Fast generic MCMC for targets with expensive likelihoods

C. Sherlock (Lancaster), A. Golightly (Ncl) & D. Henderson (Ncl); this talk by: Jere Koskela (Warwick)

Monte Carlo Techniques, Paris, July 2016
Motivation

Metropolis-Hastings (MH) algorithms create a realisation: $\theta^{(1)}, \theta^{(2)}, \ldots$ from a Markov chain with stationary density $\pi(\theta)$. For each $\theta^{(i)}$ we evaluate $\pi(\theta^{(i)})$ - and then discard it.

Pseudo-marginal MH algorithms create a realisation of $\hat{\pi}(\theta^{(i)}, u^{(i)})$ - and then discard it.

In many applications evaluating $\pi(\theta)$ or $\hat{\pi}(\theta, u)$ is computationally expensive.

We would like to reuse these values to create a more efficient MH algorithm that still targets the correct stationary distribution.
Motivation

Metropolis-Hastings (MH) algorithms create a realisation: $\theta^{(1)}, \theta^{(2)}, \ldots$ from a Markov chain with stationary density $\pi(\theta)$. For each $\theta^{(i)}$ we evaluate $\pi(\theta^{(i)})$ - and then discard it.

Pseudo-marginal MH algorithms create a realisation of $\hat{\pi}(\theta^{(i)}, u^{(i)})$ - and then discard it.

In many applications evaluating $\pi(\theta)$ or $\hat{\pi}(\theta, u)$ is computationally expensive.

We would like to reuse these values to create a more efficient MH algorithm that still targets the correct stationary distribution.

We will focus on the [pseudo-marginal] random walk Metropolis (RWM) with a Gaussian proposal:

$$\theta' | \theta \sim N(\theta, \lambda^2 V).$$
This talk

- Creating an approximation to $\pi(\theta')$: k-NN.
- Using an approximation: Delayed acceptance [PsM]MH.
- Storing the values: KD-trees.
- Choice of $P$(fixed kernel).
- Simulation study.
Creating an approximation

At iteration $n$ of the MH we have $\pi(\theta^{(1)}), \pi(\theta^{(2)}), \ldots, \pi(\theta^{(n)})$, and we would like to create $\hat{\pi}_a(\theta') \approx \pi(\theta')$. 
Creating an approximation

At iteration $n$ of the MH we have $\pi(\theta^{(1)}), \pi(\theta^{(2)}), \ldots, \pi(\theta^{(n)})$, and we would like to create $\hat{\pi}_a(\theta') \approx \pi(\theta')$.

**Gaussian process** to $\log \pi$?
Creating an approximation

At iteration $n$ of the MH we have $\pi(\theta^{(1)}), \pi(\theta^{(2)}), \ldots, \pi(\theta^{(n)})$, and we would like to create $\hat{\pi}_a(\theta') \approx \pi(\theta')$.

**Gaussian process** to $\log \pi$?:
[problems with cost of fitting and evaluating as $n \uparrow$]
Creating an approximation

At iteration $n$ of the MH we have $\pi(\theta^{(1)})$, $\pi(\theta^{(2)})$, $\ldots$, $\pi(\theta^{(n)})$, and we would like to create $\hat{\pi}_a(\theta') \approx \pi(\theta')$.

**Gaussian process** to $\log \pi$?:
[problems with cost of fitting and evaluating as $n \uparrow$]

**Weighted average** of k-nearest neighbour $\pi$ values:
(i) Fitting cost: 0 (actually $O(n_0)$).
Creating an approximation

At iteration $n$ of the MH we have $\pi(\theta^{(1)}), \pi(\theta^{(2)}), \ldots, \pi(\theta^{(n)})$, and we would like to create $\hat{\pi}_a(\theta') \approx \pi(\theta')$.

**Gaussian process** to $\log \pi$?:
[problems with cost of fitting and evaluating as $n \uparrow$]

**Weighted average** of $k$-nearest neighbour $\pi$ values:

(i) Fitting cost: 0 (actually $O(n_0)$).
(ii) Per-iteration cost: $O(n)$. 
Creating an approximation

At iteration \( n \) of the MH we have \( \pi(\theta^{(1)}), \pi(\theta^{(2)}), \ldots, \pi(\theta^{(n)}) \), and we would like to create \( \hat{\pi}_a(\theta') \approx \pi(\theta') \).

**Gaussian process** to \( \log \pi \):  
[problems with cost of fitting and evaluating as \( n \uparrow \)]

**Weighted average** of **k-nearest neighbour** \( \pi \) values:  
(i) Fitting cost: 0 (actually \( \mathcal{O}(n_0) \)).  
(ii) Per-iteration cost: \( \mathcal{O}(n) \).  
(iii) Accuracy \( \uparrow \) with \( n \).
Delayed acceptance MH (1)

(Christen and Fox, 2005). Suppose we have a computationally-cheap approximation to the posterior: \( \hat{\pi}_a(\theta) \).

\[
\alpha_{da}(\theta; \theta') := \alpha_1(\theta; \theta') \alpha_2(\theta; \theta')
\]

where 

\[
\alpha_1 := 1 \wedge \frac{\hat{\pi}_a(\theta')q(\theta'|\theta)}{\hat{\pi}_a(\theta)q(\theta'|\theta)}
\]

\[
\alpha_2 := 1 \wedge \frac{\pi(\theta')}{\hat{\pi}_a(\theta')}
\]

Detailed balance (with respect to \( \pi(\theta) \)) is still preserved with \( \alpha_{da} \) because 

\[
\pi(\theta)q(\theta'|\theta)\alpha_{da}(\theta; \theta') = \hat{\pi}_a(\theta)q(\theta'|\theta)\alpha_1 \times \frac{\pi(\theta)}{\hat{\pi}_a(\theta)}\alpha_2.
\]

But this algorithm mixes worse than the equivalent MH algorithm (Peskun, 1973; Tierney, 1998).
Delayed acceptance MH (1)

(Christen and Fox, 2005). Suppose we have a computationally-cheap approximation to the posterior: $\hat{\pi}_a(\theta)$.

Define $\alpha_{da}(\theta; \theta') := \alpha_1(\theta; \theta') \alpha_2(\theta; \theta')$, where

$$\alpha_1 := 1 \wedge \frac{\hat{\pi}_a(\theta')q(\theta|\theta')}{\hat{\pi}_a(\theta)q(\theta'|\theta)}$$

and

$$\alpha_2 := 1 \wedge \frac{\pi(\theta')/\hat{\pi}_a(\theta')}{\pi(\theta)/\hat{\pi}_a(\theta)}.$$

Detailed balance (with respect $\pi(\theta)$) is still preserved with $\alpha_{da}$ because
Delayed acceptance MH (1)

(Christen and Fox, 2005). Suppose we have a computationally-cheap approximation to the posterior: \( \hat{\pi}_a(\theta) \).

Define \( \alpha_{da}(\theta; \theta') := \alpha_1(\theta; \theta') \alpha_2(\theta; \theta') \), where

\[
\alpha_1 := 1 \wedge \frac{\hat{\pi}_a(\theta') q(\theta|\theta')}{\hat{\pi}_a(\theta) q(\theta'|\theta)} \quad \text{and} \quad \alpha_2 := 1 \wedge \frac{\pi(\theta')/\hat{\pi}_a(\theta')}{\pi(\theta)/\hat{\pi}_a(\theta)}.
\]

Detailed balance (with respect \( \pi(\theta) \)) is still preserved with \( \alpha_{da} \) because

\[
\pi(\theta) q(\theta'|\theta) \alpha_{da}(\theta; \theta')
\]
Delayed acceptance MH (1)

(Christen and Fox, 2005). Suppose we have a computationally-cheap approximation to the posterior: \( \hat{\pi}_a(\theta) \).

Define \( \alpha_{da}(\theta; \theta') := \alpha_1(\theta; \theta') \alpha_2(\theta; \theta') \), where

\[
\alpha_1 := 1 \wedge \frac{\hat{\pi}_a(\theta')q(\theta|\theta')}{\hat{\pi}_a(\theta)q(\theta'|\theta)} \quad \text{and} \quad \alpha_2 := 1 \wedge \frac{\pi(\theta')/\hat{\pi}_a(\theta')}{\pi(\theta)/\hat{\pi}_a(\theta)}.
\]

Detailed balance (with respect \( \pi(\theta) \)) is still preserved with \( \alpha_{da} \) because

\[
\pi(\theta)q(\theta'|\theta) \alpha_{da}(\theta; \theta') = \hat{\pi}_a(\theta)q(\theta'|\theta) \alpha_1 \times \pi(\theta) \hat{\pi}_a(\theta) \alpha_2.
\]
Delayed acceptance MH (1)

(Christen and Fox, 2005). Suppose we have a computationally-cheap approximation to the posterior: \( \hat{\pi}_a(\theta) \).

Define \( \alpha_{da}(\theta; \theta') := \alpha_1(\theta; \theta') \alpha_2(\theta; \theta') \), where

\[
\alpha_1 := 1 \wedge \frac{\hat{\pi}_a(\theta')q(\theta'|\theta)}{\hat{\pi}_a(\theta)q(\theta'|\theta)} \quad \text{and} \quad \alpha_2 := 1 \wedge \frac{\pi(\theta')/\hat{\pi}_a(\theta')}{\pi(\theta)/\hat{\pi}_a(\theta)}.
\]

Detailed balance (with respect \( \pi(\theta) \)) is still preserved with \( \alpha_{da} \) because

\[
\pi(\theta) q(\theta'|\theta) \alpha_{da}(\theta; \theta')
= \hat{\pi}_a(\theta) q(\theta'|\theta) \alpha_1 \times \frac{\pi(\theta)}{\hat{\pi}_a(\theta)} \alpha_2.
\]

But this algorithm mixes worse than the equivalent MH algorithm (Peskun, 1973; Tierney, 1998).
Delayed-acceptance [PsM]MH (2)

Using $\alpha_{da} = \alpha_1(\theta; \theta')\alpha_2(\theta; \theta')$ mixes worse but CPU time/iteration can be much reduced.
**Delayed-acceptance [PsM]MH (2)**

Using $\alpha_{da} = \alpha_1(\theta; \theta')\alpha_2(\theta; \theta')$ mixes worse but CPU time/iteration can be much reduced.

Accept $\Leftrightarrow$ accept at Stage 1 (w.p. $\alpha_1$) and accept at Stage 2 (w.p. $\alpha_2$).
Delayed-acceptance [PsM]MH (2)

Using $\alpha_{da} = \alpha_1(\theta; \theta')\alpha_2(\theta; \theta')$ mixes worse but CPU time/iteration can be much reduced.

Accept $\Leftrightarrow$ accept at Stage 1 (w.p. $\alpha_1$) and accept at Stage 2 (w.p. $\alpha_2$).

$\alpha_1$ is quick to calculate.
Delayed-acceptance [PsM]MH (2)

Using $\alpha_{da} = \alpha_1(\theta; \theta') \alpha_2(\theta; \theta')$ mixes worse but CPU time/iteration can be much reduced.

Accept $\iff$ accept at Stage 1 (w.p. $\alpha_1$) and accept at Stage 2 (w.p. $\alpha_2$).

$\alpha_1$ is quick to calculate.

There is no need to calculate $\alpha_2$ if we reject at Stage One.
Delayed-acceptance [PsM]MH (2)

Using $\alpha_{da} = \alpha_1(\theta; \theta')\alpha_2(\theta; \theta')$ mixes worse but CPU time/iteration can be much reduced.

Accept $\iff$ accept at Stage 1 (w.p. $\alpha_1$) and accept at Stage 2 (w.p. $\alpha_2$).

$\alpha_1$ is quick to calculate.

There is no need to calculate $\alpha_2$ if we reject at Stage One.

If $\hat{\pi}_a$ is accurate then $\alpha_2 \approx 1$. 

Delayed-acceptance [PsM]MH (2)

Using \( \alpha_{da} = \alpha_1(\theta; \theta')\alpha_2(\theta; \theta') \) mixes worse but CPU time/iteration can be much reduced.

Accept \( \iff \) accept at Stage 1 (w.p. \( \alpha_1 \)) and accept at Stage 2 (w.p. \( \alpha_2 \)).

\( \alpha_1 \) is quick to calculate.

There is no need to calculate \( \alpha_2 \) if we reject at Stage One.

If \( \hat{\pi}_a \) is accurate then \( \alpha_2 \approx 1 \).

If \( \hat{\pi}_a \) is also cheap then (RWM) can use large jump proposals [EXPLAIN].
Delayed-acceptance [PsM]MH (2)

Using $\alpha_{da} = \alpha_1(\theta; \theta')\alpha_2(\theta; \theta')$ mixes worse but CPU time/iteration can be much reduced.

Accept $\Leftrightarrow$ accept at Stage 1 (w.p. $\alpha_1$) and accept at Stage 2 (w.p. $\alpha_2$).

$\alpha_1$ is quick to calculate.

There is no need to calculate $\alpha_2$ if we reject at Stage One.

If $\hat{\pi}_a$ is accurate then $\alpha_2 \approx 1$.

If $\hat{\pi}_a$ is also cheap then (RWM) can use large jump proposals [EXPLAIN].

Delayed-acceptance PMMH:

$$\alpha_2 := 1 \land \frac{\hat{\pi}(\theta', u')/\hat{\pi}_a(\theta')}{\hat{\pi}(\theta, u)/\hat{\pi}_a(\theta)}.$$
We: use an inverse-distance-weighted average of the $\pi$ values from the $k$ nearest neighbours.
Cheap and accurate approximation?

We: use an inverse-distance-weighted average of the $\pi$ values from the $k$ nearest neighbours.

But the cost is still $O(n)/\text{iter}$. 
**$k$-nn and the binary tree**

Imagine a table with $n$ values.

\[
\begin{array}{cccc|c}
\theta_1^{(1)} & \theta_2^{(1)} & \ldots & \theta_d^{(1)} & \pi(\theta^{(1)}) \\
\theta_1^{(2)} & \theta_2^{(2)} & \ldots & \theta_d^{(2)} & \pi(\theta^{(2)}) \\
\ldots & \ldots & \ldots & \ldots & \ldots \\
\theta_1^{(n)} & \theta_2^{(n)} & \ldots & \theta_d^{(n)} & \pi(\theta^{(n)}) \\
\end{array}
\]

Look-up of $k$ nearest neighbours to some $\theta'$ is $O(n)$. 
$k$-nn and the binary tree

Imagine a table with $n$ values.

\[
\begin{array}{cccc}
\theta_1^{(1)} & \theta_2^{(1)} & \ldots & \theta_d^{(1)} \\
\theta_1^{(2)} & \theta_2^{(2)} & \ldots & \theta_d^{(2)} \\
\vdots & \vdots & \ddots & \vdots \\
\theta_1^{(n)} & \theta_2^{(n)} & \ldots & \theta_d^{(n)} \\
\end{array}
\pi(\theta^{(1)}) \\
\pi(\theta^{(2)}) \\
\vdots \\
\pi(\theta^{(n)})
\]

Look-up of $k$ nearest neighbours to some $\theta'$ is $O(n)$.

If $d = 1$ then could sort list or create a standard binary tree for $O(\log n)$ look up.

For $d > 1$ a solution is the KD-tree.
$k$-nn and the binary tree

Imagine a table with $n$ values.

\[
\begin{array}{cccc}
\theta_1^{(1)} & \theta_2^{(1)} & \cdots & \theta_d^{(1)} \\
\theta_1^{(2)} & \theta_2^{(2)} & \cdots & \theta_d^{(2)} \\
\vdots & \vdots & \ddots & \vdots \\
\theta_1^{(n)} & \theta_2^{(n)} & \cdots & \theta_d^{(n)}
\end{array}
\quad \pi(\theta^{(1)})
\quad \pi(\theta^{(2)})
\quad \pi(\theta^{(n)})
\]

Look-up of $k$ nearest neighbours to some $\theta'$ is $O(n)$.

If $d = 1$ then could sort list or create a standard binary tree

![Binary Tree Diagram]

for $O(\log n)$ look up. For $d > 1$ a solution is the KD-tree.
KD-tree (d=2)

\[ m\{S\} = \text{branch splitting according to } \theta_{\text{d-split}} \text{ on median of } S. \]

\[ [L] = \text{leaf node with a maximum of } 2b - 1 \text{ leaves.} \]
**KD-tree (d=2)**

\[
\begin{align*}
\text{d-split} \\
1 & \quad m_1 := m\{\theta_1\} \\
2 & \quad m_2_- := m\{\theta_2: \theta_1 < m_1\} \\
& \quad m_2_+ := m\{\theta_2: \theta_1 > m_1\}
\end{align*}
\]

\[m\{S\} = \text{branch splitting according to } \theta_{d\text{-split}} \text{ on median of } S.\]

\[[L] = \text{leaf node with a maximum of } 2b - 1 \text{ leaves.}\]
KD-tree (d=2)

\[ d \text{-split} \]

\[ m_1 := \{ \theta_1 \} \]
\[ m_2^- := \{ \theta_2 : \theta_1 < m_1 \} \]
\[ m_2^+ := \{ \theta_2 : \theta_1 > m_1 \} \]

\[ m_1^- := \{ \theta_1 : \theta_1 < m_1, \theta_2 < m_2^- \} \]
\[ m_1^+ := \{ \theta_1 : \theta_1 > m_1, \theta_2 < m_2^- \} \]

\[ m \{ S \} = \text{branch splitting according to } \theta_{d \text{-split}} \text{ on median of } S. \]

\[ [L] = \text{leaf node with a maximum of } 2b - 1 \text{ leaves.} \]
**KD-tree (d=2)**

$m\{S\} = \text{branch splitting according to } \theta_{d\text{-split}} \text{ on median of } S.$

$[L] = \text{leaf node with a maximum of } 2b - 1 \text{ leaves.}$
$$m\{S\} = \text{ branch splitting according to } \theta_{d\text{-split}} \text{ on median of } S.$$ 
$$[L] = \text{ leaf node with a maximum of } 2b - 1 \text{ leaves.}$$

Our KD-tree is useful if (roughly) $$n/(3b/2) > 2^d.$$
Adaptive $k$-nn using a KD-tree

Initial, training run of $n_0$ iterations. Build initial KD-tree.

Set-up is $O(n_0 \log n_0^2)$; updating is $O(\log n)$, evaluation is $O(\log n)$ and accuracy $\uparrow$ as the MCMC progresses.

provided the tree is balanced.

[Skip, for lack of time]
Adaptive $k$-nn using a KD-tree

Initial, training run of $n_0$ iterations. Build initial KD-tree.

Main run: ‘every time’ $\pi(\theta')$ is evaluated, add $(\theta', \pi(\theta'))$ to the KD-tree.
Adaptive $k$-nn using a KD-tree

Initial, training run of $n_0$ iterations. Build initial KD-tree.

Main run: ‘every time’ $\pi(\theta')$ is evaluated, add $(\theta', \pi(\theta'))$ to the KD-tree.

Set-up is $O(n_0(\log n_0)^2)$; updating is $O(\log n)$ evaluation is $O(\log n)$ and accuracy $\uparrow$ as the MCMC progresses.
Adaptive $k$-nn using a KD-tree

Initial, training run of $n_0$ iterations. Build initial KD-tree.

Main run: ‘every time’ $\pi(\theta')$ is evaluated, add $(\theta', \pi(\theta'))$ to the KD-tree.

Set-up is $O(n_0(\log n_0)^2)$; updating is $O(\log n)$ evaluation is $O(\log n)$ and accuracy ↑ as the MCMC progresses.

Provided the tree is balanced. [Skip, for lack of time]
Refinements

Training dataset ⇒ better distance metric. Transform $\theta'$ to approximately isotropic before creating tree, or adding new node.
Refinements

Training dataset ⇒ better distance metric. Transform $\theta'$ to approximately isotropic before creating tree, or adding new node.

Minimum distance $\epsilon$. If $\exists \theta$ s.t. $||\theta' - \theta|| < \epsilon$ then
(i) MH: ignore new value.
(ii) PMMH: combine $\hat{\pi}(y|\theta', u')$ with $\hat{\pi}(y|\theta, u)$ (running average).
Adaptive Algorithm

Components

- A fixed [pseudo-marginal] kernel $P([\theta, u]; [d\theta', du'])$.
- An adaptive [pseudo-marginal] DA kernel $P_\gamma([\theta, u]; [d\theta', du'])$.
Adaptive Algorithm

Components

- A fixed [pseudo-marginal] kernel $P([\theta, u]; [d\theta', du'])$.
- An adaptive [pseudo-marginal] DA kernel $P_\gamma([\theta, u]; [d\theta', du'])$.
- Both $P$ and $P_\gamma$ generate $\hat{\pi}(\theta', u)$ in the same way.
- A fixed probability $\beta \in (0, 1]$.
- A set of probabilities: $p_n \to 0$.

Algorithm At the start of iteration $n$, the chain is at $[\theta, u]$ and the DA kernel would be $P_\gamma$. 
Adaptive Algorithm

Components

- A fixed [pseudo-marginal] kernel $P([\theta, u]; [d\theta', du'])$.
- An adaptive [pseudo-marginal] DA kernel $P_\gamma([\theta, u]; [d\theta', du'])$.
- Both $P$ and $P_\gamma$ generate $\hat{\pi}(\theta', u)$ in the same way.
- A fixed probability $\beta \in (0, 1]$.
- A set of probabilities: $p_n \to 0$.

Algorithm At the start of iteration $n$, the chain is at $[\theta, u]$ and the DA kernel would be $P_\gamma$.

1. Sample $[\theta', u']$ from
   \[
   \begin{cases} 
   P & \text{w.p. } \beta \\
   P_\gamma & \text{w.p. } 1 - \beta.
   \end{cases}
   \]
Adaptive Algorithm

Components

- A fixed [pseudo-marginal] kernel $P([\theta, u]; [d\theta', du'])$.
- An adaptive [pseudo-marginal] DA kernel $P_\gamma([\theta, u]; [d\theta', du'])$.
- Both $P$ and $P_\gamma$ generate $\hat{\pi}(\theta', u)$ in the same way.
- A fixed probability $\beta \in (0, 1]$.
- A set of probabilities: $p_n \to 0$.

Algorithm

At the start of iteration $n$, the chain is at $[\theta, u]$ and the DA kernel would be $P_\gamma$.

1. Sample $[\theta', u']$ from $\begin{cases} P & \text{w.p. } \beta \\ P_\gamma & \text{w.p. } 1 - \beta. \end{cases}$

2. W.p. $p_n$ ‘choose a new $\gamma$’: update the kernel by including all relevant information since the kernel was last updated.
Adaptive Algorithm

Components

- A fixed [pseudo-marginal] kernel $P([\theta, u]; [d\theta', du'])$.
- An adaptive [pseudo-marginal] DA kernel $P_\gamma([\theta, u]; [d\theta', du'])$.
- Both $P$ and $P_\gamma$ generate $\hat{\pi}(\theta', u)$ in the same way.
- A fixed probability $\beta \in (0, 1]$.
- A set of probabilities: $p_n \to 0$.

Algorithm

At the start of iteration $n$, the chain is at $[\theta, u]$ and the DA kernel would be $P_\gamma$.

1. Sample $[\theta', u']$ from

$$
\left\{ \begin{array}{ll}
P & \text{w.p. } \beta \\
P_\gamma & \text{w.p. } 1 - \beta.
\end{array} \right.
$$

2. W.p. $p_n$ ‘choose a new $\gamma$’: update the kernel by including all relevant information since the kernel was last updated.

Set: $p_n = 1/(1 + ci_n)$, where $i_n = \#$ expensive evaluations so far.
Ergodicity

Assumptions on the fixed kernel.

1. **Minorisation**: there is a density $\nu(\theta)$ and $\delta > 0$ such that $q(\theta' | \theta) \alpha(\theta; \theta') > \delta \nu(\theta')$, where $\alpha$ is the acceptance rate from the idealised version of the fixed MH algorithm.
Ergodicity

Assumptions on the fixed kernel.

1. **Minorisation**: there is a density $\nu(\theta)$ and $\delta > 0$ such that $q(\theta' | \theta)\alpha(\theta; \theta') > \delta \nu(\theta')$, where $\alpha$ is the acceptance rate from the idealised version of the fixed MH algorithm.

2. **Bounded weights**: the support of $W := \hat{\pi}(\theta, U)/\pi(\theta)$ is uniformly (in $\theta$) bounded above by some $\overline{w} < \infty$. 
**Ergodicity**

**Assumptions** on the fixed kernel.

1. **Minorisation**: there is a density $\nu(\theta)$ and $\delta > 0$ such that $q(\theta' | \theta) \alpha(\theta; \theta') > \delta \nu(\theta')$, where $\alpha$ is the acceptance rate from the idealised version of the fixed MH algorithm.

2. **Bounded weights**: the support of $W := \hat{\pi}(\theta, U) / \pi(\theta)$ is uniformly (in $\theta$) bounded above by some $\overline{w} < \infty$.

**Theorem** Subject to Assumptions 1 and 2, the adaptive pseudo-marginal algorithm is ergodic.
Ergodicity

Assumptions on the fixed kernel.

1. **Minorisation**: there is a density $\nu(\theta)$ and $\delta > 0$ such that $q(\theta' | \theta) \alpha(\theta; \theta') > \delta \nu(\theta')$, where $\alpha$ is the acceptance rate from the idealised version of the fixed MH algorithm.

2. **Bounded weights**: the support of $W := \hat{\pi}(\theta, U)/\pi(\theta)$ is uniformly (in $\theta$) bounded above by some $\overline{w} < \infty$.

Theorem Subject to Assumptions 1 and 2, the adaptive pseudo-marginal algorithm is ergodic.

NB. For DAMH, as opposed to DAPMMMH, only the minorisation assumption is required.
Choice of $\beta$

DARWM is more efficient when $\lambda > \hat{\lambda}_{RWM}$.
Choice of $\beta$

DARWM is more efficient when $\lambda > \hat{\lambda}_{RWM}$. The lower $\alpha_1$ does not matter.
Choice of $\beta$

DARWM is more efficient when $\lambda > \hat{\lambda}_{RWM}$.
The lower $\alpha_1$ does not matter.

But low $\alpha_1 \Rightarrow$ fewer expensive evaluations of $\pi$ [or of $\hat{\pi}$].
Choice of $\beta$

DARWM is more efficient when $\lambda > \hat{\lambda}_{RWM}$. The lower $\alpha_1$ does not matter.

But low $\alpha_1 \Rightarrow$ fewer expensive evaluations of $\pi$ [or of $\hat{\pi}$].

$$P(\text{expensive}) = \beta + (1 - \beta)\alpha_1.$$
Choice of $\beta$

DARWM is more efficient when $\lambda > \hat{\lambda}_{RWM}$.
The lower $\alpha_1$ does not matter.

But low $\alpha_1 \Rightarrow$ fewer expensive evaluations of $\pi$ [or of $\hat{\pi}$].

$$P(\text{expensive}) = \beta + (1 - \beta)\alpha_1.$$ 

If $\alpha_1 << 1$ most of the expensive evaluations come from the fixed kernel and much of the benefit from the adaptive kernel will be lost.
Choice of $\beta$

DARWM is more efficient when $\lambda > \hat{\lambda}_{RWM}$. The lower $\alpha_1$ does not matter.

But low $\alpha_1 \Rightarrow$ fewer expensive evaluations of $\pi$ [or of $\hat{\pi}$].

$$P(\text{expensive}) = \beta + (1 - \beta)\alpha_1.$$  

If $\alpha_1 << 1$ most of the expensive evaluations come from the fixed kernel and much of the benefit from the adaptive kernel will be lost.

Fixing $\beta \propto \alpha_1$ (obtained from the training run) avoids this,
Choice of $\beta$

DARWM is more efficient when $\lambda > \hat{\lambda}_{RWM}$. The lower $\alpha_1$ does not matter.

But low $\alpha_1 \Rightarrow$ fewer expensive evaluations of $\pi$ [or of $\hat{\pi}$].

$$P(\text{expensive}) = \beta + (1 - \beta)\alpha_1.$$  

If $\alpha_1 << 1$ most of the expensive evaluations come from the fixed kernel and much of the benefit from the adaptive kernel will be lost.

Fixing $\beta \propto \alpha_1$ (obtained from the training run) avoids this, but the guaranteed worst-case TVD from $\pi$ after $n$ iterations gets larger.
Choice of $\beta$

DARWM is more efficient when $\lambda > \hat{\lambda}_{RWM}$. The lower $\alpha_1$ does not matter.

But low $\alpha_1 \Rightarrow$ fewer expensive evaluations of $\pi$ [or of $\hat{\pi}$].

$$P(\text{expensive}) = \beta + (1 - \beta)\alpha_1.$$ 

If $\alpha_1 << 1$ most of the expensive evaluations come from the fixed kernel and much of the benefit from the adaptive kernel will be lost.

Fixing $\beta \propto \alpha_1$ (obtained from the training run) avoids this, but the guaranteed worst-case TVD from $\pi$ after $n$ iterations gets larger.

Consider a fixed computational budget $\approx$ fixed number of expensive evaluations. This preserves the provable worst-case TVD from $\pi$. 
Examples

Lotka-Volterra MJP daPMRWM with $d = 5$

LNA approximation to autoregulatory system daRWM with $d = 10$
Examples

Lotka-Volterra MJP daPMRWM with $d = 5$

LNA approximation to autoregulatory system daRWM with $d = 10$

RWM: $\theta' \sim N(\theta, \lambda^2 \hat{\Sigma})$ where $\hat{\Sigma}$, obtained from training run (also gives pre-map).
Examples

Lotka-Volterra MJP daPMRWM with $d = 5$

LNA approximation to autoregulatory system daRWM with $d = 10$

RWM: $\theta' \sim N(\theta, \lambda^2 \hat{\Sigma})$ where $\hat{\Sigma}$, obtained from training run (also gives pre-map).

Scaling, $\lambda$ [and number of particles $m$] chosen to be optimal for RWM.
Examples

Lotka-Volterra MJP daPMRWM with \( d = 5 \)

LNA approximation to autoregulatory system daRWM with \( d = 10 \)

RWM: \( \theta' \sim N(\theta, \lambda^2 \hat{\Sigma}) \) where \( \hat{\Sigma} \), obtained from training run (also gives pre-map).

Scaling, \( \lambda \) [and number of particles \( m \)] chosen to be optimal for RWM.

\( n_0 = 10000 \) (from training run), \( b = 10, c = 0.001 \).

\[
\text{Efficiency} = \frac{\min_{j=1\ldots d} \text{ESS}_j}{\text{CPU time}}
\]
Results: LV

\[ \text{RelESS} = \frac{\text{efficiency of DA[PM]RWM}}{\text{efficiency of optimal RWM}}. \]

\[ \hat{\lambda}_{RWM} \]

\( \xi = 1 \) corresponds to the DA using the scaling that is optimal for the standard RWM algorithm. i.e. DA scaling = \( \xi \times \hat{\lambda}_{RWM} \).
Results: Autoreg.

Solid=shorter dataset; dashed=longer dataset.
**LV: further experiments**

<table>
<thead>
<tr>
<th>$c$</th>
<th>Tree Size</th>
<th>$\hat{\alpha}_1$</th>
<th>$\hat{\alpha}_2$</th>
<th>Rel. mESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0001</td>
<td>41078</td>
<td>0.00772</td>
<td>0.339</td>
<td>7.28</td>
</tr>
<tr>
<td>0.001</td>
<td>40256</td>
<td>0.00915</td>
<td>0.276</td>
<td>6.80</td>
</tr>
<tr>
<td>0.01</td>
<td>43248</td>
<td>0.0121</td>
<td>0.204</td>
<td>4.67</td>
</tr>
<tr>
<td>$\infty$</td>
<td>10000</td>
<td>0.0175</td>
<td>0.136</td>
<td>3.46</td>
</tr>
</tbody>
</table>

Using a list rather than the KD-tree reduced the efficiency by a factor of $\approx 2$. 
LV: further experiments

<table>
<thead>
<tr>
<th>$c$</th>
<th>Tree Size</th>
<th>$\hat{\alpha}_1$</th>
<th>$\hat{\alpha}_2$</th>
<th>Rel. mESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0001</td>
<td>41078</td>
<td>0.00772</td>
<td>0.339</td>
<td>7.28</td>
</tr>
<tr>
<td>0.001</td>
<td>40256</td>
<td>0.00915</td>
<td>0.276</td>
<td>6.80</td>
</tr>
<tr>
<td>0.01</td>
<td>43248</td>
<td>0.0121</td>
<td>0.204</td>
<td>4.67</td>
</tr>
<tr>
<td>$\infty$</td>
<td>10000</td>
<td>0.0175</td>
<td>0.136</td>
<td>3.46</td>
</tr>
</tbody>
</table>

Using a list rather than the KD-tree reduced the efficiency by a factor of $\approx 2$. 
Summary

Store the expensive evaluations of $\pi$ or $\hat{\pi}$ in a KD-tree.
Summary

Store the expensive evaluations of $\pi$ or $\hat{\pi}$ in a KD-tree.

Use a delayed-acceptance [PsM]RWM algorithm.
Summary

Store the expensive evaluations of $\pi$ or $\hat{\pi}$ in a KD-tree.
Use a delayed-acceptance [PsM]RWM algorithm.
Adaptive algorithm converges subject to conditions.
Summary

Store the expensive evaluations of $\pi$ or $\hat{\pi}$ in a KD-tree.

Use a delayed-acceptance [PsM]RWM algorithm.

Adaptive algorithm converges subject to conditions.

Need $\beta \propto \alpha_1$ for adaptive portion to play a part.
Summary

Store the expensive evaluations of $\pi$ or $\hat{\pi}$ in a KD-tree.

Use a delayed-acceptance [PsM]RWM algorithm.

Adaptive algorithm converges subject to conditions.

Need $\beta \propto \alpha_1$ for adaptive portion to play a part.

Improvement in efficiency by factor of between 4-7
Summary

Store the expensive evaluations of $\pi$ or $\hat{\pi}$ in a KD-tree.

Use a delayed-acceptance [PsM]RWM algorithm.

Adaptive algorithm converges subject to conditions.

Need $\beta \propto \alpha_1$ for adaptive portion to play a part.

Improvement in efficiency by factor of between 4-7

Code easy-to-use, generic C code for the KD-tree is available.
Summary

Store the expensive evaluations of $\pi$ or $\hat{\pi}$ in a KD-tree.

Use a delayed-acceptance [PsM]RWM algorithm.

Adaptive algorithm converges subject to conditions.

Need $\beta \propto \alpha_1$ for adaptive portion to play a part.

Improvement in efficiency by factor of between 4-7.

Code easy-to-use, generic C code for the KD-tree is available.

Limitations: Need $n \gg 2^d$ for KD-tree to give worthwhile speedup and adequate coverage.
Summary

Store the expensive evaluations of $\pi$ or $\hat{\pi}$ in a KD-tree.

Use a delayed-acceptance [PsM]RWM algorithm.

Adaptive algorithm converges subject to conditions.

Need $\beta \propto \alpha_1$ for adaptive portion to play a part.

Improvement in efficiency by factor of between 4-7

Code easy-to-use, generic C code for the KD-tree is available.

Limitations: Need $n \gg 2^d$ for KD-tree to give worthwhile speedup and for adequate coverage.

Other: could use k nearest neighbours to estimate gradient and curvature;

Thank you!
Summary

Store the expensive evaluations of $\pi$ or $\hat{\pi}$ in a KD-tree.

Use a delayed-acceptance [PsM]RWM algorithm.

Adaptive algorithm converges subject to conditions.

Need $\beta \propto \alpha_1$ for adaptive portion to play a part.

Improvement in efficiency by factor of between 4-7

Code easy-to-use, generic C code for the KD-tree is available.

Limitations: Need $n \gg 2^d$ for KD-tree to give worthwhile speedup and for adequate coverage.

Other: could use k nearest neighbours to estimate gradient and curvature; even to fit a local GP?
Summary

Store the expensive evaluations of $\pi$ or $\hat{\pi}$ in a KD-tree.

Use a delayed-acceptance [PsM]RWM algorithm.

Adaptive algorithm converges subject to conditions.

Need $\beta \propto \alpha_1$ for adaptive portion to play a part.

Improvement in efficiency by factor of between 4-7

Code easy-to-use, generic C code for the KD-tree is available.

Limitations: Need $n \gg 2^d$ for KD-tree to give worthwhile speedup and for adequate coverage.

Other: could use k nearest neighbours to estimate gradient and curvature; even to fit a local GP?

Thank you!