


# Anytime Monte Carlo

Lawrence M. Murray<sup>1</sup>, Sumeetpal S. Singh<sup>2\*</sup>  and Anthony Lee<sup>3</sup>

<sup>1</sup>Uber AI, San Francisco, CA, USA

<sup>2</sup>The Alan Turing Institute, University of Cambridge, Cambridge, United Kingdom

<sup>3</sup>The Alan Turing Institute, University of Bristol, Bristol, United Kingdom

\*Corresponding author. E-mail: sss40@cam.ac.uk

**Received:** 16 February 2021; **Revised:** 02 April 2021; **Accepted:** 05 May 2021

**Keywords:** Real-time computing; Sequential Monte Carlo; Markov Chain Monte Carlo; Parallel computing

## Abstract

Monte Carlo algorithms simulate some prescribed number of samples, taking some random real time to complete the computations necessary. This work considers the converse: to impose a real-time budget on the computation, which results in the number of samples simulated being random. To complicate matters, the real time taken for each simulation may depend on the sample produced, so that the samples themselves are not independent of their number, and a length bias with respect to compute time is apparent. This is especially problematic when a Markov chain Monte Carlo (MCMC) algorithm is used and the final state of the Markov chain—rather than an average over all states—is required, which is the case in parallel tempering implementations of MCMC. The length bias does not diminish with the compute budget in this case. It also occurs in sequential Monte Carlo (SMC) algorithms, which is the focus of this paper. We propose an *anytime* framework to address the concern, using a continuous-time Markov jump process to study the progress of the computation in real time. We first show that for any MCMC algorithm, the length bias of the final state's distribution due to the imposed real-time computing budget can be eliminated by using a multiple chain construction. The utility of this construction is then demonstrated on a large-scale SMC<sup>2</sup> implementation, using four billion particles distributed across a cluster of 128 graphics processing units on the Amazon EC2 service. The anytime framework imposes a real-time budget on the MCMC move steps within the SMC<sup>2</sup> algorithm, ensuring that all processors are simultaneously ready for the resampling step, demonstrably reducing idleness to due waiting times and providing substantial control over the total compute budget.

## Impact Statement

Real-time budgets arise in numerous settings, our focus is on the deployment of Monte Carlo methods on large-scale distributed computing systems, using real-time budgets to ensure the simultaneous readiness of multiple processors for collective communication, minimizing wasteful idle times that precedes it. A conceptual solution is found in the *anytime algorithm*. Our first major contribution is a framework for converting any existing Monte Carlo algorithm into an anytime Monte Carlo algorithm, by running multiple Markov chains in a particular manner. The second major contribution is a demonstration on a large-scale Monte Carlo implementation executing billions of samples distributed across a cluster of 128 graphics processing units. Our anytime algorithm demonstrably reduces idleness and provides control over the total compute budget.

## 1. Introduction

Real-time budgets arise in embedded systems, fault-tolerant computing, energy-constrained computing, distributed computing and, potentially, management of cloud computing expenses and the fair

computational comparison of methods. Here, we are particularly interested in the development of Monte Carlo algorithms to observe such budgets, as well as the statistical properties—and limitations—of these algorithms. While the approach has broader applications, the pressing motivation in this work is the deployment of Monte Carlo methods on large-scale distributed computing systems, using real-time budgets to ensure the simultaneous readiness of multiple processors for collective communication, minimizing the idle wait time that typically precedes it.

A conceptual solution is found in the *anytime algorithm*. An anytime algorithm maintains a numerical result at all times, and will improve upon this result if afforded extra time. When interrupted, it can always return the current result. Consider, for example, a greedy optimization algorithm: its initial result is little more than a guess, but at each step it improves upon that result according to some objective function. If interrupted, it may not return the optimal result, but it should have improved on the initial guess. A conventional Markov chain Monte Carlo (MCMC) algorithm, however, is not anytime if we are interested in the final state of the Markov chain at some interruption time: as will be shown, the distribution of this state is length-biased by compute time, and this bias does not reduce when the algorithm is afforded additional time.

MCMC algorithms are typically run for some long time and, after removing an initial burn-in period, the expectations of some functionals of interest are estimated from the remainder of the chain. The prescription of a real-time budget,  $t$ , introduces an additional bias in these estimates, as the number of simulations that will have been completed is a random variable,  $N(t)$ . This bias diminishes in  $t$ , and for long-run Markov chains may be rendered negligible. In motivating this work, however, we have in mind situations where the final state of the chain is most important, rather than averages over all states. The bias in the final state does not diminish in  $t$ . Examples where this may be important include (a) sequential Monte Carlo (SMC), where, after resampling, a small number of local MCMC moves are performed on each particle before the next resampling step and (b) parallel tempering, where, after swapping between chains, a number of local MCMC moves are performed on each chain before the next swap. In a distributed computing setting, the resampling step of SMC, and the swap step of parallel tempering, require the synchronization of multiple processes, such that all processors must idle until the slowest completes. By fixing a real-time budget for local MCMC moves, we can reduce this idle time and eliminate the bottleneck, but must ensure that the length bias imposed by the real-time budget is negligible, if not eliminated entirely. In a companion paper (d’Avigneau et al., 2020), we also develop this idea for parallel tempering-based MCMC.

The compute time of a Markov chain depends on exogenous factors such as processor hardware, memory bandwidth, I/O load, network traffic, and other jobs contesting the same processor. But, importantly, it may also depend on the states of the Markov chain. Consider (a) inference for a mixture model where one parameter gives the number of components, and where the time taken to evaluate the likelihood of a dataset is proportional to the number of components; (b) a differential model that is numerically integrated forward in time with an adaptive time step, where the number of steps required across any given interval is influenced by parameters; (c) a complex posterior distribution simulated using a pseudomarginal method (Andrieu and Roberts, 2009), where the number of samples required to marginalize latent variables depends on the parameters; (d) a model requiring approximate Bayesian computation with a rejection sampling mechanism, where the acceptance rate is higher for parameters with higher likelihood, and so the compute time lower, and vice versa. Even in simple cases, there may be a hidden dependency. Consider, for example, a Metropolis–Hastings sampler where there is some seemingly innocuous book-keeping code, such as for output, to be run when a proposal is accepted, but not when a proposal is rejected. States from which the proposal is more likely to accept then have longer expected hold time due to this book-keeping code. In general, we should assume that there is indeed a dependency, and assess whether the resulting bias is appreciable.

The first major contribution of the present work is a framework for converting any existing Monte Carlo algorithm into an anytime Monte Carlo algorithm, by running multiple Markov chains in a particular manner. The framework can be applied in numerous contexts where real-time considerations might be beneficial. The second major contribution is an application in one such context: an SMC<sup>2</sup>

algorithm deployed on a large-scale distributed computing system. An anytime treatment is applied to the MCMC moves steps that precede resampling, which requires synchronization between processors, and can be a bottleneck in distributed deployment of the algorithm. The real-time budget reduces wait time at synchronization, relieves the resampling bottleneck, provides direct control over the compute budget for the most expensive part of the computation, and in doing so provides indirect control over the total compute budget.

Anytime Monte Carlo algorithms have recently garnered some interest. Paige et al. (2014) propose an anytime SMC algorithm called the *particle cascade*. This transforms the structure of conventional SMC by running particles to completion one by one with the sufficient statistics of preceding particles used to make birth–death decisions in place of the usual resampling step. To circumvent the sort of real-time pitfalls discussed in this work, a random schedule of work units is used. This requires a central scheduler, so it is not immediately obvious how the particle cascade might be distributed. In contrast, we propose, in this work, an SMC algorithm with the conventional structure, but including parameter estimation as in SMC<sup>2</sup> (Chopin et al., 2013), and an anytime treatment of the move step. This facilitates distributed implementation, but provides only indirect control over the total compute budget.

The construction of Monte Carlo estimators under real-time budgets has been considered before. Heidelberger (1988), Glynn and Heidelberger (1990), and Glynn and Heidelberger (1991) suggest a number of estimators for the mean of a random variable after performing independent and identically distributed (iid) simulation for some prescribed length of real time. Bias and variance results are established for each. The validity of their results relies on the exchangeability of simulations conditioned on their number, and does not extend to MCMC algorithms except for the special cases of regenerative Markov chains and perfect simulation. The present work establishes results for MCMC algorithms (for which, of course, iid sampling is a special case) albeit for a different problem definition.

A number of other recent works are relevant. Recent papers have considered the distributed implementation of Gibbs sampling and the implications of asynchronous updates in this context, which involves real time considerations (Terenin et al., 2015; De Sa et al., 2016). As mentioned above, optimization algorithms already exhibit anytime behavior and it is natural to consider whether they might be leveraged to develop anytime Monte Carlo algorithms, perhaps in a manner similar to the weighted likelihood bootstrap (Newton and Raftery, 1994). Meeds and Welling (2015) suggest an approach along this vein for approximate Bayesian computation. Beyond Monte Carlo, other methods for probabilistic inference might be considered in an anytime setting. Embedded systems, with organic real-time constraints, yield natural examples (e.g., Ramos and Cozman, 2005).

The remainder of the paper is structured as follows. Section 2 formalizes the problem and the framework of a proposed solution; proofs are deferred to Appendix 7 in Supplementary Material. Section 3 uses the framework to establish SMC algorithms with anytime moves, among them an algorithm suitable for large-scale distributed computing. Section 4 validates the framework on a simple toy problem, and demonstrates the SMC algorithms on a large-scale distributed computing case study. Section 5 discusses some of the finer points of the results, before Section 6 concludes.

## 2. Framework

Let  $(X_n)_{n=0}^\infty$  be a Markov chain with initial state  $X_0$ , evolving on a space  $\mathbb{X}$ , with transition kernel  $X_n | x_{n-1} \sim \kappa(dx | x_{n-1})$  and target (invariant) distribution  $\pi(dx)$ . We do not assume that  $\kappa$  has a density, for example, it may be a Metropolis–Hastings kernel. (In contrast the notation for a probability density would be  $\kappa(x | x_{n-1})$ .) A computer processor takes some random and positive real time  $H_{n-1}$  to complete the computations necessary to transition from  $X_{n-1}$  to  $X_n$  via  $\kappa$ . Let  $H_{n-1} | x_{n-1} \sim \tau(dh | x_{n-1})$  for some probability distribution  $\tau$ . The cumulative distribution function (cdf) corresponding to  $\tau$  is denoted  $F_\tau(h | x_{n-1})$ , and its survival function  $\bar{F}_\tau(h | x_{n-1}) = 1 - F_\tau(h | x_{n-1})$ . We assume (a) that  $H > \epsilon > 0$  for some minimal time  $\epsilon$ , (b) that  $\sup_{x \in \mathbb{X}} \mathbb{E}_\tau[H | x] < \infty$ , and (c) that  $\tau$  is homogeneous in time.

The distribution  $\tau$  captures the dependency of hold time on the current state of the Markov chain, as well as on exogenous factors such as contesting jobs that run on the same processor, memory

management, I/O, network latency, and so on. The first two assumptions seem reasonable: on a computer processor, a computation must take at least one clock cycle, justifying a lower bound on  $H$ , and be expected to complete, justifying finite expectation. The third assumption, of homogeneity, is more restrictive. Exogenous factors may produce transient effects, such as a contesting process that begins part way through the computation of interest. We discuss the relaxation of this assumption—as future work—in Section 5.

Besides these assumptions, no particular form is imposed on  $\tau$ . Importantly, we do not assume that  $\tau$  is memoryless. In general, nothing is known about  $\tau$  except how to simulate it, precisely by recording the length of time  $H_{n-1}$  taken to simulate  $X_n | x_{n-1} \sim \kappa(dx | x_{n-1})$ . In this sense, there exists a joint kernel  $\kappa(dx_n, dh_{n-1} | x_{n-1}) = \kappa(dx_n | h_{n-1}, x_{n-1})\tau(dh_{n-1} | x_{n-1})$  for which the original kernel  $\kappa$  is the marginal over all possible hold times  $H_{n-1}$  in transit from  $X_{n-1}$  to  $X_n$ :

$$\kappa(dx | x_{n-1}) = \int_0^\infty \kappa(dx | x_{n-1}, h_{n-1})\tau(dh_{n-1} | x_{n-1}).$$

We now construct a real-time semi-Markov jump process to describe the progress of the computation in real time. The states of the process are given by the sequence  $(X_n)_{n=0}^\infty$ , with associated hold times  $(H_n)_{n=0}^\infty$ . Define the arrival time of the  $n$ th state  $X_n$  as

$$A_n := \sum_{i=0}^{n-1} H_i,$$

for all  $n \geq 1$ , and  $a_0 = 0$ , and a process counting the number of arrivals by time  $t$  as

$$N(t) := \max \{n : A_n \leq t\}.$$

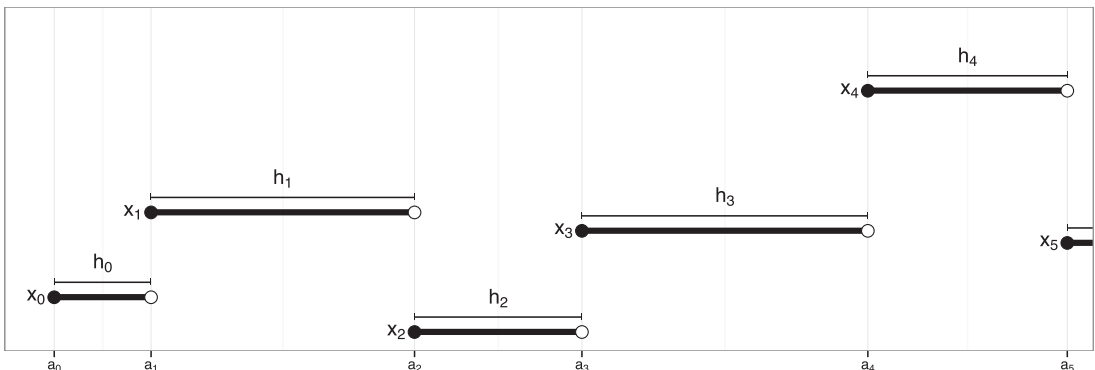
Figure 1 illustrates a realization of the process.

Now, construct a continuous-time Markov jump process to chart the progress of the simulation in real time. Let  $X(t) := X_{N(t)}$  (the interpolated process in Figure 1) and define the lag time elapsed since the last jump as  $L(t) := t - A_{N(t)}$ . Then  $(X, L)(t)$  is a Markov jump process.

The process  $(X, L)(t)$  is readily manipulated to establish the following properties. Proofs are given in Appendix A in Supplementary Material.

**Proposition 1.** *The stationary distribution of the Markov jump process  $(X, L)(t)$  is*

$$\alpha(dx, dl) = \frac{\bar{F}_\tau(l|x)}{\mathbb{E}_\tau[H]} \pi(dx) dl.$$



**Figure 1.** A realization of a Markov chain  $(X_n)_{n=0}^\infty$  in real time, with hold times  $(H_n)_{n=0}^\infty$  and arrival times  $(A_n)_{n=0}^\infty$ .

**Corollary 2.** *With respect to  $\pi(dx)$ ,  $\alpha(dx)$  is length-biased by expected hold time:*

$$\alpha(dx) = \frac{\mathbb{E}_\tau[H|x]}{\mathbb{E}_\tau[H]} \pi(dx).$$

The interpretation of these results are as follows: if the chain  $(X, L)(t)$  is initialized at some time  $t = b < 0$  such that it is in stationarity at time  $t = 0$ , then  $X(0)$ 's distribution is precisely given in Corollary 2. By Corollary 2, in stationarity, the likelihood of a particular  $x$  appearing is proportional to the likelihood  $\pi$  with which it arises under the original Markov chain, and the expected length of real time for which it holds when it does.

**Corollary 3.** *Let  $(X, L) \sim \alpha$ , then the conditional probability distribution of  $X$  given  $L < \epsilon$  is  $\pi$ .*

Corollary 3 simply recovers the original Markov chain from the Markov jump process. This result follows since  $\bar{F}_\tau(l|x) = 1$  (of Proposition 1) for all  $x$  and  $l \leq \epsilon$ .

We refer to  $\alpha$  as the *anytime distribution*. It is precisely the stationary distribution of the Markov jump process. The new name is introduced to distinguish it from the stationary distribution  $\pi$  of the original Markov chain, which we continue to refer to as the *target distribution*.

Finally, we state an ergodic theorem from Alsmeyer (1997); see also Alsmeyer (1994). Rather than study the process  $(X, L)(t)$ , we can equivalently study  $(X_n, A_n)_{n=1}^\infty$ , with the initial state being  $(X_0, 0)$ . This is a Markov renewal process. Conditioned on  $(X_n)_{n=0}^\infty$ , the hold times  $(H_n)_{n=0}^\infty$  are independent. This conditional independence can be exploited to derive ergodic properties of  $(X_n, A_n)_{n=1}^\infty$ , based on assumed regularity of the driving chain  $(X_n)_{n=0}^\infty$ .

**Proposition 4** (Alsmeyer, 1997, Corollary1). *Assume that the Markov chain  $(X_n)_{n=1}^\infty$  is Harris recurrent. For a function  $g : \mathbb{X} \rightarrow \mathbb{R}$  with  $\int |g(x)| \alpha(dx) < \infty$ ,*

$$\lim_{t \rightarrow \infty} \mathbb{E}[g(X(t)) | x(0), l(0)] = \int g(x) \alpha(dx)$$

for  $\pi$ -almost all  $x(0)$  and all  $l(0)$ .

A further interpretation of this result confirms that, regardless of how we initialize the chain  $(X, L)(t)$  at time  $t = 0$ , at a future time  $t = T$  the distribution of  $X(T)$  is close to  $\alpha$  and approaches  $\alpha$  as  $T \rightarrow \infty$ .

### 2.1. Establishing anytime behavior

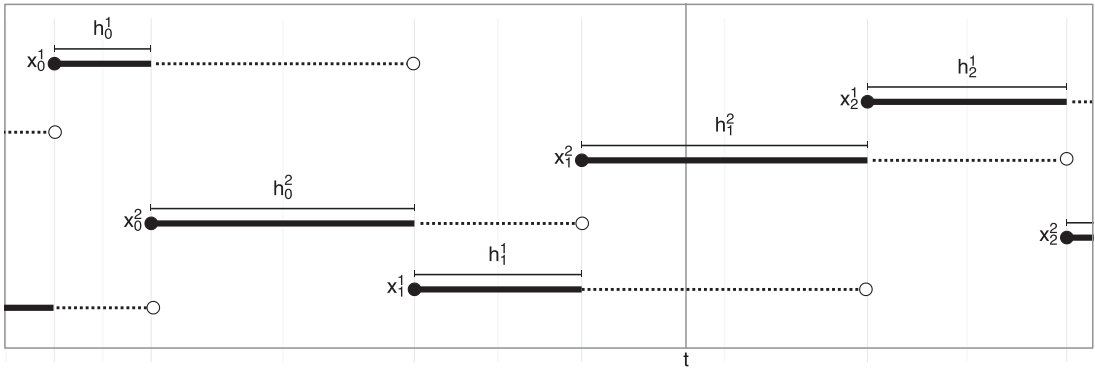
The above results establish that, when interrupted at real time  $t$ , the state of a Monte Carlo computation is distributed according to the anytime distribution,  $\alpha$ . We wish to establish situations in which it is instead distributed according to the target distribution,  $\pi$ . This will allow us to draw samples of  $\pi$  simply by interrupting the running process at any time  $t$ , and will form the basis of anytime Monte Carlo algorithms.

Recalling Corollary 2, a sufficient condition to establish  $\alpha(dx) = \pi(dx)$  is that  $\mathbb{E}_\tau[H|x] = \mathbb{E}_\tau[H]$ , that is for the expected hold time to be independent of  $X$ . For iid sampling, this is trivially the case: we have  $\kappa(dx|x_{n-1}) = \pi(dx)$ , the hold time  $H_{n-1}$  for  $X_{n-1}$  is the time taken to draw  $X_n \sim \pi(dx)$  and so is independent of  $X_{n-1}$ , and  $\mathbb{E}_\tau[H|x] = \mathbb{E}_\tau[H]$ .

For non-iid sampling, first consider the following change to the Markov kernel:

$$X_n | x_{n-2} \sim \kappa(dx | x_{n-2}).$$

That is, each new state  $X_n$  depends not on the previous state,  $x_{n-1}$ , but on one state back again,  $x_{n-2}$ , so that odd- and even-numbered states are independent. The hold times of the even-numbered states are the



**Figure 2.** Illustration of the multiple chain concept with two Markov chains. At any time, one chain is being simulated (indicated with a dotted line) while one chain is waiting (indicated with a solid line). When querying the process at some time  $t$ , it is the state of the waiting chain that is reported, so that the hold times of each chain are the compute times of the other chain. For  $K + 1 \geq 2$  chains, there is always one chain simulating while  $K$  chains wait, and when querying the process at some time  $t$ , the states of all  $K$  waiting chains are reported.

compute times of the odd-numbered states and vice versa, so that hold times are independent of states, and the desired property  $\mathbb{E}_\tau[H | x] = \mathbb{E}_\tau[H]$ , and so  $\alpha(dx) = \pi(dx)$ , is achieved. This sampling strategy is illustrated in Figure 2 where the odd chain is Chain 1 (Superscript 1) and the even chain is Chain 2. When querying the sampler at any time  $t$ , the sampler returns the chain that is *holding* and not being worked, which is Chain 2 in the figure. It follows then that the returned sample is distributed according to  $\pi$  as desired. In an attempt to increase the efficiency of this procedure for SMC methods in Section 3, we extend the idea to  $K + 1$  chains as follows. The processor works/samples each of the  $K + 1$  chains in sequential order. At any time  $t$ , when queried, all but one chain is being worked and the processor returns the states of the  $K$  chains that are not being worked on. Each state will then be an independent sample with distribution  $\pi$ .

Formally, suppose that we are simulating  $K$  number of Markov chains, with  $K$  a positive integer, plus one extra chain. Denote these  $K + 1$  chains as  $(X_n^{1:K+1})_{n=0}^\infty$ . For simplicity, assume that all have the same target distribution  $\pi$ , kernel  $\kappa$ , and hold time distribution  $\tau$ . The joint target is

$$\Pi(dx^{1:K+1}) = \prod_{k=1}^{K+1} \pi(dx^k).$$

The  $K + 1$  chains are simulated on the same processor, one at a time, in a serial schedule. To avoid introducing an index for the currently simulating chain, it is equivalent that chain  $K + 1$  is always the one simulating, but that states are rotated between chains after each jump. Specifically, the state of chain  $K + 1$  at step  $n - 1$  becomes the state of Chain 1 at step  $n$ , and the state of each other chain  $k \in \{1, \dots, K\}$  becomes the state of chain  $k + 1$ . The transition can then be written as

$$X_n^{1:K+1} | x_{n-1}^{1:K+1} \sim \kappa(dx_n^1 | x_{n-1}^{K+1}) \prod_{k=1}^K \delta_{x_{n-1}^k} (dx_n^{k+1}).$$

As before, this joint Markov chain has an associated joint Markov jump process  $(X^{1:K+1}, L)(t)$ , where  $L(t)$  is the lag time elapsed since the last jump. This joint Markov jump process is readily manipulated to yield the following properties, analogous to the single chain case. Proofs are given in Appendix A in Supplementary Material.

**Proposition 5.** *The stationary distribution of the Markov jump process  $(X^{1:K+1}, L)(t)$  is*

$$A(dx^{1:K+1}, dl) = \alpha(dx^{K+1}, dl) \prod_{k=1}^K \pi(dx^k). \tag{1}$$

This result for the joint process extrapolates Proposition 1. To see this, note that the key ingredients of Proposition 1,  $\bar{F}_\tau(l|x)$  and  $\pi(dx)$ , are now replaced with  $\bar{F}_\tau(l|x^{K+1})$  and  $\Pi(dx^{1:K+1})$ . The hold-time survival function for the product chain is  $\bar{F}_\tau(l|x^{K+1})$ , since it related to the time taken to execute the kernel  $\kappa(dx_n^1|x_{n-1}^{K+1})$ , which depends on the state of chain  $K + 1$  only.

**Corollary 6.** *With respect to  $\Pi(dx^{1:K+1})$ ,  $A(dx^{1:K+1})$  is length-biased by expected hold time on the extra state  $X^{K+1}$  only:*

$$A(dx^{1:K+1}) = \alpha(dx^{K+1}) \prod_{k=1}^K \pi(dx^k).$$

**Corollary 7.** *Let  $(X^{1:K+1}, L) \sim A$ , then the conditional probability distribution of  $X^{1:K+1}$  given  $L < \epsilon$  is  $\Pi(dx^{1:K+1})$ .*

We state without proof that the product chain construction also satisfies the ergodic theorem under the same assumptions as Proposition 4. Numerical validation is given in Section 4.1.

The practical implication of these properties is that any MCMC algorithm running  $K \geq 1$  chains can be converted into an anytime MCMC algorithm by interleaving one extra chain. When the computation is interrupted at some time  $t$ , the state of the extra chain is distributed according to  $\alpha$ , while the states of the remaining  $K$  chains are independently distributed according to  $\pi$ . The state of the extra chain is simply discarded to eliminate the length bias.

### 3. Methods

It is straightforward to apply the above framework to design anytime MCMC algorithms. One simply runs  $K + 1$  chains of the desired MCMC algorithm on a single processor, using a serial schedule, and eliminating the state of the extra chain whenever the computation is interrupted at some real time  $t$ . The anytime framework is particularly useful within a broader SMC method (Del Moral et al., 2006), where there already exist multiple chains (particles) with which to establish anytime behavior. We propose appropriate SMC methods in this section.

#### 3.1. Sequential Monte Carlo

We are interested in the context of sequential Bayesian inference targeting a posterior distribution  $\pi(dx) = p(dx|y_{1:V})$  for a given dataset  $y_{1:V}$ . For the purposes of SMC, we assume that the target distribution  $\pi(dx)$  admits a density  $\pi(x)$  in order to compute importance weights.

Define a sequence of target distributions  $\pi_0(dx) = p(dx)$  and  $\pi_v(dx) \propto p(dx|y_{1:v})$  for  $v = 1, \dots, V$ . The first target equals the prior distribution, while the final target equals the posterior distribution,  $\pi_V(dx) = p(dx|y_{1:V}) = \pi(dx)$ . Each target  $\pi_v$  has an associated Markov kernel  $\kappa_v$ , invariant to that target, which could be defined using an MCMC algorithm.

An SMC algorithm propagates a set of weighted samples (particles) through the sequence of target distributions. At step  $v$ , the target  $\pi_v(dx)$  is represented by an empirical approximation  $\hat{\pi}_v(dx)$ , constructed with  $K$  number of samples  $x_v^{1:K}$  and their associated weights  $w_v^{1:K}$ :

$$\hat{\pi}_v(dx) = \frac{\sum_{k=1}^K w_v^k \delta_{x_v^k}(dx)}{\sum_{k=1}^K w_v^k}. \tag{2}$$

A basic SMC algorithm proceeds as in Algorithm 1.

---

**Algorithm 1.** A basic SMC algorithm. Where  $k$  appears, the operation is performed for all  $k \in \{1, \dots, K\}$ .

---

1. Initialize  $x_0^k \sim \pi_0(dx_0)$ .
  2. For  $v = 1, \dots, V$ 
    - (a) Set  $x_v^k = x_{v-1}^k$  and weight  $w_v^k = \pi_v(x_v^k) / \pi_{v-1}(x_v^k) \propto p(y_v | x_v^k, y_{1:v-1})$ , to form the empirical approximation  $\hat{\pi}_v(dx_v) \approx \pi_v(dx_v)$ .
    - (b) Resample  $x_v^k \sim \hat{\pi}_v(dx_v)$ .
    - (c) Move  $x_v^k \sim \kappa_v(dx_v | x_v^k)$  for  $n_v$  steps.
- 

An extension of the algorithm concerns the case where the sequence of target distributions requires marginalizing over some additional latent variables. In these cases, a pseudomarginal approach (Andrieu and Roberts, 2009) can be adopted, replacing the exact weight computations with unbiased estimates (Fearnhead et al., 2008, 2010). For example, for a state-space model at step  $v$ , there are  $v$  hidden states  $z_{1:v} \in \mathbb{Z}^v$  to marginalize over:

$$\pi_v(dx) = \int_{\mathbb{Z}^v} \pi_v(dx, dz_{1:v}).$$

Unbiased estimates of this integral can be obtained by a nested SMC procedure targeting  $\pi_v(dz_{1:v} | x)$ , leading to the algorithm known as SMC<sup>2</sup> (Chopin et al., 2013), where the kernels  $\kappa_v$  are particle MCMC moves (Andrieu et al., 2010). This is used for the example of Section 4.2.

### 3.2. SMC with anytime moves

In the conventional SMC algorithm, it is necessary to choose  $n_v$ , the number of kernel moves to make per particle in Step 2(c). For anytime moves, this is replaced with a real-time budget  $t_v$ . Move steps are typically the most expensive steps—certainly so for SMC<sup>2</sup>—with the potential for significant variability in the time taken to move each particle. An anytime treatment provides control over the budget of the move step which, if it is indeed the most expensive step, provides substantial control over the total budget also.

The anytime framework is used as follows. Associated with each target distribution  $\pi_v$ , and its kernel  $\kappa_v$ , is a hold time distribution  $\tau_v$ , and implied anytime distribution  $\alpha_v$ . At step  $v$ , after resampling, an extra particle and lag  $(x_v^{K+1}, l_v)$  are drawn (approximately) from the anytime distribution  $\alpha_v$ . The real-time Markov jump process  $(X_v^{1:K+1}, L_v)(t)$  is then initialized with these particles and lag, and simulated forward until time  $t_v$  is reached. The extra chain and lag are then eliminated, and the states of the remaining chains are restored as the  $K$  particles  $x_v^{1:K}$ .

The complete algorithm is given in Algorithm 2.

---

**Algorithm 2.** SMC with anytime moves. Where  $k$  appears, the operation is performed for all  $k \in \{1, \dots, K\}$ .

---

1. Initialize  $x_0^k \sim \pi_0(dx_0)$ .
2. For  $v = 1, \dots, V$ 
  - (a) Set  $x_v^k = x_{v-1}^k$  and weight  $w_v^k = \pi_v(x_v^k) / \pi_{v-1}(x_v^k) \propto p(y_v | x_v^k, y_{1:v-1})$ , to form the empirical approximation  $\hat{\pi}_v(dx_v) \approx \pi_v(dx_v)$ .
  - (b) Resample  $x_v^k \sim \hat{\pi}_v(dx_v)$ .



- (c) Draw (approximately) an extra particle and lag  $(x_v^{K+1}, l_v) \sim \alpha_v(dx_v, dl_v)$ . Construct the real-time Markov jump process  $(X_v^{1:K+1}, L_v)(0) = (x_v^{1:K+1}, l_v)$  and simulate it forward for some real time  $t_v$ . Set  $x_v^{1:K} = X_v^{1:K}(t_v)$ , discarding the extra particle and lag.

By the end of the move Step 2(c), as  $t_v \rightarrow \infty$ , the particles  $x_v^{1:K+1}$  become distributed according to  $A_v$ , regardless of their distribution after the resampling Step 2(b). This is assured by Proposition 4. After eliminating the extra particle, the remaining  $x_v^{1:K}$  are distributed according to  $\Pi_v$ .

In practice, of course, it is necessary to choose some finite  $t_v$  for which the  $x_v^{1:K}$  are distributed only approximately according to  $\Pi_v$ . For any given  $t_v$ , their divergence in distribution from  $\Pi_v$  is minimized by an initialization as close as possible to  $A_v$ . We have, already, the first  $K$  chains initialized from an empirical approximation of the target,  $\hat{\pi}_v$ , which is unlikely to be improved upon. We need only consider the extra particle and lag.

An easily-implemented choice is to draw  $(x_v^{K+1}, l_v) \sim \hat{\pi}_v(dx_v)\delta_0(dl_v)$ . In practice, this merely involves resampling  $K + 1$  rather than  $K$  particles in Step 2(b), setting  $l_v = 0$  and proceeding with the first move.

An alternative is  $(x_v^{K+1}, l_v) \sim \delta_{x_{v-1}^{K+1}}(dx_v)\delta_{l_{v-1}}(dl_v)$ . This resumes the computation of the extra particle that was discarded at step  $v - 1$ . As  $l_{v-1} \rightarrow \infty$ , it amounts to approximating  $\alpha_v$  by  $\alpha_{v-1}$ , which is sensible if the sequence of anytime distributions changes only slowly.

### 3.3. Distributed SMC with anytime moves

While the potential to parallelize SMC is widely recognized (see e.g., Lee et al., 2010; Murray, 2015), the resampling Step 2(b) in Algorithm 1 is acknowledged as a potential bottleneck when in a distributed computing environment of  $P$  number of processors. This is due to collective communication: all processors must synchronize after completing the preceding steps in order for resampling to proceed. Resampling cannot proceed until the slowest among them completes. As this is a maximum among  $P$  processors, the expected wait time increases with  $P$ . Recent work has considered either global pairwise interaction (Murray, 2011; Murray et al., 2016) or limited interaction (Vergé et al., 2015; Lee and Whiteley, 2016; Whiteley et al., 2016) to address this issue. Instead, we propose to preserve collective communication, but to use an anytime move step to ensure the simultaneous readiness of all processors for resampling.

SMC with anytime moves is readily distributed across multiple processors. The  $K$  particles are partitioned so that processor  $p \in \{1, \dots, P\}$  has some number of particles, denoted  $K^p$ , and so that  $\sum_{p=1}^P K^p = K$ . Each processor can proceed with initialization, move and weight steps independently of the other processors. After the resampling step, each processor has  $K^p$  number of particles. During the move step, each processor draws its own extra particle and lag from the anytime distribution, giving it  $K^p + 1$  particles, and discards them at the end of the step leaving it with  $K^p$  again. Collective communication is required for the resampling step, and an appropriate distributed resampling scheme should be used (see e.g., Bolić et al., 2005; Vergé et al., 2015; Lee and Whiteley, 2016).

In the simplest case, all workers have homogeneous hardware and the obvious partition of particles is  $K^p = K/P$ . For heterogeneous hardware another partition may be set *a priori* (see e.g., Rosenthal, 2000). Note also that with heterogenous hardware, each processor may have a different compute capability and therefore different distribution  $\tau_v$ . For processor  $p$ , we denote this  $\tau_v^p$  and the associated anytime distribution  $\alpha_v^p$ . This difference between processors is easily accommodated, as the anytime treatment is local to each processor.

A distributed SMC algorithm with anytime moves proceeds as in Algorithm 3.

**Algorithm 3.** SMC with anytime moves. Where  $k$  appears, the operation is performed for all  $k \in \{1, \dots, K^p\}$ .

1. On each processor  $p$ , initialize  $x_0^k \sim \pi_0(dx)$ .

2. For  $v = 1, \dots, V$ 
  - (a) On each processor  $p$ , set  $x_v^k = x_{v-1}^k$  and weight  $w_v^k = \pi_v(x_v^k) / \pi_{v-1}(x_v^k) \propto p(y_v | x_v^k, y_{1:v-1})$ . Collectively, all  $K$  particles form the empirical approximation  $\widehat{\pi}_v(dx_v) \approx \pi_v(dx_v)$ .
  - (b) Collectively resample  $x_v^k \sim \widehat{\pi}_v(dx_v)$  and redistribute the particles among processors so that processor  $p$  has exactly  $K^p$  particles again.
  - (c) On each processor  $p$ , draw (approximately) an extra particle and lag  $(x_v^{K^p+1}, l_v) \sim \alpha_v^p(dx, dl)$ . Construct the real-time process  $(X_v^{1:K^p+1}, L_v)(0) = (x_v^{1:K^p+1}, l_v)$  and simulate it forward for some real time  $t_v$ . Set  $x_v^{1:K^p} = X_v^{1:K^p}(t_v)$ , discarding the extra particle and lag.

The preceding discussion around the approximate anytime distribution still holds *for each processor in isolation*: for any given budget  $t_v$ , to minimize the divergence between the distribution of particles and the target distribution,  $\widehat{A}_v^p$  should be chosen as close as possible to  $A_v^p$ .

### 3.4. Setting the compute budget

We set an overall compute budget for move steps, which we denote  $t$ , and apportion this into a quota for each move step  $v$ , which we denote  $t_v$  as above. This requires some *a priori* knowledge of the compute profile for the problem at hand.

Given  $\widehat{\pi}_{v-1}$ , if the compute time necessary to obtain  $\widehat{\pi}_v$  is constant with respect to  $v$ , then a suitable quota for the  $v$ th move step is the obvious  $t_v = t/V$ . If, instead, the compute time grows linearly in  $v$ , as is the case for SMC<sup>2</sup>, then we expect the time taken to complete the  $v$ th step to be proportional to  $v + c$  (where the constant  $c$  is used to capture overheads). A sensible quota for the  $v$ th move step is then

$$t_v = \left( \frac{v + c}{\sum_{u=1}^V (u + c)} \right) t = \left( \frac{2(v + c)}{V(V + 2c + 1)} \right) t. \tag{3}$$

For the constant, a default choice might be  $c = 0$ ; higher values shift more time to earlier time steps.

This approximation does neglect some complexities. The use of memory-efficient path storage (Jacob et al., 2015), for example, introduces a time-dependent contribution of  $\mathcal{O}(K)$  at  $v = 1$ , increasing to  $\mathcal{O}(K \log K)$  with  $v$  as the ancestry tree grows. Nonetheless, for the example of Section 4.2 we observe, anecdotally, that this partitioning of the time budget produces surprisingly consistent results with respect to the random number of moves completed at each move step  $v$ .

### 3.5. Resampling considerations

To reduce the variance in resampling outcomes (Douc and Cappé, 2005), implementations of SMC often use schemes such as systematic, stratified (Kitagawa, 1996) or residual (Liu and Chen, 1998) resampling, rather than the multinomial scheme (Gordon et al., 1993) with which the above algorithms have been introduced. The implementation of these alternative schemes does not necessarily leave the random variables  $X_v^1, \dots, X_v^K$  exchangeable; for example, the offspring of a particle are typically neighbors in the output of systematic or stratified resampling (see e.g., Murray et al., 2016).

Likewise, distributed resampling schemes do not necessarily redistribute particles identically between processors. For example, the implementation in LibBi (Murray, 2015) attempts to minimize the transport of particles between processors, such that the offspring of a parent particle are more likely to remain on the same processor as that particle. This means that the distribution of the  $K^p$  particles on each processor may have greater divergence from  $\pi_v$  than the distribution of the  $K$  particles overall.

In both cases, the effect is that particles are initialized further from the ideal  $A_v$ . Proposition 4 nonetheless ensures consistency as  $t_v \rightarrow \infty$ . A random permutation of particles may result in a better initialization, but this can be costly, especially in a distributed implementation where particles must be transported between processors. For a fixed total budget, the time spent permuting may be better spent by increasing  $t_v$ . The optimal choice is difficult to identify in general; we return to this point in the discussion.

## 4. Experiments

This section presents two case studies to empirically investigate the anytime framework and the proposed SMC methods. The first uses a simple model where real-time behavior is simulated in order to validate the results of Section 2. The second considers a Lorenz '96 state-space model with nontrivial likelihood and compute profile, testing the SMC methods of Section 3 in two real-world computing environments.

### 4.1. Simulation study

Consider the model

$$\begin{aligned} X &\sim \text{Gamma}(k, \theta) \\ H|x &\sim \text{Gamma}(x^p/\theta, \theta), \end{aligned}$$

with shape parameter  $k$ , scale parameter  $\theta$ , and polynomial degree  $p$ . The two Gamma distributions correspond to  $\pi$  and  $\tau$ , respectively. The anytime distribution is:

$$\alpha(\mathrm{d}x) = \frac{\mathbb{E}_\tau[H|x]}{\mathbb{E}_\tau[H]} \pi(\mathrm{d}x) \propto x^{k+p-1} \exp\left(-\frac{x}{\theta}\right) \mathrm{d}x,$$

which is  $\text{Gamma}(k+p, \theta)$ .

Of course, in real situations,  $\tau$  is not known explicitly, and is merely implied by the algorithm used to simulate  $X$ . For this first study, however, we assume the explicit form above and simulate virtual hold times. This permits exploration of the real-time effects of polynomial computational complexity in a controlled environment, including constant ( $p=0$ ), linear ( $p=1$ ), quadratic ( $p=2$ ) and cubic ( $p=3$ ) complexity.

To construct a Markov chain  $(X_n)_{n=0}^\infty$  with target distribution  $\text{Gamma}(k, \theta)$ , first consider a Markov chain  $(Z_n)_{n=0}^\infty$  with target distribution  $\mathcal{N}(0, 1)$  and kernel

$$Z_n | z_{n-1} \sim \mathcal{N}(\rho z_{n-1}, 1 - \rho^2),$$

where  $\rho$  is an autocorrelation parameter. Now define  $(X_n)_{n=0}^\infty$  as

$$x_n = F_\gamma^{-1}(F_\phi(z_n); k, \theta),$$

where  $F_\gamma^{-1}$  is the inverse cdf of the Gamma distribution with parameters  $k$  and  $\theta$ , and  $F_\phi$  is the cdf of the standard normal distribution. By construction,  $\rho$  parameterizes a Gaussian copula inducing correlation between adjacent elements of  $(X_n)_{n=0}^\infty$ .

For the experiments in this section, we set  $k=2$ ,  $\theta=1/2$ ,  $\rho=1/2$ , and use  $p \in \{0, 1, 2, 3\}$ . In all cases Markov chains are initialized from  $\pi$  and simulated for 200 units of virtual time.

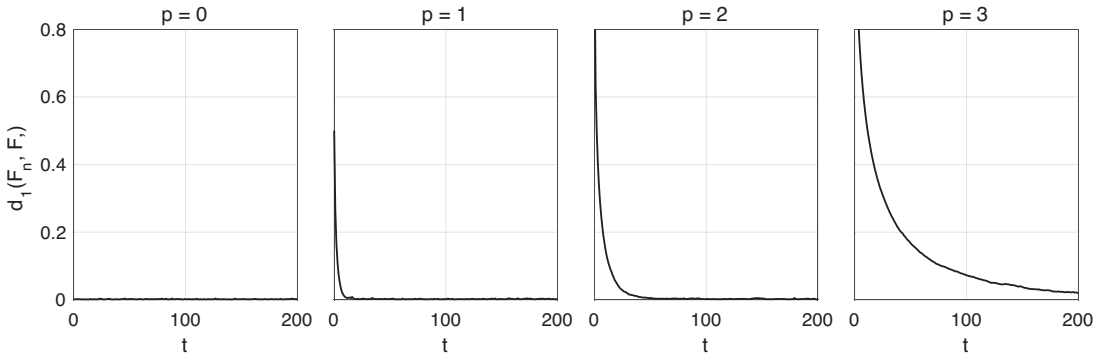
We employ the one-Wasserstein distance to compare distributions. For two univariate distributions  $\mu$  and  $\nu$  with associated cdfs  $F_\mu(x)$  and  $F_\nu(x)$ , the one-Wasserstein distance  $d_1(F_\mu, F_\nu)$  can be evaluated as (Shorack and Wellner, 1986, p. 64)

$$d_1(F_\mu, F_\nu) = \int_{-\infty}^{\infty} |F_\mu(x) - F_\nu(x)| \mathrm{d}x,$$

which, for the purposes of this example, is sufficiently approximated by numerical integration. The first distribution will always be the empirical distribution of a set of  $n$  samples, its cdf denoted  $F_n(x)$ . If those samples are distributed according to the second distribution, the distance will go to zero as  $n$  increases. See Figure 3.

#### 4.1.1. Validation of the anytime distribution

We first validate, empirically, that the anytime distribution is indeed  $\text{Gamma}(k+p, \theta)$  as expected. We simulate  $n=2^{18}$  Markov chains. At each integer time we take the state of all  $n$  chains to construct an



**Figure 3.** Convergence of Markov chains to the anytime distribution for the simulation study, with constant ( $p = 0$ ), linear ( $p = 1$ ), quadratic ( $p = 2$ ), and cubic ( $p = 3$ ) expected hold time. Each plot shows the evolution of the one-Wasserstein distance between the anytime distribution and the empirical distribution of  $2^{18}$  independent Markov chains initialized from the target distribution.

empirical distribution. We then compute the one-Wasserstein distance between this and the anytime distribution, using the empirical cdf  $F_n(x)$  and anytime cdf  $F_a(x)$ .

Figure 2 plots the results over time. In all cases the distance goes to zero in  $t$ , slower for larger  $p$ . This affirms the theoretical results obtained in Section 2.

**4.1.2. Validation of the multiple chain strategy**

We next check the efficacy of the multiple chain strategy in eliminating length bias. For  $K + 1 \in \{2, 4, 8, 16, 32\}$ , we initialize  $K + 1$  chains and simulate them forward in a serial schedule. For  $n = 2^{18}$ , this is repeated  $n / (K + 1)$  times. We then consider ignoring the length bias versus correcting for it. In the first case, we take the states of all  $K + 1$  chains at each time, giving  $n$  samples from which to construct an empirical cdf  $F_n(x)$ . In the second case, we eliminate the extra chain but keep the remaining  $K$ , giving  $nK / (K + 1)$  samples from which to construct an empirical cdf  $F_{nK / (K + 1)}(x)$ . In both cases, we compute the one-Wasserstein distance between the empirical and target distributions, using the appropriate empirical cdf, and the target cdf  $F_\pi(x)$ .

Figure 4 plots the results over time for both the uncorrected (top) and corrected (bottom) cases. For the uncorrected case, the one-Wasserstein distance between the empirical distribution and target distribution does not converge to zero. Neither does it become arbitrarily bad: the distance is due to one of the  $K + 1$  chains being distributed according to  $\alpha$  and not  $\pi$ , the influence of which decreases as  $K$  increases.

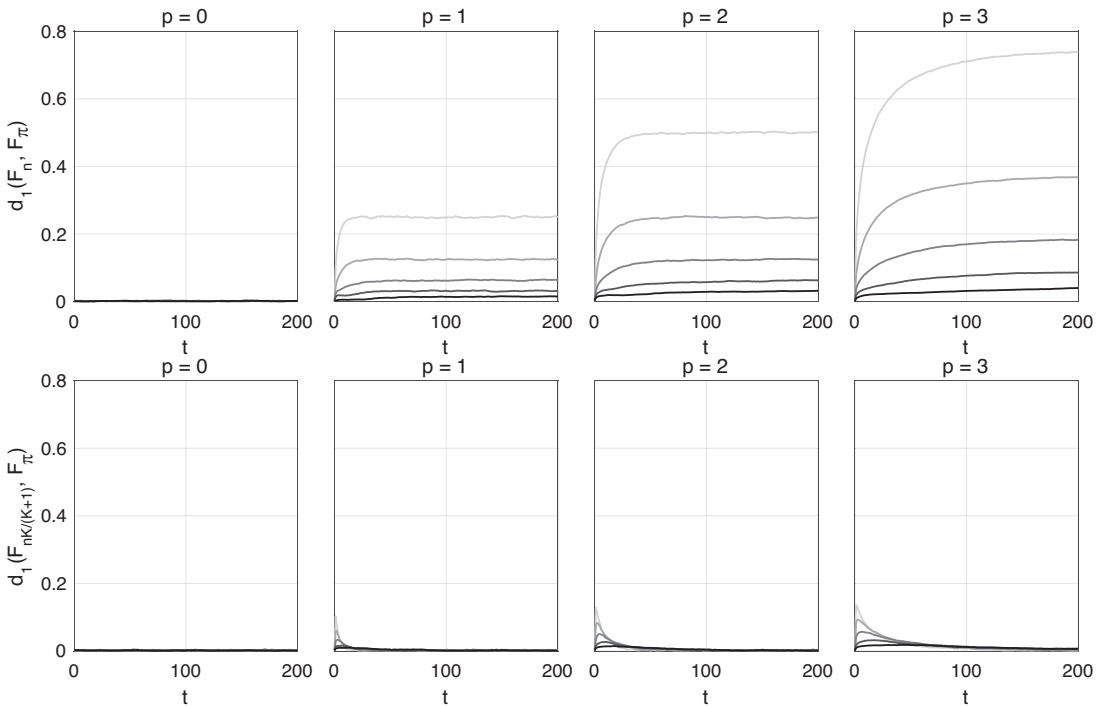
For the corrected case, where the extra chain is eliminated, the distance converges to zero in time. This confirms the efficacy of the multiple chain strategy in yielding an anytime distribution equal to the target distribution.

**4.2. Distributed computing study**

Consider a stochastic extension of the deterministic Lorenz '96 (Lorenz, 2006) model described by the stochastic differential equation (SDE)

$$dX_d = (X_{d-1}(X_{d+1} - X_{d-2}) - X_d + F)dt + \sigma dW_d, \tag{4}$$

with parameter  $F$ , constant  $\sigma$ , state vector  $\mathbf{X}(t) \in \mathbb{R}^D$ , and Wiener process vector  $\mathbf{W}(t) \in \mathbb{R}^D$ , with elements of those vectors indexed cyclically by subscripts (i.e.,  $X_{d-D} \equiv X_d \equiv X_{d+D}$ ). The SDE may be equivalently interpreted in the Ito or Stratonovich sense, as the noise term is additive (Kloeden and Platen, 1992, p. 157). The observation model is given by



**Figure 4.** Correction of length bias for the simulation study, using  $K + 1 \in \{2, 4, 8, 16, 32\}$  chains (light to dark), with constant ( $p = 0$ ), linear ( $p = 1$ ), quadratic ( $p = 2$ ), and cubic ( $p = 3$ ) expected hold time. Each plot shows the evolution of the one-Wasserstein distance between the empirical and target distributions. On the top row, the states of all chains contribute to the empirical distribution, which does not converge to the target. On the bottom row, the state of the extra chain is eliminated, contributing only the remaining states to the empirical distribution, which does converge to the target.

$$Y_d(t) \sim \mathcal{N}(x_d(t), \zeta^2).$$

We fix  $D = 8$ ,  $\sigma^2 = 10^{-4}$ ,  $\zeta^2 = 10^{-6}$  and set a prior on the parameter  $F$  and initial conditions  $\mathbf{X}(0)$  of:

$$F \sim \mathbf{U}([0, 7])$$

$$X_d(0) \sim \mathcal{N}(0, \sigma^2).$$

The SDE can be approximately decomposed into a deterministic drift component given by the ordinary differential equation (ODE)

$$\frac{dx_d}{dt} = x_{d-1}(x_{d+1} - x_{d-2}) - x_d + F,$$

and a diffusion component given by the Wiener process. On a fixed time step  $\Delta t = 5 \times 10^{-2}$ , the drift component is first simulated using an appropriate numerical scheme for ODEs. Then, a Wiener process increment  $\Delta W_d \sim \mathcal{N}(0, \Delta t)$  is simulated and added to the result. This numerical scheme yields a result similar to that of Euler–Maruyama for the original SDE but, for drift, substitutes the usual first-order Euler method with a higher-order Runge–Kutta method. This is advantageous in low-noise regimes where  $\sigma$  is close to zero, as here. In such cases, the dynamics are drift-dominated and can benefit from the higher-order scheme (see e.g., Milstein and Tret'yakov, 2004, chapter 3).

The RK4(3)5[2R+]C algorithm of Kennedy et al. (2000) is used to simulate the drift. This provides a fourth order solution to the ODE with an embedded third-order solution for error estimates. Adaptive step-size adjustment is then used as in Hairer et al. (1993). The complete method is implemented (Murray, 2012) on a graphics processing unit (GPU) as in the LibBi software ([www.libbi.org](http://www.libbi.org); Murray, 2015).

The Lorenz '96 model exhibits intricate qualitative behaviors that depend on the parameter  $F$ . These range from decay, to periodicity, to chaos and back again (Figure 5, top row). With an adaptive step-size adjustment, the number of steps required to simulate trajectories within given error bounds generally increases with  $F$ , so that compute time does also.

We produce a dataset by setting  $F = 4.8801$ , simulating a single trajectory for 10 time units and taking partial observations  $\mathbf{Y}_{1:4}(t)$  every 0.4 time units. This gives 100 observations in total. We then use SMC<sup>2</sup> to attempt to recover the correct posterior distribution over  $F$  given this dataset. This is nontrivial: this particular value of  $F$  is in a region where the qualitative behavior of the Lorenz '96 model appears to switch frequently, in  $F$ , between periodic and chaotic regimes (Figure 5, top right), suggesting that the marginal likelihood may not be smooth in the region of the posterior, and inference may be difficult.

The marginal likelihood  $p(y|F)$  cannot be computed exactly, but it can be unbiasedly estimated with SMC. For each value of  $F$  on a regular grid, we run SMC with  $2^{20}$  particles to estimate the marginal likelihood. These estimates are shown in the middle left of Figure 5. The likelihood is clearly multi-modal, and the estimator heteroskedastic. Nevertheless, the variance in the estimator is tolerable in the region of the posterior distribution (middle right of Figure 5), suggesting that  $F$  can be recovered. The real time taken to compute these estimates is shown in the lower plots of Figure 5. The computations were performed in a random order through the grid points on  $F$  so as to decorrelate  $F$  with any transient exogenous effects on compute time. There appears, in fact, to have been some such effect: note the dotted line of points above the bulk on each plot, suggesting that a subset of runs have been slowed. This is most likely due to a contesting process on the shared server on which these computations were run. As expected, compute time tends to increase in  $F$  (after an initial plateau where other factors dominate). Furthermore, variance appears to increase with  $F$  in the higher regions.

We now run SMC<sup>2</sup> using the LibBi software on two platforms:

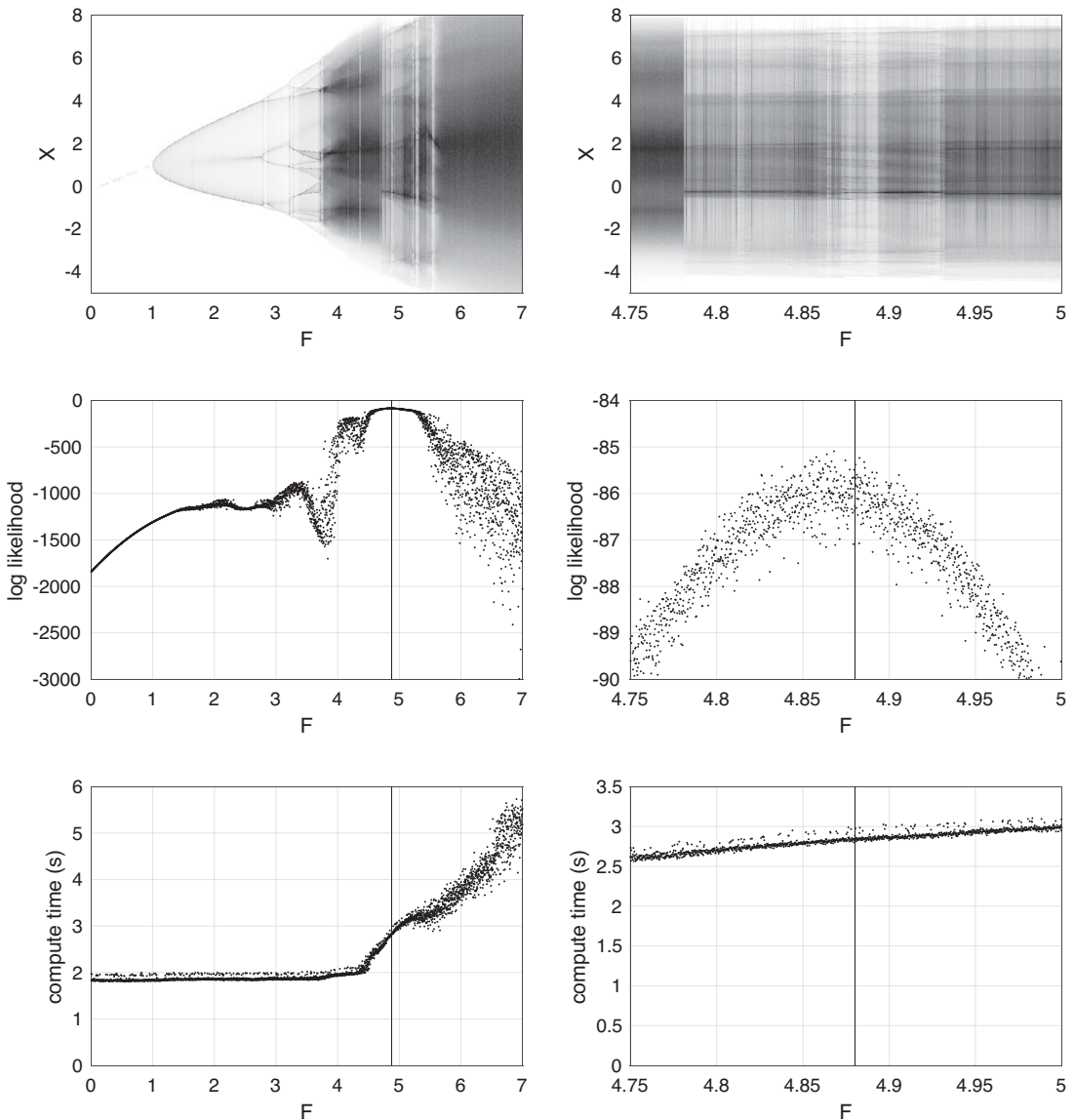
1. A shared-memory machine with 8 GPUs, each with 1,536 cores, for approximately 12,000-way parallelism, using  $2^{10}$  particles for  $F$ , each with  $2^{20}$  particles for  $\mathbf{X}(t)$ , for approximately 1 billion particles overall. This is a shared machine where contestation from other jobs is expected.
2. A distributed-memory cluster on the Amazon EC2 service, with 128 GPUs, each with 1,536 cores, for approximately 200,000-way parallelism, using  $2^{12}$  particles for  $F$ , each with  $2^{20}$  particles for  $\mathbf{X}(t)$ , for approximately 4 billion particles overall. This is a dedicated cluster where contestation from other jobs is not expected.

In order to obtain a more repeatable comparison between conventional SMC<sup>2</sup> and SMC<sup>2</sup> with anytime moves, we choose to use the same number of samples of  $\mathbf{X}(t)$  for all time steps, rather than adapting this in time as recommended in Chopin et al. (2011). For the same reason, we resample at all steps rather than use an adaptive trigger. With anytime moves, the extra particle and lag are drawn as  $(x_v^{K+1}, l_v) \sim \hat{\pi}_v(dx_v)\delta_0(dl_v)$ .

We first run conventional SMC<sup>2</sup>, making  $n_v = 10$  moves per particle at each step  $v$ . We then run SMC<sup>2</sup> with anytime moves, prescribing a total budget for move steps of 60 min for the 8 GPU configuration, and 5 min for the 128 GPU configuration, apportioned as in equation (3).

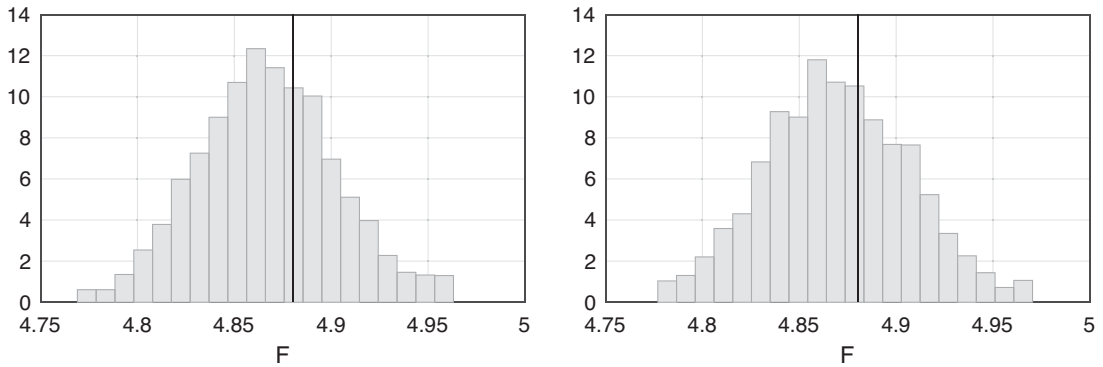
The results of the 128 GPU runs are given in Figure 6. Recalling that  $F = 4.8801$  for the simulated dataset, these suggest that the posterior has indeed been recovered successfully, and there is no indication that the posterior obtained with anytime move steps is much different from that obtained using the conventional method.

Compute profiles for the runs are given in Figure 7, showing the busy and wait times of all processors involved in the computations. We see obvious wait time with conventional SMC<sup>2</sup>, far more pronounced in the eight GPU cases, where a contesting process on one processor has encumbered the entire computation.



**Figure 5.** Elucidating the Lorenz '96 model. The left column shows the range  $F \in [0, 7]$  as in the uniform prior distribution, while the right column shows a narrower range of  $F$  in the vicinity of the posterior distribution. The solid vertical lines indicate the value  $F = 4.8801$ , with which data are simulated. The first row is a bifurcation diagram depicting the stationary distribution of any element of  $\mathbf{X}(t)$  for various values of  $F$ . Each column is a density plot for a particular value of  $F$ ; darker for higher density values, scaled so that the mode is black. Note the intricate behaviors of decay, periodicity, and chaos induced by  $F$ . The second row depicts estimates of the marginal log-likelihood of the simulated dataset for the same values of  $F$ , using SMC with  $2^{20}$  particles. Multiple modes and heteroskedasticity are apparent. The third row depicts the compute time taken to obtain these estimates, showing increasing compute time in  $F$  after an initial plateau.

The anytime move step grants a robustness to this contesting process, and wait times are significantly reduced. For the 128 GPU case, even in the absence of such exogenous problems, wait times are noticeably reduced.



**Figure 6.** Posterior distributions over  $F$  for the Lorenz '96 case study. On the left, from conventional SMC<sup>2</sup>, on the right, from SMC<sup>2</sup> with anytime moves, both running on the 128 GPU configuration.

## 5. Discussion

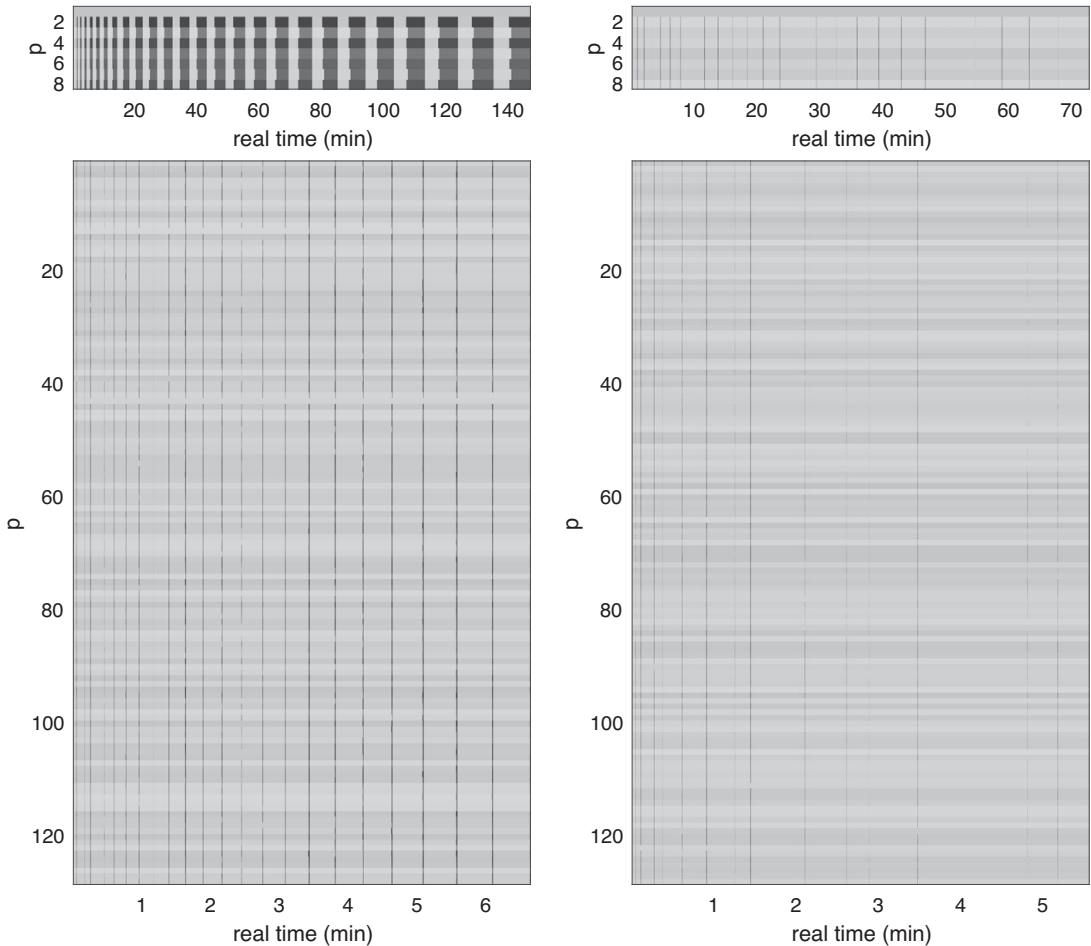
The framework presented is a generic means by which any MCMC algorithm—including iid sampling as a special case—can be made an anytime Monte Carlo algorithm. This facilitates the configuration of Monte Carlo computation in real-time terms, rather than in the number of simulations. The benefits of this have been demonstrated in a distributed computing context where, by setting real-time compute budgets, wait times are significantly reduced for an SMC algorithm that requires collective communication. The framework has potential applications elsewhere, for example as a foundation for real-time, fault-tolerant and energy-constrained Monte Carlo algorithms, for the management of cloud computing budgets, or for the fair computational comparison of methods.

We have assumed throughout that an algorithm is given to simulate the target distribution  $\pi$ , and that the anytime distribution  $\alpha$  is merely a consequence of this. The aim has then been to correct the length bias in  $\alpha$ . This is a pragmatic approach, as it leverages existing Monte Carlo algorithms. A tantalizing alternative is to develop algorithms that, from the outset, yield  $\pi$  as the anytime distribution. This might be done with an underlying Markov chain that targets something other than  $\pi$  but that, by design, yields  $\pi$  once length biased. We expect, however, that to do this even approximately will require at least some knowledge of  $\tau$ , which will restrict its applicability to specific cases only.

The proposed SMC method uses anytime move steps, but is not a complete anytime algorithm, as it does not provide control over the total compute budget. Its objective is to minimize the wait time that precedes resampling steps in a distributed implementation of SMC. On this account, it succeeds. A complete anytime SMC algorithm (of a conventional structure) will require, in addition, anytime weighting and anytime resampling steps, as well as the apportioning of the total compute budget between these. Because approximations may appear in each of these steps, the apportioning is not straightforward, and will involve tradeoffs. As already identified, for example, the redistribution of particles after resampling in a distributed environment is an expensive operation, and all or part of that time may be better invested in the budget allocation for anytime moves. Such investigations have been left to future work. An alternative means to an anytime SMC algorithm is to use a different structure to the conventional, as in the particle cascade (Paige et al., 2014). Whatever the structure, these anytime algorithms are somewhat more elaborate than the standard SMC algorithms for which theoretical results have been established, and may warrant further study.

Finally, we return to the strongest of the assumptions of the whole framework: that of the homogeneity of  $\tau$  in time. This may be unrealistic in the presence of transient exogenous factors, such as intermittent network issues, or contesting processes running on the same hardware only temporarily. If the assumption is relaxed, so that  $\tau$  varies in time, the anytime distribution will vary as well, and ergodicity will not hold. Figure 4 suggests that, for example, an exogenous switching factor in  $\tau$  would induce transient effects in





**Figure 7.** Compute profiles for the Lorenz '96 case study. On the left is a conventional distributed SMC<sup>2</sup> method with a fixed number of moves per particle after resampling. On the right is distributed SMC<sup>2</sup> with anytime move steps. Each row represents the activity of a single processor over time: light gray while active and dark gray while waiting. The top profiles are for an eight GPU shared system where contesting processes are expected. The conventional method on the left exhibits significant idle time on processors 2–8 due to a contesting job on Processor 1. The two bottom profiles are for the 128 GPU configuration with no contesting processes. Wait time in the conventional methods on the left is significantly reduced in the anytime methods on the right.

the anytime distribution that are not necessarily eliminated by the multiple chain strategy. There may be weaker assumptions under which comparable results and appropriate methods can be established, but this investigation is left to future work.

## 6. Conclusion

This work has presented an approach to allow any MCMC algorithm to be made an anytime Monte Carlo algorithm, eliminating the length bias associated with real-time budgets. This is particularly important in situations where the final state of a Markov chain is more important than computing averages over all states. It has applications in embedded, distributed, cloud, real-time, and fault-tolerant computing. To demonstrate the usefulness of the approach, a new SMC<sup>2</sup> method has been

presented, which exhibits significantly reduced wait time when run on a large-scale distributed computing system.

**Acknowledgments.** The authors would like to thank the Isaac Newton Institute for Mathematical Sciences, Cambridge, for support and hospitality during the program *Monte Carlo Methods for Complex Inference Problems* (MCMW01), where some of this work was undertaken. This work was also financially supported by an EPSRC-Cambridge Big Data Travel Grant and EPSRC (EP/K020153/1), The Alan Turing Institute under the EPSRC grant EP/N510129/1, and the Swedish Foundation for Strategic Research (SSF) via the project *ASSEMBLE*. The authors would like to thank Pierre Jacob for helpful conversations.

**Competing Interests.** The authors declare no competing interests exist.

**Data Availability Statement.** The methods introduced in this paper are implemented in LibBi version 1.3.0, and the empirical results may be reproduced with the LibBi *Anytime* package. Both are available from [www.libbi.org](http://www.libbi.org).

**Author Contributions.** Conceptualization, All; Methodology, All; Formal analysis, All; Data curation, L.M.M.; Writing-original draft, L.M.M. and S.S.S.; Writing-review & editing, All; Funding acquisition, All.

**Supplementary Materials.** To view supplementary material for this article, please visit <http://dx.doi.org/10.1017/dce.2021.6>.

## References

- Alsmeyer G (1994) On the Markov renewal theorem. *Stochastic Processes and their Applications* 50, 37–56.
- Alsmeyer G (1997) The Markov renewal theorem and related results. *Markov Processes and Related Fields* 3, 103–127.
- Andrieu C, Doucet A and Holenstein R (2010) Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society B* 72, 269–302. <http://dx.doi.org/10.1111/j.1467-9868.2009.00736.x>.
- Andrieu C and Roberts GO (2009) The pseudo-marginal approach for efficient Monte Carlo computations. *Annals of Statistics*, 37 (2), 697–725. <https://doi.org/10.1214/07-AOS574>.
- Bolić M, Djurić PM and Hong S (2005) Resampling algorithms and architectures for distributed particle filters. *IEEE Transactions on Signal Processing* 53, 2442–2450. <http://dx.doi.org/10.1109/TSP.2005.849185>.
- Chopin N, Jacob P and Papaspiliopoulos O (2011) SMC<sup>2</sup>: A sequential Monte Carlo algorithm with particle Markov chain Monte Carlo updates. <https://hal.archives-ouvertes.fr/hal-00634215>
- Chopin N, Jacob P and Papaspiliopoulos O (2013) SMC<sup>2</sup>: An efficient algorithm for sequential analysis of state space models. *Journal of the Royal Statistical Society B* 75, 397–426. <http://dx.doi.org/10.1111/j.1467-9868.2012.01046.x>.
- d’Avigneau A, Singh S and Murray L (2020) Anytime parallel tempering. *arXiv.org e-Print archive*.
- De Sa C, Olukotun K and Ré C (2016) Ensuring rapid mixing and low bias for asynchronous Gibbs sampling.
- Del Moral P, Doucet A and Jasra A (2006) Sequential Monte Carlo samplers. *Journal of the Royal Statistical Society B* 68, 441–436. <http://dx.doi.org/10.1111/j.1467-9868.2006.00553.x>.
- Douc R and Cappé O (2005) Comparison of resampling schemes for particle filtering. *Image and Signal Processing and Analysis, 2005. ISPA 2005. Proceedings of the 4th International Symposium on*, pp. 64–69.
- Fearnhead P, Papaspiliopoulos O and Roberts GO (2008) Particle filters for partially observed diffusions. *Journal of the Royal Statistical Society B* 70, 755–777. <http://dx.doi.org/10.1111/j.1467-9868.2008.00661.x>.
- Fearnhead P, Papaspiliopoulos O, Roberts GO and Stuart A (2010) Random-weight particle filtering of continuous time processes. *Journal of the Royal Statistical Society B* 72(4), 497–512. <http://dx.doi.org/10.1111/j.1467-9868.2010.00744.x>.
- Glynn PW and Heidelberger P (1990) Bias properties of budget constraint simulations. *Operations Research* 38(5), 801–814.
- Glynn PW and Heidelberger P (1991) Analysis of parallel replicated simulations under a completion time constraint. *ACM Transactions on Modeling and Computer Simulations* 1(1), 3–23.
- Gordon N, Salmund D and Smith A (1993) Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEE Proceedings-F*, 140, 107–113. <https://doi.org/10.1049/ip-f-2.1993.0015>.
- Hairer E, Nørsett S and Wanner G (1993) *Solving Ordinary Differential Equations I: Nonstiff Problems*, 2nd Edn. Berlin: Springer-Verlag.
- Heidelberger P (1988) Discrete event simulations and parallel processing: Statistical properties. *SIAM Journal on Scientific and Statistical Computing* 9(6), 1114–1132. <http://dx.doi.org/10.1137/0909077>.
- Jacob PE, Murray LM and Rubenthaler S (2015) Path storage in the particle filter. *Statistics and Computing* 25(2), 487–496. <https://doi.org/10.1007/s11222-013-9445-x>.
- Kennedy CA, Carpenter MH and Lewis RM (2000) Low-storage, explicit Runge–Kutta schemes for the compressible Navier–Stokes equations. *Applied Numerical Mathematics* 35, 177–219.
- Kitagawa G (1996) Monte Carlo filter and smoother for non-Gaussian nonlinear state space models. *Journal of Computational and Graphical Statistics* 5, 1–25. <http://dx.doi.org/10.2307/1390750>.
- Kloeden PE and Platen E (1992) *Numerical Solution of Stochastic Differential Equations*. Berlin: Springer-Verlag.
- Lee A and Whiteley N (2016) Forest resampling for distributed sequential Monte Carlo. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 9(4), 230–248. <https://doi.org/10.1002/sam.11280>.

- Lee A, Yau C, Giles MB, Doucet A and Holmes CC** (2010) On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods. *Journal of Computational and Graphical Statistics* 19, 769–789. <http://dx.doi.org/10.1198/jcgs.2010.10039>.
- Liu JS and Chen R** (1998) Sequential Monte-Carlo methods for dynamic systems. *Journal of the American Statistical Association* 93, 1032–1044.
- Lorenz EN** (2006) *Predictability of Weather and Climate, Chapter 3: Predictability—A Problem Partly Solved*. Cambridge: Cambridge University Press, pages 40–58. <https://doi.org/10.1017/CBO9780511617652.004>.
- Meeds T and Welling M** (2015) Optimization Monte Carlo: Efficient and embarrassingly parallel likelihood-free inference. In Cortes C, Lawrence ND, Lee DD, Sugiyama M and Garnett R (eds), *Advances in Neural Information Processing Systems*, vol. 28. Red Hook, NY: Curran Associates, Inc., pp. 2080–2088.
- Milstein GN and Tretyakov MV** (2004) *Stochastic Numerics for Mathematical Physics. Scientific Computation*. Berlin: Springer-Verlag. <http://dx.doi.org/10.1007/978-3-662-10063-9>.
- Murray LM** (2011) GPU acceleration of the particle filter: The metropolis resampler. In *DMMD: Distributed Machine Learning and Sparse Representation with Massive Data Sets*. <http://arxiv.org/abs/1202.6163>.
- Murray LM** (2012) GPU acceleration of Runge–Kutta integrators. *IEEE Transactions on Parallel and Distributed Systems* 23, 94–101. <http://dx.doi.org/10.1109/TPDS.2011.61>.
- Murray LM** (2015) Bayesian state-space modelling on high-performance hardware using LibBi. *Journal of Statistical Software* 67 (10), 1–36. <https://doi.org/10.18637/jss.v067.i10>.
- Murray LM, Lee A and Jacob PE** (2016) Parallel resampling in the particle filter. *Journal of Computational and Graphical Statistics* 25(3), 789–805. <http://dx.doi.org/10.1080/10618600.2015.1062015>.
- Newton MA and Raftery AE** (1994) Approximate Bayesian inference with the weighted likelihood bootstrap. *Journal of the Royal Statistical Society B* 56(1), 3–48.
- Paige B, Wood F, Doucet A and Teh YW** (2014) Asynchronous anytime sequential Monte Carlo. *Advances in Neural Information Processing Systems* 27, 3410–3418.
- Ramos FT and Cozman FG** (2005) Anytime anyspace probabilistic inference. *International Journal of Approximate Reasoning*, 38(1), 53–80. <https://doi.org/10.1016/j.ijar.2004.04.001>.
- Rosenthal JS** (2000) Parallel computing and Monte Carlo algorithms. *Far East Journal of Theoretical Statistics* 4, 207–236.
- Shorack GR and Wellner JA** (1986) *Empirical Processes with Applications to Statistics*. New York: Wiley.
- Terenin A, Simpson D and Draper D** (2015) Asynchronous Gibbs sampling. <https://arxiv.org/abs/1509.08999>.
- Vergé C, Dubarry C, Del Moral P and Moulines E** (2015) On parallel implementation of sequential Monte Carlo methods: The island particle model. *Statistics and Computing* 25(2), 243–260. <https://doi.org/10.1007/s11222-013-9429-x>.
- Whiteley N, Lee A and Heine K** (2016) On the role of interaction in sequential Monte Carlo algorithms. *Bernoulli* 22(1), 494–529. <http://dx.doi.org/10.3150/14-BEJ666>.