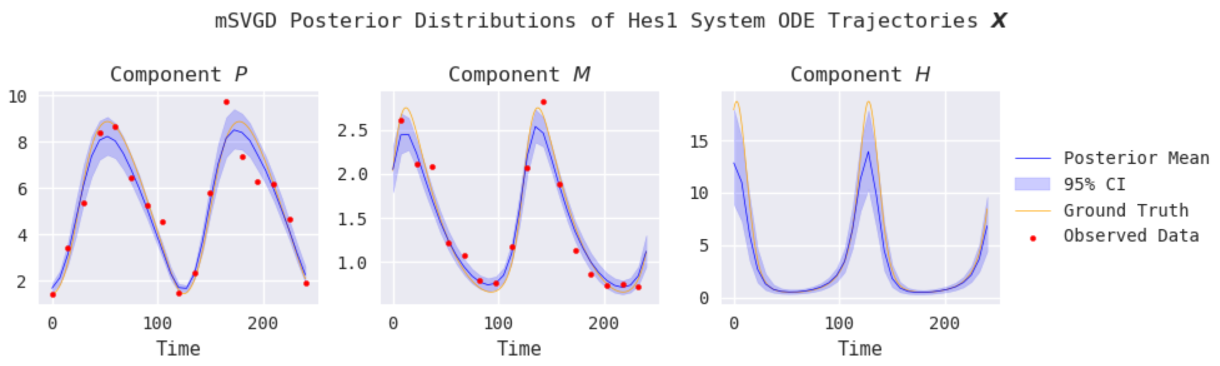


Figure 6.3.3: Posterior distribution of Hes1 system trajectories $\boldsymbol{X}$ learned from mSVGD for dataset with NumPy seed 2025.

than for the FN model. As discussed above, the Hes1 model is a more difficult inference with a complex target posterior with strong local modes. This was what inspired the addition of the `X_guess` setting, which has empirically demonstrated the ability to escape local modes during initialization by recursively solving for the initialization state of unknown components in $\boldsymbol{X}$ while resetting $\boldsymbol{\theta}$.

Due to the log-transformation, attempting to set early stopping criteria was very unstable. Absolute tolerance was additive with respect to $\boldsymbol{\theta}$ and $\tilde{\boldsymbol{X}}$, but multiplicative with respect to the original trajectories $\boldsymbol{X}$. Likewise, relative tolerance was multiplicative with respect to $\boldsymbol{\theta}$ and $\tilde{\boldsymbol{X}}$, but polynomial with respect to the original trajectories $\boldsymbol{X}$. This led to poor stopping behavior, so we set both `atol=0` and `rtol=0` to disable tolerance-based early stopping, and simply allowed mSVGD to run the number of iterations described by `max_iter`. The challenge was to tune `max_iter` to be large enough for the earlier splits to sufficiently converge while being small enough to minimize computation time.

Remark. Tuning is required for the Hes1 model setting as a whole. After doing so, mSVGD remains stable regardless of the simulated dataset.

While mSVGD provides more accurate inference and has significantly faster computation speed, the existing HMC-based package requires slightly less hand-on tuning, which is a point in its favor. We conclude that solidly mSVGD outperforms HMC, though the ease of package usage is less one-sided.

6.4 Lorenz Model

The Lorenz model is a system of ODEs that was first studied by American mathematician and meteorologist Edward Lorenz as a simplified model of weather patterns. It is notable for having chaotic solutions for certain parameter values and initial conditions, a phenomenon closely associated with the term "butterfly effect." This means that tiny changes in parameters or initial conditions may result in completely different trajectories, making it a system whose behavior is difficult to learn or predict.

The system describes the properties of a two-dimensional fluid layer uniformly warmed from below and cooled from above, relating the rate of convection (x), the horizontal temperature variation (y) and the vertical temperature variation (z). The three-component system $\boldsymbol{X} = (x, y, z)$ follows the ODEs

$$\boldsymbol{f}(\boldsymbol{X}, \boldsymbol{\theta}, t) = \begin{pmatrix} \sigma(y - x) \\ x(\rho - z) - y \\ xy - \beta z \end{pmatrix} \tag{6.8}$$

where $\boldsymbol{\theta} = (\beta, \rho, \sigma)$ are the associated parameters. These are the variable names conventionally used for the Lorenz system, though there exists a somewhat confusing near-overlap with variables that we have already defined for MAGI. I will ensure that the distinction is clear when referring to either.

We set the ground truth parameters $\boldsymbol{\theta} = (8/3, 28, 10)$ and generate the true trajectories from the initial conditions $x(0) = y(0) = z(0) = 2$. We simulate datasets under additive

Gaussian noise with unknown $\boldsymbol{\sigma} = (2.96546738, 3.78528167, 4.52163049)$ and sample 26 evenly spaced observations of each component at times $\boldsymbol{\tau}_d = (0, 0.1, 0.2, \ldots, 2.5)$.

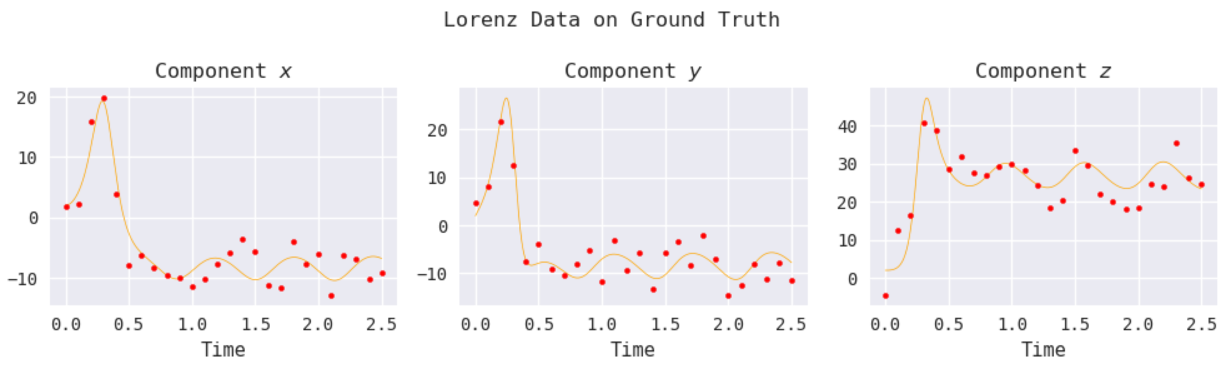


Figure 6.4.1: Lorenz ground truth trajectories and dataset with NumPy seed 2025.

To facilitate computation, we also derive the gradients of $\boldsymbol{f}$ with respect to $\boldsymbol{X}$ and $\boldsymbol{\theta}$.

$$\frac{\partial \boldsymbol{f}(\boldsymbol{X}, \boldsymbol{\theta}, t)}{\partial \boldsymbol{X}} = \begin{pmatrix} -\sigma & \sigma & 0 \\ \rho - z & -1 & -x \\ y & x & -\beta \end{pmatrix} \tag{6.9}$$

$$\frac{\partial \boldsymbol{f}(\boldsymbol{X}, \boldsymbol{\theta}, t)}{\partial \boldsymbol{\theta}} = \begin{pmatrix} 0 & 0 & y - z \\ 0 & x & 0 \\ -z & 0 & 0 \end{pmatrix} \tag{6.10}$$

6.4.1 Experiment Setting

This test case demonstrates the capability of MAGI to handle chaotic ODE systems and experimental settings with large, unknown noise levels. We also compare the performance of mSVGD to standard SVGD, which is equivalent to mSVGD restricted to $M = 0$. We set $4\times$ discretization such that $\boldsymbol{I} = (0, 0.025, 0.05, \ldots, 2.5)$, giving us $n = |\boldsymbol{I}| = 101$ total discretization points.

To compare mSVGD to SVGD, we first run mSVGD on the Lorenz model. Then, we record the results of different SVGD tuning settings:

- (SVGD 1) Approximately the same runtime as mSVGD with the same number of total particles (`k=3200`, $t \approx 12.2s$).

- (SVGD 2) Approximately the same runtime as mSVGD with fewer total particles (`k=400`, $t \approx 12.2s$).
- (SVGD 3) The same absolute tolerance stopping criterion with the same number of total particles (`k=3200`, `atol=0.005`).
- (SVGD 4) The same absolute tolerance stopping criterion with fewer total particles (`k=400`, `atol=0.005`).
- (SVGD 5) A more lenient absolute tolerance stopping criterion with the same number of total particles (`k=3200`, `atol=0.05`).

	PRMSE			MTRMSE			$\boldsymbol{\sigma}$-RMSE			Runtime
Setting	β	ρ	σ	x	y	z	σ_β	σ_ρ	σ_σ	(seconds)
mSVGD	0.303	1.212	2.097	0.899	1.201	1.763	0.437	0.618	0.567	12.2
SVGD 1	0.865	1.824	5.940	2.640	2.967	3.761	3.862	3.049	2.368	12.5
SVGD 2	0.359	1.261	2.528	1.251	1.850	2.277	3.437	2.066	2.032	12.6
SVGD 3	–	–	–	–	–	–	–	–	–	DNF*
SVGD 4	–	–	–	–	–	–	–	–	–	DNF*
SVGD 5	0.555	1.298	2.271	1.128	2.420	1.996	3.035	2.514	1.831	126.9

Table 6.4.1: Results of Lorenz model inference for mSVGD and various tuning settings of SVGD, based on $B = 100$ simulated datasets. Approximately 6 seconds are spent on the initialization procedure, with the rest of the runtime spent running mSVGD/SVGD.
* DNF indicates that there was no consistent convergence behavior within the 600 seconds time-out.

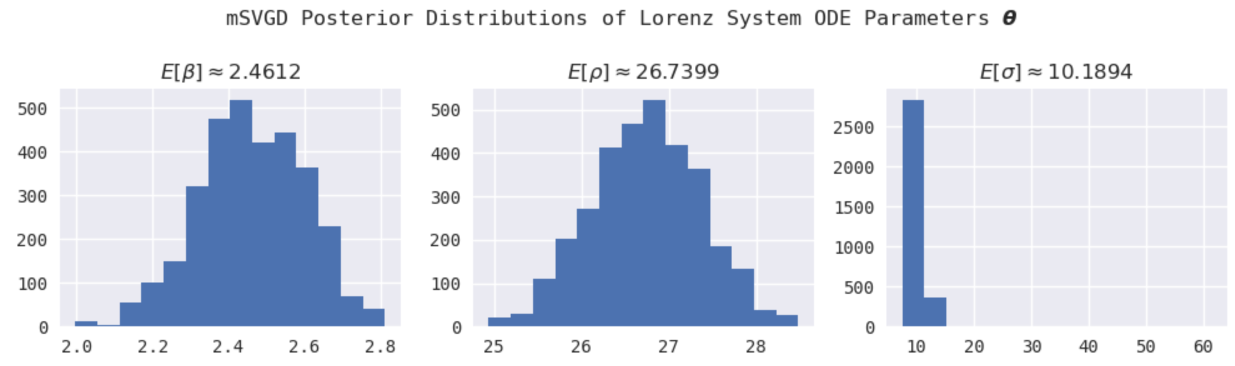


Figure 6.4.2: Posterior distribution of Lorenz system parameters $\boldsymbol{\theta}$ learned from mSVGD for dataset with NumPy seed 2025. Ground truth: $\boldsymbol{\theta} \approx (2.667, 28, 10)$.

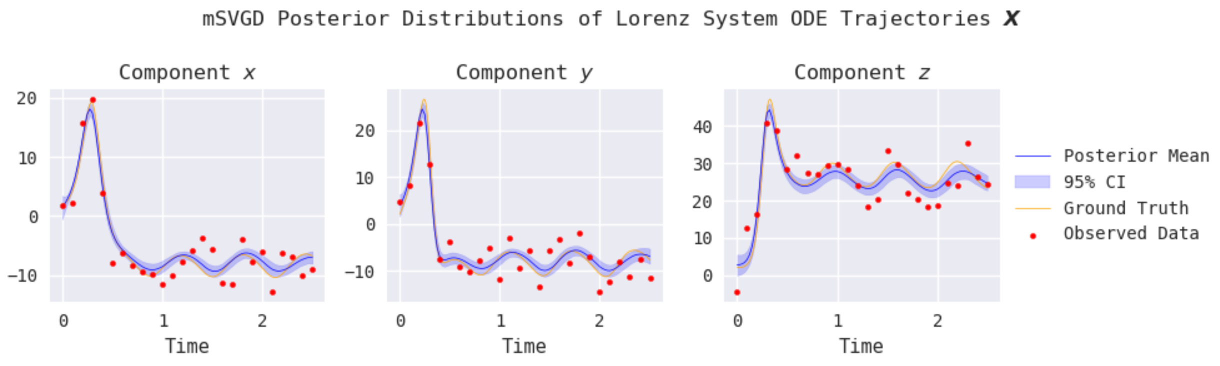


Figure 6.4.3: Posterior distribution of Lorenz system trajectories $\boldsymbol{X}$ learned from mSVGD for dataset with NumPy seed 2025.

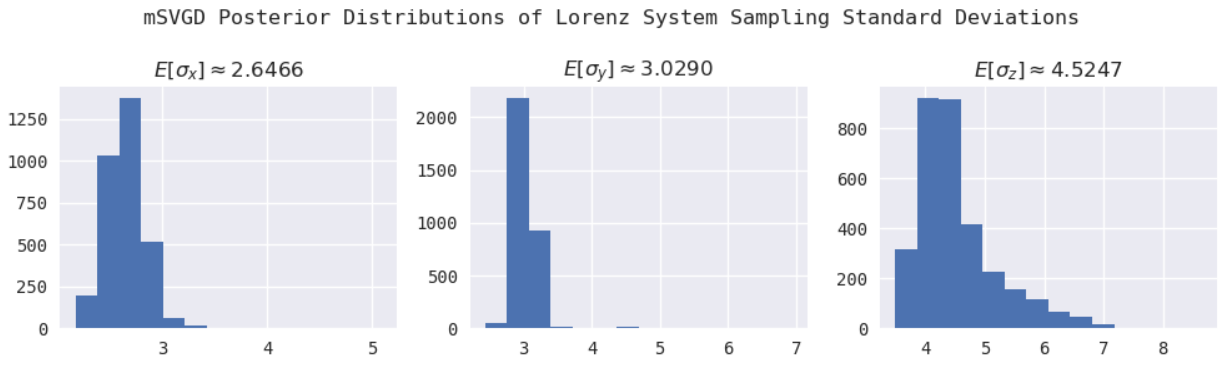


Figure 6.4.4: Posterior distribution of Lorenz system sampling standard deviations $\boldsymbol{\sigma}$ learned from mSVGD for dataset with NumPy seed 2025. Ground truth: $\boldsymbol{\sigma} \approx (2.965, 3.785, 4.522)$.

6.4.2 Discussion of Results

Based on the results, we observe that mSVGD provides more consistent and faster convergence than standard SVGD. Cases 1 and 2 of SVGD did not converge within the runtime benchmark set by mSVGD, resulting in much worse inference given the same time budget, especially when estimating the trajectories $\boldsymbol{X}$ and standard deviations $\boldsymbol{\sigma}$. Case 5 took much longer to converge than mSVGD, even to a far more lenient stopping tolerance, and still yielded poor inference for $\boldsymbol{\sigma}$. One trial in Case 5 had exploding gradients and never converged, and was discarded from the results. Cases 3 and 4 of SVGD did not converge consistently within 10 minutes. With some careful tuning, there were some trials that reached convergence, but the results were inconsistent across different hyperparameter settings, datasets, and particle initializations, justifying the DNF score. In contrast, we were able to take a one-setting-fits-all approach when using mSVGD, exemplifying its superior robustness.

7
Discussion, Future Work, and Conclusion

7.1 Discussion

This thesis introduced mitotic Stein variational gradient descent as an enhancement to Stein variational gradient descent, with the goal of improving convergence speed, consistency of convergence behavior, and robustness to hyperparameter choices and initialization conditions. The primary motivation was to increase computational efficiency while maintaining accuracy in manifold-constrained Gaussian process inference tasks for learning parameters and trajectories of ordinary differential equation models.

We presented a Python package implementing mSVGD and demonstrated its usage. Performance evaluations conducted on the FitzHugh-Nagumo, Hes1 protein, and Lorenz models highlighted several important insights. First, mSVGD achieved a speedup of approximately 30× to 50× over the existing MAGI package while maintaining comparable or better accuracy in parameter estimation and trajectory reconstruction. Second, mSVGD demonstrated resilience in handling sparse and noisy settings with complex posterior density landscapes, an essential characteristic for real-world applications of dynamical system inference. Finally, mSVGD provided a faster and more consistent convergence profile than standard SVGD, reducing the need for manual tuning of hyperparameters.

The primary shortcoming of mSVGD is the necessity of tuning hyperparameters, particularly the initial standard deviation of the particles, the gradient descent learning rate, and the maximum number of iterations per mitosis split. Although the existing HMC-based package requires less manual input, we assert that the performance benefits of mSVGD outweigh the costs. Furthermore, mSVGD was shown to be more robust to these hyperparameters than standard SVGD, requiring far less precise tuning to stabilize results and convergence behavior.

7.2 Future Work

Despite the strengths of mSVGD for MAGI, several limitations and open questions remain, some of which were discussed throughout the thesis. Beyond those mentioned, a few other prominent directions for interesting future work arise from this research.

7.2.1 Package Improvements

As discussed in the thesis, there is the potential for further optimization of the Python package, and fixes for the TensorFlow front-end.

It may also be good to implement an option to use automatic differentiation to compute the gradients of the ODE, so the user does not have to manually input `dfdx` and `dfdtheta`. The downside is that autograd is slower and less numerically stable, as seen from the gradient checker in `magi_msvgd.tests.test_helpers`. Another potential objective is to integrate classical MCMC methods — such as HMC — into the package, so a user would have multiple sampling schemes to choose from.

7.2.2 More Comparison Tests

Despite our gallery of tests, there are many questions left to be answered. Not only can we apply mSVGD to different ODE systems, we can also experiment with different hyperparameter settings and gradient descent algorithms. Note that we exclusively used the Adam optimizer for the tests in this thesis.

An interesting testbed that was not included in this thesis is forecasting for ODE systems, i.e. extrapolating trajectory construction beyond the observed time interval. There has been some investigation into forecasting with MAGI, which has been shown to be more accurate than other ODE inference frameworks. It would be interesting to see how mSVGD performs in this task.

Although PRMSE and MTRMSE are informative with respect to the accuracy and behavior of point estimators, our testing has left us with little insight into the spread of the particles. Like all Bayesian methods, MAGI yields interval estimators in the form of credible intervals. Thus, coverage probability is a natural evaluation metric to consider. A task for future work is to compare the coverage probability of intervals constructed by HMC, mSVGD, and SVGD.

7.2.3 mSVGD and SVGD

In addition to the investigation of the SVGD kernel and bandwidth parameter, there exist many other areas for future work surrounding mSVGD and SVGD. As mentioned in Section 4.3.2, two avenues of research for mSVGD are the study of theoretical properties and guarantees and the investigation of other particle splitting methods. Another is to implement automatic hyperparameter tuning methods to further reduce the already-minimal amount of required user intervention. Lastly, future research may incorporate other work on SVGD. For example, Chen and Ghattas proposed projected SVGD in 2020, which projects the target variables into a low-dimensional subspace of the parameter space [3]. This method has been shown to be more scalable with respect to the number of parameters, and has the potential to further stabilize and accelerate MAGI tasks.

7.3 Conclusion

This thesis presented mSVGD as an effective alternative to HMC for Bayesian inference of dynamical systems via MAGI, providing faster and more accurate results. It also showcased mSVGD as a powerful upgrade to standard SVGD. By leveraging a hierarchical particle expansion strategy, mSVGD successfully mitigates common challenges associated with particle-based variational inference, including sensitivity to initialization and slow convergence rates. Through comprehensive experimentation on benchmark ODE models, we demonstrated that mSVGD achieves superior robustness, efficiency, and scalability.

The contributions of this work extend beyond the application to dynamical systems. The insights gained from the development of mSVGD can inform broader research efforts in variational inference and optimization While further refinements and applications remain to be explored, the findings of this thesis underscore the potential of mSVGD as a powerful tool for Bayesian inference in scientific domains.

A
Miscellaneous Proofs

A.1 Proof of Theorem 4.2.0.1

Theorem (4.2.0.1)**.** *The* $\boldsymbol{\phi} \in \mathcal{H}^d$ *that maximizes* $\{[\mathbb{E}_{x\sim q}\mathrm{tr}\,(\mathcal{A}_p\boldsymbol{\phi}(x))]^2,\ \text{s.t.}\ \ \|\boldsymbol{\phi}\|_{\mathcal{H}^d} \leq 1\}$ *is*

$$\boldsymbol{\phi}(x) = \frac{\boldsymbol{\phi}^*_{q,p}(x)}{\left\|\boldsymbol{\phi}^*_{q,p}\right\|_{\mathcal{H}^d}},\ \text{where}\ \boldsymbol{\phi}^*_{q,p}(\cdot) = \mathbb{E}_{x\sim q}[\mathcal{A}_p\mathcal{K}(x,\cdot)] \tag{A.1}$$

with the KSD evaluating to

$$\mathbb{S}(q,p) = \left\|\boldsymbol{\phi}^*_{q,p}\right\|^2_{\mathcal{H}^d} \tag{A.2}$$

Proof. Let $\beta(\cdot) = \boldsymbol{\phi}^*_{q,p}(\cdot) = \mathbb{E}_{x\sim q}[\mathcal{A}_p\mathcal{K}(x,\cdot)]$. Let $s_r(x) = \nabla_x \log r(x)$ denote the score function of a density function r. We first prove Equation A.2 by applying the reproducing property $\mathcal{K}(x,x') = \langle\mathcal{K}(x,\cdot),\mathcal{K}(x',\cdot)\rangle_{\mathcal{H}}$ on the definition of KSD (Equation 4.4).

$$\begin{aligned}
\mathbb{S}(q,p) &= \mathbb{E}_{x,x'\sim q}[(s_p(x) - s_q(x))^\top\mathcal{K}(x,x')(s_p(x') - s_q(x'))] \\
&= \mathbb{E}_{x,x'\sim q}[(s_p(x) - s_q(x))^\top\langle\mathcal{K}(x,\cdot),\mathcal{K}(x',\cdot)\rangle_{\mathcal{H}}(s_p(x') - s_q(x'))] \\
&= \sum_{i=1}^{d}\langle\mathbb{E}_x[(s^i_p(x) - s^i_q(x))\mathcal{K}(x,\cdot)],\mathbb{E}_x[\mathcal{K}(x,\cdot)(s^i_p(x) - s^i_p(x))]\rangle_{\mathcal{H}} \\
&= \sum_{i=1}^{d}\langle\beta_i,\beta_i\rangle_{\mathcal{H}} = \|\beta\|^2_{\mathcal{H}^d}
\end{aligned}$$

Then, for a vector function $\mathbf{f} \in \mathcal{H}^d$,

$$\begin{aligned}
\langle \mathbf{f}, \beta \rangle_{\mathcal{H}^d} &= \sum_{i=1}^{d} \langle f_i, \mathbb{E}_{x\sim q}[s_p^i(x)\mathcal{K}(x,\cdot) + \nabla_{x_i}\mathcal{K}(x,\cdot]\rangle_{\mathcal{H}} \\
&= \sum_{i=1}^{d} \mathbb{E}_{x\sim q}[s_p^i(x)\langle f_i, \mathcal{K}(x,\cdot)\rangle_{\mathcal{H}} + \langle f_i, \nabla_{x_i}\mathcal{K}(x,\dot{)}\rangle_{\mathcal{H}}] \\
&= \sum_{i=1}^{d} \mathbb{E}_{x\sim q}[s_p^i(x) f_i(x) + \nabla_{x_i} f_i(x)] \\
&= \mathbb{E}_{x\sim q}[\operatorname{tr}(\mathcal{A}_p \mathbf{f}(x))]
\end{aligned}$$

If we constrain $\|\mathbf{f}\|_{\mathcal{H}^d} \leq 1$, then we see $\mathbb{E}_{x\sim q}[\operatorname{tr}(\mathcal{A}_p\mathbf{f}(x))]$ maximizes when $\mathbf{f} = \beta / \|\beta\|_{\mathcal{H}^d}$. Thus, $\beta/\|\beta\|_{\mathcal{H}^d}$ maximizes $\{[\mathbb{E}_{x\sim q}\operatorname{tr}(\mathcal{A}_p\boldsymbol{\phi}(x))]^2, \text{ s.t. } \|\boldsymbol{\phi}\|_{\mathcal{H}^d} \leq 1\}$. Note that β is in the Stein class of q because $\mathcal{K}(x,\cdot)$ and $\nabla_x\mathcal{K}(x,\dot{)}$ are in the Stein class of q for any fixed x. □

A.2 Proof of Theorem 4.2.1.1

Theorem (4.2.1.1)**.** *Let p be the target density. Let $z = T(x) = x + \epsilon\boldsymbol{\phi}(x)$ and $q_{[T]}(z)$ the density of z when $x \sim q(x)$. Then, the gradient with respect to ϵ of the KL divergence between $q_{[T]}$ and p can be related to the Stein operator as follows.*

$$\nabla_\epsilon \mathcal{D}_{KL}(q_{[T]} \,||\, p) \,|_{\epsilon=0} = -\mathbb{E}_{x\sim q}[\operatorname{tr}(\mathcal{A}_p\boldsymbol{\phi}(x))]$$

where $\mathcal{A}_p\boldsymbol{\phi}(x) = \nabla_x \log p(x)\boldsymbol{\phi}(x)^\top + \nabla_x\boldsymbol{\phi}(x)$ is the Stein operator.

Proof. Let $p_{[T^{-1}]}(z)$ be the density of $z = T^{-1}(x)$ when $x \sim p(x)$. By change of variables, we show invariance of KL divergence under parameter transformation.

$$\begin{aligned}
\mathcal{D}_{\text{KL}}(q_{[T]} \,||\, p) &= \int_z q_{[T]}(z) \log \frac{q_{[T]}(z)}{p(z)}\, dz \\
&= \int_{T^{-1}(z)} q(T^{-1}(z)) \left|\det(\nabla_x T^{-1}(z))\right| \log \frac{q(T^{-1}(z))\,|\det(\nabla_x T^{-1}(z))|}{p_{[T^{-1}]}(T^{-1}(z))\,|\det(\nabla_x T^{-1}(z))|}\, dz \\
&= \int_{T^{-1}(z)} q(T^{-1}(z)) \log \frac{q(T^{-1}(z))}{p_{[T^{-1}]}(T^{-1}(z))}\, dz \left|\det(\nabla_x T^{-1}(z))\right| \\
&= \int_x q(x) \log \frac{q(x)}{p_{[T^{-1}]}(x)}\, dx = \mathcal{D}_{\text{KL}}(q \,||\, p_{[T^{-1}]})
\end{aligned}$$

Then, using this identity and the Leibniz integral rule,

$$\begin{aligned}
\nabla_\epsilon \mathcal{D}_{\text{KL}}(q_{[T]} \,||\, p) &= \nabla_\epsilon \mathcal{D}_{\text{KL}}(q \,||\, p_{[T^{-1}]}) \\
&= \nabla_\epsilon \mathbb{E}_{x\sim q}[\log q(x)] - \nabla_\epsilon \mathbb{E}_{x\sim q}[\log p_{[T^{-1}]}(x)] \\
&= -\mathbb{E}_{x\sim q}[\nabla_\epsilon \log p_{[T^{-1}]}(x)]
\end{aligned}$$

The last step is to evaluate $\nabla_\epsilon \log p_{[T^{-1}]}(x)$. Let $s_p(x) = \nabla_x \log p(x)$. Then,

$$\nabla_\epsilon \log p_{[T^{-1}]}(x) = s_p(T(x))^\top \nabla_\epsilon T(x) + \mathrm{tr}\left((\nabla_x T(x))^{-1} \nabla_\epsilon \nabla_x T(x)\right)$$

When $T(x) = x + \epsilon \boldsymbol{\phi}(x)$ and $\epsilon = 0$, we have

$$T(x) = x, \qquad \nabla_\epsilon T(x) = \boldsymbol{\phi}(x), \qquad \nabla_x T(x) = I, \qquad \nabla_\epsilon \nabla_x T(x) = \nabla_x \boldsymbol{\phi}(x)$$

Then, we plug these in to solve for the gradient of the KL divergence.

$$\begin{aligned}
-\mathbb{E}_{x\sim q}[\nabla_\epsilon \log p_{[T^{-1}]}(x)] &= -\mathbb{E}_{x\sim q}[(\nabla_x \log p(x))^\top \boldsymbol{\phi}(x) + \mathrm{tr}\left(\nabla_x \boldsymbol{\phi}(x)\right)] \\
&= -\mathbb{E}_{x\sim q}[\mathrm{tr}\left((\nabla_x \log p(x))^\top \boldsymbol{\phi}(x)\right) + \mathrm{tr}\left(\nabla_x \boldsymbol{\phi}(x)\right)] \\
&= -\mathbb{E}_{x\sim q}[\mathrm{tr}\left(\nabla_x \log p(x) \boldsymbol{\phi}(x)^\top\right) + \mathrm{tr}\left(\nabla_x \boldsymbol{\phi}(x)\right)] \\
&= -\mathbb{E}_{x\sim q}[\mathrm{tr}\left(\nabla_x \log p(x) \boldsymbol{\phi}(x)^\top + \nabla_x \boldsymbol{\phi}(x)\right)] \\
&= -\mathbb{E}_{x\sim q}[\mathrm{tr}\left(\mathcal{A}_p \boldsymbol{\phi}(x)\right)]
\end{aligned}$$

□

B
TensorFlow Tutorial

First, we import the necessary libraries.

```
import numpy as np
import tensorflow as tf
from magi_msvgd.tf.magi import MAGISolver
```

Next, we define the ODE system and its gradients with respect to $\boldsymbol{X}$ and $\boldsymbol{\theta}$.

```
@tf.function
def ode(X, theta, t=None):
    n = Xs.shape[0]
    Vs = Xs[:,0]
    Rs = Xs[:,1]
    theta_rep = tf.tile(tf.expand_dims(thetas, axis=1), [1, n])
    a = theta_rep[0]
    b = theta_rep[1]
    c = theta_rep[2]

    return tf.stack([c * (Vs - Vs**3/3 + Rs),
                    -1/c * (Vs - a + b*Rs)], axis=1)

@tf.function
def dfdx(X, theta, t=None):
    n = Xs.shape[0]
    Vs = Xs[:,0]
    Rs = Xs[:,1]
    theta_rep = tf.tile(tf.expand_dims(thetas, axis=1), [1, n])
    a = theta_rep[0]
    b = theta_rep[1]
    c = theta_rep[2]

    return tf.stack([tf.stack([c * (1 - Vs**2), c], axis=1),
                    tf.stack([-1/c, -b/c], axis=1)], axis=2))

@tf.function
def dfdtheta(X, theta, t=None):
    n = Xs.shape[0]
    Vs = Xs[:,0]
    Rs = Xs[:,1]
    theta_rep = tf.tile(tf.expand_dims(thetas, axis=1), [1, n])
    a = theta_rep[0]
    b = theta_rep[1]
    c = theta_rep[2]
    zero = tf.zeros(a.shape, dtype=a.dtype)

    return tf.stack([tf.stack([zero, zero, Vs - Vs**3/3 + Rs], axis=1),
    ↪  tf.stack([1/c, -Rs/c, (Vs - a + b*Rs)/c**2], axis=1)], axis=2)
```

When using the TensorFlow front-end, we can no longer use the provided code to generate data, since `magi_msvgd.tests.make_test.ODEmodel` is built in PyTorch. Instead, we must generate a dataset manually. We can still use `magi_msvgd.tests.test_helpers` to discretize the data, but we will demonstrate manual discretization here.

```
from scipy.integrate import solve_ivp

# reformat ode for numerical solver
theta_true = np.array([0.2, 0.2, 3.0])
sigma_true = np.array([0.2. 0.2])
reformat_ode = lambda t, x: ode(x.reshape(1,-1), np.array([0.2, 0.2, 3.0]),
↪  t)

# times to numerically integrate, round to avoid float imprecision
times = np.round(np.linspace(0, T, 200001), 4)

ode_solution = solve_ivp(fun=reformat_ode, t_span=(times[0], times[-1]),
                        y0=np.array([-1., 1.]), t_eval=times).y.T
solution = np.concatenate([times.reshape(-1, 1), ode_solution], axis=1)

# define observation times (0, 0.5, 1, ..., 20)
tau = [np.linspace(0, 20, 41), np.linspace(0, 20, 41)]

# define observation times matrix (fully observed, 1 is truthy)
obs_t = np.unique(np.concatenate(tau))
obs_t = np.round(obs_t, 3)
```

```python
# create an observed time mask
obs_times = np.zeros([obs_t.shape[0], len(tau)+1])
obs_times[:,0] = obs_t
for d, tau_d in enumerate(tau):
    obs_times[np.where(np.isin(obs_times[:,0],
                np.round(tau_d, 3))),d+1] = 1.0

obs_mask = obs_times[:,1:].copy()
obs_ind = obs_mask.astype(bool)
obs_mask[~obs_ind] = np.nan

# extract ground truth at times tau, sample data according to sigma_true
ground_truth = solution[np.where(np.isin(solution[:,0], obs_t))]
sample = ground_truth.copy()
sample[:,1:] = np.random.normal(ground_truth[:,1:] * obs_mask, sigma_true)


# define discretization set
I = np.round(np.linspace(0, 20, 161), 3)

data_disc = np.full(shape=(len(I), sample.shape[1]), fill_value=np.nan)
data_disc[:,0] = I
data_disc[np.isin(data_disc[:,0], sample[:,0])] = sample
```

From here, defining the `MAGISolver` object is the same process as with the PyTorch front-end. The major difference is the `pos_X` and `pos_theta` settings do not work in the TensorFlow version due to the fact that TensorFlow tensors are immutable and cannot be updated in-place to clip negative values. Setting `pos_X` or `pos_theta` to `True` will cause initial particles to be sampled from $|q_0|$, but will otherwise have no effect on the mSVGD algorithm.

```python
magisolver = MAGISolver(
    ode=ode,
    dfdx=dfdx,
    dfdtheta=dfdtheta,
    data=data_disc,
    theta_guess=np.array([1.,1.,1.]),
    theta_conf=np.array([0.,0.,0.]),
    sigmas=np.array([0.2,0.2]),
    mu=None, mu_dot=None,
    pos_X=False, pos_theta=False, # these settings do nothing
    temper_prior=True
    bayesian_sigma=True
)
```

Finally, we initialize the particles and run mSVGD. Note that TensorFlow automatically manages device usage, so we do not specify a device when running `initialize_particles`.

```
magisolver.initialize_particles(k_0=200, dtype=tf.float32, device=None,
 ↪  init_sd=0.01, random_seed=2025)

# define optimizer object and parameters
optimizer = tf.keras.optimizers.Adam
optimizer_kwargs = {'learning_rate':1e-1}

# run solver
X_result, theta_result, sigma_result, trajectories =
 ↪  magisolver.solve(optimizer=optimizer, optimizer_kwargs=optimizer_kwargs,
                 max_iter=300, mitosis_splits=3, atol=0.1, rtol=0,
                  ↪  monitor_convergence=20)
```

C
mSVGD Configuration Settings for Test Cases

C.1 FN Model

The code for `ode`, `dfdx`, and `dfdtheta` can be found in the package in `magi_msvgd.tests.models.fitzhugh_nagumo`, along with the corresponding source code in the GitHub repository [6].

```
magisolver = MAGISolver(
    ode=ode,
    dfdx=dfdx,
    dfdtheta=dfdtheta,
    data=data_disc,
    theta_guess=np.array([1.,1.,1.]),
    theta_conf=np.array([0.,0.,0.]),
    sigmas=np.array([0.2,0.2]),
    X_guess=1,
    mu=None, mu_dot=None,
    pos_X=False, pos_theta=False,
    prior_temperature=None
    bayesian_sigma=True
)
```

```
magisolver.initialize_particles(k_0=200, dtype=torch.float32, device='cuda',
↪   init_sd=0.01, random_seed=None)

optimizer = torch.optim.Adam
optimizer_kwargs = {'lr':0.1}
```

```
X_result, theta_result, sigma_result = magisolver.solve(optimizer=optimizer,
↪  optimizer_kwargs=optimizer_kwargs,
                max_iter=max_iter, mitosis_splits=M, atol=0.1, rtol=0,
                ↪  monitor_convergence=False)
```

Test setting	M	max_iter
$\|\boldsymbol{I}\| = 41$	3	200
$\|\boldsymbol{I}\| = 81$	3	200
$\|\boldsymbol{I}\| = 161$	3	300
$\|\boldsymbol{I}\| = 321$	2	500
$\|\tau_d\| = 21$	2	500

Remark. The algorithm stops early in later splits before reaching `max_iter`.

C.2 Hes1 Model

The code for `ode`, `dfdx`, and `dfdtheta` can be found in the package in `magi_msvgd.tests.models.hes1`, along with the corresponding source code in the GitHub repository [6].

```
magisolver = MAGISolver(
    ode=ode,
    dfdx=dfdx,
    dfdtheta=dfdtheta,
    data=data_disc,
    theta_guess=np.array([0.,0.,0.,0.,1.,20.,0.]),
    theta_conf=0,
    sigmas=np.array([0.15,0.15,None]),
    X_guess=3,
    mu=None, mu_dot=None,
    pos_X=False, pos_theta=True,
    prior_temperature=None
    bayesian_sigma=True
)
```

```
magisolver.initialize_particles(k_0=200, dtype=torch.float32, device='cuda',
↪   init_sd=0.01, random_seed=None)

optimizer = torch.optim.Adam
optimizer_kwargs = {'lr':0.001}

X_result, theta_result, sigma_result = magisolver.solve(optimizer=optimizer,
↪   optimizer_kwargs=optimizer_kwargs,
                max_iter=1200, mitosis_splits=2, atol=0, rtol=0,
                ↪   monitor_convergence=False)
```

C.3 Lorenz Model

The code for `ode`, `dfdx`, and `dfdtheta` can be found in the package in `magi_msvgd.tests.models.lorenz`, along with the corresponding source code in the GitHub repository [6].

```
magisolver = MAGISolver(
    ode=ode,
    dfdx=dfdx,
    dfdtheta=dfdtheta,
    data=data_disc,
    theta_guess=np.array([2., 25., 10.]),
    theta_conf=0,
    sigmas=None,
    X_guess=1,
    mu=None, mu_dot=None,
    pos_X=False, pos_theta=False,
    prior_temperature=None
    bayesian_sigma=True
)
```

```
magisolver.initialize_particles(k_0=k_0, dtype=torch.float32, device='cuda',
↪   init_sd=1, random_seed=None)

optimizer = torch.optim.Adam
optimizer_kwargs = {'lr':1}

X_result, theta_result, sigma_result = magisolver.solve(optimizer=optimizer,
↪   optimizer_kwargs=optimizer_kwargs,
                max_iter=max_iter, mitosis_splits=M, atol=atol, rtol=0,
                ↪   monitor_convergence=False)
```

Test setting	`k_0`	`M`	`atol`	`max_iter`
mSVGD	400	3	0.005	1000
SVGD 1	3200	0	N/A	400
SVGD 2	400	0	N/A	2000
SVGD 3	3200	0	0.005	`until convergence`
SVGD 4	400	0	0.005	`until convergence`
SVGD 5	3200	0	0.05	`until convergence`

Remark. In the mSVGD case, the algorithm stops early in later splits before reaching `max_iter`.

References

[1] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, April 2017.

[2] Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. *Handbook of Markov Chain Monte Carlo*. Chapman and Hall/CRC, May 2011.

[3] Peng Chen and Omar Ghattas. Projected stein variational gradient descent, 2020.

[4] DartML. Stein variational gradient descent. `https://github.com/dartml/stein-variational-gradient-descent`, 2016.

[5] Hua Liang, Hongyu Miao, and Hulin Wu. Estimation of constant and time-varying dynamic parameters of hiv infection in a nonlinear differential equation model. *The Annals of Applied Statistics*, 4(1), Mar 2010.

[6] Jamie Liu. Mitotic stein variational gradient descent for manifold-constrained gaussian process inference. `https://github.com/jamieliu2/magi_msvgd`, 2025.

[7] Qiang Liu, Jason D. Lee, and Michael I. Jordan. A kernelized stein discrepancy for goodness-of-fit tests and model evaluation, 2016.

[8] Qiang Liu and Dilin Wang. Stein variational gradient descent: A general purpose bayesian inference algorithm, 2016.

[9] Richard McElreath. *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*. Chapman and Hall/CRC, 2020.

[10] uqfoundation. dill. `https://github.com/uqfoundation/dill`, 2025.

[11] Samuel W. K. Wong. Manifold-constrained gaussian process inference (magi). `https://github.com/wongswk/magi`, 2024.

[12] Samuel W. K. Wong, Shihao Yang, and S.C. Kou. magi: A package for inference of dynamic systems from noisy and sparse data via manifold-constrained Gaussian processes. *Journal of Statistical Software*, 109(4):1–47, 2024.

[13] Shihao Yang, Samuel W. K. Wong, and S. C. Kou. Inference of dynamic systems from noisy and sparse data via manifold-constrained gaussian processes. *Proceedings of the National Academy of Sciences*, 118(15):e2020397118, 2021.