Noname manuscript No.
(will be inserted by the editor)

Anytime Parallel Tempering for Approximate Bayesian Computation

Alix Marie d'Avigneau · Sumeetpal Singh · Lawrence M. Murray

Received: date / Accepted: date

Abstract The development of efficient and scalable Markov chain Monte Carlo (MCMC) algorithms is indispensable in the Bayesian inference literature. Indeed, they are the methods best equipped to deal with the ever increasing complexity of models. However, computing the likelihood of the data for these models is often prohibitively costly, or outright impossible. Solutions include the simulation-based class of algorithms known as Approximate Bayesian computation (ABC), which provides a likelihood-free approach to inference. Those algorithms are however often subject to slow mixing. To improve their performance, a possible solution is parallel tempering (PT), which provides a more efficient exploration of the state space. For that, multiple Markov chain replicas are updated individually in parallel, and exchange moves are performed between the chains. Another problem is then encountered: all chains must be simultaneously ready for exchange moves to occur, and the real time taken for computations to complete may vary between chains. To solve this problem, an Anytime Monte Carlo framework is proposed: to impose real-time deadlines on the parallel computations. The resulting algorithm is adapted to ABC and applied to a parameter estimation problem for a stochastic Lotka-Volterra predator-prey model − where the likelihood cannot be computed − to demonstrate the improvements in performance it provides.

Keywords Bayesian inference · Markov chain Monte Carlo (MCMC) · parallel tempering · anytime Monte Carlo (AMC) · Approximate Bayesian computation (ABC) · big data.

A. Marie d'Avigneau
University of Cambridge
E-mail: agem2@cam.ac.uk

S. Singh
University of Cambridge
E-mail: sss40@cam.ac.uk

L. M Murray
Uber AI
E-mail: lawrence.murray@uber.com

1 Introduction

Nowadays, in the age of big data, models devised to accurately represent the dynamics of data generating processes become more and more complex and high-dimensional. To deal with such problems, the widely applicable *Markov Chain Monte Carlo* (MCMC) algorithms are the best suited methods.

Consider a set of m observations $y = \{y_1, \dots, y_m\} \subset \mathbb{R}^m$ following a probability model with underlying parameters $\theta \in \Theta$ and associated *likelihood* $f(y \,|\, \theta) = f(y_1, \dots, y_m \,|\, \theta)$. The parameters θ describing the data y are unknown and considered random variables with *prior* density $p(\mathrm{d}\theta)$. The aim of Bayesian inference is to use the information contained in the prior $p(\mathrm{d}\theta)$ − representing prior belief − and the likelihood $f(y \,|\, \theta)$ − representing evidence from the data − to obtain the *posterior* density $\pi(\mathrm{d}\theta)$ of the parameters θ, following Equation 1, where the symbol $\propto$ represents proportionality up to a constant. Summary statistics such as parameter estimates and credible intervals can subsequently be inferred from the posterior obtained.

$$\pi(\mathrm{d}\theta) \propto p(\mathrm{d}\theta) f(y \,|\, \theta) \tag{1}$$

In most cases however, the posterior π is intractable and must be approximated using computational tools

such as MCMC algorithms. A commonly used and easily adaptable MCMC algorithm is the *Metropolis-Hastings* (M-H) algorithm, described in Robert and Casella [2004]. The Metropolis-Hastings algorithm starts at user-defined state θ_0. At the n-th iteration, given current state θ_n, a new candidate $\theta' \sim q(\mathrm{d}\theta' \,|\, \theta)$ is proposed, where q is the proposal density. Then, set $\theta_{n+1} = \theta'$ with probability

$$\alpha\left(\theta_n, \theta'\right) = \min\left\{1, \frac{\pi\left(\theta'\right)\, q\left(\theta_n \,|\, \theta'\right)}{\pi\left(\theta_n\right)\, q\left(\theta' \,|\, \theta_n\right)}\right\}$$

otherwise retain $\theta_{n+1} = \theta_n$. Following these steps, a Markov chain is constructed whose stationary or invariant distribution is π. This enables one to sample from and hence provide an empirical approximation of the posterior.

One of main drawbacks of MCMC algorithms is the computational cost due to the fact that they require evaluating a likelihood function $f(y \,|\, \theta)$ for the full dataset. From this, two problems may arise: the likelihood may be intractable, and it may be too computationally costly to evaluate the likelihood, either because it is too complex or because the datasets are too large. In recent years, many methods have been devised to tackle these issues and increase the efficiency of MCMC algorithms. These include divide-and-conquer methods, which reduce computing time by dividing the data into batches, which are then updated in parallel before combining their result to obtain an approximation of the posterior. They also include subsampling algorithms, which speed up computations by reducing the amount of likelihood evaluations that occur at each iteration. Simulation-based methods such as Approximate Bayesian computation (ABC), explored in this paper, have also been devised to avoid computing the likelihood altogether. The use of distributed computing has the potential to greatly speed up computations by performing several tasks in parallel on multiple processors. However, running multiple inefficient chains in parallel on multiple processors might increase the overall output, but won't improve their mixing. A particular method that is compatible with distributed computing and improves mixing is parallel tempering. It allows for a more efficient exploration of the sample space by introducing exchange moves between multiple Markov chains running in parallel, targeting different temperatures of the posterior.

It should nonetheless be noted that when dealing with distributed computing, real-time budgets arise. For example, the use of cloud computing is often financially costly and users have to deal with factors such as processor hardware, memory bandwidth, network traffic, I/O load, competing jobs on the same processors as well as potential unforeseen interruptions due to e.g. system failures, all of which affect the compute time of algorithms. Because of this, the anytime Monte Carlo framework was devised in Murray et al. [2016] to provide control over the total compute budget of Monte Carlo algorithms and ensure it is respected. This includes ensuring that all processors are simultaneously ready before they communicate instead of staying idle until the last of them has completed computations. In this paper, we combine parallel tempering with the anytime Monte Carlo framework to create Anytime Parallel Tempering Monte Carlo (APTMC) algorithm. While parallel tempering provides an increase in efficiency, the anytime framework essentially provides control over the budget of the parallel tempering algorithm and eliminates the potential bottleneck. An application of the algorithm to Approximate Bayesian computation subsequently provides a taster of its benefits in situations where the likelihood function is either unavailable or too computationally costly.

This paper is structured as follows. First, Section 2 offers a short review of existing scalable MCMC methods, including overviews of the parallel tempering algorithm and the anytime Monte Carlo framework. Section 3 introduces the focus of this report: the Anytime Parallel Tempering Monte Carlo algorithm. Then, an application of the algorithm to ABC is provided in Section 4 and applied to the problem of estimating the parameters of a stochastic Lotka-Volterra predator-prey model, in which the likelihood is unavailable.

2 Background

In recent years, there have been many efforts to develop increasingly efficient or scalable MCMC algorithms, and in this chapter we will review a few examples, spending more time on the methods directly relevant to the main algorithm presented in this paper.

2.1 Overview of scalable MCMC methods

Comprehensive reviews of existing methods that aim to scale up the Metropolis-Hastings algorithm for big data applications – so when the computation of the likelihood is too costly – are available in Robert et al. [2018] and Bardenet et al. [2015]. The authors of Bardenet et al. [2015] divide the available approaches into

two categories: divide-and-conquer and subsampling methods.

First of all, *divide-and-conquer* methods aim to divide the data into batches, then run the MCMC algorithm on each batch separately, usually in parallel, before combining the subposteriors obtained to form an approximation of the full posterior. Examples include works by Neiswanger et al. [2013], Wang and Dunson [2013], Xu et al. [2014] and Minsker et al. [2014]. The main issues these methods must address are how to keep communication between batches minimal, and how to efficiently combine the subposteriors. They do however have an important advantage, which is the possibility of running the algorithm on multiple processors, and thus significantly speeding up computations. This is the case of the consensus Monte Carlo algorithm presented by Scott et al. [2016].

On the other hand, *subsampling* methods aim to reduce the number of likelihood evaluations to speed up computations. This approach is related to *pseudo-marginal* MCMC methods, which employ unbiased estimators of the unnormalised target distribution. This means in this case that only a subsample of the data is used at each iteration, which speeds up computations. A few examples of subsampling MCMC algorithms are the Bootstrap Metropolis-Hastings algorithm by Liang et al. [2016] and the confidence sampler developed by Bardenet et al. [2014] and extended in Bardenet et al. [2015] and in Kohn et al. [2016]. More examples of subsampling MCMC are available in Quiroz et al. [2016] and Korattikara et al. [2014]. It is also possible to use delayed acceptance MCMC, such as the Firefly algorithm in Maclaurin and Adams [2014]. The advantage of delayed acceptance MCMC is that it avoids computation of the likelihood if there is evidence that the proposal will be rejected, however in general it will compute the likelihood on the full dataset otherwise, which is not ideal when the likelihood itself is too computationally costly. This is avoided in Quiroz et al. [2017], where delayed acceptance is combined with subsampling.

Other methods to increase the efficiency of MCMC algorithms include parallelising the Metropolis-Hastings algorithm for use on multiple processors, as described in Calderhead [2014], and a likelihood-free approach to MCMC in the form of Approximate Bayesian computation (ABC) is detailed in Section 4.1. Another category of algorithms called Hamiltonian Monte Carlo (HMC), was introduced in Duane et al. [1987] and further developed in MacKay and Mac Kay [2003], Neal [2012] and Neal [2011]. To improve mixing, it employs approximate Hamiltonian dynamics, and at each iteration makes gradient-based proposals that are more efficient compared to the often used Gaussian random-walk. A comprehensive introduction to Hamiltonian Monte Carlo is available in Betancourt [2017]. Parallel tempering, as described in Section 2.2, also improves mixing by introducing tempering to provide a more efficient exploration of the state space. These types of algorithms are particularly useful when the target distribution is multimodal.

2.2 Parallel Tempering

The notion of *Parallel Tempering* (PT) was initially proposed by Swendsen and Wang [1986] and further developed under the name Metropolis-coupled Markov chain Monte Carlo (MC)^3 by Geyer [1991]. A parallel tempering algorithm allows for steps of various sizes to be made when exploring the parameter space, and is particularly effective when the distribution we wish to sample from has multiple modes. Consider the (global) Markov chain $\left(X^{1:\Lambda}\right)_{n=1}^{\infty} = \left(X^1, \ldots, X^\Lambda\right)_{n=1}^{\infty}$ with initial state $\left(X_0^{1:\Lambda}\right)$ and target distribution

$$\pi(\mathrm{d}x^{1:\Lambda}) \propto \prod_{\lambda=1}^{\Lambda} \pi_\lambda(\mathrm{d}x^\lambda)$$

where the $\pi_\lambda(\cdot)$ are asymptotically independent marginals corresponding to the target distribution of each of Λ chains, running in parallel at different temperatures indexed by λ. One of these chains is the *cold* chain and its target distribution $\pi_\lambda = \pi$ is the posterior of interest. In parallel tempering, Geyer [2011] identifies two types of update that can be performed at each iteration:

1. *Within-component update* or *local moves*: generally a standard Gibbs or Metropolis-Hastings update applied to each tempered chain X^λ in parallel. The local moves can also be performed sequentially.
2. *Between-component update*, or *exchange moves*: propose to swap the states $x \sim \pi_\lambda$ and $x' \sim \pi_{\lambda'}$ of one or more pairs of adjacent chains. For each pair, accept a swap with probability

$$\min\left\{1, \frac{\pi_\lambda(x')\pi_{\lambda'}(x)}{\pi_\lambda(x)\pi_{\lambda'}(x')}\right\} \tag{2}$$

otherwise, the chains in the pair retain their current states. With the cold chain providing more precision and the warmer chains more freedom of movement when exploring the parameter space, the combination

of the two types of update allows all chains to mix much faster than any one of them would mix on its own. This provides a way to jump from mode to mode in far fewer steps than would be required under a standard Metropolis-Hastings algorithm.

A particular advantage of parallel tempering is that it is possible to perform the local moves in parallel on multiple processors. However, these must be synchronised before exchange moves can be performed. This means that all processors must be idle until the slowest of them has completed its set of local moves. The next section introduces the anytime framework which reduces this idle time and eliminates the ensuing bottleneck.

2.3 Anytime Monte Carlo

Generally, Monte Carlo algorithms aim to simulate a pre-determined number of samples, taking a random amount of time to complete the necessary computations. The *Anytime Monte Carlo* (`AMC`) framework, developed in Murray et al. [2016], considers instead the situation in which we impose a real-time deadline on computations such that it is the number of samples that is random. The main appeal of the anytime framework is its application to distributed computing. Recalling parallel tempering, when one performs the local moves on multiple processors, they must be synchronised − meaning that all processors must wait for the slowest of them to complete − before local moves can be performed. By fixing a real-time budget on all processors, this waiting time can be controlled and the overall algorithm is rendered more computationally efficient.

The real time taken to draw each sample may depend on the states of the Markov chain. For example, Section 4 deals with Approximate Bayesian computation (ABC), in which a 'race' takes place to determine the next sample in the Markov chain. Parameters with higher likelihood will be accepted sooner and so yield a lower computation time. More generally, the samples may not be independent of their number and a length bias with respect to computation time becomes apparent. This bias diminishes with time, and when an empirical approximation or average over all post burn-in samples is required, it may be rendered negligible for a long enough computation. However, the bias in the final state does not diminish with time, and when this final state is important − which is the case in parallel tempering − the bias cannot be avoided by running the algorithm for longer, and other methods must be devised to correct it.

Let $(X)_{n=0}^{\infty}$ be a Markov chain with initial state X_0, evolving on state space $\mathbb{X}$, with transition kernel $X_n \,|\, x_{n-1} \sim \kappa(\mathrm{d}x|x_{n-1})$ and target distribution $\pi(\mathrm{d}x)$. Define the *hold time* H_{n-1} as the random and positive real time required to complete the computations necessary to transition from state X_{n-1} to X_n via the kernel κ. Then let $H_{n-1} \,|\, x_{n-1} \sim \tau(\mathrm{d}h_{n-1}|x_{n-1})$ where τ is the hold time distribution.

Given assumptions that the hold time $H > \epsilon > 0$ for minimal time ϵ, we have $\sup_{x\in\mathbb{X}} \mathbb{E}\left[H \,|\, x\right] < \infty$, and the hold time distribution τ is homogeneous in time, it is possible to construct a Markov jump process $(X, L)\,(t)$ with stationary distribution

$$\alpha(\mathrm{d}x, \mathrm{d}l) = \frac{\bar{F}_\tau\left(l \,|\, x\right)}{\mathbb{E}\left[H\right]}\pi(\mathrm{d}x)\mathrm{d}l$$

where $F_\tau\left(l \,|\, x\right)$ is the cdf of $\tau(\mathrm{d}h_n|x_n)$ and $\bar{F}_\tau\left(l \,|\, x\right) = 1 - F_\tau\left(l \,|\, x\right)$, and of which the marginal is

$$\alpha(\mathrm{d}x) = \frac{\mathbb{E}\left[H \,|\, x\right]}{\mathbb{E}\left[H\right]}\pi(\mathrm{d}x) \tag{3}$$

The distribution α is referred to as the *anytime distribution*. When interrupted at real time t, the state of a Monte Carlo computation targeting π is distributed according to the anytime distribution α, which can essentially be seen as a length biased target distribution.

Different situations can be established to correct this bias. The main idea is to make it so expected hold time is independent of X, which leads to $\mathbb{E}\left[H \,|\, x\right] = \mathbb{E}\left[H\right]$ and hence $\alpha(\mathrm{d}x) = \pi(\mathrm{d}x)$, following Equation 3. While this is trivially the case for iid sampling, non-iid sampling requires that for $K > 0$, we simulate $K + 1$ Markov chains, each targeting π using the same transition kernel κ and hold time distribution τ. By simulating these $K + 1$ chains on the same processor in a serial schedule, we ensure that whenever the real-time deadline t is reached, states from all but one of the chains, say the $(K+1)$th chain, are independently distributed according to π. Since the $(K + 1)$th chain is the currently working chain, i.e. the latest to go through the simulation process, its state at the real-time deadline is distributed according to α. Simply discarding or ignoring the state of this $(K + 1)$th chain eliminates the length bias.

Using this multiple chain construction, it is thus possible draw samples from π by interrupting the process at any time t. This and Section 2.2 set the basis for the focus of this paper: the Anytime Parallel Tempering Monte Carlo algorithm, described next.

3 Introduction to Anytime Parallel Tempering Monte Carlo

3.1 Overview

Consider the problem in which we wish to sample from target distribution $\pi(\mathrm{d}x)$. In a parallel tempering framework, construct Λ Markov chains where each individual chain λ targets the tempered distribution

$$\pi_\lambda(\mathrm{d}x) = \pi(\mathrm{d}x)^{\frac{\lambda}{\Lambda}}$$

and is associated with kernel $\kappa_\lambda(\mathrm{d}x_n \,|\, \mathrm{d}x_{n-1})$ and hold time distribution $\tau_\lambda(\mathrm{d}h_n \,|\, x_n)$. For each chain λ, a real-time Markov jump process $(X^\lambda, L)(t)$ can be constructed targeting the anytime distribution $\alpha_\lambda(\mathrm{d}x, \mathrm{d}l)$. In addition to constructing an anytime Monte Carlo algorithm, we aim to temporarily interrupt the computations on a real-time schedule of times $t_1, t_2, t_3, \ldots$ to perform exchange moves between adjacent pairs of chains before resuming.

3.2 Exchange moves

3.2.1 Exchanging on α

We propose to swap the states $(x, l) \sim \alpha_\lambda$ and $(x', l') \sim \alpha_{\lambda'}$ where the chains λ and λ' are adjacent. Similarly to Equation 2, this swap is accepted with probability

$$\frac{\alpha_\lambda(x', l')\, \alpha_{\lambda'}(x, l)}{\alpha_\lambda(x, l)\, \alpha_{\lambda'}(x', l')} \tag{4}$$

Assume that the hold time distributions are temperature-homogeneous, such that $\tau_\lambda(\mathrm{d}h_n \,|\, x_n) = \tau(\mathrm{d}h_n \,|\, x_n)$ for all $\lambda = 1, \ldots, \Lambda$. As a direct consequence, we also have $\bar{F}_\lambda(l \,|\, x) = \bar{F}(l \,|\, x)$ for all $\lambda = 1, \ldots, \Lambda$, i.e. the hold time cdfs do not change with temperature. Therefore, under this assumption, the expression in Equation 4 simplifies to the standard exchange probability for $x \sim \pi_\lambda$ and $x' \sim \pi_{\lambda'}$ given in Equation 2, and standard parallel tempering algorithms are actually implementing exchange moves on the corresponding tempered anytime distributions.

However, when exchanging on α, the (current) final state of each Markov chain is used, meaning that there is still a length bias present, such that

$$\mathbb{E}\left\{\frac{1}{n}\sum_{i=1}^{n} \zeta\left(X_i^\Lambda\right)\right\} \neq \int \zeta(x)\pi_\Lambda(\mathrm{d}x)$$

where ζ is an arbitrary function, π_Λ is the desired (cold) distribution and X_Λ^i for $i = 1, \ldots, n$ is the sampled trajectory, post burn-in, of the (cold) α_Λ chain. This sample bias diminishes as n increases, and would easily become negligible if the whole sampled trajectory, rather than the final state, was required here, or if exchange moves didn't occur often enough compared to local moves.

3.2.2 Exchanging on π

Here, we adapt the multi-chain construction devised to remove the bias present when sampling from Λ Markov chains, where each chain λ targets the distribution π_λ for $\lambda = 1, \ldots, \Lambda$. Associated with each chain is MCMC kernel $\kappa_\lambda(\mathrm{d}x_n^\lambda \,|\, \mathrm{d}x_{n-1}^\lambda)$ and hold time distribution $\tau_\lambda(\mathrm{d}h \,|\, x)$. The corresponding joint anytime distribution is

$$A(\mathrm{d}x^{1:\Lambda}, \mathrm{d}l, j) = \frac{1}{\Lambda}\left(\alpha_j(\mathrm{d}x^j, \mathrm{d}l) \prod_{\lambda=1,\, \lambda\neq j}^{\Lambda} \pi_\lambda(\mathrm{d}x^\lambda)\right) \frac{\mathbb{E}\left[H \,|\, j\right]}{\mathbb{E}\left[H\right]}$$

for $\lambda = 1, \ldots, \Lambda$. As updates are performed sequentially, j represents here the chain being updated while the other ones are not. Conditioning on x^j, j and l we obtain

$$A(\mathrm{d}x^{1:\Lambda\setminus j} \,|\, x^j, l, j) = \prod_{\lambda=1,\, \lambda\neq j}^{\Lambda} \pi_\lambda(\mathrm{d}x^\lambda) \tag{5}$$

Therefore, if exchange moves on the conditional $A(\mathrm{d}x^{1:\Lambda\setminus j} \,|\, x^j, l, j)$ are performed by 'eliminating' the j-th chain to obtain the expression in Equation 5, they are being performed involving only chains distributed according to π and thus the bias is eliminated.

3.3 Implementation

3.3.1 One processor

On a single processor, the algorithm may proceed as in Algorithm 1, where in Step 3 the Λ chains are simulated one at a time in a serial schedule. Figure 1 provides an illustration of how the algorithm works.

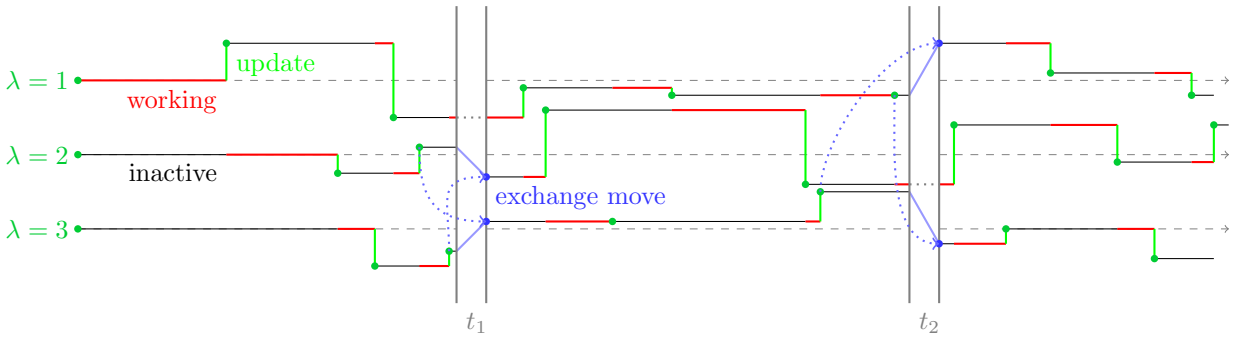

Fig. 1 Illustration of the progression of three chains in the Anytime Parallel Tempering Monte Carlo (APTMC) algorithm on a single processor. The *green* (local move) and *blue* (exchange move) dots represent samples from the posterior being recorded as their respective local and exchange moves are completed. When exchange moves occur (at t_1 and t_2), the chain that is currently moving cannot participate in exchange moves without introducing a bias. Therefore it is ignored, and the exchange moves are performed on the remaining (inactive) chains.

Algorithm 1 Anytime Parallel Tempering Monte Carlo on one processor

1: Initialise real-time Markov jump process $\left(X^{1:\Lambda}, L, J\right)(0) = \left(x_0^{1:\Lambda}, 0, 1\right)$
2: **for** $i = 1, 2, \ldots$ **do**
3: Simulate real-time Markov jump process $\left(X^{1:\Lambda}, L, J\right)(t)$ until real time t_i
4: Perform exchange steps on the conditional

$$A(\mathrm{d}x^{1:\Lambda\setminus j} \mid x^j, l, j) = \prod_{\lambda=1,\ \lambda\neq j}^{\Lambda} \pi_\lambda(\mathrm{d}x^\lambda)$$

5: **end for**

Algorithm 2 Anytime Parallel Tempering Monte Carlo on multiple processors

1: On worker w, initialise the real-time Markov jump process $\left(X_w^{1:K}, L_w, J_w\right)(0) = \left(x_0^{1:K}, 0, 1\right)$
2: **for** $i = 1, 2, \ldots$ **do**
3: On each worker, simulate the real-time Markov jump process $\left(X_w^{1:K}, L_w, J_w\right)(t)$ until real time t_i
4: Across all workers, perform exchange steps on the conditional

$$A(\mathbf{d}\boldsymbol{x}^{1:K\setminus \boldsymbol{j}} \mid \boldsymbol{x}^{\boldsymbol{j}}, \boldsymbol{l}, \boldsymbol{j}) = \prod_{w=1}^{W} \prod_{k=1,\ k\neq j_w}^{K} \pi_w(\mathrm{d}x_w^k)$$

where $\mathbf{d}\boldsymbol{x}^{1:K\setminus \boldsymbol{j}} = \left(\mathrm{d}x_1^{1:K\setminus j_1}, \ldots, \mathrm{d}x_W^{1:K\setminus j_W}\right)$, $\boldsymbol{x}^{\boldsymbol{j}} = \left(x_1^{j_1}, \ldots, x_W^{j_W}\right)$, $\boldsymbol{l} = l_{1:W}$ and $\boldsymbol{j} = j_{1:W}$
5: **end for**

3.3.2 Multiple processors

When multiple processors are available, the Λ chains may be run in parallel. However, running a single chain on each processor means that when the real-time deadline occurs, all chains will be distributed according to their respective anytime distributions α^λ, and thus be biased as exchange moves occur. Therefore, all processors must contain at least two chains, and the algorithm is defined as described in Algorithm 2. Each worker uses $K > 2$ chains where either all chains have the same target distribution, e.g. for $W = \Lambda$ workers, worker $w = \lambda$ contains K chains targeting π_λ, or each chain has a different target distribution. For example, with $W = \frac{\Lambda}{2}$ workers, worker w could contain two chains, one with target π_w and one with target π_{2w}, or alternatively one one with target π_{2w-1} and one with target π_{2w}. Note that the multiple chain construction eliminates the intractable densities in the acceptance ratio for the exchange step when τ differs between processors.

4 Application to Approximate Bayesian Computation (ABC)

In this section we adapt the anytime parallel tempering Monte Carlo framework to Approximate Bayesian computation (ABC).

4.1 Overview of Approximate Bayesian Computation

The notion of ABC was developed by Tavaré et al. [1997] and Pritchard et al. [1999]. It can be seen as a likelihood-free way to perform Bayesian inference, using instead simulations from the model or system of interest, and comparing them to the observations available.

Let $y \in \mathbb{R}^d$ be some data with underlying unknown parameters $\theta \sim p(\mathrm{d}\theta)$, where $p(\theta)$ denotes the prior for $\theta \in \Theta$. Suppose we are in the situation in which the likelihood $f(y \mid \theta)$ is either intractable or too computationally expensive, which means that MCMC cannot

be performed as normal. Assuming that it is possible to sample from the density $f(\cdot \mid \theta)$ for all $\theta \in \Theta$, approximate the likelihood by introducing an artificial likelihood f^ε of the form

$$f^\varepsilon(y \mid \theta) = \mathrm{Vol}(\varepsilon)^{-1} \int_{B_\varepsilon(y)} f(x \mid \theta)\mathrm{d}x \tag{6}$$

where $B_\varepsilon(y)$ denotes a metric ball centred at y of radius $\varepsilon > 0$ and $\mathrm{Vol}(\varepsilon)$ is its volume. The resulting approximate posterior is given by

$$p^\varepsilon(\theta \mid y) = \frac{p(\theta) f^\varepsilon(y \mid \theta)}{\int p(\vartheta) f^\varepsilon(y \mid \vartheta)\mathrm{d}\vartheta}$$

The likelihood $f^\varepsilon(y \mid \theta)$ cannot be evaluated either, but a MCMC kernel can be constructed to obtain samples from the approximate posterior $\pi^\varepsilon(\theta, x)$ defined as

$$\begin{aligned}\pi^\varepsilon(\theta, x) &= p^\varepsilon(\theta, x \mid y) \\ &\propto p(\theta) f(x, \theta) \mathbb{1}_\varepsilon(x) \mathrm{Vol}(\varepsilon)^{-1}\end{aligned}$$

where $\mathbb{1}_\varepsilon(x)$ is the indicator function for $x \in B_\varepsilon(y)$. This is referred to as *hitting* the ball $B_\varepsilon(y)$. In the MCMC kernel, one can propose $\theta' \sim q(\mathrm{d}\theta' \mid \theta)$ for some proposal density q, simulate the dataset $x \sim f(\mathrm{d}x \mid \theta')$ and accept θ' as a sample from the posterior if $x \in B_\varepsilon(y)$.

The *1-hit MCMC kernel*, proposed by Lee [2012] and described in Algorithm 3 introduces local moves in the form of a 'race': given current and proposed parameters θ and θ', respectively simulate corresponding datasets x and x' sequentially. The state associated with the first dataset to hit the ball $B_\varepsilon(y)$ 'wins' and is accepted as the next sample in the Markov chain. The proposal θ' is also accepted as if both x and x' hit the ball at the same time.

Algorithm 3 ABC: 1-hit MCMC kernel

Given current state (θ_n, x_n)
for $i := 1, 2, \ldots$ **do**
  Propose $\theta' \sim q(\mathrm{d}\theta \mid \theta_n)$ ▷ *propose a local move*
  Compute preliminary acceptance probability ▷ *prior check*

$$\alpha(\theta_n, \theta') = \min\left\{1, \frac{p(\theta')q(\theta_n \mid \theta')}{p(\theta_n)q(\theta' \mid \theta_n)}\right\}$$

  Sample $u \sim \mathrm{Uniform}(0, 1)$
  if $u < \alpha(\theta_n, \theta')$ **then**
    RACE := TRUE
  else
    RACE := FALSE
    retain $(\theta_{n+1}, x_{n+1}) = (\theta_n, x_n)$ ▷ *automatically reject θ' as it is unlikely to win*
  end if
  while RACE **do**
    Simulate $x \sim f(\mathrm{d}x \mid \theta_n)$ and $x' \sim f(\mathrm{d}x' \mid \theta')$
    if $x \in B_\varepsilon(y)$ **or** $x' \in B_\varepsilon(y)$ **then** ▷ *stop the race once either x or x' hits the ball*
      RACE := FALSE
    end if
  end while
  if x' falls within ε of y **then** ▷ *accept or reject move*
    set $(\theta_{n+1}, x_{n+1}) = (\theta', x')$
  else
    retain $(\theta_{n+1}, x_{n+1}) = (\theta_n, x_n)$
  end if
  $n := n + 1$
end for

4.2 ABC Anytime Parallel Tempering Monte Carlo (ABC-APTMC)

We now enter an anytime Monte Carlo setting and introduce exchange moves to the 1-hit MCMC kernel.

4.2.1 Exchange moves

Let (θ, x) and (θ', x') be the states of two chains targeting π^ε and $\pi^{\varepsilon'}$, respectively, where $\varepsilon' > \varepsilon$. Here, this is equivalent to saying θ' is the state of the 'warmer' chain. We already know that x' falls within ε' of the observations y, i.e. $x' \in B_{\varepsilon'}(y)$. Similarly, we also know that $x \in B_\varepsilon(y)$, and clearly that $x \in B_{\varepsilon'}(y)$. If x' also falls within ε of y, then swap the states, otherwise do not swap. The odds ratio is

$$\begin{aligned}&\frac{\pi^{\varepsilon'}(\theta, x)\pi^\varepsilon(\theta', x')}{\pi^\varepsilon(\theta, x)\pi^{\varepsilon'}(\theta', x')} \\ &= \frac{p(\theta)f(x \mid \theta)\mathrm{Vol}(\varepsilon')p(\theta')f(x' \mid \theta')\mathbb{1}_\varepsilon(x')\mathrm{Vol}(\varepsilon)}{p(\theta)f(x \mid \theta)\mathrm{Vol}(\varepsilon)p(\theta')f(x' \mid \theta')\mathrm{Vol}(\varepsilon')} \\ &= \mathbb{1}_\varepsilon(x')\end{aligned}$$

so the probability of the swap being accepted is the probability of x' also hitting the ball of radius ε centred at y. This type of exchange move is summarised in Algorithm 4.

4.2.2 Implementation

The full implementation of the ABC Anytime Parallel Tempering Monte Carlo (`ABC-APTMC`) algorithm on a single processor is described in Algorithm 5. The multi-processor algorithm can similarly be modified to reflect these new exchange moves.

4.2.3 Remarks

It is important to note that if the race in the local move of, say, the $(K+1)$th chain takes too long to complete,

none of the other K chains can progress locally. Therefore, any new samples on those 'idle' chains will be a result of exchange moves, and they will be essentially swapping around the same K samples until the race on the working chain finishes. This is less of an issue on multiple processors, as the chains on other workers are still able to move forward locally. This means that, depending on where the chains were initialised, there is a risk at time T that very few effective updates will have occurred, as the race step in Algorithm 3 will have taken up most of the computation time. To suppress such a risk, solutions are the following:

- To ensure enough local updates and exchange moves occur, initialise the real-time Markov process near the centre of the distribution, e.g. simulate a few samples from the posterior using ABC rejection sampling and initialise the chains from those. This is the solution used in throughout this paper.
- Alternatively, initialise the algorithm with balls of relatively large radii $\varepsilon_0^{1:\Lambda}$ to increase the chances that they will be hit, and then reduce $\varepsilon_t^{1:\Lambda}$ with time t to progressively increase accuracy.

Algorithm 4 ABC: exchange move between two chains

Given $\omega_n = ((\theta, x), (\theta', x'))$ where $\theta \sim \pi$, $x \sim f(\mathrm{d}x \,|\, \theta)$ and $\theta' \sim \pi'$, $x' \sim f(\mathrm{d}x' \,|\, \theta')$.
▷ *both (θ, x) and (θ', x') are outputs from Algorithm 3 for different $\varepsilon' > \varepsilon$*

if $x' \in B_\varepsilon(y)$ **then** ▷ *accept or reject swap depending on whether x' also hits the ball of radius ε*
  set $\omega_{n+1} = ((\theta', x'), (\theta, x))$
else
  retain $\omega_{n+1} = \omega_n$
end if
$n := n + 1$

5 Experiments

In this section, we first illustrate the workings of the algorithms presented in Section 3.3 on a simple model, in which real-time behaviour is simulated using virtual time and an artificial hold distribution. The model is also employed to demonstrate the gain in efficiency provided by the inclusion of exchange moves. Then, the ABC version of the algorithms, as presented in Section 4, is applied to two case studies. The first case is a simple model and serves to verify the workings of the ABC algorithm, including bias correction. The second case considers the problem of estimating the parameters of a stochastic Lotka-Volterra predator-prey model – in which the likelihood is unavailable – and serves to evaluate the performance of the anytime parallel tempering version of the ABC-MCMC algorithm, as opposed to the standard versions (with and without exchange moves) on both a single and multiple processors. The exchange moves are set up so that multiple pairs could be swapped at each iteration. All experiments in this paper were tun on MATLAB and the code is available at https://github.com/alixma/ABCAPTMC.git.

Algorithm 5 ABC: Anytime Parallel Tempering Monte Carlo Algorithm

Initialise the real-time Markov jump process $(\theta^{1:\Lambda}, L, J) = (\theta_0^{1:\Lambda}, 0, 1)$.
Set $n := 0$
for $i := 1, 2, \ldots$ **do**
SIMULATE THE REAL-TIME MARKOV JUMP PROCESS $(\theta, L, J)(t)$ UNTIL REAL TIME t_i
  Perform local moves on $\left(\theta_n^j, x_n^j\right)$ according to Algorithm 3.
  $j := j + 1$
  if $j > \Lambda$ **then**
    $j := 1$
  end if
PERFORM EXCHANGE STEPS ON THE CONDITIONAL:

$$A(\mathrm{d}\theta^{1:\Lambda} \,|\, \theta^j, l, j) = \prod_{\lambda=1, \lambda\neq j}^{\Lambda} \pi_\lambda(\mathrm{d}\theta^\lambda)$$

  Perform exchange moves on $\omega_n = ((\theta_n^\lambda, x_n^\lambda), (\theta_n^{\lambda'}, x_n^{\lambda'}))$ according to Algorithm 4.
end for

5.1 Toy example: Gamma mixture model

In this example we attempt to sample from an equal mixture of two Gamma distributions using the Anytime Parallel Tempering Monte Carlo (`APTMC`) algorithm. Define the target $\pi(\mathrm{d}x)$ and an 'artificial' hold time $\tau(\mathrm{d}h \,|\, x)$ distributions as follows:

$$X \sim \phi\,\mathrm{Gamma}(k_1, \theta_1) + (1-\phi)\,\mathrm{Gamma}(k_2, \theta_2)$$
$$H \,|\, x \sim \psi\,\mathrm{Gamma}\left(\frac{x^p}{\theta_1}, \theta_1\right) + (1-\psi)\,\mathrm{Gamma}\left(\frac{x^p}{\theta_2}, \theta_2\right)$$

with mixture coefficients $\phi = \frac{1}{2}$ and ψ, where $\mathrm{Gamma}(\cdot, \cdot)$ denotes the pdf of a Gamma distribution, with shape and scale parameters (k_1, θ_1) and (k_2, θ_2) for each components, respectively, and with polynomial degree p, assuming it remains constant for both components of the mixture.

In this example, virtual time is employed instead of real time. Usually, almost nothing is known about the

hold time distribution τ, and in particular not its explicit form. However, for this toy example we assume an explicit form for τ is known and simulate virtual hold times. These artificial hold times are introduced such that what in a real-time example would be the effects of polynomial computational complexity can be studied, including constant ($p = 0$), linear ($p = 1$), quadratic ($p = 2$) and cubic ($p = 3$) complexity. Another advantage is that the anytime distribution $\alpha_\Lambda(\mathrm{d}x)$ of the cold chain can be computed analytically and is the following mixture of two Gamma distributions

$$\begin{aligned}\alpha_\Lambda(\mathrm{d}x) = \varphi(p, k_1, k_2, \theta_1, \theta_2)\,\mathrm{Gamma}\,(k_1 + p, \theta_1) \\ + [1 - \varphi(p, k_1, k_2, \theta_1, \theta_2)]\,\mathrm{Gamma}\,(k_2 + p, \theta_2)\end{aligned} \tag{7}$$

where

$$\varphi(p, k_1, k_2, \theta_1, \theta_2) = \frac{1}{1 + \frac{\Gamma(k_1)\Gamma(p+k_2)\theta_2^p}{\Gamma(k_2)\Gamma(p+k_1)\theta_1^p}}$$

We refer the reader to Appendix A.1 for the proof of Equation 7. In the anytime distribution, one of the components of the Gamma distribution will have an associated mixture coefficient $\varphi(p)$ or $1-\varphi(p)$ which increases with p while the coefficient of the other component decreases proportionally. Note that for constant ($p = 0$) computational complexity, the anytime distribution is equal to the target distribution π.

5.1.1 Implementation

On a single processor, the Anytime Parallel Tempering Monte Carlo algorithm is implemented as follows: simulate $\Lambda = 8$ Markov chains, each targeting the distribution $\pi_\lambda(\mathrm{d}x) = \pi(\mathrm{d}x)^{\frac{\lambda}{\Lambda}}$. To construct a Markov chain $(X^\lambda)_{n=0}^\infty$ with target distribution

$$\pi_\lambda(x) \propto \left[\frac{1}{2}\mathrm{Gamma}\,(k_1, \theta_1) + \frac{1}{2}\,\mathrm{Gamma}\,(k_2, \theta_2)\right]^{\frac{\lambda}{\Lambda}}$$

for $\lambda = 1, \ldots, \Lambda$, simply use a *Random Walk Metropolis* update, i.e. symmetric proposal distribution $\mathcal{N}(x_n^\lambda, \sigma^2)$ with mean x_n^λ and standard deviation $\sigma = 0.5$. Set $(k_1, k_2) = (3, 20)$, $(\theta_1, \theta_2) = (0.15, 0.25)$ and use $p \in \{0, 1, 2, 3\}$. The single processor algorithm is run for $T = 10^8$ units of virtual time with exchange moves alternating between occurring on all even $(1,2), (3,4), (5,6)$ and all odd $(2,3), (4,5), (6,7)$ pairs of inactive chains every $\delta_T = 5$ time steps. When the algorithm is running, a sample is recorded every time a local or exchange move occurs.

On multiple processors, the Anytime Parallel Tempering Monte Carlo algorithm is implemented similarly. A number of $W = \Lambda = 8$ workers is used, where each worker $w = \lambda$ contains $K = 2$ chains, all targeting the same π_λ for $\lambda = 1, \ldots, \Lambda$. The multiple processor algorithm is run for $T = 10^7$ units of virtual time, with exchange moves alternating between occurring on all even $(1,2), (3,4), (5,6), (7,8)$ and all odd $(2,3), (4,5), (6,7)$ pairs of workers every $\delta_T = 5$ steps. On each worker, the chain which was not working when calculations were interrupted is the one included in the exchange moves.

5.1.2 Verification of bias correction

To check that the single and multiple processor algorithms are successfully correcting for bias, they are also run *uncorrected*, i.e. with exchange steps being performed on the conditional $A(\mathrm{d}x^{1:\Lambda} \,|\, l)$ instead of $A(\mathrm{d}x^{1:\Lambda\setminus j} \,|\, x^j, l, j)$. This means that exchange moves are performed on α instead of π, thus causing the algorithm to yield biased results. Since the bias is introduced by the exchange moves (when they are performed on α), we attempt to create a 'worst case scenario', i.e. maximise the amount of bias present when the single processor algorithm is uncorrected. The algorithm is further adjusted such that local moves are not performed on the cold chain and it is instead solely made up of samples resulting from exchange moves with the warmer chains. The fact that exchange moves occur every $\delta_T = 5$ time steps also means that a high proportion of the samples in a warmer chain come from exchange moves. The multi-processor algorithm is not run in a 'worst case scenario' as the virtual time required to properly verify that the corrected algorithm converges to the true posterior when $p = 3$ is too high. Local moves on the cold chain of the multi-processor algorithm are therefore allowed, and the bias caused by failing to correct when performing exchange moves across workers should still be apparent, if less strongly.

Figure 2 shows kernel density estimates of the post burn-in cold chains resulting from runs of the single and multi-processor algorithms, uncorrected and corrected for bias. As expected, a constant ($p = 0$) computational complexity does not return any bias. While coming close, but not quite completely reaching the corresponding anytime distributions − which would be the most extreme case of bias −, the cold chains for the single-processor algorithm with computational complexity $p \in \{1, 2, 3\}$ have clearly converged to a shifted distribution which puts more weight the second Gamma mixture component, instead of an equal weight. Additionally, the bias becomes stronger as computational complexity p increases. A similar observation can be made for the cold chains from the multi-processor ex-

periment − which display a milder bias due to local moves occurring on the cold chain. On the other hand, the dashed densities indicate that when the algorithms are corrected, i.e. when the currently working chain is not included in exchange moves, it successfully eliminates the bias for all $p \in \{1, 2, 3\}$ to return the correct posterior π − despite even this being the 'worst case scenario' in the case of the single processor algorithm. Note that in the single processor experiment, the uncorrected density estimates never quite reach their corresponding anytime distributions because they are not solely made up of biased samples. Indeed, if chain $k = 2$ was not working before the uncorrected exchange move with the cold chain ($k = 1$) was accepted, then the new sample on the cold chain is unbiased as it's receiving a sample from π_2. Conversely, if chain 2 was working and the swap is performed, then the new sample on the cold chain will be from α_2 and therefore biased. By eliminating the local moves on the cold chain, we have significantly augmented the proportion of biased samples, but they still don't make up 100% of the chain. Additionally, the stronger bias for higher computational complexities is due to more frequent exchange moves occurring on each chain, as their local moves are 'slower'.

5.1.3 Performance evaluation

Next we verify that introducing the parallel tempering element to the anytime Monte Carlo algorithm improves performance. A standard `MCMC` algorithm is run for $T = 10^6$ units of virtual time and computational complexity $p \in \{0, 1, 2, 3\}$, applying the random walk Metropolis update described in Section 5.1.1. The single and multiple processor Anytime Parallel Tempering Monte Carlo algorithms are run again for the same amount of virtual time, with exchange moves occurring every $\delta_T = 5$ time steps. The single processor version is run on $\Lambda_s = 8$ chains, and the multi-processor on $W = 8$ workers, with $K = 2$ chains per worker, so $\Lambda_m = 16$ chains in total. This time, local moves are performed on the cold chain of the single processor `APTMC` algorithm.

To compare results, kernel density estimates of the posterior are obtained from the post burn-in cold chains for each algorithm using the `kde` function in MATLAB [2019], developed by Botev et al. [2010]. It is also important to note that even though all algorithms run for the same (virtual) duration, the standard `MCMC` algorithm is performing local moves on a single chain uninterrupted until the deadline while the single-processor `APTMC` algorithm has to update $\Lambda = 8$ chains in sequence and each worker w of the multi-processor `APTMC` algorithm has to update $K = 2$ chains in sequence before exchange moves occur. Therefore, the algorithms are not expected to return samples of the similar sizes. For a fair performance comparison, the sample autocorrelation function (acf) is estimated first of all. When available, the acf is averaged over multiple chains to reduce variance in its estimates. Other tools employed are

- *Integrated Autocorrelation Time* (IAT), the computational inefficiency of a MCMC sampler. Defined as

$$IAT_s = 1 + 2\sum_{\ell=1}^{\infty} \rho_s(\ell)$$

  where $\rho_s(\ell)$ is the autocorrelation at the ℓ-th lag of chain s. It measures the average number of iterations required for an independent sample to be drawn, or in other words the number of correlated samples with same variance-reducing power as one independent sample. Hence, a more efficient algorithm will have lower autocorrelation values and should yield a lower IAT value. Here, the IAT is estimated using a method initially suggested in Sokal [1997] and Goodman and Weare [2010], and implemented in the Python package `emcee` by Foreman-Mackey et al. [2013] (Section 3). Let

$$\hat{IAT}_s = 1 + 2\sum_{\ell=1}^{M} \hat{\rho}_s(\ell)$$

  where M is a suitably chosen cutoff, such that noise at the higher lags is reduced. Here, the smallest M is chosen such that $M \geq C\hat{\rho}_s(M)$ where $C \approx 6$. More information on the choice of C is available in Sokal [1997].
- *Effective Sample Size* (ESS), the amount of information obtained from a MCMC sample. It is closely linked to the IAT by definition:

$$ESS_s = \frac{N_s}{IAT_s}$$

  where N_s is the size of the current sample s. The ESS measures the number of independent samples obtained from MCMC output.

The resulting ESS and IAT for different algorithms and computational complexities are computed and shown in Table 1. For the multi-processor `APTMC` algorithm, the IAT is computed using the averaged acf over all chains available, and the ESS is summed over the output cold chains. While the proposal for local moves depends on the value of the previous state, the state of a chain if an exchange move is accepted does not, meaning that the autocorrelation in a chain

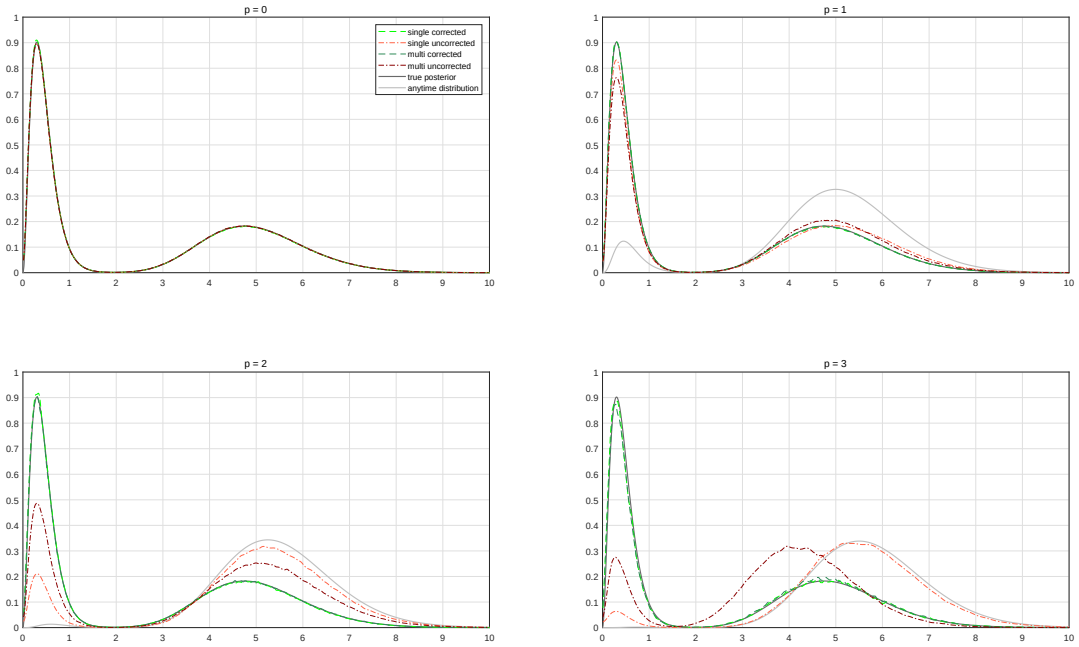

Fig. 2 Density estimates of the cold chain for bias corrected and uncorrected runs of the single- and multi-processor `APTMC` algorithm on various computational complexities $p \in \{0, 1, 2, 3\}$. In the single-processor case, the cold chains are made up entirely of updates resulting from exchange moves. The *dark gray* line represents the true posterior density π and the *light gray* line the anytime distribution α. The case $p = 0$ represents an instance in which, in a real-time situation, the local moves do not take a random time to complete, and therefore all densities are identical.

containing a significant proportion of accepted samples from exchange moves will be lower. Finally, for low p, significantly more local moves occur as hold times are short while for a higher p the hold times will be longer and hence fewer local moves are able to occur before each deadline. We therefore expect a higher proportion of samples from exchange moves for higher values of p and a more important increase in efficiency.

5.1.4 Performance results

In Figure 3 we observe, unsurprisingly, that the quality of the posterior estimates decreases as p increases. As a matter of fact, 10^6 units of virtual time tend to not be enough for the some of the cold chains to completely converge. The single processor `APTMC` algorithm overestimates the first mode and underestimates the second mode of the true posterior for $p = 2$, while none of the algorithms appear to have fully converged when $p = 3$. In general, the multi-processor `APTMC` returns results closest to the true cold posterior for $p = \{0, 1, 2\}$.

As for efficiency, Table 1 displays a much lower IAT and much higher ESS for both `APTMC` algorithms, indicating that they are much more efficient than the `AMC` algorithm. This is further supported by the sample autocorrelation decaying much more quickly for `APTMC` algorithms than for the `MCMC` algorithm for all p in Figure 4. The multi-processor `APTMC` also yields IAT values that are lower than those returned by the single processor `APTMC` algorithm, and similarly yields effective sample sizes that are higher for $p < 3$. It is however important to note that comparing these values for $p = 3$ can be misleading, since in Figure 3, none of the algorithm outputs have converged to the desired distributions for this value of p. Indeed, values of IAT may have been underestimated, especially for the single processor `APTMC` algorithm, which returned the lowest number of samples. The efficiency of exchange moves on a single processor compared to standard MCMC, along with the reasonably accurate results displayed in Figure 3 indicate that they are a good choice for algorithms with low computational complexity if access multiple processors is not easily available. However if one has access to parallel computing tools, the multi-processor `APTMC` is the most efficient and accurate choice for this problem.

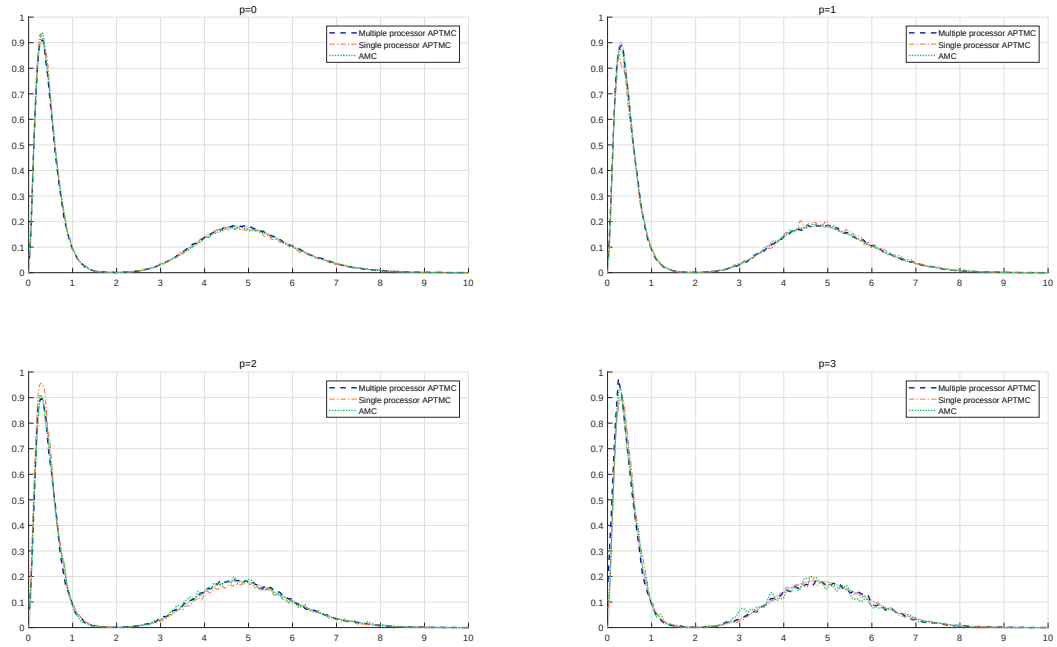

Fig. 3 Plots of kernel density estimates of the cold posterior for 10^6 units long runs of the single (*orange*) and multiple (*blue*) processor Anytime Parallel Tempering Monte Carlo (`APTMC`) algorithm as well as the standard (*green*) (`MCMC`) algorithm. Each plot corresponds to a different computational complexity $p \in \{0, 1, 2, 3\}$.

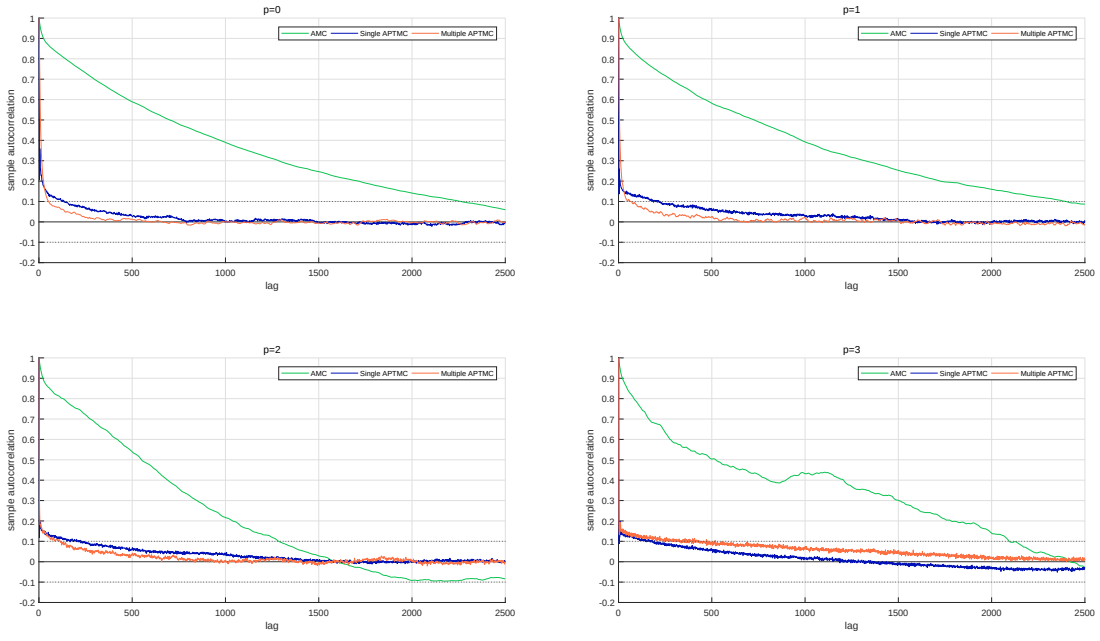

Fig. 4 Plots of the sample autocorrelation function up to lag 2500 of the post burn-in cold chain for runs of the single (*orange*) and multiple (*blue*) processor Anytime Parallel Tempering Monte Carlo (`APTMC`) algorithm as well as for the output of the standard Anytime Monte Carlo (`AMC`) algorithm (*green*). Each plot corresponds to a different computational complexity $p \in \{0, 1, 2, 3\}$.

p	Multi-processor		Single-processor		Standard	
	IAT	ESS	IAT	ESS	IAT	ESS
0	50.931	12757	70.584	1382.3	1853.4	269.71
1	46.733	5723.3	100.18	665.83	1489.0	129.52
2	54.785	1715.4	94.357	633.1	1163.5	33.748
3	210.16	281.09	77.387	735.12	876.85	9.1784

Table 1 Integrated autocorrelation time (IAT) and effective sample size (ESS) for 10^6 units long runs of the single-, multi-processor anytime Parallel Tempering and standard MCMC algorithms for different computational complexities $p \in \{0, 1, 2, 3\}$.

Next, we consider an application of the anytime parallel tempering Monte Carlo framework to a class of algorithms that are well-adapted to situations in which the likelihood is either intractable or computationally prohibitive. They are called Approximate Bayesian computation and feature a non-artificial hold time at each MCMC iteration, making them ideal candidates for adaptation to the anytime parallel tempering framework.

5.2 ABC toy example: univariate Normal distribution

To validate the results of Section 4.2, consider another simple example, initially featured in Lee [2012], and adapted here within the anytime parallel tempering Monte Carlo framework. Let Y be a Gaussian random variable, i.e. $Y \sim \mathcal{N}(y; \theta, \sigma^2)$, where the standard deviation σ is known but the mean θ is not. The ABC likelihood here is

$$f^{\varepsilon}(y \,|\, \theta) = \Phi\left(\frac{y + \varepsilon - \theta}{\sigma}\right) - \Phi\left(\frac{y - \varepsilon - \theta}{\sigma}\right)$$

for $\varepsilon > 0$. Using numerical integration tools in MATLAB, it is possible to obtain a good approximation of the true posterior for any ε for visualisation. Let $y = 3$ be an observation of Y and $\sigma^2 = 1$, and put the prior $p(\theta) = \mathcal{N}(\theta; 0, 5)$ on θ. In this example, the exact posterior distribution for θ can easily be shown to be $\mathcal{N}\left(\theta; \frac{5}{2}, \frac{5}{6}\right)$.

When performing local moves (Algorithm 3), use a Gaussian random walk proposal with standard deviation $\xi = 0.5$. The real-time Markov jump process is run using $\Lambda = 10$ chains. The algorithm is run on a single processor for one hour or $T = 3600$ seconds in real time after a 30 second burn-in, with exchange moves occurring every $\delta_T = 5 \times 10^{-4}$ seconds (or 0.5 milliseconds). The radii of the balls $\varepsilon^{1:\Lambda}$ are defined to vary between $\varepsilon^1 = 0.1$ and $\varepsilon^{\Lambda} = 1.1$.

5.2.1 *Verification of bias correction*

First of all, we verify that bias correction must be applied for all chains to converge to the correct posterior. This is done by comparing density estimates of each of the post burn-in chains to the true corresponding posterior (obtained by numerical integration). When bias correction is not applied, meaning that the `ABC-APTMC` algorithm is run including the currently working chain j, every chain converges to an erroneous distribution which overestimates the mode of its corresponding posterior, as is clearly visible in Figure 5. On the other hand, correcting the algorithm for such bias ensures that every chain converges to the correct corresponding posterior.

Next, we compare the performance of the `ABC-APTMC` algorithm to that of a standard `ABC` algorithm. For that, a more applied parameter estimation example is considered, for which the adoption of a likelihood-free approach is beneficial.

5.3 Stochastic Lotka-Volterra model

In this section, we consider the stochastic Lotka-Volterra predator-prey model (Lotka [1926], Volterra [1927]), further exploring the example considered in Lee and Łatuszyński [2014], which is itself based on an example in Chapter 6 of Wilkinson [2011]. In this case study, the posterior is intractable and some of the components of the parameters θ (namely θ_2 and θ_3) exhibit strong correlations. Let $X_{1:2}(t)$ be a bivariate, integer-valued pure jump Markov process with initial values $X_{1:2}(0) = (50, 100)$, where $X_1(t)$ represents the number of preys and $X_2(t)$ the number of predators at time t. For small time interval Δt, we describe the predator-prey dynamics in the following way

$$\mathbb{P}\left\{X_{1:2}(t + \Delta t) = z_{1:2} \,|\, X_{1:2}(t) = x_{1:2}\right\} = \begin{cases} \theta_1 x_1 \Delta t + o(\Delta t), & \text{if } z_{1:2} = (x_1 + 1, x_2) \\ \theta_2 x_1 x_2 \Delta t + o(\Delta t), & \text{if } z_{1:2} = (x_1 - 1, x_2 + 1) \\ \theta_3 x_2 \Delta t + o(\Delta t), & \text{if } z_{1:2} = (x_1, x_2 - 1) \\ o(\Delta t), & \text{otherwise} \end{cases}$$

In this example, the only observations available are the number of preys, i.e. X_1 at 10 discrete time points. Following theory in Wilkinson [2011] (Chapter 6), the process can be simulated and discretised using the Gillespie [1977] algorithm, in which the inter-jump times follow an exponential distribution. The observations employed were simulated in Lee and Łatuszyński

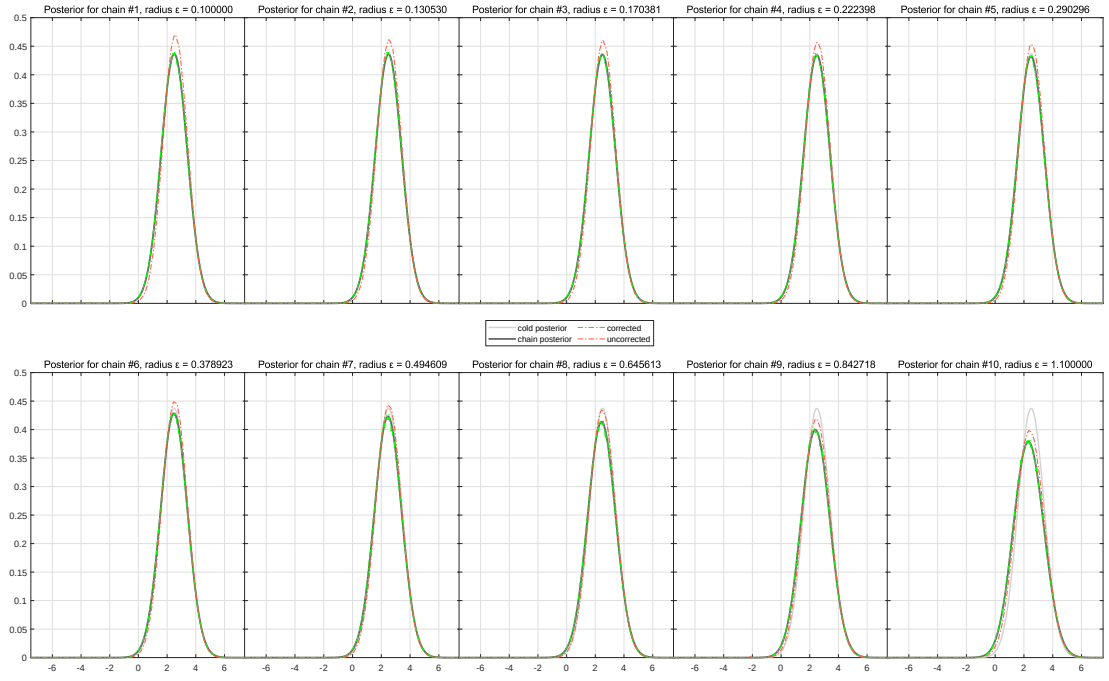

Fig. 5 Kernel density estimates of all chains for corrected and uncorrected runs of the single processor `ABC-APTMC` algorithm. In each subplot, the *light gray* line is fixed and represents the cold posterior for reference, the *dark gray* line represents each chain's target posterior (obtained by numerical integration), the dot-dashed *green* lines are kernel density estimates of the chain's posterior returned by the corrected algorithm and agree with the dark gray line. The *orange* lines are kernel density estimates for the uncorrected algorithm, and do not agree with the gray line, as expected.

[2014] with true parameters $\theta = (1, 0.005, 0.6)$, giving $y = \{88, 165, 274, 268, 114, 46, 32, 36, 53, 92\}$ at times $\{1, \ldots, 10\}$. For ABC, the 'ball' considered takes the following form for $\varepsilon > 0$

$$B_\varepsilon(y) = \{X_1(t) : |\log[X_1(i)] - \log[y(i)]| \leq \varepsilon, \forall i = 1, \ldots, 10\} \tag{8}$$

therefore, a set of simulated $X_1(t)$ is considered as 'hitting the ball' if all 10 simulated data points are at most e^ε times (and at least $e^{-\varepsilon}$ times) the magnitude of the corresponding observation in y.

5.3.1 Methods and settings

In Lee and Łatuszyński [2014], the 1-hit MCMC kernel (`ABC`), is shown to return the most reliable results by comparison with other MCMC kernels which are not considered here. While it can be reasonably fast, it is highly inefficient as it has a very low acceptance rate, and thus the autocorrelation between samples for low lags is very high. Another important issue in this particular example is that the race in the 1-hit kernel is prone to getting stuck for extended periods of time. Therefore, we aim to first of all improve performances by introducing exchange moves on a single processor (`ABC-PTMC`). Then – and most importantly – we further improve the algorithm by implementing both the single and multi-processor parallel tempering algorithms within the anytime framework (`ABC-APTMC`).

5.3.1.1 One processor

Departing slightly from the example in Lee and Łatuszyński [2014], define the prior on $\theta \in [0, \infty)^3$ for the single processor experiment to be $p(\theta) = \exp\{-\theta_1 - \theta_2 - \theta_3\}$, i.e. three independent exponential priors, all with mean 1. The proposal distribution is a truncated normal, i.e. $\theta' \,|\, \theta \sim T\mathcal{N}(\theta, \Sigma)$, $\theta' \in (0, 10)$ with mean θ and covariance $\Sigma = \text{diag}(0.25, 0.0025, 0.25)$. The truncated normal is used in order to ensure that all proposals remain non-negative. For reference, 2364 independent samples from the posterior are obtained via ABC rejection sampling with $\varepsilon = 1$ and the density estimates in Figure 6 of Lee and Łatuszyński [2014] are reproduced. To obtain these posterior samples, 10^7 independent samples from the prior were required, yielding the very low 0.024% acceptance rate. This method of sampling

from the posterior is therefore extremely inefficient, and the decision to resort to MCMC kernels in order to improve efficiency is justified.

On a single processor, the three algorithms considered are the standard 1-hit MCMC kernel (`ABC`), the single processor version of the algorithm with added exchange moves (`ABC-PTMC-1`) and the same but within the anytime framework (`ABC-APTMC-1`). They are run ten times for 100800 seconds (28 hours) − after 3600 seconds (1 hour) of burn-in − and their main settings are summarised in Table 2. It is important to note that the parallel tempering algorithms, having to deal with updating multiple chains sequentially, are likely to return cold chains with fewer samples. The algorithms must therefore be properly set up such that the gain in efficiency introduced by exchange moves is not overshadowed by the greater number of chains and computational cost of having to update them all. In this experiment, this means the parallel tempering algorithms must be run on just 6 chains, each targeting posteriors associated with balls of radii $\varepsilon^{1:6} = \{1, 1.1447, 1.3104, 1.5, 11, 15\}$ and the proposal distribution has covariance $\Sigma^{1:6}$ where $\Sigma^{\lambda} = \text{diag}\left(\sigma^{\lambda}, \sigma^{\lambda}10^{-2}, \sigma^{\lambda}\right)$ and $\sigma^{1:6} = \{0.008, 0.025, 0.05, 0.09, 0.25, 0.5\}$. Exchange moves are performed as described in Algorithm 4 and alternate between even $(1,2),(3,4),(5,6)$ (excluding $(5,6)$ in the anytime version) and odd $(2,3),(4,5)$ pairs of eligible chains. In the anytime version, in order to determine how long the local moves should run for before exchange moves occur, the time taken for the standard algorithm to perform a fixed number δ_T of local moves is measured at each iteration, and the median over all iterations is taken. This ensures that both algorithms spend the same median time performing local moves.

5.3.1.2 Multiple processors

Then, we aim to demonstrate the gain in efficiency introduced by running the parallel tempering algorithm, not only within the anytime framework, but also on multiple processors. The algorithms considered are the single processor `ABC-PTMC-1`, `ABC-APTMC-1` as well as their multi-processor counterparts `ABC-PTMC-W` and `ABC-APTMC-W`. This time, instead of relying on an informative, exponential prior on θ, which leads to lower acceptance rates on the warmer chains, we define a uniform prior between 0 and 3. The proposal distribution is still a truncated normal, but with tighter limits (corresponding to the prior) i.e. $\theta' \,|\, \theta \sim T\mathcal{N}(\theta, \Sigma)$, $\theta' \in (0,3)$. Again, 1988 independent samples from the posterior are obtained for reference. They are generated via ABC rejection sampling with $\varepsilon = 1$. To obtain these posterior samples, 10^8 independent samples from the prior were required, yielding the even lower 0.002% acceptance rate.

Since the standard `ABC` algorithm is not considered here, there is no need to balance gain in efficiency with the reduced number of samples on the cold chain present because of the multiple chains. Therefore, the four algorithms are run on 20 chains, each targeting posteriors associated with balls of radii ranging from $\varepsilon^1 = 1$ to $\varepsilon^{20} = 11$ and proposal distribution covariances $\Sigma^{1:20}$ where $\Sigma^k = \text{diag}\left(\sigma^k, \sigma^k 10^{-2}, \sigma^k\right)$ for chain k and where values range from $\sigma^1 = 0.008$ to $\sigma^{20} = 0.5$. These are tuned so that the acceptance rates of exchange moves between adjacent chains remain on average greater than 70%. The algorithms are run four times for 864000 seconds (24 hours) and their main settings are summarised in Table 3.

In this experiment, multiple chains running at different temperatures are present on each worker. In order to account for the approx. 0.8 second communication overhead when switching between local moves − running in parallel on all workers − and exchange moves − running on the master −, exchange moves are divided into two types:

1. Exchange moves *within workers*: performed on each individual worker in parallel, alternating between even $(1,2)$ and odd $(2,3)$ pairs of adjacent chains. No communication between workers is necessary in this case.
2. Exchange moves *between workers*: performed on the master by alternating between selecting all even $(1,2),(3,4),\ldots,(19,20)$ and odd $(2,3),\ldots,(18,19),(20,21)$ pairs of adjacent eligible chains, excluding $(20,21)$ in the anytime version. This time, communication between workers is required.

After (parallel) local moves are performed, each iteration alternates between *within* and *between workers* exchange moves. Communication between workers is therefore only needed once every two iterations, thus reducing the total communication overhead. In this new algorithm construction, we adapt the method determining how long the workers in the anytime version of the algorithm should work in parallel before between-worker exchange moves occur. The time (excluding communication overhead) taken for the `ABC-PTMC-W` algorithm to perform a set of parallel updates − i.e. δ_T local moves, an exchange move within workers and δ_T more local moves − is measured on

Label	*Workers* W	*Chains* K	*Chains per worker*	*Exchange moves*	*Anytime*
`ABC`	1	1	1	none	No
`ABC-PTMC-1`	1	6	6	every 6 local moves	No
`ABC-APTMC-1`	1	6	6	every 2.59 seconds	Yes

Table 2 Algorithm settings for stochastic Lotka-Volterra predator-prey model on a single processor.

Label	*Workers* W	*Chains* K	*Chains per worker*	*Exchange moves*	*Anytime*
`ABC-PTMC-1`	1	21	21	every 63 local moves	No
`ABC-APTMC-1`	1	21	21	approx. every 10.3 seconds	Yes
`ABC-PTMC-W`	7	21	3	every 9 local moves	No
`ABC-APTMC-W`	7	21	3	every 9 local moves (*within* workers) and approx. every 15.3 seconds (*between* workers)	Yes

Table 3 Algorithm settings for stochastic Lotka-Volterra predator-prey model on multiple processors.

each worker and at each iteration. Then, the median over all iterations is taken for each worker, and the `ABC-APTMC-W` algorithm is set to perform parallel updates for as long as the slowest of all workers. This ensures both that all workers have a chance to perform the δ_T local moves, and that workers containing chains which are quicker to complete their local moves do not sit idle waiting for the worker containing the slowest chains to finish. It was observed that the slowest workers were most often the ones containing the colder half of the chains.

5.3.2 Performance evaluation

All algorithms returned density estimates that were reasonably close to those obtained via rejection sampling, but all would require to run for a much longer period of time to be indistinguishable from them, which is why in this experiment we chose to focus mainly on comparing their efficiency. In order to compare the performance of the algorithms, as stated above, all algorithms compared are set to run for the same real time period. Once again the integrated autocorrelation time (IAT) and effective sample size (ESS) are computed for all algorithms. While the IAT and sample autocorrelation plots are good tools for comparing efficiency, they do not take into account the computational cost of running 6 chains instead of a single ones. The ESS on the other hand gives us how many effective samples the different algorithms can return within a fixed time frame. For example, a very fast algorithm could still return a higher ESS even if it has a much higher IAT. To illustrate how the anytime version of the parallel tempering algorithms is more computationally efficient compared to standard `ABC-PTMC`, the real times all algorithms take to perform local and exchange moves are measured and their timelines plotted.

5.3.2.1 One processor

Both the `ABC-PTMC-1` and `ABC-APTMC-1` algorithm display an improvement in performances: they return IATs that are respectively 3.55 and 2.16 times lower on average than those of the standard `ABC` algorithm in Table 4, and display a steeper decay in sample acf in Figure 6. In the 28 hours (post burn-in) during which the algorithms ran, both parallel algorithms also yielded an increased effective sample size. As a matter of fact, introducing exchange moves every 6 local moves has multiplied the ESS by approximately 1.173 on average, and performing them every 2.59 seconds within the anytime framework has further increased it, returning effective sample sizes that are approximately 2.09 times as large as that of the standard `ABC` algorithm. We note that in this example, the `ABC-PTMC-1` returned a lower IAT than its anytime counterpart. This is due to the low number of chains needed in this experiment. The 6 radii corresponding to each chain are further apart than they would usually be, and because of this, the proportion of rejected exchange moves in the `ABC-APTMC-1` algorithm is higher, as the construction of the anytime algorithm often requires one of the exchange moves to occur on a pair of chains that are not adjacent. On an experiment with a higher number of chains, this issue becomes negligible.

The median time spent performing local moves was the same for the `ABC-PTMC-1` and `ABC-APTMC-1` algorithm. However, the distribution of times spent per-

forming local moves in the run of the ABC-APTMC-1 is more closely concentrated around the median. While many of the local moves in the run of the ABC-PTMC-1 algorithm took as little as 0.413 seconds to complete, others took over 40 seconds. Some local moves even took nearly two hours to complete. On the other hand, since a deadline was implemented in the anytime version of the algorithm, Figure 7 displays more consistent local move times.

	Standard ABC		ABC-PTMC-1		ABC-APTMC-1	
	IAT	*ESS*	*IAT*	*ESS*	*IAT*	*ESS*
θ_1	67.993	7606.7	20.038	8529.5	36.797	13589
θ_2	118.07	4380.4	30.962	5520	49.086	10187
θ_3	145.39	3557.4	44.456	3844.5	68.419	7308.5

Table 4 Cumulative effective sample size (ESS) and mean integrated autocorrelation time (IAT) over ten 28-hour runs of the ABC, ABC-PTMC-1 and ABC-APTMC-1 algorithms to estimate the posterior distributions of the parameters $\theta = (\theta_1, \theta_2, \theta_3)$ of a stochastic Lotka-Volterra model.

5.3.2.2 Multiple processors

In the multi-processor case study, both the ABC-PTMC-1 and ABC-PTMC-W were set so that on each worker an exchange move occurred after all chains had been updated locally once. In theory, this should naturally give an advantage to the multi-processor version, as the various chains working in parallel are able to perform more local moves in the given time, and exchange moves occur every 5 local moves instead of every 20. In practice, if the inter-worker communication overhead occurring once every two exchange moves is too long, it may occupy too much of the allocated time and thus cause the multi-processor version of the algorithm to return fewer samples, and hence fewer effective samples. Here, this issue was avoided: the total number of samples returned by the ABC-PTMC-W algorithm is much higher for all chains (see Table 6) and therefore the approx. 1 second communication overhead was not prohibitive. Additionally, the cold chain returned by the ABC-PTMC-W algorithm contains approximately twice as many samples originating from exchange moves as that returned by its single processor counterpart ABC-PTMC-1 (see Table 7) and has an integrated autocorrelation time that is on average 1.92 times smaller, as suggested in Table 5 and by the steeper decay in sample acf in Figure **??**. Because of this, Table 5 shows that the effective sample sizes for the cold chain returned by the ABC-PTMC-W algorithm are on average 1.74 times higher than those of the single processor version.

The same observations can be made for the anytime versions of the algorithm, though note that the real time deadline implemented means that the number of local and within-worker exchange moves occurring before a between-worker exchange move varies, and therefore it is possible for two exchange moves to occur consecutively (i.e within workers and then between workers). The cold chain returned by the ABC-APTMC-W algorithm has an integrated autocorrelation time that is on average 1.72 times smaller than its single processor counterpart ABC-APTMC-1, as suggested in Table 5. Comparing the percentages of exchange moves present in the various chains, columns ABC-APTMC-1 and ABC-APTMC-W in Table 7 indicate that these percentages have doubled to tripled after switching to multiple processors. Even more striking is the great increase in total sample size returned for each chain. Indeed, Table 6 shows that while the number of samples on the cold chain is already 2.21 times higher, we observe by switching to multiple processors a 10-fold increase in the number of samples on warmer chains. Because of this, the effective sample sizes for the cold chain returned by the ABC-APTMC-W algorithm in Table 5 are on average 3.62 times higher than those of the single processor version.

As for the main comparison – namely anytime vs standard ABC with exchange moves – Table 5 indicates that the single processor ABC-APTMC-1 algorithm returns an effective sample size on average 2.26 times larger than the ABC-PTMC-1 algorithm. On multiple processors, the improvement is even greater, with the ABC-APTMC-W algorithm returning an effective sample size on average 4.82 times larger than the ABC-PTMC-W algorithm. Figures 8 and 9 illustrate the advantage of implementing a real-time deadline to local moves. At most local moves, the issue in which all workers sit idle waiting for the slowest to finish arises for the ABC-PTMC-W algorithm. This issue is most clearly visible on Figure 8 with Worker 2 from around 80 seconds on. On the other hand, Figure 9 clearly shows that the anytime version of the algorithm is making better use of the allocated computational resources. The anytime framework essentially ensures that all workers do not have to wait for the slowest among them to finish, allowing for more exploration of the sample space in the faster workers. Additionally, the real time deadline ensures that even if chain k on Worker w remains stuck in a race for an extended period of time, the other workers are still updating. Therefore, while the remaining four

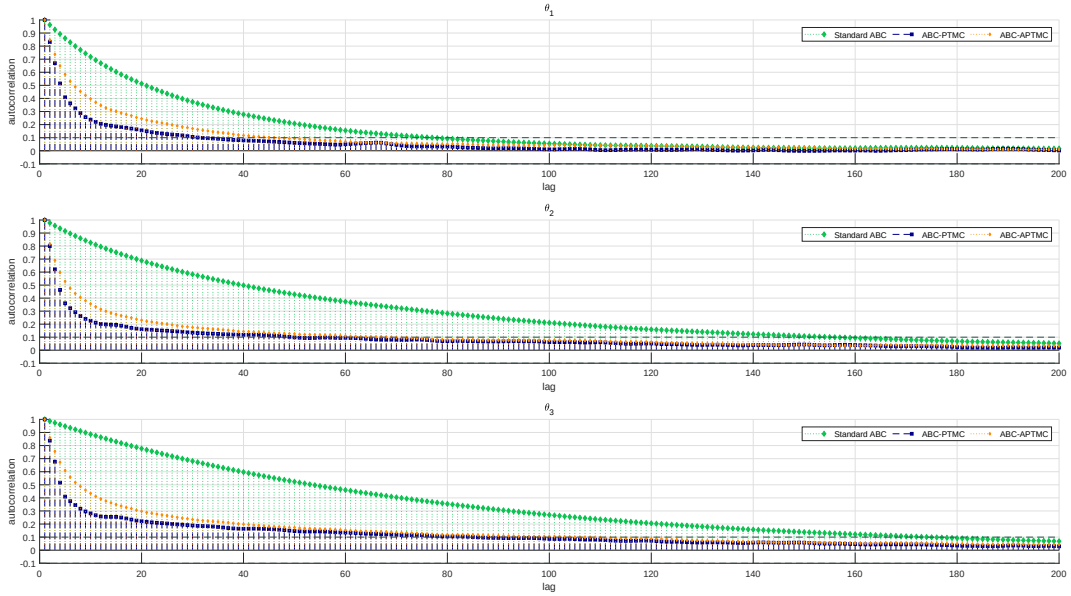

Fig. 6 Plots of the sample autocorrelation function up to lag 200 of the cold chain for runs of the standard `ABC` (*green*), single processor `ABC-PTMC-1` (*blue*) and multi-processor `ABC-PTMC-1` (*orange*) algorithms to estimate the posterior distributions of the parameters $\theta = (\theta_1, \theta_2, \theta_3)$ of a stochastic Lotka-Volterra model.

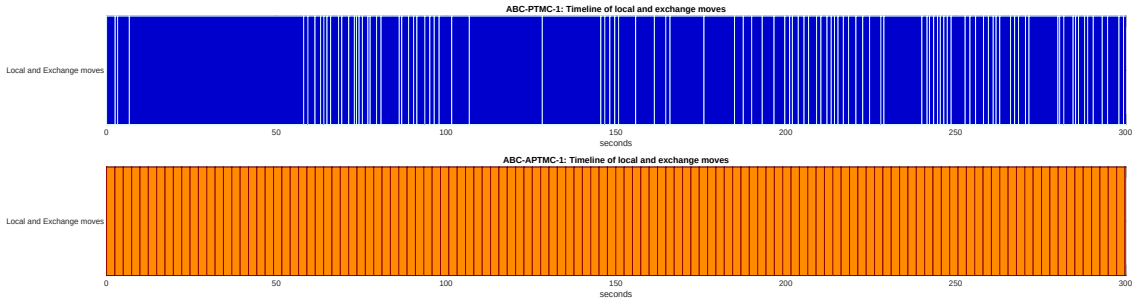

Fig. 7 Timeline of local and exchange moves for the `ABC-PTMC-1` and `ABC-APTMC-1` algorithms for the first 300 seconds. The exchange moves are represented by the *white* and *red* lines and the local moves by the *dark blue* and *orange* coloured blocks.

chains on Worker w wait for chain k to complete its race, they also continue to be updated at regular intervals thanks to the exchange moves with other workers. Because of this, while the the integrated autocorrelation time in Table 5 for the cold chain does not seem to be significantly lower for the `ABC-APTMC-W` algorithm, its sample size in Table 6 has quadrupled compared to that of the `ABC-PTMC-W` algorithm.

between the cold chains returned by the `ABC-PTMC-W` and `ABC-APTMC-W` algorithms, and by the roughly similar decay in sample acf in Figure **??**.

The addition of ABC exchange moves in his case study proved fruitful, as the effective sample size for the parameters of the Lotka-Volterra model was increased. However this required fine tuning of the settings. The benefits of ABC parallel tempering will be stronger and more easily visible in a problem in which the parameters to be estimated have a multimodal distribution, as a single chain may get stuck in local optima while multiple tempered chains will explore more of the sample space. Nonetheless, the introduction of the anytime framework is clearly an important improvement. It ensures the various chains in ABC parallel tempering continue to be updated (via exchange moves) even when one of them is stuck performing local moves for longer than expected, and encourages the algorithm to make better use of its allocated resources, especially on multiple processors.

	One processor				Multiple processors			
	ABC-PTMC-1		ABC-APTMC-1		ABC-PTMC-W		ABC-APTMC-W	
	IAT	ESS	IAT	ESS	IAT	ESS	IAT	ESS
θ_1	50.444	297.94	40.041	748.08	30.992	439.93	22.153	2679.9
θ_2	37.376	339.35	36.852	726.38	22.139	580.53	22.909	2549
θ_3	65.253	238.84	53.809	509.99	26.566	482.88	31.073	1919.4

Table 5 Cumulative effective sample size (ESS) and mean integrated autocorrelation time (IAT) over four 24-hour runs of the `ABC-PTMC-1`, `ABC-APTMC-1`, `ABC-PTMC-W` and `ABC-APTMC-W` algorithms to estimate the posterior distributions of the parameters $\theta = (\theta_1, \theta_2, \theta_3)$ of a stochastic Lotka-Volterra model.

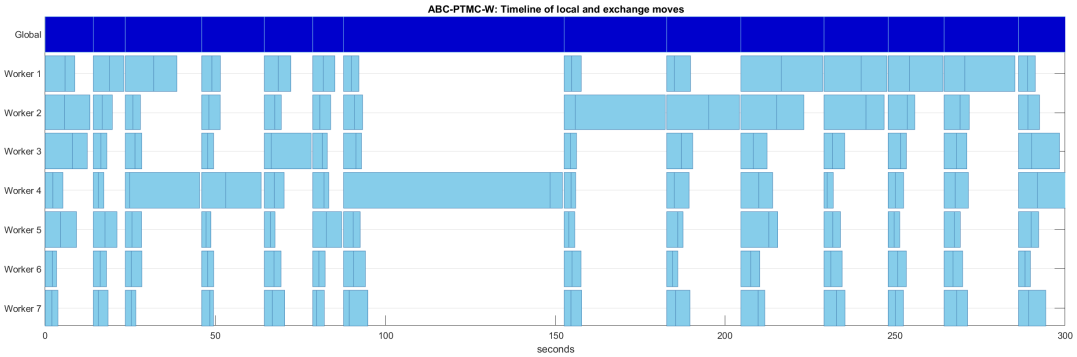

Fig. 8 Timeline of local and exchange moves for the `ABC-PTMC-W` algorithm for the first 300 seconds.

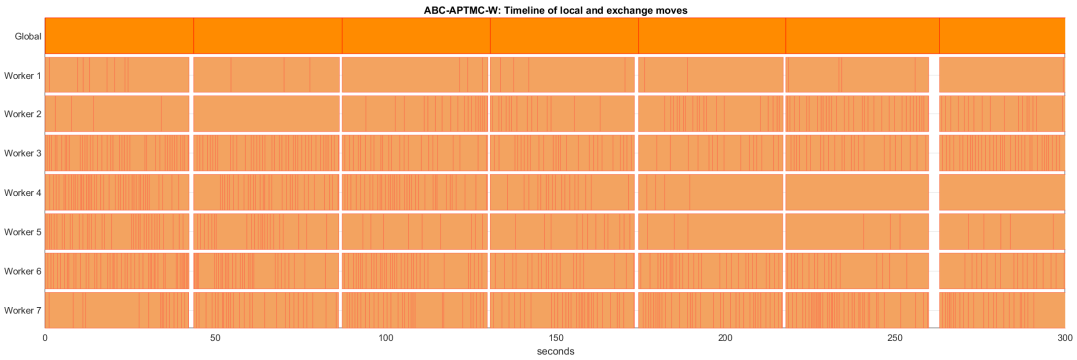

Fig. 9 Timeline of local and exchange moves for the `ABC-APTMC-W` algorithm for the first 300 seconds.

6 Conclusion

In an effort to increase the efficiency of MCMC algorithms, in particular for use on distributed computing, and for situations in which the likelihood is either unavailable or too computationally costly, the Anytime Parallel Tempering Monte Carlo algorithm was developed. The algorithm combines the enhanced exploration of the state space, provided by the between-chain exchange moves in parallel tempering, with control over the real-time budget and robustness to interruptions available within the anytime Monte Carlo framework.

Initially, the construction of the anytime Monte Carlo algorithm, with the inclusion of exchange moves on a single and multiple processors, was verified on a toy Gamma mixture example, and the performance improvements they brought were demonstrated by comparing the algorithm to a standard MCMC algorithm. Subsequently, the exchange moves were adapted for pairing with the 1-hit ABC-MCMC kernel, a simulation-based algorithm the within Approximate Bayesian computation (ABC) framework, which provides an attractive, likelihood-free approach to MCMC. The construction of the adapted ABC algorithm was verified using a simple univariate normal example. Then, the increased efficiency of the inclusion of ex-

change moves was demonstrated in comparison to that of a standard ABC algorithm on a parameter estimation problem involving the coefficients of a moving average MA(q) process. Anytime parallel tempering ABC was finally employed to estimate the parameters of a stochastic Lotka-Volterra predator-prey model based on partial and discrete data − a problem in which the likelihood is unavailable. On a single processor, it was shown that introducing exchange moves provides an improvement in performance and an increase in the effective sample size compared to that of the standard, single chain ABC algorithm. This improvement is additionally boosted by the inclusion of the anytime framework. On multiple processors, it was shown that the anytime framework helps the parallel tempering ABC algorithm to make better use of the computational resources and thus provides an even stronger boost to effective sample size.

References

R. Bardenet, A. Doucet, and C. Holmes. An adaptive subsampling approach for MCMC inference in large datasets. In *Proceedings of The 31st International Conference on Machine Learning*, pages 405–413, 2014.

R. Bardenet, A. Doucet, and C. Holmes. On Markov chain Monte Carlo methods for tall data. *arXiv preprint arXiv:1505.02827*, 2015.

M. Betancourt. A conceptual introduction to Hamiltonian Monte Carlo. *arXiv preprint arXiv:1701.02434*, 2017.

Z. I. Botev, J. F. Grotowski, D. P. Kroese, et al. Kernel density estimation via diffusion. *The Annals of Statistics*, 38(5):2916–2957, 2010.

B. Calderhead. A general construction for parallelizing Metropolis- Hastings algorithms. *Proceedings of the National Academy of Sciences*, 111(49):17408–17413, 2014.

S. Duane, A. D. Kennedy, B. J. Pendleton, and D. Roweth. Hybrid Monte Carlo. *Physics letters B*, 195(2):216–222, 1987.

D. Foreman-Mackey, D. W. Hogg, D. Lang, and J. Goodman. emcee: The MCMC Hammer. *Publications of the Astronomical Society of the Pacific*, 125(925):306, 2013.

C. Geyer. Importance sampling, simulated tempering and umbrella sampling. *Handbook of Markov Chain Monte Carlo*, pages 295–311, 2011.

C. J. Geyer. Markov chain Monte Carlo maximum likelihood. *Interface Foundation of North America*, 1991.

D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The journal of physical chemistry*, 81(25):2340–2361, 1977.

J. Goodman and J. Weare. Ensemble samplers with affine invariance. *Communications in applied mathematics and computational science*, 5(1):65–80, 2010.

R. Kohn, M. Quiroz, M.-N. Tran, and M. Villani. Speeding up MCMC by Efficient Data Subsampling. Working Papers 2123/16205, University of Sydney Business School, Discipline of Business Analytics, 2016. URL https://ideas.repec.org/p/syb/wpbsba/2123-16205.html.

A. Korattikara, Y. Chen, and M. Welling. Austerity in MCMC land: Cutting the Metropolis-Hastings budget. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 181–189, 2014.

A. Lee. On the choice of MCMC kernels for approximate Bayesian computation with SMC samplers. In *Simulation Conference (WSC), Proceedings of the 2012 Winter*, pages 1–12. IEEE, 2012.

A. Lee and K. Łatuszyński. Variance bounding and geometric ergodicity of markov chain monte carlo kernels for approximate bayesian computation. *Biometrika*, 101(3):655–671, 2014.

F. Liang, J. Kim, and Q. Song. A bootstrap Metropolis–Hastings algorithm for Bayesian analysis of big data. *Technometrics*, 58(3):304–318, 2016.

A. J. Lotka. Elements of Physical Biology. *Science Progress in the Twentieth Century (1919-1933)*, 21 (82):341–343, 1926.

D. J. MacKay and D. J. Mac Kay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

D. Maclaurin and R. P. Adams. Firefly Monte Carlo: Exact MCMC with Subsets of Data. In *UAI*, pages 543–552, 2014.

MATLAB. *version 9.7.0.1190202 (R2019b)*. The MathWorks Inc., Natick, Massachusetts, 2019.

S. Minsker, S. Srivastava, L. Lin, and D. Dunson. Scalable and robust Bayesian inference via the median posterior. In *International Conference on Machine Learning*, pages 1656–1664, 2014.

L. M. Murray, S. Singh, P. E. Jacob, and A. Lee. Anytime Monte Carlo. *arXiv preprint arXiv:1612.03319*, 2016. URL https://arxiv.org/abs/1612.03319.

R. Neal. Handbook of Markov Chain Monte Carlo ed S. Brooks, A. Gelman, GL Jones and X.-L. Meng, 2011.

R. M. Neal. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 2012.

W. Neiswanger, C. Wang, and E. Xing. Asymptotically exact, embarrassingly parallel MCMC. *arXiv preprint arXiv:1311.4780*, 2013.

J. K. Pritchard, M. T. Seielstad, A. Perez-Lezaun, and M. W. Feldman. Population growth of human Y chromosomes: a study of Y chromosome microsatellites. *Molecular biology and evolution*, 16(12):1791–1798, 1999.

M. Quiroz, M. Villani, and R. Kohn. Exact subsampling MCMC. *arXiv preprint arXiv:1603.08232*, 2016.

M. Quiroz, M.-N. Tran, M. Villani, and R. Kohn. Speeding up MCMC by delayed acceptance and data subsampling. *Journal of Computational and Graphical Statistics*, 2017.

C. Robert and G. Casella. *Monte Carlo Statistical Methods*, chapter The Metropolis-Hastings Algorithm. Springer Texts in Statistics, Springer, New York, 2004. ISBN 978-1-4757-4145-2. doi: 10.1007/978-1-4757-4145-2_7.

C. P. Robert, V. Elvira, N. Tawn, and C. Wu. Accelerating MCMC algorithms. *Wiley Interdisciplinary Reviews: Computational Statistics*, 10(5):e1435, 2018.

S. L. Scott, A. W. Blocker, F. V. Bonassi, H. A. Chipman, E. I. George, and R. E. McCulloch. Bayes and big data: The consensus Monte Carlo algorithm. *International Journal of Management Science and Engineering Management*, 11(2):78–88, 2016.

A. Sokal. Monte Carlo Methods in Statistical Mechanics: Foundations and New Algorithms. In *Functional integration*, pages 131–192. Springer, 1997.

R. H. Swendsen and J.-S. Wang. Replica Monte Carlo simulation of spin-glasses. *Physical Review Letters*, 57(21):2607, 1986.

S. Tavaré, D. J. Balding, R. C. Griffiths, and P. Donnelly. Inferring coalescence times from DNA sequence data. *Genetics*, 145(2):505–518, 1997.

V. Volterra. *Variazioni e fluttuazioni del numero d'individui in specie animali conviventi*. C. Ferrari, 1927.

X. Wang and D. B. Dunson. Parallelizing MCMC via Weierstrass sampler. *arXiv preprint arXiv:1312.4605*, 2013.

D. J. Wilkinson. *Stochastic Modelling for Systems Biology*. CRC press, 2011.

M. Xu, B. Lakshminarayanan, Y. W. Teh, J. Zhu, and B. Zhang. Distributed Bayesian posterior sampling via moment sharing. In *Advances in Neural Information Processing Systems*, pages 3356–3364, 2014.

A Proofs

A.1 Anytime distribution of the cold chain

Proof To obtain the anytime distribution in the Gamma mixture example in Section 5.1, compute the three components of the expression in Equation (3):

1. The density of X

$$\pi(\mathrm{d}x) = \frac{x^{k_1-1}}{2\Gamma(k_1)\theta_1^{k_1}} e^{-\frac{x}{\theta_1}} + \frac{x^{k_2-1}}{2\Gamma(k_2)\theta_2^{k_2}} e^{-\frac{x}{\theta_2}}\,\mathrm{d}x$$

where $\Gamma(\,\cdot\,)$ is the gamma function.

2. The expectation of $H \,|\, x$

$$\mathbb{E}\left[H \,|\, x\right] = \psi x^p + (1-\psi)x^p = x^p$$

Note that the ψ factors cancel out, meaning that the anytime distribution is independent of ψ and therefore its value can be chosen to be 1 for convenience.

3. To compute $\mathbb{E}\left[H\right]$, use a property of conditional expectation and the honesty conditions of the Gamma(k_1+p,θ_1) and Gamma(k_2+p,θ_2) distributions:

$$\begin{aligned}
\mathbb{E}\left[H\right] &= \mathbb{E}\left[\mathbb{E}\left(H \,|\, x\right)\right] = \mathbb{E}\left[x^p\right] \\
&= \int \frac{x^{p+k_1-1}}{2\Gamma(k_1)\theta_1^{k_1}} e^{-\frac{x}{\theta_1}}\,\mathrm{d}x + \int \frac{x^{p+k_2-1}}{2\Gamma(k_2)\theta_2^{k_2}} e^{-\frac{x}{\theta_2}}\,\mathrm{d}x \\
&= \frac{\Gamma(p+k_1)\theta_1^{p+k_1}}{2\Gamma(k_1)\theta_1^{k_1}} \cdot 1 + \frac{\Gamma(p+k_2)\theta_2^{p+k_2}}{2\Gamma(k_2)\theta_2^{k_2}} \cdot 1 \\
&= \frac{\Gamma(k_2)\Gamma(p+k_1)\theta_1^{p} + \Gamma(k_1)\Gamma(p+k_2)\theta_2^{p}}{2\Gamma(k_1)\Gamma(k_2)} \\
&= \frac{C}{2\Gamma(k_1)\Gamma(k_2)}
\end{aligned}$$

letting $C = \Gamma(k_2)\Gamma(p+k_1)\theta_1^p + \Gamma(k_1)\Gamma(p+k_2)\theta_2^p$.

Combining the three components,

$$\begin{aligned}
&\alpha(\mathrm{d}x) \\
&= \frac{2\Gamma(k_1)\Gamma(k_2)}{C}\left(\frac{x^{p+k_1-1}}{2\Gamma(k_1)\theta_1^{k_1}} e^{-\frac{x}{\theta_1}} + \frac{x^{p+k_2-1}}{2\Gamma(k_2)\theta_2^{k_2}} e^{-\frac{x}{\theta_2}}\right)\mathrm{d}x \\
&= \underbrace{\frac{\Gamma(k_2)\Gamma(p+k_1)\theta_1^{p+k_1}}{C\theta_1^{k_1}}}_{\varphi(p)} \underbrace{\frac{x^{p+k_1-1}}{\Gamma(p+k_1)\theta_1^{p+k_1}} e^{-\frac{x}{\theta_1}}}_{\text{Gamma}(p+k_1,\theta_1)} \\
&+ \underbrace{\frac{\Gamma(k_1)\Gamma(p+k_2)\theta_2^{p+k_2}}{C\theta_2^{k_2}}}_{\varphi'(p)} \underbrace{\frac{x^{p+k_2-1}}{\Gamma(p+k_2)\theta_2^{p+k_2}} e^{-\frac{x}{\theta_2}}}_{\text{Gamma}(p+k_2,\theta_2)}\,\mathrm{d}x
\end{aligned}$$

And now substituting back the expression C in $\varphi(p)$:

$$\begin{aligned}
\varphi(p) &= \frac{\Gamma(k_2)\Gamma(p+k_1)\theta_1^p}{\Gamma(k_2)\Gamma(p+k_1)\theta_1^p + \Gamma(k_1)\Gamma(p+k_2)\theta_2^p} \\
&= \frac{1}{1+\frac{\Gamma(k_1)\Gamma(p+k_2)\theta_2^p}{\Gamma(k_2)\Gamma(p+k_1)\theta_1^p}}
\end{aligned}$$

Similarly, we can obtain $\varphi'(p) = 1-\varphi(p)$. Therefore, the anytime distribution $\alpha(\mathrm{d}x)$ is the following mixture of two Gamma distributions:

$$\begin{aligned}
\alpha(\mathrm{d}x) = \varphi(p)\,&\text{Gamma}\left(k_1+p,\theta_1\right) \\
&+ (1-\varphi(p))\,\text{Gamma}\left(k_2+p,\theta_2\right)
\end{aligned}$$

B Tables

Chain k	ε^k	σ^k	ABC-PTMC-1	ABC-PTMC-W	ABC-APTMC-1	ABC-APTMC-W
1	1	0.008	2667.5	3241.8	6570.3	14488
2	1.046	0.009	2790	3564.8	6934.8	16902
3	1.094	0.011	2796.8	3567.5	6941	16837
4	1.145	0.012	2797.3	3604.5	6924.8	17264
5	1.197	0.014	2793.8	3719.8	6931.3	15710
6	1.253	0.016	2786.3	3748.8	6951.5	17157
7	1.31	0.019	2784.8	3629.3	6961.3	18947
8	1.371	0.022	2795.5	3615	6941.5	18551
9	1.434	0.025	2805.5	3608.5	6950.8	18759
10	1.5	0.029	2803.5	3711.5	6962.8	17276
11	1.661	0.034	2798.5	3799.8	6962.3	46350
12	1.84	0.039	2803.3	3693.3	6983	53716
13	2.038	0.045	2814.8	3656.5	6995.3	53289
14	2.257	0.052	2799	3658.3	7008.8	53458
15	2.5	0.06	2783.5	3796.3	7029.8	46597
16	3.362	0.092	2787.5	4054.5	7038.5	68953
17	4.522	0.14	2783.8	3936.5	7009.5	79231
18	6.082	0.214	2781	3912.5	6982.5	78917
19	8.179	0.327	2780.8	3919.5	7002.8	79038
20	11	0.5	2665.8	3598.5	6604.3	67725

Table 6 Average sample sizes per chain returned over four 24-hour runs of the ABC-PTMC-1, ABC-APTMC-1, ABC-PTMC-W, ABC-APTMC-W algorithms to estimate the posterior distributions of the parameters θ of a stochastic Lotka-Volterra model on multiple processors in Section 5.3.2. The ball radius ε^k and proposal distribution covariance $\mathrm{diag}\left(\sigma^k, \sigma^k 10^{-2}, \sigma^k\right)$ associated with each chain k are displayed for information.

Chain k	ε^k	σ^k	ABC-PTMC-1	ABC-PTMC-W	ABC-APTMC-1	ABC-APTMC-W
1	1	0.008	4.0364	10.254	6.3818	17.032
2	1.046	0.009	8.1412	18.47	11.344	28.873
3	1.094	0.011	8.7694	18.642	11.413	28.619
4	1.145	0.012	8.8518	19.329	11.189	30.387
5	1.197	0.014	8.6446	21.793	11.27	23.531
6	1.253	0.016	8.3202	21.528	11.548	22.644
7	1.31	0.019	8.3717	18.879	11.677	29.882
8	1.371	0.022	8.8443	18.581	11.423	28.332
9	1.434	0.025	8.994	18.568	11.546	29.134
10	1.5	0.029	8.9767	20.839	11.699	23.191
11	1.661	0.034	8.8116	21.189	11.705	19.591
12	1.84	0.039	8.9863	18.829	11.968	30.582
13	2.038	0.045	9.3053	17.93	12.108	30.013
14	2.257	0.052	9.0069	18.051	12.298	30.245
15	2.5	0.06	8.628	20.982	12.581	20.011
16	3.362	0.092	8.369	19.888	12.668	18.274
17	4.522	0.14	8.1945	17.477	12.282	28.847
18	6.082	0.214	8.181	16.817	11.965	28.564
19	8.179	0.327	8.1896	17.075	12.246	28.677
20	11	0.5	4.2839	9.7452	6.8761	16.773

Table 7 Mean percentage of exchange moves per chain returned over four 24-hour runs of the ABC-PTMC-1, ABC-APTMC-1, ABC-PTMC-W, ABC-APTMC-W algorithms to estimate the posterior distributions of the parameters θ of a stochastic Lotka-Volterra model on multiple processors in Section 5.3.2. The ball radius ε^k and proposal distribution covariance $\mathrm{diag}\left(\sigma^k, \sigma^k 10^{-2}, \sigma^k\right)$ associated with each chain k are displayed for information.

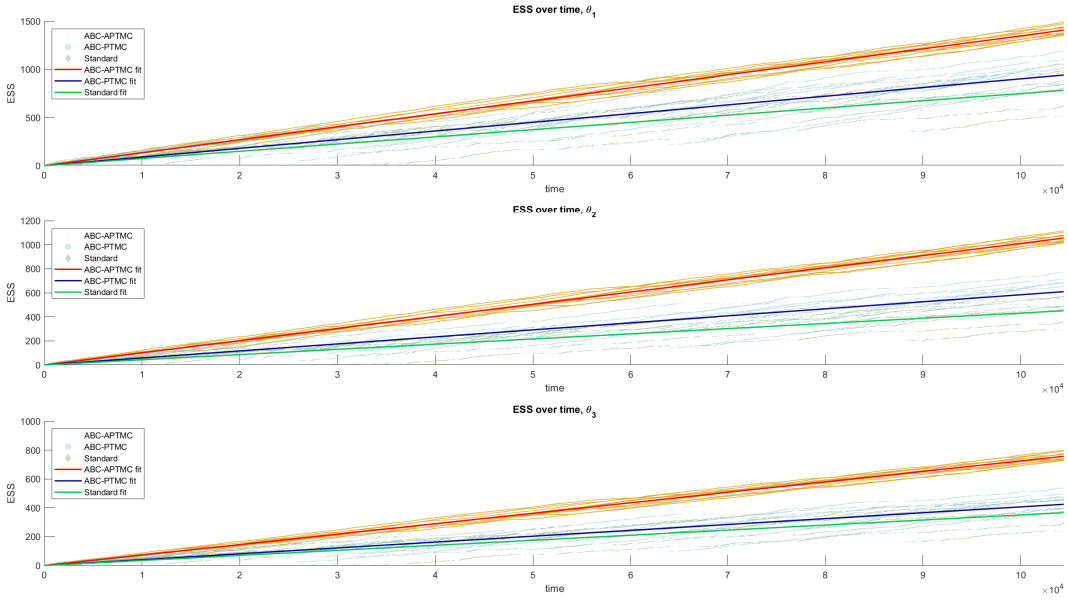

Fig. 10 Effective sample size over time for ten 28-hour runs of the standard `ABC`, `ABC-PTMC-1` and `ABC-APTMC-1` algorithms.