Scalable SDE Filtering and Inference with Apache Spark

Harish S. Bhat HBHAT@UCMERCED.EDU
R. W. M. A. Madushani RMADUSHANI@UCMERCED.EDU
Shagun Rawat SRAWAT2@UCMERCED.EDU
Applied Mathematics Unit, School of Natural Sciences, University of California, Merced, 5200 N. Lake Rd., Merced, CA, 95343, USA

Abstract

In this paper, we consider the problem of Bayesian filtering and inference for time series data modeled as noisy, discrete-time observations of a stochastic differential equation (SDE) model. We develop a Metropolis algorithm to sample from the joint posterior density of the parameters and the SDE state; the dimension of this posterior equals the number of parameters plus the length of the time series data. The efficiency of our approach relies on an innovative density tracking by quadrature (DTQ) method to compute the likelihood of the SDE, the part of the posterior that requires the most computational effort to evaluate. As we show, the DTQ method lends itself to a natural implementation using Scala and Apache Spark, an open source framework for scalable data mining. Through several tests, we study the performance of our algorithm on filtering and inference problems for both regularly and irregularly spaced time series. We also study the algorithm's scalability, both with regards to the length of the time series and the number of processors.

Keywords: Bayesian filtering, Bayesian inference, stochastic differential equations, Apache Spark, Scala, Markov Chain Monte Carlo, Metropolis algorithm

1. Introduction

In this paper, we focus on the problem of Bayesian inference and filtering for time series. The time series consist of noisy observations of a process that satisfies a stochastic differential equation (SDE). The drift and diffusion terms of this SDE contain parameters. Our goal is to use the observations to infer both the time series of the SDE's actual state (the filtering problem) and the parameters in the drift/diffusion terms (the inference problem). While there are existing packages (e.g., for use with the R language for statistical computing) for solving this problem, we believe the method and implementation described here is the first to make use of Apache Spark, an open source framework for scalable data processing and machine learning on computing clusters (http://spark.apache.org).

To solve the Bayesian inference and filtering problem, we develop a Metropolis algorithm to sample from the joint posterior density of the state series and the SDE parameters. The main bottleneck in this Metropolis algorithm is the evaluation of the posterior density at proposed values of the parameters and state series. The posterior consists of priors multiplied by two likelihood functions: the density of the observation series given the state series, and the density of the state series given the parameters. This latter conditional density, the likelihood function for the SDE, is clearly the most computationally expensive term to evaluate in the overall posterior.

The exact likelihood function for the SDE can only be computed in very special cases, i.e., when we can solve analytically for the SDE's transition density. Therefore, prior work has focused on approximating the exact likelihood, either through analytical methods, numerical methods, or a combination of the two. For a thorough review of past work on this problem, we refer the reader to Sørensen (2004); Iacus (2009); Fuchs (2013).

Consider the computation of the transition density $p_{X_{t_{j+1}}}(x_{j+1}|X_{t_j} = x_j, \theta)$. Here X_t stands for the state of a process that evolves forward in time via an SDE with parameter vector θ. The data consists of the state at two times, t_j and t_{j+1}; we label these states, respectively, as x_j and x_{j+1}.

Let $p(x, t)$ denote the density function of X_t. Then one approach to approximating the transition density is to numerically solve the forward Kolmogorov (or Fokker-Planck) equation with the initial condition $p(x, t_j) = \delta(x - x_j)$ up to time t_{j+1}. Then $p(x_{j+1}, T)$ will be a numerical approximation of the transition density. The Kolmogorov equation is a linear partial differential equation (PDE) with spatially-dependent coefficients.

Our approach is similar in that we also numerically track the density $p(x, t)$ without sampling. Instead of numerically solving a PDE, we track the density by applying quadrature to the Chapman-Kolmogorov equation associated with a time-discretization of the SDE (1a). We detail this density tracking by quadrature (DTQ) method in Section 2. As we show, the DTQ method enables one to break the computation of the likelihood into a sum of likelihoods involving consecutive pairs of observations (t_j, x_j) and (t_{j+1}, x_{j+1}) just as described above. For each such pair, the DTQ method computes the likelihood using iterated matrix multiplication. In Section 3, we show how these features of the DTQ method enable it to naturally take advantage of the efficiency and parallelism of Apache Spark and Scala.

In Section 4, we describe experimental tests of our algorithm using both regularly and irregularly spaced time series. While there is further room for improvement, especially with regards to the classical Metropolis algorithm that we use here, the tests show that our current code does solve the Bayesian inference and filtering problem at a baseline acceptable level. The tests also establish the scalability of the algorithm, both as a function of the number of Spark processors and as a function of the length of the time series.

Other methods similar to ours are those of (Pedersen, 1995) and (Santa-Clara, 1997). In these methods, one also starts with the Chapman-Kolmogorov equation for the Euler-Maruyama scheme applied to (1a). However, instead of evaluating the resulting integrals by deterministic quadrature, Pedersen and Santa-Clara evaluate the integrals by Monte Carlo methods. These methods involve generating numerical sample paths of the SDE at times in between the observation times. Unless one generates sample paths conditional on both $X_{t_j} = x_j$ and $X_{t_{j+1}} = x_{j+1}$, this approach is problematic.

The work of (Aït-Sahalia, 2002) shares our goal of computing an accurate approximation of the exact transition density and resulting likelihood function. Instead of applying quadrature, Aït-Sahalia expands the transition density in a Gram-Charlier series and then computes the expansion coefficients up to a certain order.

Other approaches that have been explored in the literature include likelihood-free methods such as Approximate Bayesian Computation (Picchini, 2014), variational methods (Archambeau et al., 2007b; Vrettas et al., 2015), and/or Gaussian processes (Archambeau et al., 2007a; Ruttor et al., 2013). We reserve for future work a detailed comparison of these methods to our method.

2. Statistical Method

The fundamental model considered in this paper is

$$dX_t = f(X_t;\theta)dt + g(X_t;\theta)dW_t \tag{1a}$$

$$Y_t = X_t + \epsilon_t \tag{1b}$$

The first part of the above system (1a) is a stochastic differential equation (SDE) driven by Brownian motion W_t. The second part (1b) models the observation process Y_t by the addition of noise ϵ_t to the state process X_t. In this work, we assume that ϵ_t is i.i.d. (independent and identically distributed) Gaussian with mean 0 and variance σ_ϵ^2.

2.1. Inference Problem

Suppose that we have data of the form $\mathbf{y} = (y_0, \ldots, y_L)$ where $y_j = Y_{t_j}$, the observation at time t_j. Here $t_0 < t_1 < \cdots < t_L$ is a sequence of times, not necessarily equispaced, at which we collect observations. Using $\mathbf{y}$, we seek to infer the following objects:

- the discrete-time path taken by the state process, $\mathbf{x} = (x_0, \ldots, x_L)$. Here $x_j = X_{t_j}$, the state of the SDE at time t_j.
- the parameter vector $\theta \in \mathbb{R}^N$, and
- the variance σ_ϵ^2 of the noise term ϵ_t.

We view this problem as a Bayesian inference problem, and our goal is to sample from the posterior

$$p(\mathbf{x}, \theta, \sigma_\epsilon^2 \,|\, \mathbf{y}) \propto p(\mathbf{y} \,|\, \mathbf{x}, \theta, \sigma_\epsilon^2) p(\mathbf{x}, \theta, \sigma_\epsilon^2) \tag{2}$$

In the above expression, the left-hand side is the conditional density of the random variables $X_{t_0}, X_{t_1}, \ldots, X_{t_L}, \theta, \sigma_\epsilon^2$ given the random variables $Y_{t_0}, Y_{t_1}, \ldots, Y_{t_L}$. To save space and make our equations more readable, we will omit these random variables in what follows.

It is clear from (1b) that $\mathbf{y}$ is conditionally independent of θ given $\mathbf{x}$ and σ_ϵ^2. It is also clear from (1b) that the observation/state pair at time t_j is independent of all other observation/state pairs. Hence we can write

$$p(\mathbf{y} \,|\, \mathbf{x}, \theta, \sigma_\epsilon^2) = \prod_{j=0}^{L} p(y_{t_j} \,|\, x_j, \sigma_\epsilon^2). \tag{3}$$

Next, we examine the second term on the right-hand side of (2). It is clear that σ_ϵ^2 is independent of the other random variables, so we can write:

$$p(\mathbf{x}, \theta, \sigma_\epsilon^2) = p(\mathbf{x}, \theta) p(\sigma_\epsilon^2) = p(\mathbf{x}|\theta) p(\theta) p(\sigma_\epsilon^2). \tag{4}$$

Putting it all together, we have the following expression for the posterior:

$$p(\mathbf{x}, \theta, \sigma_\epsilon^2 \,|\, \mathbf{y}) \propto \left[\prod_{j=0}^{L} p(y_{t_j} \,|\, x_j, \sigma_\epsilon^2) \right] p(\mathbf{x}|\theta) p(\theta) p(\sigma_\epsilon^2) \tag{5}$$

From (1b), we have that $y_{t_j} \,|\, x_j, \sigma_\epsilon^2$ is Gaussian with mean x_j and variance σ_ϵ^2. The terms $p(\theta)$ and $p(\sigma_\epsilon^2)$ are priors. The only other term, $p(\mathbf{x}|\theta)$, is the likelihood of θ under the model (1a). We describe the computation of this likelihood next.

2.2. Likelihood Computation via Density Tracking by Quadrature

Here we describe how to compute the likelihood $p(\mathbf{x}|\theta)$ under the model (1a). Our first step is to apply a Markov property satisfied by (1a): the random variable $X_{t_{j+1}}$, given X_{t_j}, is conditionally independent of all random variables $X_{t_{j-k}}$ for $k \geq 1$. With this property, the likelihood factors:

$$p(\mathbf{x}|\theta) = p(x_0) \prod_{j=0}^{L-1} p(x_{j+1} \,|\, x_j, \theta). \tag{6}$$

Each term in the product can be interpreted as follows: we start the SDE (1a) with the initial condition $X_{t_j} = x_j$ and fixed parameter vector θ. We then solve for the probability density function (pdf) of $X_{t_{j+1}}$, and evaluate that pdf at x_{j+1}. By following these steps, we have calculated $p(x_{j+1} \,|\, x_j, \theta)$.

We now outline a convergent method to compute the aforementioned pdf (Bhat and Madushani, 2016). Because this method computes an approximation to the density via iterated quadrature, we refer to the method as DTQ (density tracking by quadrature). The first step of the method consists of discretizing (1a) via the Euler-Maruyama discretization. When describing this discretization, we specialize to the case where we seek $p(x_{j+1} \,|\, x_j, \theta)$. That is, we take $\{\tau_i\}_{i=0}^n$ to be a temporal grid such that $\tau_0 = t_j$, $\tau_n = t_{j+1}$, and $h = (t_{j+1} - t_j)/n > 0$. Then, for $0 = 1, 2, \ldots, n$, we have $\tau_i = t_j + ih$. On this temporal grid, the Euler-Maruyama discretization of (1a) is:

$$\tilde{x}_{i+1} = \tilde{x}_i + f(\tilde{x}_i; \theta)h + g(\tilde{x}_i; \theta)h^{1/2} Z_{i+1} \tag{7}$$

where $\{Z_i\}_{i=1}^n$ is an i.i.d. family of Gaussian random variables, each with mean 0 and variance 1. For $i \geq 1$, we think of $\tilde{x}_i$ as a numerical approximation to X_{τ_i}. Note that $\tilde{x}_0 = X_{\tau_0} = X_{t_j}$ which is constrained to equal the data point x_j in this calculation.

The next step in deriving the DTQ method is to write down the Chapman-Kolmogorov equation corresponding to (7). Let $p(\tilde{x}_i)$ denote the pdf of $\tilde{x}_i$ given the initial condition $\tilde{x}_0 = x_j$. Then purely based on the laws of probability we can write:

$$p(\tilde{x}_{i+1}) = \int_{\tilde{x}_i} p(\tilde{x}_{i+1} \,|\, \tilde{x}_i) p(\tilde{x}_i) \, d\tilde{x}_i. \tag{8}$$

Inspecting (7), we see that conditional on $\tilde{x}_i$, the pdf of $\tilde{x}_{i+1}$ is Gaussian with mean $\tilde{x}_i + f(\tilde{x}_i; \theta)h$ and variance $g^2(\tilde{x}_i; \theta)h$. Let us define the function

$$G(a, b) = \frac{1}{\sqrt{2\pi g^2(b; \theta)h}} \exp\left(-\frac{(a - b - f(b; \theta)h)^2}{2g^2(b; \theta)h} \right). \tag{9}$$

Then (8) becomes

$$p(\tilde{x}_{i+1}) = \int_{\tilde{x}_i} G(x_{i+1}, x_i) p(\tilde{x}_i) \, d\tilde{x}_i. \tag{10}$$

The last step is to spatially discretize the pdf's and the integration over $\tilde{x}_i$. Let $k > 0$ be constant; then $z_j = jk$ with $j \in \mathbb{Z}$ is an equispaced grid with spacing k. We represent the

function $p(\tilde{x}_i)$ by a vector $\mathbf{p}_i$ such that the j-th component of $\mathbf{p}_i$ is $p_i^j = p(\tilde{x}_i = z_j)$. We then apply the trapezoidal rule to (10), resulting in

$$p_{i+1}^{j'} = \sum_j kG(z_{j'}, z_j)p_i^j. \tag{11}$$

We now truncate the spatial domain. Let $M > 0$ be an integer. We take both $j, j' \in \{-M, \ldots, 0, \ldots, M\}$; this means that each vector $\mathbf{p}_i$ has dimension $2M + 1$.

Now that we have a method to compute the required pdf's, here is how we compute $p(x_{j+1} \,|\, x_j, \theta)$:

1. Because $\tilde{x}_0 = x_j$, we have that $p(\tilde{x}_0) = \delta(\tilde{x}_0 - x_j)$. Inserting this in (10) with $i = 0$, we obtain
$$p(\tilde{x}_1) = G(\tilde{x}_1, x_j). \tag{12}$$
Evaluating the right-hand side with $\tilde{x}_1$ equal to each point in the spatial grid $\{z_j\}_{j=-M}^M$, we obtain $\mathbf{p}_1$.

2. With $\mathbf{p}_1$ in hand, we iterate (11) a total of $n - 2$ times. This takes us to $\mathbf{p}_{n-1}$.

3. Then, by (10), we have that the density of $\tilde{x}_{i+1}$ evaluated at the data point x_{j+1} is
$$p(\tilde{x}_{i+1})\Big|_{\tilde{x}_{i+1}=x_{j+1}} = \int_{\tilde{x}_i} G(x_{j+1}, \tilde{x}_i)p(\tilde{x}_i)\, d\tilde{x}_i.$$
Applying trapezoidal quadrature to the right-hand side, we have
$$p(x_{j+1} \,|\, x_j, \theta) \approx \sum_j kG(x_{j+1}, z_j)p_{n-1}^j. \tag{13}$$

2.3. Metropolis Algorithm

Now that we understand how to compute the likelihood term, we return to the problem of sampling from the posterior (5). We give a classical Metropolis algorithm that incorporates the DTQ method described above for computing the likelihood. The fundamental idea here is to construct a discrete-time, continuous-space Markov chain whose equilibrium density is the posterior density (5). We then compute a sample path of this Markov chain beginning at a particular initial condition in parameter space. The sample path consists of a sequence of iterates; let $\mathbf{x}_i$, θ_i, and $(\sigma_\epsilon^2)_i$ denote the i-th iterates of the respective parameters.

We now describe how to proceed from the i-th to the $(i+1)$-st iterate of the Markov chain. To compute proposed iterates, we require access to random vectors/variables $Z_{\mathbf{x}} \in \mathbb{R}^{L+1}$, $Z_\theta \in \mathbb{R}^N$, and $Z_\sigma \in \mathbb{R}$. Then the Metropolis algorithm is as follows:

- Generate proposals by combining the old iterates with samples from the Z random variables:
$$\begin{aligned} \mathbf{x}_{i+1}^* &= \mathbf{x}_i + Z_{\mathbf{x}} \\ \theta_{i+1}^* &= \theta_i + Z_\theta \\ \log(\sigma_\epsilon^2)_{i+1}^* &= \log(\sigma_\epsilon^2)_i + Z_\sigma \end{aligned}$$
The log transformation ensures that the variance parameter σ_ϵ^2 only takes positive values.

- Calculate the ratio
$$\rho = \frac{p(\mathbf{x}^*_{i+1}, \theta^*_{i+1}, (\sigma^2_\epsilon)^*_{i+1} \,|\, \mathbf{y})}{p(\mathbf{x}_i, \theta_i, (\sigma^2_\epsilon)_i \,|\, \mathbf{y})}. \tag{14}$$
In practice, the denominator of this ratio has already been calculated at the previous step; only the numerator needs to be calculated. To compute the numerator, we use (5) together with the procedure described in Section 2.2, including (12), (10), and (13).

- Let u be a sample from a uniform random variable on $[0, 1]$. If $\rho > u$, we accept the proposal, setting $\mathbf{x}_{i+1} = \mathbf{x}^*_{i+1}$, $\theta_{i+1} = \theta^*_{i+1}$, and $(\sigma^2_\epsilon)_{i+1} = (\sigma^2_\epsilon)^*_{i+1}$. If $\rho \leq u$, we reject the proposal, setting $\mathbf{x}_{i+1} = \mathbf{x}_i$, $\theta_{i+1} = \theta_i$, $(\sigma^2_\epsilon)_{i+1} = (\sigma^2_\epsilon)_i$.

3. Scalable Implementation

There are two elements to our strategy of implementing the MCMC algorithm from Section 2 in a scalable fashion. The first aspect has to do with representing the main DTQ step (11) using Scala and Breeze. The second aspect has to do with using Apache Spark to evaluate each term in the product (6) in a parallel, distributed fashion. Note that all of our codes and data are available at https://github.com/hbhat4000/sdeinference/tree/master/sparkdtq. The main code to perform MCMC inference is `sparkdtq.sc`. Also note that all development and testing was carried out on a server with 24 effective cores (2 Intel Xeon E5-2620 chips at 2.0 GHz), 16 GB of RAM, and 4 TB of disk space.

3.1. Scala/Breeze

A typical approach in computational statistics to implement (11) would be to view the equation as matrix multiplication. Indeed, it is conceptually simple to view $kG(z_{j'}, z_j)$ as the (j', j) element of a $(2M+1) \times (2M+1)$ matrix $\mathcal{G}$, in which case (11) can be written as

$$\mathbf{p}_{i+1} = \mathcal{G}\mathbf{p}_i. \tag{15}$$

Multiplication by $\mathcal{G}$ steps the density forward by h units of time; for this reason, we refer to $\mathcal{G}$ as the propagator.

The above approach, while mathematically correct, does not recognize the sparsity of $\mathcal{G}$. In fact, we have an accurate estimate of where the nonzero entries of $\mathcal{G}$ are located: near the diagonal. From (9), we see that the argument to the exponential is zero when $a = b + f(b; \theta)h$. If we suppose that f is smooth, then the mean-value theorem implies that there exists ξ such that $b = (a - f(0)h)/(1 + f'(\xi)h)$. If f has bounded derivative (in this context, equivalent to assuming f is Lipschitz), then this implies that $b = a + O(h)$. The upshot is that for fixed a, when b is near a, the exponential term in (9) will be maximal. Similarly, we can conclude that for fixed a, when b is far from a, the exponential term in (9) will be negligible. We have focused on the exponential term under the assumption that the diffusion coefficient g, and hence the normalization term in (9), does not itself grow exponentially. In practice, this and other assumptions made above are quite reasonable.

Because of the decay of the Gaussian, for each fixed j', the (j', j) element of $\mathcal{G}$ need not be computed for all j. We fix a window size $\gamma > 0$ and then only compute $\mathcal{G}_{(j',j)}$ for those

j that satisfy both $-M \le j \le M$ and $j' - \gamma \le j \le j' + \gamma$. We choose γ large enough such that each density $\mathbf{p}_{i+1}$ is correctly normalized, i.e., such that $k \sum_j p_i^j$ is sufficiently close to 1. In all of our tests, we have been able to choose $\gamma \ll M$ while maintaining normalization to machine precision.

The main DTQ step (11), as it is implemented in Scala using the window parameter γ, is represented graphically in Figure 1. The code implementing this step makes use of the Breeze library (`https://github.com/scalanlp/breeze`) for numerical linear algebra in Scala. For additional efficiency, we utilize Breeze's support for the Intel Math Kernel Library (MKL).

Let us explain the procedure diagrammed in Figure 1. In the Metropolis algorithm given in Section 2.3, each time we must evaluate the numerator of ρ in (14), we must evaluate the likelihood function for a particular choice of $\mathbf{x}$ and θ. For this choice of θ, we evaluate each row of the propagator matrix $\mathcal{G}$ over a window of size $2\gamma+1$. These rows are represented by the pink rectangles; there are $2M+1$ such rows. Because each such row has the same size, we store the resulting collection as a Breeze `DenseVector` of `DenseVector`. This computation, which comprises the upper half of Figure 1, occurs once per Metropolis step.

To implement (15), we must now multiply the propagator matrix by the vector representing the pdf at time step i. This matrix-vector product can be equivalently described as a collection of $2M+1$ vector-vector dot products, where each vector is of size $2\gamma+1$. To generate the vectors to dot against collection of propagator vectors (already computed), we apply a windowing technique. Namely, for each element p_i^j of the pdf vector $\mathbf{p}_i$, we construct a window of size $2\gamma+1$ around the element p_i^j: the window consists of $(p_i^{j-\gamma}, \ldots, p_i^j, \ldots, p_i^{j+\gamma})$. Of course, it is understood here that $p_i^c = 0$ whenever $|c| > M$. In this way, we build a collection of windowed pdf vectors, represented in Figure 1 as the lower-right stack of blue rectangles. As above, the collection consists of $2M+1$ vectors, each of size $2\gamma+1$.

With the propagator collection denoted by `propagator` and the collection of windowed pdf vectors denoted by `allwins`, the Scala and Breeze syntax for computing all required dot products at once is, simply

```
px = propagator dot allwins
```
(16)

This line of code completes the implementation of (15). To iterate the procedure, we apply the windowing procedure—diagrammed in the lower half of Figure 1—to `px`, store the resulting collection in `allwins`, and then repeat (16).

The entire procedure diagrammed in Figure 1 makes use of functional programming techniques—specifically, Scala's `map` construct—to entirely avoid explicit loops. Additionally, note that this procedure is inherently more efficient than using a sparse matrix representation for the propagator matrix; we know where the nonzero entries belong, so we do not need to allocate either space or time to this task.

3.2. Spark

Spark enables parallel/distributed computation using the notion of a resilient distributed dataset (RDD). Since the main bottleneck in our Metropolis algorithm is the computation of the likelihood $p(\mathbf{x} \,|\, \theta)$, we turn to the question of converting the state time series $\mathbf{x}$ into an RDD. We think of this time series as a sequence of pairs (t_j, x_j) as depicted in the top

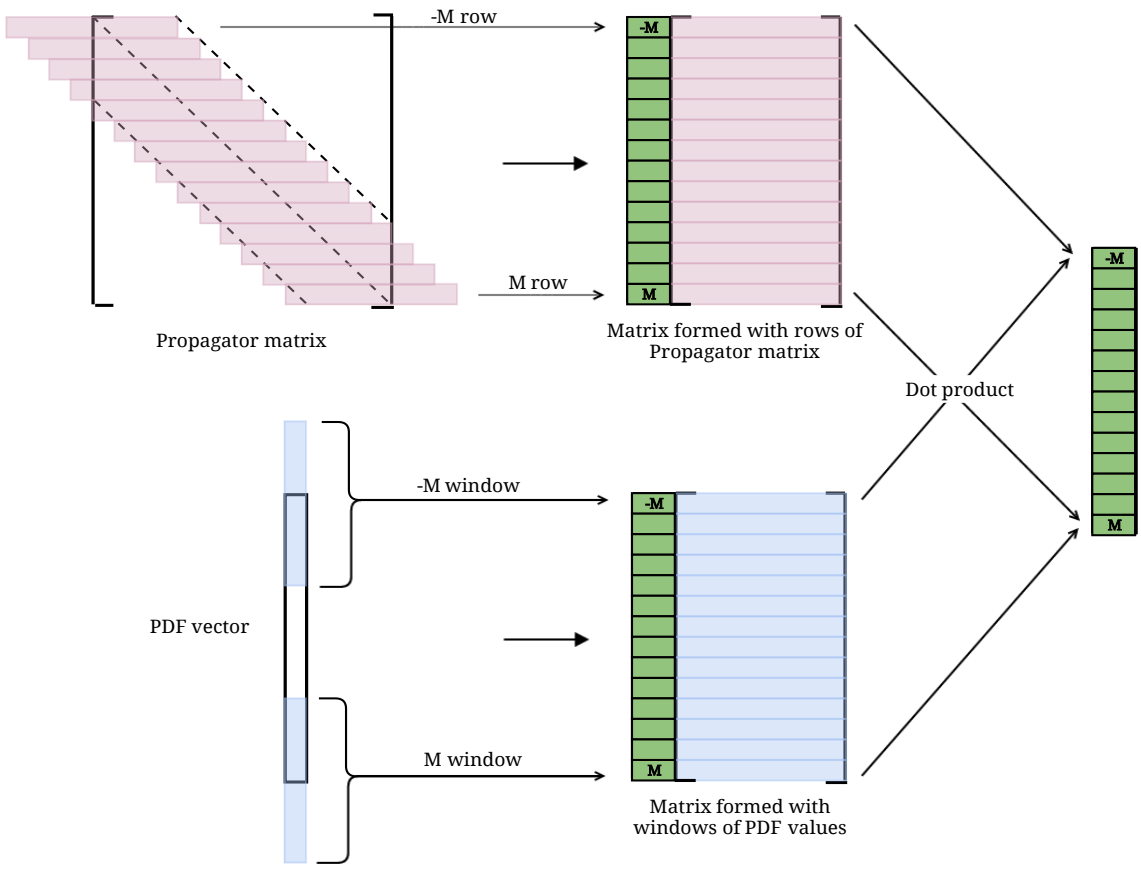

Figure 1: In order to implement the matrix-vector multiplication in (15) in a scalable way, we make use of the structure of the propagator matrix $\mathcal{G}$. Instead of computing all entries of this matrix, we compute and store only those entries that are close to the diagonal—the pink rectangles in the upper half of the diagram. The blue rectangles in the lower half of the diagram correspond to windowed versions of the pdf vector $\mathbf{p}_i$. In both cases, there is one windowed vector per row; the row numbers go from $-M$ to M as labeled. Both the pink and blue rectangles correspond to vectors of length $2\gamma+1$, with $\gamma \ll M$. The matrix-vector multiplication $\mathcal{G}\mathbf{p}_i$ then corresponds to a collection of $2M+1$ vector-vector dot products. This representation of (15) makes efficient use of Scala, Breeze, and the Intel MKL. For more details, see the description in Section 3.1.

line of Figure 2. When we examine (6), we see that to compute a given term in the product, we need access to neighboring pairs (t_j, x_j) and (t_{j+1}, x_{j+1}).

Hence, we map the original sequence of pairs, labeled as $\overrightarrow{tx}$ in Figure 2, to `tslices`, a Scala `Array` where each element is a vector of neighboring pairs. We convert this array to an RDD, `tslicesRDD`, using Spark's `sc.parallelize` method. When we subsequently

use a `map` operation to compute the log likelihood $\log p(x_{j+1} \,|\, x_j, \theta)$ term corresponding to each element of `tslicesRDD`, the computation takes place in parallel. Spark automatically distributes the propagator and the θ vector. For a non-equispaced time series problem, each calculation of $p(x_{j+1} \,|\, x_j, \theta)$ will take more (respectively, less) time when $t_{j+1} - t_j$ is larger (respectively, smaller). Again, Spark automatically assigns tasks to workers to compute the overall log likelihood efficiently.

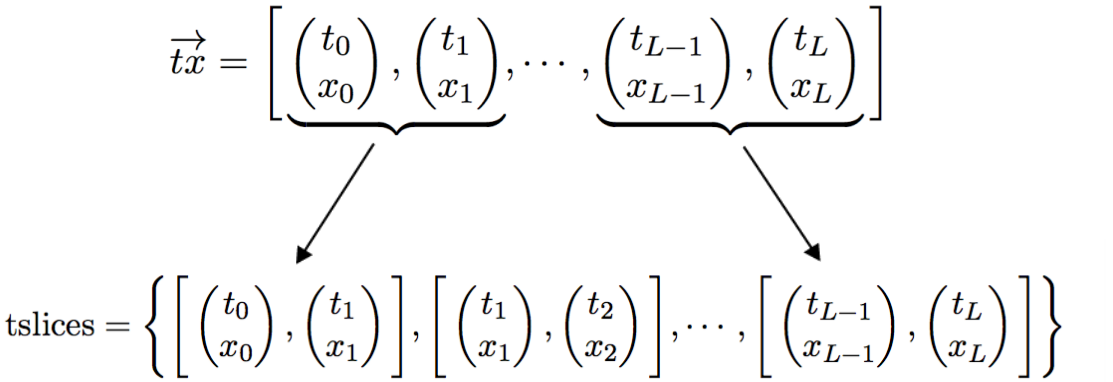

Figure 2: We use Spark to parallelize the computation of the likelihood (6). We accomplish this by converting the original time series (for states $\mathbf{x}$, not observations $\mathbf{y}$) from a vector of pairs to an array where each element is a vector of consecutive pairs. The original vector of pairs is labeled as $\overrightarrow{tx}$, and the Scala `Array` of consecutive pairs is `tslices`. This latter object can be easily converted into a Spark RDD; subsequent `map` operations on this RDD are executed in parallel.

4. Results

We begin with results on artificial data sets. The model used to generate these data sets is the Ornstein-Uhlenbeck SDE

$$dX_t = \theta_1(\theta_2 - X_t)dt + \theta_3 dW_t \tag{17}$$

together with the observation process

$$Y_t = X_t + \epsilon_t. \tag{18}$$

Specifically, what we do is apply the Euler-Maruyama discretization to (17) with a time step of $\kappa = 10^{-6}$. Suppose our temporal grid consists of $t_j = n(j)\kappa$, where $n(0) = 0$, and $n(j+1) > n(j)$. In some of the examples we pursue, $n(j)$ will be deterministic and the temporal grid will be equispaced, while in other examples, $n(j)$ will be stochastic and the temporal grid will be non-equispaced. Either way, we take $n(j)$ to have expected value 2×10^5, so that the average difference between temporal grid points $t_{j+1} - t_j$ is 0.2.

We then start at a random initial condition by sampling X_0 from a Gaussian random variable with mean 0 and variance 1. We step forward one step at a time (with time step

κ), saving the numerical solution X_t at points in time corresponding to the temporal grid points $\{t_j\}_{j=0}^{L}$. We label the points we save as $(x_0, x_1, \ldots, x_L) =: \mathbf{x}$, and then perturb them via (18) to generate $\mathbf{y}$. In particular, we set $y_j = x_j + Z$ where Z is normally distributed with mean 0 and variance σ^2.

The DTQ method described in Section 2.2 has four internal parameters: the time step h, the spatial grid spacing k, the spatial grid cutoff M, and the decay width γ of the Gaussian kernel G. For the tests described below, we will give the value of h that was used. All other parameters are as follows: $k = h^{0.75}$, $M = \lceil \pi/k^{1.5} \rceil$, and $\gamma = 25$.

4.1. Equispaced Time Series

In the first set of experiments, we follow the procedure outlined above to generate artificial data $\mathbf{y}$ with the temporal grid defined by $t_j = n(j)\kappa = (0.2)j$. The ground truth for the parameters consists of $\theta = (0.5, 1, 0.25)$ and $\sigma^2 = 0.01$. In addition to the other parameters, we focus on inferring θ_1 and θ_2; we fix $\theta_3 = 0.25$ throughout. In the Metropolis algorithm, we take as initial conditions $\mathbf{x}_0 = \mathbf{y}$, $\theta = (1, 0.1, 0.25)$, and $\sigma_\epsilon^2 = 1$.

For priors for θ_1 and θ_2, we use Gaussian densities with respective parameters $(\mu = 0.5, \sigma = 1)$ and $(\mu = 2, \sigma = 10)$. For σ_ϵ^2, we use an exponential prior with parameter $\lambda = 1$.

In the Metropolis algorithm, we take all proposal random variables to be independent Gaussians. In particular, $Z_{\mathbf{x}}$ is a collection of $L + 1$ independent Gaussians, each with parameters $(\mu = 0, \sigma = 0.02)$, Z_θ consists of two independent Gaussians, each with parameters $(\mu = 0, \sigma = 0.05)$, and Z_σ consists of a Gaussian with parameters $(\mu = 0, \sigma = 0.02)$. These distributions have been chosen, via trial and error, to yield a Metropolis acceptance rate that is between $20 - 40\%$ in all tests we have conducted.

For two values of the DTQ internal time step ($h = 0.02$ and $h = 0.01$), we apply the Metropolis algorithm to generate $10,000$ samples of the posterior (5). Note that $h = 0.02$ implies $M = 257$ and $h = 0.01$ implies $M = 559$; hence $\gamma = 25$ gives at least a 10-fold reduction in computational effort.

We discard the first 100 samples as burn-in samples, i.e., samples that are taken before the Metropolis Markov chain has, to a good approximation, reached equilibrium.

Using the samples thus obtained, we plot the posterior densities for θ_1, θ_2 and $\log_{10} \sigma_\epsilon^2$ in Figure 3. In this paper, all density plots use a Gaussian kernel with the "nrd" (or normal reference rule) bandwidth (Scott, 2015). The density in blue (respectively, black) corresponds to the samples produced using the DTQ method with $h = 0.02$ (respectively, $h = 0.01$). The red vertical lines indicate the ground truth values for each parameter.

Overall, we see that the Metropolis algorithm does a reasonable job of inferring the parameters. For the Orstein-Uhlenbeck model (17), Bayesian inference is non-trivial when the temporal spacing between observations is sufficiently large. Keeping in mind the logarithmic scale for the density of the third parameter, we still conclude that our method has the greatest room for improvement here. As we show below, however, the mean inferred value of σ_ϵ is consistent with the observation series $\mathbf{y}$ and the mean inferred state series $\mathbf{x}$.

4.2. Non-equispaced Time Series

In this next set of experiments, we follow a nearly identical procedure to that described in Section 4.1. The main difference is that the temporal grid is defined by $t_j = n(j)\kappa$ where $n(j)$

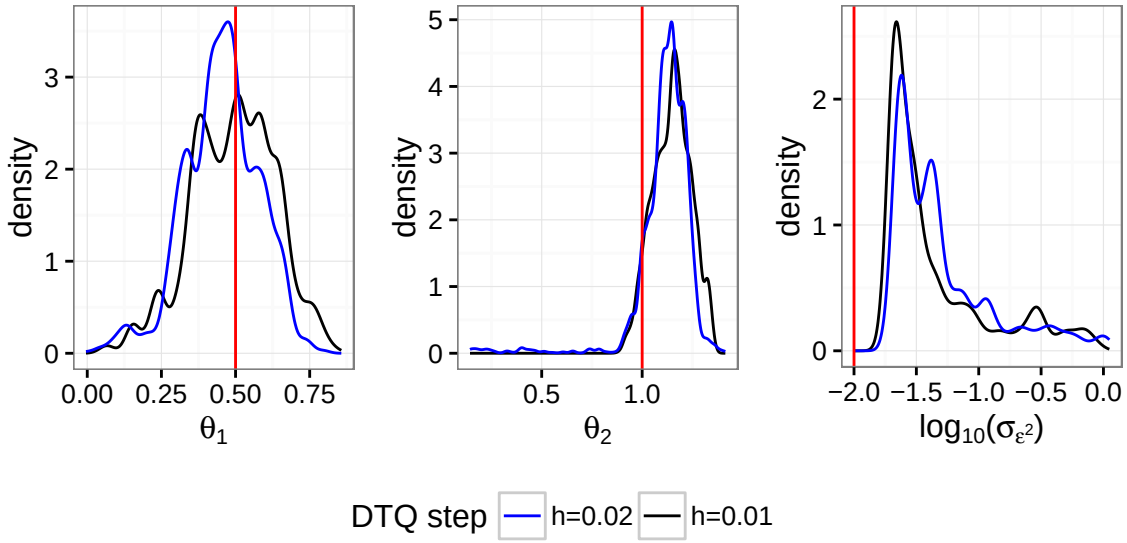

Figure 3: Posterior densities for the inference/filtering problem with equispaced time series $(\mathbf{t}, \mathbf{y})$. Each density is calculated on the basis of 9900 post-burn-in Metropolis samples computed using the indicated value of the internal DTQ time step parameter h. Overall, we see reasonable agreement between the ground truth values (indicated by red vertical lines) and the posterior densities.

is uniformly distributed on the integers between 4×10^4 and $4 \times 10^5 - 4 \times 10^4$. Effectively, this generates a time series with minimum, mean, and maximum spacings $t_{j+1} - t_j$ of, respectively, 0.04, 0.2, and 0.36.

We use the same priors and Metropolis initial conditions as in Section 4.1. We change the proposal distributions slightly. The parameters for $Z_{\mathbf{x}}$ and Z_σ are now $(\mu = 0, \sigma = 0.01)$, while for Z_θ the parameters are still $(\mu = 0, \sigma = 0.05)$.

For two values of the DTQ internal time step ($h = 0.02$ and $h = 0.01$), we apply the Metropolis algorithm to generate $10,000$ samples of the posterior (5). Again, we discard the first 100 samples as burn-in samples.

Using the samples thus obtained, and using the same procedure described in Section 4.1, we plot the posterior densities for θ_1, θ_2 and $\log_{10} \sigma_\epsilon^2$ in Figure 4. Overall, as compared with Figure 3, we see improved agreement between the ground truth values and the posteriors for θ_2 and $\log_{10}$, while the posterior for θ_1 is still reasonably accurate.

We turn to the filtering results, focusing on the post-burn-in samples of $\mathbf{x}$ generated with DTQ parameter $h = 0.01$. In Figure 5, we plot the original non-equispaced observation series $\mathbf{y}$ in red. We have plotted in grey each of the 9900 samples of the state series $\mathbf{x}$; the mean of these samples is plotted in black. We see that the Metropolis sampler does indeed explore a number of different trajectories for the state series, and that the mean inferred state series corresponds to a smoothed version of the original observation series.

In Figure 6, we again plot the original non-equispaced observation series $\mathbf{y}$ in red and the mean inferred state series $\mathbf{x}$ in black. This time, however, we add/subtract the mean

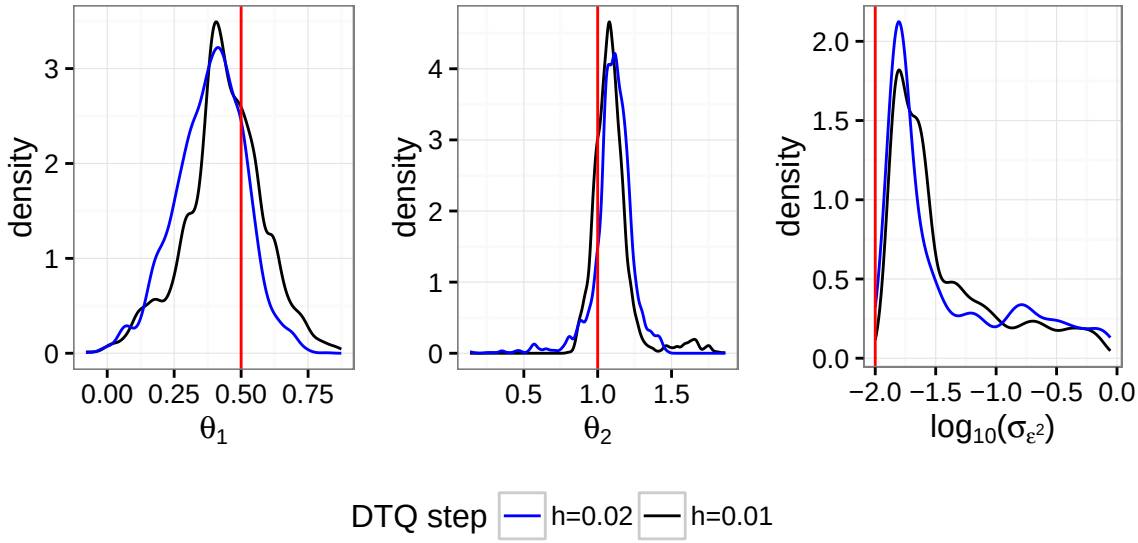

Figure 4: Posterior densities for the inference/filtering problem with non-equispaced time series $(\mathbf{t}, \mathbf{y})$. Each density is calculated on the basis of 9900 post-burn-in Metropolis samples computed using the indicated value of the internal DTQ time step parameter h. Overall, we see reasonable agreement between the ground truth values (indicated by red vertical lines) and the posterior densities.

inferred value of σ_ϵ to $\mathbf{y}$ to obtain error bars associated with each observation. These error bars are plotted in grey. The idea behind this plot is that if (1b) holds, then given the symmetry of the random variables ϵ_t, it should also be true that $X_t = Y_t + \epsilon'_t$, where ϵ'_t has the same distribution as ϵ_t. To be self-consistent, the observation series should, at least most of the time, lie within one (inferred) standard deviation σ_ϵ of the (inferred) state series. The plot in Figure 6 confirms that this is the case.

4.3. Scaling

We have conducted tests to explore the relationship between running time and L, the length of the observation series. For each $L \in \{124, 250, 500, 2500\}$, and for each $h \in \{0.02, 0.01\}$, we have generated a time series of the indicated length, run our inference/filtering code, and recorded the amount of time T required to generate 1000 Metropolis samples of the posterior. The results are plotted in Figure 7 with log-transformed axes. For each set of points, we fit a line of the form $\log T = \beta_0 + \beta_1 \log L$, which corresponds to the law $T = e^{\beta_0} L^{\beta_1}$. The solid lines in Figure 7 correspond to this latter law, plotted on the same log-transformed axes. For the $h = 0.02$ line, we obtain $\beta_1 = 0.9092$; for the $h = 0.01$ line, we obtain $\beta_1 = 0.9639$. These results are consistent with $O(L)$ temporal scaling.

We have also explored how the running time of our code changes as we change the number ν of Spark processors. We begin with a non-equispaced time series of length 2501. For each $h \in \{0.02, 0.01\}$, we recorded the time T required to generate 10 Metropolis samples. For each set of points, we fit a line of the form $\log T = \beta_0 + \beta_1 \log \nu$. Both the raw

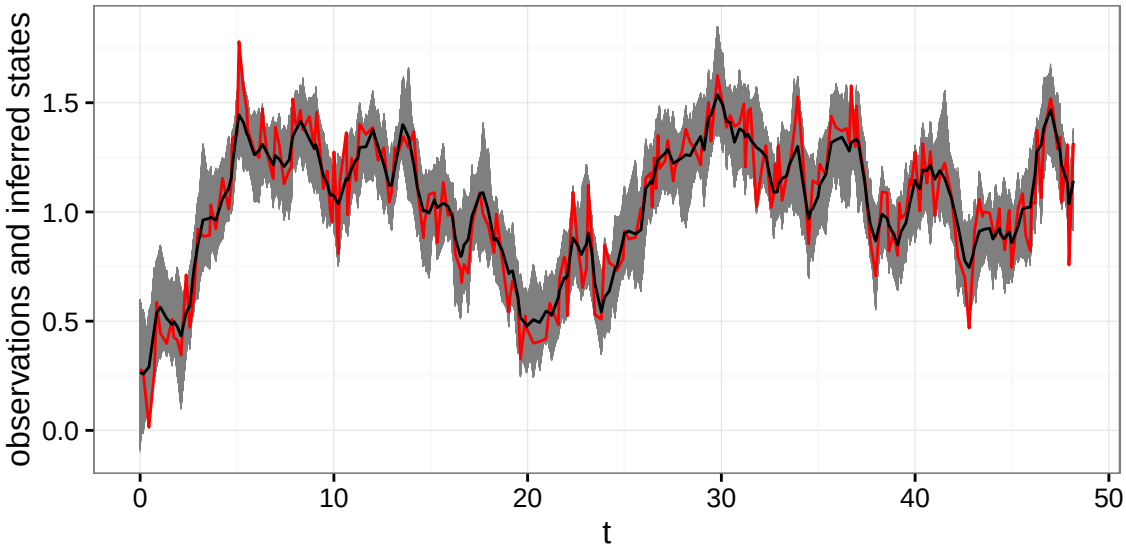

Figure 5: We plot the observations (in red) together with each of the samples of the state series $\mathbf{x}$. Each such sample is a grey curve, and the mean of all such grey curves is plotted in black. We refer to the black curve as the mean inferred state series.

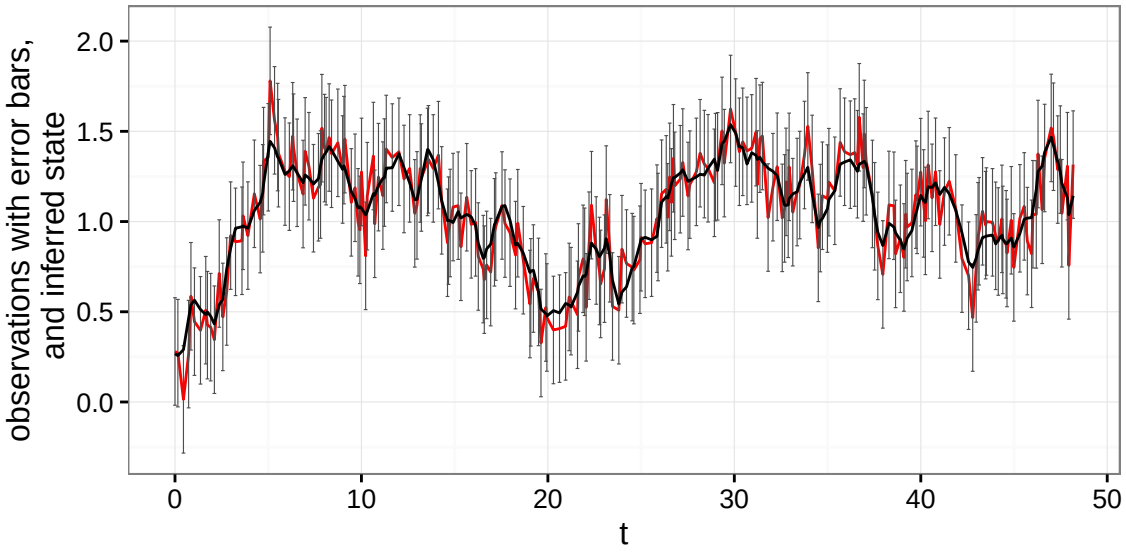

Figure 6: We plot the observations (in red) together with the mean inferred state series (in black). The error bars (grey) are computed by adding/subtracting the mean inferred value of σ_ϵ to/from the observation series $\mathbf{y}$. Note that the mean inferred state is typically within one σ_ϵ of the corresponding observation.

data and lines are plotted on log-transformed axes in Figure 8. For the $h = 0.02$ line, we

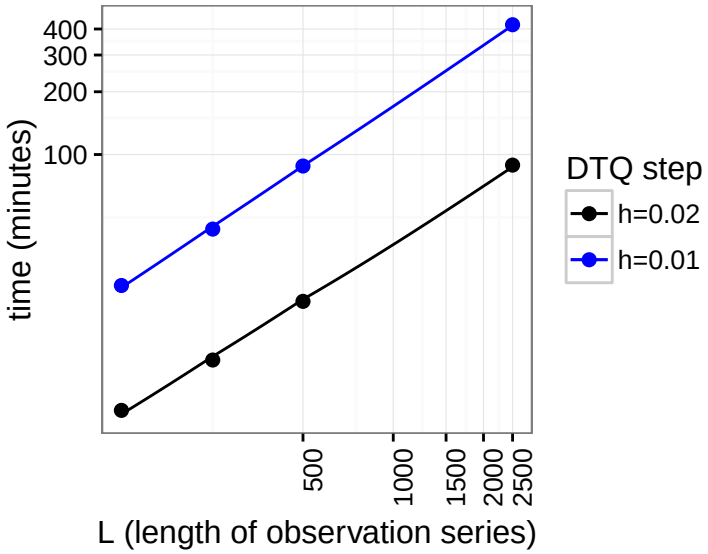

Figure 7: For each indicated value of L, we have generated a time series of length L, run our inference/filtering code, and recorded the amount of time T required to generate 1000 Metropolis samples of the posterior. We fit lines to $\log T$ as a function of $\log L$—both the lines and the original data are plotted on log-transformed axes. The slopes of the lines are less than 1, consistent with $O(L)$ temporal scaling.

obtain $\beta_1 = -0.4164$, while for the $h = 0.01$ line, we obtain $\beta_1 = -0.4699$. These results suggest $O(\nu^{-1/2})$ scaling.

5. Discussion and Conclusion

The results from Section 4 show that while decreasing the value of the DTQ internal step h does change the sampled values and the plotted densities, the change is not significant. We believe much greater gains (in terms of agreement between the posterior modes and the ground truth values) would be achieved by improving both the proposal strategy and the vanilla Metropolis accept/reject step; such ideas have been pursued successfully by Fuchs (2013) and others, and we seek to implement these improvements in future work. For now, however, we conclude that the implementation described in this paper does indeed perform Bayesian filtering and inference at a baseline acceptable level.

Aside from improving the Metropolis algorithm, we see three main areas for improvement. First, we have yet to test and tune our implementation on a large-scale, distributed Spark cluster. Second, we believe we can derive large gains in performance by adapting our algorithm to work in a streaming fashion. Specifically, instead of inferring the entire state series $\mathbf{x}$ at once, as we currently do, we can proceed one step a time through the temporal sequence of observations. Third, we have already begun to incorporate the DTQ into an adjoint method suitable for computing gradients of the likelihood. This will enable us to apply techniques such as stochastic gradient descent or Hamiltonian Monte Carlo to

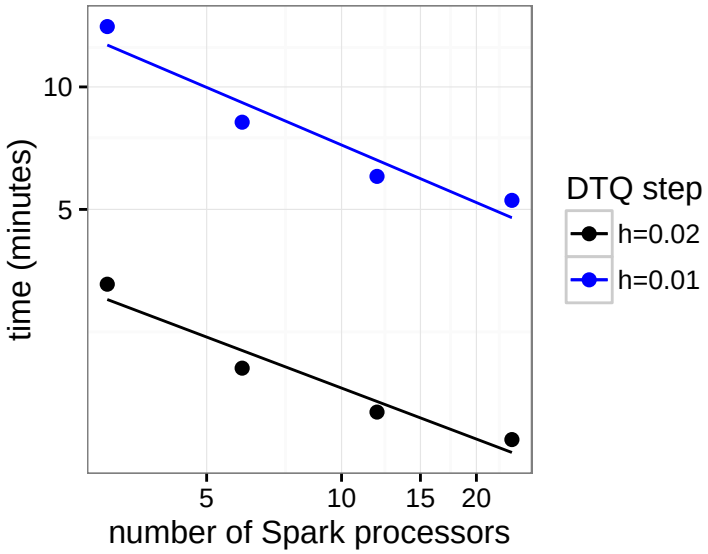

Figure 8: For a non-equispaced time series of length 2501, and for each $h \in 0.02, 0.01$, we ran our code with the indicated number of Spark processors ν. We recorded T, the time required to generate 10 Metropolis samples of the posterior. We fit lines to $\log T$ as a function of $\log \nu$—both the lines and the original data are plotted on log-transformed axes. The slopes of the lines are close to -0.5, suggesting $O(\nu^{-1/2})$ scaling.

our inference problem for either fast MLE/MAP point estimation or accelerated sampling from posteriors. These steps, and possibly others, will become necessary as we adapt our methods to multi-dimensional time series problems, a task we have already begun.

Acknowledgments

This work was partially supported by a grant from UC Merced's Committee on Research (awarded to H.S. Bhat) and by UC Merced USAP fellowships (awarded to R.W.M.A. Madushani and S. Rawat).

References

Yacine Aït-Sahalia. Maximum likelihood estimation of discretely sampled diffusions: A closed-form approximation approach. *Econometrica*, 70(1):223–262, 2002.

C. Archambeau, D. Cornford, M. Opper, and J. Shawe-Taylor. Gaussian process approximations of stochastic differential equations. *Journal of Machine Learning Research: Workshop and Conference Proceedings*, 1:1–16, 2007a.

C. Archambeau, M. Opper, Y. Shen, D. Cornford, and J. Shawe-Taylor. Variational inference for diffusion processes. In *Advances in Neural Information Processing Systems 20*, pages 17–24, 2007b.

Harish S. Bhat and R. W. M. A. Madushani. Density tracking by quadrature for stochastic differential equations. Preprint, 2016.

C. Fuchs. *Inference for Diffusion Processes: With Applications in Life Sciences*. Springer, Berlin, 2013.

S. M. Iacus. *Simulation and Inference for Stochastic Differential Equations: With R Examples*. Springer Series in Statistics. Springer, New York, 2009.

Asger Roer Pedersen. A new approach to maximum likelihood estimation for stochastic differential equations based on discrete observations. *Scandinavian Journal of Statistics*, 22(1):55–71, 1995.

U. Picchini. Inference for SDE models via approximate Bayesian computation. *Journal of Computational and Graphical Statistics*, 23(4):1080–1100, 2014.

A. Ruttor, P. Batz, and M. Opper. Approximate Gaussian process inference for the drift function in stochastic differential equations. In *Advances in Neural Information Processing Systems 26*, pages 2040–2048, 2013.

P. Santa-Clara. Simulated likelihood estimation of diffusions with an application to the short term interest rate. Working Paper 12-97, UCLA Anderson School of Management, 1997.

D. W. Scott. *Multivariate Density Estimation: Theory, Practice, and Visualization*. Wiley, Hoboken, NJ, second edition, 2015.

H. Sørensen. Parametric inference for diffusion processes observed at discrete points in time: a survey. *International Statistical Review*, 72(3):337–354, 2004.

M. D. Vrettas, M. Opper, and D. Cornford. Variational mean-field algorithm for efficient inference in large systems of stochastic differential equations. *Physical Review E*, 91 (012148), 2015.