



MAGI: A Package for Inference of Dynamic Systems from Noisy and Sparse Data via Manifold-constrained Gaussian Processes

Samuel W.K. Wong

University of Waterloo

Shihao Yang

Georgia Institute of Technology

S.C. Kou

Harvard University

Abstract

This article presents the **MAGI** software package for the inference of dynamic systems. The focus of **MAGI** is on dynamics modeled by nonlinear ordinary differential equations with unknown parameters. While such models are widely used in science and engineering, the available experimental data for parameter estimation may be noisy and sparse. Furthermore, some system components may be entirely unobserved. **MAGI** solves this inference problem with the help of manifold-constrained Gaussian processes within a Bayesian statistical framework, whereas unobserved components have posed a significant challenge for existing software. We use several realistic examples to illustrate the functionality of **MAGI**. The user may choose to use the package in any of the R, MATLAB, and Python environments.

Keywords: dynamic systems, ordinary differential equations, Bayesian inference, Gaussian processes, unobserved components.

1. Introduction

Ordinary differential equations (ODEs) are widely used as models for dynamic systems in science and engineering, including gene regulation (Bolouri 2008), chemical reactions (Walas 1991; Wong, Yang, and Kou 2023), epidemiology and ecology (Busenberg 2012), economics (Tu 2012), etc. We focus here on the case where the ODEs are nonlinear with unknown parameters governing their behavior. The problem of interest is to recover the unobserved system trajectories as well as to estimate the parameters from experimental or observational data, where the observations taken from the system may be subject to measurement noise and may only be available at a sparse number of time points. Further, some components in the system may be entirely unobserved. This paper introduces the **MAGI** software package,

named after its corresponding method (MANifold-constrained Gaussian process Inference; Yang, Wong, and Kou 2021) which provided fast and accurate inference for this statistical problem on a variety of examples, including the case when there are unobserved system components.

Specifically, **MAGI** handles parameter estimation for models where the system components are governed by a set of ODEs, which we denote by

$$\dot{\mathbf{x}}(t) = \frac{d\mathbf{x}(t)}{dt} = \mathbf{f}(\mathbf{x}(t), \boldsymbol{\theta}, t), \quad (1)$$

where $\mathbf{x}(t)$ is the D -dimensional system trajectory over time $0 \leq t \leq T$ (i.e., $\mathbf{x} : [0, T] \rightarrow \mathbb{R}^D$), and $\dot{\mathbf{x}}(t)$ is shorthand for the vector of derivatives $d\mathbf{x}(t)/dt$, which are specified via the known function $\mathbf{f}$. The vector $\boldsymbol{\theta}$ denotes the model parameters to be estimated, which govern the behavior of the system. We let $\mathbf{y}(\boldsymbol{\tau})$ denote the observed data, namely the noisy measurements taken from the system at observation time points $\boldsymbol{\tau}$. Throughout this article, we use $\boldsymbol{\tau} = (\tau_1, \tau_2, \dots, \tau_D)$ to denote the collection of observation time points, where τ_d is the vector of time points at which component d is observed, $d = 1, \dots, D$. Each system component can have its own set of observation times τ_d , and some components may not be observed at all (for which $\tau_d = \emptyset$). We assume that the noise is additive and Gaussian, i.e., $\mathbf{y}(\boldsymbol{\tau}) = \mathbf{x}(\boldsymbol{\tau}) + \boldsymbol{\epsilon}(\boldsymbol{\tau})$, where the error term $\boldsymbol{\epsilon}$ has noise level $\boldsymbol{\sigma}$ (which may be known or unknown). The key feature of **MAGI** is to infer $\mathbf{x}(t)$ and $\boldsymbol{\theta}$ from $\mathbf{y}(\boldsymbol{\tau})$ without the need for any numerical integration, even when there are unobserved system components.¹ This is achieved by taking a Gaussian process (GP) as a prior for $\mathbf{x}(t)$ and constraining it to a manifold that satisfies the ODE system. Inference is then carried out within a principled Bayesian statistical framework, that is, we condition on all known information and quantities and apply Bayesian techniques to the resulting posterior distribution.

MAGI is available for R, MATLAB, and Python which enable practitioners to input and work with custom ODE systems in their preferred computing environment. (The packages share a common C++ codebase, which ensures a consistent method implementation across all three environments.) The main text of this article will provide code examples in R (R Core Team 2022). Equivalent code for the examples in MATLAB and Python are provided in the replication materials, along with usage instructions in the Appendices A and B.

1.1. Illustrative example: oscillation of Hes1 mRNA and protein levels

To begin with a concrete example, consider the three-component dynamic system, which for short we write as $X = (P, M, H)$, introduced in Hirata, Yoshiura, Ohtsuka, Bessho, Harada *et al.* (2002), governed by the ODEs

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

where P and M are the protein and messenger ribonucleic acid (mRNA) levels in cultured cells. In experimental data, P and M levels exhibit oscillatory cycles approximately every 2 hours, and the H component is a Hes1-interacting factor that helps regulate this oscillation

¹In practice, **MAGI** infers $\mathbf{x}(t)$ for all $t \in \mathbf{I}$, where $\mathbf{I}$ is any finite set of discretization points in $[0, T]$ specified by the user, as subsequently demonstrated.

via a negative feedback loop. The parameters of this system are $\theta = (a, b, c, d, e, f, g)$, where a and b can be interpreted as synthesis rates; c, d and g as decomposition rates; and e and f as inhibition rates.

The remainder of this subsection describes a realistic sample dataset that is simulated from this system. The key features of the dataset are: (i) M and P are measured at different sets of time points, (ii) the observations for M and P are noisy (i.e., have measurement error), (iii) H is never observed. In Section 3, we will then demonstrate how to use **MAGI** to recover the system trajectories and parameters from the dataset, without the use of any numerical solvers.

We first define a function that computes $\mathbf{f}$ for this ODE system. Its inputs are a vector of parameters θ , a matrix for X (with columns corresponding to components), and a vector of time points `tvec`. The function then returns a matrix of values of $\mathbf{f}$, with rows corresponding to the time points in `tvec` and columns corresponding to the components of X :

```
R> hes1modelODE <- function(theta, x, tvec) {
+   P = x[, 1]
+   M = x[, 2]
+   H = x[, 3]
+
+   PMHdt = array(0, c(nrow(x), ncol(x)))
+   PMHdt[, 1] = -theta[1] * P * H + theta[2] * M - theta[3] * P
+   PMHdt[, 2] = -theta[4] * M + theta[5] / (1 + P^2)
+   PMHdt[, 3] = -theta[1] * P * H + theta[6] / (1 + P^2) - theta[7] * H
+
+   PMHdt
+ }
```

Following the real experimental setup in [Hirata *et al.* \(2002\)](#), measurements of P and M are taken every 15 minutes over a four-hour period, but asynchronously: P is observed at $t = 0, 15, 30, \dots, 240$ minutes, while M is observed at $t = 7.5, 22.5, \dots, 232.5$ minutes, and H is never observed.

We shall simulate from this system using the parameter values studied theoretically in [Hirata *et al.* \(2002\)](#): $a = 0.022$, $b = 0.3$, $c = 0.031$, $d = 0.028$, $e = 0.5$, $f = 20$, $g = 0.3$; the initial conditions $P(0) = 1.439$, $M(0) = 2.037$, $H(0) = 17.904$ are taken to mimic the authors' setting, where the system is initialized at the point in the stable oscillation cycle where P is at its minimum. The observation noise in the experiment is approximately 15% of both the P and M levels, which we treat as multiplicative noise following a log-normal distribution with known standard deviation 0.15. For convenience, we setup a list containing these input values for the simulation:

```
R> param.true <- list(
+   theta = c(0.022, 0.3, 0.031, 0.028, 0.5, 20, 0.3),
+   x0 = c(1.439, 2.037, 17.904),
+   sigma = c(0.15, 0.15, NA))
```

Since H is never observed, measurement noise is not applicable to that component. Next, to simulate the data for our analysis, we use a numerical solver to construct the system

trajectories implied by these values. In R, we can utilize the ODE solvers available in the **deSolve** package (Soetaert, Petzoldt, Setzer, Brown, Byrne *et al.* 2020), by defining a wrapper that satisfies the syntax of its `ode` function. We emphasize that the numerical ODE solver is only used here for generating the data; throughout the **MAGI** method for inferring the system trajectories and parameters, no numerical solver is ever needed.

```
R> modelODE <- function(tvec, state, parameters) {
+   list(as.vector(hes1modelODE(parameters, t(state), tvec)))
+ }
```

We may now numerically solve the ODE trajectories $\mathbf{x}$ over the time period of interest, from $t = 0$ to 4 hours (specified as 240 minutes):

```
R> x <- deSolve::ode(y = param.true$x0, times = seq(0, 60 * 4, by = 0.01),
+                  func = modelODE, parms = param.true$theta)
```

Next, we extract the true values of the trajectory at the time points according to the schedule of observations described, and simulate noisy measurements $\mathbf{y}$ for P and M . The seed 12321 is set for reproducibility.

```
R> set.seed(12321)
R> y <- as.data.frame(x[ x[, "time"] %in% seq(0, 240, by = 7.5), ])
R> names(y) <- c("time", "P", "M", "H")
R> y$P <- y$P * exp(rnorm(nrow(y), sd = param.true$sigma[1]))
R> y$M <- y$M * exp(rnorm(nrow(y), sd = param.true$sigma[2]))
```

For system components that are unobserved at a time point, we fill in the corresponding values with `NaN`, recalling that P and M are observed asynchronously and H is never observed:

```
R> y$H <- NaN
R> y$P[y$time %in% seq(7.5, 240, by = 15)] <- NaN
R> y$M[y$time %in% seq(0, 240, by = 15)] <- NaN
```

Now the dataset $\mathbf{y}$ is prepared. Based on this dataset, **MAGI** will infer the underlying trajectories $\mathbf{X}$ and estimate the seven parameters in $\boldsymbol{\theta}$. We plot the observed data in Figure 1, with the points showing the noisy measurements available for P and M and the solid curves showing the true system trajectories of $\mathbf{X}$. The following commands create this plot:

```
R> compnames <- c("P", "M", "H")
R> matplot(x[, "time"], x[, -1], type = "l", lty = 1,
+         xlab = "Time (min)", ylab = "Level")
R> matplot(y$time, y[, -1], type = "p", col = 1:(ncol(y)-1), pch = 20, add = TRUE)
R> legend("topright", compnames, lty = 1, col = c("black", "red", "green"))
```

1.2. Overview of related software

MAGI handles the so-called *inverse problem* for ODEs, namely to recover the system trajectories and parameters from a set of observational or experimental data. Existing software for

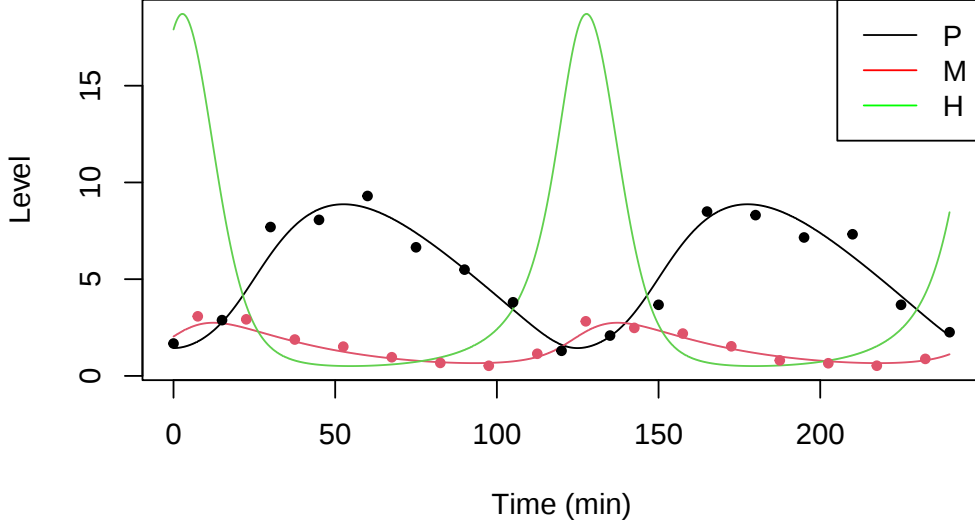


Figure 1: True system trajectories (solid curves) and sample noisy observations (points) from the Hes1 system. The H component is never observed. In Section 3, we demonstrate how **MAGI** recovers the system trajectories and parameters from this sample dataset of noisy observations.

this problem can be broadly categorized into methods that rely on using numerical solvers for ODEs and those that do not.

In Equation 1, the system trajectory $\mathbf{x}(t)$ may be determined for a given set of parameter values $\boldsymbol{\theta}$ and initial conditions $\mathbf{x}(0)$ by integration. For nonlinear functions $\mathbf{f}$, numerical integrators (e.g., Runge-Kutta) are often needed for solving the ODEs in this way. We may denote this numerical solution as $\hat{\mathbf{x}}(t; \mathbf{x}(0), \boldsymbol{\theta})$ to indicate its deterministic relationship with $\boldsymbol{\theta}$ and $\mathbf{x}(0)$. A simple approach can thus repeatedly solve for $\hat{\mathbf{x}}(t; \mathbf{x}(0), \boldsymbol{\theta})$ to optimize a likelihood or least-squares criterion for the data $\mathbf{y}(\boldsymbol{\tau})$, as a function of $\boldsymbol{\theta}$ (and also $\mathbf{x}(0)$ if the initial conditions are unknown). A least-squares criterion to minimize would take the form $\|\mathbf{y}(\boldsymbol{\tau}) - \hat{\mathbf{x}}(\boldsymbol{\tau}; \mathbf{x}(0), \boldsymbol{\theta})\|^2$ where $\|\cdot\|$ is the usual Euclidean norm, i.e., by evaluating $\hat{\mathbf{x}}$ (which is determined for all t) at the observation time points $\boldsymbol{\tau}$. This can be viewed as a non-linear least squares (NLS) problem, which could be handled in R using `nls` from **stats** (R Core Team 2022) together with one of the numerical integrators in **deSolve**. In MATLAB, the **System Identification** toolbox (Ljung 1995) and the **Data2Dynamics** modeling environment (Raue, Steiert, Schelker, Kreutz, Maiwald, Hass, Vanlier, Tönsing, Adlung, Engesser *et al.* 2015) also provide functionality for inverse problems with the help of numerical solvers. Bayesian approaches for parameter estimation using numerical solvers are also available, such as in the **deBInfer** package (Boersch-Supan, Ryan, and Johnson 2017) in R. In this case, the numerical solution is used to construct a likelihood $p(\mathbf{y}(\boldsymbol{\tau}) | \hat{\mathbf{x}}(\boldsymbol{\tau}; \boldsymbol{\theta}, \mathbf{x}(0)), \boldsymbol{\sigma})$ and priors are placed on all the unknown quantities among $\boldsymbol{\theta}$, $\mathbf{x}(0)$, and $\boldsymbol{\sigma}$. While methods based on numerical solvers are generally applicable for ODE inverse problems (including when system components are

unobserved), they may encounter significant computational bottlenecks: the numerical solver must be invoked repeatedly for values of θ and $\mathbf{x}(0)$ used in the estimation procedure.

As a result, the second broad category consists of methods designed to estimate θ without the need for numerical integration; **MAGI** belongs to this category. These methods use various techniques to curve-fit the observations while following the ODE system dynamics (e.g., by gradient matching). We overview some representative methods with software available in R in the following:

- A pioneering collocation approach proposed a B-spline basis to fit the system trajectories, where $\mathbf{x}(t)$ is represented by $\hat{\mathbf{x}}(t) = \mathbf{c}^\top \Phi(t)$ for basis functions $\Phi(t)$ and coefficients $\mathbf{c}$. Then, a penalized likelihood of the form $\|\mathbf{y}(\tau) - \hat{\mathbf{x}}(\tau)\|^2 + \lambda \int [\dot{\hat{\mathbf{x}}}(t) - \mathbf{f}(\hat{\mathbf{x}}(t), \theta, t)]^2 dt$ is optimized, where the first term measures fit to the data and the second term (with penalty parameter λ and using B-spline derivatives for $\dot{\hat{\mathbf{x}}}(t)$) ensures fidelity to the ODEs (Ramsay, Hooker, Campbell, and Cao 2007). This method is available in the packages **CollocInfer** (Hooker, Ramsay, and Xiao 2016) and **pCODE** (Wang and Cao 2019).
- Reproducing kernel Hilbert spaces (RKHS) have also been used for gradient matching (Niu, Rogers, Filippone, and Husmeier 2016). The d -th component of $\mathbf{x}(t)$ is represented by $\hat{x}_d(t) = \mathbf{b}_d^\top \mathbf{k}_d(t)$ with $\mathbf{k}_d(t) = [k(t, t_1), \dots, k(t, t_n)]$, where $k(t, \cdot)$ is a kernel function from the Hilbert space, $t_1, \dots, t_n$ are the observation times, and $\mathbf{b}_d$ are kernel coefficients. A criterion of the form $\|\mathbf{y}(\tau) - \hat{\mathbf{x}}(\tau)\|^2 + \lambda \|\dot{\hat{\mathbf{x}}}(\tau) - \mathbf{f}(\hat{\mathbf{x}}(\tau), \theta, \tau)\|^2$ is then minimized, where the two terms have similar interpretation as in **CollocInfer**, and the regularization parameter λ may be obtained by cross-validation. Different variants of RKHS methods, including transformations on t to better accommodate potential time inhomogeneity of the ODE solutions, are implemented in **KGode** (Niu, Wandy, Daly, Rogers, and Husmeier 2021).
- Inference based on a separable integral-matching approach is implemented in **simode** (Dattner and Yaari 2020). First, $\mathbf{x}(t)$ is approximated via fitting a spline representation $\hat{\mathbf{x}}(t)$ to the data, to bypass numerical integration of the ODEs. Noting that the true ODE solution may be expressed as $\mathbf{x}(t) = \mathbf{x}(0) + \int_0^t \mathbf{f}(\mathbf{x}(s), \theta, s) ds$, integral matching then seeks to minimize the criterion $\int_0^T \left\| \hat{\mathbf{x}}(t) - \mathbf{x}(0) - \int_0^t \mathbf{f}(\hat{\mathbf{x}}(s), \theta, s) ds \right\|^2 dt$ as a function of θ and $\mathbf{x}(0)$. In systems where $\mathbf{f}$ is linear or semi-linear in θ , the estimates can be obtained efficiently by taking advantage of the separable parameters in the optimization procedure (otherwise, non-linear optimization will be required).
- A Bayesian approach using GPs for fitting the trajectories is available in the **deGradInfer** package (Macdonald and Dondelinger 2020). This implements the gradient matching method of Dondelinger, Husmeier, Rogers, and Filippone (2013), which places a GP prior on $\mathbf{x}(t)$ (with hyper-parameters ϕ) so that $\mathbf{y}, \mathbf{x}, \dot{\mathbf{x}}$ have a joint GP specification. The joint distribution over all the quantities, namely $p(\mathbf{y}, \mathbf{x}, \dot{\mathbf{x}}, \theta, \phi, \sigma)$, is then factorized as $p(\mathbf{y}, \mathbf{x}, \dot{\mathbf{x}}, \theta, \phi, \sigma) = p(\mathbf{y}|\mathbf{x}, \sigma)p(\dot{\mathbf{x}}|\mathbf{x}, \theta, \phi)p(\mathbf{x}|\phi)p(\phi)p(\theta)p(\sigma)$, where $p(\mathbf{y}|\mathbf{x}, \sigma)$ denotes the likelihood of the observations, $p(\mathbf{x}|\phi)$ is the GP prior on $\mathbf{x}(t)$, and priors are assigned to θ, ϕ, σ . To carry out inference in this method, the term $p(\dot{\mathbf{x}}|\mathbf{x}, \theta, \phi)$ is expressed as a heuristic product that combines the contributions of the GP and ODEs, namely $p(\dot{\mathbf{x}}|\mathbf{x}, \theta, \phi) \propto p(\dot{\mathbf{x}}|\mathbf{x}, \phi)p(\dot{\mathbf{x}}|\mathbf{x}, \theta)$, where $p(\dot{\mathbf{x}}|\mathbf{x}, \phi)$ comes from the GP and $p(\dot{\mathbf{x}}|\mathbf{x}, \theta)$ comes from the specification of $\mathbf{f}$ in the ODE (see Equation 1).

While some of these methods can handle unobserved system components in theory, the available software implementations tend to lack this functionality in general. Only **CollocInfer** and **pCODE** can accommodate an unobserved component in the estimation procedure; however, substantial manual input is required to carry out the analysis. (This is subsequently demonstrated in Section 5, where we carry out an illustrative comparison between methods.) Thus, one distinct contribution of the **MAGI** package is that it provides a ready-made solution for systems with unobserved components, in addition to its principled inference framework that is rooted in Bayesian statistics.

2. Manifold-constrained Gaussian process inference

This section explains the key points of the **MAGI** method for inferring the system trajectories $\mathbf{x}(t)$ and parameters $\boldsymbol{\theta}$ given the observed data $\mathbf{y}(\boldsymbol{\tau})$. The interested reader may refer to Yang *et al.* (2021) for additional details.

The **MAGI** method places a GP prior on $\mathbf{x}(t)$, so that $\dot{\mathbf{x}}(t)$ conditional on $\mathbf{x}(t)$ also has a convenient GP form for facilitating inference without the need for numerical integration. Previous authors adopting this basic idea, e.g., Calderhead, Girolami, and Lawrence (2009); Dondelinger *et al.* (2013); Barber and Wang (2014); Wenk, Gotovos, Bauer, Gorbach, Krause, and Buhmann (2019), have noted that this setup can cause $\dot{\mathbf{x}}(t)$ to be conceptually specified in two incompatible ways: first via the function $\mathbf{f}$ in the ODE (see Equation 1), and second via the GP, e.g., as seen above in the setup of **deGradInfer**. The **MAGI** method addressed this conceptual difficulty by conditioning the GP on a manifold constraint that satisfies the ODEs specified by $\mathbf{f}$.

This manifold constraint can be described as follows. First, let D denote the number of system components, with $x_d(t)$ and $\mathbf{f}(\mathbf{x}(t), \boldsymbol{\theta}, t)_d$ denoting the d -th component of $\mathbf{x}(t)$ and $\mathbf{f}$ respectively, $d = 1, \dots, D$, and let $C^1[0, T]$ be the set of differentiable functions on $[0, T]$. Then for $\mathbf{x} : [0, T] \rightarrow \mathbb{R}^D$ and a given parameter space $\Omega_{\boldsymbol{\theta}}$, we define a manifold $\mathcal{X}$ on which the derivative $\dot{\mathbf{x}}$ satisfies the dynamics specified by the ODE:

$$\begin{aligned} \mathcal{X} = \{ \mathbf{x} = (x_1, \dots, x_D) \mid x_d \in C^1[0, T], \text{ for all } d = 1, \dots, D, \\ \dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \boldsymbol{\theta}, t) \text{ for all } t \in [0, T] \text{ and some } \boldsymbol{\theta} \in \Omega_{\boldsymbol{\theta}} \}, \end{aligned}$$

i.e., $\mathbf{x} \in \mathcal{X}$ lies on the manifold of the ODE solutions. Second, to incorporate this manifold as a constraint in the GP, we define a variable W according to

$$W = \sup_{t \in [0, T], d \in \{1, \dots, D\}} |\dot{X}_d(t) - \mathbf{f}(\mathbf{X}(t), \boldsymbol{\theta}, t)_d|,$$

i.e., W quantifies the maximum discrepancy between a derivative trajectory and the dynamics implied by the ODEs. Thus, $W = 0$ if and only if a realization $\mathbf{X} = \mathbf{x}$ of the GP satisfies $\mathbf{x} \in \mathcal{X}$. Under a Bayesian paradigm, the joint posterior distribution of $\boldsymbol{\theta}$ and $\mathbf{X}(t)$ is then conditioned on $W = 0$ and the observed data $\mathbf{y}(\boldsymbol{\tau})$, i.e., the ideal posterior of interest is $p(\boldsymbol{\theta}, \mathbf{x}(t) \mid W = 0, \mathbf{y}(\boldsymbol{\tau}))$. However to be practically computable, $W = 0$ needs to be approximated by finite discretization. Let $\mathbf{I} = \{t_1, t_2, \dots, t_n\}$ be a set of discretization points in $[0, T]$, with $\boldsymbol{\tau} \subset \mathbf{I}$; then as a discretized analogue to W , we define a variable $W_{\mathbf{I}}$ according to

$$W_{\mathbf{I}} = \max_{t \in \mathbf{I}, d \in \{1, \dots, D\}} |\dot{X}_d(t) - \mathbf{f}(\mathbf{X}(t), \boldsymbol{\theta}, t)_d|.$$

This allows us to approximate $W = 0$ by setting $W_I = 0$, i.e., $\dot{\mathbf{X}}(t)$ from the GP is constrained to equal $\mathbf{f}(\mathbf{X}(t), \boldsymbol{\theta}, t)$ from the ODE for each component $d = 1, \dots, D$ at each time point $t \in \mathbf{I}$. With $W_I = 0$ as the manifold constraint in practice, the corresponding posterior distribution is $p(\boldsymbol{\theta}, \mathbf{x}(\mathbf{I}) | W_I = 0, \mathbf{y}(\boldsymbol{\tau}))$. As we shall see below, this construction induces a closed-form posterior such that standard techniques of Bayesian inference can be applied, while ensuring that posterior samples of $\mathbf{x}(\mathbf{I})$ respect the ODE dynamics. Our construction also contrasts with penalized likelihood or regularization-based approaches (e.g., Ramsay *et al.* 2007; Wang and Cao 2019; Niu *et al.* 2021); applying the rules of conditional probability with $W_I = 0$ ensures that the ODE dynamics are exactly followed for all time points in $\mathbf{I}$, without the need to construct any penalty or regularization term.

To illustrate this concept of manifold constraint, we guide the reader through a simple example in Figure 2. We begin with a selected GP prior on a 1-dimensional $x(t)$ over the interval $[0, 10]$. (For practical computation, the GP will be evaluated at $\mathbf{I} = \{0, 0.05, 0.1, \dots, 10\}$.) Suppose four noisy observations $\mathbf{y}(\boldsymbol{\tau})$ are taken at $\boldsymbol{\tau} = \{1, 3, 4, 5\}$; these are shown as black points in the panels of Figure 2. Then the GP posterior conditional on these four observations is visualized in Figure 2a. Five sample trajectories drawn from this GP posterior are shown via the colored curves, and the grey bands in Figure 2a represent 95% credible intervals of this GP posterior. Next, as an example suppose $x(t)$ satisfies the linear ODE $\dot{x}(t) = f(x(t), \boldsymbol{\theta}, t) = \theta_1 x(t) + \theta_2$ with parameters $\boldsymbol{\theta} = (\theta_1, \theta_2)$, which has analytic solution $x(t) = c \exp(\theta_1 t) - \theta_2 / \theta_1$ for some constant c . We now proceed to condition the GP on both $\mathbf{y}(\boldsymbol{\tau})$ and the ODE manifold constraint $W_I = 0$. Five draws of $\mathbf{x}(\mathbf{I})$ and $\boldsymbol{\theta}$ from this manifold-constrained GP posterior are visualized via the colored curves in Figure 2b. We see that they each obey the functional form of the known analytic solution $x(t)$ in this case, namely $x(t) = c \exp(\theta_1 t) - \theta_2 / \theta_1$ with different values of c , θ_1 , and θ_2 , i.e., $\dot{x}(t)$ from the GP satisfies the ODE specified by f . The grey shaded area in Figure 2b represents 95% credible intervals of this manifold-constrained GP posterior. Contrasting the two panels of Figure 2, this analytical example provides an intuitive depiction for the key idea of the ODE manifold constraint. The R code to produce Figure 2 is given in the replication script.

We return to our discussion of the joint posterior distribution on $\mathbf{x}(\mathbf{I})$ (i.e., the system trajectories at $t_1, \dots, t_n$) and $\boldsymbol{\theta}$, given $\mathbf{y}(\boldsymbol{\tau})$ and $W_I = 0$, which is expressed using Bayes' rule and factorized according to

$$\begin{aligned} & p(\boldsymbol{\theta}, \mathbf{x}(\mathbf{I}) | W_I = 0, \mathbf{y}(\boldsymbol{\tau})) \\ & \propto p(\boldsymbol{\Theta} = \boldsymbol{\theta}, \mathbf{X}(\mathbf{I}) = \mathbf{x}(\mathbf{I}), W_I = 0, \mathbf{Y}(\boldsymbol{\tau}) = \mathbf{y}(\boldsymbol{\tau})) \\ & = \pi(\boldsymbol{\theta}) \times p(\mathbf{X}(\mathbf{I}) = \mathbf{x}(\mathbf{I}) | \boldsymbol{\Theta} = \boldsymbol{\theta}) \times p(\mathbf{Y}(\boldsymbol{\tau}) = \mathbf{y}(\boldsymbol{\tau}) | \mathbf{X}(\mathbf{I}) = \mathbf{x}(\mathbf{I}), \boldsymbol{\Theta} = \boldsymbol{\theta}) \\ & \quad \times p(W_I = 0 | \mathbf{Y}(\boldsymbol{\tau}) = \mathbf{y}(\boldsymbol{\tau}), \mathbf{X}(\mathbf{I}) = \mathbf{x}(\mathbf{I}), \boldsymbol{\Theta} = \boldsymbol{\theta}). \end{aligned}$$

Next, we note that both the GP prior for $\mathbf{X}$ and the observations are independent of $\boldsymbol{\Theta}$, so we have the simplifications $p(\mathbf{X}(\mathbf{I}) = \mathbf{x}(\mathbf{I}) | \boldsymbol{\Theta} = \boldsymbol{\theta}) = p(\mathbf{X}(\mathbf{I}) = \mathbf{x}(\mathbf{I}))$ and $p(\mathbf{Y}(\boldsymbol{\tau}) = \mathbf{y}(\boldsymbol{\tau}) | \mathbf{X}(\mathbf{I}) = \mathbf{x}(\mathbf{I}), \boldsymbol{\Theta} = \boldsymbol{\theta}) = p(\mathbf{y}(\boldsymbol{\tau}) | \mathbf{x}(\mathbf{I}))$. To simplify the last term, we substitute the definition of $W_I = 0$ and use the fact that the GP derivative $\dot{\mathbf{X}}(\mathbf{I})$ given $\mathbf{X}(\mathbf{I})$ has a multivariate normal distribution that is conditionally independent of $\boldsymbol{\Theta}$ and the observations:

$$\begin{aligned} & p(W_I = 0 | \mathbf{Y}(\boldsymbol{\tau}) = \mathbf{y}(\boldsymbol{\tau}), \mathbf{X}(\mathbf{I}) = \mathbf{x}(\mathbf{I}), \boldsymbol{\Theta} = \boldsymbol{\theta}) \\ & = p(\dot{\mathbf{X}}(\mathbf{I}) = \mathbf{f}(\mathbf{x}(\mathbf{I}), \boldsymbol{\theta}, \mathbf{I}) | \mathbf{Y}(\boldsymbol{\tau}) = \mathbf{y}(\boldsymbol{\tau}), \mathbf{X}(\mathbf{I}) = \mathbf{x}(\mathbf{I}), \boldsymbol{\Theta} = \boldsymbol{\theta}) \\ & = p(\dot{\mathbf{X}}(\mathbf{I}) = \mathbf{f}(\mathbf{x}(\mathbf{I}), \boldsymbol{\theta}, \mathbf{I}) | \mathbf{x}(\mathbf{I})). \end{aligned}$$

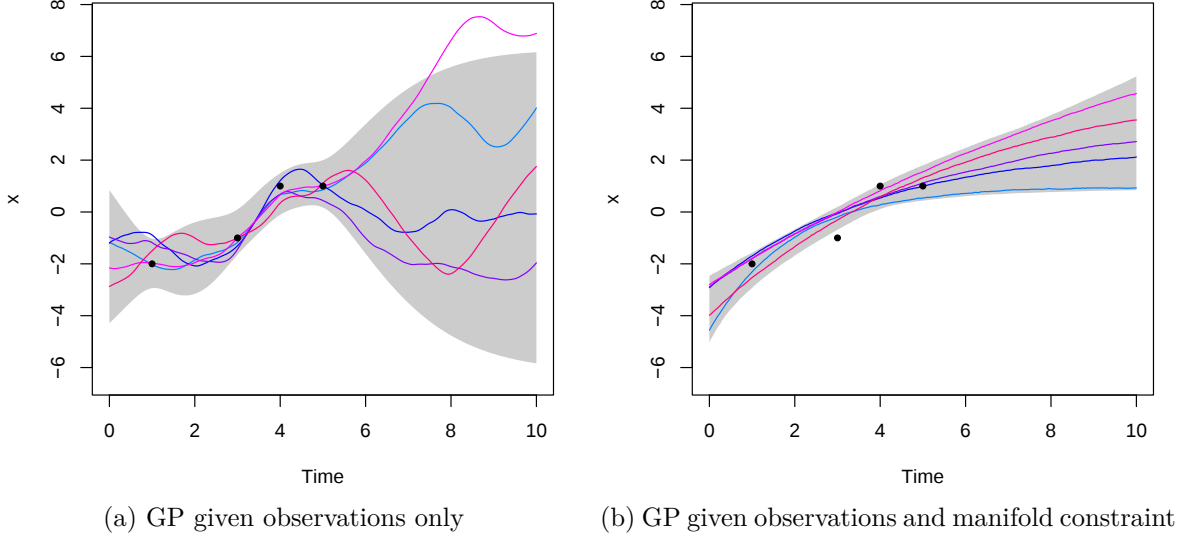


Figure 2: Example visualization of the ODE manifold constraint. In panel (a), the GP is conditioned on four noisy observations (solid black dots). The colored curves are five random trajectories drawn from the resulting GP posterior, and the grey shaded area represents the 95% credible interval. In panel (b), the GP is conditioned on the same four noisy observations and also the ODE manifold constraint. The colored curves are five random trajectories drawn from the resulting GP posterior, and the grey shaded area represents the 95% credible interval with the manifold constraint.

Thus, we finally obtain

$$\begin{aligned}
 & p(\boldsymbol{\theta}, \mathbf{x}(\mathbf{I}) | W_{\mathbf{I}} = 0, \mathbf{y}(\boldsymbol{\tau})) \\
 & \propto \pi(\boldsymbol{\theta}) \times p(\mathbf{x}(\mathbf{I})) \times p(\mathbf{y}(\boldsymbol{\tau}) | \mathbf{x}(\mathbf{I})) \times p(\dot{\mathbf{X}}(\mathbf{I}) = \mathbf{f}(\mathbf{x}(\mathbf{I}), \boldsymbol{\theta}, t_{\mathbf{I}}) | \mathbf{x}(\mathbf{I})), \quad (3)
 \end{aligned}$$

where the four terms are: $\pi(\boldsymbol{\theta})$ the prior density of the parameters, $p(\mathbf{x}(\mathbf{I}))$ the multivariate normal density for the GP prior on $\mathbf{x}(t)$ evaluated at the points in $\mathbf{I}$, $p(\mathbf{y}(\boldsymbol{\tau}) | \mathbf{x}(\mathbf{I}))$ the likelihood of the noisy observations, and $p(\dot{\mathbf{X}}(\mathbf{I}) = \mathbf{f}(\mathbf{x}(\mathbf{I}), \boldsymbol{\theta}, t_{\mathbf{I}}) | \mathbf{x}(\mathbf{I}))$ the multivariate normal density for the conditional distribution of $\dot{\mathbf{X}}(\mathbf{I})$ given $\mathbf{X}(\mathbf{I})$ evaluated at $\dot{\mathbf{X}}(\mathbf{I}) = \mathbf{f}(\mathbf{x}(\mathbf{I}), \boldsymbol{\theta}, t_{\mathbf{I}})$. Equation 3 is the basis of inference for **MAGI**.

Note that the GP prior on $\mathbf{x}(t)$ involves hyper-parameters that govern the mean and covariance functions of the GP; we denote these hyper-parameters by $\boldsymbol{\phi}$. The likelihood $p(\mathbf{y}(\boldsymbol{\tau}) | \mathbf{x}(\mathbf{I}))$ may additionally depend on noise parameters, we denote these by $\boldsymbol{\sigma}$, which may be known or unknown depending on the application.

The **MAGI** method obtains Markov chain Monte Carlo (MCMC) samples for $\mathbf{x}(\mathbf{I})$ and $\boldsymbol{\theta}$ from Equation 3. The method proceeds according to the following steps:

1. For system components that have observations, obtain values of $\boldsymbol{\phi}$ and $\boldsymbol{\sigma}$ by finding the optimal hyper-parameters and noise level based on fitting a GP to the data. (If the noise level $\boldsymbol{\sigma}$ is known or given, optimization is applied to $\boldsymbol{\phi}$ only.) The $\boldsymbol{\sigma}$ obtained is used to initialize MCMC sampling when the noise level is unknown. MCMC sampling for $\mathbf{x}(\mathbf{I})$ is initialized by linearly interpolating the observations $\mathbf{y}(\boldsymbol{\tau})$.

2. For system components that are unobserved, obtain values of ϕ and $\mathbf{x}(\mathbf{I})$ together with θ by a joint optimization of Equation 3, treating $(\phi, \sigma, \mathbf{x}(\mathbf{I}))$ for the observed components obtained in step 1 as fixed. If there are no unobserved components, only θ is optimized in Equation 3. This step provides values at which MCMC sampling for θ is initialized, along with $(\phi, \mathbf{x}(\mathbf{I}))$ for unobserved components.
3. Hamiltonian Monte Carlo (HMC, Neal 2011) is used as the MCMC sampling algorithm for obtaining joint draws of $\mathbf{x}(\mathbf{I})$ and θ (together with σ if the noise parameters are unknown). During this posterior sampling, the GP hyper-parameters ϕ are fixed at the values obtained in steps 1 and 2. Our implementation of HMC automatically tunes the leapfrog step sizes of HMC during burn-in to achieve an acceptance rate of 60–90%. (The theoretical optimal acceptance rate for HMC is 65%, as suggested in Neal (2011).)

At the conclusion of MCMC sampling (and after discarding the burn-in iterations), we may treat the posterior means of $\mathbf{x}(\mathbf{I})$ and θ as estimates of the trajectories and parameters respectively. The MCMC samples can also be used to characterize the uncertainty in these estimates, by computing credible intervals (CIs).

More methodological details of the **MAGI** method are discussed in Yang *et al.* (2021); the remainder of this paper focuses on practical usage of **MAGI** and the functionalities of the software package.

3. Using the MAGI package

This section illustrates the main functionalities of the **MAGI** package by analyzing the sample dataset for the Hes1 model discussed in the Introduction.

After installing the **MAGI** package (Yang and Wong 2023) from CRAN, we load it into R

```
R> library("magi")
```

The overall method is carried out via the core function `MagiSolver`, which initializes the GP hyper-parameters ϕ and then carries out MCMC sampling for the parameters together with the system trajectories. The basic syntax is

```
R> MagiSolver(y, odeModel, control = list())
```

where `y` is a data matrix that includes a column named `time` for the time points, `odeModel` is a list that specifies the ODE functions and its parameters, and `control` is a list used to provide any additional control settings. We describe each of these in turn, as we set up `MagiSolver` for the Hes1 dataset.

Since the Hes1 observations for P and M are positive and subject to multiplicative error, we may apply a log-transform to the data and equations for the analysis, which then satisfies the framework for additive noise ϵ . Thus, continuing the data setup from Section 1, we create `y.tilde` as the log-transformed version of the observations `y` (i.e., excluding the `time` column):

```
R> y.tilde <- y
R> y.tilde[, names(y.tilde) != "time"] <-
  log(y.tilde[, names(y.tilde) != "time"])
```

Recall that any unobserved values of `y.tilde` (in this case, the entire H component column and every other value for P and M) are assigned `NaN`. The data matrix `y.tilde` for input to `MagiSolver` is now prepared, where the discretization set is $\mathbf{I} = \{0, 7.5, 15, \dots, 240\}$ minutes and corresponds to the time points where either P or M are observed. Note that with $|\mathbf{I}|$ denoting the cardinality of $\mathbf{I}$, the dimensions of the input data matrix are $|\mathbf{I}| \times (D + 1)$, to include a column for time and each system component. To set up the input data matrix with different choices of $\mathbf{I}$, a helper function `setDiscretization` is provided; see Section 4.1 for a discussion of its usage and general guidelines for setting up $\mathbf{I}$ in practice.

Next we set up the functions required for the `odeModel` list. First, we apply a log-transform to each component of the ODE system by defining $\tilde{X} = (\log(P), \log(M), \log(H))$, then applying the relation $\frac{d(\log u)}{dt} = \frac{du}{dt} \cdot \frac{1}{u}$ we see that $\tilde{X}$ satisfies

$$\mathbf{f}(\tilde{X}, \boldsymbol{\theta}, t) = \begin{pmatrix} -aH + bM/P - c \\ -d + \frac{e}{(1+P^2)M} \\ -aP + \frac{f}{(1+P^2)H} - g \end{pmatrix}, \quad (4)$$

where P , M , and H are the components in Equation 2. We may code this via the function

```
R> hes1logmodelODE <- function (theta, x, tvec) {
+   P = exp(x[, 1])
+   M = exp(x[, 2])
+   H = exp(x[, 3])
+
+   PMHdt <- array(0, c(nrow(x), ncol(x)))
+   PMHdt[, 1] = -theta[1] * H + theta[2] * M / P - theta[3]
+   PMHdt[, 2] = -theta[4] + theta[5] / (1 + P^2) / M
+   PMHdt[, 3] = -theta[1] * P + theta[6] / (1 + P^2) / H - theta[7]
+
+   PMHdt
+ }
```

which follows the same format as the function for the original (non-transformed) `hes1modelODE`.

Second, to facilitate MCMC sampling via HMC we also supply functions for the gradients of the ODEs with respect to the system components `x` and the parameters `theta`. With respect to `x`, we have the matrix of gradients as follows,

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

which are specified via a function that outputs a 3-D array with dimensions $|\mathbf{I}| \times D \times D$, where the array slice `[, i, j]` is the partial derivative of the ODE for the j -th system component with respect to the i -th system component:

```
R> hes1logmodelDx <- function (theta, x, tvec) {
+   logP = x[, 1]
+   logM = x[, 2]
```

```

+   logH = x[, 3]
+
+   Dx <- array(0, c(nrow(x), ncol(x), ncol(x)))
+   dP = -(1 + exp(2 * logP))(-2) * exp(2 * logP) * 2
+   Dx[, 1, 1] = -theta[2] * exp(logM - logP)
+   Dx[, 2, 1] = theta[2] * exp(logM - logP)
+   Dx[, 3, 1] = -theta[1] * exp(logH)
+   Dx[, 1, 2] = theta[5] * exp(-logM) * dP
+   Dx[, 2, 2] = -theta[5] * exp(-logM) / (1 + exp(2 * logP))
+   Dx[, 1, 3] = -theta[1] * exp(logP) + theta[6] * exp(-logH) * dP
+   Dx[, 3, 3] = -theta[6] * exp(-logH) / (1 + exp(2 * logP))
+
+   Dx
+ }

```

With respect to `theta`, we have the matrix of gradients as follows,

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

which are specified via a function that outputs a 3-D array with dimensions $|\mathbf{I}| \times |\boldsymbol{\theta}| \times D$, where the array slice `[, i, j]` is the partial derivative of the ODE for the j -th system component with respect to the i -th parameter in `theta`:

```

R> hes1logmodelDtheta <- function (theta, x, tvec) {
+   logP = x[, 1]
+   logM = x[, 2]
+   logH = x[, 3]
+
+   Dtheta <- array(0, c(nrow(x), length(theta), ncol(x)))
+   Dtheta[, 1, 1] = -exp(logH)
+   Dtheta[, 2, 1] = exp(logM - logP)
+   Dtheta[, 3, 1] = -1
+   Dtheta[, 4, 2] = -1
+   Dtheta[, 5, 2] = exp(-logM) / (1 + exp(2 * logP))
+   Dtheta[, 1, 3] = -exp(logP)
+   Dtheta[, 6, 3] = exp(-logH) / (1 + exp(2 * logP))
+   Dtheta[, 7, 3] = -1
+
+   Dtheta
+ }

```

At this point, it can be worthwhile to check that the gradients have been coded correctly. This can be done using `testDynamicalModel`, which tests the provided analytic gradients for correctness using numerical differentiation (via a finite difference approximation). To illustrate, we generate some test values for the data and `theta` as input into `testDynamicalModel` along with our ODE functions,

```

R> yTest <- matrix(runif(nrow(y.tilde) * (ncol(y.tilde) - 1)),
+                  nrow = nrow(y.tilde), ncol = ncol(y.tilde) - 1)
R> thetaTest <- runif(7)
R> testDynamicalModel(hes1logmodelODE, hes1logmodelDx, hes1logmodelDtheta,
+                     "Hes1 log", yTest, thetaTest, y.tilde[, "time"])

```

Hes1 log model, with derivatives
Dx and Dtheta appear to be correct

which indicate the numerical and analytic gradients match for both `hes1logmodelDx` and `hes1logmodelDtheta`.

Third, `odeModel` must specify the upper and lower bounds on the parameters `theta`. In this example, all of the seven parameters (a, b, c, d, e, f, g) are non-negative, so we may set the corresponding bounds as 0 and `Inf`.

We are now ready to define the required list containing the three ODE model functions and parameter bounds:

```

R> hes1logmodel <- list(
+   fOde = hes1logmodelODE,
+   fOdeDx = hes1logmodelDx,
+   fOdeDtheta = hes1logmodelDtheta,
+   thetaLowerBound = rep(0, 7),
+   thetaUpperBound = rep(Inf, 7)
+ )

```

Finally, additional settings can be supplied to `MagiSolver` via the list `control`, which may include any number of the following optional inputs. Brief descriptions are provided here, along with references to subsequent sections for further details.

- Settings related to the MCMC sampling setup and initialization of σ , θ and $\mathbf{x}(\mathbf{I})$.
 - `sigma`: a numeric vector of length D , specifies the noise levels σ (i.e., standard deviations of observation noise) at which to initialize MCMC sampling. By default, `MagiSolver` assumes that σ is unknown and initializes it via fitting a GP to the data. If the noise levels are known, supply `sigma` together with the option `useFixedSigma = TRUE`, which will then omit σ from MCMC sampling.
 - `useFixedSigma`: logical, set to `TRUE` if σ is known. If `useFixedSigma = TRUE`, the known values of σ must be supplied via the `sigma` control setting. Default is `FALSE`.
 - `xInit`: a numeric matrix with dimension $|\mathbf{I}| \times D$, specifies values for the system trajectories at which to initialize MCMC sampling. Default is linear interpolation between the observed (non-missing) values of $\mathbf{y}$ to match the resolution of the discretization set $\mathbf{I}$. An optimization routine is applied to Equation 3 (as a function of θ , ϕ and $\mathbf{x}(\mathbf{I})$ for unobserved system components) to initialize any unobserved system components.

- **theta**: a numeric vector of the same length as θ , specifies values for the parameters θ at which to initialize MCMC sampling. By default, **MagiSolver** optimizes Equation 3 as a function of θ only (with **xInit** fixed) to initialize **theta**; if there are unobserved system components, **theta** is initialized together with them (see **xInit**).
- **priorTemperature**: numeric, a tempering factor by which to scale the contribution of the GP prior, to control the influence of the GP prior relative to the likelihood of the observations. Effectively, the log of the GP prior is divided by **priorTemperature**. Default is the total number of observations divided by the total number of discretization points, aggregated over all components; a fuller discussion is provided in Section 4.1.
- Settings related to the GP prior and its hyper-parameters. These options are discussed in detail in Section 4.2.
 - **kerneltype**: string, specifies the type of GP covariance function to use. The default and recommended choice (**generalMatern**) is a Matern kernel with degree of freedom 2.01; it has hyper-parameters ϕ_1 and ϕ_2 for each component that control the variance level and bandwidth, respectively. See Section 4.2 for further discussion regarding this choice; other available choices for **kerneltype** are listed in Appendix F.
 - **phi**: a numeric matrix with dimension $|\phi_d| \times D$, specifies the values of the GP hyper-parameters ϕ , where $|\phi_d|$ is the number of hyper-parameters for each component (i.e., $|\phi_d| = 2$ for **generalMatern**). By default, **MagiSolver** will estimate ϕ automatically for observed components via GP fitting and for unobserved system components via optimization of Equation 3 (see **xInit**).
 - **mu**: a numeric matrix with dimension $|\mathbf{I}| \times D$, specifies values for the mean function of the GP prior of each component. Default is a zero mean function. To use a custom mean function, **mu** must be specified together with **dotmu**.
 - **dotmu**: a numeric matrix with dimension $|\mathbf{I}| \times D$, specifies values for the derivatives of the GP prior mean function for each component. Default is zero.
 - **bandSize**: integer, specifies the size of the diagonal band matrix approximation used to speed up matrix operations. Default **bandSize** is 20, can be increased if **MagiSolver** returns an error indicating numerical instability.
- Settings related to the Hamiltonian Monte Carlo (HMC) sampling algorithm that is used to obtain MCMC draws from the posterior. A description of HMC and the role of these options is provided in Section 4.3.
 - **niterHmc**: integer, specifies the number of HMC iterations to run. Default is 20000.
 - **nstepsHmc**: integer, specifies the number of leapfrog steps per HMC iteration. Default is 200.
 - **burninRatio**: numeric, specifies the proportion of HMC iterations to be discarded as burn-in. Default is 0.5, which discards the first half of the MCMC samples.

- **stepSizeFactor**: numeric, initial leapfrog step size factor for HMC. Default is 0.01, and the leapfrog step size is automatically tuned during burn-in to achieve an acceptance rate between 60-90%.
- Other miscellaneous settings for specialized situations.
 - **skipMissingComponentOptimization**: logical, set to **TRUE** to override automatic optimization for unobserved components. If **skipMissingComponentOptimization** = **TRUE**, values for **xInit** and **phi** must be supplied for all system components. Default is **FALSE**.
 - **positiveSystem**: logical, set to **TRUE** to enforce the constraint that $\mathbf{x}(\mathbf{I})$ is non-negative for all system components. Default is **FALSE**.
 - **verbose**: logical, set to **TRUE** to output diagnostic and sampling progress messages to the console. Default is **FALSE**.

For most settings, the defaults are generally recommended as a reasonable starting point for using **MagiSolver**. The examples provided in this paper will illustrate some cases where it is necessary to override the defaults.

In the **Hes1** example, the noise standard deviations are known. We supply these values via **sigma** and set **useFixedSigma** = **TRUE** in the **control** list, as otherwise σ is treated as a parameter that is sampled within each HMC iteration. We use the defaults for the remaining settings and run **MagiSolver** on the **Hes1** dataset as follows, which stores the output in **hes1result**:

```
R> hes1result <- MagiSolver(y.tilde, hes1logmodel,
+   control = list(sigma = param.true$sigma, useFixedSigma = TRUE))
```

In R, the output of **MagiSolver** is an S3 object of class ‘**magioutput**’ which contains the following list elements:

- **theta**: matrix of MCMC samples for θ after burn-in.
- **xsampled**: array of MCMC samples for the system trajectories $\mathbf{x}(\mathbf{I})$ after burn-in.
- **sigma**: matrix of MCMC samples for σ after burn-in.
- **lp**: vector of log-posterior values at each HMC iteration, after burn-in.
- **phi**: matrix of estimated GP hyper-parameters ϕ .
- **y**, **tvec**, **odeModel**: the data matrix, time vector, and **odeModel** specification from the inputs to **MagiSolver**.

For convenience in R, ‘**magioutput**’ objects have the following associated methods to provide basic inferences from the MCMC samples:

- **print()**: displays a brief summary of the settings used for the **MagiSolver** run.
- **summary()**: generates a table of parameter estimates and credible intervals.

- `plot()`: visualizes the inferred trajectories and credible bands for each component, or generates diagnostic traceplots (i.e., plots of MCMC sampled values vs. the number of iterations) for the parameters.

We have allowed the hyper-parameters ϕ to be automatically estimated in this example (including for the unobserved H component), which is often sufficient in our experience. Further guidelines for setting the GP prior and hyper-parameters are discussed in Section 4.2.

Turning to the MCMC samples, Figure 3 shows the traceplots of `theta` and `lp` (`sigma` is omitted since it is treated as known in this example) as an informal check for convergence, produced using the `plot()` convenience function with `type = "trace"`:

```
R> theta.names <- c("a", "b", "c", "d", "e", "f", "g")
R> plot(hes1result, type = "trace", par.names = theta.names, nplotcol = 4)
```

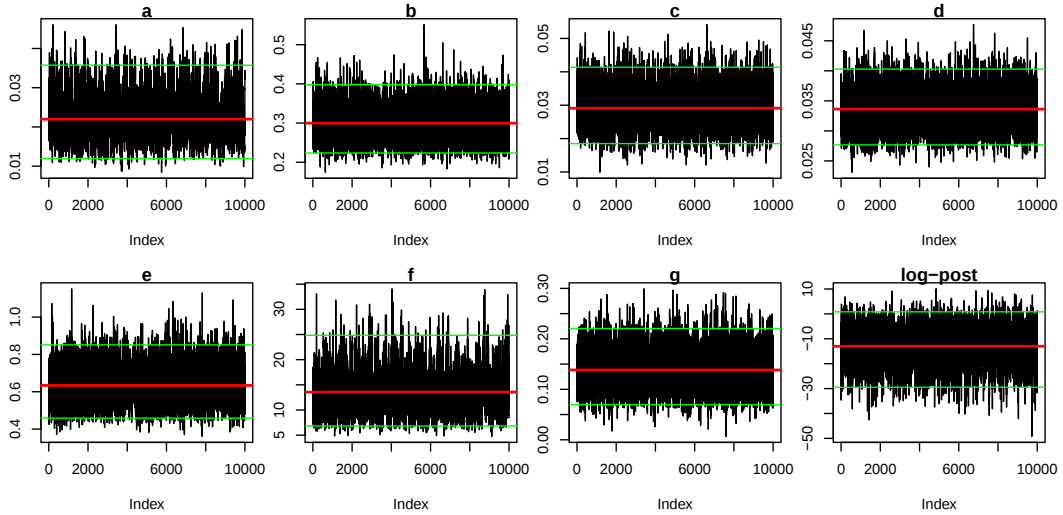


Figure 3: MCMC traceplots for the seven parameters of the Hes1 system and the log-posterior values. The horizontal lines in the plots indicate the posterior mean (red) and limits of the 95% credible interval (green) for each parameter.

The MCMC samples of each parameter randomly scatter around their posterior means (red horizontal lines), which visually indicate that convergence has occurred. The 95% credible intervals (via 2.5 to 97.5 percentiles of the MCMC samples) are shown via the green horizontal lines. We generally suggest taking the posterior mean as the parameter estimate for θ .² The numerical values of these parameter estimates and credible intervals can be extracted using the convenience `summary()` method:

```
R> summary(hes1result, par.names = theta.names)
```

²Other commonly-used Bayesian point estimates include the median (which may be taken component-wise) and the mode (which may be approximated by the MCMC sample with the highest log-posterior value). These can be obtained in **MAGI** by passing the `est` argument to the `plot()` or `summary()` methods. Since **MAGI** is based on a GP prior (and Gaussian tails are thin), the posterior mean (for both θ and $x(I)$) tends to be sufficiently robust and works well in practice.

	a	b	c	d	e	f	g
Mean	0.0220	0.300	0.0291	0.0336	0.634	13.50	0.1380
2.5%	0.0119	0.224	0.0185	0.0277	0.459	6.82	0.0694
97.5%	0.0358	0.398	0.0414	0.0403	0.851	24.80	0.2200

The true parameter values are well contained in the 95% credible intervals, with the exception of g , which only governs the unobserved H component as seen in Equation 4.

Next, we can extract and visualize the sampled system trajectories $\mathbf{x}(\mathbf{I})$. We treat the posterior means as the inferred trajectories, and use the 2.5 to 97.5 percentiles of the MCMC samples to provide 95% credible intervals at each time point in $\mathbf{I}$. These are the default settings of the convenience `plot()` method, which we use to generate Figure 4:

```
R> plot(hes1result, lwd = 2, col = "forestgreen", comp.names = compnames,
+       xlab = "Time", ylab = "log(Level)")
```

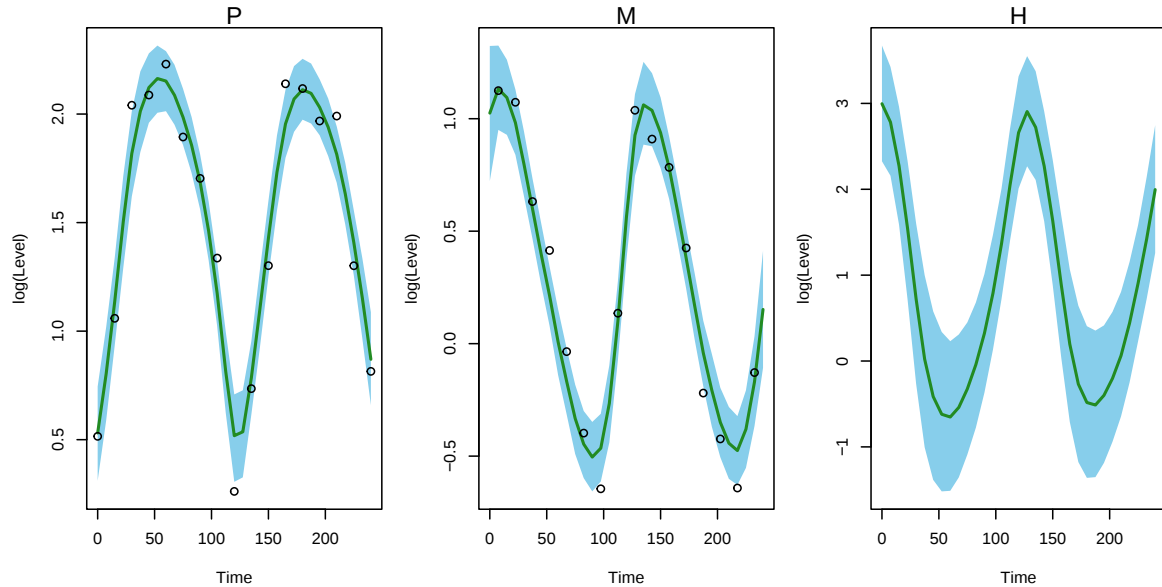


Figure 4: Inferred trajectories from **MAGI** for the three components of the Hes1 system on the log-scale (green curves), generated using the `plot()` method. The blue shaded areas represent 95% credible intervals. The asynchronous noisy observations of P and M are plotted as black circles.

In this example it is helpful to do some further post-processing, and generate a customized plot on the original scale with the true trajectory overlaid. We exponentiate to convert each component to the original scale of measurement:

```
R> xLB <- exp(apply(hes1result$xsampled, c(2, 3),
+                  function(x) quantile(x, 0.025)))
R> xMean <- exp(apply(hes1result$xsampled, c(2, 3), mean))
R> xUB <- exp(apply(hes1result$xsampled, c(2, 3),
+                  function(x) quantile(x, 0.975)))
```

The inferred trajectories (green curves) and blue shaded areas representing the 95% credible intervals are shown in Figure 5, with the noisy observations and true trajectories overlaid as black points and red curves, respectively. The system trajectories are recovered well: the green curves for P and M are consistent with the observed data points and the truth for the entirely unobserved H component is correctly inferred. The plotting code for Figure 5 is given in the replication script.

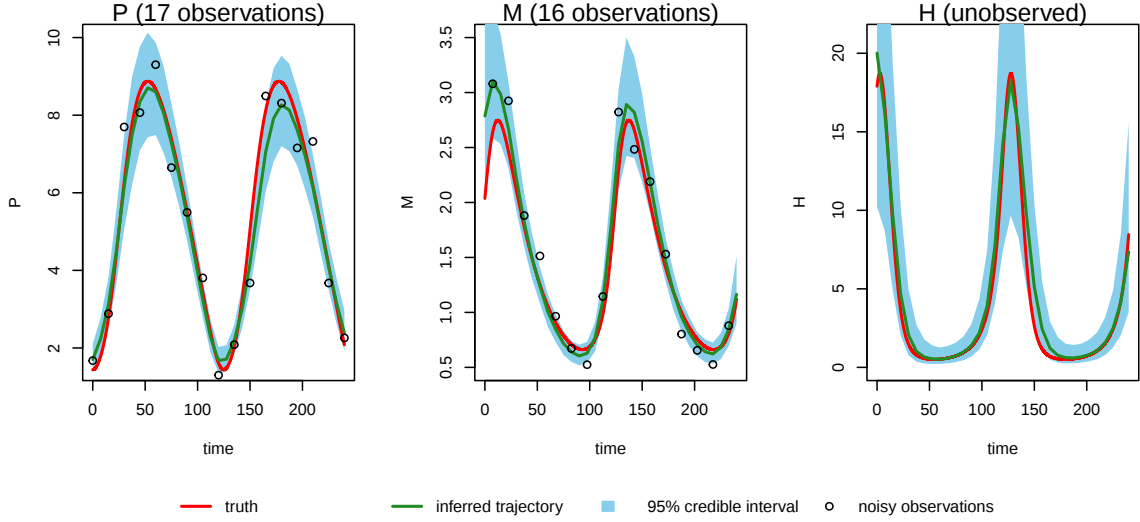


Figure 5: Inferred trajectories from **MAGI** for the three components of the Hes1 system (green curves). The blue shaded areas represent 95% credible intervals. The asynchronous noisy observations of P and M are plotted as black circles, and the red curves represent the true underlying trajectories.

4. Finer control of inference: features and examples

This section presents two additional dynamic system examples to illustrate the role of the discretization set $\mathbf{I}$, the GP prior and its hyper-parameters ϕ , and the HMC algorithm. We discuss how to use these features to obtain finer control over the inference results. The second example (in Section 4.2) also demonstrates a system with equations that explicitly depend on time.

4.1. Choice of discretization set

MAGI constrains the GP to satisfy the ODE system derivatives at the points in the discretization set $\mathbf{I}$. Therefore, increasing the denseness of $\mathbf{I}$ may lead to more accurate inference in some cases, with the trade-off being longer computation time. In practice, it can be a useful strategy to consider running **MagiSolver** with an increasing sequence of discretization sets $\mathbf{I}_0 \subset \mathbf{I}_1 \subset \dots$ to ensure that the estimates obtained are stable.

For the initial run, we recommend taking $\mathbf{I}_0$ as the smallest evenly-spaced set (or approximately so) that includes the observation time points τ . This can help ensure that the system

dynamics are adequately captured throughout the modelled time interval — note that this is not a requirement for running **MagiSolver** itself, which can handle any kind of spacing between time points. Then, we can construct subsequent sets $\mathbf{I}_j$, $j \geq 1$ by inserting one equally-spaced point between each pair of adjacent time points in $\mathbf{I}_{j-1}$.

The function `setDiscretization` can be used to prepare data matrices $\mathbf{y}$ according to this strategy. The command `setDiscretization(y, level = j)` returns a data matrix with $2^j - 1$ equally-spaced points inserted between each observation of $\mathbf{y}$ (i.e., $j = 0$ returns the original matrix $\mathbf{y}$); this works well when $\mathbf{y}$ are evenly spaced. The alternative syntax `setDiscretization(y, by = incr)` can be useful when the observations in $\mathbf{y}$ are unevenly spaced: it returns a data matrix with time points inserted (as needed) to form an equally-spaced discretization set from the first to last observations of $\mathbf{y}$, with interval `incr` between successive discretization points.

Mathematically, as the discretization set becomes more dense, the contributions of the terms in Equation 3 associated with $\mathbf{I}$, i.e., the GP prior $p(\mathbf{x}(\mathbf{I}))$ and $p(\dot{\mathbf{X}}(\mathbf{I}) = \mathbf{f}(\mathbf{x}(\mathbf{I}), \boldsymbol{\theta}, t_{\mathbf{I}}) | \mathbf{x}(\mathbf{I}))$, would become larger relative to the likelihood of the observations. This is because the likelihood term in Equation 3 simplifies as $p(\mathbf{y}(\boldsymbol{\tau}) | \mathbf{x}(\mathbf{I})) = p(\mathbf{y}(\boldsymbol{\tau}) | \mathbf{x}(\boldsymbol{\tau}))$ for any $\boldsymbol{\tau} \subset \mathbf{I}$ and does not change as $\mathbf{I}$ becomes more dense, i.e., only the points in $\mathbf{I}$ corresponding to observations have an associated likelihood contribution. Therefore, **MAGI** automatically uses a tempering hyper-parameter β to maintain the balance between the GP prior and the likelihood across different discretization sets. This helps ensure that parameter inference reaches a stable result over an increasing sequence of discretization sets. Specifically, $p(\mathbf{x}(\mathbf{I}))p(\dot{\mathbf{X}}(\mathbf{I}) = \mathbf{f}(\mathbf{x}(\mathbf{I}), \boldsymbol{\theta}, t_{\mathbf{I}}) | \mathbf{x}(\mathbf{I}))$ is tempered as $\left[p(\mathbf{x}(\mathbf{I}))p(\dot{\mathbf{X}}(\mathbf{I}) = \mathbf{f}(\mathbf{x}(\mathbf{I}), \boldsymbol{\theta}, t_{\mathbf{I}}) | \mathbf{x}(\mathbf{I})) \right]^{1/\beta}$, where our recommended value $\beta = D|\mathbf{I}| / \sum_{d=1}^D |\boldsymbol{\tau}_d|$ is the total number of discretization points divided by the total number of observations (aggregated over all components). For example if $|\mathbf{I}|$ is doubled, then tempering effectively reduces the GP contribution on the log-scale by half to compensate. (A custom value for β can be set by the user via `priorTemperature` in the optional `control` list to **MagiSolver**, but this is not generally recommended.)

We illustrate this idea of increasing discretization sets, on a dataset of noisy observations simulated from the classic FitzHugh-Nagumo (FN) equations for $X = (V, R)$ that model spike potentials of neurons (FitzHugh 1961):

$$\mathbf{f}(X, \boldsymbol{\theta}, t) = \begin{pmatrix} c(V - \frac{V^3}{3} + R) \\ -\frac{1}{c}(V - a + bR) \end{pmatrix}, \quad (5)$$

where V and R are the voltage and recovery variables, and $\boldsymbol{\theta} = (a, b, c)$ are the parameters to be estimated.

We begin by loading the dataset and setting a random seed for reproducibility:

```
R> data("FNdat")
R> set.seed(12321)
```

The observation time points of `FNdat` are $t = 0, 0.5, 1, \dots, 10$ at intervals of 0.5, along with $t = 11, 12, 13, 14, 15, 17, 20$. Following the suggested strategy above, we create a data matrix corresponding to the first discretization set $\mathbf{I}_0$, taken to be the 41 equally-spaced points $\{0, 0.5, 1, \dots, 20\}$:

```
R> y_I0 <- setDiscretization(FNdat, by = 0.5)
```

We also create data matrices corresponding to the denser sets $\mathbf{I}_1, \mathbf{I}_2, \mathbf{I}_3$ by successively inserting one equally-spaced time point between existing ones:

```
R> y_I1 <- setDiscretization(y_I0, level = 1)
R> y_I2 <- setDiscretization(y_I0, level = 2)
R> y_I3 <- setDiscretization(y_I0, level = 3)
```

The gradients of the ODEs with respect to X and θ are as follows:

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

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

Functions that code the FN equations (`fnmodelODE`) and their gradients (`fnmodelDx` and `fnmodelDtheta`) are set up analogously to the `Hes1` model, and are shown in Appendix C. Using these, we create the `odeModel` list input:

```
R> fnmodel <- list(
+   fOde = fnmodelODE,
+   fOdeDx = fnmodelDx,
+   fOdeDtheta = fnmodelDtheta,
+   thetaLowerBound = c(0, 0, 0),
+   thetaUpperBound = c(Inf, Inf, Inf)
+ )
```

We can now run `MagiSolver` with the discretization sets we constructed and 10000 HMC iterations. Since the noise level is unknown in this dataset, σ will also be inferred via MCMC sampling. Note that the dimensionality of the variables being sampled effectively doubles with each successive discretization set. Thus, the outputs of the HMC iterations can become increasingly ‘sticky’ (i.e., having higher autocorrelation) with denser discretization sets. For more detail on this point, see Section 4.3 for a discussion of HMC and its settings. One way to circumvent this is to increase the number of leapfrog steps per HMC iteration. We have illustrated that below, by setting `nstepsHmc = 1000` for our densest set $\mathbf{I}_3$ (otherwise the default is 200 leapfrog steps).

```
R> FNres0 <- MagiSolver(y_I0, fnmodel, control=list(niterHmc = 10000))
R> FNres1 <- MagiSolver(y_I1, fnmodel, control=list(niterHmc = 10000))
R> FNres2 <- MagiSolver(y_I2, fnmodel, control=list(niterHmc = 10000))
R> FNres3 <- MagiSolver(y_I3, fnmodel,
+   control=list(niterHmc = 10000, nstepsHmc = 1000))
```

To compare the estimates, we make use of `summary()` to extract the posterior means and 95% credible intervals for both θ and σ from each model fit:

```
R> FNpar.names <- c("a", "b", "c", "sigmaV", "sigmaR")
R> FNsummary <- lapply(list(FNres0, FNres1, FNres2, FNres3),
+                       function(x) summary(x, sigma = TRUE, par.names = FNpar.names))
```

We then plot these posterior summaries for each parameter and discretization set:

```
R> layout(rbind(c(1:5), rep(6, 5)), heights = c(5, 0.25))
R> for (i in 1:length(FNpar.names)) {
+   par(mar = c(2, 4, 1.5, 1))
+   estCI <- sapply(FNsummary, function(x) x[,i])
+   plot(1:4, xlim = c(0, 5), ylim = c(min(estCI[2, ]), max(estCI[3, ])),
+        xaxt = 'n', xlab = '', ylab = '', type = 'n')
+   segments(1:4, y0 = estCI[2, ], y1 = estCI[3, ], col = 1:4, lwd = 2)
+   mtext(FNpar.names[i])
+   points(1:4, estCI[1, ], col = 1:4, cex = 2)
+ }
R> par(mar = rep(0, 4))
R> plot(1, type = 'n', xaxt = 'n', yaxt = 'n',
+      xlab = NA, ylab = NA, frame.plot = FALSE)
R> legend("center", c("I0", "I1", "I2", "I3"),
+      col = 1:4, lwd = 4, horiz = TRUE, bty = "n")
```

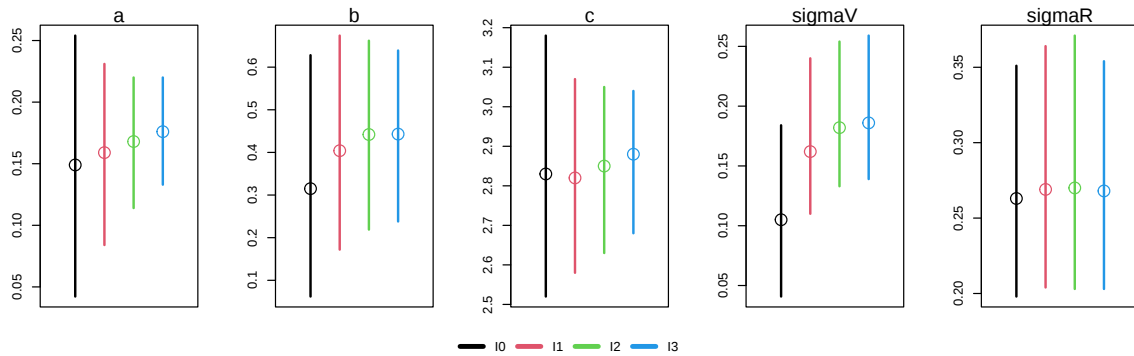


Figure 6: Posterior means (points) and 95% credible intervals (vertical bars) for each parameter, from using `MagiSolver` on the simulated FN dataset with the discretization sets $I_0 \subset I_1 \subset I_2 \subset I_3$.

The panels of Figure 6 show that the posterior means (points) and credible intervals (vertical bars) visibly shift for b and σ_V , as we use the successive discretization sets from I_0 to I_1 to I_2 . Meanwhile comparing I_2 and I_3 for all the parameters, the posterior means are fairly similar and the credible intervals largely overlap, indicating that the inference results are stable.

To provide a further check, we can use the ODE solver to reconstruct the trajectories implied by the estimates of the parameters and the initial conditions. Again, we define a wrapper to facilitate calling `ode` from `deSolve`:

```
R> fnmodelODEsolve <- function(tvec, state, parameters) {
+   list(as.vector(fnmodelODE(parameters, t(state), tvec)))
+ }
```

We define a helper function that invokes the ODE solver using the posterior means of θ and $V(0), R(0)$ from the `MagiSolver` output. Note that the MCMC samples for the initial conditions can be extracted from the `xsampled` array, as shown below to obtain `x0.est`:

```
R> tvec <- seq(0, 20, by = 0.01)
R> FNcalcTraj <- function(res) {
+   x0.est <- apply(res$xsampled[, 1, ], 2, mean)
+   theta.est <- apply(res$theta, 2, mean)
+
+   x <- deSolve::ode(y = x0.est, times = tvec,
+                     func = fnmodelODEsolve, parms = theta.est)
+   x
+ }
```

We then compute these reconstructed trajectories for the estimates based on the four discretization sets and plot them in Figure 7. Visually, all four reconstructed trajectories fit the observed data well. There are some minor shifts in the trajectories as the discretization sets get denser from I_0 to I_1 to I_2 , while those for I_2 and I_3 (green and blue) become nearly indistinguishable (except for $0 \leq t \leq 2$ of the R component). The code to produce the figure is below:

```
R> FNtr <- lapply(list(FNres0, FNres1, FNres2, FNres3), FNcalcTraj)

R> layout(rbind(c(1, 2), c(3, 3)), heights = c(5, 0.25))
R> plot(FNdat$time, FNdat$V, xlab = "Time", ylab = "V")
R> matplot(tvec, sapply(FNtr, function(x) x[, 2]),
+   type = "l", lty = 1, add = TRUE)
R> plot(FNdat$time, FNdat$R, xlab = "Time", ylab = "R")
R> matplot(tvec, sapply(FNtr, function(x) x[, 3]),
+   type = "l", lty = 1, add = TRUE)
```

The following code adds the legend at the bottom:

```
R> par(mar = rep(0, 4))
R> plot(1, type = 'n', xaxt = 'n', yaxt = 'n',
+   xlab = NA, ylab = NA, frame.plot = FALSE)
R> legend("center", c("I0", "I1", "I2", "I3"),
+   col = 1:4, lwd = 4, horiz = TRUE, bty = "n")
```

For a numerical comparison, we can calculate the root-mean-squared deviations (RMSDs) between the noisy observations and the reconstructed trajectories at those time points. We can obtain these as follows:

```
R> FN.rmsd <- sapply(FNtr, function(x)
+   sqrt(colMeans((subset(x, time %in% FNdat$time) - FNdat[, 2:3])^2)))
R> colnames(FN.rmsd) <- c("I0", "I1", "I2", "I3")
R> round(FN.rmsd, 3)
```

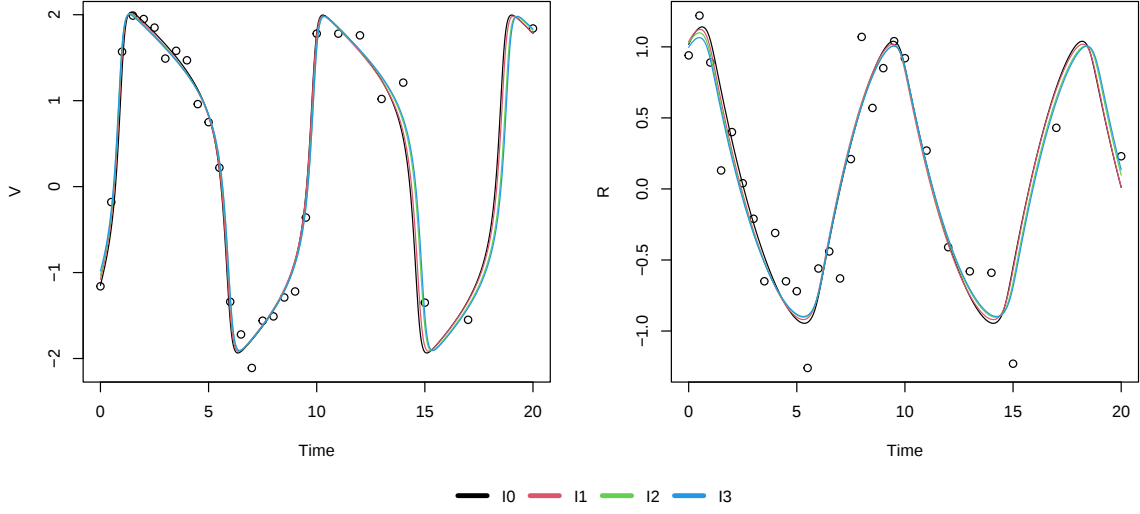


Figure 7: Reconstructed system trajectories (solid curves) based on the estimated parameters and initial conditions from the simulated FN dataset using **MagiSolver** with the discretization sets $\mathbf{I}_0 \subset \mathbf{I}_1 \subset \mathbf{I}_2 \subset \mathbf{I}_3$. The noisy observations (points) are also plotted.

	$\mathbf{I}_0$	$\mathbf{I}_1$	$\mathbf{I}_2$	$\mathbf{I}_3$
V	0.234	0.212	0.176	0.167
R	0.281	0.277	0.260	0.255

These results indicate that the estimated parameters fit the observed data better (lower RMSDs) as the denseness of the discretization set is increased. The improvement going from $\mathbf{I}_2$ and $\mathbf{I}_3$ is fairly minimal, which confirms that a stable inference result has been achieved and there is no need to further increase the discretization set to $\mathbf{I}_4$.

4.2. Setting of hyper-parameters

The GP prior on $\mathbf{x}(t)$ has two ingredients: the mean function $\boldsymbol{\mu}(t)$ and covariance kernel/function $\mathcal{K}$. In applications of GPs, it is customary to set $\boldsymbol{\mu}(t) = 0$ in the absence of specific prior information (see, e.g., chapter 2 of [Williams and Rasmussen 2006](#)). The reason this can work well in practice is that the GP, once *conditioned on observations*, will have a mean function that tends to follow the observations. In **MAGI**, the GP is additionally conditioned on the ODE manifold constraint, which further aligns the GP mean function with the dynamics specified by the ODEs. Thus, our general recommendation is to assume a zero-mean GP prior for simplicity; all of the examples in the paper take this approach and have good inference results. The user may input a custom prior mean function to **MAGI** by evaluating $\boldsymbol{\mu}(\mathbf{I})$ and $\dot{\boldsymbol{\mu}}(\mathbf{I})$, i.e., $\boldsymbol{\mu}(t)$ and its derivative evaluated at the discretization set $\mathbf{I}$, and providing them to **MagiSolver** via the optional arguments `mu` and `dotmu`. One potential scenario where this might be helpful is to provide prior guidance for unobserved components, though as demonstrated by the recovery of the unobserved H component in the Hes1 example (Section 3), this kind of input is not generally needed.

For a given covariance function $\mathcal{K}$, the GP hyper-parameters ϕ control the overall prior variance level and prior smoothness/bumpiness of each component’s trajectory. Specifically, the default choice and our recommendation for general usage in **MAGI** is a Matern covariance function $\mathcal{K}$ of the form

$$\mathcal{K}(s, t) = \phi_1 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu} \frac{r}{\phi_2} \right)^\nu B_\nu \left(\sqrt{2\nu} \frac{r}{\phi_2} \right), \quad (6)$$

where $r = |s - t|$ is the absolute difference between two time points s and t , $\nu = 2.01$ is the smoothness parameter (which ensures twice-differentiable curves), Γ is the gamma function, and B_ν is the modified Bessel function of the second kind. Larger values of ϕ_1 therefore favor curves with higher variance, and larger values of ϕ_2 favor curves with more time-dependence between nearby time points. Each system component has its own set of (ϕ_1, ϕ_2) values to ensure that the GP has sufficient flexibility to model its dynamics.

Other covariance functions are available in **MAGI** and may be selected using the `kerneltype` optional argument to `MagiSolver`. See Appendix F for their specification and details. Note that for a covariance function to be compatible with **MAGI**, the corresponding GP prior on $\mathbf{x}(t)$ must satisfy two conditions: (i) the GP derivative $\dot{\mathbf{x}}(t)$ exists, as implied by the definition of the ODE structure; (ii) $\dot{\mathbf{x}}(t)$ is also a GP. Together, this requires the covariance function $\mathcal{K}$ to be twice-differentiable (i.e., $\frac{\partial^2}{\partial s \partial t} \mathcal{K}(s, t)$ exists). The number of times that $\mathcal{K}$ is differentiable (with respect to r) relates to the smoothness of the GP; higher-order differentiability corresponds to smoother curves. Taking the Matern covariance in Equation 6 specifically, $\mathcal{K}$ is k -times differentiable if and only if $\nu > k$; thus, smaller values of ν are more capable of modeling rough or bumpy trajectories (see, e.g., pp. 84-85 of Williams and Rasmussen 2006). This motivates our default choice of `generalMatern` (i.e., Equation 6 with $\nu = 2.01$, so that the kernel meets the twice-differentiable requirement and is capable to model relatively rough curves), which in our experience gives the best inference results over the widest range of system behavior (whether rough or smooth). Other covariance functions commonly used in GP applications (such as the Matern covariance with $\nu = 5/2$ or the radial basis function kernel) encode stronger smoothness assumptions that hinder the GP from capturing sharp changes in $\mathbf{x}(t)$, which may in turn lead to bias in the parameter estimates.³

By default, `MagiSolver` automatically estimates ϕ for each system component, depending on the availability of observed data. Briefly, for components with observations, a GP with the selected covariance function is fitted to the data via *maximum a posteriori* (MAP) estimation with a weakly informative Normal prior for ϕ_2 (and flat priors otherwise), which provides ϕ and a value of σ to initialize MCMC sampling (if σ is unknown). Then to handle unobserved components, optimization is applied to the full posterior in Equation 3 as a function of θ and ϕ , $\mathbf{x}(\mathbf{I})$ for unobserved components, with the previously initialized values of σ , ϕ , and $\mathbf{x}(\mathbf{I})$ for observed components held fixed.

³In general, given that the underlying ODE system can be rough or smooth, it is our experience that using the Matern covariance function with a small value of ν , such as $\nu = 2.01$, provides the GP approximation the capability to model a wide range of dynamic systems (rough or smooth). Our experience suggests that another potential source of bias in **MAGI** is the fact that the ODE manifold constraint in practical computation can only be applied at a finite set of time points (i.e., $W_I = 0$) rather than on the entire interval (i.e., $W = 0$). For systems where $\mathbf{f}$ has very sharp changes, **MAGI** may still be applicable by choosing a smaller value of the hyper-parameter ϕ_2 so that those sharp changes can be better approximated by the GP. In this case, visual assessments of the GP fit to the data can provide a good indication of whether subsequent inference with **MAGI** will be successful. The example in this section demonstrates how to apply this strategy.

The values of ϕ are held fixed during MCMC sampling for θ , σ , and $\mathbf{x}(\mathbf{I})$. This may be contrasted with a full Bayesian approach for handling ϕ , where ϕ would also be sampled. MCMC sampling for ϕ is however expensive as each update requires recomputing the covariance matrices associated with $\mathbf{x}(\mathbf{I})$ (e.g., see Titsias, Rattray, and Lawrence 2011) and $\dot{\mathbf{x}}(\mathbf{I})$ as needed in **MAGI**. Thus, we follow the approach of estimating ϕ based on its marginal likelihood and holding it fixed (e.g., see chapter 5 of Williams and Rasmussen 2006); one potential disadvantage is that this approach does not account for uncertainty in ϕ . In practice, credible intervals for θ tend to be fairly stable so long as ϕ lies within a range that is appropriate for the data; an empirical assessment for this point is provided at the end of this section.

Fixing ϕ also allows us to leverage techniques to speed up calculations on the covariance matrices associated with the GPs. Specifically, let $\mathcal{K}_d(s, t)$ be the fitted GP covariance function for component d , then the following $|\mathbf{I}| \times |\mathbf{I}|$ matrices are involved in its GP multivariate normal distribution at the discretization points in $\mathbf{I}$:

$$\begin{cases} C_d &= \mathcal{K}_d(\mathbf{I}, \mathbf{I}) \\ m_d &= {}'\mathcal{K}_d(\mathbf{I}, \mathbf{I})\mathcal{K}_d(\mathbf{I}, \mathbf{I})^{-1} \\ \Psi_d &= \mathcal{K}''_d(\mathbf{I}, \mathbf{I}) - {}'\mathcal{K}_d(\mathbf{I}, \mathbf{I})\mathcal{K}_d(\mathbf{I}, \mathbf{I})^{-1}\mathcal{K}'_d(\mathbf{I}, \mathbf{I}) \end{cases}$$

where $'\mathcal{K}_d = \frac{\partial}{\partial s}\mathcal{K}_d(s, t)$, $\mathcal{K}'_d = \frac{\partial}{\partial t}\mathcal{K}_d(s, t)$, and $\mathcal{K}''_d = \frac{\partial^2}{\partial s \partial t}\mathcal{K}_d(s, t)$. Here, C_d is the covariance matrix of $x_d(\mathbf{I})$, Ψ_d is the covariance matrix of $\dot{x}_d(\mathbf{I})$, and m_d is the projection matrix that maps $x_d(\mathbf{I})$ to the mean function of $\dot{x}_d(\mathbf{I})$. With the GP structure, the covariance between two time points tends to be non-negligible only when they are nearby; thus the off-diagonal entries of C_d , Ψ_d , and their inverses quickly decay to zero. Likewise, the relation between $x_d(\mathbf{I})$ and its derivative $\dot{x}_d(\mathbf{I})$ is local, so off-diagonal entries of m_d also decay to zero quickly. Since ϕ is fixed, the matrices C_d^{-1} , Ψ_d^{-1} , and m_d needed in the multivariate normal densities (Equation 3) can be pre-computed. Furthermore, we may approximate each of them with sparse band matrices (i.e., non-zero only within `bandSize` diagonals on either side of the main diagonal), which reduces the matrix multiplication complexity in Equation 3 from $O(|\mathbf{I}|^2)$ to $O(|\mathbf{I}|)$. This band matrix approximation works best when $\mathbf{I}$ is evenly-spaced, as recommended in Section 4.1. In our experience, a band size of 20 works for most problems and is the default in **MAGI**. If the approximation fails (i.e., the quadratic form diverges), a warning message that suggests a larger band size will be automatically shown. A different band size can be chosen by setting `bandSize` in the `control` list to `MagiSolver`.

Often, the automatic estimates of ϕ will be within a reasonable range that permits accurate inference of θ ; however, this is not guaranteed for all datasets, in which case we may manually override their values for better control over the inference results. This is done by supplying `phi` in the `control` list to `MagiSolver`. **MAGI** includes the `gpsmoothing` function for fitting a GP to data, along with the `gpmean` and `gpcov` functions to compute the resulting mean vector and covariance matrix conditioned on the data. This allows the user to examine and assess the estimated values of ϕ prior to running `MagiSolver`.

The example in this section demonstrates how these functions can be used, and how the appropriateness of hyper-parameter choices can be assessed, in the context of a system with equations that explicitly depend on time. We consider the three-component system $X = (T_U, T_I, V)$ for HIV infection described in the simulation study of Liang, Miao, and Wu (2010), where T_U, T_I are the concentrations of uninfected and infected cells, and V is the

viral load:

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

In this system, $\eta(t) = 9 \times 10^{-5} \times (1 - 0.9 \cos(\pi t / 1000))$ is an oscillating infection rate over time (in days), and the parameters to be estimated are $\boldsymbol{\theta} = (\lambda, \rho, \delta, N, c)$. Functions for these ODEs (`hivtdmodelODE`) and their gradients (`hivtdmodelDx` and `hivtdmodelDtheta`) are shown in Appendix D. Then the `odeModel` list for input to `MagiSolver` is as follows:

```
R> hivtdmodel <- list(
+   fOde = hivtdmodelODE,
+   fOdeDx = hivtdmodelDx,
+   fOdeDtheta = hivtdmodelDtheta,
+   thetaLowerBound = rep(0, 5),
+   thetaUpperBound = rep(Inf, 5)
+ )
```

We define the component names and labels for later use:

```
R> compnames <- c("TU", "TI", "V")
R> complabels <- c("Concentration", "Concentration", "Load")
```

We also create a list with the simulation inputs (parameter values `theta`, initial conditions `x0`, noise levels `sigma`, and observation times) that mimic those in the referenced paper:

```
R> param.true <- list(
+   theta = c(36, 0.108, 0.5, 1000, 3),
+   x0 = c(600, 30, 1e5),
+   sigma = c(sqrt(10), sqrt(10), 10),
+   times = seq(0, 20, 0.2)
+ )
```

Next we invoke the numerical solver and simulate noisy observations from this system over the specified time interval ($t = 0$ to 20) with measurements at intervals of 0.2 and noise SD `param.true$sigma`, again with a random seed set for reproducibility:

```
R> set.seed(12321)
R> modelODE <- function(tvec, state, parameters) {
+   list(as.vector(hivtdmodelODE(parameters, t(state), tvec)))
+ }

R> xtrue <- deSolve::ode(y = param.true$x0, times = param.true$times,
+   func = modelODE, parms = param.true$theta)
R> y <- data.frame(xtrue)
R> for(j in 1:(ncol(y) - 1)){
+   y[, 1+j] <- y[, 1+j] + rnorm(nrow(y), sd = param.true$sigma[j])
+ }
```

These noisy observations are plotted as the black points in Figure 8a for each component. The system dynamics are characterized by rapid changes from $t = 0$ to $t = 5$, with the V component exhibiting a particularly steep decline during the first day.

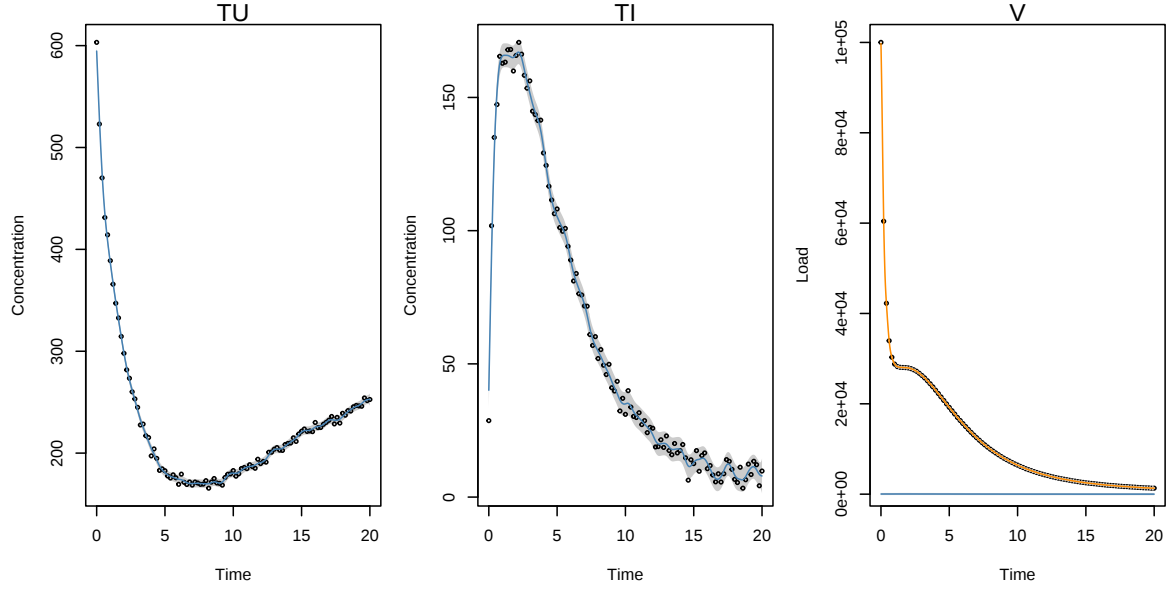
We can use `gpsmoothing` to perform a preliminary GP fit and obtain estimates of ϕ and σ for each component. The inputs to `gpsmoothing` are the noisy observations and vector of time points (ODE information is not used at this stage), and the function returns a list with `phi` and `sigma` elements. Its usage is demonstrated as follows, where we create `phiEst` and `sigmaInit` to store the results, then perform GP fitting for each system component and extract the estimates:

```
R> phiEst <- matrix(0, nrow = 2, ncol = ncol(y) - 1)
R> sigmaInit <- rep(0, ncol(y) - 1)
R> for (j in 1:(ncol(y) - 1)) {
+   hyperparam <- gpsmoothing(y[, j+1], y[, "time"])
+   phiEst[, j] <- hyperparam$phi
+   sigmaInit[j] <- hyperparam$sigma
+ }
```

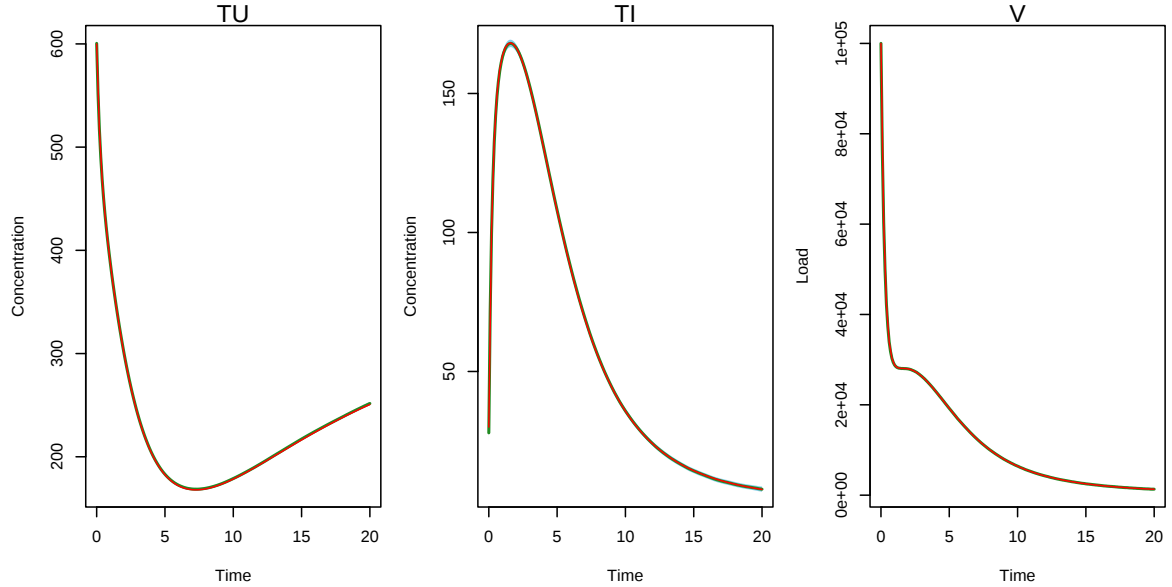
Next, we can visualize the GP fit implied by these values of ϕ and σ , with the help of the `gpmean` and `gpcov` functions. These compute the GP mean vector and covariance matrix conditioned on the observations, ϕ , and σ . To plot a smooth curve of the GP fit, we carry out this computation on a fairly dense set of time points (`tOut`), and the diagonal of the covariance matrix can be used to produce credible bands (e.g., ± 1.96 SDs):

```
R> tOut <- seq(0, 20, by = 0.025)
R> for (j in 1:3) {
+   plot(y[, "time"], y[, j+1], type = 'n',
+        xlab = "Time", ylab = complabels[j])
+   mtext(compnames[j])
+   fitMean <- gpmean(y[, j+1], y[, "time"], tOut, phiEst[, j], sigmaInit[j])
+   fitCov <- gpcov(y[, j+1], y[, "time"], tOut, phiEst[, j], sigmaInit[j])
+   gp_UB <- fitMean + 1.96 * sqrt(diag(fitCov))
+   gp_LB <- fitMean - 1.96 * sqrt(diag(fitCov))
+   polygon(c(tOut, rev(tOut)), c(gp_UB, rev(gp_LB)),
+           col = "grey80", border = NA)
+   points(y[, "time"], y[, j+1], cex = 0.5)
+   lines(tOut, fitMean, type = 'l', col = "steelblue")
+ }
```

The resulting blue curves in Figure 8a show the GP means of each component, and the grey bands are 95% intervals (i.e., mean ± 1.96 SD, where the SDs are extracted from the corresponding GP covariance matrix). We see that the automatically estimated GP hyperparameters ϕ_1 and ϕ_2 allow the mean curves to follow the observations reasonably well for components T_U and T_I ; however, the sharp trend in component V is not captured. The fitted mean curve for V is close to zero, or equivalently, the observations for V are incorrectly attributed as noise. Recall that at this stage, the mean curves are conditional on the



(a) Initial GP fitting to a sample dataset simulated from the HIV model. The black points are the noisy observations. The blue curves represent **MAGI**'s automatic fit of the GP mean conditional on the data (without ODE information), and the grey bands represent 95% intervals based on the corresponding GP covariance. The automatically estimated GP hyper-parameters ϕ_1 and ϕ_2 allow the curves to follow the observations reasonably well for components T_U and T_I ; however, the sharp trend in component V is not captured. By using a custom specification of ϕ_1 and ϕ_2 for component V , the resulting mean curve (orange) can follow the observations well.



(b) Final inferred trajectories (green curves) for the HIV model with 95% credible bands (light blue). The true trajectories are superimposed in red. Note that the credible bands are very narrow, and are only visible for the T_I component.

Figure 8: Hyper-parameter setup and inference for a sample dataset simulated from the HIV model. Panel (a) provides a visual assessment of the GP hyper-parameters and their initial fit to the data. Panel (b) shows the final inferred trajectories from **MAGI**, which closely follow the true trajectories.

observations only (i.e., ODE manifold constraint is not yet included), so some wiggleness is not unusual, as seen for components T_U and T_I . The key visual indicator for a reasonable ϕ estimate is a mean curve that roughly captures the trajectory implied by the observations.

The phenomenon seen in the V component might be explained by the fact that GP fitting tends to prefer smoother curves (due to the twice-differentiable requirement, see Section 4.2), which has the effect here of “oversmoothing” the trajectory to a near-constant mean function (see also footnote 3). Using these default hyper-parameters could in turn lead to incorrect inference of θ . To address this issue, we could (i) choose a smaller value of ϕ_2 so that the GP can model sharper changes, and (ii) adjust ϕ_1 according to the overall scale of the component. We examine the automatic estimates of ϕ and σ so that we can make adjustments:

```
R> colnames(phiEst) <- compnames
R> phiEst
```

	TU	TI	V
[1,]	37538.16828	11083.870501	14299.089897
[2,]	3.91358	2.755717	1.731937

```
R> sigmaInit
```

[1]	3.378930	3.757803	14158.670863
-----	----------	----------	--------------

These estimates appear reasonable for the T_U and T_I components, with a fairly small `sigmaInit` that corroborates Figure 8a, i.e., the GP fit follows the observed data closely. This is further evidenced by the ϕ_1 values for T_U and T_I , where we see that $\sqrt{\phi_1}$ resembles the scale of those components: ~ 600 for T_U and ~ 100 for T_I . However, the very large value $\sigma \approx 14000$ for V confirms that the sharp decline in its trajectory is being incorrectly fitted as noise. This is also reflected in its ϕ estimates: $\phi_1 \approx 14300$ is too small to adequately capture the variation in V , while the bandwidth $\phi_2 \approx 1.7$ is too large to model the sharp initial decline.

Following the strategy described above, we use the `control` list to manually specify more suitable values of ϕ for the V component. In practice, inference is relatively insensitive to ϕ over a reasonable range of values, so a high level of precision is not required for this step. We increase ϕ_1 to 10^7 and decrease ϕ_2 to 0.5:

```
R> phiEst[, 3] <- c(1e7, 0.5)
```

Since the noise levels σ are treated as unknown, the values obtained by GP fitting are only used to initialize the MCMC sampler. We may initialize σ at a more reasonable (smaller) value for V that corresponds with the update to ϕ , which can also help obtain faster MCMC convergence:

```
R> sigmaInit[3] <- 100
```

To assess whether these manual adjustments to ϕ and σ for V are reasonable, we can recalculate the GP mean and covariance with these new values:

```
R> j <- 3
R> fitMean <- gpmean(y[, j+1], y[, "time"], tOut, phiEst[, j], sigmaInit[j])
R> fitCov <- gpcov(y[, j+1], y[, "time"], tOut, phiEst[, j], sigmaInit[j])
```

We add the updated mean curve (plotted in orange) and 95% intervals to Figure 8a:

```
R> gp_UB <- fitMean + 1.96 * sqrt(diag(fitCov))
R> gp_LB <- fitMean - 1.96 * sqrt(diag(fitCov))
R> polygon(c(tOut, rev(tOut)), c(gp_UB, rev(gp_LB)),
+         col = "grey80", border = NA)
R> lines(tOut, fitMean, col = "darkorange")
```

We see that the mean curve now follows the overall trajectory of the V observations. (The 95% bands are very narrow and not visible in the plot.) This visually confirms that the new values of ϕ for component V are reasonable as input to `MagiSolver` for carrying out further inference.

We proceed to create a discretization set $\mathbf{I}$ that adds one equally-spaced time point between observations:

```
R> y_I <- setDiscretization(y, level = 1)
```

Then we run `MagiSolver`, supplying `phi` and `sigma` using the values we specified (recall that `phi` is fixed at its supplied value, while `sigma` will be sampled via MCMC starting from its supplied value):

```
R> HIVresult <- MagiSolver(y_I, hivtdmodel,
+                          control = list(phi = phiEst, sigma = sigmaInit))
```

We compute posterior means and 95% credible intervals for θ and σ :

```
R> summary(HIVresult, sigma = TRUE, par.names =
+   c("lambda", "rho", "delta", "N", "c", "sigma_TU", "sigma_TI", "sigma_V"))
```

	lambda	rho	delta	N	c	sigma_TU	sigma_TI	sigma_V
Mean	35.9	0.1070	0.499	958	2.88	3.21	3.41	13.60
2.5%	34.1	0.0982	0.494	943	2.84	2.79	2.95	2.23
97.5%	37.7	0.1150	0.504	973	2.92	3.71	3.96	37.80

The estimates for λ , ρ , δ are very close to the true values, while N and c have some slight bias. The estimates of σ likewise reflect their true simulation values, including σ_V which we had initialized at 100. We also extract and plot the inferred trajectories (green) in Figure 8b together with 95% credible bands (light blue) and the truth (red) superimposed:

```
R> xMean <- apply(HIVresult$xsampled, c(2, 3), mean)
R> xLB <- apply(HIVresult$xsampled, c(2, 3), function(x) quantile(x, 0.025))
R> xUB <- apply(HIVresult$xsampled, c(2, 3), function(x) quantile(x, 0.975))

R> par(mfrow = c(1, 3), mar = c(4, 4, 1.5, 1))
R> for (i in 1:3) {
+   plot(y_I$time, xMean[, i], type = "n", xlab = "Time", ylab = complabels[i])
+   mtext(compnames[i])
+ }
```

```

+   polygon(c(y_I$time, rev(y_I$time)), c(xUB[, i], rev(xLB[, i])),
+         col = "skyblue", border = NA)
+   lines(y_I$time, xMean[, i], col = "forestgreen", lwd = 2)
+   lines(param.true$times, xtrue[, i+1], col = "red", lwd = 1)
+ }

```

In Figure 8b, the inferred trajectories are indistinguishable from the truth, and the 95% credible bands are very narrow. The bands are only visible for portions of the T_I trajectory. Thus, good inference results can be obtained in this example by ensuring that the choice of the hyper-parameters ϕ is reasonable.

Lastly, we provide an empirical assessment for the sensitivity of the parameter inference to the GP hyper-parameters ϕ . Since ϕ is held fixed during MCMC sampling, estimates and credible intervals for θ can have some dependence on the specific value of ϕ used. In this HIV example, we used ϕ values that were automatically fit for T_U and T_I , and manually specified for the V component. To assess the effect of ϕ , we considered randomly sampling new values for each ϕ_1 and ϕ_2 , and re-running **MagiSolver** with those values. Each entry of ϕ was drawn uniformly as $[2/3, 3/2]$ times its original value, which could be considered a reasonable range of variation:

```

R> for (j in 1:3) {
+   phiEst[, j] <- runif(2, 2/3 * phiEst[, j], 3/2 * phiEst[, j])
+ }

```

We repeated this for 100 different random seeds. Figure 9 plots the posterior means (points) and 95% credible intervals (vertical bars) for the five parameters over all these random ϕ . The upper and lower limits of the 95% credible intervals inferred using the original values of ϕ are shown via the dashed horizontal lines. For all five parameters and all 100 repetitions, the 95% credible intervals overlap with the original ones. The intervals for λ, ρ, δ are especially robust to the choice of ϕ . Overall, this experiment empirically demonstrates fairly stable inference for θ , provided that ϕ lies within a range that is appropriate for the data.

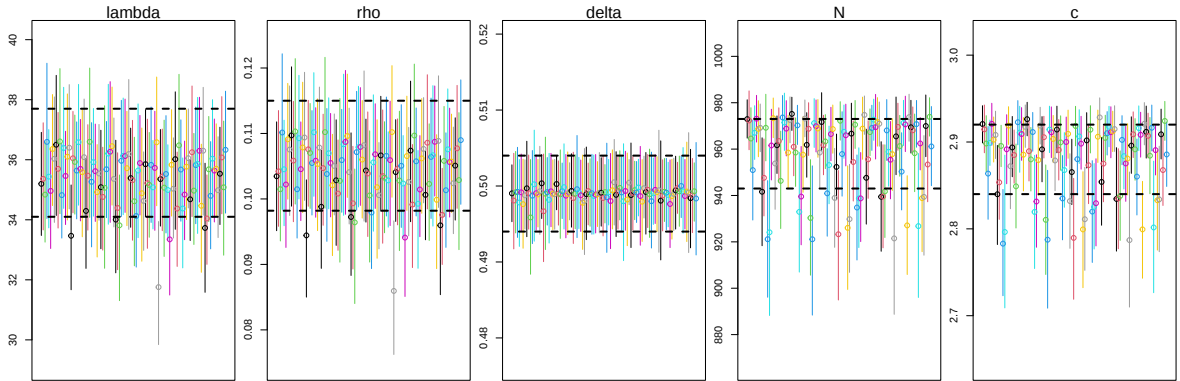


Figure 9: Effect of hyper-parameter values ϕ on the posterior means (points) and 95% credible intervals (vertical bars) for each parameter on the HIV simulated dataset. Each bar represents one random set of ϕ values. The upper and lower limits of the 95% credible intervals inferred using the original values of ϕ are shown via the dashed horizontal lines.

4.3. Hamiltonian Monte Carlo

HMC is an MCMC sampling algorithm that leverages *Hamiltonian dynamics* (see, e.g., [Leimkuhler and Reich 2004](#)) to obtain draws from a target probability distribution. Via its joint consideration of “position” and “momentum” variables, samples generated by HMC can explore the target distribution more effectively than those of random walks ([Neal 2011](#)). In this subsection, we review the key concepts of HMC and its implementation options available in **MAGI**.

The ingredients of HMC are setup as follows. Let $\mathbf{q}$ be a set of “position” variables with negative log-density $U(\mathbf{q})$ (up to an additive constant), so that the target probability density can be written as $\pi(\mathbf{q}) = \frac{1}{Z} \exp(-U(\mathbf{q}))$ with Z being the normalizing constant. Under Hamiltonian dynamics, $U(\mathbf{q})$ is interpreted as the *potential energy* at position $\mathbf{q}$. Further, let $\nabla U(\mathbf{q})$ denote the gradient vector of $U(\mathbf{q})$, with respect to $\mathbf{q}$. In **MAGI**, $\mathbf{q}$ consists of all the variables to be sampled: the trajectories $\mathbf{x}(\mathbf{I})$ and the parameters $\boldsymbol{\theta}$ (which also includes the noise levels $\boldsymbol{\sigma}$ if they are unknown); the function $U(\cdot)$ is the negative log of their joint posterior density, as shown in Equation 3. Next, HMC introduces “momentum” variables $\mathbf{p}$ with the same dimension as $\mathbf{q}$, and we define their *kinetic energy* as $K(\mathbf{p}) = \mathbf{p}^\top \mathbf{p} / 2$. The *Hamiltonian* then combines the kinetic and potential energies, defined as $H(\mathbf{q}, \mathbf{p}) = U(\mathbf{q}) + K(\mathbf{p})$.

With this setup, $\exp[-H(\mathbf{q}, \mathbf{p})]$ specifies a joint density (up to a multiplicative constant) of $\mathbf{q}$ and $\mathbf{p}$, which can be interpreted as follows: (i) $\mathbf{q}$ and $\mathbf{p}$ are independent random variables; (ii) the marginal distribution of $\mathbf{q}$ is the target posterior of interest in Equation 3; (iii) the marginal distribution of $\mathbf{p}$ is multivariate standard normal. Intuitively, HMC works on this joint density so that we obtain the samples of interest for $\mathbf{q}$, and the role of $\mathbf{p}$ is to facilitate the sampling efficiency.

From a current state for $\mathbf{q}$, one iteration of the HMC algorithm generates the next state for $\mathbf{q}$ as follows, where L is a positive integer and $\boldsymbol{\epsilon} > 0$ is a vector of step sizes:

1. Draw $\mathbf{p}$ from a standard multivariate normal distribution, then set $(\mathbf{q}_0, \mathbf{p}_0) = (\mathbf{q}, \mathbf{p})$.
2. For $l = 1, \dots, L$, the *leapfrog method* is used to approximate the Hamiltonian dynamics:
 - (a) Take a half-step for the momentum by setting $\tilde{\mathbf{p}} = \mathbf{p}_{l-1} - (\boldsymbol{\epsilon}/2) \cdot \nabla U(\mathbf{q}_{l-1})$.
 - (b) Take a full-step for the position by setting $\mathbf{q}_l = \mathbf{q}_{l-1} + \boldsymbol{\epsilon} \cdot \tilde{\mathbf{p}}$.⁴
 - (c) Take a half-step for the momentum (using the updated position) by setting $\mathbf{p}_l = \mathbf{p}_{l-1} - (\boldsymbol{\epsilon}/2) \cdot \nabla U(\mathbf{q}_l)$.
3. The proposed state is $(\mathbf{q}^*, \mathbf{p}^*) \equiv (\mathbf{q}_L, \mathbf{p}_L)$. Accept $\mathbf{q}^*$ as the next state of $\mathbf{q}$ with the usual Metropolis acceptance probability, i.e., $\min[1, \exp(-H(\mathbf{q}^*, \mathbf{p}^*) + H(\mathbf{q}, \mathbf{p}))]$. If the proposed state is rejected, the next state of $\mathbf{q}$ is set to be the same as its current state.

Two main aspects of the HMC algorithm can thus be tuned: the step size vector $\boldsymbol{\epsilon}$, and the number of leapfrog steps L . In **MAGI**, these can be supplied by the user in the list of optional inputs to **MagiSolver**: L is specified via `nstepsHmc` and a starting factor for $\boldsymbol{\epsilon}$ is specified via `stepSizeFactor`. Two other options in **MagiSolver** related to MCMC sampling are also worth mentioning: the number of HMC iterations to run is specified via `niterHmc`,

⁴When $\mathbf{q}$ has upper or lower bounds (e.g., parameters that are strictly positive), this step is modified slightly to handle the bounds. For details see p. 149 of [Neal \(2011\)](#).

and the proportion of MCMC samples to discard as an initial burn-in period is specified via `burninRatio`. We discuss each of these and our practical recommendations below.

The step-size vector ϵ controls the accuracy of the Hamiltonian approximation: if ϵ is too large, the HMC proposals will have a high rejection rate; while if ϵ is too small, the HMC proposals will move slowly around the target posterior distribution. Thus ϵ needs to be carefully tuned to achieve good HMC performance. As suggested in Neal (2011), the optimal acceptance rate of HMC is 65%; moreover, the tuning of ϵ can be done independently of L . **MAGI** handles these aspects of ϵ via automatic tuning during the burn-in period (i.e., the first `burninRatio * niterHmc` iterations). Briefly, **MAGI** uses a moving window of 100 iterations to monitor: (i) the acceptance rate, so that ϵ is increased (or decreased) if the acceptance rate is above 90% (or below 60%); (ii) the SD of each variable in $\mathbf{q}$, so that the individual step sizes in ϵ are adapted to follow the scale of each variable.⁵ The optional `stepSizeFactor` input can be a scalar (applied to all the variables in $\mathbf{q}$) or a vector (with the same length as $\mathbf{q}$), that specifies the starting value of ϵ . In our experience, **MAGI**'s automatic tuning can quickly identify the range of ϵ needed for efficient HMC sampling, so that there is usually no need to manually specify `stepSizeFactor`. If the acceptance rate of the first many (e.g., thousands) HMC iterations is 0% or 100%, overriding the default `stepSizeFactor = 0.01` with a value that is several orders of magnitude smaller (if the acceptance rate is 0%) or larger (if the acceptance rate is 100%) could help speed up convergence.

Since automatic tuning of ϵ is limited to the burn-in period, the default `burninRatio = 0.5` usually provides a good balance between samples used for convergence/tuning and samples for final inference. The number of HMC iterations (default `niterHmc = 20000`) can be set to balance computational time constraints with the Monte Carlo variance of the resulting estimates. To monitor sampling progress when running `MagiSolver`, setting `verbose = TRUE` will print a message to the console every 100 HMC iterations.

In **MAGI**, the number of leapfrog steps L is fixed for all HMC iterations. The default value of `nstepsHmc` is 200 and should work well in many cases.⁶ We recommend using traceplots (which can be generated via the convenience `plot()` function on the output of `MagiSolver`) to visually diagnose whether a larger L might be needed. The overall cost per HMC iteration is roughly proportional to L , so L should only be increased if necessary. As the discretization set $\mathbf{I}$ is taken to be increasingly dense (see Section 4.1), the increased dimensionality of $\mathbf{q}$ may lead to ‘sticky’ HMC samples with high autocorrelation. To illustrate, let us revisit the FN dataset and run `MagiSolver` on `y_I3` (which has 321 discretization points per component) with the default $L = 200$:

```
R> data("FNdat")
R> set.seed(12321)
R> y_I0 <- setDiscretization(FNdat, by = 0.5)
R> y_I3 <- setDiscretization(y_I0, level = 3)
R> FNres3b <- MagiSolver(y_I3, fnmodel, control = list(niterHmc = 10000))
```

⁵Neal (2011) recommends randomizing the step size for each HMC iteration to further improve the stability of HMC; so in practice, at each iteration **MAGI** draws a random step size vector uniformly from the range $[\epsilon, 2\epsilon]$.

⁶Since the dimensionality of $\mathbf{q}$ in **MAGI** includes all the discretization points $\mathbf{x}(\mathbf{I})$, which is often fairly large, we use a more conservative default of $L = 200$ compared to the $L = 100$ rule-of-thumb suggested by Neal (2011).

Then, we examine the traceplots of the parameters, shown in Figure 10:

```
R> plot(FNres3b, type = "trace", par.names = FNpar.names, sigma = TRUE)
```

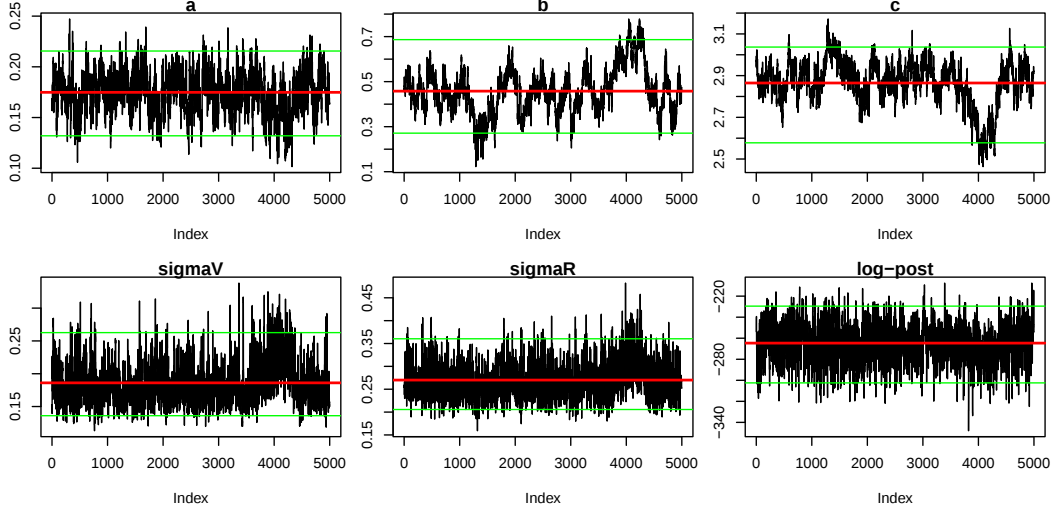


Figure 10: MCMC traceplots that indicate relatively high autocorrelation, based on an example run for the FN system with a dense discretization set and 200 leapfrog steps per HMC iteration. The horizontal lines in the plots indicate the posterior mean (red) and limits of the 95% credible interval (green) for each parameter.

We see that the MCMC samples exhibit some non-stationary patterns, rather than appearing as random scatters for each parameter. This is most evident for parameters b and c , where the Markov chain can remain in one region of their distribution for several hundred iterations at a time (hence ‘sticky’). This issue can usually be alleviated by increasing L , for example by running `MagiSolver` with `nstepsHmc = 1000` as in Section 4.1, then the re-drawn traceplots can be seen to indicate good convergence (not shown for brevity).

5. Benchmark comparisons with other methods

As noted in the Introduction, unobserved system components pose a challenge to most other methods of ODE inference and their software implementations. In this section, we take the Hes1 example where component H is unobserved (as presented in the Introduction and Section 3), and compare the inference accuracy and runtime of different software packages that can handle this problem in R.

Alongside **MAGI**, we consider the **deBInfer** and **CollocInfer** packages (see Section 1.2). The **deBInfer** package represents a Bayesian approach to parameter estimation with the help of numerical ODE solvers, and hence is generally applicable to systems with unobserved components. The **CollocInfer** package is a collocation-based penalized likelihood method that uses a B-spline basis. While **CollocInfer** (along with **pCODE**, which is based on the same underlying methodology) can be used to perform inference with an unobserved component, an estimate of the B-spline basis must be supplied by the user, as we detail below. The other R packages described in Section 1.2 do not have the capacity for unobserved components.

To generate simulated datasets for this comparison, we follow the same procedure described in Section 1.1, with 100 different random seeds. Since **deBInfer** and **CollocInfer** do not directly provide estimates of the inferred trajectories, for fair comparison we use each method to obtain estimates of the parameters $\theta = (a, b, c, d, e, f, g)$ and initial conditions $P(0), M(0), H(0)$. Using these estimated parameters and initial conditions, we run the numerical solver to reconstruct the trajectory implied by the estimates. For a given method, we then compute the RMSE between this reconstructed trajectory and the true trajectory (i.e., the solid curves in Figure 1) for each system component, at the 33 observation time points. We call this the *trajectory RMSE* metric, as in Yang *et al.* (2021).

Thus, on each simulated dataset, the three methods are run as follows to obtain estimates of θ and $P(0), M(0), H(0)$:

- **MAGI** is run as described in Section 3, which infers the parameters and system trajectories. The posterior means of θ and $P(0), M(0), H(0)$ from the **MagiSolver** output are taken as the estimates.
- For **deBInfer**, we set up a normal likelihood for the observations of P and M on the log-scale, with known SD from the simulation setup. The priors for θ were set to be uniform over restricted ranges: a, b, c, d, e are taken to be uniform on $[0, 2]$, f is uniform on $[0, 100]$, and g is uniform on $[0, 10]$. Without imposing these informative priors on θ (i.e., which contrast with the uniform priors over all positive real numbers, as used in **MAGI**), the method would often fail to converge at a reasonable result. The priors for the initial conditions were uniform on the log-scale. We ran 20000 MCMC iterations, to match the number of iterations run in **MAGI**. The posterior means of θ and $P(0), M(0), H(0)$ from the **de_mcmc** output are taken as the estimates, after discarding the first 10000 iterations as burn-in.
- For **CollocInfer**, the method begins by computing initial estimates of the B-spline basis. However, the package does not have the ability to compute such estimates when there are unobserved system components. Therefore, manual input is needed in this case to supply these estimates, which we do as follows. First, we use the package functions to fit the B-spline basis given the *true* values of all the system components at the 33 observation time points. (In real data analyses, such true values would not be available, so **CollocInfer** is given an additional advantage by taking this approach.) Second, we take the B-spline fit for the unobserved component obtained from the first step, together with the actual noisy observations of P and M , and run the main **CollocInfer** method with its default settings to obtain the final estimates of θ and $P(0), M(0), H(0)$.

The full code to benchmark each method is provided in the replication materials.

The results of the three methods over the 100 simulated datasets are summarized in Table 1. For each method, the average runtime and trajectory RMSE of each system component is shown. For the unobserved H component, **MAGI** can reliably recover its trajectory, while **deBInfer** and **CollocInfer** cannot, as indicated by the large RMSEs. For the protein (P) and mRNA (M) components, where observations are available, the results are relatively closer, though **MAGI** still outperforms the other two methods. In terms of runtime, **MAGI** is the fastest of the three methods, averaging 6.2 minutes per dataset. **CollocInfer**, as a frequentist-based optimization method, is also relatively fast; **deBInfer**, which relies on the numerical

solver at each iteration to compute the likelihood, was significantly slower than the other two methods. All of the computations were carried out using a single CPU core of an Intel Xeon 3.7 GHz processor. Overall, these results illustrate the favorable performance of **MAGI** on this inference problem.

Table 1: Performance of the three compared methods over 100 simulated datasets from the Hes1 model. For each method, the mean runtime and trajectory RMSE of each component is reported. The H component is never observed.

	MAGI	deBInfer	CollocInfer
Trajectory RMSE of P	0.95	1.36	1.49
Trajectory RMSE of M	0.20	0.33	0.45
Trajectory RMSE of H	2.48	53.6	225.9
Runtime (minutes)	6.2	46.7	7.4

6. Conclusion and discussion

The inference problem for dynamic systems is a vital task in science and engineering, which widely use ODE models. In practice, the experimental data collected from these systems may often be noisy and sparse. Furthermore, some components of the system may be entirely unobserved. These features pose challenges for estimating the unknown parameters and reconstructing the system trajectories without numerical integration. Existing software packages for this task, to the best of our knowledge, cannot readily handle unobserved components without substantial manual input. This paper presented our package for the **MAGI** method (Yang *et al.* 2021), which capably handles these inference problems in a principled Bayesian framework using manifold-constrained Gaussian processes. The user may choose any of R, MATLAB, and Python to carry out the analyses. Scripts that demonstrate the equivalent functionality in all three environments are included in the replication materials.

We believe the methodological approach of **MAGI** is quite extensible. We discuss some interesting directions for future developments to the software package in the following.

- Extending the inference framework to allow for time-varying parameters. The current implementation of **MAGI** assumes time-constant parameters θ , which covers a broad range of dynamic systems. In some cases, a more flexible time-varying specification $\theta(t)$ is needed, e.g., for pharmacokinetic parameters (Li, Brown, Lee, and Gupta 2002) and disease transmission rates (Keller, Zhou, Kaplan, Anderson, and Zhou 2022). GP priors for $\theta(t)$ could potentially be incorporated into the **MAGI** framework as well to handle this situation.
- Incorporating more flexible choices for the measurement error model and priors for the parameters. The **MAGI** package currently assumes the noise term can be adequately modeled as additive and Gaussian. Multiplicative noise can be handled by taking a log-transformation, as demonstrated in the Hes1 example. If the measurement noise is significantly non-Gaussian, allowing a custom specification for the likelihood $p(\mathbf{y}(\tau)|\mathbf{x}(\mathbf{I}))$ could be a useful feature. Further, the priors for θ are assumed to be flat (uniform) between the user-provided bounds `thetaLowerBound` and `thetaUpperBound`.

While this assumption may often be adequate (as transformations may also be applied to parameters as needed), custom priors $\pi(\boldsymbol{\theta})$ could allow more specific prior knowledge of the parameters to be incorporated into the inference.

- Functions to help set up the ODE gradients with respect to $\boldsymbol{\theta}$ and X . Supplying the analytic gradients $\partial \mathbf{f}(X, \boldsymbol{\theta}, t) / \partial X$ and $\partial \mathbf{f}(X, \boldsymbol{\theta}, t) / \partial \boldsymbol{\theta}$ enables **MAGI** to draw MCMC samples efficiently via its HMC implementation for the target posterior density. The ability to easily generate code for these gradients from the ODEs (e.g., with the help of symbolic differentiation) or have them automatically computed (i.e., automatic differentiation or *autograd*) could potentially reduce the time needed to set up a dynamic system in **MAGI**. In R, support for *autograd* is currently experimental and computationally inefficient compared with supplying the analytic gradients; a proof-of-concept that uses *autograd* with **MAGI** is provided in Appendix E for the interested reader.

References

- Barber D, Wang Y (2014). “Gaussian Processes for Bayesian Estimation in Ordinary Differential Equations.” In *International Conference on Machine Learning*, pp. 1485–1493.
- Boersch-Supan PH, Ryan SJ, Johnson LR (2017). “**deBInfer**: Bayesian Inference for Dynamical Models of Biological Systems in R.” *Methods in Ecology and Evolution*, **8**(4), 511–518.
- Bolouri H (2008). *Computational Modeling of Gene Regulatory Networks – A Primer*. World Scientific Publishing Company.
- Busenberg S (2012). *Differential Equations and Applications in Ecology, Epidemics, and Population Problems*. Elsevier.
- Calderhead B, Girolami M, Lawrence ND (2009). “Accelerating Bayesian Inference Over Non-linear Differential Equations With Gaussian Processes.” In *Advances in Neural Information Processing Systems*, pp. 217–224.
- Dattner I, Yaari R (2020). **simode**: *Statistical Inference for Systems of Ordinary Differential Equations using Separable Integral-Matching*. R package version 1.2.0, URL <https://CRAN.R-project.org/package=simode>.
- Dondelinger F, Husmeier D, Rogers S, Filippone M (2013). “ODE Parameter Inference Using Adaptive Gradient Matching With Gaussian Processes.” In *International Conference on Artificial Intelligence and Statistics*, pp. 216–228.
- Falbel D, Luraschi J (2023). **torch**: *Tensors and Neural Networks with ‘GPU’ Acceleration*. R package version 0.10.0, URL <https://CRAN.R-project.org/package=torch>.
- FitzHugh R (1961). “Impulses and Physiological States in Theoretical Models of Nerve Membrane.” *Biophysical Journal*, **1**(6), 445–466.
- Hirata H, Yoshiura S, Ohtsuka T, Bessho Y, Harada T, *et al.* (2002). “Oscillatory Expression of the bHLH Factor Hes1 Regulated by a Negative Feedback Loop.” *Science*, **298**(5594), 840–843.

- Hooker G, Ramsay JO, Xiao L (2016). “**CollocInfer**: Collocation Inference in Differential Equation Models.” *Journal of Statistical Software*, **75**(2), 1–52.
- Keller JP, Zhou T, Kaplan A, Anderson GB, Zhou W (2022). “Tracking the Transmission Dynamics of COVID-19 with a Time-varying Coefficient State-space Model.” *Statistics in Medicine*, **41**(15), 2745–2767.
- Leimkuhler B, Reich S (2004). *Simulating Hamiltonian Dynamics*. Cambridge University Press.
- Li L, Brown MB, Lee KH, Gupta S (2002). “Estimation and Inference for a Spline-Enhanced Population Pharmacokinetic Model.” *Biometrics*, **58**(3), 601–611.
- Liang H, Miao H, Wu H (2010). “Estimation of Constant and Time-Varying Dynamic Parameters of HIV Infection in a Nonlinear Differential Equation Model.” *The Annals of Applied Statistics*, **4**(1), 460–483.
- Ljung L (1995). *System Identification Toolbox: User’s Guide*. The MathWorks Inc., Natick, MA.
- Macdonald B, Dondelinger F (2020). **deGradInfer**: *Parameter Inference for Systems of Differential Equation*. R package version 1.0.1, URL <https://CRAN.R-project.org/package=deGradInfer>.
- Neal RM (2011). “MCMC Using Hamiltonian Dynamics.” In S Brooks, A Gelman, G Jones, X Meng (eds.), *Handbook of Markov Chain Monte Carlo*, Chapman & Hall/CRC Handbooks of Modern Statistical Methods, chapter 5, pp. 113–162. CRC Press. ISBN 9781420079425.
- Niu M, Rogers S, Filippone M, Husmeier D (2016). “Fast Parameter Inference in Nonlinear Dynamical Systems Using Iterative Gradient Matching.” In *International Conference on Machine Learning*, pp. 1699–1707.
- Niu M, Wandy J, Daly R, Rogers S, Husmeier D (2021). “R Package for Statistical Inference in Dynamical Systems Using Kernel Based Gradient Matching: **KGode**.” *Computational Statistics*, **36**(1), 715–747.
- Ramsay JO, Hooker G, Campbell D, Cao J (2007). “Parameter Estimation for Differential Equations: A Generalized Smoothing Approach.” *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, **69**(5), 741–796.
- Raue A, Steiert B, Schelker M, Kreutz C, Maiwald T, Hass H, Vanlier J, Tönsing C, Adlung L, Engesser R, *et al.* (2015). “**Data2Dynamics**: A Modeling Environment Tailored to Parameter Estimation in Dynamical Systems.” *Bioinformatics*, **31**(21), 3558–3560.
- R Core Team (2022). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL <https://www.R-project.org/>.
- Soetaert K, Petzoldt T, Setzer RW, Brown PN, Byrne GD, *et al.* (2020). **deSolve**: *Solvers for Initial Value Problems of Differential Equations*. R package version 1.28, URL <https://CRAN.R-project.org/package=deSolve>.

- Titsias MK, Rattray M, Lawrence ND (2011). “Markov chain Monte Carlo algorithms for Gaussian processes.” In D Barber, AT Cemgil, S Chiappa (eds.), *Bayesian Time Series Models*, chapter 14, pp. 295–315. Cambridge University Press. ISBN 9780521196765.
- Tu PN (2012). *Dynamical Systems: An Introduction With Applications in Economics and Biology*. Springer-Verlag Science & Business Media.
- Walas SM (1991). *Modeling With Differential Equations in Chemical Engineering*. Boston: Butterworth-Heinemann.
- Wang H, Cao J (2019). **pCODE**: Estimation of an Ordinary Differential Equation Model by Parameter Cascade Method. R package version 0.9.3, URL <https://CRAN.R-project.org/package=pCODE>.
- Wenk P, Gotovos A, Bauer S, Gorbach NS, Krause A, Buhmann JM (2019). “Fast Gaussian Process Based Gradient Matching for Parameter Identification in Systems of Nonlinear ODEs.” In *International Conference on Artificial Intelligence and Statistics*, pp. 1351–1360.
- Williams CK, Rasmussen CE (2006). *Gaussian Processes for Machine Learning*. MIT Press, Cambridge, MA.
- Wong SW, Yang S, Kou S (2023). “Estimating and Assessing Differential Equation Models with Time-Course Data.” *The Journal of Physical Chemistry B*, **127**(11), 2362–2374.
- Yang S, Wong SW (2023). **magi**: MANifold-Constrained Gaussian Process Inference. R package version 1.2.2, URL <https://CRAN.R-project.org/package=magi>.
- Yang S, Wong SW, Kou S (2021). “Inference of Dynamic Systems From Noisy and Sparse Data via Manifold-Constrained Gaussian Processes.” *Proceedings of the National Academy of Sciences*, **118**(15), e2020397118.

A. MAGI usage in MATLAB

To use **MAGI** in MATLAB, first clone the repository at <https://github.com/wongswk/magi>, e.g., by executing on the command-line `git clone https://github.com/wongswk/magi`. On Linux-compatible systems, build the core C++ **MAGI** library by running the `build.sh` shell script, then run the `MATLAB_build.sh` shell script in the `MATLABmagi` directory to generate the compiled MEX files. For Windows systems, pre-compiled versions of the **MAGI** library and MEX files are provided in the `MATLABmagi/windows` directory. Ensure that `libcmagi.so` (or `libcmagi.dll` in Windows), along with the accompanying `.m` routines and compiled MEX files of the package, are located in either the working directory or the path of MATLAB.

The replication package (supplementary .zip file provided with this article) includes the MATLAB directory which contains the materials for running the three examples discussed in this paper in MATLAB. Specifically:

- The `models/` subdirectory contains the functions for the ODE systems, since the MATLAB convention is one function per file. For example, for the FitzHugh-Nagumo (FN) equations,
 - `fnmodelODE.m` codes the ODEs,
 - `fnmodelDx.m` codes their gradients with respect to X
 - `fnmodelDtheta.m` codes their gradients with respect to θ
 - `fnmodelODEsolve.m` codes the system in a form suitable for invoking ODE solvers such as `ode45`.
- `replication.m` is the MATLAB replication script that carries out the same analyses described in the main paper. Please note that the random seeds between R and MATLAB are not interchangeable so their corresponding numerical results are expected to have slight differences attributable to random-number generation. Otherwise, the functionalities of the R and MATLAB replication scripts are identical, so that users can easily follow the equivalent syntax in MATLAB for the examples given in this paper.

We briefly point out two main differences of note between the syntax in R and MATLAB:

- In MATLAB, `struct` arrays are used in place of `list` objects in R. Taking the FN equations as an example, to set up `odeModel` for input to `MagiSolver` we create a `struct` with the five required elements, where `f0de`, `f0deDx`, and `f0deDtheta` are assigned their corresponding function handles:

```
>> fnmodel.f0de = @fnmodelODE;
>> fnmodel.f0deDx = @fnmodelDx;
>> fnmodel.f0deDtheta = @fnmodelDtheta;
>> fnmodel.thetaLowerBound = [0 0 0];
>> fnmodel.thetaUpperBound = [Inf Inf Inf];
```

Similarly, the `control` list is set up by creating a `struct` in MATLAB, e.g.,

```
>> config.niterHmc = 10000;
```

to run 10000 HMC iterations, and then `config` can be passed as the `control` argument to `MagiSolver`.

The output of `MagiSolver` is also a `struct`; e.g., if `FNres0` contains the output of a `MagiSolver` run, the matrix of MCMC samples for θ would be accessed as `FNres0.theta`. The convenience functions `summaryMagiOutput()` and `plotMagiOutput()` are provided (analogous to `summary()` and `plot()` in R) to generate a table of parameter estimates (with credible intervals) and visualize the inferred trajectories from the output of `MagiSolver`.

- In MATLAB, optional function arguments need to be explicitly skipped by passing `[]`. For example, `MagiSolver` allows the time vector to be passed as a separate argument from the data matrix `y`. When the time vector is included as the first column in `y`, we would skip the third argument as follows:

```
>> FNres0 = MagiSolver(y, fnmodel, [], config);
```

B. MAGI usage in Python

To use **MAGI** in Python, first clone the repository at <https://github.com/wongswk/magi>, e.g., by executing on the command-line `git clone https://github.com/wongswk/magi`. Build the core C++ **MAGI** library by running the `build.sh` shell script, then run the `py_build.sh` shell script in the `pymagi` directory to build the `pymagi.so` Python library. Ensure that this `pymagi` directory is contained in Python's path.

The replication package (supplementary .zip file provided with this article) includes the `python` directory which contains the Python script `replication.py` that carries out the same analyses for the three examples discussed in the main paper. Please note that the random seeds between R and Python are not interchangeable so their corresponding numerical results are expected to have slight differences attributable to random-number generation. Otherwise, the functionalities of the R and Python replication scripts are identical, so that users can easily follow the equivalent syntax in Python.

We briefly point out two main differences of note between the syntax in R and Python:

- In Python, we construct the `odeModel` input to `MagiSolver` by using the helper function `ode_system`, rather than setting up an R list. Taking the FN equations as an example, suppose `fnmodelOde`, `fnmodelDx`, and `fnmodelDtheta` respectively are the functions for the ODEs, gradients with respect to X , and gradients with respect to θ . Then we can call `ode_system` as follows:

```
>> fn_system = ode_system("FN-python",
                           fnmodelOde, fnmodelDx, fnmodelDtheta,
                           thetaLowerBound = np.array([0, 0, 0]),
                           thetaUpperBound = np.array([np.inf, np.inf, np.inf]))
```

where the first argument can be any string that provides a name for the system.

- In Python, the dictionary data type (`dict`) is used in place of the R list for the `control` argument to `MagiSolver`. For example, with `fn_system` as defined above, we can provide the `control` argument in the call to `MagiSolver` as follows:

```
>> FNres3 = MagiSolver(y = y_I3, odeModel = fn_system,
                      control = dict(niterHmc = 10000, nstepsHmc = 1000))
```

The output of `MagiSolver` is also a dict; e.g., if `FNres0` contains the output of a `MagiSolver` run, the matrix of MCMC samples for θ would be accessed as

```
>> FNres0['theta']
```

The convenience functions `summaryMagiOutput()` and `plotMagiOutput()` are provided (analogous to `summary()` and `plot()` in R) to generate a table of parameter estimates (with credible intervals) and visualize the inferred trajectories from the output of `MagiSolver`.

C. Functions for Fitzhugh-Nagumo ODEs and their gradients

The following R functions encode the ODEs and gradients for the FN equations discussed in Section 4.1.

```
R> fnmodelODE <- function(theta, x, tvec) {
+   V <- x[, 1]
+   R <- x[, 2]
+
+   result <- array(0, c(nrow(x), ncol(x)))
+   result[, 1] = theta[3] * (V - V^3 / 3.0 + R)
+   result[, 2] = -1.0/theta[3] * (V - theta[1] + theta[2] * R)
+
+   result
+ }
```

```
R> fnmodelDx <- function(theta, x, tvec) {
+   resultDx <- array(0, c(nrow(x), ncol(x), ncol(x)))
+   V = x[, 1]
+
+   resultDx[, 1, 1] = theta[3] * (1 - V^2)
+   resultDx[, 2, 1] = theta[3]
+
+   resultDx[, 1, 2] = -1.0 / theta[3]
+   resultDx[, 2, 2] = -theta[2] / theta[3]
+
+   resultDx
+ }
```

```
R> fnmodelDtheta <- function(theta, x, tvec) {
+   resultDtheta <- array(0, c(nrow(x), length(theta), ncol(x)))
+
+   V = x[, 1]
```

```

+   R = x[, 2]
+
+   resultDtheta[, 3, 1] = V - V^3 / 3.0 + R
+
+   resultDtheta[, 1, 2] = 1.0 / theta[3]
+   resultDtheta[, 2, 2] = -R / theta[3]
+   resultDtheta[, 3, 2] = 1.0 / (theta[3]^2) * (V - theta[1] + theta[2] * R)
+
+   resultDtheta
+ }

```

D. Functions for HIV model ODEs and their gradients

The following R functions encode the ODEs and gradients for the HIV model discussed in Section 4.2.

```

R> hivtdmodelODE <- function(theta, x, tvec) {
+   TU <- x[, 1]
+   TI <- x[, 2]
+   V <- x[, 3]
+
+   lambda <- theta[1]
+   rho <- theta[2]
+   delta <- theta[3]
+   N <- theta[4]
+   c <- theta[5]
+
+   eta <- 9e-5 * (1 - 0.9 * cos(pi * tvec / 1000))
+
+   result <- array(0, c(nrow(x), ncol(x)))
+   result[, 1] = lambda - rho * TU - eta * TU * V
+   result[, 2] = eta * TU * V - delta * TI
+   result[, 3] = N * delta * TI - c * V
+
+   result
+ }

R> hivtdmodelDx <- function(theta, x, tvec) {
+   resultDx <- array(0, c(nrow(x), ncol(x), ncol(x)))
+
+   TU <- x[, 1]
+   TI <- x[, 2]
+   V <- x[, 3]
+
+   lambda <- theta[1]

```

```

+   rho <- theta[2]
+   delta <- theta[3]
+   N <- theta[4]
+   c <- theta[5]
+
+   eta <- 9e-5 * (1 - 0.9 * cos(pi * tvec / 1000))
+
+   resultDx[, , 1] = cbind(-rho - eta * V, 0, -eta * TU)
+   resultDx[, , 2] = cbind(eta * V, -delta, eta * TU)
+   resultDx[, , 3] = cbind(rep(0, nrow(x)), N * delta, -c)
+
+   resultDx
+ }

R> hivtdmodelDtheta <- function(theta, x, tvec) {
+   resultDtheta <- array(0, c(nrow(x), length(theta), ncol(x)))
+
+   TU <- x[, 1]
+   TI <- x[, 2]
+   V <- x[, 3]
+
+   delta <- theta[3]
+   N <- theta[4]
+
+   resultDtheta[, , 1] = cbind(1, -TU, 0, 0, 0)
+   resultDtheta[, , 2] = cbind(0, 0, -TI, 0, 0)
+   resultDtheta[, , 3] = cbind(0, 0, N * TI, delta * TI, -V)
+
+   resultDtheta
+ }

```

E. Combining automatic differentiation with MAGI

We can utilize automatic differentiation (autograd) with **MAGI** by leveraging the autograd functionality provided by the **torch** package (Falbel and Luraschi 2023). First, we install the **torch** package in R by executing the command `install.packages("torch")`.

To integrate autograd with **MAGI**, it is necessary to rewrite the ODE function using torch tensors instead of R arrays. For the Hes1 example presented in Section 3, a single line of code needs to be modified: the R array `PMHdt = array(0, c(nrow(x), ncol(x)))` is replaced with the torch tensor `PMHdt = torch_empty(dim(x))`. An adapted version of `hes1logmodelODE` suitable for autograd is as follows:

```

R> hes1logmodelODE_torch <- function (theta, x, tvec) {
+   P = exp(x[, 1])
+   M = exp(x[, 2])

```

```

+   H = exp(x[, 3])
+
+   PMHdt <- torch_empty(dim(x))
+   PMHdt[, 1] = -theta[1] * H + theta[2] * M / P - theta[3]
+   PMHdt[, 2] = -theta[4] + theta[5] / (1 + P^2) / M
+   PMHdt[, 3] = -theta[1] * P + theta[6] / (1 + P^2) / H - theta[7]
+
+   PMHdt
+ }

```

With this new implementation of the `hes1logmodelODE_torch` function, the following function can be used to calculate the derivatives with respect to both X and θ :

```

R> ode_autograd <- function(ode_func_torch, theta, x, tvec) {
+   # Convert input arguments to torch tensors with requires_grad = TRUE
+   theta <- torch_tensor(theta, requires_grad = TRUE)
+   x <- torch_tensor(x, requires_grad = TRUE)
+   tvec <- torch_tensor(tvec)
+
+   # Calculate output using torch operations
+   output = ode_func_torch(theta, x)
+
+   # Initialize gradient matrices
+   ode_dtheta = array(dim=c(nrow(output), length(theta), ncol(output)))
+   ode_dx = array(dim=c(nrow(output), ncol(output), ncol(output)))
+
+   # Calculate gradients for each element in the output
+   for (i in 1:nrow(output)) {
+     for (j in 1:ncol(output)) {
+       # Zero out gradients from previous iterations
+       if (length(theta$grad) > 0) {
+         theta$grad$zero_()
+       }
+       if (length(x$grad) > 0) {
+         x$grad$zero_()
+       }
+
+       # use keep_graph=TRUE or retain_graph=TRUE depending on torch version
+       output[i, j]$backward(retain_graph=TRUE)
+
+       ode_dtheta[i, , j] = as_array(theta$grad)
+       ode_dx[i, , j] = as_array(x$grad[i,])
+     }
+   }
+
+   list(ode_dtheta = ode_dtheta, ode_dx = ode_dx)
+ }

```

The correctness of the derivative calculations can be confirmed by comparing the output of `hes1logmodelDtheta(theta, x, tvec)` or `hes1logmodelDx(theta, x, tvec)` with the output of `ode_autograd(hes1logmodelODE_torch, theta, x, tvec)` for any given `theta`, `x`, and `tvec`. It is important to note, however, that the computation speed of the autograd version `ode_autograd(hes1logmodelODE_torch, theta, x, tvec)` is significantly slower than that of the hand-coded derivatives `hes1logmodelDtheta(theta, x, tvec)` or `hes1logmodelDx(theta, x, tvec)`.

To utilize **MAGI** with autograd, we can now proceed to define the `odeModel` list containing the three ODE model functions and the parameter bounds:

```
R> hes1logmodel <- list(
+   fOde = hes1logmodelODE,
+   fOdeDx = function(theta, x, tvec)
+     ode_autograd(hes1logmodelODE_torch, theta, x, tvec)$ode_dx,
+   fOdeDtheta = function(theta, x, tvec)
+     ode_autograd(hes1logmodelODE_torch, theta, x, tvec)$ode_dtheta,
+   thetaLowerBound = rep(0, 7),
+   thetaUpperBound = rep(Inf, 7)
+ )
```

Note that the original R array implementation `hes1logmodelODE` must still be passed to the `MagiSolver` function, as **MAGI** does not currently support direct use of torch tensors. A complete R script that demonstrates this approach of using **MAGI** with autograd is provided in the replication package.

Although autograd offers a convenient method for calculating derivative information, its computational speed is slower than hand-coded analytical gradients. For optimal performance, we recommend using hand-coded analytical gradients, as discussed in the main text of this paper.

F. Other covariance functions available in MAGI

As discussed in Section 4.2, the default and recommended GP covariance function for use in **MAGI** is the Matern (Equation 6) with $\nu = 2.01$. Several other covariance kernels are also available in the package, which include some of the common choices discussed in Chapter 4 of [Williams and Rasmussen \(2006\)](#). Their specification and features are presented below. In each case, r is the absolute difference between two time points and ϕ are the hyper-parameters for the kernel; larger values of ϕ_1 favor curves with higher variance, and larger values of ϕ_2 favor curves with more time-dependence between nearby time points. They may be selected for use in `gpsmoothing` and `MagiSolver` by specifying their corresponding `kerneltype` string.

- Radial basis function or squared exponential (`kerneltype = "rbf"`):

$$\mathcal{K}(r) = \phi_1 \exp\left(-\frac{r^2}{2\phi_2^2}\right)$$

This is an infinitely differentiable kernel, and hence is associated with very smooth GPs. It may be too smooth to adequately model many physical processes (p. 83, [Williams and Rasmussen 2006](#)).

- Matern with $\nu = 5/2$ (`kerneltype = "matern"`):

$$\mathcal{K}(r) = \phi_1 \left(1 + \frac{\sqrt{5}r}{\phi_2} + \frac{5r^2}{3\phi_2^2} \right) \exp \left(-\frac{\sqrt{5}r}{\phi_2} \right) \quad (7)$$

Equation 7 is a simplification of Equation 6 in the special case $\nu = 5/2$. It is faster to compute than $\nu = 2.01$ but has a stronger smoothness assumption, which limits its applicability to systems that are known to have smooth curves.

- Compact kernel (`kerneltype = "compact1"`):

$$\mathcal{K}(r) = \phi_1 \left[\max \left(1 - \frac{r}{\phi_2}, 0 \right) \right]^4 \left(\frac{4r}{\phi_2} + 1 \right)$$

This is a kernel with compact support, i.e., the covariance decays to zero for $r \geq \phi_2$, so that points more than ϕ_2 apart are *a priori* independent. Its polynomial construction also tends to favor smooth curves.

- Periodic Matern (`kerneltype = "periodicMatern"`):

This follows a “time warping” idea to create a non-stationary kernel that could be appropriate for systems that are known to be exactly periodic. Define the “time warping” transformation $r' = |2 \sin(r\pi/\phi_3)|$, where ϕ_3 is the periodicity parameter. Then the covariance is given by $\mathcal{K}(r')$ using Equation 7.

Affiliation:

Samuel W.K. Wong
 Department of Statistics and Actuarial Science
 University of Waterloo
 200 University Ave W
 Waterloo, ON N2L 3G1, Canada
 E-mail: samuel.wong@uwaterloo.ca

Shihao Yang
 H. Milton Stewart School of Industrial and Systems Engineering
 Georgia Institute of Technology
 755 Ferst Drive NW
 Atlanta, GA 30332, USA
 E-mail: shihao.yang@isye.gatech.edu

S.C. Kou
Department of Statistics
Harvard University
1 Oxford St, 7th floor
Cambridge, MA 02138, USA
E-mail: kou@stat.harvard.edu