NTNU
Innovation and Creativity

Parallel Multiple Proposal MCMC Algorithms

Haakon Michael Austad

Master of Science in Physics and Mathematics
Submission date: June 2007
Supervisor: Håkon Tjelmeland, MATH

Norwegian University of Science and Technology
Department of Mathematical Sciences

Problem Description

The candidate will study Metropolis-Hastings algorithms with several proposals in each iterations. In particular he will consider how parallel computation can be used in this context, and how one can use also the rejected states to estimate quantities of interest.

Assignment given: 15. January 2007
Supervisor: Håkon Tjelmeland, MATH

Preface

This report represents my work on my Master's thesis during the spring semester of 2007. It represents 20 weeks of work and completes my five year Master of Science program at the Department of Mathematical Sciences of the Norwegian University of Science and Technology (NTNU) in Trondheim.

I have found the work on my thesis to be highly challenging but also very rewarding. In this report I have tried to maintain a focus on the subject we wished to investigate. I have tried to give intuitive explanations of the algorithms at hand, and at the same time give a precise presentation of the theory behind them. The report should be accessible for most students with a background in mathematics and statistics at a university level, although some familiarity with MCMC algorithms would be advisable.

I would like to take the opportunity to thank my supervisor Associate Professor Håkon Tjelmeland for all his guidance and assistance. His ideas and opinions have been very helpful throughout the project.

Trondheim, June 2007
Haakon M. Austad

Abstract

We explore the variance reduction achievable through parallel implementation of multi-proposal MCMC algorithms and use of control variates. Implemented sequentially multi-proposal MCMC algorithms are of limited value, but they are very well suited for parallelization. Further, discarding the rejected states in an MCMC sampler can intuitively be interpreted as a waste of information. This becomes even more true for a multi-proposal algorithm where we discard several states in each iteration. By creating an alternative estimator consisting of a linear combination of the traditional sample mean and zero mean random variables called control variates we can improve on the traditional estimator. We present a setting for the multi-proposal MCMC algorithm and study it in two examples. The first example considers sampling from a simple Gaussian distribution, while for the second we design the framework for a multi-proposal mode jumping algorithm for sampling from a distribution with several separated modes. We find that the variance reduction achieved from our control variate estimator in general increases as the number of proposals in our sampler increase. For our Gaussian example we find that the benefit from parallelization is small, and that little is gained from increasing the number of proposals. The mode jumping example however is very well suited for parallelization and we get a relative variance reduction pr time of roughly 80% with 16 proposals in each iteration.

Contents

1 Introduction **1**

2 Markov chain Monte Carlo and the Metropolis-Hastings algorithm **2**
2.1 MCMC 2
2.2 The Metropolis-Hastings algorithm 3

3 Combination of kernels and mode jumping proposals **4**
3.1 Combination of kernels 4
3.2 Mode jumping 6

4 Multi-proposal MCMC **7**
4.1 The idea, and initial setting 8
4.2 Transition matrix 9
4.3 Parallel implementation 10

5 Mode jumping with multiple proposals **11**
5.1 Multi-proposal mode jumping algorithm 11
5.2 Notation and Remarks 12

6 Control variates and use of rejected states **13**
6.1 Using control variates to estimate means 14
6.2 Using all proposed states to estimate the mean 14

7 Examples **17**
7.1 Gaussian toy example 18
7.2 Mode jumping example 26

8 Closing remarks **31**

1 Introduction

This report focuses on two variance reduction techniques for MCMC algorithms, the multiple proposal technique and the control variate technique. Assume we want to estimate the mean μ of a function $f(x)$, with x distributed according to some target distribution $\pi(\cdot)$. MCMC algorithms are often the only viable alternative for estimating μ, in particular if x is of a high dimension and $\pi(\cdot)$ is complex. The most common MCMC algorithm, the Metropolis-Hastings scheme (Hastings 1970) runs as follows. Assuming we are in the state x of the Markov chain, a new state y is proposed according to a proposal distribution. The new state is then accepted or rejected according to an acceptance probability α. If the new state is accepted we continue with y as the current state of the Markov chain, while if it is rejected the proposal y is discarded and we continue with x. Once the algorithm has run long enough, and after discarding a burn-inn period the selected states are roughly distributed according to the target distribution $\pi(\cdot)$. The most common estimator for μ is then the sample mean of $f(x)$. Several methods exist for reducing the variance of our estimator, in Roberts & Casella (1999) and Casella & Robert (1996) a Rao-Blackwellised version of the traditional estimator is shown to give significant variance reduction. Liu (2001) also points out several variance reduction techniques.
Our project has focused on the multi-proposal MCMC algorithm. The algorithm is based on the idea of proposing several new states in each iteration instead of just one. Implemented sequentially, there is little hope of the multi-proposal MCMC algorithm reducing the variance of our estimator as a function of time. Tjelmeland (2004) explores this in two relatively simple examples. However the multi-proposal algorithm can be improved through parallel implementation and control variates. The algorithm is very well suited for parallel implementation, with multiple proposals we can have each processor generate its own proposal. The control variate technique is based around the idea that we can do better than the sample mean by constructing an estimator that is a linear combination of the original sample mean and other random variables called control variates. Intuitively discarding proposals would seem like a waste of information, so our control variate is based upon the use of rejected states. With several rejected proposals for each iteration, as we have in the multi-proposal algorithm, the amount of information in the control variates should increase. Tjelmeland & Hammer (2005) explore several control variates of this type for the standard MCMC algorithm. One of the benefits of this method over for example the *Rao-Blackwellisation* scheme is that the amount of computation time is of order N oppose to N^2, where N is the number of iterations in the algorithm.
In this report we explore the effectiveness of the techniques for two examples each implemented in parallel. First we look at a toy Gaussian example similar to the examples in Tjelmeland (2004) and then compare this to a more advanced example using a mode jumping proposal technique inspired by Tjelmeland & Hegstad (2001). The paper starts with Section 2 giving a short introduction to the Metropolis-Hastings algorithm, Sections 3 and 4 explain the mode jumping and multi-proposal MCMC algorithm and Section 5 combines the two ideas into a multi-proposal mode jumping algorithm. Section 6 then

explains theory behind control variates. In Section 7 we present the results from our two examples, and finish with some closing remarks in Section 8.

2 Markov chain Monte Carlo and the Metropolis-Hastings algorithm

In this section we will discuss the Markov chain Monte Carlo (MCMC) idea and give a short outline of the Metropolis-Hastings algorithm. A more general introduction to MCMC can be found in Roberts & Casella (1999) and a thorough discussion of the Metropolis-Hastings algorithm in Hastings (1970).

2.1 MCMC

The primary purpose of MCMC is to simulate samples from a distribution and use these samples to estimate a mean. Assume we have some distribution which is difficult to simulate from, $\pi(\cdot)$. We wish to generate N samples from this distribution $x^1, \ldots, x^N$, in order to estimate its mean,

$$\mu = E[f(x)] = \int f(x)\pi(x)dx. \tag{1}$$

The most common estimator is the sample mean,

$$\hat{\mu} = \frac{1}{N}\sum_{i=0}^{N} f(x^i). \tag{2}$$

The MCMC idea is to simulate these samples by constructing a Markov chain with a transition kernel P, whose invariant distribution is equal to the target distribution $\pi(\cdot)$. In this way samples can be generated by running the chain for a sufficiently long time for the distribution to have converged to the limiting distribution. Assuming the target distribution to be continuous on $\Re^n$ a transition kernel P defines a Markov chain with invariant distribution $\pi(\cdot)$ if,

$$\int_A \pi(x)dx = \int_{\Re^n} P(A|x)\pi(x)dx \;\forall A \in \zeta, \tag{3}$$

where ζ is the Borel σ-algebra on $\Re^n$. In the case of a discrete target distribution on Ω this simplifies into,

$$\pi(y) = \sum_x \pi(x)P(x|y) \;\forall y \in \Omega. \tag{4}$$

Since equations (3) and (4) leave us with a lot of freedom in our choice of P it is common to require the Markov chain to be time reversible, which gives us the more restricting detailed balance condition,

$$\int_A \pi(x)P(B|x)dx = \int_B \pi(x)P(A|x)dx \;\forall A, B \in \zeta. \tag{5}$$

For a discrete target distribution this becomes,

$$\pi(y)P(x|y) = \pi(x)P(y|x)\ \forall x, y \in \Omega. \tag{6}$$

In MCMC algorithms it is often desirable to combine two or more transition kernels. Assuming we have two transition kernels P^0 and P^1 who both fulfill (5) it is immediately obvious that combining the two into a new transition kernel in the following manner, $P = \alpha P^0 + (1-\alpha)P^1$ for some $\alpha \in [0,1]$, will also fulfill (5). We can take things one step further and assume that we have a class of kernels P^ϕ indexed with a continuous parameter $\phi \in \Re^d$. Assuming that $f(\phi)$ is a distribution on $\Re^d$ it is clear that the transition kernel,

$$P(A|x) = \int_{\Re^d} P^\phi(A|x) f(\phi) d\phi \tag{7}$$

also fulfills the detailed balance condition.

2.2 The Metropolis-Hastings algorithm

The Metropolis-Hastings algorithm gives us a general framework for MCMC algorithms. It is based on splitting the transition kernel into a proposal kernel Q and an acceptance probability α. For a continuous distribution the Metropolis-Hastings algorithm constructs the transition kernel as,

$$P(A|x) = \int_A q(y|x)\alpha(y|x)dy + I(x \in A)r(x), \tag{8}$$

where $I(\cdot)$ is the indicator function, $q(y|x)$ is the proposal distribution and $r(x) = \int q(y|x)(1-\alpha(y|x))dy$ is the probability of remaining in x. For a discrete distribution the transition kernel appears as,

$$P(y|x) = q(y|x)\alpha(y|x) \text{ for } y \neq x. \tag{9}$$

The algorithm runs as a two step process, first a new state y is proposed from a proposal distribution $q(y|x)$. Calculating a probability of acceptance $\alpha(y|x)$, the new state is then either accepted or rejected in favor of the old state x. There are many available choices for $\alpha(y|x)$, in Hastings (1970) a general expression is shown to be,

$$\alpha(y|x) = \frac{s(x,y)}{1 + \frac{\pi(x)}{\pi(y)}\frac{q(y|x)}{q(x|y)}} \tag{10}$$

for some symmetric function $s(x,y) \geq 0$, that gives $\alpha(y|x) \leq 1$. It is then well known that the optimal acceptance probability in respect to the asymptotic variance of the estimates is given, as shown in Peskun (1973) by,

$$\alpha(y|x) = \min\left\{1, \frac{\pi(y)}{\pi(x)}\frac{q(x|y)}{q(y|x)}\right\}. \tag{11}$$

We should note that equations (10) and (11) hold for both the continuous and discrete cases. As we discussed in the previous section, it is possible to combine two or more transition kernels. With the Metropolis-Hastings algorithm we can consider the possibility of having to proposal kernels Q^0 and Q^1 available. There are two obvious ways in which we can combine these and still fulfill the detailed balance condition. The first is to use each of the proposal kernels to produce two separate transition kernels P^0 and P^1 and combine these as before. The second is to create a proposal kernel $Q = \alpha Q^0 + (1-\alpha)Q^1$ and use this in the transition kernel. Both of these options can easily be shown to fulfill the detailed balance condition. In the next section we discuss a third way of combining kernels.

3 Combination of kernels and mode jumping proposals

This section is based on Tjelmeland & Hegstad (2001) and contains a description of mode jumping in MCMC algorithms. Mode jumping is a technique used to achieve good sampling from multi-modal distributions. These distributions appear as the one illustrated in Figure 1 and are characterized by areas of high probability (modes) separated by areas of low probability. The typical Metropolis-Hastings algorithm presented in Hastings (1970) will often give bad mixing when sampling from such a distribution. This is due to the fact that only small changes are proposed in the state vector, which again is due to the fact that large changes will have a small probability of acceptance. By only proposing small changes, the algorithm rarely moves between the modes and the samples will be concentrated around one of the modes. We will discuss how you can overcome this problem by using mode jumping. To explain it we first have to describe a different way of combining proposal kernels than the one discussed in the previous section. Then, using this we discuss the mode jumping idea.

3.1 Combination of kernels

Tjelmeland & Hegstad (2001) discuss an alternative way of combining two proposal kernels in the design of an MCMC algorithm for a continuous Markov chain. We explore the subject both for a continuous Markov chain and a discrete Markov chain, as we apply the technique to a discrete distribution later in the report. In the previous section we explained two ways of combining several proposal kernels in the Metropolis-Hastings algorithm. Here we discuss a third way, which changes the algorithm slightly and defines an acceptance probability different from equation (11). Although our new acceptance probability is not optimal in relation to Peskun's acceptance probability, we will argue that in the case where sampling from Q_0 and Q_1 is computer intensive, this new way of constructing the transition kernel may be better in respect to CPU time if not in the number of iterations.

First, assume that we construct our new transition kernel as,

$$P(A|x) = \frac{1}{2}\int_A q_0(y|x)\alpha_{0,1}(y|x)dy + \frac{1}{2}\int_A q_1(y|x)\alpha_{1,0}(y|x)dy + I(x \in A)r(x), \qquad (12)$$

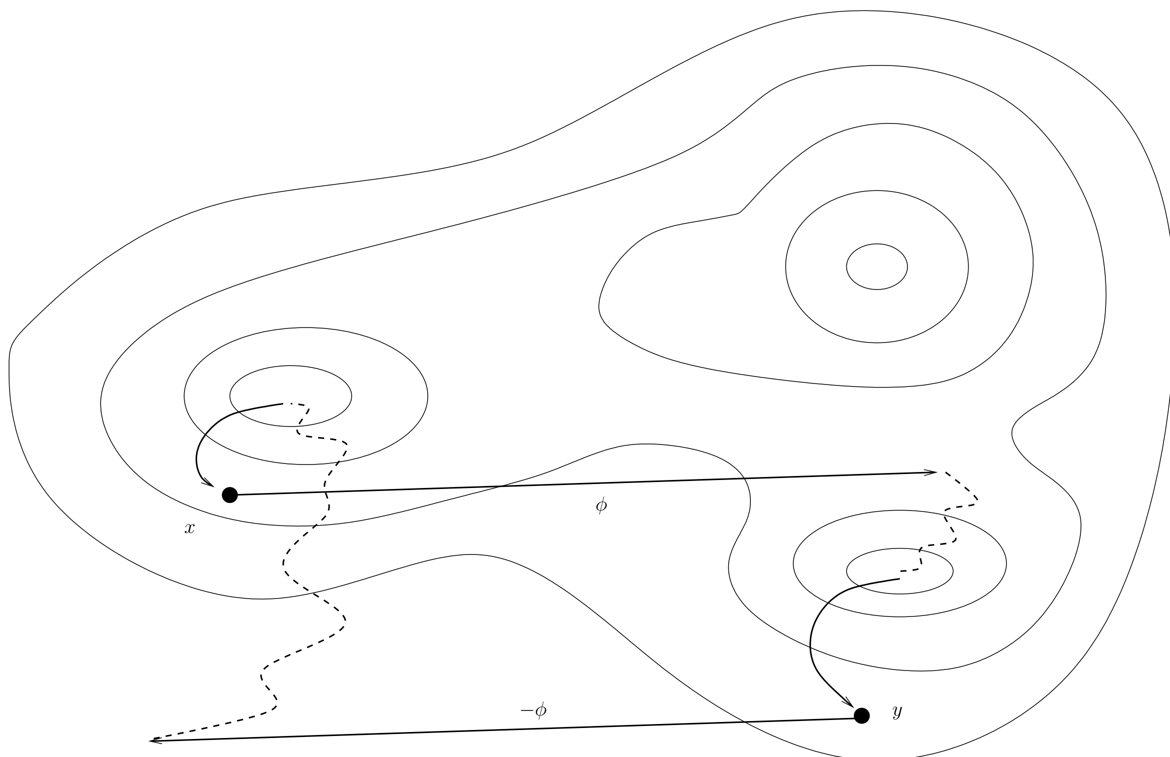

Figure 1: Illustration of one iteration of the mode jumping algorithm. A new proposal y is generated by adding ϕ and locating the center of the closest mode, from which we can sample y. The return probability is found by adding $-\phi$ to y.

for the continuous case and,

$$P(y|x) = \frac{1}{2}q_0(y|x)\alpha_{0,1}(y|x) + \frac{1}{2}q_1(y|x)\alpha_{1,0}(y|x) \text{ for } y \neq x, \tag{13}$$

for the discrete case. The acceptance probabilities $\alpha_{0,1}(y|x)$ and $\alpha_{1,0}(y|x)$ are given as,

$$\alpha_{i,1-i}(y|x) = \min\left\{1, \frac{\pi(y)}{\pi(x)}\frac{q_{1-i}(x|y)}{q_i(y|x)}\right\}, \text{ for } i = 1, 2. \tag{14}$$

The rejection probability $r(x)$ in (12) now becomes,

$$r(x) = \frac{1}{2}\int_{\Re^n} q_0(y|x)(1 - \alpha_{0,1}(y|x))dy + \frac{1}{2}\int_{\Re^n} q_1(y|x)(1 - \alpha_{1,0}(y|x))dy \tag{15}$$

We show that the resulting transition kernel is in fact within the class of Metropolis-Hastings defined by equations (8) and (10) by showing that all we have done is to combine two proposal distributions as we did in Section 2. To get our transition kernel we have simply defined $q(y|x) = \frac{1}{2}q_0(y|x) + \frac{1}{2}q_1(y|x)$ and

$$s(x, y) = \frac{1}{2}(\pi(x)q_0(y|x)\alpha_{0,1}(y|x)+\pi(y)q_0(x|y)\alpha_{0,1}(x|y))(\frac{1}{\pi(x)q(y|x)}+\frac{1}{\pi(y)q(x|y)}). \tag{16}$$

Inserting (16) in (10) gives us the acceptance probability as,

$$\alpha(y|x) = \frac{q_0(y|x)\alpha_{0,1}(y|x) + q_1(y|x)\alpha_{1,0}(y|x)}{q_0(y|x) + q_1(y|x)}. \tag{17}$$

The true difference lies in the way we perform the algorithm. Normally, we would determine which of the proposal distributions Q_0 and Q_1 to use (with probability $\frac{1}{2}$ for each) and propose a new state using that proposal distribution. Then we would calculate the acceptance probability in equation (17) to determine whether we should accept or not. Instead of this we now use the acceptance probability in equation (14) corresponding to our choice of i. We can think of this as using one of the proposal distributions to perform the proposal step, and then the other to perform the return step to our previous state. In other words, our acceptance probability depends on which Q_i was used to propose the new state. Of course this means that our acceptance probability is no longer optimal with respect to the number of iterations. However, in the case where evaluating Q_i is computer intensive it has an advantage over the Peskun (1973) algorithm, as we only need to compute $q_i(y|x)$ and $q_{1-i}(x|y)$ to find the acceptance probability. If we study equation (17), we see that using that acceptance probability we would need to compute $q_i(y|x)$, $q_i(x|y)$, $q_{1-i}(y|x)$ and $q_{1-i}(x|y)$. This means that our new algorithm may be better in the case we are about to study, where evaluating Q_i is computer intensive.

3.2 Mode jumping

In this section we discuss how to sample from a multi-modal distribution using a combination of two specifically designed transition kernels. In a multi-modal distribution

we have areas of high probability separated by areas of low probability. With mode jumping, we hope to achieve direct transitions between the areas of high probability, as this will give us good mixing from the distribution. Transitions between these areas will require large proposed changes in the state vector. However two problems immediately occur with such proposals. First, how do we ensure that the proposed state is in the high probability area, and second, how do we achieve a sufficiently large acceptance probability for such a proposal. Mode jumping overcomes the first of these problems using local optimization, and the second using a combination of kernels as described in the previous section.
Assuming we are currently in a state x, the algorithm is illustrated in Figure 1. First we add a sufficiently large random vector ϕ to our current state x. We then locate a mode by performing a deterministic local minimization of the energy function $U(x) = -\ln(\pi(x))$, started in $x + \phi$. We use the notation $\mu(x + \phi)$ for the location of the minimum we find. With this we can now construct a proposal distribution Q^{ϕ}, as

$$q^{\phi}(y|x) = N(\mu(x + \phi), \Sigma(x + \phi)), \tag{18}$$

where $\Sigma(x + \phi)$ is the inverse Hessian at the minimum, i.e.

$$\Sigma(x + \phi) = [(\nabla^2 U)(\mu(x + \phi)]^{-1}.$$

In other words the idea is to locate the center of another mode in the distribution, and then sample from a normal distribution centered in that mode. By proposing a new state y in this manner we should achieve a transfer between the high probability areas, thus we have overcome the first of our two problems. Now we have to construct another proposal distribution to perform the return step, so as to ensure a high probability of acceptance. Since y is located somewhere in the vicinity of the mode that $\mu(x+\phi)$ found for us, it is not unreasonable that $y - \phi$ should place us somewhere in the vicinity of the mode we left. Again we can perform a deterministic local minimization and construct a second proposal distribution,

$$q^{-\phi}(x|y) = N(\mu(y - \phi), \Sigma(y - \phi)). \tag{19}$$

The acceptance rate can then be calculated from equation (14). In this way, the acceptance rate should be sufficient to give us frequent jumps between the modes in the distribution.

4 Multi-proposal MCMC

Traditional MCMC revolves around the concept of constructing a Markov chain with a distribution that converges in time towards the distribution we wish to sample from. We construct this chain according to the Metropolis Hastings algorithm by splitting the transition probabilities into a proposal part and an accept/reject part. We then sample from the Markov chain until we believe it to have converged. The Multi-proposal MCMC algorithm is based on the same idea, the difference being that instead of proposing only

one new state, we propose several at once. As in the traditional Metropolis-Hastings algorithm we still only accept one new state for each iteration. This section is based on Tjelmeland (2004) and establishes the basic framework for the multi-proposal MCMC algorithm, then shortly discusses acceptance probabilities and parallel implementation of the algorithm. For further information and examples of multi-proposal MCMC we refer to Stormark (2006), Craiu & Lemieux (2005) and Liu, Liang & Wong (2000).

4.1 The idea, and initial setting

Assume we wish to sample from a target distribution $\pi(\cdot)$ continuous on $\Re^n$, using a multi-proposal MCMC algorithm. We start by defining $m \geq 1$ as the number of new proposals in each iteration and introduce the stochastic variable $\kappa \in \{0, 1, \ldots, m\}$ as an assistance variable to keep track of which proposal is the current accepted state, y_κ. In each iteration we generate m new proposals which we store, together with the old state in a vector $\mathbf{y} = \{y_0, y_1, \ldots, y_m\}$. We denote the m new states $\mathbf{y}_{-\kappa}$ using the notation where $\mathbf{y}_{-j} = \{y_0, \ldots, y_{j-1}, y_{j+1}, \ldots, y_m\}$. The idea behind the multi-proposal MCMC algorithm is now to define a Markov chain over the states $\{\{\mathbf{y}^1, \kappa^1\}, \ldots \{\mathbf{y}^N, \kappa^N\}\}$ with an invariant distribution $p(\mathbf{y}, \kappa)$. If the invariant distribution $p(\cdot)$ is constructed correctly each $y^i_{\kappa^i}$ should be distributed according to $\pi(\cdot)$. To define $p(\cdot)$ we start by letting κ have a uniform distribution and define $p_\kappa(\mathbf{y})$ as the conditional distribution of $\mathbf{y}$ given κ. Using Bayes law this gives us,

$$p(\mathbf{y}, \kappa) = \frac{1}{m+1} p_\kappa(\mathbf{y}), \tag{20}$$

as the joint distribution of $\mathbf{y}$ and κ. We further define $p_\kappa(\mathbf{y})$ from $\pi(\cdot)$ as,

$$p_\kappa(\mathbf{y}) = \pi(y_\kappa) q_\kappa(\mathbf{y}_{-\kappa} | y_\kappa). \tag{21}$$

Here $q_k(\mathbf{y}_{-k}|y_k),\ k = 0, 1, \ldots, m$ are the proposal densities for sampling a new set of states given the current one. The proposal distribution can, as in standard MCMC be chosen quite freely. The only real condition is that it is easy to sample from. Later, in Section 7, we test the algorithm on two different proposal distributions. We note that by constructing our invariant distribution in this way, y_κ will be distributed according to $\pi(\cdot)$. The Markov chain itself is then defined by switching between two types of updates. To ensure that we get the correct invariant distribution for this Markov chain, it is important that both steps are invariant with respect to our target distribution $p(\cdot)$. Update (i) is to substitute all the current values of $\mathbf{y}_{-\kappa}$ by values sampled from the proposal distribution $q_\kappa(\mathbf{y}_{-\kappa}|y_\kappa)$. This is the equivalent to sampling from $p(\mathbf{y}_{-\kappa}|y_\kappa, \kappa)$ and is a Gibbs step and thus will automatically be invariant with respect to the target distribution. For more background on the Gibbs algorithm see Roberts & Casella (1999). Update (ii) is to replace the current value of κ with a new one sampled according to a transition matrix $\mathbf{P}(\mathbf{y}) = [P_{k,l}(\mathbf{y})]_{k,l=0}^m$. To ensure that this update is invariant with respect to $p(\mathbf{y}, \kappa)$ we need to have

$$p(\mathbf{y}, l) = \sum_{k=0}^{m} p(\mathbf{y}, k) P_{k,l}(\mathbf{y}) \text{ for } l \in \{0, 1, \ldots, m\} \text{ and } \mathbf{y} \in \Re^{n(m+1)}. \tag{22}$$

If we insert (20) into (22) this rewrites into

$$p_l(\mathbf{y}) = \sum_{k=0}^{m} p_k(\mathbf{y}) P_{k,l}(\mathbf{y}) \text{ for } l \in \{0, 1, \dots, m\} \text{ and } \mathbf{y} \in \Re^{n(m+1)}. \tag{23}$$

In addition to this, $\mathbf{P}(\mathbf{y})$ must also fulfill the requirements of a transition matrix, that is

$$P_{k,l}(\mathbf{y}) \geq 0 \text{ for } k, l \in \{0, 1, \dots, m\} \text{ and } \mathbf{y} \in \Re^{n(m+1)} \tag{24}$$

$$\sum_{l=0}^{m} P_{k,l}(\mathbf{y}) = 1 \text{ for } k \in \{0, 1, \dots, m\} \text{ and } \mathbf{y} \in \Re^{n(m+1)}. \tag{25}$$

We discuss the construction of the transition matrix in Section 4.2. In studying the algorithm it is important to realize that the invariant distribution of our Markov chain is $p(\mathbf{y}, \kappa)$ and not $\pi(\cdot)$, as we are accustomed to. However, the $y^i_{k^i}$'s will be distributed according to $\pi(\cdot)$. Despite this we can easily compare our algorithms update steps to the propose and accept/reject structure of the Metropolis-Hastings algorithm. Update (i) is the proposal step, we propose m new states according to our proposal distribution $q_\kappa(\mathbf{y}_{-\kappa}|y_\kappa)$. Update (ii) is the accept/reject step, where a change of the value of κ indicates an acceptance of a new state, whereas not changing the value of κ indicates a rejection of all the new states. All that remains is to specify the proposal distribution and transition matrix.

The fact that we propose several new potential states for each iteration means that there are two options we should explore. The first of these is to use control variates to create a new estimate for the mean. This is a technique that has been used in traditional MCMC algorithms, but may become even more important in the multi-proposal setting. Very often we are interested in estimating the mean based on our samples, and since we generate several samples for each iteration it would seem like a waste of information to not include them in our estimator. We explore this in Section 6. The second idea is related to the implementation of the algorithm. Since several states are proposed for each iteration, and especially if this is computer intensive (as we will later see that it can bee) it would be optimal to apply parallelization. We discussed this further in Section 4.3.

4.2 Transition matrix

Unlike standard MCMC, where the choice of acceptance probability is practically given by Peskuns optimization theory, in multi proposal MCMC we have several choices for the transition matrix. Here we will outline one alternative that we have applied in our examples. This is probably in no way the optimal choice, but it suffices for our examples. An obvious choice of P which clearly fulfills the requirements set in the previous section is,

$$P_{k,l}(\mathbf{y}) = \frac{p_l(\mathbf{y})}{\sum_{j=0}^{m} p_j(\mathbf{y})}. \tag{26}$$

This is easily verified by inserting the expression in equations (23) and (25). However, this choice of P can be easily improved by running it through a so called Peskunization process. For more background on this algorithm we refer you to Tjelmeland (2004) where the algorithm is presented in depth.
The algorithm then runs as follows.

1. Set $t = 0$ and let $P^0(\boldsymbol{y})$ be the transition matrix defined in (26).
2. Set $A^t = \{k : P^t_{k,k} > 0, k = 0, 1, \ldots, m\}$.
3. If $|A^t| \leq 1$, set $P(\boldsymbol{y}) = P^t(\boldsymbol{y})$ and end the process.
4. Set
$$u^t = \min_{k \in A^t} \left(\frac{1 - \sum_{l \notin A^t} P^t_{k,l}(\boldsymbol{y})}{\sum_{l \in A^t \setminus \{k\}} P^t_{k,l}(\boldsymbol{y})} \right). \tag{27}$$
5. Let $P^{t+1}(\boldsymbol{y})$ be defined as,
$$P^{t+1}_{k,l}(\boldsymbol{y}) = P^t_{k,l}(\boldsymbol{y}) \text{ if } k \notin A^t \text{ or } l \notin A^t \tag{28}$$
$$P^{t+1}_{k,l}(\boldsymbol{y}) = u^t P^t_{k,l}(\boldsymbol{y}) \text{ if } k \in A^t \text{ and } k \neq l \tag{29}$$
$$P^{t+1}_{k,k}(\boldsymbol{y}) = 1 - \sum_{l \neq k} P^t_{k,l}(\boldsymbol{y}) \text{ for } k \in A^t \tag{30}$$
6. Assign $t = t + 1$ and go to 2.

In the simulation algorithm one should of course modify this so that only row κ is computed, as this is all you need.

4.3 Parallel implementation

A multi-proposal MCMC algorithm is ideally suited for parallelization. With parallelization we mean distributing tasks among several processors, thus completing several task simultaneously. A multi-proposal MCMC sampler will consist of some parallel code and some sequential code. With m proposals we will need one ”master” processor, and m ”slave” processors. In each iteration the master starts by distributing the current state y_k to all the m slave processors. Each slave then generates a new proposal according to our proposal scheme, and returns the proposal as well as the probability $\pi(y_l)$ of the proposal. The Master then construct the transition matrix $\mathbf{P}(\mathbf{y})$ and selects a new state κ, which completes one iteration. When discussing parallel programs we need to introduce the concept of ”overhead”. When we speak of overhead in parallel programs we are referring to the time lag that occurs from message passing between processors. Transferring messages between processors is not a simultaneous process, and the more processors involved, the larger the overhead. Thus there is an increase in run time associated with a large number of processors communicating. This means that if an MCMC algorithm has a simple and fast proposal scheme, the time gain from having each processor generate it's own proposal may disappear because of the overhead. If however, the proposals are CPU intensive, the time gain will become large compared to the overhead.

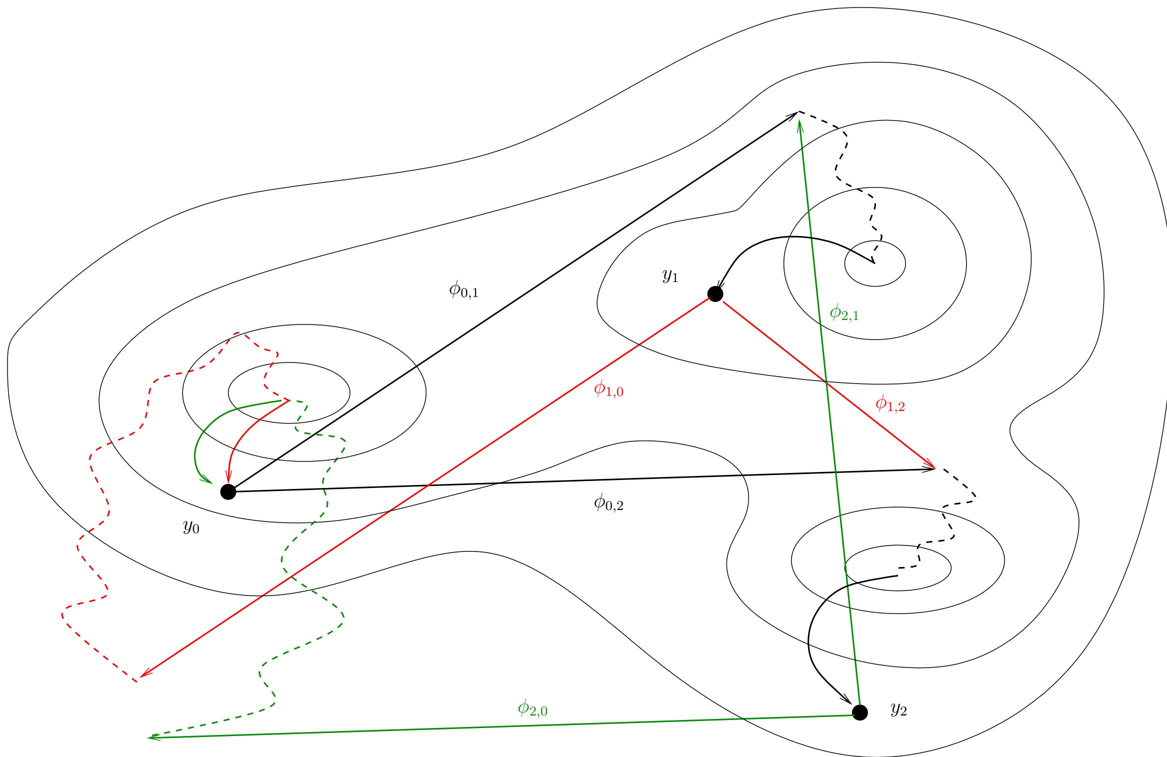

Figure 2: Illustration of one iteration of a multi-proposal mode jumping algorithm. First y_1 and y_2 are proposed by generating the vectors $\phi_{0,1}$ and $\phi_{0,2}$. The return probabilities are then calculated by using the vectors $\phi_{1,0}$, $\phi_{1,2}$ and $\phi_{2,0}$, $\phi_{2,1}$ as illustrated in red and green respectively.

5 Mode jumping with multiple proposals

We now wish to combine the two ideas presented so far in this paper and create a multi-proposal MCMC algorithm with a mode jumping proposal distribution. This means suggesting m new states in each iteration and using a proposal distribution as introduced in the mode-jumping section. The two algorithms can be combined quite easily, requiring only a small expansion of the notation. We start by illustrating the algorithm with a walk-through of an iteration with $m = 2$. We then define the algorithm more precisely and introduce necessary notation, before finishing with some remarks and comments about the algorithm.

5.1 Multi-proposal mode jumping algorithm

Assume we have a probability distribution with several modes that we wish to sample from, using multiple proposals in each iteration. A walk-through of an iteration with $m = 2$ is presented here, and illustrated in Figure 2. Assume we have the multi proposal framework introduced in Section 4, with states $\mathbf{y} = \{y_0, y_1, y_2\}$ and assistant variable

κ. At the start of the iteration we assume that $\kappa = 0$. As in Section 3 we want to move between the different modes. Each of our two proposals is generated by adding a large independent random vector $\phi_{0,j}$ for $j = 1, 2$. As before we then locate the local center of high probability, through a deterministic local minimization routine, which we denote $\mu(y_0 + \phi_{0,j})$. Our two independent proposals can then be generated as before $y_j \sim N(\mu(y_0 + \phi_{0,j}), \Sigma(y_0 + \phi_{0,j}))$, where $\Sigma(y_0 + \phi_{0,j})$ again is the inverse Hessian at the minimum. So far we haven't really done anything new. The next step is to calculate the acceptance probability. In Section 3 we created two different transition kernels to achieve a high acceptance rate. Now we have to construct a family of $m + 1$ transition kernels. We can see from equation (26) that in calculation of the acceptance probabilities we need to calculate the probability of going from our new state to all the others. This means that to achieve an acceptably high acceptance rate, we need to have a sufficiently large probability of going to all the other states from our potential new states, not just the previous state as in Section 3. A second important thing to keep in mind at this stage is the performance speed of the algorithm in CPU time. In Section 3 we applied a second minimization in the return kernel. If we where to perform a minimization for each of the other states, we would have to perform m^2 minimizations for each iteration. It is time consuming to perform deterministic minimization, especially in high dimensions, so this would be unacceptable as we increase m. This means that if possible, we would like to reuse the minimizations we calculated during the generation of the proposals. With this in mind we design a class of transition kernels for returning to the other states. Consider first y_1. To get back to state y_0 we apply the same technique as in Section 3 and add a vector $\phi_{1,0} = -\phi_{0,1}$. As before, it is reasonable to assume, that a deterministic local minimization routine started from $y_1 - \phi_{0,1}$ should give us the location of a mode close to y_0. To get to state y_2 however, we need to do something different. We add a vector $\phi_{1,2} = \phi_{0,2} - (y_1 - y_0)$, see Figure 2. This puts us in the location where we started our previous local minimization, so we can use our results from that to find the local center of high probability. For y_2 we do exactly the same using the vectors $\phi_{2,0} = -\phi_{0,2}$ and $\phi_{2,1} = \phi_{0,1} - (y_2 - y_0)$. Figure 2 illustrates the process better than words. In this manner we only perform $2m$ deterministic local minimization's for each iteration, which is acceptable.

5.2 Notation and Remarks

In this section we introduce some notation and explore the multi-proposal mode jumping algorithm in more detail. We start by defining the invariant distribution of our Markov chain, which is almost the same as in (31), we now have,

$$p(\mathbf{y}, \kappa, \boldsymbol{\phi}_{-\kappa}) = \frac{1}{m+1} p_\kappa(\mathbf{y}, \boldsymbol{\phi}_{-\kappa}), \tag{31}$$

where we are using the same notation as before, and $\boldsymbol{\phi}_{-\kappa} = \{\phi_{\kappa,0}, \ldots, \phi_{\kappa,\kappa-1}, \phi_{\kappa,\kappa+1}, \ldots, \phi_{\kappa,m}\}$ contains the vectors introduced in the previous section. As in Section 4 we construct $p_\kappa(\cdot)$ in such a way that $y_\kappa \sim \pi(\cdot)$,

$$p_\kappa(\mathbf{y}, \boldsymbol{\phi}_{-\kappa}) = \pi(y_\kappa) f(\boldsymbol{\phi}_{-\kappa}) q_\kappa(\mathbf{y}_{-\kappa} | y_\kappa, \boldsymbol{\phi}_{-\kappa}). \tag{32}$$

The vectors $\phi_{\kappa,j}$ are independent of each other and are typically distributed according to a normal distribution with a large variance so that they are easy to sample. Our proposal distribution $q_\kappa(\mathbf{y}_{-\kappa}|y_\kappa, \boldsymbol{\phi}_{-\kappa})$ can now be considered a class of proposal kernels defined as,

$$q_l(\mathbf{y}_{-l}|y_l, \boldsymbol{\phi}_{-l}) = \prod_{j=0, j\neq l}^{m} N_n(\mu(y_l + \phi_{l,j}), \Sigma(y_l + \phi_{l,j}))(y_j), \text{ for } l \in \{0, \dots, m\}, \tag{33}$$

where the vectors $\phi_{l,j}$ are defined as,

$$\phi_{l,j} = \begin{cases} -\phi_{\kappa,j} & \text{if} \quad j = \kappa \text{ and } l \neq \kappa, \\ \phi_{\kappa,j} - (y_l - y_\kappa) & \text{if} \quad j \neq \kappa \text{ and } l \neq \kappa. \end{cases} \tag{34}$$

As before we intend to sample from this distribution by switching between two types of updates, and we must ensure that both are invariant with respect to $p(\mathbf{y}, \kappa, \boldsymbol{\phi}_{-\kappa})$. Update (i) generate a new $\boldsymbol{\phi}_{-\kappa}$ and sample $\mathbf{y}_{-\kappa}$ from our proposal distribution. As before this step is a Gibbs step since we condition on the remaining variables, and as such automatically be invariant. Update (ii) sample a new κ according to the transition matrix $\mathbf{P}(\mathbf{y}) = [P_{k,l}(\mathbf{y})]_{k,l=0}^{m}$ and update $\boldsymbol{\phi}_{-\kappa}$ according to equation (34). The requirements on the matrix $\mathbf{P}(\mathbf{y})$ will be equivalent to those in Section 4, and the matrix can be constructed as before as described in Section 4.2.
There are several aspects of the algorithm which should be mentioned.

1. Obviously the algorithm should not be used on it's own as it is presented here. As with the single proposal mode jumping algorithm, it should be combined with a standard Metropolis-Hastings algorithm to produce samples around the different modes. Mode jumping is then used to move between the modes.

2. As discussed in Section 4.3 multi-proposal MCMC algorithms are very well suited for parallelization. The multi-proposal mode jumping algorithm may be even better suited for this as each of the proposals are very CPU intensive. We examine this further in the examples in Section 7.

3. When implementing the algorithm one should keep in mind that it needs to be quite robust. The generating of proposals includes several numeric optimization steps and these will occasionally return errors. Steps must be taken to handle these errors properly.

6 Control variates and use of rejected states

In this section we discuss the use of control variates for variance reduction in the estimate of our mean. We start by giving a short introduction to what a control variate is, and then show how we can design a control variate to use with the multi-proposal MCMC algorithm. For further information we refer to Tjelmeland & Hammer (2005) and Tjelmeland (2004).

6.1 Using control variates to estimate means

Assume we have samples $x^1, \ldots, x^N$ from our sample distribution $\pi(\cdot)$ and wish to estimate the mean $\mu = E[f(x)]$. It is desirable in a simulation setting to achieve a low variance on our estimate of the mean, and as such it is natural to seek a way to reduce the variance. We hope to achieve this by introduction of the control variate.
Assume that our estimate for the mean is the traditional sample mean (2), as usual. The idea is then to introduce a random variable v, the control variate. Assuming that we define v such that it is correlated to $\hat{\mu}$ and has expectation $E(v) = 0$ we can design a new unbiased estimator for μ,

$$\tilde{\mu} = \hat{\mu} + cv, \tag{35}$$

for any value of c. Now we wish to minimize, as a function of c, the variance of our new estimator (35),

$$\mathrm{Var}(\tilde{\mu}) = \mathrm{Var}(\hat{\mu}) + c^2\mathrm{Var}(v) + 2c\mathrm{Cov}(\hat{\mu}, v). \tag{36}$$

We find the minimum of this expression by taking the derivative with respect to c and solving this when set equal to 0,

$$\frac{\partial \mathrm{Var}(\tilde{\mu})}{\partial c} = 2c\mathrm{Var}(v) + 2\mathrm{Cov}(\hat{\mu}, v). \tag{37}$$

Setting this equal to 0 and solving with respect to c gives us,

$$c_{opt} = -\frac{\mathrm{Cov}(\hat{\mu}, v)}{\mathrm{Var}(v)}. \tag{38}$$

We can calculate the relative variance reduction we achieve by using c_{opt},

$$\begin{aligned}
\frac{\mathrm{Var}(\hat{\mu}) - \mathrm{Var}(\tilde{\mu})}{\mathrm{Var}(\hat{\mu})} &= \frac{\mathrm{Var}(\hat{\mu}) - \mathrm{Var}(\hat{\mu}) + \frac{\mathrm{Cov}(\hat{\mu},v)^2}{\mathrm{Var}(v)}}{\mathrm{Var}(\hat{\mu})} \\
&= \frac{\mathrm{Cov}(\hat{\mu}, v)^2}{\mathrm{Var}(\hat{\mu})\mathrm{Var}(v)} = \mathrm{Corr}(\hat{\mu}, v)^2.
\end{aligned}$$

We see from this that we want to design our control variate so that it is has a high correlation with $\hat{\mu}$. By achieving variance reduction of our mean in this manner, we save time by being able to run our algorithm for a shorter time after convergence. With control variates we may reduce the time it takes to produce a good estimate with low variance. We discuss the estimation of c_{opt} in the next Section.

6.2 Using all proposed states to estimate the mean

This section is based on Tjelmeland (2004) and discusses the design of our control variate v. Assume we have run a multi-proposal MCMC algorithm, and have generated $\{\mathbf{y}^i, \kappa^i\}_{i=1}^N$ where $\mathbf{y}^i = (y_0^i, \ldots, y_m^i)$ as discussed in Section 4. We wish to use all the

proposed states to estimate the mean, using the control variate method in the previous section. We consider the control variate,

$$v = \frac{1}{N}\sum_{i=1}^{N}\Big[\sum_{l=0}^{m} w_{\kappa^i,l}(\mathbf{y}^i)f(y_l^i) - f(y_{\kappa^i}^i)\Big], \tag{39}$$

where $\mathbf{w}(\mathbf{y}) = [w_{k,l}(\mathbf{y})]_{k,l=0}^{m}$ is a weigth matrix function. As this is clearly correlated with $\hat{\mu}$ we hope to reduce the variance of our estimate by applying the estimator in equation (35). However for this to be a valid control variate, $\mathbf{w}(\mathbf{y})$ needs to be such that,

$$E_p(v) = 0. \tag{40}$$

To find the requirements this puts on $\mathbf{w}(\mathbf{y})$, we insert expression (39) in the left hand side of (40),

$$E_p(v) = E_p\Big(\sum_{l=0}^{m} w_{\kappa,l}(\mathbf{y})f(y_l) - f(y_\kappa)\Big). \tag{41}$$

Since $E[f(y_\kappa)] = f(\mu)$ we can rewrite this to,

$$E_p(v) = \sum_{k=0}^{m}\left[\int \frac{1}{m+1}p_k(\mathbf{y})\sum_{l=0}^{m} w_{k,l}(\mathbf{y})f(y_l)d\mathbf{y}\right] - f(\mu). \tag{42}$$

We then separate $w_{k,k}(\mathbf{y})$ from the innermost sum,

$$\begin{aligned} E_p(v) = {} & \frac{1}{m+1}\sum_{k=0}^{m}\left[\int p_k(\mathbf{y})\sum_{l\neq k}^{m} w_{k,l}(\mathbf{y})f(y_l)d\mathbf{y}\right] + \\ & \frac{1}{m+1}\sum_{k=0}^{m}\left[\int p_k(\mathbf{y})f(y_k)w_{k,k}(\mathbf{y})d\mathbf{y}\right] - f(\mu). \end{aligned} \tag{43}$$

By requiring $\sum_{l=0}^{m} w_{k,l}(\mathbf{y}) = 1$, we can rewrite this,

$$\begin{aligned} E_p(v) = {} & \frac{1}{m+1}\sum_{k=0}^{m}\left[\int p_k(\mathbf{y})\sum_{l\neq k}^{m} w_{k,l}(\mathbf{y})f(y_l)d\mathbf{y}\right] + \\ & \frac{1}{m+1}\sum_{k=0}^{m}\left[\int p_k(\mathbf{y})f(y_k)\Big(1 - \sum_{l\neq k}^{m} w_{k,l}(\mathbf{y})\Big)d\mathbf{y}\right] - f(\mu). \end{aligned} \tag{44}$$

Since $\sum_{k=0}^{m}\Big[\int \frac{1}{m+1}p_k(\mathbf{y})f(y_k)d\mathbf{y}\Big] = f(\mu)$, we get,

$$\begin{aligned} E_p(v) = {} & \frac{1}{m+1}\sum_{k=0}^{m}\left[\int p_k(\mathbf{y})\sum_{l\neq k}^{m} w_{k,l}(\mathbf{y})f(y_l)d\mathbf{y}\right] - \\ & \frac{1}{m+1}\sum_{k=0}^{m}\left[\int p_k(\mathbf{y})f(y_k)\sum_{l\neq k}^{m} w_{k,l}(\mathbf{y})d\mathbf{y}\right]. \end{aligned} \tag{45}$$

We then rewrite the sums and switch the summation indices k and l in the last sum,

$$E_p(v) = \frac{1}{m+1}\int \sum_{k=0}^{m}\sum_{l\neq k}^{m} w_{k,l}(\mathbf{y})f(y_l)p_k(\mathbf{y})d\mathbf{y} - \frac{1}{m+1}\int \sum_{k=0}^{m}\sum_{l\neq k}^{m} w_{l,k}(\mathbf{y})f(y_l)p_l(\mathbf{y})d\mathbf{y}. \tag{46}$$

For $E_p(v) = 0$ we see that a sufficient condition is,

$$\sum_{l\neq k}^{m} w_{k,l}(\mathbf{y})p_k(\mathbf{y}) = \sum_{l\neq k}^{m} w_{l,k}(\mathbf{y})p_l(\mathbf{y}) \tag{47}$$

Again using $\sum_{l=0}^{m} w_{k,l}(\mathbf{y}) = 1$ we can write this as,

$$(1 - w_{k,k})p_k(\mathbf{y}) = \sum_{l\neq k}^{m} w_{l,k}(\mathbf{y})p_l(\mathbf{y}) \tag{48}$$

Which in turn is equal to,

$$p_k(\mathbf{y}) = \sum_{l=0}^{m} w_{l,k}(\mathbf{y})p_l(\mathbf{y}) \tag{49}$$

This requirement combined with $\sum_{l=0}^{m} w_{k,l}(\mathbf{y}) = 1$ will thus be sufficient conditions for our control variate to be unbiased. We notice that these conditions on $\mathbf{w}(\mathbf{y})$ closely resemble the conditions on our transition matrix $\mathbf{P}(\mathbf{y})$, see equations (23) and (25), the only difference being that there is no requirement that the elements of $\mathbf{w}(\mathbf{y}) \geq 0$. Therefore we can follow the same approach as in Section 4.2 and choose

$$w_{k,l}(\mathbf{y}) = \frac{p_l(\mathbf{y})}{\sum_{j=0}^{m} p_j(\mathbf{y})} \text{ for } k \neq l \text{ and} \tag{50}$$

$$w_{k,k}(\mathbf{y}) = 1 - \sum_{l\neq k} w_{k,l}(\mathbf{y}). \tag{51}$$

Inserting this into equation (39), we get

$$\begin{aligned} v &= \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{l\neq k^i}\frac{p_l(\mathbf{y}^i)}{\sum_{j=0}^{m} p_j(\mathbf{y})}f(y_l^i) + \Big(1 - \sum_{l\neq k^i}\frac{p_l(\mathbf{y}^i)}{\sum_{j=0}^{m} p_j(\mathbf{y})}\Big)f(y_{k^i}^i) - f(y_{k^i}^i)\right], \\ &= \frac{1}{N}\sum_{i=1}^{N}\left[\frac{\sum_{l\neq k} p_l(\mathbf{y}^i)\{f(y_l^i) - f(y_{k^i}^i)\}}{\sum_{j=0}^{m} p_j(\mathbf{y}^i)}\right]. \end{aligned} \tag{52}$$

We can then use this as our control variate. To calculate c_{opt} we will need to estimate the values of $\text{Cov}(\hat{\mu}, v)$ and $\text{Var}(v)$ since these are unknowns. Although there are other

ways of estimating these values, we have chosen a simple method applied in Tjelmeland & Hammer (2005). We divide our N Metropolis-Hastings iterations after convergence into M batches. By choosing M such that the batch length N/M is much larger than the correlation length of the Markov chain, we can assume that the batches are close to independent. If we denote the calculated mean and control variate in batch i as $\hat{\mu}^i$ and v^i respectively, then our estimate of c_{opt} becomes,

$$c_{opt} = -\frac{\frac{1}{M}\sum_{i=1}^{M}(\hat{\mu}^i - \hat{\mu})(v^i - v)}{\frac{1}{M}\sum_{i=1}^{M}(\hat{\mu}^i - \hat{\mu})^2}. \tag{53}$$

To chose M we plot the estimated auto-correlation function of the Markov chain. In Figure 3 we have plotted an example of the estimated auto-correlation function.

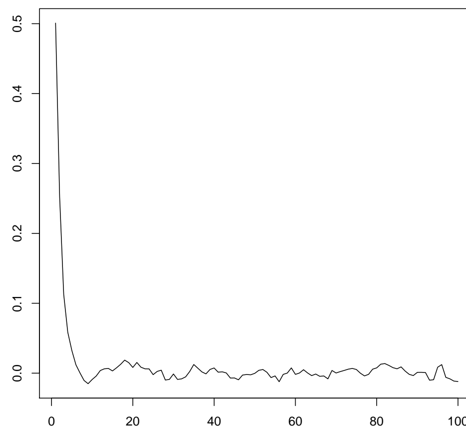

Figure 3: Auto-correlation function for the Markov chain with $m = 16$ in the mode jumping example.

7 Examples

So far in the report we have introduced several techniques which can be applied to MCMC algorithms, the mode jumping proposals, the multi-proposal MCMC algorithm, and the control variate. In this section we test these techniques in two examples, both of which are implemented according to the parallelization scheme in Section 4.3 and run for various values of m. The first example is a toy example where we sample from a multi-Gaussian distribution using random-walk proposals. In the second example we revisit a mixture model for a data-set originally presented in Brooks, Morgan, Rideout & Pack (1997) and later explored in Tjelmeland & Hegstad (2001) and Tjelmeland & Hammer (2005). This example is our main focus where we test the multi-proposal mode jumping

algorithm. The algorithms were implemented in C and the runs performed on a cluster of workstations at the students laboratory using MPI inter-process communication. We were also given the opportunity to run the Gaussian example on the supercomputer Njord at NTNU, which is capable of linking far more processors than the students laboratory. Unfortunately, due to time restraints, we were not able to run the mode jumping example on Njord as well. Our examples where chosen to illustrate how well parallel implementation is suited for different kinds of MCMC algorithms. The proposal scheme in the Gaussian example is very simple compared to the proposal scheme in the mode jumping example. For the Gaussian example we use a simple random walk scheme while in the mode jumping example each proposal requires several numerical optimizations, which is CPU intensive. With a parallelization scheme like the one presented in Section 4.3 we expect the mode jumping example to be very well suited for parallelization, while the Gaussian example should mark the other end of the scale. In our analysis of the results from both examples we start with a general look at the results. This includes a quick look at convergence and the distribution of our samples, as well as looking at the acceptance rates and the time it takes for our sampler to finish (run-time). We denote the run-time per iteration of a sampler with m proposals by t_m. We also study the variance of our estimators and give some confidence intervals. After this we get to the primary focus of our analysis which is centered around three aspects of the algorithm. First, how does the relative variance reduction achieved by using $\tilde{\mu}$ instead of $\hat{\mu}$ change for various values of m. Second, how much variance reduction do we get for increasing values of m. And last, perhaps most importantly, how much variance reduction do we get for increasing values of m as a function of time. To estimate the variance of our estimators we applied the technique introduced at the end of Section 6.1. The samples from a run are split into M batches, with M chosen so that the batches can be considered approximately independent.

7.1 Gaussian toy example

In this example we set $\pi(\cdot)$ to be a two dimensional normal distribution with mean equal to zero and a covariance matrix equal to the identity matrix. We look at the first element of $\mathbf{x}$ and define $f(x) = x_1$, which we of course know to have mean $E(x_1) = 0$. To sample from the distribution we apply a multi-proposal algorithm as it is presented in Section 4, and calculate control variates to produce two estimators for the mean, $\hat{\mu}$ and $\tilde{\mu}$, as discussed in Section 6. We define the proposal distribution $q_k(\mathbf{y}_{-k}|y_k)$ by generating m new proposals $\mathbf{y}_{-k}$ according to the following algorithm,

1. sample $\phi \sim N(y_k, \frac{1}{2}\sigma^2 I)$,
2. independently for $j = 0, \dots, k-1, k+1, \dots, m$ sample $y_j \sim N(\phi, \frac{1}{2}\sigma^2 I)$.

Here σ^2 is the variance we chose to use for our proposals. One should note that for $m = 1$ this simplifies to the random-walk proposal. We ran the algorithm for 100000 iterations, for all combinations of $m = 1, 2, 4, 8, 16$ and $\sigma^2 = 1, 2, 4, 8$. The algorithm was implemented in parallel according to the description in Section 4.3.

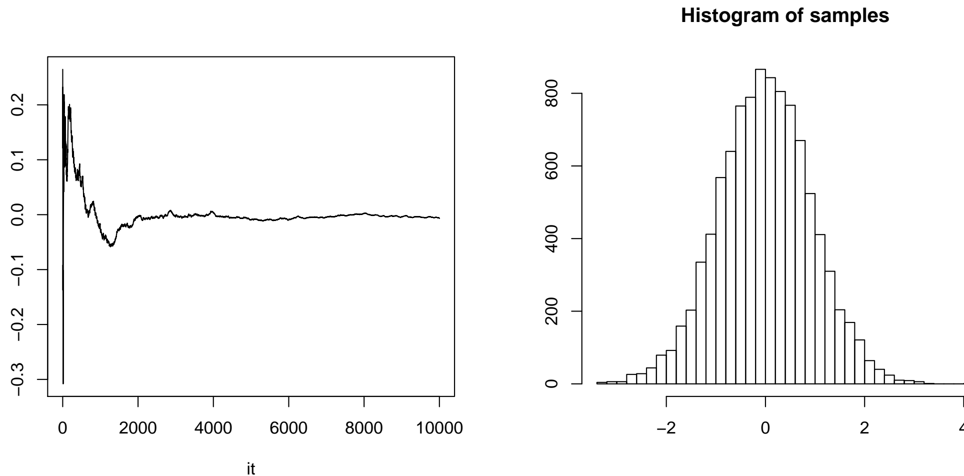

Figure 4: (Gaussian example:) Convergence diagnostics for $m = 16$ and $\sigma^2 = 8$. Cumulative mean plot (left) and histogram (right) of $f(x)$ for the last 10000 generated values.

We start the analysis by taking a general look at the results. Figure 4 shows a cumulative mean plot as well as a histogram of the last 10000 iterations from a run with $m = 16$ and $\sigma^2 = 8$. It is clear from the cumulative mean plot that the algorithm seems to have converged, and as expected the distribution of the samples appears to be the normal distribution centered in 0. We achieved similar results for the other runs. Figure 5 shows the acceptance rates as well as the relative run-times ($\frac{t_m}{t_1}$) of the algorithm, for various values of m. In the plot the colors red, green, blue and black correspond to $\sigma^2 = 1, 2, 4$ and 8 respectively. The x-axis corresponds to increasing values of m on a $log2$ scale. As we can see from the figure the acceptance rates climb towards 1 as we increase m. This is as one would expect, since by increasing the number of proposals pr iteration we would expect to get more good proposals. From the same plot we also see that the acceptance rates drop as σ^2 increases, since an increase in σ^2 means that we will be attempting larger jumps, and as such will get a lower probability of acceptance. From the plot on the right in Figure 5 we can see that there is a certain amount of overhead in the algorithm. Proposing a new state is not computer intensive and as such only a little time is saved by having each processor propose its own state, since the time saved partially disappears in the overhead increase. Running the algorithm with 16 proposals pr iteration takes approximately 8 times as long as running it with only 1 proposal pr iteration. The relative run-time also seems to increase faster as m becomes larger. This is probably caused by two things. First the run-time of the sequential part of the code will increase due to an increase in the size of the transition matrix $\mathbf{P}$. And second, as the number of processors that need to communicate increase the overhead becomes larger. If we study the graph closely there appears to be a slight turning point at $m = 4$ ($log_2(m) + 1 = 3$ in the figure). The small differences in run-time for various

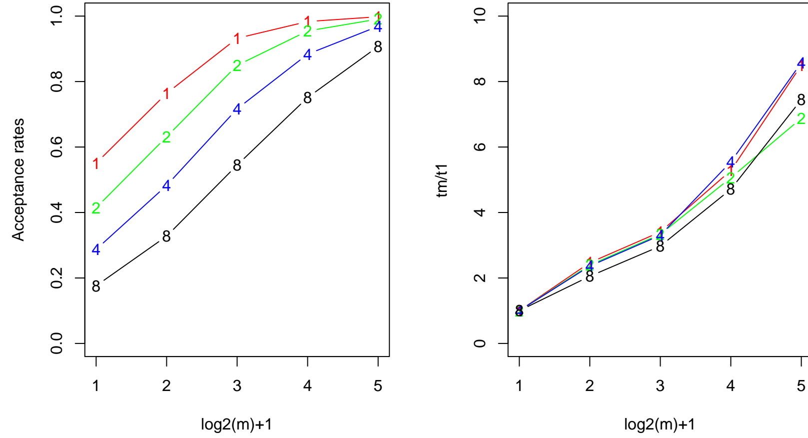

Figure 5: (Gaussian example:) Acceptance rates (right) and run-times (left) for $\sigma = 1, 2, 4, 8$ (red, green, blue and black respectivley) and $m = 1, 2, 4, 8, 16$.

m	$\sigma^2 = 1$	$\sigma^2 = 2$	$\sigma^2 = 4$	$\sigma^2 = 8$	m	$\sigma^2 = 1$	$\sigma^2 = 2$	$\sigma^2 = 4$	$\sigma^2 = 8$
1	0.0739	0.0646	0.0612	0.0933	1	0.0598	0.0519	0.0521	0.0816
2	0.0434	0.0388	0.0387	0.0465	2	0.0339	0.0299	0.0302	0.0381
4	0.0219	0.0206	0.0235	0.0279	4	0.0134	0.0138	0.0166	0.0206
8	0.0197	0.0176	0.0159	0.0198	8	0.0111	0.0103	0.0113	0.0135
16	0.0164	0.0133	0.0129	0.0130	16	0.0082	0.0068	0.0069	0.0084

Table 1: (Gaussian example:) Estimated variance of $\hat{\mu}$ (left) and $\tilde{\mu}$ (right).

values of σ^2 are random variances in the run time. We have verified this by running the test procedure several times. Figure 6 shows the variance of our estimator for μ, the left plot shows the estimated variance of $\hat{\mu}$, while the right plot show the estimated variance of $\tilde{\mu}$. The corresponding numbers are shown in Table 1. From the plots we can see that the variance of $\tilde{\mu}$ is somewhat smaller than the variance of $\hat{\mu}$ and the shape of the curves appear to be roughly equal. From the plots we can see that the curves appear to level out as m increases, the gain is the greatest for the smaller values of m. In this respect the runs with $\sigma^2 = 8$ distinguish themselves, we get a large decrease in variance when going from $m = 1$ to $m = 2$. With large proposed changes, like we get with $\sigma^2 = 8$, we get a low acceptance rate and a doubling of the acceptance rate has a large impact on the variance of our estimator. Figure 7 shows confidence intervals for μ as functions of m for different values of σ^2. Since we saw in Figure 6 that the variance decreased as a function m, we would expect to see the confidence intervals behave as they do in Figure 7. As m increases our confidence intervals become smaller, and from the solid line we can see that using $\tilde{\mu}$ as an estimator further reduces the size of the intervals.

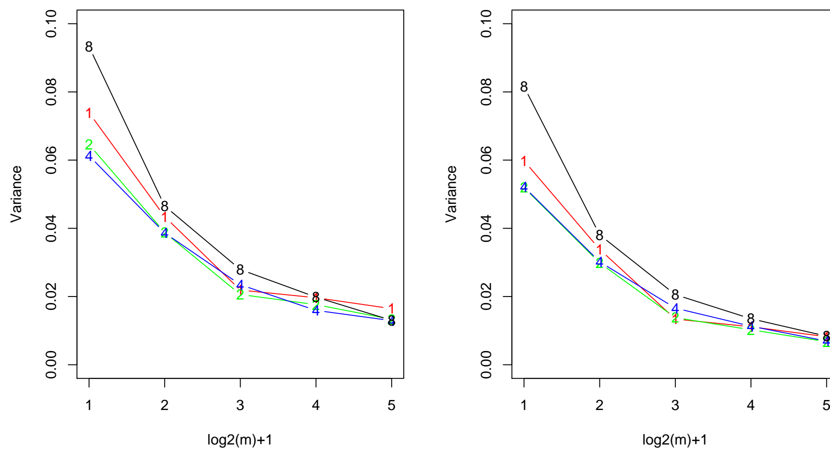

Figure 6: (Gaussian example:) Estimated variance of $\hat{\mu}$ (left) and $\tilde{\mu}$ (right) for $\sigma = 1, 2, 4, 8$ and $m = 1, 2, 4, 8, 16$.

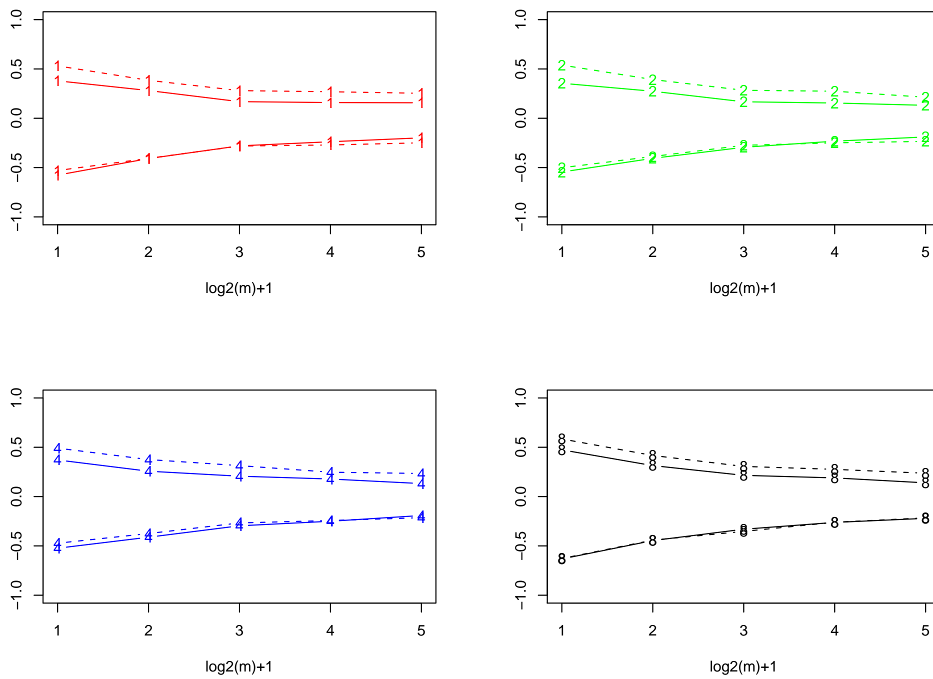

Figure 7: (Gaussian example:) 95% Confidence intervals for μ using $\tilde{\mu}$ (solid line) and $\hat{\mu}$ (dashed line) for $\sigma = 1$, (upper left) 2, (upper right) 4, (lower left) 8 (lower right) and $m = 1, 2, 4, 8, 16$.

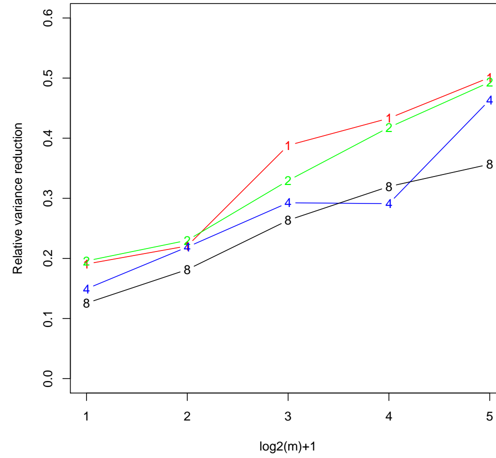

Figure 8: (Gaussian example:) Relative variance reduction with $\tilde{\mu}$ instead of $\hat{\mu}$ for $m = 1, 2, 4, 8, 16$.

We now wish to study the variance reduction we get from the control variates for various values of m. In Figure 8 we have plotted the relative variance reduction, $\frac{\text{Var}(\hat{\mu})-\text{Var}(\tilde{\mu})}{\text{Var}(\hat{\mu})}$, when using $\tilde{\mu}$ instead of $\hat{\mu}$. From the figure we can see that we get a relative variance reduction of $0.1 - 0.2$ for $m = 1$, which increases to around $0.4 - 0.5$ for $m = 16$. We can see from the plot that the benefit of control variates increases as m increases. We can also see that the gain from control variates seems to be less for large values of σ^2. Intuitively this is as one might expect, with more proposals we still only choose one, so more information goes into the control variate, thus we would expect it to be more valuable. With larger values for σ^2 we would expect there to be less good proposals and as such there will be less information in the control variate.
By increasing the number of proposals we would expect a reduction in variance pr iteration. Figure 9 shows the relative variance reduction pr iteration achieved from increasing m for our estimators $\hat{\mu}$ and $\tilde{\mu}$. As we saw indications of in Figure 6, the reduction in variance seems to level out as m increases and approach some limit.
We now come to our primary interest, which is the variance reduction pr time achieved from increasing the value of m. We would like to measure the variance reduction achieved for an equal run-time, since a run with $m = 16$ takes longer than a run with $m = 1$. To investigate this we multiplied our variance estimates with the relative time difference $\frac{t_m}{t_1}$. This gives us an estimate of what the variance would be for different values of m, if we only ran our simulations for a fixed time, in this case as long as it takes the run with $m = 1$ to finish. In Figure 10 we have plotted the relative variance reduction that we now get from increasing m. A negative value means that the variance has increased, and as such we can see that the variance of our estimate does not necessarily improve with increased m. This means that although a larger value of m will give a lower vari-

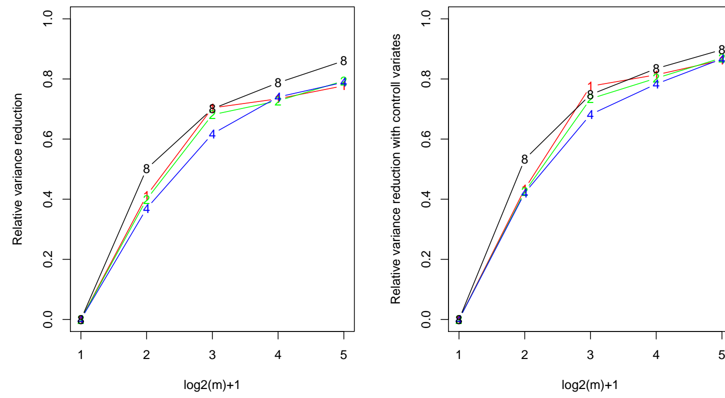

Figure 9: (Gaussian example:) Relative variance reduction pr iteration from increasing m for $\hat{\mu}$ (left) and $\tilde{\mu}$ (right).

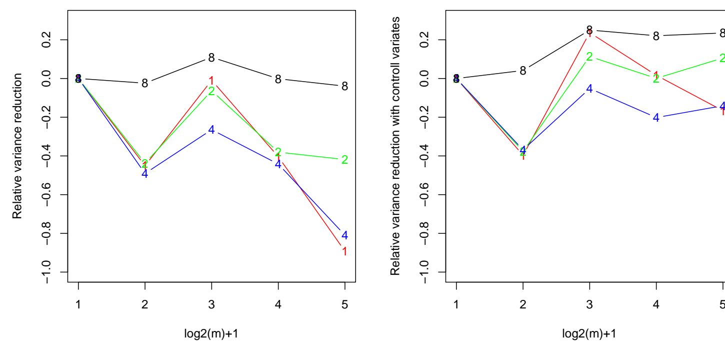

Figure 10: (Gaussian example:) Relative variance reduction pr time from increasing m for $\hat{\mu}$ (left) and $\tilde{\mu}$ (right).

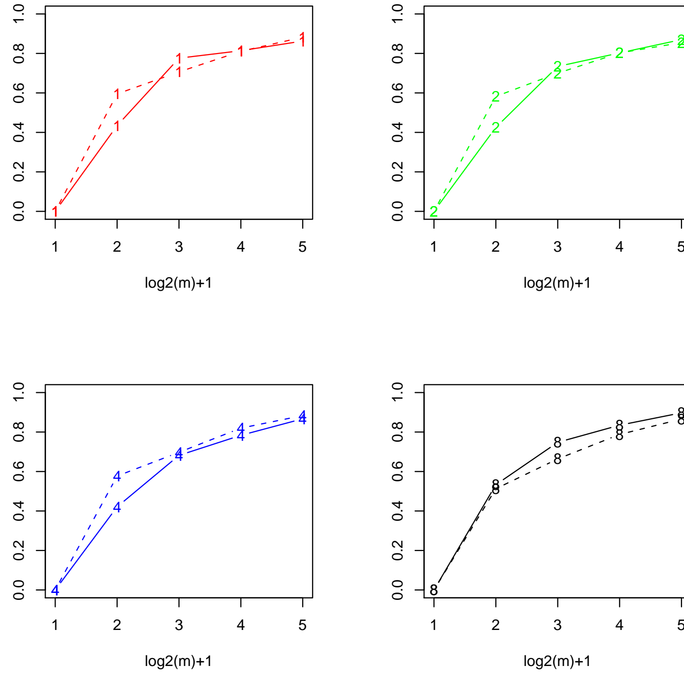

Figure 11: (Gaussian example:) Relative variance reduction from increasing m for $\tilde{\mu}$ (solid line) and relative variance reduction from increasing the run-time (or number of iterations) for $m = 1$ to the run time for $m = 2, 4, 8, 16$ (dashed line).

ance if run for the same number of iterations, the equivalent is not true if we run the algorithms for the same amount of time. From the plots we can see that increasing m is more valuable if we use control variates in our estimator. This is not unexpected as we saw from Figure 8 that control variates become more valuable as m increases. So by increasing m we get a further increased gain from the control variates, as well as the increased acceptance rate. Common for all the different values of σ^2 is a small peak for $m = 4$. This indicates that $m = 4$ gives us a good balance between a high acceptance rate and fast run time, which corresponds well with Figures 5 and 9. Again $\sigma^2 = 8$ behaves a bit different from the others due to it's lower acceptance rates. As we saw in Figure 6 we gain a lot from increasing m when we have a low acceptance rate. In general, however, increasing m is not very beneficial when sampling with a random walk proposal from a Gaussian distribution. Figure 11 illustrates this quite well, where we have simply estimated the relative variance reduction of our estimator for $m = 1$ if we let the algorithm run as long as it takes to run $m = 2, 4, 8, 16$. The figure shows us that we could achieve an almost equal or in some cases better variance reduction simply by letting $m = 1$ run as long as it takes to run $m = 2, 4, 8, 16$. This corresponds to Figure 10 where we saw that there was little to gain in variance reduction pr time from increasing

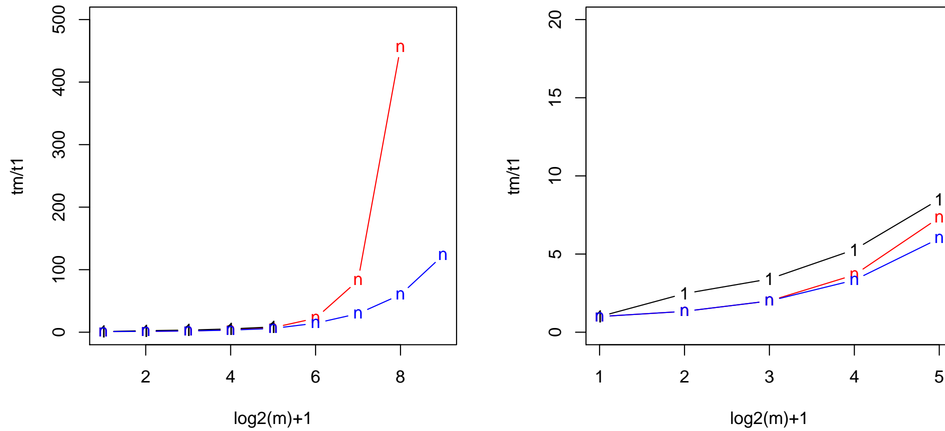

Figure 12: (Gaussian example:) Relative run-times for $m = 1, 2, 4, 8, 16$ on the original cluster (black line), for $m = 1, 2, 4, 8, 16, 32, 64, 128$ on Njord (red line) and for $m = 1, 2, 4, 8, 16, 32, 64, 128, 256$ on Njord without Peskunizing the transition matrix (blue line).

m further.
One final result should be mentioned for the Gaussian example. Shortly before this report was due we were given the chance to run our Gaussian example on NTNU's supercomputer Njord. Our previous runs where performed on a cluster where the number of processors available was restricted to 24. With Njord we were able to perform runs with up to almost 400 processors. This allowed us to explore the relative run time of our algorithm as m increased even further. The results are shown in Figure 12. In the plot the black line represents the runs on the student workstations, the red line represents the runs made on Njord and the blue line represents the runs made on Njord excluding the Peskunization process on the transition matrix $\mathbf{P}(\mathbf{y})$. As we can see from the right plot the relative run-times are slightly improved on Njord, due to a reduced overhead. From the left plot however, we can see that the relative run-times explode once m becomes sufficiently large. We suspected that this was not because of the overhead but due to the Peskunizing process that is applied to the transition matrix $\mathbf{P}$ (see Section 4.2). To test this we ran the algorithm on Njord again, but this time simply applied the transition matrix you get from equation (26). The results are shown in the blue line in the left plot, and as we can see this greatly reduces the run-time. However as we also can see from the plot, even without the Peskunization process, when m becomes sufficiently large the overhead becomes more dominating.

7.2 Mode jumping example

In this section we consider an example selected from Tjelmeland & Hegstad (2001), where a mixture model is used for a data-set concerning fetal deaths in litters of mice. The model is a mixture of a beta-binomial and binomial distributions,

$$p(\lambda|\eta) = \gamma\Big[\binom{\eta}{\lambda}\prod_{r=0}^{\lambda-1}\frac{\mu + r\theta}{1+r\theta}\prod_{r=0}^{\eta-\lambda-1}\frac{1-\mu-r\theta}{1+r\theta}\Big] + (1-\gamma)\Big[\binom{\eta}{\lambda}\nu^{\lambda}(1-\nu)^{\eta-\lambda}\Big], \quad (54)$$

where λ is the number of deaths and η the number of implants or fetuses. The model parameters are $\gamma \in [0,1]$, $\mu \in [0,1]$, $\theta \geq 0$ and $\nu \in [0,1]$, to which independent vague priors are assigned. The distribution of interest is the posterior distribution for the parameters given the data. Before the sampler is constructed the model is reparametrized to ensure a posterior density which is positive on all $\Re^4$. For the parameters we adopt the transformations,

$$\gamma = \frac{exp(\tilde{\gamma})}{1+exp(\tilde{\gamma})},\ \mu = \frac{exp(\tilde{\mu})}{1+exp(\tilde{\mu})},\ \theta = ln(\tilde{\theta}) \text{ and } \nu = \frac{exp(\tilde{\nu})}{1+exp(\tilde{\nu})}, \quad (55)$$

where $\tilde{\gamma}, \tilde{\mu}, \tilde{\theta}$ and $\tilde{\nu}$ are the transformed parameters. The distribution of the transformed parameters is distributed in two separated modes. For our example we will focus on $\tilde{\nu}$, chosen because this parameter separates the two modes clearly. We want to determine the probability mass in the smaller mode, and design our function $f(\cdot)$ accordingly,

$$f(\tilde{\nu}) = \begin{cases} 1 & \text{if} \quad \tilde{\nu} \geq -1.5, \\ 0 & \text{if} \quad \tilde{\nu} < -1.5. \end{cases} \quad (56)$$

We sample from the distribution using a mode jumping algorithm with multiple proposals as presented in Section 5 and calculate control variates as discussed in Section 6. We alternate between one mode jumping step and 100 local random walk steps, calling the total one iteration. In the design of the control variates we only used the mode jumping steps. Since we are attemting to estimate the probability mass in the smaller mode of ν, the local steps are of little interest to us when estimating μ. The algorithm was run for 20000 iterations and as in the Gaussian example, we implemented the algorithm in parallel according to Section 4.3. The vectors $\phi_{k,j}$ where sampled from a Normal distribution centered in the origin and a covariance matrix with only diagonal elements, all equal to 20^2.

Again we start with a general look at the results and look at convergence. In Figure 13 we have plotted the last 1000 global steps for the four parameters from the run with $m = 16$. Figure 14 shows a cumulative mean plot for $f(\tilde{\nu})$ as well as the estimated distribution of the smaller mode in ν. As we can see from the two figures, the algorithm appears to have converged and we get frequent jumps between the two modes. The probability mass in the smaller mode approaches a value of roughly 0.015. The acceptance rate and relative run-time ($\frac{t_m}{t_1}$) of the separate runs as a function of m are shown in Figure 15. In contrast to the Gaussian example we can see that there is a large time gain to be found

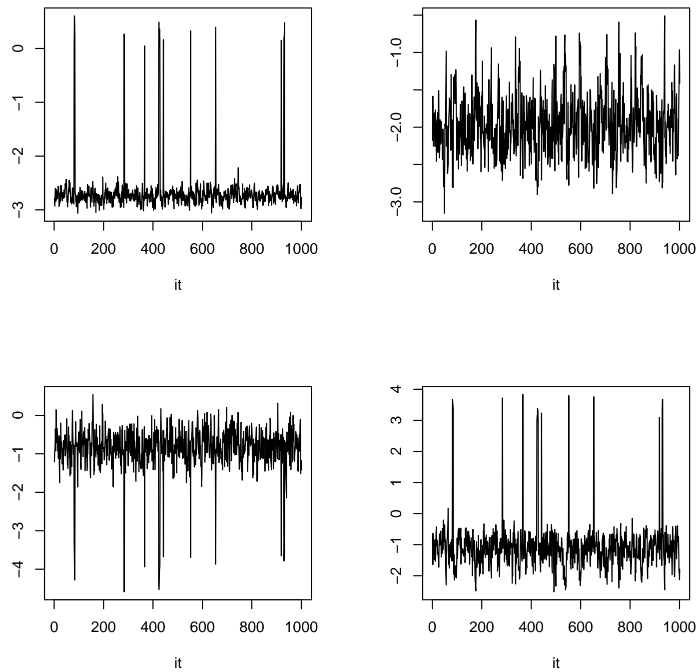

Figure 13: (Mode jumping example:) Trace plots for $\tilde{\nu}$ (upper left), $\tilde{\mu}$ (upper right), $\tilde{\theta}$ (lower left) and $\tilde{\gamma}$ (lower right) for the last 1000 generated values.

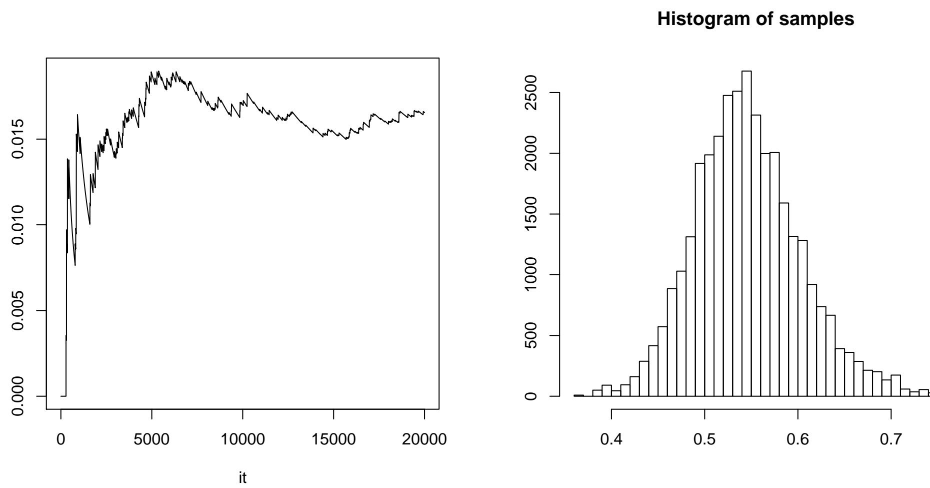

Figure 14: (Mode jumping example:) Convergence diagnostics, Cumulative mean plot of $f(\tilde{\nu})$ (left) and histogram of the samples in the smaller mode in ν (right).

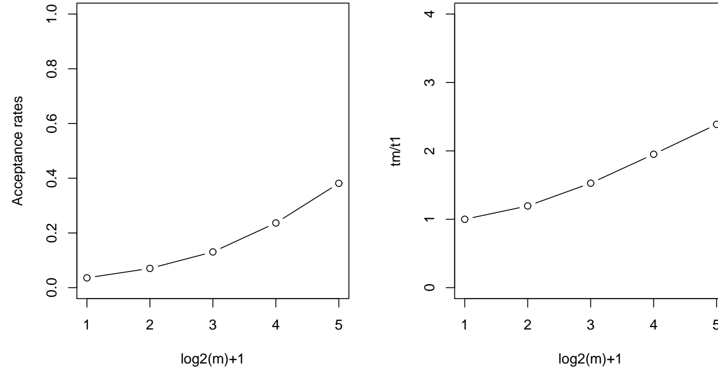

Figure 15: (Mode jumping example:) Acceptance rate (left) and relative run-time (right) for $m = 1, 2, 4, 8, 16$.

m	$\mathrm{Var}(\hat{\mu})$	$\mathrm{Var}(\tilde{\mu})$
1	0.00569	0.00224
2	0.00363	0.00192
4	0.00139	0.00089
8	0.00075	0.00041
16	0.00049	0.00021

Table 2: (Mode jumping example:) Estimated variance of $\hat{\mu}$ and $\tilde{\mu}$ for $m = 1, 2, 4, 8, 16$.

in implementing the algorithm in parallel. Since each new proposal is CPU intensive to produce, the overhead becomes negligible and we see only a small increase in time as we increase m. At the same time we can see that the acceptance rate is roughly proportional to m (remember that we plot m on a log2 scale). Especially since the acceptance rate is so low for $m = 1$, we would expect this to be very beneficial in reducing the variance of our estimators. This can be seen in Figure 16, where we have plotted the variance of $\hat{\mu}$ (dashed red line) and $\tilde{\mu}$ (solid line) for different values of m. The corresponding values are presented in Table 2. In Figure 17 we have plotted the two confidence intervals we get for μ. The dashed red line represents the confidence intervals we get without using control variates while the solid line represents the confidence interval achieved with control variates. We see that the intervals are reduced as m increases as one would expect, and the use of control variates reduces them even further. It may seem strange that the confidence intervals using control variates allow for negative probability masses, but this is simply due to the fact that the control variates can have negative values.
We want to investigate how the variance reduction achieved from the control variates changes for increasing values of m. To investigate this we have plotted the relative variance reduction achieved from using $\tilde{\mu}$ instead of $\hat{\mu}$ for different values of m, see Figure 18. We immediately notice that this differs from the results we got in the Gaussian

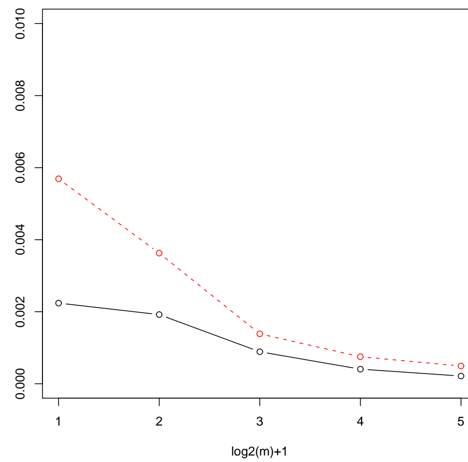

Figure 16: (Mode jumping example:) Estimated variance of $\hat{\mu}$ (dashed red line) and $\tilde{\mu}$ (solid line) as a function of m.

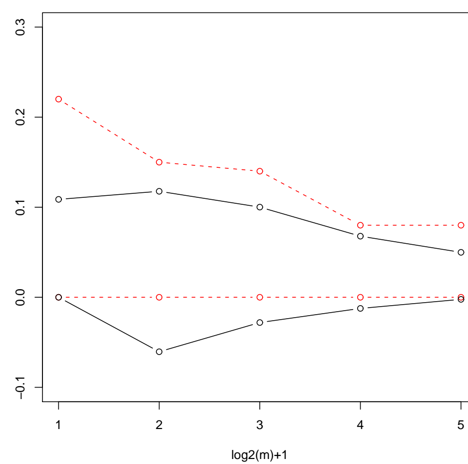

Figure 17: (Mode jumping example:) 95%Confidence intervals for μ, using $\hat{\mu}$ (dashed red line) and $\tilde{\mu}$ (solid line) for $m = 1, 2, 4, 8, 16$.

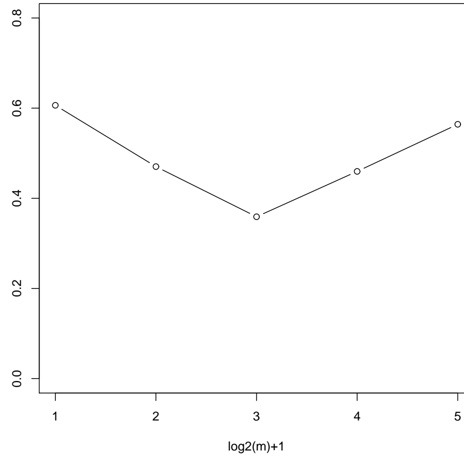

Figure 18: (Mode jumping example:) Relative variance reduction achieved from using $\tilde{\mu}$ instead of $\hat{\mu}$ for $m = 1, 2, 4, 8, 16$.

example where the relative variance reduction increased steadily with m (see Figure 8). The reason for this is probably the low acceptance rate in the mode jumping example. We can rewrite the relative variance reduction as $\frac{\text{Var}(\hat{\mu})-\text{Var}(\tilde{\mu})}{\text{Var}(\hat{\mu})} = 1 - \frac{\text{Var}(\tilde{\mu})}{\text{Var}(\hat{\mu})}$, so a decreasing relative variance reduction may mean that the variance of $\hat{\mu}$ decreases faster than the variance of $\tilde{\mu}$ as m increases. If we study Figure 16 we see that this is the case for small m, which again is due to the low acceptance rate. When the acceptance rate is so low our estimator will have a relatively large variance, doubling the acceptance rate will then greatly reduce the variance. By using control variates for $m = 1$ we have already reduced the variance a great deal, and as such there is less to gain. Once the acceptance rate increases beyond a certain point, due to m increasing, we approach the situation we had in the Gaussian example where a larger m means more proposals discarded and thus more information in each control variate. For small m the benefits from increasing the acceptance rate are larger than the benefits from more information in the control variate. But as m and the acceptance rate increase, the variance reduction from increasing the acceptance rate decreases. This is verified when we look at the second area of focus for our analysis, the relative reduction in variance pr iteration from increasing m.

In Figure 19 we have plotted the relative variance reduction pr iteration from increasing m for $\hat{\mu}$ (dashed red line) and $\tilde{\mu}$ (solid line). From the plot we see that the relative variance reduction is larger for $\hat{\mu}$ than for $\tilde{\mu}$ for small m while as m grows larger this evens out.

Finally, as in the Gaussian example, we would like to get an idea of the relative variance reduction pr time from increasing m. In the Gaussian example we discovered that there was little to be gained from increasing m since the overhead in the algorithm was so large. In this case we saw from Figure 15 that the overhead was much smaller due to

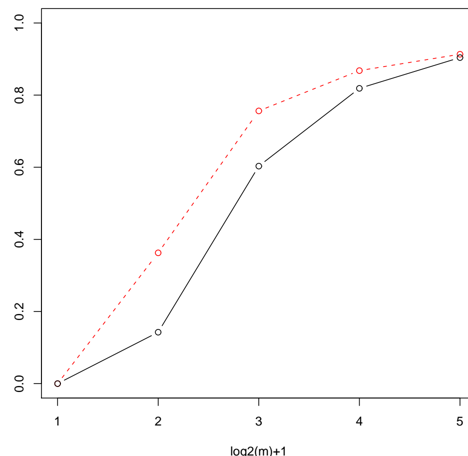

Figure 19: (Mode jumping example:) Relative variance reduction from increasing m for $\hat{\mu}$ (dashed red line) and $\tilde{\mu}$ (solid line).

each proposal being so CPU-intensive to calculate, and as such we might hope that there is something to be gained from increasing m. This turns out to be the case, as we can see in Figures 20 and 21. In Figure 20 we see the relative variance reduction from increasing m for $\hat{\mu}$ and $\tilde{\mu}$ when our variance estimates have first been multiplied by a factor $\frac{t_m}{t_1}$. As we can see we still get a significant reduction in variance, the exception being the run for $m = 2$ with control variates. In particular there seems to be a significant gain from increasing m from 2 to 4. This is further illustrated in Figure 21 where we have compared the relative variance reduction we would have gotten from running the $m = 1$ run longer, to the variance reduction we get in $\tilde{\mu}$ by increasing m. For $m = 2$ we see that we get roughly the same variance reduction, but when we increase m to 4 there is a significant gain compared to simply running the original algorithm longer.

8 Closing remarks

In this report we have considered the variance reduction achieved through use of control variates and parallel implementation of multi-proposal MCMC algorithms. Tjelmeland (2004) showed that multi-proposal MCMC gives little variance reduction per time implemented sequentially, however, by implementing the algorithm in parallel we expected to greatly increase the variance reduction per time. To investigate the benefits of these two techniques we have implemented them in two examples. First a toy Gaussian example where we sampled from a two dimensional normal distribution using a multi-proposal random walk algorithm. Second we designed a multi-proposal mode jumping algorithm inspired from the standard mode jumping algorithm in Tjelmeland & Hegstad (2001) and applied it to an example originally gathered from Brooks et al. (1997). To a certain

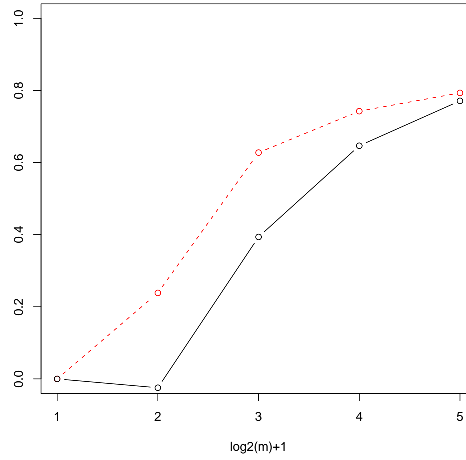

Figure 20: (Mode jumping example:) Relative variance reduction pr time from increasing m for $\hat{\mu}$ (dashed red line) and $\tilde{\mu}$ (solid line).

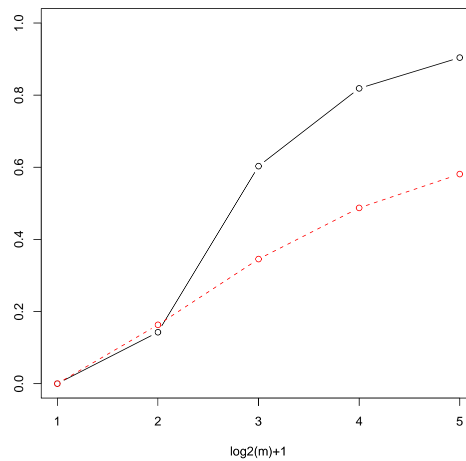

Figure 21: Relative variance reduction from increasing m for $\tilde{\mu}$ (solid line) and relative variance reduction from increasing the run time (or number of iterations) for $m = 1$ to the run time for $m = 2, 4, 8, 16$ (dashed red line).

degree our examples were chosen to demonstrate two extremes within MCMC algorithms. In the Gaussian example each proposal requires very little work, while in the mode jumping example each proposal requires several numerical optimizations, and is extremely CPU intensive. With a parallelization scheme where each processor proposes its own proposal, we expected the mode jumping example to be well suited for parallelization, while the Gaussian example would mark the other end of the scale. As we can see from the results this turned out to be the case, in the Gaussian example the overhead is much more prominent than in the mode jumping example.

The control variate scheme presented in this paper proves itself to be quite beneficial in both examples. Calculating the control variates in the sampling algorithm is a simple procedure with a computation time of order N, and calculating the value of the estimator $\tilde{\mu}$ after the sampler has completed requires little time. Thus there is very little cost associated with the use of control variates. As we saw both in the Gaussian example and the mode jumping example we achieved lower variance through the use of our control variate. Except for small acceptance rates, where the gain from the increase in m became dominant, the relative variance reduction achieved through control variates also seemed to increase as m increased. Intuitively this is not surprising as an increase in m means an increase in the number of rejected states pr iteration, and thus more information stored in the control variate. If the acceptance rate is low however, there are very few good proposals, and the rejected proposals stored in the control variate have little value compared to an increase in the acceptance rate. It would be interesting to examine what happens to the relative variance reduction achieved from control variates as m becomes larger than 16.

We saw that an increased number of proposals pr iteration increases the acceptance rate and as such reduces the variance of our estimator. However, increasing the number of proposals also of course increases the run-time. If the algorithm is well suited for parallelization, for example if the proposals are CPU intensive to compute, then the run-time increase can be made small by implementing the algorithm in parallel on several processors. In this case, as we saw in the second example, we can achieve a considerable variance reduction pr time. In the first example however, the algorithm was not that well suited for parallelization, and there was little gain to be had in increasing m. When m becomes sufficiently large the sequential code and overhead will grow rapidly. Because of this there is probably an ideal number of proposals where the relative variance reduction pr time is largest. We were given the opportunity to run our Gaussian example on the supercomputer Njord for large values of m, which gave us an impression of how the sampler behaved for large m values. As the acceptance rate approached 1 most of the gain from increasing m comes from increased information in the control variates. This combined with an increase in overhead and a time increase in the sequential code means that a smaller value for m is preferable. It would be interesting to run the mode jumping example for similar large values of m to see if some of the same behavior is found there. The mode jumping example has a much lower acceptance rate and is much better suited for parallelization, so we would expect it to be beneficial to increase m even further. However, Figure 20 might indicate that the relative variance reduction per

time is approaching a limit. It would also be interesting to test parallel multi-proposal MCMC algorithms on other examples. As mentioned earlier our two examples where chosen to demonstrate the two end of the scale, the technique should be tested for other examples as well.

References

Brooks, S. P., Morgan, B. J., Rideout, M. S. & Pack, S. E. (1997), 'Finite mixture models for proportions', *Biometrics* **53**, 1097–1115.

Casella, G. & Robert, C. P. (1996), 'Rao-Blackwellisation of sampling schemes', *Biometrica* **83**, 81–94.

Craiu, R. V. & Lemieux, C. (2005), 'Acceleration of the multiple-try Metropolis using antithetic and stratified sampling'.

Hastings, W. K. (1970), 'Monte Carlo sampling methods using Markov chains and their applications', *Biometrika* **57**, 97–109.

Liu, J. S. (2001), *Monte Carlo Strategies in Scientific Computing*, Springer, Berlin.

Liu, J. S., Liang, F. & Wong, W. H. (2000), 'The multiple-try method and local optimization in Metropolis sampling', *Journal of the American Statistical Association* **95**, 121–134.

Peskun, P. H. (1973), 'Optimum Monte-Carlo sampling using Markov chains.', *Biometrica* **60**, 607–612.

Roberts, C. P. & Casella, G. (1999), *Monte Carlo statistical methods*, Springer, Berlin.

Stormark, K. (2006), Multiple proposal strategies for Markov Chain Monte Carlo, Master's thesis, Department of Mathematical Sciences, Norwegian University of Science and Technology.

Tjelmeland, H. (2004), 'Using all Metropolis-Hastings proposals to estimate mean values', *Statistics No 4/2004, Department of Mathematical Sciences, Norwegian University of Science and Technology* .

Tjelmeland, H. & Hammer, H. (2005), 'Control variates for the Metropolis-Hastings algorithm', *Statistics No 8/2005, Department of Mathematical Sciences, Norwegian University of Science and Technology* .

Tjelmeland, H. & Hegstad, B. K. (2001), 'Mode jumping Proposals in MCMC', *Scandinavian journal of statistics* **28**, 205–223.