Efficient Implementation of Non-Reversible Rejection-Free Markov Chain Monte Carlo Algorithms

Tomas Vaškevičius

St Anne's College

University of Oxford

A project report submitted in partial fulfilment of the requirements for the degree of

MMathCompSci

May 2017

Abstract

Recently, a new class of Markov chain Monte Carlo (MCMC) algorithms has emerged, which relies on simulation of continuous-time non-reversible piecewise-deterministic Markov processes (PDMP). Such algorithms have been shown to compare favourably to the current state-of-the-art methods and are particularly suitable for the big-data setting, due to the ability to exploit the structure of target distributions. In this project, we implement a flexible and efficient framework for simulating and analysing arbitrary PDMPs. We further show how the current PDMP based MCMC algorithms can be implemented within our framework, and how our framework can assist in solving the current research problems.

Acknowledgements

I would like to express my sincere gratitude to Professor Peter Jeavons, for carefully reading early drafts of this report and helpful advice that he gave. Not only has he been one of my project's supervisors, but also a wonderful personal tutor for the past four years that I have spent at Oxford.

I am also greatly indebted to my other supervisors – Professor Arnaud Doucet and Professor Alexandre Bouchard-Côté. Their guidance and passion for the field has immensely contributed to my appreciation of Markov chain Monte Carlo methods.

Finally, I would like to thank my family – Rytis for enthusiastically reading early drafts of this report, Ieva, for being the best friend I could have ever asked for, and most importantly my parents, without whose love, support and encouragement I would have never even dreamed of studying here.

To my parents

Contents

1 Introduction **1**
- 1.1 A Brief History of the Monte Carlo Method 1
- 1.2 Project Goals 2
- 1.3 Project Outline 2

2 Background Material **3**
- 2.1 Non-Homogeneous Poisson Processes Simulation 3
- 2.2 Markov Processes and their Infinitesimal Generators 7
- 2.3 Piecewise-Deterministic Markov Processes 9

3 Generic Implementation of PDMPs **12**
- 3.1 Designing the framework 12
- 3.2 A basic policy-based implementation of PDMPs 14
- 3.3 Exploiting the structure of the process 18
- 3.4 Implementing efficient policies 20
- 3.5 Testing 27

4 Building a Framework for Output Analysis **28**
- 4.1 Analysis Utilities 28
- 4.2 Running the Process 29
- 4.3 Output Processors 30

5 Applications to MCMC Algorithms **32**
- 5.1 The Bouncy Particle Sampler 32
- 5.2 The Zig-Zag Process 34

6 Conclusion **38**
- 6.1 Reflection on the Development Process 38

A Background Material on MCMC Methods **40**
- A.1 Motivation for Using Monte Carlo Simulations 40
- A.2 The Metropolis-Hastings Algorithm 41

B The Toy Example Poisson Process Time Simulation **44**

C On MCMC Output Analysis **46**
- C.1 Autocorrelation Function 46
- C.2 Effective Sample Size 47

References **49**

1 Introduction

1.1 A Brief History of the Monte Carlo Method

In 1946 a Polish-American mathematician Stanisław Ulam[1] was playing Canfield solitaire while recovering from brain surgery. He got interested in computinig the probability of winning based on the initial layout of cards. After unsucessfully spending a lot of time trying to calculate it by pure combinatorial approaches it occured to him that simply playing the game, say one hundred times and obtaining a rough estimate might be of more practical use [11]. Using repeated random sampling to obtain numerical approximations for intractable problems is at the core of the Monte Carlo method.

The idea of statistical sampling itself was not new at the time. For example, the Italian physicist Enrico Fermi used it to study neutron diffusion in the early 1930s. Such techniques had fallen out of use because of the length and tediousness of the calculations, which at the time had to be performed by humans. However, in February 1946 one of the earliest electronic general-purpose computer ENIAC was announced to the public. The key insight of Stanisław Ulam was that ENIAC's computing power could be used to revive these statistical sampling techniques [23], whose significance was soon recognised by John von Neumann. Thus began the rapid development of Monte Carlo techniques.

The first unclassified paper on the Monte Carlo method appeared in 1949, written by Metropolis and Ulam [24]. Four years later, the first Markov chain Monte Carlo (MCMC) approach was developed in Los Alamos by physicists Metropolis et al. [25]. In 1970 it was generalized by Hastings [21] to what is now known as the Metropolis-Hastings algorithm. Despite extensive use of MCMC algorithms by physicists, it only made an impact on the statistical community in the early 1990s, marked by the seminal paper of Gelfand and Smith [16]. Since then MCMC has been a very active area of research in Statistics. Indeed, almost half of the articles found by a Google Scholar search using the keyword "MCMC" were published after the year 2000.

In 2012 a fundamentally different MCMC algorithm by Peters and de With [27] appeared in the physics literature. This new algorithm relies on simulation of a continuous-time non-reversible piecewise-deterministic Markov (PDMP) process. In contrast, many other MCMC techniques are variations of the Metropolis-Hastings algorithm and hence rely on simulation of a discrete-time reversible Markov chain. The new algorithm by Peters and de With [27] was generalised by Bouchard-Côté et al. [4], where theoretical guarantees and extensive analysis is given. It has been also shown to compare favourably with the current state-of-the-art methods on various Bayesian inference tasks and to be particularly suitable for the big data setting. Similarly promising results were obtained by an even more recent PDMP based MCMC algorithm by Bierkens et al. [2].

[1] Stan, currently one of the most widely used probabilistic programming languages providing full Bayesian inference, is named after Stanisław Ulam [33].

1.2 Project Goals

Apart from very encouraging performance results, the Bouncy Particle Sampler (BPS) [27, 4] and the Zig-Zag process [2] possess one very desirable quality that is absent from most of the other algorithms – non-reversibility of the simulated Markov process. See, for example, Neal [26] or Sun et al. [31] for the role that non-reversibility plays in MCMC algorithms. PDMP based Monte Carlo algorithms are thus rapidly becoming a very active and promising research direction.

Implementing such algorithms is, unfortunately, not the easiest task. The main difficulty comes from the simulation of PDMPs; however, the technique for simulating PDMPs across different MCMC algorithms could be reused. It is hence very inefficient to develop such algorithms from scratch, which unfortunately seems to currently be the case. Writing a generalised and reusable software is difficult and time consuming. Most researchers thus understandably opt to implement their novel algorithms in the quickest and the most direct way possible, which results in code that is hard to reuse.

In this project, we aim to contribute to future research of novel PDMP based Monte Carlo algorithms by developing a framework for efficient simulation and analysis of as general PDMPs as possible. This framework will allow to focus on the structure of the underlying PDMPs rather than on the tedious implementation details. We expect our software to be *efficient*, *flexible*, *reliable*, *relevant* and *easy to use*.

1.3 Project Outline

- In *Chapter 2* we provide the background material on PDMPs and their simulation.
- In *Chapter 3* we design and implement a framework for efficiently simulating arbitrary PDMPs.
- In *Chapter 4* we design an output analysis framework.
- In *Chapter 5* we show how our framework can be used to implement the Bouncy Particle Sampler and the Zig-Zag process.
- In *Chapter 6* we summarise what we managed to achieve in this project and reflect on the development process.

2 Background Material

In this chapter we provide the background material necessary for understanding the current MCMC algorithms which rely on simulation of piecewise-deterministic Markov processes. We further complement this chapter with Appendix A where we provide motivation for using Monte Carlo techniques in the context of Bayesian statistics and introduce the Metropolis-Hastings algorithm – a cornerstone for many MCMC algorithms including the Bouncy Particle Sampler.

2.1 Non-Homogeneous Poisson Processes Simulation

Non-homogeneous Poisson processes play a central role in the simulation of piecewise-deterministic Markov processes. We will later see that it is the main source of randomness in the Bouncy Particle Sampler algorithm. In this section we first give the definition of homogeneous Poisson processes and then discuss why such processes appear so naturally in many real-world applications. Next, we generalize to the non-homogeneous case and present a number of different simulation techniques. We mostly refer to textbooks by Devroye [9], Feller [13] and Ross [30], where more careful treatment of the following material can be found.

2.1.1 Homogeneous Poisson Processes

Consider a sequence of events occuring at random times $0 < T_1 < T_2 < \dots$ Let N be a counting process, given by $N(s,t] := \#\{i \mid T_i \in (s,t]\}$ where $0 \leq s \leq t$. In many rea-world applications, such as modelling the number of incoming phone calls at a telephone call centre or the number of decays from a radioactive source in a given time interval, the following assumptions are met:

1. *Independent increments.*
   Let $0 \leq s_1 \leq t_1 \leq s_2 \leq \cdots \leq s_n \leq t_n$. Then the random variables $N(s_1, t_1], \dots, N(s_n, t_n]$ are independent.

2. *Stationarity.*
   For any $0 \leq s \leq t$ the random variables $N(s,t]$ and $N(0, t-s]$ have the same distribution, so that the number of events in a given time interval depends only on the length of that interval.

The remarkable consequence of these conditions is that $\exists \lambda \geq 0$ such that for any $0 \leq s \leq t$ the random variable $N(s,t]$ has a Poisson distribution with parameter $\lambda(t-s)$.

We now sketch a proof of the above claim (see [13] for full details). Assume that $\mathbb{E}[N(0,1]] = \lambda < \infty$. Partition an interval of unit length into m subintervals of length $h := m^{-1}$. By the two conditions above, the number of events in each of these subintervals are independent and identically distributed (i.i.d) with distribution of $N(0,h]$. Let $p_k(h)$ be a shorthand for $\mathbb{P}(N(0,h] = k)$. Then the expected number of

subintervals containing at least one event is $h^{-1}(1-p_0(h))$. Intuitively, as $h \to 0$ the number of subintervals containing more than one event vanishes and hence we expect that $\lim_{h\to 0} h^{-1}(1-p_0(h)) = \lambda$. Hence we can write

$$\begin{aligned} p_0(h) &= 1 - \lambda h + o(h), \\ p_1(h) &= \lambda h + o(h). \end{aligned} \tag{2.1}$$

The proof concludes by considering $p_k(t+h)$ conditioned on the number of events that happened up to time t, which allows us to derive a set of ordinary differential equations whose solutions confirm that $N(0,t] \sim \text{Poisson}(\lambda t)$ for any $t \geq 0$.

Finally, note that for $k \in \mathbb{N}$ we have $\mathbb{P}(T_{k+1} > T_k + t \mid T_k) = \mathbb{P}(N(0,t] = 0) = \exp(-\lambda t)$ and hence the interarrival time $T_{k+1} - T_k$ conditioned on T_k is exponentially distributed with parameter λ. From independent increments and stationarity conditions it also follows that interarrival times are independent. Hence, we can simulate a homogeneous Poisson process with rate λ using Algorithm 1.

Algorithm 1 Homogeneous Poisson process simulation

```
T_0 ← 0
for i = 1, 2, ... do
    Generate U ~ Uniform(0, 1)
    E_λ ← −log(U)/λ          ▷ E_λ is an Exponential random variable with rate λ
    T_i ← T_{i−1} + E_λ
end for
```

2.1.2 Generalising to the Non-Homogeneous Case

Suppose we want to model the number of customers arriving at a supermarket. While it is reasonable to assume that different customers arrive independently of one another, the customer counting process is not stationary – the number of customers will clearly be larger in the evening than, for example, during the regular working hours. We can model such processes by dropping the stationarity condition introduced in the previous section.

Instead of parameterizing our process by some constant $\lambda \geq 0$, we introduce an intensity function $\lambda(t) \geq 0$, where t is the time parameter, and further assume that $\int_0^\infty \lambda(s)\,ds = \infty$. Let $p_k(t,h)$ be a shorthand for $\mathbb{P}(N(t,t+h] = k)$. The price we pay for non-stationarity is that Equations 2.1 become:

$$\begin{aligned} p_0(t,h) &= 1 - \lambda(t)h + o(h), \\ p_1(t,h) &= \lambda(t)h + o(h). \end{aligned} \tag{2.2}$$

Let $\Lambda(t) := \int_0^t \lambda(s)\,ds$ where $t \geq 0$. Just like in the homogeneous case, by considering $p_k(s,t+h)$, conditioning on the number of events that happened between times s and $s+t$ and using equations given in 2.2, we can derive a set of differential equations, whose unique solution shows that $N(s,t+s] \sim \text{Poisson}(\Lambda(t+s) - \Lambda(s))$. We can

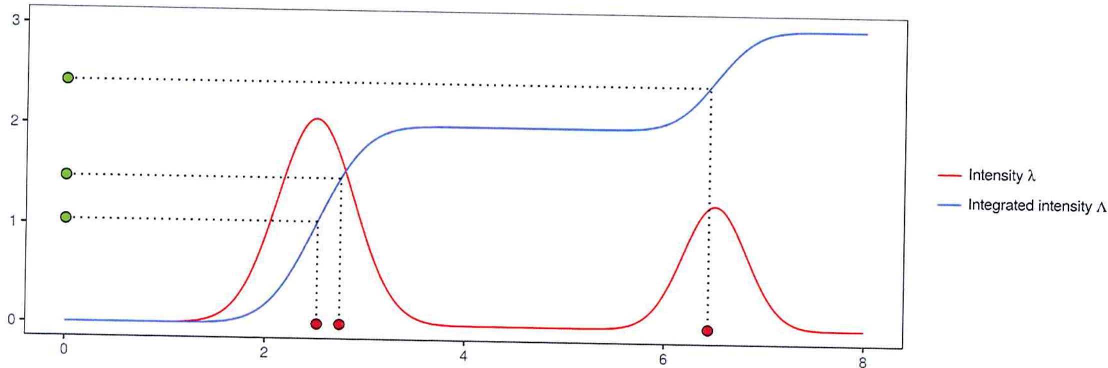

Figure 2.1: The time scale transformation method for simulating a non-homogeneous Poisson process with intensity function $\lambda(t)$. We simulate a homogeneous Poisson process (with rate 1) along the y-axis and represent the generated samples with the green points. After mapping these points along the quantile function of integrated intensity Λ, we obtain samples from a non-homogeneous Poisson process with rate λ, shown using the red points on the x-axis.

see that if $\lambda(t)$ is constant, then Equations 2.2 do not depend ont t and we go back to Equations 2.1 for the homogeneous Poisson process. We will now present three techniques for non-homogeneous Poisson process simulation, all of which can be found in Devroye [9].

Simulation by Time Scale Transformation

A non-homogeneous Poisson process can be seen as a homogeneous Poisson process under a certain time scale transformation. Let $\{N(0,t] \mid t \geq 0\}$ be a Poisson process of rate 1 and define a new counting process $M(0,t] := N(0,\Lambda(t)]$. Since $\Lambda(t)$ is non-decreasing, M has the independent increments property. Also $M(s,t]$ is distributed as $N(\Lambda(s),\Lambda(t)] \sim \text{Poisson}(\Lambda(t) - \Lambda(s))$ which is enough to recover Equations 2.2. Hence, $\{M(0,t] \mid t \geq 0\}$ is a non-homogeneous Poisson process with intensity function $\lambda(t)$. See Figure 2.1 for a graphical representation. Algorithm 2 shows how the above idea can be used to simulate a non-homogeneous Poisson process. This algorithm is

Algorithm 2 Non-homogeneous Poisson process simulation by time scale transformation

```
1 T_0 ← 0
2 S ← 0                                 ▷ Last event time of a Poisson process N with rate 1
3 for i = 1, 2, ... do
4     S ← next sample from N (given by, for example, Algorithm 1)
5     T_i ← inf{t | Λ(t) ≥ S}
6 end for
```

computationally tractable only for intensity functions where we can compute line 5 quickly; however, most often we do not even have Λ in closed form.

Simulation by Thinning

Suppose M is a non-homogeneous Poisson process with intensity $\lambda(t)$. Let $\mu(t)$ be a function such that $\forall t \geq 0$ $\lambda(t) \leq \mu(t)$. Then we can think of M as a thinned out version of a non-homogeneous Poisson process N with intensity $\mu(t)$ – an event generated by N at time t is counted (independently of others) with probability $\frac{\lambda(t)}{\mu(t)}$. Using 2.2 we can derive:

$$\begin{aligned}
\mathbb{P}(M(t,t+h]=0) &= \mathbb{P}(M(t,t+h]=0 \mid N(t,t+h]=0)\,(1-\mu(t)h+o(h)) \\
&\quad + \mathbb{P}(M(t,t+h]=0 \mid N(t,t+h]=1)\,(\mu(t)h+o(h)) \\
&\quad + o(h) \\
&= (1-\mu(t)h+o(h)) + (1-\frac{\lambda(t)}{\mu(t)}+o(h))(\mu(t)h+o(h)) + o(h) \\
&= 1-\lambda(t)h+o(h), \\
\mathbb{P}(M(t,t+h]=1) &= \mathbb{P}(M(t,t+h]=1 \mid N(t,t+h]=1)\,(\mu(h)+o(h)) + o(h) \\
&= \lambda(t)h+o(h).
\end{aligned}$$

Also, the independent increments property for M holds (since it does for N) and hence M is a non-homogeneous Poisson process with intensity $\lambda(t)$ as required. We provide the pseudocode for the above technique in Algorithm 3.

Algorithm 3 Non-homogeneous Poisson process simulation by thinning

```
T ← 0          ▷ Last event time of a Poisson process N with intensity μ(t) ≥ λ(t)
for i = 1,2,... do
    counted ← false
    while not counted do
        T ← next sample from N (given by, for example, Algorithm 2)
        Generate U ~ Uniform(0,1)
        if U ≤ λ(T)/μ(T) then counted ← true
    end while
    T_i ← T
end for
```

Simulation by the Composition Method

The technique to be presented plays a crucial role in the big-data setting as we will see in the following chapters. It will allow us to exploit special structual properies of probability distributions for efficient simulation.

Sometimes the intensity function $\lambda(t)$ can be naturally represented as a sum of non-negative functions: $\lambda(t) = \sum_{i=1}^{m} \lambda_i(t)$. Denote the associated integrated intensity functions by $\Lambda_i(t)$. Let the main Poisson process be N, and denote by N_i a Poisson process induced by the intensity function $\lambda_i(t)$. Assume further that processes $N_1, N_2, \ldots, N_m$ are independent. Then, to simulate the first event time T of Poisson

process N we can simulate event times T_i of processes N_i for $i \in \{1, 2, \dots, m\}$ and take the minimum. To see why this works, let E_i be i.i.d. Exp(1) random variables and note that:

$$\begin{aligned}
\mathbb{P}(T_1 > t, T_2 > t, \dots, T_m > t) &= \prod_{i=1}^{m} \mathbb{P}(T_i > t) && \text{(by independence)} \\
&= \prod_{i=1}^{m} \mathbb{P}(E_i > \Lambda_i(t)) && \text{(by Algorithm 2)} \\
&= \prod_{i=1}^{m} \exp(-\Lambda_i(t)) \\
&= \exp(-\Lambda(t)) \\
&= \mathbb{P}(T > t) \, .
\end{aligned}$$

Assume that the first event time t_1 was induced by the process N_k. To simulate the second event time t_2, we can repreat the same procedure: simulate the first event times of Poisson processes with intensity functions $\lambda_i'(t) := \lambda_i(t + t_1)$, take the minimum and add t_1 to the result. Now comes the great part – instead of resimulating event times for all processes, we can reuse the results we got before and only simulate the next event for process k. To see why this is true, note that for $i \neq k$ we have:

$$\begin{aligned}
\mathbb{P}(T_i > t \mid T_i > t_1) &= \mathbb{P}(E_i > \Lambda_i(t) \mid E_i > \Lambda_i(t_0)) \\
&= \mathbb{P}(E_i > \Lambda_i(t) - \Lambda_i(t_1)) \\
&= \mathbb{P}(E_i > \Lambda_i'(t - t_1)) \\
&= \mathbb{P}(N_i(t_1, t] = 0) \, .
\end{aligned}$$

This concludes the proof of correctness for simulation via the composition method. See Algorithm 4 for pseudocode.

We end this section by providing a more intuitive argument as to why this algorithm is valid. Note that for the main Poisson process N, the probability of exactly one event happening in $(t, t+h)$ is $\lambda(t) + o(h)$. We can rewrite it as:

$$\lambda(t) + o(h) = \left(\sum_{i=1}^{m} \lambda_i(t) \right) + o(h) = \sum_{i=1}^{m} (\lambda_i(t) + o(h)) \, .$$

Each term in the RHS of the above equation show the contribution of processes N_i to the main process N. Hence simulating events from individual components N_i and ordering the event times yield the event times of Poisson process with rate $\lambda(t)$.

2.2 Markov Processes and their Infinitesimal Generators

First, we briefly recall how a Markov process is defined. Let $(X_t)_{t \geq 0}$ be a stochastic process taking values in some measurable space $(E, \mathcal{E})$. Let $\{\mathbb{P}_x \mid x \in E\}$ [1]

[1] We also assume that the randomness of our Markov process is carried by letting $X_t = X_t(\omega)$, where $\omega \in \Omega$. Further, we assume that $(X_t)_{t \geq 0}$ is adapted to some filtration $\{\mathcal{F}_t\}$ and $\forall x \in E$ $(\Omega, \mathcal{F}, \{\mathcal{F}_t\}, \mathbb{P}_x)$ is a filtered probability space.

Algorithm 4 Non-homogeneous Poisson process simulation by the composition method

$D \leftarrow$ Some data structure for efficient storage of event times (e.g. heap)
for $i = 1, 2, \ldots, m$ **do**
  $T_i \leftarrow$ The first event time of Poisson process N_i
  Insert (T_i, i) to D where the elements are ordered by the first parameter.
end for
for $i = 1, 2, \ldots$ **do**
  $(T, k) \leftarrow$ An element with the smallest time in D
  $T_k \leftarrow$ Simulate the next event time of Poisson process N_k
  Insert (T_k, k) to D
  $T^{(i)} \leftarrow T$ $\triangleright$ Set the i-th event time of the main Poisson process.
end for

be a collection of probability measures and further define the *transition function* $P : [0, \infty) \times E \times \mathcal{E} \to [0, 1]$ by:

$$P(t, x, \Gamma) = \mathbb{P}_x(\{X_t \in \Gamma\}).$$

It represents the probability, that our process, started at position $x \in E$, is in some set $\Gamma \in \mathcal{E}$ at time t. Finally, we require that $\forall t, s \geq 0, x \in E, \Gamma \in \mathcal{E}$

$$\mathbb{P}_x(\{x_{t+s} \in \Gamma\} \mid \mathcal{F}_t) = P(s, x_t, \Gamma) \text{ (a.s. with respect to } \mathbb{P}_x).$$

This condition ensures the *Markov property* – the evolution of the process after time t is independent of the past if we know the value of the process at time t. See the classical monograph by Dynkin [10, Chapter 3] for more details and a rigorous definition of Markov processes.

One can view the transition function as a family of linear operators $\{P_t\}_{t \geq 0}$ acting on bounded $\mathcal{E}$-measurable functions f by:

$$P_t f(x) = \int_E f(y) P(t, x, dy).$$

From the Markov property, we can deduce that $\{P_t\}$ forms a (contraction) semigroup in the following way:

$$\forall t, s \geq 0 \;\; P_t P_s = P_{t+s}.$$

Sometimes, instead of defining the transition function, it is easier to define a Markov process in terms of its movement on an infinitesimal time scale. An *infinitesimal generator* $\mathcal{A}$ of a semigroup $\{P_t\}$ is given by:

$$\mathcal{A} f(x) = \lim_{t \to 0^+} \frac{P_t f(x) - f(x)}{t}. \tag{2.3}$$

We denote the set of functions for which the above limit is well-defined by $\mathcal{D}_{\mathcal{A}}$ and call it the *domain* of $\mathcal{A}$. Intuitively, for small t we expect that

$$P_t f(x) \approx f(x) + t \mathcal{A} f(x)$$

should hold. Under some very mild assumptions, the infinitesimal generator uniquely determines the transition function of the associated transition semigroup. See [10, Chapters 1 and 2] for discussion on infinitesimal operators of contraction semigroups and relation to the transition functions.

For MCMC applications, we are particularly interested in *invariant distributions.* A probability measure μ is invariant, if a Markov process distributed according to μ at time 0, stays at distribution μ forever. More formally, a probability measure μ on $(E, \mathcal{E})$ is invariant for the Markov process defined above, if:

$$\forall \Gamma \in \mathcal{E} \ \forall t \geq 0 \ \mu(\Gamma) = \int_E P(t, x, \Gamma)\mu(dx).$$

Verifying the above equation is often very hard. An alternative way to check if μ is an invariant distribution is available if we know the infinitesimal generator $\mathcal{A}$ of our process. Since for invariant measure μ the distribution of our process does not change over time and since $\mathcal{A}$ describes the movement of our process, we expect that on average, our process "does not move" with respect to μ, if μ is invariant. Indeed, it was shown in [8, Proposition 34.7] that μ is invariant if and only if:

$$\forall f \in \mathcal{D}_{\mathcal{A}} \ \int_E \mathcal{A}f(x)\mu(dx) = 0. \tag{2.4}$$

To help better understand the infinitesimal generators of Markov processes, we finish this section with a concrete example. Let $N_t := N(0, t]$ be a homogeneous Poisson process (following the notation from Section 2.1) with rate $\lambda > 0$. Recall that the interarrival times follow i.i.d Exp(λ) distribution. By the momoryless property of the exponential distribution (it is in fact the only continuous time distribution having such property), one can deduce that the process starts afresh at every time step $t > 0$ in the sense that $N_s^t := N_{t+s} - N_t$ is again a Poisson process with rate λ. So $(N_t)_{t \geq 0}$ is a Markov process. What can we say about its infinitesimal generator $\mathcal{A}$? Let f be some function defined on $\mathbb{N}_{\geq 0}$. By Equation 2.1 we have:

$$P_h f(n) = f(n)(1 - \lambda h + o(h)) + f(n+1)(\lambda h + o(h)) + o(h)$$

and hence by Equation 2.3 we deduce that

$$\mathcal{A}f(n) = \lambda(f(n+1) - f(n)). \tag{2.5}$$

This expression is not surprising – on an infinitesimal time scale our function f can only change by $f(n+1) - f(n)$ while the rate λ controls how often such changes can occur.

2.3 Piecewise-Deterministic Markov Processes

We will now introduce a class of stochastic processes governed by random jumps at a sequence of random times $T_1 < T_2 < \ldots$ and deterministic motion in-between. Piecewise-deterministic Markov processes (PDMP) were introduced by Davis [7] where

it was claimed that this class of stochastic processes is rich enough to include virtually all non-diffusion models in applied probability. The material below is not presented in full generality and mathematical rigour; instead, we attempt to provide an intuitive understanding to feel comfortable with the following chapters of this project report. We refer to the textbook by Davis [8].

2.3.1 The definition

A PDMP is determined by the three *local characteristics* $(\phi, \lambda, \mathcal{Q})$:

1. Let $\phi : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$ be a continuous map differentiable in its second parameter such that $\forall t, s \in \mathbb{R}\ \ \phi(\cdot, t+s) = \phi(\phi(\cdot, s), t)$. We call ϕ the *flow function* since the PDMP process evolves deterministically according to ϕ in-between the jump times.

2. The *jump rate* $\lambda : \mathbb{R}^d \to \mathbb{R}_+$ together with the flow determines the jump times $T_1, T_2, \dots$ In particular, if the current time is t and the process is at state $x \in \mathbb{R}^d$, the next jump time $t + T$ is determined by the first arrival time T of a non-homogeneous Poisson process with intensity function $\lambda(\phi(x, \cdot))$. We assume that $\forall x \in \mathbb{R}^d\ \exists \varepsilon > 0$ such that $\lambda(\phi(x, \cdot))$ is integrable on $(0, \varepsilon)$.

3. Finally, the actual jumps at the jump times are governed by the *Markov kernel* $\mathcal{Q} : \mathbb{R}^d \times \mathcal{B}(\mathbb{R}^d) \to [0, 1]$ (i.e. $\forall x \in \mathbb{R}^d\ \ \mathcal{Q}(x, \cdot)$ is a probability measure and $\forall B \in \mathcal{B}(\mathbb{R}^d)\ \ \mathcal{Q}(\cdot, B)$ is measurable). We also require that $\forall x\ \ \mathcal{Q}(x, \{x\}) = 0$.

The great thing about PDMPs is that we know precisely what their infinitesimal generators are. It was shown in [8, Section 26], that an infinitesimal generator $\mathcal{A}$ of a PDMP (ϕ, λ, Q) is given by:

$$\mathcal{A}f(x) = \frac{\mathrm{d}}{\mathrm{d}t} f(\phi(x, t))\Big|_{t=0} + \lambda(x) \int_{\mathbb{R}^d} (f(y) - f(x)) \mathcal{Q}(x, dy), \tag{2.6}$$

where $\langle \cdot, \cdot \rangle$ is the dot product. The domain $\mathcal{D}_{\mathcal{A}}$ of the generator $\mathcal{A}$ is also shown to be the set of bounded functions f such that $\forall x \in \mathbb{R}^d\ \ f(\phi(x, t))$ is absolutely continuous as a function of t. As explained in [8], the above expression has an easy intuitive interpretation. If $\lambda \equiv 0$, we end up with the first term only, which describes the deterministic motion of our process. On the other hand, if there is no flow (i.e. $\forall t, x\ \ \phi(x, t) = x$), then we end up with a generator of a Markov jump process (see how this relates to Equation 2.5).

2.3.2 A Toy Example

Suppose we want to model a particle moving along straing lines and changing direction at random in a two dimensional space. Further, assume that we want to make it hard for the particle to go too far away from the origin (where the difficulty increases with the distance from the origin). We can model such process as a PDMP over the space $Z = \mathbb{R}^4 = \mathbb{R}^2 \times \mathbb{R}^2 = (X, V)$, as follows:

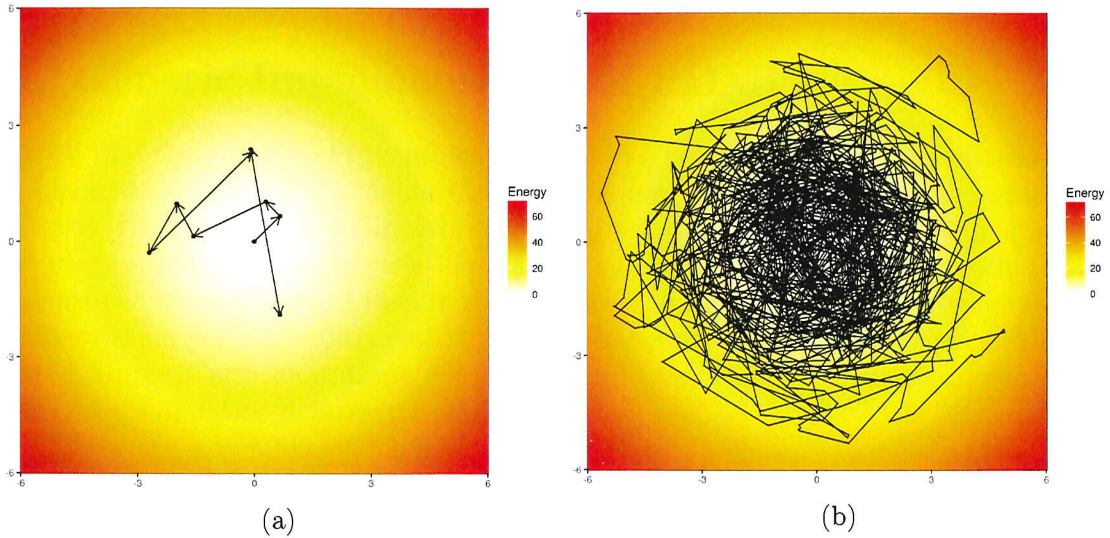

Figure 2.2: The toy example considered in Section 2.3.2. Figure a shows the first 7 simulated events. We can see how the process evolved after 1000 events in Figure b.

1. Since we want the particle to move in straight lines, we can, for example, let $\phi(z,t) = \phi((x,v),t) = (x+vt,v)$. That is, we change the particle's position x based on velocity v. The velocity does not change during the deterministic motion.

2. Now we want to make it hard for the particle to go far away from the origin. We can achieve that by letting the jump-rate function λ be defined as $z = (x,v) \mapsto \max\{0, \langle x,v\rangle\}$. We take the inner product, since we want to take the particle's movement direction into account. We also take the maximum function, since the jump rate must be non-negative. Note that in such a scenario, the particle can move freely directly towards the origin. However, as the distance from the origin increases, the jump rate also increases, so that the particle is more likely to change direction and go back.

3. At jump times, we leave the particle location unchanged, but sample a new velocity uniformly from, $S^1 := \{(v_1,v_2) \in \mathbb{R}^2 \mid v_1^2 + v_2^2 = 1\}$.

See Figure 2.2 for a sample simulation of this process. The Bouncy Particle Sampler is a very similar algorithm. By carefully choosing the jump rate function and the transition kernel, we will be able to guide our particle, so that the resulting invariant distribution is the one we want to sample from.

3 Generic Implementation of PDMPs

Recall that the purpose of this project is to build a flexible and efficient framework for implementing and testing Monte Carlo techniques based on piecewise-deterministic Markov process (PDMP) simulation. We hope that such tools will contribute to future research by allowing new ideas to be quickly implemented and benchmarked against the current state-of-the-art. It is, however, hard to anticipate the exact details of algorithms yet to be discovered. Therefore, before discussing the current PDMP based Monte Carlo algorithms, we design and implement a framework capable of efficiently simulating arbitrary PDMPs.

There are a number of trade-offs that need to be considered before choosing a language for implementing our framework. Doing it in Python or R, for instance, would allow us to effortlessly leverage rich plotting and statistical output analysis packages at the cost of execution speed. Speed is, however, of cruicial importance to us. Implementing novel algorithms using the wrong tools and bencharking against the current state-of-the-art MCMC algorithms (often implemented, at least partially, in C/C++) might provide too pessimistic results due to not realising the full potential of the new algorithm. Other possiblilities are fast JIT compiled languages like Java, C# or Julia, the latter being a particularly interesting choice, with its focus on fast numerical computations and built-in support for parallelism and distributed computing.

We choose to implement the framework in C++. Even though in some cases JIT compiled languages can potentially execute faster than C/C++, most of the benchmarks still show C/C++ superiority in terms of efficiency. It is also worth noting that more than a decade ago processsor manufacturers' focus has shifted on engineering multi-core architectures rather than increasing clock speeds (see Sutter [32]). It is thus natural to expect that the algorithms to come will eventually be forced to exploit multi-core hardware to its full potential in order to be competetive. Writing our framework in C++ allows to easily integrate CUDA code and take advantage of modern GPUs (not explored in this project). Finally, the C++ language is very expressive and allows us to make some design decisions which would not be possible in most of the other languages (discussed in the next section).

3.1 Designing the framework

In Section 2.3.1 we defined a PDMP by its three local characteristics $(\phi, \lambda, \mathcal{Q})$, namely the deterministic flow, intensity function and the Markov kernel. We would expect our framework to be flexible enough to easily change any of these characteristics with no/minimal modifications to the others. For example, consider two PDMPs, which differ only by their Markov kernels. Having implemented the first algorithm, we should have an easy way to implement the other, simply by changing the Markov kernel and not touching any of the other existing code.

Implementing such a modular framework requires to identify a set of components

allowing to simulate an arbitraty PDMP and implementing them in a loosely coupled way. We also need a unified way to manage these components in a coherent manner and a mechanism to easily substitute one component for another (e.g. changing one Markov kernel to another as discussed above).

One possible way to achieve this is by using the *Strategy pattern* (fully described in Gamma et al. [15]). We can agree on the interfaces of each component and implement the main PDMP class, which knows how to manipulate each of the components in a logically feasible way. The user of the framework can then reimplement any component of the algorithm and pass it to the main PDMP class, thus choosing the behaviour of the resulting PDMP. While this is a good solution to our problem, there is one inefficiency we would like to get rid of. The Strategy pattern is designed to choose the behaviour at runtime, which comes at the cost of relying on virtual methods. As analysed in this great blog post[1], the performance penalties associated with virtual methods are: an extra layer of indirection, difficulties in inlining the method calls and incresing object sizes by an extra pointer, thus potentially contributing to more L1 cache misses. Luckily, we do not need to customize the behaviour of our PDMP at runtime, since we know exactly what kind of process we want to run, before it even starts executing. A compile-time variant of the Strategy pattern called the *policy-based design* (popularised in Alexandrescu [1]) seems to be an excellent choice for us. We will now briefly describe how it works.

The idea of the policy-based design is to assemble a class (called the *host*) with complex behaviour from many other smaller classes (called *policies*). The host class is parameterized via template parameters which are instantiated with particular implementations of policies. Each of the policies should define an orthogonal (as much as possible) aspect of the behaviour of the resulting class, via its *implicit interfaces.* The host class either inherits from its policies or uses them as class members. A particular advantage of inheriting from the policies is that the resulting host class has an interface specified by its policies (so the usual relationship between the base and derived classes is inverted here). This allows for complex interaction between different types of policies, of which the host class need not be aware, as we shall see in Section 3.4. Since we can easily change between different policies by simply changing the template parameter, we can achieve a combinatorial number of behaviours by implementing a core set of policies. Finally, since C++ templates is a compile-time mechanism, any incompatibilities between particular policies used will be found at compilation time.

This policy-based design requires support for templates and multiple inheritance. Therefore, it is often associated to the C++ language, as not many other languages possess both of these features. For a much more detailed discussion on why, when and how policy-based design should be used, including specific code examples, refer to [1]. We will now move on to the next section where we will see the policy-based design in action.

[1] http://eli.thegreenplace.net/2013/12/05/the-cost-of-dynamic-virtual-calls-vs-static-crtp-dispatch-in-c

3.2 A basic policy-based implementation of PDMPs

Since the policy-based design is rather specific to C++, instead of presenting pseudocode examples we will illustrate the design of our PDMPs framework with C++ code examples. These examples will reflect the main ideas of the design; however, due to space limitations and for clarity reasons some parts of the design will be omitted, while others significantly simplified. The true implementation can be seen at the project's code repository[2].

The most challenging part of designing policy-based software is deciding on what the actual policies are. In the ideal case, policies would be completely orthogonal and unaware of each other; however, our case is not that simple. The Poisson process simulation, for example, needs to be aware of the deterministic motion and the intensity function. On the other hand, the flow and intensity does not need to know about the details of the Poisson process simulation. As we will see in Section 3.3, for efficiency reasons even the Markov kernel may need to communicate with the Poisson process simulation policy.

To simulate a PDMP, we only need to know how to simulate one iteration of the process, since the process has no memory (by definition). One iteration of a PDMP (started at state x) can be simulated as follows:

1. Simulate the first arrival time T of an inhomogeneous Poisson process with rate $t \mapsto \lambda(\phi(x, t))$.
2. Set $x' := \phi(x, T)$.
3. Calculate x_{new} by sampling from the measure $\mathcal{Q}(x', \cdot)$.

This algorithm naturally suggests us to choose three policies: the Poisson process simulation policy, the Markov kernel policy and the deterministic flow policy. See Listing 1 for the host class implementation using these three policies, which will serve as the cornerstone of our PDMPs framework. Note that we can also look at this implementation as a compile-time variant of the *template method pattern* (also described in [15]). Indeed, the host class provides the skeleton for a PDMP simulation algorithm, while the exact implementation details are delegated to the policy classes.

Lets now elaborate on what is going on in Listing 1. First note, that the simulation method allows for any definition of the State class. This way we allow the client of the library to customise its process state representation. For example, some users might want to represent the states using a built in array or standard library containers, while others might use some third party libraries or their own data structures customised for their use cases. The code will compile as long as the state representation's implicit interface is compatible with all three given policies.

For this project purposes, we will only be interested in the $\mathbb{R}^{2d}$ state spaces, where the first d coordinates represent the position of a particle and the last d coordinates

[2]The project code can be found at https://github.com/TomasVaskevicius/bouncy-particle-sampler. The examiner can see the content of the repository at the day of submission deadline (22nd of May, 2017), by clicking on commits and then clicking on "<>" button located at the right of the commit of the latest valid date.

Listing 1 A host class for simulating PDMPs.

```
1 template<class PoissonProcess, class MarkovKernel, class Flow = LinearFlow>
2 class Pdmp :
3   public PoissonProcess, public MarkovKernel, public Flow {
4
5  public:
6
7   // Specify the implicit interfaces for the policy classes.
8   using PoissonProcess::getJumpTime;
9   using MarkovKernel::jump;
10  using Flow::advanceStateByFlow;
11
12  template<class State>
13  State simulateOneIteration(const State& state) {
14    auto iterationTime = getJumpTime(state, *this);
15    State stateBeforeJump = advanceStateByFlow(state, iterationTime);
16    State stateAfterJump = jump(stateBeforeJump, *this);
17    return stateAfterJump;
18  }
19 }
```

represent the velocity (similarly to the example process in Section 2.3.2). We do not consider other state spaces in this project, since all current PDMP based Monte Carlo algorithms use such state space. See Listing 2 for the interface. Internally, we use a lightweight and fast linear algebra library Eigen3 [20] for representing real valued vectors (see lines 5 and 6), which also provides a lot of useful functionality, such as norm and inner product calculations, matrix operations, operator overloads for multiplying a vector with a scalar, etc. Note how this representation makes the PDMP host class completely agnostic of the specifics of our state space internals.

Since the current PDMP based Monte Carlo algorithms all rely on constant velocity flow (such flow was used in the toy example Section 2.3.2), we will restrict ourselves to a LinearFlow policy (the default template argument in line 1 of Listing 1), implemented in Listing 3. Note that the implementation of the advanceStateByFlow method places the following requirements on the State representation:

1. It should have a data member called position.
2. It should have a data member called velocity.
3. It should be constructible by passing in a position and velocity representation.
4. The velocity vector can be multiplied by a scalar (time) and added to the position vector.

If any of these conditions are not satistied, the compiler will point it out. This is the intended behaviour, since the LinearFlow conceptually would not make any sense for a state not having position and velocity. Also note that the clients of our library do not have to use our state representation given in Listing 2. They can define their own representations, as long as the implicit interface is satisfied.

Listing 2 An interface for states given by position and velocity vectors.

```
1 template<typename RealType, int Dimension>
2 struct PositionAndVelocityState {
3
4   template<int N>
5   using RealVector = Eigen::Matrix<RealType, N, 1>;
6   using DynamicRealVector = Eigen::Matrix<RealType, Eigen::Dynamic, 1>;
7
8   static const int kDimension = Dimension;
9
10  PositionAndVelocityState(
11    RealVector<Dimension / 2> position,
12    RealVector<Dimension / 2> velocity);
13
14  DynamicRealVector getSubvector(std::vector<int> ids) const;
15
16  template<class VectorType>
17  PositionAndVelocityState constructStateWithModifiedVariables(
18    std::vector<int> ids, VectorType modification) const;
19
20  const RealVector<Dimension / 2> position;
21  const RealVector<Dimension / 2> velocity;
22 };
```

Two things remain to be discussed regarding the PDMP host class implementation given in Listing 1. First, in lines 14 and 16 we pass the host class object to the policy classes via the *this argument. As a result, the policy class can call any methods provided by the other policies, if needed[3]. For example, this technique allows the Poisson process policy to access the flow implementation provided by the flow policy.

The second (and final) issue is dealing with the policy classes constructor parameters. We do not want to restrict the policy classes in any way, so we cannot put any constraints on the number and types of constructor parameters. This is indeed possible to achieve by wrapping constructor parameters of each policy class in a tuple (provided by the standard library) and using template metaprogramming tricks and modern C++ features to unwrap each of the tuples and pass as arguments to the associated policy classes. See the project code repository for implementation details.

This concludes our basic implementation of a generic way to simulate PDMPs. See Listing 4 for example code constructing the toy PDMP from Section 2.3.2 using our framework. In the following sections we will discuss some efficient simulation opportunities which arise for special kind of PDMPs and provide an implementation based on the framework presented in this section.

[3]Alternatively, instead of passing the *this object, we could only provide the type of *this object via explicitly specifying it in the method template parameter via *decltype*(*this). Then, the policy class can access all host class methods by safely downcasting itself to the host class by calling static_cast<HostClassType*>(this)->someMethodProvidedByAnotherPolicy().

Listing 3 An implementation of the linear flow policy.

```
 1 class LinearFlow {
 2  public:
 3
 4   template<class State, typename RealType>
 5   static State advanceStateByFlow(State state, RealType time) {
 6     auto position = state.position;
 7     auto velocity = state.velocity;
 8     return State(position + velocity * time, velocity);
 9   }
10 };
```

Listing 4 An implementation of the toy PDMP from the Section 2.3.2.

```
 1 class ToyMarkovKernel {
 2  public:
 3
 4   template<class State, class HostClass>
 5   State jump(State state, const HostClass& hostClassObject) {
 6     auto newVelocity = state.velocity;
 7     for (int i = 0; i < newVelocity.size(); i++) {
 8       newVelocity[i] = ...; // Sample from Normal(0, 1) distribution.
 9     }
10     newVelocity /= newVelocity.norm(); // Normalise the new velocity.
11     return State(state.position, newVelocity);
12   }
13 };
14
15 class ToyPoissonProcessSimulator {
16  public:
17
18   // See Appendix B for the derivation of the equations used here.
19   template<class State, class HostClass>
20   auto getJumpTime(State state, const HostClass& hostClassObject) {
21     auto xv = state.position.dot(state.velocity);
22     auto vSquaredNorm = pow(state.velocity.norm(), 2);
23     auto U = ...; // Sample from Uniform(0, 1) distribution.
24     if (xv <= 0) {
25       return (-xv + sqrt(-2 * vSquaredNorm * log(U))) / vSquaredNorm;
26     } else {
27       return (-xv + sqrt(xv * xv - 2 * vSquaredNorm * log(U)))
28              / vSquaredNorm;
29     }
30   };
31 };
32
33 // Recall that we have set the LinearFlow as a default template argument.
34 using ToyPdmp = Pdmp<ToyPoissonProcessSimulator, ToyMarkovKernel>;
35 ToyPdmp toyPdmp;
36
37 ... // Generate samples from the toy PDMP.
```

3.3 Exploiting the structure of the process

In the previous section we have presented a general framework for implementing PDMPs. It is, however, of a relatively low level of abstraction and delegates many specific implementation details to the user. We will build another layer of higher abstraction on top of the current implementation. This additional layer will automatically exploit the structure of the process, where possible, in order to achieve remarkable computational gains. The exact implementation details will be hidden from the client of our library, but remain flexible in case some parts need to be customised for special needs. In this section, we identify this special structure of certain PDMPs and discuss how to exploit it. The implementation details will be explored in the next section.

Recall Equation 2.6 giving a general form[4] for infinitesimal generators of PDMPs:

$$\mathcal{A}f(x) = \underbrace{\frac{\mathrm{d}}{\mathrm{d}t} f(\phi(x,t))\bigg|_{t=0}}_{\text{Deterministic motion}} + \underbrace{\lambda(x)}_{\substack{\text{Poisson process}\\ \text{intensity}}} \underbrace{\int_{\mathbb{R}^d} (f(y) - f(x)) \mathcal{Q}(x, dy)}_{\text{Jump}}. \tag{3.1}$$

Now suppose we want to simulate a PDMP with a Markov kernel $\mathcal{Q}$ inducing two fundamentally different actions. That is, assume that there are two other Markov kernels $\mathcal{Q}_1$ and $\mathcal{Q}_2$ and functions p_1, p_2, such that $\forall x \; p_1(x) + p_2(x) = 1$ and

$$\mathcal{Q}(x, \cdot) = p_1(x)\mathcal{Q}_1(x, \cdot) + p_2(x)\mathcal{Q}_2(x, \cdot).$$

That is, if the process is at state x and we want to jump according to the Markov kernel $\mathcal{Q}$, we first sample $i \in \{1, 2\}$ from the discrete distribution $(p_1(x), p_2(x))$ and then jump according to the measure $\mathcal{Q}_i(x, \cdot)$. Now define $\lambda_i(x) := p_i(x)\lambda(x)$ and see what happens to the second term of the infinitesimal generator:

$$\begin{aligned}
&\lambda(x) \int_{\mathbb{R}^d} (f(y) - f(x))\mathcal{Q}(x, dy) \\
=&\lambda(x) \int_{\mathbb{R}^d} (f(y) - f(x))(p_1(x)\mathcal{Q}_1(x, dy) + p_2(x)\mathcal{Q}_2(x, dy)) \\
=&\lambda_1(x) \int_{\mathbb{R}^d} (f(y) - f(x))\mathcal{Q}_1(x, dy) + \lambda_2(x) \int_{\mathbb{R}^d} (f(y) - f(x))\mathcal{Q}_2(x, dy)
\end{aligned}$$

We can then simulate our PDMP started at x as follows:

1. Simulate times T_1 and T_2 from Poisson processes with intensities $\lambda_1(\phi(x,t))$ and $\lambda_2(\phi(x,t))$ respectively.

2. Set $i := \underset{k \in \{1,2\}}{\arg\min}\, T_k$. The resulting time T_i is a jump time of Poisson process with intensity $\lambda(\phi(x,t))$. See Section 2.1.2 for justification.

[4] Note how the explanations for each term in Equation 3.1 reassure us, that choosing the Poisson process simulation, the Markov jump kernel and the deterministic flow as policies in our PDMP framework's design makes perfect sense.

3. Let $x' := \phi(x, T_i)$.

4. Generate x_{new} by sampling from the measure $Q_i(x', \cdot)$. Note that at state x' Q_i will be invoked with probability $\frac{\lambda_i(x')}{\lambda(x')} = p_i(x')$ (determined by the first step), which is exactly what we want.

The above procedure generalises trivially to cases where we have n different Markov kernels instead of just 2.

The opportunity for more efficient simulation arises from the composition method for Poisson process simulation introduced in Section 2.1.2. This technique, however, cannot be applied directly. Even though we can decompose our intensity function as $\lambda(x) = \sum_{i=1}^{n} \lambda_i(x)$, the resulting components are not necessarily independent of each other. Lets illustrate that with specific examples.

Suppose we want to simulate a two dimensional process governed by two Markov kernels Q_1 and Q_2, with associated intensities $\lambda_1(x_1, x_2) = x_1^2$ and $\lambda_2(x_1, x_2) = x_2^2$. While these intensities look like they are independent from one another, suppose that Q_1 simply subtracts 1 from the second component. That is, let

$$Q_1((x_1, x_2), (dx_1, dx_2)) = \delta_{(x_1, x_2 - 1)}(dx_1, dx_2)$$

where δ is the Dirac measure. Note that in this case, if we perform the simulation procedure outlined above, and jump accodring to Q_1, then we cannot reuse the time simulated for the second Poisson process.

A dependency between the two Poisson processes can occur in a more subtle way. Now assume that the Markov kernels Q_1 and Q_2 act only on the first and second coordinates of the process respectively. Even this is not enough to ensure independence properties which would allow us to reuse simulation results. Indeed, suppose that the flow ϕ of our PDMP is given by:

$$\phi((x_1, x_2), t) = (x_1 + t, x_2 + x_1 t). \tag{3.2}$$

Then the Markov kernel Q_1, changing the coordinate x_1, affects the speed of the process in the second coordinate x_2, which in turn affects the simulation outcome of the second Poisson process.

We formalise this idea, by defining a function:

$$\text{Dep}_\phi : \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N}) \tag{3.3}$$

which maps a set of variables (denoted by their indices) to an expanded set of variables, by including the variables which will be afected by ϕ, as a result of a change to a variable in S (obviously, changing a variable in S affects a variable in S, which is why we have $\forall S \in \mathcal{P}(\mathbb{N})\; S \subseteq \text{Dep}_\phi(S)$.) For example, for the flow given in Equation 3.2, we have: $\text{Dep}_\phi(\{1\}) = \{1, 2\}$ and $\text{Dep}_\phi(\{2\}) = \{2\}$.

We are now ready to formulate sufficient conditions for the independence of Poisson processes. Let the generator of our PDMP be given by:

$$\mathcal{A}f(x) = \frac{\mathrm{d}}{dt} f(\phi(x,t))\Big|_{t=0} + \sum_{i=1}^{n} \lambda_i(x) \int_{\mathbb{R}^d} (f(y) - f(x)) Q_i(x, dy). \tag{3.4}$$

Further assume, that for $i = 1, 2, \ldots, n$ we have simulated times T_i distributed as the first events of Poisson processes with intensities $t \mapsto \lambda_i(\phi(x,t))$. Assume that the minimum of these times was T_k so that we jump according to the measure $\mathcal{Q}_k(\phi(x, T_k), \cdot)$. This finishes one iteratrion of our PDMP simulation. Assume that the jump has modified the state space variables from some set S. Then to simulate the next iteration, we can reuse the time T_i (shifted by T_k, of course), if $i \neq k$ and evaluating $\lambda_i(x)$ does not depend on any of the variables from the set $\mathrm{Dep}_\phi(S)$. We would like to conclude this section by drawing the reader's attention to the fact, that the independence between separate components of our PDMPs is not a symmetric relation. To see that, consider a component, whose intensity is simply a constant, but whose associated Markov kernel modifies all state space variables. Such a component does not depend on any of the other components, while all the other components depend on it.

3.4 Implementing efficient policies

Now that we know how to exploit the structure of PDMPs, we can start thinking about the implementation. The main problem we face is in representing different components of our PDMP and calculating dependencies between them, while keeping the policies of our framework as loosely coupled as possible. It is particularly challenging, because we need all the policies to communicate in some way. The Poisson process simulator needs to know the set of variables S modified by the last jump implemented by the Markov kernel policy. It further needs a way to access the Dep_ϕ function, which naturally belongs to the flow policy implementation. Finally, the Markov kernel policy needs to be aware of which Poisson process intensity factors λ_i were responsible for the resulting jump times, in order to invoke the appropriate Markov kernel components $\mathcal{Q}_i$. We solve these problems by introducing a concept of the *dependencies graph*. We can then use this graph to perform the Poisson process simulation by adapting Algorithm 4 (simulation by the composition method) for our particular case.

3.4.1 The dependencies graph

Our graph will consist of three types of nodes (see Figure 3.1 for a graphical representation):

1. *Markov kernel nodes.* Each node of this type represents the Markov kernel component $\mathcal{Q}_i$ and holds the indices of all state space variables, which can be modified by this kernel.

2. *Variable nodes* These nodes only hold the indices of intensity factors λ_i, which depend on the state space variable represented by such a node.

3. *Factor nodes* This is the most complex type of node. It is responsible for the Poisson process time generation associated to the intensity factor of λ_i. It also holds the process state space variable indices, which are needed to evaluate λ_i. Finally, it can optionally provide access for pointwise evaluation of λ_i, if needed.

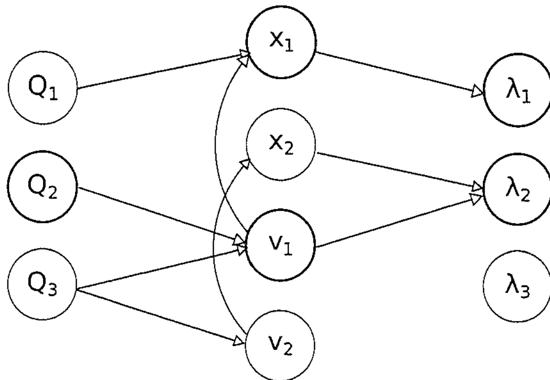

Figure 3.1: A graphical representation of an imaginary dependencies graph for a PDMP on a 4-dimensional position-velocity state space. The arrows represent the dependencies between the nodes. The vertical arrows between the state space variables are induced by Dep_ϕ function (in this case ϕ is the linear flow, whose implementation is given in Listing 3). The red nodes and edges depict the procedure for finding which factors need to be resimulated, after invocation of a specific Markov kernel (which is $\mathcal{Q}_2$ in this image).

The user of our library will be able to specify each part of the process separatelly, by creating markov Kernel and factor nodes, so the modularity of our framework is not lost. Each factor and Markov kernel node can be thought of as a particular (sub)policy class. Our implementation of the main Markov kernel and Poisson process policies will be responsible for managing these subpolicies in a logically valid and efficient way, which will be hidden from the client of our framework.

There are implicit assumptions that the user of our library has to be aware of. We assume that the generator of the process is of the form given in Equation 3.4, so that the number of factor and Markov kernel nodes must be the same (our implementation of the graph performs compile-time checking of this assumption) and the number of variable nodes has to match the dimensionality of the state space. Also, it is assumed that the i-th factor node corresponds to the i-th Markov kernel. Finally, the flow policy interface has to be expanded to support Dep_ϕ calculations. We provide such functionality for the linear flow policy[5] from the Listing 3 (see the project's code repository for details).

We will now describe how to integrate this graph into our PDMPs simulation framework. It is indeed very easy to do: we simply instantiate the Markov kernel and the Poisson process policies, by providing the graph via a constructor argument. Both policies can then decide exactly what they need to take from the graph in order to set up their internal state. In our implementation, the Markov kernel policy simply takes all the Markov kernel nodes, while the Poisson process policy takes the factor nodes and will repeatedly query the graph for the dependencies between the separate Poisson process components.

The interface of the dependencies graph is shown in Listing 5. We apply the policy-based design to the graph's class itself: customisation is allowed by changing node types in the template parameters. There is one last design choice to be consider before implementing the dependencies graph. One way to do the dependencies calculations is to look at the variables modified by the markov Kernel at every iteration of the process and every time solve a small graph problem of finding dependent intensity

[5]A particularly nice feature of C++ class templates is that if a method of some template class is never called, the compiler ignores it so that it does not even appear in the resulting binary. So extending the flow interface does not impose any performance penalties for PDMP simulations which do not require such functionality.

factors. Another option is to find the factors dependent on each Markov kernel (as shown in Figure 3.1) and then cache the results. While the first choice would be great for Markov kernels, whose components act on a random subset of variables, we could actually split up such kernels into even smaller components. Therefore, for efficiency reasons, we decided to implement the second method which relies on caching.

Listing 5 An interface of the dependencies graph.

```
1 template<
2   int N, int StateSpaceDim,
3   class MarkovKernelNodeType, class VariableNodeType, class FactorNodeType>
4 class DependenciesGraph {
5  public:
6
7   using MarkovKernelNodes = array<shared_ptr<MarkovKernelNodeType>, N>;
8   using VariableNodes = array<shared_ptr<VariableNodeType>, StateSpaceDim>;
9   using FactorNodes = array<shared_ptr<FactorNodeType>, N>;
10
11   // Create the graph by simply providing the nodes.
12   DependenciesGraph(
13     const MarkovKernelNodes& markovKernelNodes,
14     const VariableNodes& variableNodes,
15     const FactorNodes& factorNodes);
16
17   // Will be repeatedly called by the main Poisson process policy.
18   template<class Flow>
19   const std::vector<int>& getFactorDependencies(int factorId);
20
21   const MarkovKernelNodes markovKernelNodes;
22   const VariableNodes variableNodes;
23   const FactorNodes factorNodes;
24 };
```

The next thing we need to look at is the implementation of the actual nodes. Our aim is to provide an intuitive syntax for encapsulating the behaviour specified by the user in an appropriate type of node. Implementation of the variable nodes is trivial as they only hold the edges to the factor nodes. We will not show the implementation of factor nodes here, which are quite similar to the Markov kernel nodes but have some additional rather intricate technical details; however, factor nodes can be created by the client by using the same syntactic structure as for the Markov kernel nodes. We show our implementation of the Markov kernel nodes together with a usage example in Listing 6. There is one technical detail that requires attention. Each Markov kernel node will implement a different jump function, as provided by the user. As a result, each node will be of a different type, and thus we cannot store such nodes in any standard library containers. We resolve this problem by creating a common parent class for all markov Kernel nodes which is denoted by MarkovKernelNodeBase in Listing 6. This is the type that will be provided to the dependencies graph as a template argument.

Listing 6 The implementation of Markov kernel nodes.

```
1 template<class State>
2 struct MarkovKernelNodeBase {
3   ~MarkovKernelNodeBase() = default;
4   virtual State jump(const State& state) = 0;
5   const vector<int> dependentVariableIds;
6 };
7
8 template<class State, class JumpFunctor>
9 class MarkovKernelNode : public MarkovKernelNodeBase<State> {
10  public:
11
12   MarkovKernelNode(
13     const vector<int>& dependentVariableIds,
14     const JumpFunctor& jumpFunctor);
15
16   virtual State jump(const State& state) override final {
17     auto stateSubvector = state.getSubvector(this->dependentVariableIds);
18     auto modifiedSubvector = this->jumpFunctor_(stateSubvector);
19     return state.constructStateWithModifiedVariables(
20       this->dependentVariableIds, modifiedSubvector);
21   }
22
23  private:
24   JumpFunctor jumpFunctor_;
25 };
26
27 // The client code can then create the nodes as follows:
28 auto jumpKernel = [] (auto stateSubvector) -> {
29   ... // Do something with stateSubvector.
30   return modifiedStateSubvector;
31 };
32 MarkovKernelNodeBase* = new MarkovKernelNode<decltype(jumpKernel)>(
33     {1, 2}, // Specify the variable indices on which this kernel acts.
34     jumpKernel);
```

3.4.2 Adapting the composition method for jump time simulation

As promised at the beginning of this section, we will use the dependencies graph and an adapted version of simulation by the composition method to implement the main Poisson process policy. Take a quick second look at Algorithm 4. It is easy to see how this algorithm needs to be modified – instead of resimulating one event only at each step, we also resimulate all its dependencies. See Listing 7 for a skech of the implementation, which we will now briefly explain.

We first need to decide on the data structure that we will use for managing a set of future events – in particular, we need to be able to quickly add new elements and quickly extract the smallest element. The C++ standard library provides a priority queue implementation with $O(1)$ time for minimal element extraction and $O(\log N)$ for inserting a new element or removing an arbitrary element. The Boost C++ library [3] provides an implementation of a *Fibonacci heap*, which provides the same operations with complexities $O(1), O(1)$ and $O(\log N)$ respectively. Such an implementation is of particular intereset to us, since we can avoid calling the operation of arbirary elemen removal as we will see in a while. By default, we set our event scheduling data structure to C++ standard library's priority queue. A user might switch to Boost's Fibonacci heap by simply changing the template argument to our Poisson process policy (or for any other implementation. See Devroye [9, Chapter XIV Section 5] for a review of discrete event simulation techniques).

Lets now point out the two improvements that we make over the most naive implementation. Let EventType be some abstract data structure, encapsulating all the information that we associate to a particular simulation time that we want to put into our event scheduler. For each factor node, our Poisson process policy class holds a pointer to the last EventType object provided to the event scheduler. If the last time proposed by some factor node becomes invalid (due to jump induced by a dependenct factor), we can simply modify the state of this the associated EventType data structure through our internal pointer, marking it as invalid without any communication with the event scheduler. Once such an invalid factor gets returned by the event scheduler, we simply ignore it (lines 17-21 in Listing 7).

Another improvement is made in connection to the Poisson process simulation by thinning (as discussed in Algorithm 3). Simulating an event using this procedure might be expensive, if rejections are very common. It becomes a total waste of computational power if such a time proposal gets invalidated due to a jump induced by a dependent factor. To avoid unnecessary calculations, we can instead delay the rejection step to the point when the jump time proposal is the minimum among all the other scheduled times. A skech realisation of this is shown in lines 22-26 of Listing 7.

Finally, take a look at the line 37. We expand our Poisson process policy's public interface by exposing the id of the intensity factor which was the last one to induce a jump. It allows the Markov kernel policy to decide, which jump component needs to be invoked. The Markov kernel policy can call this method (not shown here, see code repository for details) in a similar way to how the Poisson process policy figures out what kind of flow policy is used by the host class (lines 28-29).

We summarise the full design of our PDMPs framework in Figure 3.2.

Listing 7 The Poisson process policy based on the dependencies graph.

```
 1 template<
 2   class DependenciesGraph,
 3   template<class> class EventScheduler = PriorityQueueEventScheduler>
 4 class PoissonProcessPolicy {
 5  public:
 6
 7   // Provide the dependencies graph via a constructor argument.
 8   PoissonProcess(DependenciesGraph graph);
 9
10   // The method which will be invoked by the PDMP host class.
11   template<class State, class HostClass>
12   auto getJumpTime(const State& state, const HostClass& hostClass) {
13     resimulateEventsForFactors(factorsToResimulate_);
14     while (true) {
15       EventType event = eventScheduler_.pop();
16       int factorId = event.factorId;
17       if (!event.isValid) {
18         // This event was invalidated because a new event was
19         // simulated for this factorId.
20         continue;
21       }
22       if (!event.shouldAccept()) {
23         // Event was rejected due to the Poisson process thinning step.
24         resimulateEventForFactor(factorId);
25         continue;
26       }
27       // Found an event that is valid and not rejected.
28       factorsToResimulate_ =
29         graph_.getFactorDependencies<HostClass::FlowPolicy>(factorId);
30       auto iterationTime = event.time - currentTime_;
31       currentTime_ = event.time;
32       return iterationTime;
33     }
34   }
35
36   // This method will be accessed by the Markov kernel policy.
37   int getLastFactorId() const;
38
39  private:
40   EventScheduler<EventType> eventScheduler_;
41   DependenciesGraph graph_;
42   vector<int> factorsToResimulate_;
43   double currentTime_ = 0.0f;
44 };
```

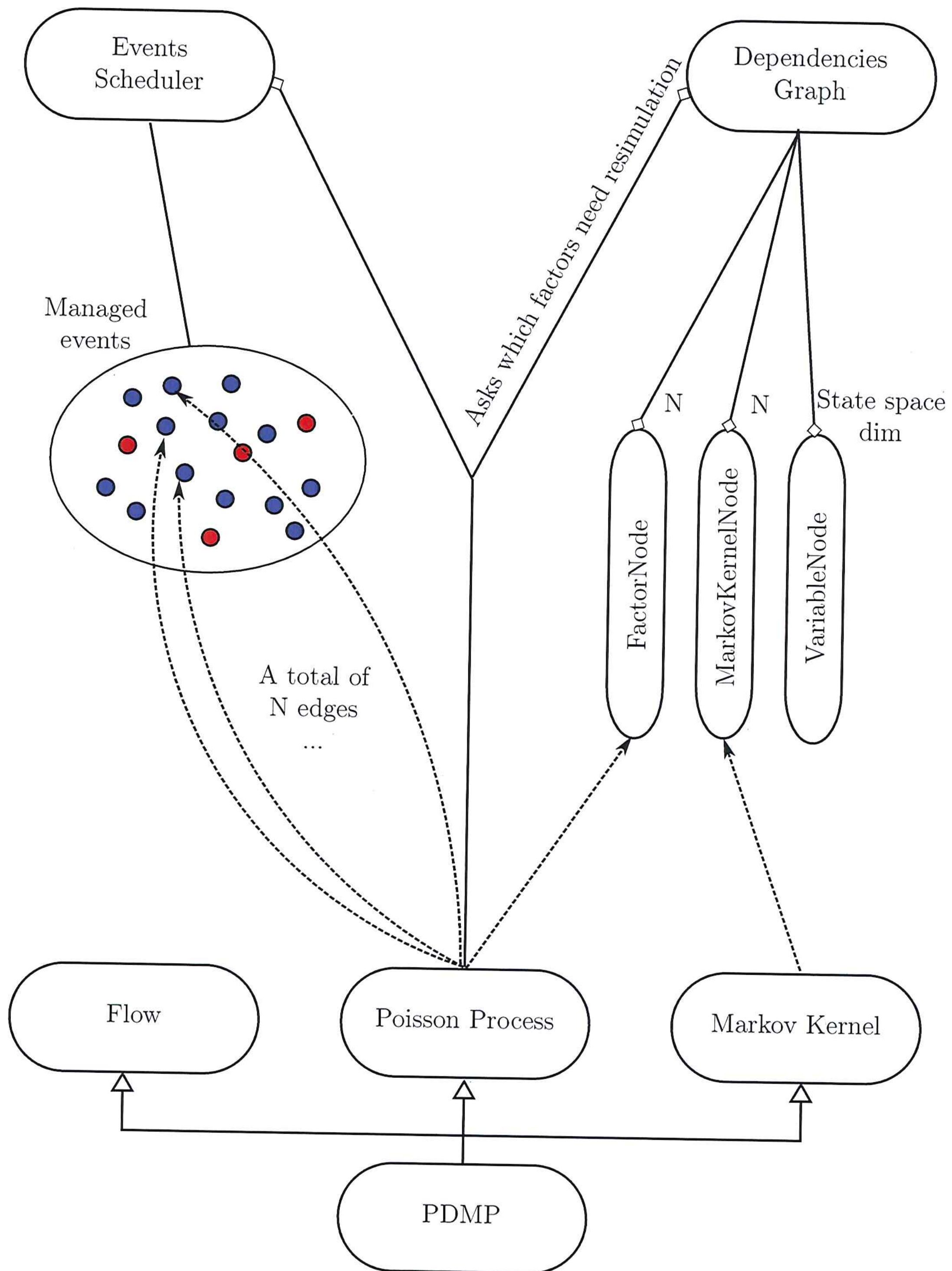

Figure 3.2: A UML diagram depicting the final design of our PDMPs simulation framework. The dashed lines represent pointers, edges ending in a white triangle represent inheritance, edges with a diamond shaped ending represent a has-a relationship. Red circles represent the events not yet removed from the events scheduler, but who have already been invalidated. The user of our framework is only concerned about defining the graph nodes.

3.5 Testing

The importance of writing automated tests for any kind of software cannot be overstated. As the project grows in size, small code changes can have seemingly unexpected consequences. It becomes hard to reason about the correctness of our code and as a result, subtle bugs get introduced to the production code, which can be especially hard to track down later. To help alleviate these problems, we turn to unit and integration testing, which will try to make sure, that any future changes to the code will not break the correctness conditions.

We use Google's googletest [18] framework for writing tests and mocking objects. At first we test each of the individual component discussed in this chapter by replacing their dependencies with mock copies, whose behaviour can be controlled precisely. This idea is at the heart of unit testing – we are able to test the correctness of a specific component regardless of the correctness of its dependencies (and the dependencies of its dependencies for that matter). Next, we write integration tests to see if separate components behave correctly as a unit. An interesting issue arises in our case, because we want to test inherently random processes. The process outcome is hence allowed to be different during different runs, which makes testing harder. We resolve this issue by mocking the random number generators (RNG), then performing a short simulation by hands to carefully craft some test cases (using the mock RNG output, which we set up ourselves) and test if our library produces expected results.

We have attempted to cover as many edge cases as possible by writing tests for the PDMPs framework discussed in this chapter. A total of 66 test cases were written, the source code of which can be found in the tests/core/ directory of the project's GitHub page.

4 Building a Framework for Output Analysis

In the previous chapter we designed and implemented a flexible and efficient library for creating arbitrary PDMPs. Assume that a user of our library wants to test a new MCMC algorithm based on a PDMP. A natural research workflow could then go as follows: first, we build the process using our framework, then, we output the results to a text file and finally, we load them with, for example, R or Python to analyse the output. The shortcomings of such an approach are easy to see – by outputting the simulation results to a text file we lose all the information about the state space representation and what the deterministic flow function of the process is. This information is necessary in order to recreate the complete process trajectory (from the points at event times) so it would have to be reprogrammed in the tool used for analysing the output. Furthermore, due to the inherent randomness of our process, doing statistical analysis requires simulating the same process a number of times. These days, even the cheapest laptop computers have at least 4 logical processors, with the more expensive ones reaching 8. It would therefore be nice, if there was some easy way to use all the cores to run independent processes, turning an experiment that could take a few days into a simple overnight simulation. In this chapter, we will try to address the concerns laid out above by extending our library with support for analysing the output[1]. Once again, our main focus is flexibility – we want out output analysis utilities to be easily extendable and customisable by the client for its special needs.

4.1 Analysis Utilities

Let's first discuss the simplest tools that are necessary to do any kind of analysis on the output of our processes.

1. *Support for running multiple processes in parallel.* This is very simple to do, since our separate processes will be independent from each other and will require no communication. Since the C++11 language standard, support for multithreading was added. We use it to write a simple function, which accepts a set of callable objects (e.g. lambda functions), executes them all in parallel[2] and collects the results.

2. *Support for visual analysis.* Since there is no built-in support for plotting graphs in C++, we use QCustomPlot [12] – an open source library, built on top of Qt [28], for plotting and data visualisation. We added wrappers around this library for easy plotting of line graphs, box plots and our simulated process' sample path (for the 2-dimensional case only).

[1]The code for this chapter can be found at the project's GitHub repository int the src/analysis/ directory.

[2]To avoid unnecessary overhead (e.g. context switching), we of course do not run more threads at a time than the number of available logical processors.

3. *Support for numerical integration.* Recall, that the main motivation behind our framework is MCMC algorithms. As discussed in Chapter A.1, we will want to use the output of our process to compute certain expectations. Since our algorithms output continuous trajectories rather than discrete samples from the target distribution, summation in the strong law of large numbers estimator becomes integration. We hence need tools for efficient one dimensional integraton. As a result, we added the GNU Scientific Library (GSL) [19] – a C library for scientific computing which has an implementation of Gaussian quadrature. We implemented wrappers around the C API for easy and idiomatic C++ use.

4. *Support for timing.* Roughly speaking, one algorithm is better than the other, if it performs the same task faster. Hence it is natural that we need tools to easily time (possibly specific parts only) of our processes. Care must be taken when choosing the timing technique. One possibility is to use the wall-clock time. It is, however, very susceptible to noise due to operating system's activities. Another option is to use the CPU time. It has a downside, however, that an algorithm effectively using many cores and finishing much faster than another algorithm using one core only, may have, for instance, equal CPU times. All the algorithms in our project will run on one core only. We will want to run many independent processes at the same time using separate threads. Hence, we provide a wrapper around a timer measuring CPU time separately for each thread. Clients of our library may implement their own timers and use them with the existing code by simply changing template parameters where appropriate.

4.2 Running the Process

So far, our PDMPs' host class interface only supports simulation of one iteration of the process. Every user of our library thus has to write their own code for running the whole process and deciding when to stop. This is clearly unacceptable, since many people would be writing essentially the same code. To solve this, we implement a separate class capable of running any PDMP[3]. We again reuse the policy-based design, by creating a host class for all runners and delegating the specific running and stopping decisions to its policy class. The host class will be responsible for registration of *output processors*, which we will introduce in the next section. We provide two running policies – one for simply performing a fixed number of iterations, another for stopping the process after the specified amount of time (with extra support for the *burn-in*[4] time and optionally excluding the time spent by the output processors.

[3]The runner class is general enough to run any kind of PDMP, not even necessarily built using our own framework. We could thus wrap any other PDMP implemented in C++ to return results compatible with our framework, and such third-party algorithm could fully benefit from our analysis framework's functionality.

[4]Running a Markov chain for a while and discarding its output. We hope that our process has converged to the invariant distribution during this time.

4.3 Output Processors

This is the most interesting decision we make in the output analysis framework's design. To the best of our knowledge, most of MCMC libraries' APIs only support generating a pre-specified number of samples from the target probability distribution. Suppose we also did that and that we want to generated samples over 10^4 dimensional space using single precision floating point numbers. Then, generating 25000 samples would take at least 1 GB of RAM. Running 8 such chains simultaneously, would require 8 GB of RAM, which is the limit in most current laptop computers at the time. This puts severe limitations on the simulations that we can perform (without outputting samples to a hard drive).

Note, however, that we do not actually need to save all the generated samples. Some of the most useful output analysis statistics can be calculated by simply looping through the generated samples only once. To solve our memory issue we can simply do our calculations while the process is running and discard the old samples which allows our analysis framework to use only $O(1)$ memory. We realise this idea by employing the *Observer pattern* (see Gamma et al. [15]) – we create a base class called *observer* for classes implementing arbitrary calculations on the output of our process. Any class that inherits from the observer can register to our PDMPs runner class. Then, whenever the runner gets a new sample from the PDMP, it notifies all the observers with the obtained new value, thus allowing them to do the calculations online. Note that this design does not put any restrictions on the output analysis as one can implement an observer, which simply stores all the output into an array, or writes the output into a text file, which would allow us to use the standard analysis techniques.

We implement three output processors in our analysis framework – expectation, autocorrelation and asymptotic variance calculators. The last two are quite tricky to implement online and require some compromise. We refer an interested reader to Appendix C for details on what autocorrelation and astymptotic variance is, why it is useful in output analysis of MCMC algorithms and a brief explanation on how we calculate these quantities online. See Figure 4.1 for a graphical representation of the analysis framework discussed in this chapter.

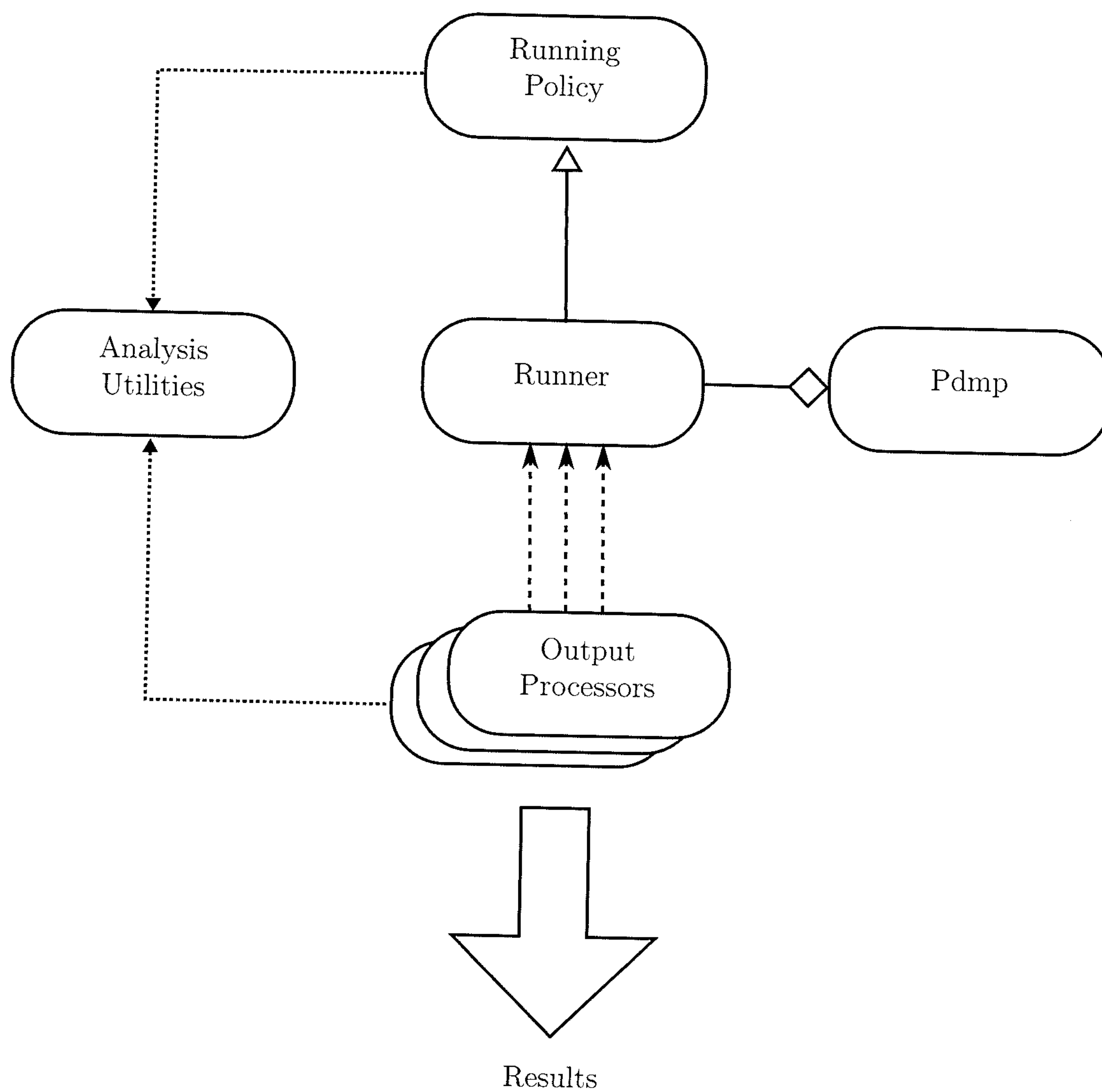

Figure 4.1: A diagram representing the design of our output analysis framework. The output processors of choice have to register to the PDMP runner. The runner notifies all of the processors with latest iteration results of the PDMP which is being simulated. The running policy and output processors can use our library's analysis utilities (e.g. timers).

5 Applications to MCMC Algorithms

In this chapter we show how the current PDMP based MCMC algorithms can be efficiently implemented by using our framework, namely, the Bouncy Particle Sampler [4] and the Zig-Zag process [2]. We further show how our analysis framework can assist in solving the current research problems.

5.1 The Bouncy Particle Sampler

5.1.1 The Global Scheme

Suppose we want to generate samples from some probability distribution π over $\mathbb{R}^d$ which we can evaluate only up to a positive normalising constant, that is:

$$\pi(x) \propto \gamma(x).$$

Further, let

$$U(x) = -\log \gamma(x)$$

and call U the *energy function.*

Now consider a $\mathbb{R}^{2d} = (X, V)$ state space, where the first d variables represent position of a particle, while the last d variables represent velocity (just like in the toy PDMP example introduced in Section 2.3.2). Define the *intensity function* λ on such state space as:

$$\lambda(x, v) = \langle \nabla U(x), v \rangle^+$$

and define the *reflection operation*:

$$R(x)v = \left(I_d - 2\frac{\nabla U(x)(\nabla U(x))^T}{\|\nabla U(x)\|^2} \right) v = v - 2\frac{\langle \nabla U(x), v \rangle}{\|\nabla U(x)\|^2} \nabla U(x)$$

where $\|\cdot\|$ is the L^2 norm. The reflection operator reflects the velocity v on the hyperplane tangent to the energy function gradient at the given position x.

Further, let $\lambda_{\text{ref}} > 0$ be the *refresh rate* and let ψ be a standard normal multivariate d-dimensional probability density. The Bouncy Particle Sampler is then a PDMP, given by the following generator:

$$\begin{aligned}\mathcal{A}f(x, v) &= \langle \nabla_x f(x, v), v \rangle \\ &\quad + \lambda(x, y)(f(x, R(x)v) - f(x, v)) \\ &\quad + \lambda_{\text{ref}} \int (f(x, v') - f(x, v))\psi(v')\, dv'.\end{aligned}$$

The first term is the linear flow, the second term corresponds to reflecting the velocity, while the last term corresponds to simply resampling the velocity. See [27, 4] for justification of such procedure. Every point in the resulting trajectory generated by our PDMP is a sample from the invariant distribution π. The law of large numbers becomes integrating instead of summation. See Figure 5.1 for some samples trajectories generated with different refresh rate parameters.

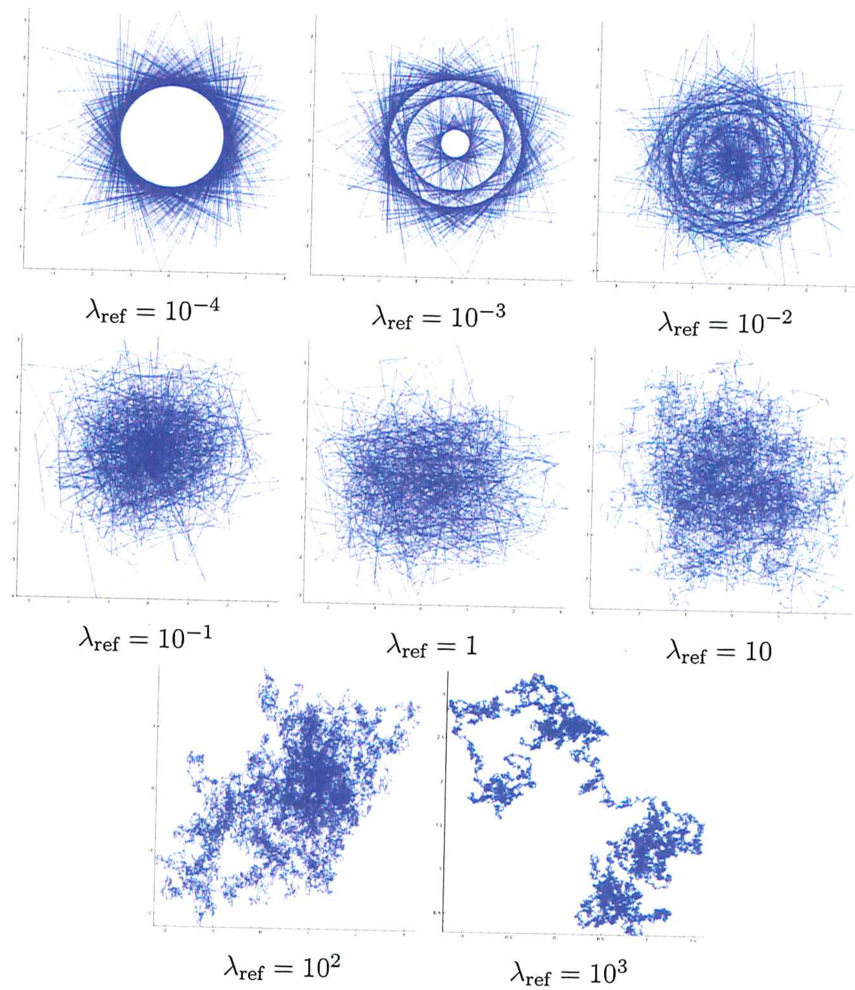

Figure 5.1: The role of the refresh rate parameter in the Bouncy Particle Sampler. In each image, a trajectory path of the BPS is plotted, where the target distribution is 2-dimensional standard normal. Both too small and too large values of λ_{ref} lead to difficulties in exploration of the state space. Indeed, it was shown in Bouchard-Côté et al. [4] that with $\lambda_{\text{ref}} = 0$ the resulting Markov process need not be ergodic. See Figure 5.4 which shows how our framework can assist in searching for an optimal value of the refresh rate. See examples/path_plotting/ in the project's GitHub repository for reproducing these plots by using our analysis framework.

5.1.2 The Local Scheme

Now supose that

$$\gamma(x) = \prod_{f \in F} \gamma_f(x_f).$$

Then the energy function writes as a sum of (local) energies. Consequently, we can associate a (local) intensity factor λ_f and a (local) reflection operator R_f with each factor $f \in F$ in the same way as we did in the global BPS scheme above. The local Bouncy Particle Sampler is then a PDMP with infinitesimal generator given by:

$$\begin{aligned}\mathcal{A}h(x,v) &= \langle \nabla_x h(x,v), v \rangle \\ &+ \sum_{f \in F} \lambda_f(x,y)(h(x, R_f(x)v) - h(x,v)) \\ &+ \lambda_{\text{ref}} \int (h(x,v') - h(x,v))\psi(v')\,dv'.\end{aligned}$$

See [4] for full details. This local BPS scheme is extremely useful in scenarios where we have a lot of factors and the dependencies between them are very sparse.

5.1.3 Implementation

The good news is that there is not really much that we need to implement. Indeed, in Chapter 3 we have already efficiently implemented processes of such type. The Poisson process simulation is model dependent and thus will have to be specified by the user. So the only thing we need to do is implement the reflection operator, which is easy to do.

To take away the burden of gradient calculations and basic probability densities definitions we have added the Stan mathematics library developed by Carpenter et al. [6] to our project, which was built for almost identical use case as ours (Hamiltonian Monte Carlo simulations).

Finally, we provided a builder class for creating the BPS algorithm, which simply takes our specificaly designed probability distribution class as input for adding factors to the generator. The probability distribution objects encapsulate how to simulate Poisson process jump times associated to them, while the BPS builder class in addition adds reflection kernels as well as the refreshment kernel. See project's code repository for full implementation details. Listing 8 a code example of using the builder and our probability distribution classes.

5.2 The Zig-Zag Process

The Zig-Zag process is similar to the BPS, except that instead of performing velocity reflections, during each event time ir multiplies one velocity component by -1 (flips that component). See Bierkens et al. [2] for details. This essentially means that we can reuse almost all the code used to construct the BPS PDMPs – we only need to change the reflection kernel with the flipping kernel. See the project's code repository

Listing 8 Example usage of the BPS algorithm within our framework.

```
1 // Create a standard normal 2-dimensional distribution object.
2 RealMatrix covariances(2, 2);
3 covariances << 1, 0,
4                0, 1;
5 RealVector mean(2);
6 mean << 0, 0;
7 mcmc::GaussianDistribution gaussianDistribution(mean, covariances);
8
9 // Create a PDMP simulating the BPS algorithm.
10 pdmp::mcmc::BpsBuilder bpsBuilder(2);
11 bpsBuilder.addFactor(
12   {0,1}, // Specify the model variables on which this factor acts.
13   gaussianDistribution);
14 // Add more factors if needed...
15 auto pdmp = bpsBuilder.build();
16
17 // Now select a PDMP runner and output processors from the analysis
18 // framework to perform arbitrary actions as needed.
```

for implementation details. See Figure 5.2 for an example trajectory generated by the Zig-Zag process. See Figure 5.3 for how our framework can be used to compare the BPS and the Zig-Zag process.

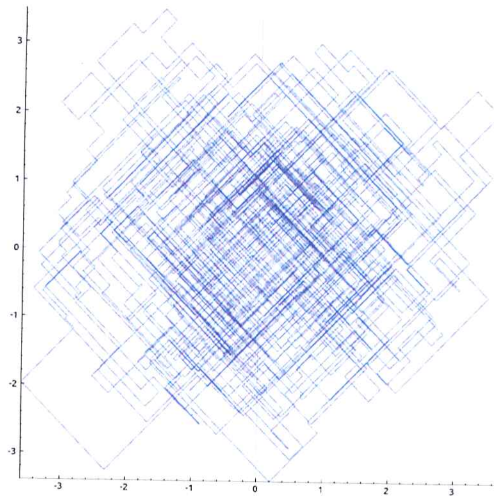

Figure 5.2: A sample path generated by the Zig-Zag process on the bivariate standard normal distribution as its target. Note that at the event times only one velocity component is changed. See /examples/path_plotting/ in the project's GitHub repository for code.

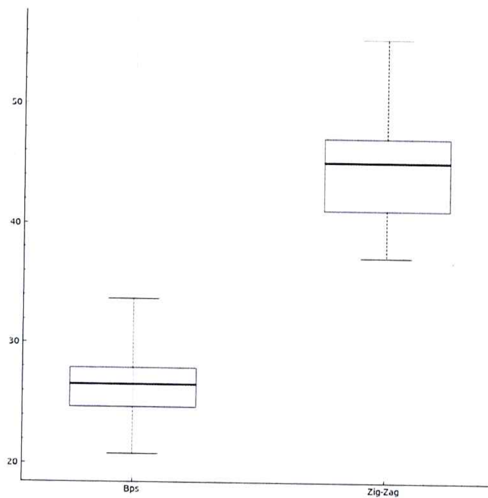

Figure 5.3: Comparison of asymptotic variances (see Appendix C) of the BPS and the Zig-Zag process on a two-dimensional standard Gaussian target, where the function we integrate along the path is the squared L^2-norm. Each algorithm was run 42 times to generate the box plot. We have used our analysis framework's to run many processes in parallel. See /examples/compare_bps_zig_zag/ directory in the project's GitHub page for the code used to run this experiment. Note that no parameters were tuned for either of the algorithms, this example is merely meant to demonstrate the capabilities of our framework. Listing 9 shows the sketch of code used efficiently generate such graphs.

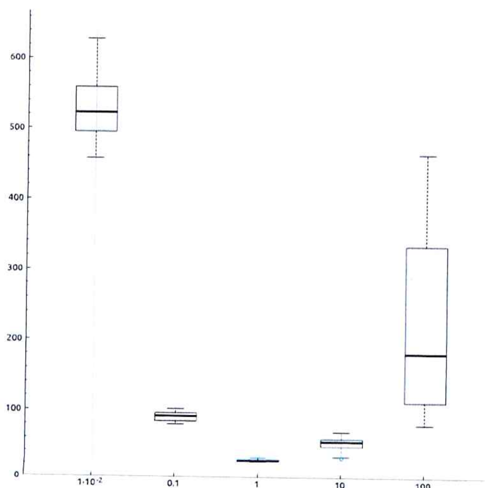

Figure 5.4: Comparison of asymptotic variances (see Appendix C) of the BPS algorithm on a two-dimensional standard Gaussian target, where the function we integrate along the path is the squared L^2-norm. This graph suggests that the best refresh rate for such target distribution is approximately 1. See /examples/bps_refresh_rate/ for code that can reproduce this plot.

Listing 9 Sketch code for comparing different algorithms just as we did in Figure 5.3

```
1 double getBpsAsymptoticVariance() {
2   // Create a new BPS process and use our output analysis framework
3   // to run it and calculate the asymptotic variance.
4   ...
5   return calculatedAsymptoticVarianceForThisRun;
6 }
7
8 double getZigZagAsymptoticVariance() {
9   // Same as above, but for the Zig-Zag process.
10  ...
11  return calculatedAsymptoticVarianceForThisRun;
12 }
13
14 int main() {
15   ParallelWorkers<double> bpsWorkers, zigZagWorkers;
16   // Launch 40 BPS processes on 8 cores.
17   auto bpsAsymptoticVariances = bpsWorkers.executeTasksInParallel(
18     getBpsAsymptoticVariance, 40, 8);
19   // Launch 40 Zig-Zag processes on 8 cores.
20   auto zigZagAsymptoticVariances = zigZagWorkers.executeTasksInParallel(
21     getZigZagAsymptoticVariance, 40, 8);
22
23   plotBoxPlot(
24     {bpsAsymptoticVariances, zigZagAsymptoticVariances},
25     {"Bps", "Zig-Zag"});
26
27   return 0;
28 }
```

6 Conclusion

In Section 1.2 we have set the goal to build an *efficient*, *flexible*, *reliable*, *relevant* and *easy to use* framework for PDMPs simulation which will hopefully speed up future research on MCMC methods. We will now briefly review how each of these 5 qualities were achieved:

1. We addressed *efficiency* of our framework in Chapter 3 by choosing the right tools and design patterns.
2. The *flexibility* of our process stems from the policy-based design and hierarchical structure ranging from low level to higher level interfaces. A user of our library can easily change any component of our framework.
3. *Reliability* was addressed by adding a carefully crafted autamated tests suite (see Section 3.5).
4. We made sure our framework is *relevant* to MCMC researchers by complementing it with MCMC output analysis tools described in Chapter 4 and Appendix C. We further showed in Chapter 5 how our framework can help to solve some of the current research problems (Figure 5.3 and Figure 5.4).
5. Finally, in Chapter 5 we demonstrated that the current PDMP based Monte Carlo algorithms are *very easy to implement* by using our library.

We are by no means trying to suggest that our framework is perfect. There is indeed a lot of more work to be done. For example, an abstract layer of standard benchmarking problems could be added (i.e., simulating from Gaussian distributions or solving a Bayesian logistic regression problem). Newly developed algorithms would then imediatelly get access to these problems. As a result, no additional code at all would have to be written to compare it against the other implemented algorithms (like the BPS or the Zig-Zag process).

Recall that one of the best features in our output analysis framework is the ability to implement output processors which do the calculations online. While due to memory limitations discussed in Section 4.3 it is necessary to do analysis online for extremely long runs, it is, however, not necessary for short simulations. In such a scenario analysing the output using a language like C++ is clearly undesirable. This problem could be completely solved by implementing an R interface to our framework.

6.1 Reflection on the Development Process

At the begining of the project I was accustomed to solving every programming problem I had by using inheritance. Being new to C++, I did not initially realise how complex this language is and how many different ways of doing the same things it offers. The implementation of our framework, as outlined in Chapter 3, was already my second attempt. My first purely object-oriented design resulted in too deep class

hierarchies and in the end reusing code was hard. This is when I came accross the book by Alexandrescu [1] where the policy-based design was introduced, which seems to be the perfect fit for our problem.

Writing a big test suite also played an important role in the development process. It allowed us to catch many bugs, which would have otherwise been hard to track down. Also, C++ compilers ignore template class methods that are never called. To avoid surprises, it is important to have some code which calls all of our implemented methods, to make sure that all of our framework compiles.

Finally, this project has served me as a great introduction to the Monte Carlo methods. Designing the framework and thinking about code reuse helped me to better understand the underlying theory. I hence hope that this report can also serve as an accessible introduction to PDMPs to a non-expert in the field.

It would be nice, however, if we did not have to worry about defining the dependencies graph (from Chapter 3) nodes ourselves and only think in terms of the factors that we add to the generator. We thus implement a higher level *builder* class, which takes a *distribution* as input and adds a relevant term to the generator of our process. Also, we add Stan's mathematics library, which implements reverse-mode automatic differentiation and many common probability distribution functions so that we need not to worry about these details.

A Background Material on MCMC Methods

A.1 Motivation for Using Monte Carlo Simulations

In many scientific problems one has to compute an integral over some high-dimensional space. Finding a closed form representation is usually infeasible and numerical approximation techniques have to be used. As the dimensionality of the integral increases, most deterministic numerical integration techniques need to evaluate the integrand at exponentially more points thus making such algorithms computationally intractable. We will now briefly show how such integrals appear in Bayesian inference (which is the main motivation for this project) and how Monte Carlo techniques can be used to approximate them.

In Bayesian statistics (see Gelman et al. [17] for a great introduction), a probability model is given by a joint probability distribution over the unobservable parameters of interest $\theta \in \Theta$ (where the dimensionality of Θ is defined by our model) and observed data x. We define the model in terms of the *likelihood function* $p(x|\theta)$ and the *prior distribution* over the parameters $p(\theta)$:

$$p(\theta, x) = p(x|\theta)p(\theta).$$

By observing the dataset x we gain additional insight into the distribution of parameters θ, which is summarised in the *posterior distribution* $p(\theta|x)$. We can calculate the posterior distribution by applying Bayes' theorem:

$$p(\theta|x) = \frac{p(\theta, x)}{p(x)} = \frac{p(x|\theta)p(\theta)}{p(x)}.$$

Calculating $p(x)$ is also infeasible, except for the simplest models, and hence we usually only have access to an unnormalised posterior density:

$$p(\theta|x) \propto p(x|\theta)p(\theta).$$

We can express our quantities of interest in terms of expectations of the form:

$$\mathbb{E}_{\theta\sim p(\theta|x)}[f(\theta)] = \int_{\Theta} f(\theta)p(\theta|x)\, d\theta.$$

Now suppose we can draw random samples $\theta^{(1)}, \theta^{(2)}, \ldots$ from the posterior distribution $p(\theta|x)$. Then, using the *strong law of large numbers* we can estimate the above integral by $S_n := \frac{1}{n}\sum_{i=1}^{n} f(\theta^{(i)})$. The *central limit theorem* then tells us that

$$\sqrt{n}(S_n - \mathbb{E}[\theta \sim p(\theta|x)]\, f(\theta)) \xrightarrow{d} N(0, \sigma^2),$$

where $N(0,\sigma^2)$ is a centred Gaussian random variable, $\sigma^2 := \mathrm{Var}_{\theta\sim p(\theta|x)}[f(\theta)]$ and $\xrightarrow{d}$ denotes the convergence in distribution. Hence the asymptotic error rate $O(n^{-\frac{1}{2}})$

does not depend on the dimensionality of Θ, which is the main advantage of the Monte Carlo techniques for approximating integrals. However, it is not fair to say that Monte Carlo algorithms beat the curse of dimensionality – drawing random samples from arbitrary distributions can be hard and the resulting variance of our estimator can also be unreasonably high.

A.2 The Metropolis-Hastings Algorithm

In this section we will introduce on of the most important Markov chain Monte Carlo algorithms – the Metropolis-Hastings algorith (see Chapter 1.1 for a brief history). We mainly refer to Liu [22] for presenting the following material.

Suppose we want to generate samples from a given probability density function[1] π defined over $\mathbb{R}^d$. In most realistic cases we can only pointwise evaluate some other function γ which satisfies $\gamma(x) \propto \pi(x)$. The normalising constant $\int_{\mathbb{R}^d} \gamma(x)\, dx$ is usually unknown and impractical to compute. In traditional Markov chain analysis, one's goal is often to calculate the stationary distribution once the Markov transition kernel is given. In contrast, the idea of the Metropolis-Hastings algorithm is to simulate a Markov chain using a carefully chosen transition kernel, so that the resulting invariant distribution is π. We will now explain this algorithm (the pseudocode is given in Algorithm 5) and prove its correctness.

Let $\{q(\cdot\,|x) \mid x \in \mathbb{R}^d\}$ be a family of probability density functions satisfying $\forall x, y \in \mathbb{R}^d \; q(x|y) > 0 \Leftrightarrow q(y|x) > 0$. Assume that we can easily generate random samples from these densities. Once our Markov chain is at state x, we propose a new state $y \sim q(\cdot\,|x)$ which is accepted with probability $r(x, y)$. The acceptance rate function r is chosen in such a way as to ensure that the invariant distribution of our Markov chain is π.

In the original paper by Metropolis et al. [25] only symmetric proposal densities were considered in the sense that $\forall x, y \in \mathbb{R}^d \; q(x|y) = q(y|x)$ and it was suggested to use:

$$r(x, y) = \min\left\{1, \frac{\gamma(y)}{\gamma(x)}\right\} = \min\left\{1, \frac{\pi(y)}{\pi(x)}\right\}. \tag{A.1}$$

It is easy to explain this acceptance rate function – if we propose a new state, which moves to a region with higer probability density with respect to π, we accept that state with probability 1. Otherwise, we reject the proposed state with some positive probability, which increases as the proposed state gets "worse" relative to the current state. However, if the proposal densities are not symmetric, the invariant distribution of the Markov chain generated using such procedure need not be π. Indeed, suppose that all proposal densities are strongly biased towards some small but very high probability region with respect to π (e.g. if π is a standard Gaussian distribution, while all proposal distributions are centered Gaussian with marginal variances very close to 0). Then nearly all proposals will be concentrated very close to the origin

[1] More precisely, we want to draw samples from some probability measure defined on a Borel σ-algebra $\mathcal{B}(\mathbb{R}^d)$. We assume that this probability measure is absolutely continuous with respect to the Lebesgue measure, so that the probability density function π exists. We will be implicitly making such an assumption throughout the whole text.

and will be accepted with a very high probability. As a result, our Markov chain's invariant distribution will overrepresent that high probability region and thus will be different from π. This issue was addressed by Hastings [21] where the following acceptance rate function was suggested:

$$r(x,y) = \min\left\{1, \frac{\gamma(y)q(x|y)}{\gamma(x)q(y|x)}\right\} = \min\left\{1, \frac{\pi(y)q(x|y)}{\pi(x)q(y|x)}\right\}. \tag{A.2}$$

The new term $q(x|y)/q(y|x)$ compensates for the flow bias introduced by the family of our proposal distributions.

A formal proof showing that the acceptance rate function given by Equation A.2 guarantees π to be the invariant distribution of our simulated Markov chain is surprisingly simple. Let $T(x, \cdot\,)$ be the transition density function once our Markov chain is at state x. For $x \neq y$, we make a transition from x to y if we propose to move to state y and also accept this move. Hence for $x \neq y$ we have $T(x,y) = q(y|x)r(x,y)$ and thus:

$$\begin{aligned}\pi(x)T(x,y) &= \pi(x)q(y|x)\min\left\{1, \frac{\pi(y)q(x|y)}{\pi(x)q(y|x)}\right\}\\ &= \min\left\{\pi(x)q(y|x), \pi(y)q(x|y)\right\}.\end{aligned}$$

Since the right hand side is symmetric in x and y it follows that the *detailed balance* equations hold:

$$\forall x,y \in \mathbb{R}^d \; \pi(x)T(x,y) = \pi(y)T(y,x).$$

We further deduce that

$$\int \pi(x)T(x,y)\,dx = \int \pi(y)T(y,x)\,dx = \pi(y)\int T(y,x)\,dx = \pi(y),$$

which proves that π is an invariant distribution of our *reversible* Markov chain. Showing that the resulting Markov chain is also ergodic (i.e., a Markov chain for which the law of large numbers holds) is somewhat more complicated and we refer an interested reader to Robert and Casella [29, Chapter 6].

An obvious difficulty of the Metropolis-Hastings algorithm is choosing the family of proposal distributions – a choice which does not capture the main properties of the target distribution π will lead to very slow exploration of the state space. It is also noteworthy to mention that unlike in ordinary Monte Carlo techniques, the samples generated by MCMC algorithms are correlated, which makes output analysis harder. We will discuss how the efficiency of correlated samples can be calculated once we introduce our MCMC algorithms analysis framework.

Algorithm 5 The Metropolis-Hastings algorithm

Initialize x_0 arbitrarily on the support of target distribution π
for $i = 1, 2, \ldots$ **do**
  Generate $y \sim q(\cdot\,|x_{i-1})$
  Generate $U \sim \text{Uniform}(0,1)$
  if $U \leq \min\left\{1, \frac{\gamma(y)q(x|y)}{\gamma(x)q(y|x)}\right\}$ **then**
    $x_i \leftarrow y$
  else
    $x_i \leftarrow x_{i-1}$
  end if
end for

B The Toy Example Poisson Process Time Simulation

We will derive exact formulas for simulating the Poisson process times for the toy PDMP from Section 2.3.2. Let the state space be $Z = \mathbb{R}^{2d} = \mathbb{R}^d \times \mathbb{R}^d = (X, Y)$. We have the flow function given by $\phi((x,v),t) = (x + vt, v)$ and the intensity function given by $\lambda(x,v) = \max\{0, \langle x, v\rangle\}$ We give all the credit to Bouchard-Côté et al. [4] for the derivation of the following results.

Note that we can write (with $\|\cdot\|$ denoting the L^2 norm):

$$\begin{aligned}\lambda(\phi((x,v),t)) &= \lambda(x+vt, v) \\ &= \langle x + vt, v\rangle^+ \\ &= \left\langle \nabla \frac{\|x+vt\|^2}{2}, \frac{\mathrm{d}}{\mathrm{d}t}(x+vt)\right\rangle^+ \\ &= \left(\frac{\mathrm{d}}{\mathrm{d}t}\frac{\|x+vt\|^2}{2}\right)^+.\end{aligned}$$

Now note that $\|\cdot\|$ is a strictly convex function. It thus follows, that for a given $(x,v) \in \mathbb{R}^{2d}$ $\exists \tau^* \geq 0$, such that $\frac{\mathrm{d}}{\mathrm{d}t}\|x+vt\|^2$ is negative for $t \in (0, \tau^*)$ and positive for $t \in (\tau^*, \infty)$. We can calculate the value of τ^* exactly, by solving $\frac{\mathrm{d}}{\mathrm{d}t}\|x+vt\|^2 = 0$ for t (and setting $\tau^* = 0$ if the solution is negative). One can easily verify that:

$$\tau^* = \left(-\frac{\langle x, v\rangle}{\|v\|^2}\right)^+. \tag{B.1}$$

Now lets see how we can use Algorithm 2 (the time scale transformation algorithm) for simulating jump times of our PDMP. We first simulate $U \sim \text{Uniform}(0,1)$ and try to find the smallest $\tau \geq 0$ such that

$$\int_0^\tau \lambda(\phi((x,v),t))\,dt = \int_0^\tau \left(\frac{\mathrm{d}}{\mathrm{d}t}\frac{\|x+vt\|^2}{2}\right)^+ dt = -\log U.$$

Now note that we can rewrite:

$$\begin{aligned}\int_0^\tau \left(\frac{\mathrm{d}}{\mathrm{d}t}\frac{\|x+vt\|^2}{2}\right)^+ dt &= \int_0^{\tau^*} 0\,dt + \int_{\tau^*}^\tau \frac{\mathrm{d}}{\mathrm{d}t}\frac{\|x+vt\|^2}{2}\,dt \\ &= \int_{\tau^*}^\tau \frac{\mathrm{d}}{\mathrm{d}t}\frac{\|x+vt\|^2}{2}\,dt \\ &= \frac{\|x+v\tau\|^2}{2} - \frac{\|x+v\tau^*\|^2}{2}\end{aligned}$$

Hence, to get the jump time we need to solve the following equation for τ:

$$\frac{||x + v\tau||^2}{2} - \frac{||x + v\tau^*||^2}{2} = -\log U$$

If $\tau^* = 0$, the equation becomes:

$$||v||^2\tau^2 + 2\langle x, v\rangle\tau + 2\log U = 0$$

and solving for the positive root, we obtain:

$$\tau = \frac{1}{||v||^2}\left(-\langle x, v\rangle + \sqrt{\langle x, v\rangle^2 - 2||v||^2 \log U}\right).$$

If $\tau^* > 0$, ten by using Equation B.1 we get the following quadratic equation:

$$||v||^2\tau^2 + 2\langle x, v\rangle\tau + 2\log U - \frac{\langle x, v\rangle^2}{||v||^2} = 0$$

with the positive root giving:

$$\tau = \frac{1}{||v||^2}\left(-\langle x, v\rangle + \sqrt{-2||v||^2 \log U}\right).$$

This concludes the derivation of equations for the jump time simulation of the toy PDMP from Section 2.3.2.

C On MCMC Output Analysis

In this chapter we present the basic theory behind the tools implemented in our output analysis framework that can be used to objectively compare different MCMC algorithms. We refer to Brooks et al. [5, Chapter 1] and Liu [22, Chapter 5], where more thorough treatment of the following material can be found.

Suppose we have generated n random samples $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$ using some MCMC sampler with invariant distribution π and further assume that $x^{(1)} \sim \pi$ (usually obtained by throwing away some initial part of the simulation, waiting for the Markov chain to converge to π). Then, for an arbirary function f, we can estimate $\mathbb{E}_\pi[f(X)]$ by:

$$\hat{\mu}_n := \frac{1}{n}\sum_{i=1}^{n} f(x^{(i)}).$$

Had our samples been independent, we would have $\mathrm{Var}[\hat{\mu}_n] = \frac{1}{n}\mathrm{Var}_\pi[f(X)]$. Unfortunatelly, samples drawn from a Markov chain are generally not independent. Under certain regularity conditions, the Markov chain CLT tells us that as $n \to \infty$:

$$\mathrm{Var}[\hat{\mu}_n] = \frac{\sigma_f^2}{n} = \frac{1}{n}(\mathrm{Var}_\pi[f(X)] + 2\sum_{k=1}^{\infty}\mathrm{Cov}[f(X_1), f(X_{1+k})]) \tag{C.1}$$

where $X_1, X_2, \ldots$ is the Markov chain (with $X_1 \sim \pi$) from which we generate the samples. The term denoted by σ_f^2 in Equation C.1 is called the *asymptotic variance.* Essentially, if we want to compare two MCMC algorithms which generate samples from the same distribution at the same speed, the one with lower asymptotic variance is better since it provides a more reliable expectation estimator.

C.1 Autocorrelation Function

Let $\rho_k := \mathrm{Corr}[X_1, X_{1+k}]$ and call it *lag-k autocorrelation.* We can then rewrite Equation C.1 as follows:

$$\mathrm{Var}[\hat{\mu}_n] = \frac{\mathrm{Var}_\pi[f(X)]}{n}(1 + 2\sum_{k=1}^{\infty}\rho_k). \tag{C.2}$$

Hence, to compare two MCMC algorithms generating samples from the same distribution we can simply plot their autocorrelation functions, which act as a proxy for estimating the asymptotic variance. See Figure C.1 for an example.

However, such an analysis does not take the speed of the samplers into account. To see why it is problematic, consider the following scenario: samples generated using one MCMC algorithm may possess bigger correlation than samples produced by another algorithm. However, assume that the first algorithm generates the samples much faster. Then it may still be preferable to use the first algorithm, which could produce better expecatation estimates within a fixed time budget. This issue is solved by comparing how many effective samples each algorithm produces per time unit. See the next section for details.

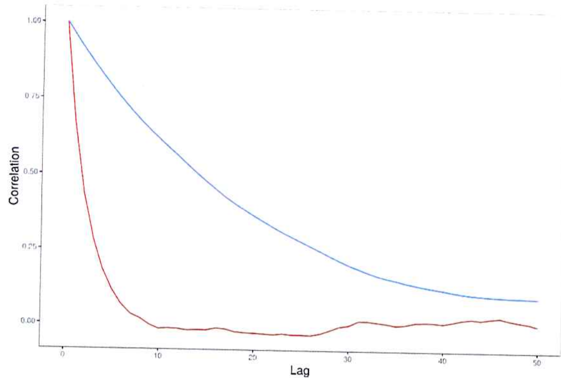

Figure C.1: An example of autocorrelation plots. We simulate two Markov chains to generate samples from $N(0,1)$ distribution according to the Metropolis scheme described in Equation A.1, with proposal densities $q(\cdot|x) = N(x, \sigma^2)$. The red curve is the autocorrelation function of the samples produces by such a MCMC algorithm using $\sigma^2 = 1$. The blue curve was generated using $\sigma^2 = 0.1$. It is easy to see from the grap, that setting σ^2 to 0.1 results in too slow exploration of the state space.

Finally, we want to briefly discuss how to calculate the autocorrelation function. Essentially, what we want to do is estimate ρ_k for any given value of k (up to some upper bound for which we want to generate our plots). Recall that:

$$\rho_k = \frac{\mathrm{Cov}[f(X_1), f(X_{1+k})]}{\mathrm{Var}_\pi[f(X)]}$$

so after generating n samples from our MCMC algorithm, we can estimate ρ_k by computing:

$$\sum_{i=1}^{n-k} \frac{(x^{(i)} - \hat{\mu}_n)(x^{(i+k)} - \hat{\mu}_n)}{\hat{\sigma}_n^2} \tag{C.3}$$

where we estimate $\hat{\sigma}_n^2$ from our MCMC output simply in the same way as we estimate any other expectation.

Recall that in our framework we implement an output processor which calculates autocorrelation statistics online. We face one difficulty, however, because we cannot reliably estimate $\hat{\mu}_n$ and $\hat{\sigma}_n^2$ at the beginning of the simulation. Even though this is not ideal, to solve this problem we simply start estimating the correlations after our output processor has obtained a pre-specified number of samples (e.g. a trajectory of length 1000 was already generated). We can then use these samples to get initial estimates of $\hat{\mu}_n$ and $\hat{\sigma}_n^2$ and start using the estimator from Equation C.3. As the simulation progresses, we keep updating the values of $\hat{\mu}_n$ and $\hat{\sigma}_n^2$ so that we get better and better estimates with the later samples. Such procedure results in an online computation of autocorrelation function at the cost of some extra noise.

C.2 Effective Sample Size

As we have briefly mentioned already, we would like to have some way to compare MCMC algorithms which also takes the running time into account. Define the *integrated autocorrelation time* (IACT) as:

$$\tau_{\text{int}} := 1 + 2\sum_{k=1}^{\infty} \rho_k.$$

Now note that by Equation C.2, having $n\tau_{\text{int}}$ samples from our MCMC algorithm has the same effect (in terms of variance of our expectation estimator) as having n

independent samples from our target distribution π. Hence, we define the *effective sample size* of n samples obtained from our simulated Markov chain by the following equation:

$$n_{\text{eff}} := \frac{n}{\tau_{\text{int}}}.$$

To compare the performance of different MCMC algorithms we can then run them for the same amount of time and compare the number of effective samples produced by each algorithm.

A question remains, however, of how to compute the τ_{int} which contains an infinite sum in its expression. Note that

$$\tau_{\text{int}} = \frac{\sigma_f^2}{\text{Var}_\pi[f(X)]}$$

where σ_f^2 is the asymptotic variance. Estimating $\text{Var}_\pi[f(X)]$ is trivial from the samples of our MCMC algorithm output, so we only need to worry about σ_f^2. We will now describe how to estimate it using a method called *batch means*[1].

Suppose we have drawn samples $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$ from our MCMC sampler. Now assume we take a contiguous segment of samples $x^{(l)}, \ldots, x^{(m)}$ and for simplicity denote $k = m - l + 1$. Then assuming that k is large enough, from the Markov chain CLT we have:

$$\frac{1}{\sqrt{k}} \sum_{i=l}^{m} f(x^{(i)}) - \sqrt{k}\mu \overset{\text{d}}{\approx} N(0, \sigma_f^2)$$

where μ denotes $\mathbb{E}_\pi[f(X)]$ and $\overset{\text{d}}{\approx}$ says that the expression on the left is approximately distributed as the random variable on the right. We can hence partition $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$ into a number of batches from which we can obtain samples from $N(0, \sigma_f^2)$, after which estimating σ_f^2 is trivial. Choosing batch sizes requires trade-offs. If we partition our samples into too many batches, the number of samples in each batch may be too small for the Markov chain CLT to hold. On the other hand, too few batches may result in too noisy estimate of σ_f^2. It is argued in Flegal et al. [14] that taking $\sqrt{n}$ batches of equal length is often a good choice.

In our MCMC output analysis framework, we provide an output processor which computes the asymptotic variance using the technique described above. The only difficulty in doing the calculations online is that we do not know what the resulting trajectory length of the process will be so we cannot choose the batch lengths in advance. We thus need to maintain batches in real time, creating new ones and joining the old ones as the process evolves. Our implementation maintains the following invariant: at any point of the process where the total trajectory length generated is T, we maintain $n \in [\frac{\sqrt{T}}{2}, \sqrt{T}]$ batches, where each bath length is in $[\sqrt{T}, 2\sqrt{T}]$. We refer readers interested in full implementation details to the project's code repository (see the src/analysis/output_processors/ directory).

[1] For ease of presentation, we will describe the discrete case version. For continuous time Markov chains we essentially just replace the summation with integration. See Bierkens et al. [2] for a detailed description of the batch means method for continuous time MCMC algorithms.

References

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[2] J. Bierkens, P. Fearnhead, and G. Roberts. The Zig-Zag Process and Super-Efficient Sampling for Bayesian Analysis of Big Data. *arXiv preprint arXiv:1607.03188*, 2016.

[3] Boost. Boost C++ Libraries. `http://www.boost.org/`, 2017.

[4] A. Bouchard-Côté, S. J. Vollmer, and A. Doucet. The Bouncy Particle Sampler: a Non-Reversible Rejection-Free Markov Chain Monte Carlo Method. *arXiv preprint arXiv:1510.02451*, 2015.

[5] S. Brooks, A. Gelman, G. L. Jones, and X.-L. Meng. *Handbook of Markov Chain Monte Carlo*. Chapman and Hall/CRC, 2011.

[6] B. Carpenter, M. D. Hoffman, M. Brubaker, D. Lee, P. Li, and M. Betancourt. The stan math library: Reverse-mode automatic differentiation in c++. *arXiv preprint arXiv:1509.07164*, 2015.

[7] M. H. Davis. Piecewise-deterministic Markov Processes: A General Class of Non-diffusion Stochastic Models. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 353–388, 1984.

[8] M. H. Davis. *Markov Models & Optimization*, volume 49. CRC Press, 1993.

[9] L. Devroye. *Non-Uniform Random Variate Generation*. Springer Science & Business Media, 2013.

[10] E. Dynkin. Markov Processes.

[11] R. Eckhardt. Stan Ulam, John von Neumann, and the Monte Carlo method. *Los Alamos Science*, 15(131-136):30, 1987.

[12] E. Eichhammer. QCustomPlot: a Qt C++ Widget for Plotting and Data Visualization. `http://www.qcustomplot.com/`, 2017.

[13] W. Feller. *An Introduction to Probability Theory and its Applications: Volume I*, volume 3. John Wiley & Sons New York, 1968.

[14] J. M. Flegal, M. Haran, and G. L. Jones. Markov Chain Monte Carlo: Can We Trust the Third Significant Figure? *Statistical Science*, pages 250–260, 2008.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[16] A. E. Gelfand and A. F. Smith. Sampling-based Approaches to Calculating Marginal Densities. *Journal of the American statistical association*, 85(410): 398–409, 1990.

[17] A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin. *Bayesian Data Analysis*, volume 2. Chapman & Hall/CRC Boca Raton, FL, USA, 2014.

[18] Google. Google's googletest C++ Testing Framework. `https://github.com/google/googletest/`, 2017.

[19] B. Gough. *GNU Scientific Library Reference Manual - Third Edition*. Network Theory Ltd., 3rd edition, 2009. ISBN 0954612078, 9780954612078.

[20] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[21] W. K. Hastings. Monte Carlo Sampling Methods Using Markov Chains and their Applications. *Biometrika*, 57(1):97–109, 1970.

[22] J. S. Liu. *Monte Carlo Strategies in Scientific Computing*. Springer Science & Business Media, 2008.

[23] N. Metropolis. The Beginning of the Monte Carlo Method. Special Issue 15. *Los Alamos Science*, 1987.

[24] N. Metropolis and S. Ulam. The Monte Carlo method. *Journal of the American statistical association*, 44(247):335–341, 1949.

[25] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[26] R. M. Neal. Improving asymptotic variance of mcmc estimators: Non-reversible chains are better. *arXiv preprint math/0407281*, 2004.

[27] E. A. J. F. Peters and G. de With. Rejection-free Monte Carlo Sampling for General Potentials. *Phys. Rev. E*, 85: 026703, Feb 2012. doi: 10.1103/PhysRevE.85.026703. URL `http://link.aps.org/doi/10.1103/PhysRevE.85.026703`.

[28] Qt. Qt: a C++ Based Framework of Libraries and Tools that Enables the Development of Powerful, Interactive and Cross-Platform Applications and Devices. `http://www.qt.io/`, 2017.

[29] C. P. Robert and G. Casella. Monte Carlo Statistical Methods (Springer Texts in Statistics). 2005.

[30] S. M. Ross. *Stochastic Processes*. 1996.

[31] Y. Sun, J. Schmidhuber, and F. J. Gomez. Improving the asymptotic performance of markov chain monte-carlo by inserting vortices. In *Advances in Neural Information Processing Systems*, pages 2235–2243, 2010.

[32] H. Sutter. The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[33] S. D. Team. *Stan Modeling Language Users Guide and Reference Manual*, 2016. URL `http://mc-stan.org`.