Introduction to Inverse Problems and Markov Chain Monte Carlo Methods

Lorenzo Stigliano

Year 4 Project
School of Mathematics
University of Edinburgh
April 2021

Abstract

In this project we will give a comprehensive introduction to inverse problems and Markov chain Monte Carlo methods. We will begin by introducing inverse problems and how to solve them, focusing on the Bayesian inference approach. Then we will argue for the value in using Markov chain Monte Carlo methods when using this approach, thus shifting the focus of our project onto these methods.

We will begin by first introducing Markov chains to ensure that the reader has the appropriate background to understand the theory behind these methods. Then, we will shift our attention to Markov chain Monte Carlo methods, in particular to the Metropolis Hastings algorithm. Subsequently, we will propose a different set of Markov chain Monte Carlo methods called non reversible samplers. These, as we shall see, have highly desirable properties. We hope to persuade the reader to use these methods by comparing the performances of these different algorithms in a series of illustrative examples. To conclude, we look will look at inverse problems in imaging.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Lorenzo Stigliano*)

I want thank my supervisor, Dr Aretha Teckentrup, for the continuous help and guidance throughout the project. Secondly, I want to want to thank my friends and family for the support throughout this whole year. Finally, I would like to dedicate this project to my mum.

Contents

Chapter 1

Introduction

Inverse problems and Markov chain Monte Carlo methods are vast fields we will only begin to scratch the surface of.

One of the first appearances of an inverse problem was in the discovery of Neptune in 1846, by Le Verrier [3]. It was found by *observing* irregularities in measurements of Uranus. This encapsulated the essence of inverse problems; determining casual factors from observed data. Since then, inverse problems are present in a range of different domains in science, such as astronomy, physics [35], and computer vision [34], drawing tools form various different branches of mathematics to solve these problems.

Markov chain Monte Carlo methods are a class of algorithms that let us sample from some desired probability distribution. They make use of some valuable properties of Markov chains in order to achieve this. The first of these novel algorithms was proposed by Nicholas Metropolis in 1953 [27], known as the Metropolis algorithm. Then, in 1970 this method was generalized by Wilfred Keith Hastings [20] to the well known Metropolis Hastings algorithm. These methods are still an area of active research, for example, in the development of non reversible samplers [25].

This project will explore both of these areas in the hopes of achieving the following; Firstly, we hope that the reader will understand what inverse problems are and the methods of solving them. Furthermore, the reader should be able to reformulate inverse problems using the Bayesian inference approach, which in turn will give rise to the inherent need to use Markov chain Monte Carlo methods. Secondly, the reader will be able to comprehend what Markov chain Monte Carlo methods are, why they would want to use them and how they work. In particular, they should be able to understand and implement the Metropolis Hastings algorithm. As well as recognizing the importance of appropriately tuning its parameters. Furthermore, we will introduce novel sampling methods called non reversible samplers. We hope that the reader will understand the importance these algorithms in the larger field of sampling methods and how to implement them. Moreover, we hope to motivate them to use these non reversible sampler through a series of examples. Finally, we will put everything we have learnt throughout this project to solve inverse problems in imaging. We hope, that by the end of the project, the reader should able to understand how to formulate and solve inverse problems in imaging.

Chapter 2

Inverse Problems

2.1 What Are Inverse Problems?

Inverse problems attempt to determine casual factors from observed data. This can be thought as a mapping between objects of interest, which we call unknowns u, and acquired information about these objects, which we call observations or measurements y. Intuitively, we are interested in reconstructing the unknowns from the measurements.

Before going further, it is also important to understand what is the antonymous of an inverse problem? One can think that the *opposite* of an inverse problem is precisely a forward problem: the problem of calculating what should be observed for a particular model [1]. We can use Figure 2.1 to illustrate the idea.

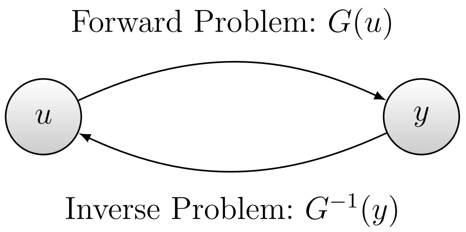


Figure 2.1: Visual representations of Inverse and Forward problems.

We can see that an inverse problem is concerned with going from observed data y to finding the unknowns u. The forward problem goes from the unknowns u which undergo the forward operator G thus creating an observation y which is what should be observed.

What is important to understand is what exactly is G? In our case G is the forward operator. This is a mathematical model of a process, which can be a linear or non-linear transformation of the unknown u.

We can now give a general mathematical framework for inverse problems. We are interested in the following problem: given observation $y \in Y$ we want to find the unknowns $u \in U$ such that they satisfy the following equation:

$$y = G(u) + \eta \tag{2.1}$$

We incorporate noise η, which arises due to inaccuracies in the measurement, and we are given a forward operator G. U and Y are Banach spaces which depend on the application. In many cases the observation space Y is usually finite dimensional and the parameter space U can be either finite dimensional or infinite dimensional.

However, the difficulty to solve these inverse problems arises from two main problems. The first one being that G^{-1}, does not exist or is not continuous. As a result, one cannot simply *invert* the forward operator to solve the equation for u.

Secondly, the observation y, in many cases, has additive noise. Noise can arise due to many reasons when we are observing the data. A possible source of error is, for example, measurement error. In many cases we do not know what the value of the noise η is or don't know what type of distribution it belongs to. As a result, we need to account for this noise in our equations for inverse problems.

Polynomial fitting

Here we will give an example, in particular Polynomial Fitting, that can be thought of as an inverse problem when formulated correctly. This example was inspired by the notes [47] and applied to the inverse problems setting.

In this problem suppose that we are given a set of observations $y = \{y_1, ..., y_n\}$ and a set of inputs $a = \{a_1, ..., a_n\}$. In this case we wish to find the degree $n-1$ polynomial that goes through these points. In particular, this is the same as finding the coefficients $x_j \in \mathbb{R}^n$ such that:

$$y_i = x_1 + x_2(a_i) + x_3(a_i)^2 + \ldots + x_n(a_i)^{n-1} \tag{2.2}$$

As a result, this corresponds to solving the linear system $A\hat{x} = \hat{y}$, where $\hat{x} \in \mathbb{R}^n$ is the vector containing $\{x_1, ..., x_n\}$ which are the coefficients we are trying to find, likewise $\hat{y} \in \mathbb{R}^n$ is the vector containing $\{y_1, ..., y_n\}$ observations and the matrix $A \in \mathbb{R}^{n\times n}$ is the $n \times n$ Vandermonde matrix with entries $a_{ij} = a_i^{j-1}$. This holds true because:

$$A\hat{x} = \hat{y} \iff (A\hat{x})_i = (\hat{y})_i \iff \sum_{j=1}^{n} a_i^{j-1} x_j = y_j \tag{2.3}$$

Therefore, the inverse problem is to find the unknowns $\hat{x} \in \mathbb{R}^n$, the coefficients of the equation, when we are given $A \in \mathbb{R}^{n\times n}$ and $\hat{y} \in \mathbb{R}^n$.

2.2 Well-Posed and Well-Conditioned Problems

In this section we will introduce the definitions of what we mean when we refer to problems as **well-posed** and **well-conditioned**, it is particularly important to understand these definitions when dealing with inverse problems.

Firstly, what is important to understand is what is meant by a well-posed problem. We will use the definition by the mathematician Jacques Hadamard, to define a well-posed problem [26]:

Definition 2.2.1 (Well-posed problem)**.** A problem is said to be well posed if it obeys the following three properties :

1. It has **a** solution.
2. The solution is **unique**.
3. The solution's behaviour depends **continuously** on the initial conditions.

Properties 1. and 2. are self-explained. However, we can elaborate on property 3. Suppose you have two different set of observations y_1 and y_2, when solving the problem we get two solutions for each set of observations, supposing 1. and 2. hold. Lets call these solutions u_1^* and u_2^*. Then we can say that the *solutions behaviour depends continuously* if $||u_1^* - u_2^*||_p \to 0$ as $||y_1 - y_2||_p \to 0$.

As a result, **ill-posed** problems are ones that fail to satisfy one of the above properties.

We can now define what is meant by well-conditioned problem [13]:

Definition 2.2.2 (Well-Conditioned problem)**.** A well-conditioned problem is one where a small change in the input u produces a small change in the output y. This can be illustrated mathematically:

$$G(u + \delta u) = y + \delta y \tag{2.4}$$

That is a small change δu results in a small change δy. However, this depends on the linearity of G.

Therefore, ill-conditioned problems is one which a small change in u (δu) can result in a large change in y (δy). The bound on the relative error is given by [13]:

$$\frac{||\delta u||_p}{||u||_p} \leq \kappa_p(G) \frac{||\delta y||_p}{||y||_p} \tag{2.5}$$

Where $u = G^{-1}y$ and $\delta u = G^{-1}\delta y$. We can see that the relative change is bounded above by $\kappa_p(G)$, the condition number of the matrix G. "The condition number of a problem measures the sensitivity of the solution to small perturbations in the input data" [21]. This is given by $\kappa_p(G) = ||G||_p \, ||G^{-1}||_p$ [21]. This measures how close a matrix is to being singular. Note that for ill-conditioned matrices the condition number is large. Therefore, if $\kappa_p(G)$ is small then $\frac{||\delta u||_p}{||u||_p}$ is small when $\frac{||\delta y||_p}{||y||_p}$ is small. Therefore, well conditioned matrices will have a smaller bound on the relative error.

In the next section we will introduce methods to solve inverse problems. We will see that the notions of ill-posed and ill-conditioned problems will be used throughout when solving inverse problems. In particular, we want our problems to be well-posed and well-conditioned.

2.3 Solving Inverse Problems

In this section we will be discussing two approaches to solve inverse problems. The first one being the optimisation approach and then give a brief motivation as to why we would want to use the Bayesian inference approach.

2.3.1 Optimisation Approach

The main idea behind the optimisation approach, is to find unknown values such that they minimizes an objective function. Formally, we want to find unknowns u that give predictions $G(u)$ which best predict the observed data y. As a result, we can formulate the optimization problem as:

$$u^* = \arg\ \min_{u \in U}\ \psi\,(G(u), y) \tag{2.6}$$

The misfit function ψ usually measures the distance between prediction $G(u)$ and the observed data y. Common distance metrics which can be used include, the L^1 norm or the squared L^2 norm, which gives an indication of how close the predictions are to the observed data.

What is also important to note is that we need to approximate or know what the forward operator G, such that we can transform the unknown u. Usually one needs to have knowledge of the application domain to find such a function [22]. However, it could be the case that we do not. As a result, how do we determine what is the forward operator G given the observations y? To find the forward operator one can use these three principles proposed in *Solution of inverse problems with limited forward solver evaluations: a Bayesian perspective.* [7]:

1. "Observe the forward operator on a well-selected set of input points."
2. "Build a surrogate based on your observations."
3. "Use the surrogate to pose and solve the inverse problem."

Common examples that use the principles mentioned above are generalized polynomial chaos surrogates and Gaussian processes, amongst others. The problem of these approaches is that they require a large number of iterations to construct a surrogate such that it is a good characterization of the forward operator G.

Furthermore, in many cases, the formulation of the optimization approach often leads to ill-posed/ill-conditioned problems. We introduce two examples to illustrate this.

Ill-posed problem

In this case, we are solving the inverse problem where we are choosing the naive optimisation approach. That is we are trying to minimize the squared L^2 norm which is the misfit function ψ in this case. The solution to the problem is given by:

$$u^* = \arg\ \min_{u \in U}\ ||y - G(u)||_2^2 \tag{2.7}$$

But this optimization problem is typically ill-posed [46]. This is because there could be more than one u that solves this problem, this means that there is **not** a unique minimizer. Furthermore, u in many cases not depend continuously on

y. Finally, it is well know that the problem is ill-conditioned, since small changes in y can lead to large changes in u [32].

Ill-conditioned problem

This an example, to show that inverse problems are ill-conditioned. We will use the problem of Polynomial Fitting, which was introduced in section 2.1.

In fact to solve this problem we are trying to solve the least squares problem of finding an $\hat{x} \in \mathbb{R}^n$ such that it minimises $||\hat{y} - A\hat{x}||_2^2$ when we are given $A \in \mathbb{R}^{n\times n}$ and $\hat{y} \in \mathbb{R}^n$, these are the same as in the previous example. As a result, we are trying to find the coefficients that have the closest *fit* to the observations.

We know that this problem has a unique minimizer if and only if A is of full rank [47, p. 42], that is A is an over determined system. The solutions are then given by the *normal equations* [47, p. 41]:

$$A^T A\hat{x} = A\hat{y} \tag{2.8}$$

As a result, if $A^T A$ is an ill-conditioned matrix then small changes in $\hat{y}$ result in large changes in $\hat{x}$. This is a direct consequence from the perturbation result, equation 2.5. In fact, we know this is the case since A is the Vandermonde matrix which is typically ill-conditioned.

As a result we need to find ways to solve ill-posed/conditioned problems. Regularization can be used to help tackle this ill-posed problems. Regularization takes into account a priori information on u. Acting as a *regularisation term*, $R(u)$, which is added to the optimisation problem [15]:

$$u^* = \text{arg min}_{u\in U} \quad \psi\left(G(u), y\right) + \alpha R(u) \tag{2.9}$$

In this case $\alpha > 0$ is a constant term which is used to find a balance between the prior knowledge and fitness to the data, if there is a lot noise on the observation y we usually increase α [52]. Likewise, if we know that there is little to no noise on our observations y then we make α small such that we give more importance on finding u such that it has a better fit to the observation y. Typical regularization terms include $||u||_1$ and $||u||_2$ but need to be chosen carefully for the given problem.

To conclude, the optimisation approach has three main components that we need to decide on. First we need to find a forward operator G that predicts the observation y. Secondly we need to define the misfit function ψ to measure how well the model fits to the observations y and finally we need to define an optimisation algorithm to find u which optimizes the objective function.

There are several other challenges that the optimisation approach faces. One of them is scaling with large problems. An increase in the dimensionality of the problem can cause issues with over fitting and computational complexity of the algorithms [51]. Another issue rises with choosing the correct regularisation term $R(u)$, which needs to be chosen carefully for the particular application. Furthermore, and perhaps more importantly, since we are solving for some value u this approach can only produce a *point estimate* of the unknown parameter, which in many cases is sufficient. However, what if we want more than just a point estimate? As we shall see in Chapter 3, the **Bayesian Inference approach** allows us to do so.

In this chapter we have introduced what inverse problems are and how we can solve them by using the optimisation approach in the next chapter we will shift our attention to the Bayesian Inference approach.

Chapter 3

Bayesian Inference Approach

In this chapter we will give an overview of the Bayesian inference approach to solve inverse problems. We will being by motivating the reader as to why they would use this approach. Then, we will discuss the importance and how to find the main components of this approach: the Prior, the Likelihood and the Posterior distribution.

3.1 Motivation

Before diving deeper into this approach it is vital to understand what Bayesian statistics is and difference between the frequentist interpretation of probability. To be concise Bayesian statistics allows people to update their beliefs in the evidence of new data [10], while the frequentist interpretation of probability views "probability as the limit of the relative frequency of an event after many trials" [30]. The Bayesian approach is exactly what we want. We can incorporate information about the unknown parameter u through the prior.

What is important to understand is why we are using this approach and what do the solutions look like. The main objective of Bayesian inference approach is to "extract information and assess the uncertainty about the variables with the knowledge available" [23, p. 49]. In turn, this means that the Bayesian inference approach is based on the following principles [23, p. 49]:

1. "All variables included in the model are modelled as random variables."
2. "The randomness describes our degree of information concerning their realizations."
3. "The degree of information concerning these values is coded in the probability distributions."
4. "The solution of the inverse problem is the posterior probability distribution."

The last principle is perhaps the most important and where this approach is set apart from others. As mentioned before the optimisation approach produces a *point estimate*. In this case, the solution is given as a **posterior probability**

distribution, which we will denote as $\pi(u|y)$. This can be used for **uncertainty quantification** in the inferred parameter u. Furthermore, the posterior **incorporates our knowledge** of the unknown u through the prior. This is particularly powerful since we can express one's beliefs about the unknown u before the observations y have been taken into account.

Another advantage over the optimisation approach is that the inverse problems are now **well-posed** [23, p. 50], in the sense of Hadamard, definition 2.2.1:

1. There exists a **unique** posterior distribution for all $y \in \mathbb{R}^{dy}$.
   This is particularly powerful since for any y we are ensured that there is a unique posterior distribution for our problem. As we have seen, when using the optimization approach, it could have been the case that where are several minimizers for the misfit function $||y - G(u)||_2^2$. The way that the posterior distribution takes care of multiple solutions is through multiple modes. That is, these points are *peaks* in the posterior distribution. This is highlighted in subsection 3.3.2 which shows the connection between the optimisation approach and the Maximum A-Posterior estimate of the posterior distribution, justifying this statement.

2. The posterior distribution depends **continuously** on y.
   This has to do with the third point of the definition 2.2.1 of well-posed problems. If the likelihood $L(y|u)$ is locally Lipschitz in y [31], then:

$$||\pi^y - \pi^{y'}||_{TV} \leq C||y - y'||_2 \tag{3.1}$$

   Where $||\cdot||_{TV}$ is the *Total Variational Norm* which is defined in Chapter 4 definition 4.3.1. π^y is the posterior distribution under observation y, likewise $\pi^{y'}$ is the posterior distribution under observations y'.

As a result, these are the some of the reasons we would like to use the Bayesian inference approach to solve inverse problems. In the next section we will give the general framework to formulate these problems.

3.2 Framework

In this section, I will give an overview of how to formulate inverse problem such that they can be solved by using the Bayesian inference approach. The goal of the Bayesian framework is to find a posterior distribution $\pi(u|y)$. A distribution of the unknown u given observations y.

Previously, we have defined our problem in equation 2.1 as $y = G(u) + \eta$, where we wish to find u given the observations y, by taking into account the noise η. Here, we assume that $u \in U$ and $y \in Y$, come from some continuous distributions U and Y. We will use the notation $\pi(u, y)$ to denote the joint probability density of U and Y. In this case all random variables are continuous.

We first need to define a prior density on u. We will call it $\pi_{pr}(u)$. The prior density is what we know about the unknown u *prior* to the observation y. This needs to be chosen carefully for the application and is often the most challenging part of the framework, which will be explained in subsection 3.2.1.

Secondly, we need to define the likelihood of the observation y. We can call this $L(y|u)$. This is the conditional probability on y given u.

We will make use of the chain rule for probability throughout to derive the posterior distribution $\pi(u|y)$.

Definition 3.2.1 (Chain Rule for Probability)**.** If we have two random variables X and Y, then the following holds true:

$$P(X,Y) = P(X|Y)P(Y) \tag{3.2}$$

Therefore, by using definition 3.2.1 we can define the **likelihood** by:

$$L(y|u) = \frac{\pi(y,u)}{\pi_{pr}(u)} \text{ if } \pi_{pr}(u) \neq 0 \tag{3.3}$$

Finally, assuming that the measurement is given. Then the **posterior distribution** of u is given by the following formula, that is the distribution of u given y, by using definition 3.2.1:

$$\pi(u|y) = \frac{\pi(u,y)}{\pi(y)} \text{ if } \pi(y) \neq 0 \tag{3.4}$$

We can rearrange equations (3.3) and (3.4) like so; $L(y|u)\pi_{pr}(u) = \pi(y,u)$ and $\pi(u|y)\pi(y) = \pi(u,y)$. Then, we can equate them by using the fact that $\pi(y,u) = \pi(u,y)$ to derive Bayes theorem of inverse problems [23, p. 51]:

$$L(y|u)\pi_{pr}(u) = \pi(u|y)\pi(y) \iff \pi(u|y) = \frac{L(y|u)\pi_{pr}(u)}{\pi(y)} \tag{3.5}$$

Theorem 3.2.1 (Bayes theorem of inverse problems)**.** Assume that the unknown $u \in U$ has a known prior probability density $\pi_{pr}(u)$ and that we have observations $y \in Y$ such that $\pi(u|y) > 0$, where U and Y are continuous distributions. Then the posterior probability distribution of u, given the observations y is given by:

$$\pi(u|y) = \frac{L(y|u)\pi_{pr}(u)}{\pi(y)} \tag{3.6}$$

What is important to notice is the expression of the normalizing constant can be given by:

$$\pi(y) = \int_{\mathbb{R}^n} \pi(y,u)du \ = \int_{\mathbb{R}^n} \pi(y|u)\pi(u)du \tag{3.7}$$

The normalizing constant is used to ensure the probability density function has a probability of **one** when integrated over the state space. However, this expression is generally intractable and cannot be calculated explicitly. In many applications, as we shall see, this will pose problems when we need to calculate equations that involve it directly. Therefore, we have an expression proportional to the posterior distribution:

$$\pi(u|y) \propto L(y|u)\pi_{pr}(u) \tag{3.8}$$

As a result we need to find ways to explore the posterior distribution without having to worry about the normalizing constant, as we shall see in Chapter 4.

To conclude, solving an inverse problem with the Bayesian framework can be reduced to three main tasks which are directly related to **Theorem 3.2.1** [23, p. 52]; First, by using our knowledge and the prior information available on the unknown u, we use this to find an prior probability density $\pi_{pr}(u)$. Secondly, we need to find a likelihood function, $L(y|u)$, for the observations y given the unknown u. Finally, we need methods to explore the posterior distribution such that we can produce point estimates or carry out uncertainty quantification. In next subsections we will give details on finding a prior and the likelihood function.

3.2.1 Prior Models

This is perhaps the hardest part of the Bayesian inference approach to define. The problem with finding an prior density in inverse problems usually comes from the essence of the prior information we have. In many cases our knowledge of the unknown is qualitative, thus transforming it into a quantitative form is where the difficulty lies. In many cases, this requires knowledge from people within the domain that the problem comes form.

As a result, picking a prior is hard. To find a prior $\pi_{pr}(u)$ we aim for it to have the following properties [23, p. 62]. Let E be the set of *likely observations* u and U' the set of *unlikely observations* u', then:

$$\pi_{pr}(u) \gg \pi_{pr}(u') \text{ where } u \subseteq E, u' \subseteq U' \tag{3.9}$$

What this is telling us is that the prior distribution should be larger for values we expect to see thus assigning higher probabilities to these with respect to observations that are less likely to occur. Typical prior models include, Gaussian Priors, Markov Random Fields [4] and Non Informative Priors [45].

Non Informative Priors are particularly useful when there is not sufficient prior information about the unknowns u. Thus, it is difficult for us to interpret them as a probability distribution. Then, in many cases, we would like to exclude this knowledge since it may effect the posterior distribution, in an unexpected way. Since the Bayesian inference approach *needs* the presence of a prior. The objective of these is to have a small impact on the posterior distribution.

3.2.2 Likelihood Functions

In this section we will discuss how to construct likelihood functions. To do this we need to examine the general inverse problem given by equation 2.1. As we can see, we are assuming *additive* noise since we are adding the random variable η to the end of the expression. Moreover, we assume that the probability distribution of the noise η is known. Furthermore, we assume in this case that u and η

are mutually independent. Throughout this section we will call the probability density function of the noise $\pi_{noise}\left(\eta\right)$.

By rearrange equation 2.1 we have that $y - G\left(u\right) = \eta$. Therefore, it must be the case that $y - G\left(u\right)$ is distributed in the same way as η, since they are equal. Then, if we fix u and use the independence assumption of u and η then we ensure that when y is conditioned on u it is distributed like η. As a result, we get an explicit expression for the likelihood:

$$L(y|u) = \pi_{noise}(y - G\left(u\right)). \tag{3.10}$$

Therefore, if we use Theorem 3.2.1 and assume we have a prior distribution. We can substitute the likelihood function in equation 3.8 to get the following expression for the posterior distribution:

$$\pi(u|y) \propto \pi_{noise}(y - G\left(u\right))\pi_{pr}(u) \tag{3.11}$$

We have written the posterior distribution as a function which is proportional to the prior distribution times a known distribution of the noise. Note that, in many cases the noise is not additive but multiplicative, that is, $y = \eta G\left(u\right)$. Therefore, techniques such as variable splitting and constrained optimization [8] can be used to take care of multiplicative noise.

3.3 Estimators

We have shown how to find the posterior distribution for a given problem. We will now introduce point estimates we can calculate given the posterior distribution. In particular, the conditional mean (CM) and the Maximum A-Posteriori (MAP) estimate.

3.3.1 Conditional Mean

This is the defined as the expected value of the unknown u conditioned on the observed data y. This can be thought as what the value u would take on *average over* the space of U. Formally this is equal to [23, p. 53]:

$$u_{CM} = E[u|y] = \int_{u \in U} u\pi(u|y)\ du \tag{3.12}$$

Where $\pi(u|y)$ is the posterior distribution found using Bayes Theorem 3.2.1. However, in many cases we cannot explicitly calculate the CM, given by equation 3.12 since it involves an integral which is generally intractable, due to the fact we do not have an explicit form for $\pi(u|y)$. This usually arises due to [46]:

1. Incorporating the prior distribution $\pi_{pr}(u)$. Which could be intractable.
2. Observation y has noise where its parameters are unknown. Therefore, we cannot explicitly evaluate the posterior distribution due to these unknowns.

3. Conditioning on $\pi\left(y|u\right)$ by using Bayes Theorem 3.2.1 results in the normalizing constant, equation 3.7, usually being intractable.

Therefore, we can estimate the CM by:

$$\hat{u}_{CM} \approx \sum_{i=1}^{\infty} u_i \text{ where } u_i \sim \pi(u|y) \tag{3.13}$$

As a result, this motivates us to find techniques such that let us sample form a distribution we do not know the explicit form of. To do this we will introduce a new class samplers called **Markov chain Monte Carlo methods**, which we will cover in the next chapter.

3.3.2 Maximum A-Posteriori Estimate

This is a point estimate of an unknown quantity which is equal to the mode of the posterior distribution $\pi(u|y)$.

This method is closely related to maximum likelihood estimation (MLE). MLE is a method of estimating the parameters of a probability distribution by maximizing a likelihood function [23, p. 53] in contrast to a posterior distribution.

Suppose, we are given an unknown parameter Θ, and a set of observations x that belong to some distribution f, when using MLE we wish to find Θ such that $f(x|\Theta)$ is maximized, that is:

$$\Theta_{MLE}(x) = \text{arg max}_{\Theta}\ f(x|\Theta) \tag{3.14}$$

Suppose that now we want to find the MAP estimate. In this case, Θ is a random variable with a known prior distribution, we call g. Notice that in Bayesian statistics we can encode all parameters as random variables. Then we can define the posterior distribution of $f(\Theta|x)$ by:

$$f(\Theta|x) = \frac{f(x|\Theta)g(\Theta)}{\int_{\mathbb{R}^n} f(x|\delta)g(\delta)\ d\delta} \tag{3.15}$$

Notice that the denominator $\int_{\mathbb{R}^n} f(x|\delta)g(\delta)\ d\delta$, is the normalizing constant which is always positive and does not depend on the variable Θ. Therefore, we can ignore this in our calculations for the MAP estimate. The MAP estimate then finds Θ as the mode of the posterior distribution:

$$\Theta_{MAP}(x) = \text{arg max}_{\Theta}\ f(\Theta|x) = \text{arg max}_{\Theta}\ f(x|\Theta)g(\Theta) \tag{3.16}$$

Now we can link this back to inverse problems. In this case we have that Θ would be equal to the underlying observation we are trying to estimate, u. Similarly, the value for x in equation 3.16 would be equal to the observed parameters y. This estimate would be the mode, which answers the question: *What is the most likely u given the observations y?* Formally given as [23, p. 53]:

$$u^*_{MAP}(y) = \text{arg max}_{u^* \in U}\ \pi(u|y) = \text{arg max}_{u^* \in U}\ \pi(y|u)\pi_{pr}(u) \tag{3.17}$$

The MAP estimate gives has a connection to the optimisation approach since it is trying to find a value which maximises some function. For example, suppose that the probability distribution of the likelihood $\pi(y|u)$ is $\mathcal{N}(G(u), \gamma^2 I)$ and the probability density of the prior $\pi_{pr}(u)$ is given by $\mathcal{N}(0, I)$. Then the maximum a-posterior estimate is given by [46]:

$$\begin{aligned} u^*_{MAP}(y) &= \text{arg max}_{u^* \in U}\ \pi(u|y) \\ &= \text{arg max}_{u^* \in U}\ \pi(y|u)\pi_{pr}(u) \\ &= \text{arg max}_{u^* \in U}\ \exp\left(-\frac{1}{2\gamma^2}||y - G(u)||_2^2 - \frac{1}{2}||u||_2^2\right) \\ &= \text{arg min}_{u^* \in U}\ ||y - G(u)||_2^2 + \gamma^2 ||u||_2^2 \end{aligned} \tag{3.18}$$

Notice that in the third line we are simply evaluating the explicit form of the equation. In this case we can see that γ^2, the variance of the noise, takes the role of the regularization term. We can see that the MAP estimate in this case is in fact precisely a problem formulated in the same way as with the optimisation approach. Furthermore, in many cases, to find the MAP estimate one needs to use an optimisation algorithm.

3.4 Putting It All Together

We can now put together sections 3.2 and 3.3 to show an example of how we can use the posterior in the context of inverse problems. We have taken the following example from [23, p. 90].

In this example we will work with the trivial problem, which is to determine the unknown $u \in U$ from observed data $y \in Y$. We measure u directly with some additive noise $e \in E$. In this case the inverse problem is in the form of $y = u + e$, notice that in this case there is no forward operator G. Furthermore, we assume that $U \sim \mathcal{N}(0, 1)$, $E \sim \mathcal{N}(0,\ \sigma^2)$ and Y is some continuous distribution. Therefore, we have explicit expressions for the prior $\pi_{pr}(u)$ and the noise $\pi_{noise}(e)$. Given by the probability density functions of their respective normal distributions. We can use these expression to derive an explicit formula for the posterior distribution by using equation 3.8 and using the density of the noise to define the likelihood like we have done in subsection 3.2.2:

$$\pi(u|y) \propto \pi(y|u)\pi_{pr}(u) \propto \pi_{noise}(y - u)\pi_{pr}(u) \tag{3.19}$$

We have decided to omit the steps here. But a complete derivation of the posterior distribution can be found in *Appendix A*. As a result we have that the posterior distribution is proportional to:

$$\pi(u|y) \propto \exp\left(-\frac{1}{2}\left(\frac{u - \mu}{\sigma'}\right)^2\right) \text{ where } \mu = \frac{y}{1 + \sigma^2} \text{ and } (\sigma')^2 = \frac{\sigma^2}{1 + \sigma^2} \tag{3.20}$$

Therefore, in this case the posterior distribution $\pi(u|y)$ is given by a $\mathcal{N}(\mu, (\sigma')^2)$ distribution. We can now explore the posterior distribution since we have an explicit expression for it. For example, if we want to calculate the CM, then this is given by $u_{CM} = E[u|y] = \mu$. This is because the expected value of the normal distribution is its mean. Furthermore, the MAP estimate is given by $u^*_{MAP}(y) = \mu$. This is because we know that the mode of a normal distribution is precisely its mean. As a result, this example illustrated how we once we have explicit forms of the prior and likelihood we can derive an expression for the posterior and calculate point estimates.

To conclude, in this chapter we have given an overview of what the Bayesian inference approach is and motivated the reader as to why they would want to use it. Then, we have given a general framework that one can use to define problems in this setting. Furthermore, we covered two point estimates which are of interest to calculate. We then introduced an example of how to put all the pieces together. However, we saw that there is a inherent need to use Markov chain Monte Carlo methods to be able calculate the conditional mean when we do not have an explicit form of the posterior distribution, thus the next chapter will be concerned with these sampling methods.

Chapter 4

Markov Chain Monte Carlo Methods

In this section we will introduce Markov chain Monte Carlo (MCMC) methods. We will begin by going through Markov chain definitions. After we have introduced these definitions we motivate the reader as to why they would want to use MCMC methods. Then we will dive into the Metropolis Hastings (MH) algorithm, understanding how the different parts of the algorithm work and the connection between the algorithm and Markov chain theory. As well as, highlighting some of the challenges the algorithm faces.

4.1 Markov Chains

Before we introduce Markov chain Monte Carlo (MCMC) methods we need to cover some key Markov chain definitions. These are the fundamental building blocks on which Markov chain Monte Carlo methods rely on. As a result, it is vital for us to present Markov chains in order to understand these methods and the intuition behind why they work.

To be concrete a Markov chain is a stochastic process, that is a collection of random variables defined on a common probability space [2, p. 8]. We will define these random variables as the set $\{x_i\}_{i=1}^{n}$. Furthermore, this set of random variable must have a special property known as the Markov Property, which we will introduce shortly. However, before we do we need to define need three additional basic elements of Markov chains [24]:

1. A **continuous state space** Ω, which uniquely defines the set of possible states the system can be in.

2. A **transition kernel** $K(x, A)$, For all $x \in \Omega$, $K(x, A)$ is defined as the probability of reaching the measurable set A from state x.

3. A **transition kernel density** $k(x, \cdot)$, such that for all $x \in \Omega$, $k(x, \cdot)$ is defined as a non-negative function such that

$$K(x, A) = \int_{y \in A} k(x, y)\ dy \tag{4.1}$$

Note that for a given x, $k(x, \cdot)$ is a probability density function, and therefore we have:

$$K(x, \Omega) = \int_{\Omega} k(x, y) \, dy = 1 \tag{4.2}$$

Definition 4.1.1 (Markov property)**.** A stochastic process is said to have Markov property if given the present state, the future events are independent of the past. For a discrete time sequence of random variables $\{X_1, X_2, ..., X_n\}$ on a continuous state space Ω, for any $A \in \Omega$ we have:

$$\begin{aligned} P[X_n \in A | X_{n-1} = x_{n-1}, ..., X_1 = x_1] &= P[X_n \in A | X_{n-1} = x_{n-1}] \\ &= K(x_{n-1}, A) = \int_{y \in A} k(x_{n-1}, y) \, dy \end{aligned}$$

Therefore, we have that:

$$P[X_n \in A | X_{n-1} = x_{n-1}, ..., X_1 = x_1] = \int_{y \in A} k(x_{n-1}, y) \, dy \tag{4.3}$$

For all $x_n \in \Omega$, $X_n \in \mathbb{R}^{d_x}$ and $n \in \mathbb{N}$.[39, p. 209].

Notice that in this set of definitions we are working with **continuous** state spaces as opposed to discrete state spaces. This is because when working with MCMC methods the distributions they sample from are, in many cases, continuous.

Furthermore, we need to note that we will be discussing discrete time Markov chains. This is because when we sample points the system evolves through discrete time steps, as we are sampling them sequentially. In contrast a continuous-time Markov chain is one in which changes to the system can happen at any time along a continuous interval [2, p. 57].

Definition 4.1.2 (N-step transition kernel)**.** We can also define an entry in the n-step transition kernel by:

$$K^{(n-1)}(x, A) = \int_{\Omega} k^{(n-1)}(x, y) k^{(1)}(y, A) \, dy \tag{4.4}$$

for all $x, y \in \Omega$ and $n \in N$. This is telling us that to get from x to A, some measurable set, we will pass by some state y before we reach A [39, p. 210].

Another important point is the dependency of transition probabilities at time n. If the transition probabilities do not depend on time, that is $K^{(n)}(x, A) = K(x, A)$ for all $x \in \Omega$, $n \in \mathbb{R}$ and A is a measurable set, we say that the Markov chain is **time homogeneous** [39, p. 209]. We will focus on time homogeneous chains in this project.

Example 1. This example highlights how we can define a Markov chain in discrete state spaces, inspired by example 2.2.1 in [2, p. 12]. Suppose every day you are eating at one of two restaurants and you are trying to model to what restaurant you will go to. In this case we have two states, the restaurants, lets call them A and B. Now that we have defined the state space $\Omega = \{A, B\}$ and let X_n be restaurant you are at on day n. We can define transition probabilities, that is the probability of going from one state to another, given the state you are currently in by:

$$P = \begin{bmatrix} \alpha & 1-\alpha \\ 1-\beta & \beta \end{bmatrix}$$

This means that when the previous day you were at restaurant A you stay at with probability α and eat at restaurant B with probability $1-\alpha$. Thus moving between the restaurants will produce a Markov chain. Below is the state diagram for this model.

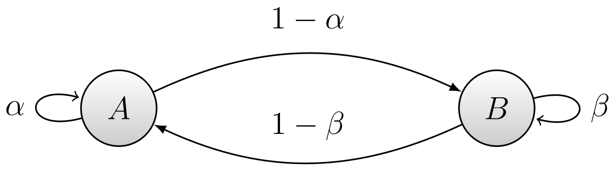


Figure 4.1: Visualization of the state space the chain can move in with the associated probabilities.

We need to introduce more definitions. In particular we need to be familiar with the concepts of irreducibility and periodicity with regards to Markov chains.

Definition 4.1.3 (ϕ-Irreducible)**.** For each measurable set A with $\phi(A) > 0$, for each x there exists n such that $K^{(n)}(x, A) > 0$. That is it has positive probability of reaching every subset A with $\phi(A) > 0$ from every state $x \in X$. [41] Furthermore, if a stationary distribution (Definition 4.1.5) exists, then it is unique if the chain is irreducible.

What this is telling us is that for any subset we can reach any other subset with positive probability after n steps [39, p. 213].

Definition 4.1.4 (Periodicity)**.** If there does **not** exists an $n \geq 2$ and a sequence of nonempty, disjoint measurable sets $A_1, A_2, ..., A_n$ such that $\forall x \in A_j$ $(j < n)$, $K(x, A_{j+1}) = 1$, and $\forall x \in A_j, K(x, A_1) = 1 \, \forall x \in A_n$, then the chain is **aperiodic**. If there does exist such an n then the chain is **periodic**. [41]

This means that if we have an aperiodic chain then from any a subset A of the state space we do not know how many steps are required to get back A.

The following definition we will introduce is vital for MCMC methods, as we shall see.

Definition 4.1.5 (Stationary Distribution)**.** A stationary distribution is a probability density function $\pi(x)$ for the Markov chain $\{x_i\}_{i=1}^n$ on Ω which satisfies the following equation:

$$\pi(x) = \int_\Omega \pi(x)k(x, y) \, dy \tag{4.5}$$

Note that if this is satisfied then $\pi(x)$ is the **stationary** distribution for the transition kernel K. Furthermore, it shows that if the chain is irreducible and allows for an a finite stationary measure, this measure $\pi(x)$ is **unique** [39, p. 223].

This is the direct analogy with discrete state spaces where the stationary distribution is a probability distribution that remain unchanged in the Markov chain as time progresses. Formally that is:

$$\pi = \pi P \tag{4.6}$$

Where π is the stationary distribution, it can be written as a vector of probabilities where $\sum_{i=1}^{n} \pi_i = 1$ and P is the transition probability matrix [2, p. 17].

The last definition we need to know is with regards to reversibility of Markov chains. Suppose we start a chain from its stationary distribution then the chain will be in its stationary state at each time step. This begs for the question: *what happens if we watch such a chain backwards in time?*

Definition 4.1.6 (Reversibility)**.** We say a chain with a transition kernel density $k(x, y)$ satisfies the **detailed balance equations** if there is a non-negative function $\pi(x)$ on Ω such that

$$\pi(y)k(y, x) = \pi(x)k(x, y),\ \forall x, y \in \Omega \tag{4.7}$$

We say that this chain is **reversible**. While it is not required that a Markov chain be reversible with respect to its stationary distribution, it is true that any distribution $\pi(x)$ satisfying equation 4.7 is in fact **stationary**. [39, p. 230]

Proof.

$$\begin{aligned}
&\pi(y)k(y, x) = \pi(x)k(x, y) \\
\iff &\int_\Omega \pi(y)k(y, x)\ dx = \int_\Omega \pi(x)k(x, y)\ dx \\
\iff &\pi(y) \int_\Omega k(y, x)\ dx = \int_\Omega \pi(x)k(x, y)\ dx \\
\iff &\pi(y) \underbrace{\int_\Omega k(y, x)\ dx}_{1} = \int_\Omega \pi(x)k(x, y)\ dx \\
\iff &\pi(y) = \int_\Omega \pi(x)k(x, y)\ dx
\end{aligned}$$

In the last step satisfies the definition of the stationary distribution 4.1.5. □

Therefore, we can see that for a discrete time Markov chain, when we reverse it, it in fact has the same stationary distribution as the forward direction. This fact will become extremely useful when we show the convergence of the Markov chains, using Markov chain Monte Carlo methods. Furthermore, reversibility will be the key to distinguish between different types of samplers as we shall see.

Now that we have introduced the definitions for Markov chains we are ready introduce Markov chain Monte Carlo methods.

4.2 Markov Chains Monte Carlo Methods

In many cases we are interested in evaluating the expected value for some function of interest φ under some target distribution $\pi(x)$ [39, p. 83]:

$$E[\varphi] = \int_{\mathbb{R}^n} \varphi(x)\pi(x)\ dx \tag{4.8}$$

We can estimate 4.8 by using a sampling method [39, p. 83]:

$$E[\varphi] = \frac{1}{N}\sum_{i=1}^{N} \varphi(x_i) \text{ where } x_i \sim \pi(x) \tag{4.9}$$

the x_i's are sampled from the distribution $\pi(x)$. Therefore, the challenge is to generate samples which are distributed according to $\pi(x)$. This difficulty can arise due to several reasons [39, pp. 35-41].

Firstly, the probability distribution one is sampling from is not known in closed form. In many cases, the normalization constant cannot be calculated and so one cannot simply generate independent samples. This is highlighted when using importance sampling, where one needs to be able to integrate the distribution they are trying to sample from in order to produce samples.

The next two issues occur when we use sampling methods, such as Monte Carlo methods, which rely on **random** sampling. It could be the case that the variable $x \in \mathbb{R}^{d_x}$ is high dimensional. This means that as the number of dimensions increases the number of possible values the samples can take increases exponentially. To illustrate this suppose we are dealing with the domain $[0, 1]$ where there are N possible values that the x can take in one dimension. Now suppose we increase the domain to $([0, 1] \times [0, 1])$, now x can take N^2 possible values. If we then generalize this to higher dimensions p, x can take up to N^p values. As a result, as the number of dimensions increases there is an exponential number of values it can take.

Finally, the probability distribution can concentrate on low-dimensional manifolds. This means that the function can concentrate in small areas of the state space. This goes hand in hand with the second problem, many of the points in high dimensions will have small density since the probability density function $\pi(x)$ is concentrated in a small space within this high dimensional space. This is in fact the *curse of dimensionality*. Therefore, random sampling will be inefficient since many samples proposed will have low density. As a result, we need to find ways of moving towards the areas of high density in order to produce samples from $\pi(x)$.

In fact, Markov chain Monte Carlo (MCMC) methods deal with many of these problems, allowing us to sample **directly** form a desired target probability distribution $\pi(x)$ which we do not know explicit form of. It suffices to know a function which is proportional to it. This is achieved by carefully constructing a Markov chain such that, in the limit, the stationary distribution of the chain is the same as the target distribution $\pi(x)$ we are trying to sample from.

It also important to note that these methods are generally used when trying to sample from **high** dimensional distributions. For single dimensional problems, methods such as adaptive rejection sampling [11] can be used to directly to return independent samples from the distribution. Furthermore, MCMC methods have the inherent problem of autocorrelated samples which we will shortly introduce.

As a result, the estimate for $E[\varphi]$, given by equation 4.8, when using an MCMC method is given by [39, p. 269]:

Definition 4.2.1 (MCMC estimator)**.** The Markov chain Monte Carlo estimator for the expected value of the function of interest φ under some target distribution $\pi(x)$ is given by:

$$\hat{E}_N^{MCMC} = \frac{1}{N}\sum_{i=1}^{N}\varphi(x_i) \tag{4.10}$$

where $\{x_i\}_{i=1}^{\infty}$ forms a *Markov chain.*

Advantages

There are **two** main advantages of using Markov chains for sampling. The first one being, that we can construct a Markov chain $\{x_i\}_{i=1}^{\infty}$ such that $x_i \sim \pi(x)$, as i tends to infinity for any x_1 [39, p. 269]. This means that we are ensured that starting from any sample x_1 eventually our chain $\{x_i\}_{i=1}^{\infty}$ be distributed according to the stationary distribution $\pi(x)$. Proved by using Theorem 4.3.1.

The second, being that we can construct a Markov chain $\{x_i\}_{i=1}^{\infty}$ such the Ergodic average converges to $E[\varphi]$ [39, p. 241]:

$$\lim_{N\to\infty}\frac{1}{N}\sum_{i=1}^{N}\varphi(x_i) \to E[\varphi] \tag{4.11}$$

where φ is a function of interest. This means that the estimator given by Definition 4.2.1 is ensured to converge to the expected value of the quantity of interest φ we are trying to calculate. This is shown by using Theorem 4.4.1. In this context *Ergodic* is a property the Markov chain must have. It must be both **π-irreducible** and **aperiodic**.

Linking this back to the Bayesian inference approach, where the posterior distribution was given by $\pi(u|y) \propto L(y|u)\pi_{pr}(u)$. In many cases, we are interested in producing an estimate for the CM. To do this we needed to evaluate some integral which was generally intractable. Therefore, we estimated the CM using 3.13 given by:

$$u_{CM} = E[u|y] \approx \frac{1}{N}\sum_{i=1}^{N} u_i \text{ where } u_i \sim \pi(u|y) \tag{4.12}$$

However, we can now use MCMC methods to sample from $\pi(u|y)$ this is because it suffices to know $L(y|u)\pi_{pr}(u)$ which is proportional to the posterior distribution. Recall, that these methods **do not** need to know the explicit form of the target distribution one is trying to sample from.

4.3 Convergence Theorem

In definition 4.1.5 we introduced the notion of the stationary distribution of a chain. That is a distribution where if we take a step to any other state we remain at this distribution. If the stationary distribution of our chain $\{x_i\}_{i=1}^{\infty}$ is $\pi(x)$ we need to address the question: *do we always converge to the desired distribution?* To answer this question we will use properties of Markov chains to ensure that our algorithm generates samples for the desired target distribution $\pi(x)$. We first need to define the concept of Total Variational Norm [39, p. 231].

Definition 4.3.1 (Total Variational Norm)**.** This is the largest possible difference between the probabilities that the two probability distributions can assign to the same event.

$$||\pi_1 - \pi_2||_{TV} = sup_A|\pi_1(A) - \pi_2(A)| \tag{4.13}$$

Using the definition above we can state the theorem for the Markov chain convergence theorem [39, p. 234].

Theorem 4.3.1 (Markov Chain Convergence Theorem)**.** Let K be a Markov transition kernel, with stationary distribution $\pi(x)$, and let K be **π-irreducible** (definition 4.1.3).

If K is **aperiodic** (definition 4.1.4) on a countable space $A \in \Omega$, we have that for every initial state x:

$$\lim_{n\to\infty} ||K^n(x, A) - \pi(A)||_{TV} = 0 \tag{4.14}$$

where $|| \cdot ||_{TV}$ is the total variation norm, given in Definition 4.3.1.

In particular if the chain is **Harris recurrent** then the chain converges for all x [39, p. 234].

This means that we can start at any x, and run the chain for a long time, and we are assured that the final sample comes from the distribution $\pi(x)$. However, the chain needs **π-irreducible** and **aperiodic**. In fact, these conditions are satisfied by the Metropolis Hastings algorithm, as we shall see.

Another aspect we need to consider is, if we know that our chain will converge to the stationary distribution. *When do we know that our chain has reached this distribution?* To address this we will introduce the notion of burn-in.

4.3.1 Burn-In

To answer this question it is important to understand that when we use these algorithms the chains can start form any initial position x_1. In most cases the first few samples of MCMC methods do not belong to the stationary distribution $\pi(x)$, since the chains require time to converge to the stationary distribution. To take this into account a *burn-in* period is added. The burn-in period discards a certain number of the initial samples from the chain. So, for example, if we want to compute the estimate for the expected value $E[\varphi]$ 4.2.1 with a burn-in phase, we can estimate like so:

$$E[\varphi] = \frac{1}{N - B} \sum_{i=B+1}^{N} \varphi(x_i) \tag{4.15}$$

Where $B \geq 0$ is the *burn-in* period. However, we need to be careful with what values of B we choose. If B is too small, then the estimator will be influenced by the starting values. However, of B is too large then the estimator will have too little samples.

There are several proposed ways of finding this B value, however it is often difficult to find this period. One way B is found using is by using a convergence diagnostic. The Markov chain output is analysed to see at what point it becomes *stable* in the sense that the samples belong to the distribution [17]. Another approach is to prove it analytically. That is we need to show that the distribution of the samples with the burn-in B in fact come from the stationary distribution $\pi(x)$ [28]. However, what is usually done, and what we will be doing in this project, is that B is selected in an ad hoc fashion depending on the problem. Furthermore, it is important to mention, that if there are enough samples and B is large enough then this is usually sufficient to ensure that the chain has reached $\pi(x)$.

4.4 Central Limit Theorem

In section 4.2 we introduced some of the reasons to use MCMC methods, in particular we stated that Ergodic average does indeed converge to the expected value of interest $E[\varphi]$. To show this we need to support it with theory. The Central Limit Theorem (CLT) ensures that this is the case [39, p. 243].

Theorem 4.4.1 (Central Limit Theorem)**.** Consider a chain $\{x_i\}_{i=1}^{n}$, which is both **π-irreducible** (definition 4.1.3) and **aperiodic** (definition 4.1.4) and suppose that $\sigma_\varphi^2 < \infty$, where σ_φ^2 is the *Asymptotic Variance*. Then, as $N \to \infty$, we have:

$$\sqrt{\frac{N}{\sigma_\varphi^2}} \left(\frac{1}{N} \sum_{i=1}^{N} \varphi(x_i) - E[\varphi] \right) \to \mathcal{N}(0, 1) \tag{4.16}$$

Furthermore, this theorem holds true for both reversible and non reversible chains [39, p. 246]. The difference lies in the derivation.

This theorem ensures that the estimator does indeed converge to our desired property we are trying to calculate, that is the expected value of the function of interest given in equation 4.11.

Furthermore, from Theorem 4.4.1 we know that the estimator $\frac{1}{N}\sum_{i=1}^{N}\varphi(x_i)$ is asymptotically normally distributed with mean $E[\varphi]$ and variance $\frac{\sigma_\varphi^2}{N}$. But what does this mean? To understand this we need to introduce a new important concept, the *Asymptotic Variance* [39, p. 244].

Definition 4.4.1 (Asymptotic Variance)**.** Given a Markov chain $\{x_i\}_{i=1}^n$ we define the asymptotic variance by:

$$\sigma_\varphi^2 = Var[\varphi(x_1)] + 2\sum_{i=2}^{\infty} Cov[\varphi(x_1), \varphi(x_i)] \tag{4.17}$$

Where φ is the function of interest. If it is the case that all the samples are independent and identically distributed then we can see that the variance of the estimator would be given by $\frac{\sigma_\varphi^2}{N} = \frac{Var[\varphi(x_1)]}{N}$. However, when dealing with MCMC methods this is not the case. In fact, samples are not independent, due to the nature of how the chains are created sequentially. As a result, we need to include the covariance term between different time lags to take this into account, notice that $Cov[\varphi(x_1), \varphi(x_i)]$ is the covariance between samples in the chain that have the same time lag between them. For example, if $i = 3$ we would calculate the covariance by using all samples with lag 2 between them that is: (x_1, x_3), (x_2, x_5),...,(x_{n-2}, x_n). This motivates us to calculate the asymptotic variance since if we are able to compute it then we know how well the algorithm is preforming. Smaller value of σ_φ^2 would mean that the variance of the estimator is smaller thus we have a tighter approximation to our value of interest $E[\varphi]$, for a fixed N. We can see the variance decreases as $\frac{1}{\sqrt{N}}$ as you generate more samples.

However, σ_φ^2 usually cannot be computed directly. There are several challenges to calculating this value, the first one being that we cannot simply sum to infinity. Notice that $2\sum_{i=2}^{\infty} Cov[\varphi(x_1), \varphi(x_i)]$ term in equation 4.17 calculates the sum of the covariance between the lags from 2 to infinity. However, we can only sum up to the length of the given Markov chain, which in this cases is n. Therefore, we can estimate the asymptotic variance by:

$$\hat{\sigma}_\varphi^2 = Var[\varphi(x_1)] + 2\sum_{i=2}^{n} Cov[\varphi(x_1), \varphi(x_i)]. \tag{4.18}$$

In many cases this is still poses a problem. For large time lags, that is large value of i's, we do not have many samples. Take for example, a chain of length 100 when we want to calculate the $Cov[\varphi(x_1), \varphi(x_i)]$ when $i = 50$, we would only have two samples, as a result the value for our asymptotic variance would not be stable.

To resolve this we again re-estimate 4.18 by finding a value $M \ll n$, this is done very much in an ad hoc fashion.

$$\hat{\sigma}_\varphi^2 = Var[\varphi(x_1)] + 2\sum_{i=2}^{M} Cov[\varphi(x_1), \varphi(x_i)]. \tag{4.19}$$

We can use a small enough M to have enough samples to find a good estimate for the asymptotic variance. This is because as the lag increases the covarince between the sample does too. So for large lags then they should have a smaller influence on the estimate for the asymptotic variance.

4.5 Normalized Autocorrelation Function

In this section we will introduce an alternative form of the variance of the estimator $E[\varphi]$ by using the integrated auto-correlation time which involves an important function known as the *normalized autocorrelation function.* We introduce this new concept because it will be used extensively throughout the analysis section when comparing the performance of different algorithms.

Before introducing the concept of integrated auto-correlation time. We first need to understand what is meant by **autocorrelation**. Autocorrelation "is a characteristic of data which shows the degree of similarity between the values of the same variables over successive time intervals" [9]. This means that over some time lag we measure how related two variables are.

Recall, that the variance for the estimator $\hat{E}_N^{MCMC} = \frac{1}{N}\sum_{i=1}^{N}\varphi(x_i)$ was given by $\frac{Var[\varphi(x_1)]}{N}$, if all the samples where **independent**. As we have explained in section 4.4 when using MCMC methods we usually produce correlated samples. So, the variance of the estimator is given by $\frac{\sigma_\varphi^2}{N}$ where σ_φ^2 is the asymptotic variance defined in equation 4.17. We will introduce a variable called the **integrated autocorrelation time** given by ν_φ, such that, we can rewrite $\frac{\sigma_\varphi^2}{N}$ in terms of ν_φ.

Definition 4.5.1 (Integrated Autocorrelation Time)**.** Given a Markov chain $\{x_i\}_{i=1}^{n}$ we define the integrated autocorrelation time ν_φ by:

$$\nu_\varphi = 1 + 2\sum_{\tau=2}^{\infty}\varrho_\varphi(\tau) \tag{4.20}$$

In the equation ϱ_φ is the **normalized autocorrelation function** of the stochastic process that generated the chain for φ, the function of interest.

Definition 4.5.2 (Normalized Autocorrelation Function)**.** Given a Markov chain $\{x_i\}_{i=1}^{n}$ and some lag τ we define the normalized autocorrelation function by:

$$\varrho_\varphi(\tau) = \frac{c_\varphi(\tau)}{c_\varphi(0)} \tag{4.21}$$

Where $c_\varphi(\tau)$ is the covariance between samples with lag τ:

$$c_\varphi(\tau) = \frac{1}{n-\tau}\sum_{i=1}^{n-\tau}(\varphi(x_i)-\mu)(\varphi(x_{i+\tau})-\mu), \text{ where } \mu = \frac{1}{n}\sum_{i=1}^{n}\varphi(x_i) \tag{4.22}$$

Notice that, $c_\varphi(\tau) = Cov[\varphi(x_1), \varphi(x_{i+\tau})]$ these expressions are the same. Furthermore, this means that when we calculate $c_\varphi(0)$ this is exactly the expression for the variance, that is $Var(\varphi(x_1))$. So, the normalized autocorrelation function is normalizing the covariance of each time lag τ by the variance of the chain. Thus, it is telling us how **correlated** samples are with the same time lag in a Markov chain. Furthermore, it is important to understand that as τ increases the samples in the chain should be less correlated with one another, since we they are further apart.

What is useful about the **normalized autocorrelation function** $\varrho_\varphi(\tau)$ is that it can be calculated for **each** time lag τ. As a result, when comparing the performance of different MCMC methods, we can plot $\varrho_\varphi(\tau)$ against time lag τ. Better preforming algorithms have a high rate of decay for this graph. This is because quicker decay for $\varrho_\varphi(\tau)$ means that the samples are less correlated for smaller time lags therefore the integrated autocorrelation time ν_φ has a smaller value (since we sum $\varrho_\varphi(\tau)$ over all time lags) which in turn means that we are reducing the variance of our estimator $\frac{1}{N}\sum_{i=1}^{N}\varphi(x_i)$, as desired.

We will now rewrite the variance of the estimator $\hat{E}_N^{MCMC}$ in terms of the **integrated autocorrelation time**:

$$\begin{aligned}
\frac{\sigma_\varphi^2}{N} &= \frac{Var[\varphi(x_1)] + 2\sum_{i=2}^{\infty} Cov[\varphi(x_1),\varphi(x_i)]}{N} \\
&= \frac{1 + 2\sum_{i=2}^{\infty} \frac{Cov[\varphi(x_1),\varphi(x_i)]}{Var[\varphi(x_1)]}}{N} Var[\varphi(x_1)] \\
&= \frac{1 + 2\sum_{i=2}^{\infty} \frac{Cov[\varphi(x_1),\varphi(x_i)]}{Cov[\varphi(x_1),\varphi(x_1)]}}{N} Var[\varphi(x_1)] \\
&= \frac{1 + 2\sum_{\tau=2}^{\infty} \frac{c_\varphi(\tau)}{c_\varphi(0)}}{N} Var[\varphi(x_1)] \\
&= \frac{1 + 2\sum_{\tau=2}^{\infty} \varrho_\varphi(\tau)}{N} Var[\varphi(x_1)] \\
&= \frac{\nu_\varphi}{N} Var(\varphi(x_1))
\end{aligned} \tag{4.23}$$

So we can write the variance of the estimator by using ν_φ. As a result, form equation 4.23, we can see that the quantity $\frac{N}{\nu_\varphi}$ is the number of samples that are needed before the chain *forgets* where it started, likewise ν_φ is the number of steps need to forget where it started. Therefore, if we can estimate ν_φ we can get an idea of how many steps we need to take, such that the chain produces independent samples.

Furthermore, the integrated autocorrelation time cannot be calculated explicitly, similar to when we wanted to calculate the asymptotic variance. In this case we cannot sum over all lags, since we have a fixed number of samples N. Therefore, we can simplify ν_φ given in equation 4.20 to:

$$\hat{\nu}_\varphi = 1 + 2\sum_{\tau=1}^{N} \varrho_\varphi(\tau) \tag{4.24}$$

However we are left with the same problem as before, where we cannot estimate it up to lag N since for large values we do not have enough samples, which would result in a unstable calculations. As a result, we pick some $M \ll N$ such that we have sufficient samples to estimate equation 4.24.

It is suggested, in [44], to pick an M such that it is the smallest value that satisfies $M \geq C\hat{\nu}_\varphi$ for a constant $C \approx 5$. Furthermore in the article it was suggested that this works best with chains longer than $1000\hat{\nu}_\varphi$.

4.6 Metropolis Hastings Algorithm

In this section we will cover the Metropolis Hastings Algorithm, perhaps one of the most well known MCMC methods. This algorithm provides a general framework for other MCMC methods.

4.6.1 Overview

The Metropolis Hastings (MH) algorithm allows us to sample from any distribution $f(x)$ if we have some function $\pi(x)$ which is proportional to it. However, we must be able to explicitly **evaluate** $\pi(x)$ for any x. This is why the MH algorithm is particularly useful. The algorithm generates a sequence of samples where we are ensured that they come from the desired target distribution $\pi(x)$ which is the distribution we are trying to sample from. As we shall see the chains produced by this algorithm have the required properties, **π-irreducible** and **aperiodic**, such that Theorem 4.3.1 holds. When using this algorithm the next sample **only** depends on the previous sample, this ensures we are creating a Markov chain.

I want to highlight that the MH algorithm is also known as a **reversible sampler** since the chain produced is reversible. This will be made evident in subsection 4.6.4.

Furthermore what is worth mentioning is that the MH algorithm is one of the many different algorithms that fall within the MCMC set of algorithms. Other types of algorithms include, but not limited to, the Gibbs sampler, Slice Sampling, the Hamiltonian Monte Carlo sampler and the Multiple-try Metropolis.

4.6.2 Algorithm

In this section I will give an overview of the algorithm and explain each of the parts. Finally, I will give an intuition for why the algorithm works.

I have outlined the algorithm using pseudo-code, in order to make the steps clear, the algorithm is given in [38]. *For a Python implementation of this algorithm refer to Appendix B.*

Algorithm 1: Metropolis Hastings Algorithm

```
randomly pick $x_1$ such that $\pi(x_1) > 0$;
for $i = 1$ to $N_{iter}$ do
    sample $x\prime \sim q(x\prime|x_i)$;
    $\alpha(x\prime|x_i) = \min\left\{1, \frac{\pi(x\prime)}{\pi(x_i)}\frac{q(x_i|x\prime)}{q(x\prime|x_i)}\right\}$;
    sample $u \sim U[0,1]$
    if $u < \alpha(x\prime|x_i)$ then $x_{i+1} = x\prime$ else $x_{i+1} = x_i$;
```

Before we run the algorithm, we let $\pi(x)$ be the target distribution, that is the distribution we wish to sample from. We also need to choose some proposal density $q(x\prime|x_i)$ to sample proposals from. We usually pick one which is easy to sample from. We also need to pick the number of iteration the algorithm will run for, N_{iter}.

The algorithm can be divided in two parts, the initialization and the iteration steps.

Initialization

We initialize the algorithm by picking a random sample, x_1 such that $\pi(x_1) > 0$. Since we only need sample points which have positive probability under the target distribution $\pi(x)$. This is line **1** of Algorithm 1.

At each iteration

- At line **3**, we need to sample some candidate $x\prime$ given the current sample value x_i which is the current position the chain, by using the proposal $q(x\prime|x_i)$. In subsection 4.6.3 it will be made clear how this can be done.

- Secondly, we need to calculate the acceptance ratio, done in line **4**. This is the probability that we accept the current sample $x\prime$. The reason it takes this form is such that it ensures that the chain produced does indeed have a stationary distribution given by $\pi(x)$.

- Finally, the chain can either remain at the sample x_i if it rejects $x\prime$ or it moves to the proposed sample $x\prime$ upon acceptance, with probability $\alpha(x\prime|x_i)$. That is, at the next iteration $i+1$, $x_{i+1} = x\prime$, if we accept $x\prime$ otherwise $x_{i+1} = x_i$. This step is vital to ensure we are sampling from $\pi(x)$. We also sample some random variable $u \in [0,1]$, in line **5**, which is then used to decide to either accept or reject $x\prime$ in line **6** if it is less than $\alpha(x\prime|x_i)$, since this is the probability of accepting. Note that in **both** cases we add the sample to the chain. Therefore, once the algorithm runs for a number of iterations N_{iter}, we will have a total of $N_{iter}+1$ samples, including the first sample x_1.

As a result, this procedure can be repeated to create the Markov chain $\{x_i\}_{i=1}^{N_{iter}+1}$ that is ensured to converge to $\pi(x)$. Thus sampling points form the desired target distribution, we will show this in subsection 4.6.4.

A natural question to ask is: *when do we accept proposals $x\prime$?* To answer this question we need to dive into the acceptance ratio: $\alpha(x\prime|x_i) = \min\left\{1, \frac{\pi(x\prime)}{\pi(x_i)}\frac{q(x_i|x\prime)}{q(x\prime|x_i)}\right\}$.

1. If $\frac{\pi(x\prime)}{\pi(x_i)} > 1$, then $\pi(x\prime) > \pi(x_i)$ this means that the proposal $x\prime$ has a higher target probability than x_i. Which means we would like to sample this point since it belongs to an area of high density. Thus, we are more likely to accept the proposal $x\prime$ since if $\frac{\pi(x\prime)}{\pi(x_i)} > 1$ then the acceptance ratio $\alpha(x\prime|x_i)$ is more likely to be close to 1.

2. If $\frac{q(x_i|x\prime)}{q(x\prime|x_i)} > 1$, then $q(x_i|x\prime) > q(x\prime|x_i)$ this means that there is a high probability of moving back to x_i from $x\prime$. This means that it is likely to go back to samples that have already been accepted, which is desirable. So, again, we are more likely to accept the proposal $x\prime$ since if $\frac{q(x_i|x\prime)}{q(x\prime|x_i)} > 1$ then the acceptance ratio $\alpha(x\prime|x_i)$ is more likely to be close to 1.

There a couple of notes. Firstly, if the proposal density is symmetric then the acceptance ratio can be reduced to $\alpha(x\prime|x_i) = \min\left\{1, \frac{\pi(x\prime)}{\pi(x)}\right\}$. To illustrate this suppose we use a one dimensional normal distribution proposal density. We know that $q(x_i|x\prime) \sim \mathcal{N}(x\prime, \beta^2)$, likewise $q(x\prime|x_i) \sim \mathcal{N}(x_i, \beta^2)$. Then due to symmetry we can see that the probability density values are exactly the same on both of the distributions. To see this we make use of the probability density function of a normal distribution which is given by: $f(x) = \dfrac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$. As a result, in this case the explicit forms of the probability density functions are given by:

$$q(x\prime|x_i) = \frac{1}{\beta\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x\prime - x_i}{\beta}\right)^2} \text{ and } q(x_i|x\prime) = \frac{1}{\beta\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x_i - x\prime}{\beta}\right)^2} \tag{4.25}$$

Notice that these two expressions are exactly the same. As a result they cancel out in the fraction of the acceptance ratio $\alpha(x\prime|x_i)$.

Furthermore, as we have stated in subsection 4.2, we do not need to know the explicit form of the target distribution $\pi(x)$, *but why is this the case?* Notice that $\pi(x)$ only appears in the acceptance ratio $\alpha(x\prime|x_i)$ in particular as a fraction; $\frac{\pi(x\prime)}{\pi(x_i)}$. As a result, this fraction uses the **same** distribution $\pi(x)$, for both the numerator and denominator, this means all constants in $\pi(x)$ *cancel* out through division. This is a subtle but important point to highlight since this is one of the main advantages of using MCMC methods.

4.6.3 Proposal Densities

We can see that in the MH algorithm we have control over the proposal density, so an important aspect of this algorithm is to consider which proposal to pick.

Proposal densities let us explore the state space by deciding on how move to the next sample on the chain. It is important that when picking between proposal densities we need to consider two main factors: accuracy and cost. What is meant by accuracy is that we want to produce samples such that we reduce the asymptotic variance. This is achieved by trying to reduce the correlation between samples. Likewise for cost we refer to the computational cost needed to produce a proposal $x\prime$. For example, proposal densities that require to compute gradients might have higher accuracy but are computationally more expensive. As a result, we need to find a balance between these two aspects. Below I will illustrate some of the most common proposal densities used.

Random Walk proposal

When using the Random Walk proposal, our algorithm is called **Random-walk Metropolis algorithm**. As the name suggests we propose samples $x\prime$ by using a random walk on the state space the algorithm is sampling from. Since we are using a symmetric distribution we can reduce the acceptance ratio to $\alpha(x\prime|x_i) = \min\left\{1, \frac{\pi(x\prime)}{\pi(x_i)}\right\}$.The Random Walk proposal density is given by [39, p. 287]:

$$q(x\prime|x_i) \sim \mathcal{N}(x_i, \beta^2 I) \tag{4.26}$$

for some $\beta > 0$, this is equivalent to $x\prime = x_i + \beta\epsilon$ where $\epsilon \sim \mathcal{N}(0, I)$.

In this case β controls the step size. We need to tune this parameter for the particular problem. If β is too small then the algorithm does not explore the state space. Similarly, if it is too large then we reject $x\prime$ too often because we take steps in areas outside of the target distribution. In both cases this leads to a large asymptotic variance. In Chapter 6 we will dive into tuning this parameter in depth and give more details as to *why* this is the case.

However there are some drawbacks from using this proposal. The main one being that it does not work well in high dimensions. For any value of β, the average acceptance ratio $\hat{\alpha} \to 0$ as dimension $d_x \to \infty$ [12], which in turn means that the asymptotic variance $\sigma_\varphi^2 \to \infty$. This is because if the average acceptance ratio is 0 then we do not accept any new sample and as a result our chain remains in the same position thus increasing the asymptotic variance.

This means it becomes hard to explore higher dimensional spaces with a Random Walk proposal, so we need to find alternatives to the Random Walk when we wish to explore higher dimensional spaces. The next proposal addresses this issue.

We define the *average acceptance ratio* $\hat{\alpha}$ to be the average value the acceptance ratio takes over all iterations of the algorithm.

Pre-conditioned Crank-Nicolson

The pre-conditioned Crank-Nicolson (pCN) proposal is well-defined for high-dimensional spaces [18]. This means that as the number of dimensions increase the asymptotic variance is independent of the dimension unlike the Random Walk proposal density. Therefore, pCN method is robust to the dimension of the problem. In fact the specific form of the pCN proposal depends on a reference measure π_0. If π_0 is $N(0, C_0)$ then pCN proposal is given by:

$$q(x\prime|x_i) \sim \mathcal{N}(\sqrt{1-\beta^2}x_i, \beta C_0) \tag{4.27}$$

therefore this is equivalent to $x\prime = \sqrt{1-\beta^2}x_i + \beta\epsilon$ where $\epsilon \sim \mathcal{N}(0, C_0)$ where in this case $0 < \beta < 1$. Notice that the step size the algorithm takes depends on **both** β and C_0. C_0 is know as the **covariance operator**.

The proposal we have introduced all follow the same principal, they do not have an idea about which parts of the state space are more probable. This is because the proposals have some sort of *randomness* about their movement. However, we would like for our proposals to move in the direction where it is more probable to sample from the desired distribution $\pi(x)$. To do this we draw inspiration from optimization algorithms such that we incorporate gradient information. We would like to propose the next candidate $x\prime$ in the following fashion:

$$x\prime = x_i + \beta\nabla\pi(x_i) \tag{4.28}$$

Notice that now we move in a deterministic way. We are losing randomness and we also lose the ability to explore the state space. This is because when we calculate the gradient, we force the movement to a certain direction. In particular, we are converging towards the local maximum of the distribution $\pi(x)$. The following proposal density follows this principle.

Metropolis Adjusted Langevin Algorithm (MALA)

Using this technique the new states are proposed using Langevin dynamics, which evaluates the gradient of the target probability density $\pi(x)$. Then proposals are accepted or rejected using the MH algorithm. In this case the MALA proposal density is given by:

$$q(x\prime|x_i) \sim \mathcal{N}(x_i + \beta \nabla log\pi(x_i), 2\beta I) \tag{4.29}$$

therefore we have that $x\prime = x_i + \beta\nabla log\pi(x_i) + \sqrt{2\beta}\epsilon$ where $\epsilon \sim \mathcal{N}(0, I)$.

4.6.4 Existence of Stationary Distribution

In this section we will prove that the Markov chain created with this MH algorithm has a unique stationary distribution which is exactly the target distribution $\pi(x)$. We will show this by using the fact that if $\pi(x)$ satisfies the detailed balanced equations, then $\pi(x)$ must be the stationary distribution. Notice, that if it satisfies the detailed balanced equations, then by definition 4.2.1 the chain created is **reversible**.

To do this we first need to define the transition kernel for the Metropolis Hastings algorithm, $P(x\prime|x)$. For this algorithm it is simply given by [39, p. 272]:

$$P(x\prime|x) = \underbrace{\alpha(x\prime|x)q(x\prime|x)}_{Accept\ x\prime} + \underbrace{r(x)\delta_x(x\prime)}_{Reject\ x\prime} \tag{4.30}$$

where $\alpha(x\prime|x)$ is the acceptance ratio and $q(x\prime|x)$ is the proposal density. Before we begin the proof, let us explain the terms in the transition probability (4.30), when we pick a new proposal $x\prime$ at x we have two options:

1. **Accept** $x\prime$ with probability $\alpha(x\prime|x)q(x\prime|x)$, that is we accept the sample with probability $\alpha(x\prime|x)$ times the probability of proposing this sample $q(x\prime|x)$.

2. **Reject** $x\prime$ with probability $r(x) = 1 - \int_\Omega \alpha(x\prime|x)q(x\prime|x)\ dy$, this is just 1 minus the probability accepting any sample other than $x\prime$, as a result we integrate over the state space Ω.

We note that $\delta_x(x\prime)$ is the dirac mass defined by:

$$\delta_x(x\prime) = \begin{cases} 1 \text{ if } x = x\prime \\ 0 \text{ if } x \neq x\prime \end{cases} \tag{4.31}$$

Therefore, we need to consider two cases, one when $x = x\prime$, we reject the proposal, and the second when $x \neq x\prime$ when we accept the proposal.

Case 1, if $x = x\prime$ then we have that, $P(x\prime|x)\pi(x) = P(x|x\prime)\pi(x\prime)$, is trivially true since x and $x\prime$ are the same.

Case 2, when $x \neq x\prime$ we need to show that it satisfies the detailed balanced equations:

$$\begin{aligned}
P(x\prime|x)\pi(x) &= \alpha(x\prime|x)q(x\prime|x)\pi(x) \\
&= \min\left\{1, \frac{\pi(x\prime)}{\pi(x)}\frac{q(x|x\prime)}{q(x\prime|x)}\right\} q(x\prime|x)\pi(x) \\
&= \min\left\{q(x\prime|x)\pi(x), \frac{\pi(x\prime)}{\pi(x)}\frac{q(x|x\prime)}{q(x\prime|x)}q(x\prime|x)\pi(x)\right\} \\
&= \min\left\{q(x\prime|x)\pi(x), q(x|x\prime)\pi(x\prime)\right\} \\
&= \min\left\{\frac{\pi(x)}{\pi(x\prime)}\frac{q(x\prime|x)}{q(x|x\prime)}, 1\right\} q(x|x\prime)\pi(x\prime) \\
&= \alpha(x|x\prime)q(x|x\prime)\pi(x\prime) \\
&= P(x|x\prime)\pi(x\prime) \qquad (4.32)
\end{aligned}$$

This is the desired result, since we have shown that it satisfies the detailed balanced equations, this means we have shown the existence of a unique stationary distribution, by definition 4.2.1. Furthermore, if we assume that $P[\alpha = 1] < 1$ then this is sufficient for aperiodicity, this is because we always have some probability of remaining in the same position for an arbitrary number of moves. Likewise if $q(x\prime|x) > 0$ for all $x, x\prime \in \epsilon$ where $\epsilon = \{x|\pi(x) > 0\}$ then this is sufficient for irreducibility [39, p. 273], this is because we are ensured to move from any set to any other set with some positive probability thus we can reach any other set, satisfying the definition of irreducibility. It is also well know Metropolis–Hastings algorithms are Harris recurrent [40], therefore by Theorem 4.3.1, we can ensure that the Metropolis Hastings algorithm converges to $\pi(x)$ for **all** initial samples x_1.

4.6.5 Challenges

To conclude this chapter I will briefly summarize some of the challenges that we are faced with when using the Metropolis Hastings algorithm, in particular convergence, correlation and picking a proposal density.

Convergence

The first challenge is *how* do we determine that we have reached our desired stationary distribution? To ensure that the algorithm is in fact sampling from our the desired target distribution, we usually use a burn-in phase to delete the first B samples, for more details on this go to subsection 4.3.1.

However, even though we are ensured by Theorem 4.3.1 to converge to the stationary distribution of the chain, this poses another question, how long do we take to arrive? Different algorithms have different times to arrive to the desired distribution. As a result this motivates us to search for algorithms that have better mixing times. Since for the same number of iterations N we sample from the distribution quicker.

Correlation

The second challenge is that the samples are usually **correlated**, as we have seen it is usually the case that samples which are close together will be correlated with one another. As a result, this means that they do not reflect the distribution which we are trying to sample from. This problem is inherit in MCMC methods due to the sequential building of the chains. To solve we need to uses proposals that reduce the correlation between the samples. Alternatively we can search for different algorithms that produce less correlated samples, which in turn reduce the asymptotic variance, thus producing a smaller variance for our estimator given in equation 4.8.

Proposal Density

Finally, we also need to consider which proposal density do we choose? As we have mentioned in subsection 4.6.3, we need to pick a proposal density which finds a balance between cost and accuracy. It is important we understand the type of problem we are dealing with since then we can appropriately pick a proposal density which finds this fine balance between cost and accuracy. In fact, there has been work focusing on adaptive MCMC which focus on improving the algorithm in the fly, by exploring different proposal densities [42].

In fact, there is another type of MCMC methods which help us tackle some of the problems mentioned above, non reversible samplers. The algorithms we mentioned in this chapter create a **reversible** Markov chain. As the name suggests these do not. In the next chapter we will highlight why we would want to use them, the work that has been made in this area and how to implement these algorithm for one dimension and higher dimensions.

To conclude, in this chapter we have introduced Markov chains and motivated the reader as to why they would want to use Markov chain Monte Carlo methods. We then, walked through the Metropolis Hastings algorithm showing how it works and how one can implement the algorithm. Furthermore, we covered various different proposal densities one can use, highlighting the parameters that need to be tuned in each. Finally, we mentioned the challenges faced by the Metropolis Hastings algorithm and proposed a new set of MCMC methods that can help tackle some of these challenges, these will be introduced in the next chapter.

Chapter 5

Non Reversible Samplers

In this section we will introduce non reversible samplers. We will motivate the reader as to why they would like to use these methods. Then we will highlight what the work done so far with respect to these samplers. Finally, we will give a extensive description on how we can implement these algorithms, in one and higher dimensions.

5.1 What Are They?

Recall, that the chains created by the Metropolis-Hastings algorithm are reversible this means that they satisfy the detailed balance equations given in equation 4.7. Therefore, non reversible chains, on the contrary, do not satisfy the detailed balanced equations. As a matter of fact, the chain being reversible is crucial in showing that they have the right stationary distribution. However, it is important to note that if a chain is reversible then this is sufficient but not necessary condition, for $\pi(x)$ to be the stationary distribution of the chain [14]. As a result, we can create non reversible Markov chains such that they can be used as MCMC algorithms since they have the same properties of reversible chains. In particular, Theorem 4.4.1, the Central Limit Theorem, and Theorem 4.3.1, the Markov Chain Convergence Theorem, also apply to non reversible chains. The proofs are different for non reversible chains [48]. This in turn, will let us compare the performance of non reversible samplers against reversible samplers. For example, we can compare the speed of convergence to the equilibrium or the asymptotic variance produced by each algorithm.

5.2 Motivation

So why do we want to use these types of samplers, what benefits, if any do they bring? In fact, there are several reasons as to why we would like to use non reversible samplers. There are some very attractive advantages of using these methods in contrast to reversible samplers.

The first one being, **faster convergence** to desired probability distribution $\pi(x)$, also know as *mixing time* with respect to reversible samplers. This is an extremely desirable advantage, since these methods can reduce the computational time needed reach to $\pi(x)$. The paper, *Irreversible Monte Carlo algorithms for efficient sampling.* [50], shows this statement experimentally. Furthermore, since we can reach the stationary distribution quicker this means that we can reduce the number of samples we discard during the burn-in phase.

Secondly, these algorithms are known to **reduce** the asymptotic variance. This is extremely powerful. In [29] it is shown that the asymptotic variance of non reversible chains are the same as reversible chains, and in many cases the asymptotic variance for non reversible chains are in fact smaller. Therefore, by using non reversible samplers instead of reversible sampler we can increase the accuracy of our estimator by simply changing the type of chain we create.

The reason as to why these chains work so well, is because these chains achieve improvement by avoiding backtracking to the state from which they just came from, by forcing a certain direction of movement. This in turn, reduces the number of samples which are correlated with one another and as a result we see these significant advantages. This will be made clear in the following sections.

5.3 Methods

In this section I will highlight the main methods researched to create non reversible samplers. There are two main ways to construct non reversible chains from reversible ones. The first method is known as the **lifting method**, where one makes “two replica of the original state space with a skew detailed balance condition to facilitate irreversibility” [50].

The second method is by the introduction of a **vorticity function**. In the paper, *Non-reversible Metropolis-Hastings.* by Joris Bierkens [6], it was suggested to turn the MH algorithm into a non reversible version in order to improve convergence. To do this, the paper introduces non reversibility without altering the state space by adding a vorticity function. This function ensures that the chain is non reversible and has the desired stationary distribution $\pi(x)$. More details about this can be found in Bierken’s paper. However, in both cases, it is not clear how to generalize these procedures.

As a result, in this project we will focus on the **I-Jump sampler** for one and higher dimensional cases. In particular, we will follow the algorithms covered in the paper *Irreversible samplers from jump and continuous Markov processes.* by Ma, Fox, Chen and Wu, [25] where they give a general framework to implement non reversible samplers. Please note, that in this project we will implement the algorithm and not dive into the theory for this particular sampler. For more details refer to the paper.

5.4 I-Jump Sampler

In this section we will show how to implement a non reversible sampler. In particular the algorithm will draw inspiration from the MH algorithm. I have decided to implement two variation one for one dimensional cases and one for higher dimensional cases. Note that when using this method it is indeed the case that $\pi(x)$ is the stationary distribution of the chain.

As mentioned before, we will be focusing on the implementing the non reversible sampling algorithms mentioned, called the **I-Jump sampler**. However, we need to understand what is meant by this. In short, we will revise the MH algorithm to make use of different two proposal distributions $f(x\prime|x_i)$ and $g(x\prime|x_i)$, instead of a single $q(x\prime|x_i)$. This *jumping* between different proposal densities, which are not symmetric, forces our chain to move in a certain direction. In turn, this results in our chain not being reversible [25].

5.4.1 One Dimensional

The one dimensional algorithm for non reversible samplers is similar to the MH algorithm. This is desirable due to the simplicity of implementing the algorithm. However, there are two main differences with respect to the MH algorithm.

1. Use of two different **one-sided proposal densities**.
   We will use two proposal $f(x\prime|x_i)$ and $g(x\prime|x_i)$ which will *force* the samples to come a particular direction. We can introduce expressions for the proposals $f(x\prime|x_i)$ and $g(x\prime|x_i)$:

   (a) $f(x\prime|x_i)$ where proposals are given by $x\prime = x_i + \xi$ where $\xi \sim \mathbf{\Gamma}(a, b)$.

   (b) $g(x\prime|x_i)$ where proposals are given by $x\prime = x_i - \xi$ where $\xi \sim \mathbf{\Gamma}(a, b)$.

   To do this we make use of the Gamma distribution $\mathbf{\Gamma}$. This distribution has the desired property that it is strictly positive. As a result, the samples are *restricted* to move a certain direction. This is enforced by the indicator variable $\boldsymbol{z}$ which, in one dimension, picks the direction. If it chooses the proposal $f(x\prime|x_i)$ then we sample points to right of the current sample x_i, likewise, when we sample from $g(x\prime|x_i)$ we move to the left of the current sample x_i. This is the key to irreversibility.

2. Introduction of an **indicator variable $\boldsymbol{z}$**.
   This indicator variable, tells us in what direction the new sample will come from. In one direction, this is simply telling us if the next sample is to the left or the right from the current sample. This indicator variable takes values $\boldsymbol{z} = \{-1, 1\}$. As a result, this variable is the one that cause the *Jump* between the proposal $f(x\prime|x_i)$ and $g(x\prime|x_i)$ thus picking the direction we move in.

To illustrate the algorithm I will use pseudo-code for clarity and then I will explain each key step. *For a Python implementation of this algorithm refer to*

Appendix B.

Algorithm 2: One-Directional I-Jump sampler

```
1  randomly pick $\boldsymbol{z} \in [-1, 1]$;
2  randomly pick $x_1$ such that $\pi(x_1) > 0$;
3  for $i = 1$ to $N_{iter}$ do
4      if $\boldsymbol{z} > 0$ then
5          sample $x' \sim f(x'|x_i)$;
6          $\alpha(x'|x_i) = \min\left\{1, \frac{\pi(x')}{\pi(x_i)} \frac{g(x_i|x')}{f(x'|x_i)}\right\}$;
7      else
8          sample $x' \sim g(x'|x_i)$;
9          $\alpha(x'|x_i) = \min\left\{1, \frac{\pi(x')}{\pi(x_i)} \frac{f(x_i|x')}{g(x'|x_i)}\right\}$;
10     sample $u \sim U[0, 1]$;
11     if $u < \alpha(x'|x_i)$ then $\boldsymbol{z} = \boldsymbol{z}$ else $\boldsymbol{z} = -\boldsymbol{z}$;
12     if $u < \alpha(x'|x_i)$ then $x_{i+1} = x'$ else $x_{i+1} = x_i$;
```

Like with the MH algorithm, before we run the algorithm, we need to define $\pi(x)$ to be the target distribution, that is the distribution we wish to sample from. We also need to pick the number of iteration the algorithm will run for, N_{iter}. However, in this case we will use fixed proposals. One can, however, make use of other one-sided proposal densities.

Notice that step **1** and **2** are similar to the initialization step in the MH algorithm the difference being that we also initialise the indicator variable $\boldsymbol{z}$, where we randomly pick it to be -1 or 1.

Furthermore, the key difference between the MH algorithm is the **if** statement in line **4**. In fact, this step is the key to create non reversible chains. We can see that at steps **5** and **9** we sample from two different proposals $f(x'|x_i)$ and $g(x'|x_i)$. As a result, the acceptance ratio $\alpha(x'|x_i)$ depends on the indicator variable $\boldsymbol{z}$:

1. If $\boldsymbol{z} > 0$ then $\alpha(x'|x_i) = \min\left\{1, \frac{\pi(x')}{\pi(x_i)} \frac{g(x_i|x')}{f(x'|x_i)}\right\}$ when the proposal x' is sampled from $f(x'|x_i)$.

2. If $\boldsymbol{z} < 0$ then $\alpha(x'|x_i) = \min\left\{1, \frac{\pi(x')}{\pi(x_i)} \frac{f(x_i|x')}{g(x'|x_i)}\right\}$ when the proposal x' is sampled from $g(x'|x_i)$.

The proposals are then given by $x' = x_i + \xi$ when we sample from $f(x'|x_i)$. We know that $x' - x_i = \xi$, as a result we have that $x' - x_i \sim \boldsymbol{\Gamma}(a, b)$ since $\xi \sim \boldsymbol{\Gamma}(a, b)$. Therefore, if substitute $x' - x_i$ into the probability density function for the gamma distribution Ξ, given by:

$$\Xi(x) = \frac{b^a}{\Gamma(a)}(x)^{a-1}e^{-bx} \tag{5.1}$$

Where in this case $\Gamma(a)$ is the gamma function. We can get an explicit form of this distribution given by the following expression:

$$\Xi(x' - x_i) = \frac{b^a}{\Gamma(a)}(x' - x_i)^{a-1}e^{-b(x' - x_i)} \tag{5.2}$$

Therefore, we have that $f(x\prime|x_i) = \Xi(x\prime - x_i)$, this is extremely useful when implementing these algorithms, since we need to evaluate this function for the acceptance ratio $\alpha(x\prime|x_i)$. Notice that is is just the gamma distribution centred x_i and evaluated at point $x\prime$.

Using a similar logic we can derive an expression for $g(x\prime|x_i)$. Since we know that $x\prime = x_i - \xi$ then we have that:

$$x\prime - x_i = -\xi \iff x_i - x\prime = \xi \iff x_i - x\prime \sim \Gamma(a, b) \tag{5.3}$$

Therefore, by substituting $x_i - x\prime$ into the probability density function Ξ given in equation 5.1 we have that $g(x\prime|x_i) = \Xi(x_i - x\prime)$.

In addition, notice that in line **11** we switch the sign of the indicator variable $\boldsymbol{z}$, the direction we are sampling from, when we reject the proposal sample $x\prime$, thus in the next iteration we are **always** proposing samples in the opposite direction.

Finally, like in the MH algorithm, in line **12**, the chain can either remain at the sample x_i if it rejects $x\prime$ or moves to the proposed sample $x\prime$ upon acceptance, with probability $\alpha(x\prime|x_i)$. That is, in the next iteration $x_{i+1} = x\prime$, if we accept $x\prime$ otherwise $x_{i+1} = x_i$.

Furthermore, it is important to notice that we have control over the variables a and b of the gamma distribution $\boldsymbol{\Gamma}$. Appropriately tuning these parameters will let us control the step size taken by the algorithm. In section 6.2 we will show how to tune these parameter appropriately.

5.4.2 High Dimensional

In this section we show how to generalize I-Jump sampler for higher dimensions. Firstly, we introduce the d-dimensional auxiliary variable $\boldsymbol{y}$. This variable, is simply one of the possible d basis vectors in the dimension of state space. For example, if the number of dimensions is three, then $\boldsymbol{y}$ can take values as the basis vectors. That is $[1, 0, 0], [0, 1, 0], [0, 0, 1]$. This will ensure that the proposal densities sample in the direction of $\boldsymbol{y}$, this is vital in high dimensions.

Furthermore, we also need to introduce two different proposal densities we will call them $f(x\prime|x_i, \boldsymbol{y})$ and $g(x\prime|x_i, \boldsymbol{y})$, which depend on d-dimensional auxiliary variable $\boldsymbol{y}$. f and g will be half-space Gaussian distributions which conserve the properties of the stationary distribution while having faster mixing times [25]. The proposal densities are given by:

$$f(x\prime|x_i, \boldsymbol{y}) = \mathbf{1}_{<\boldsymbol{y}, x\prime - x_i> \geq 0} \frac{2}{(2\pi\sigma)^{(d/2)}} e^{-\frac{1}{2\sigma}||x\prime - x_i||_2^2} \tag{5.4}$$

$$g(x\prime|x_i, \boldsymbol{y}) = \mathbf{1}_{<\boldsymbol{y}, x\prime - x_i> < 0} \frac{2}{(2\pi\sigma)^{(d/2)}} e^{-\frac{1}{2\sigma}||x\prime - x_i||_2^2} \tag{5.5}$$

Where the expression $< \, , \, >$, we are referring to the **dot product**. Furthermore, $\mathbf{1}_{cond}$ refers to the **indicator function**, where if *cond* is satisfied then it is 1 otherwise 0. Finally, d refers to the dimension of the state space we are working with, when we sample points from $f(x\prime|x_i, \boldsymbol{y})$ this is equivalent to:

$$x\prime = x_i + \eta \cdot sgn(<\eta, \boldsymbol{y}>), \eta \sim \mathcal{N}(0, \sigma^2 I) \tag{5.6}$$

Likewise for $g(x\prime|x_i, \boldsymbol{y})$ is:

$$x\prime = x_i - \eta \cdot sgn(<\eta, \boldsymbol{y}>), \eta \sim \mathcal{N}(0, \sigma^2 I) \tag{5.7}$$

Where sgn is defined as $sgn(x) = 2\mathbf{1}_{x \geq 0} - 1$. Furthermore, these half spaced Gaussian distributions behave in a similar fashion to the one sided proposal densities we suggested in the one dimensional case. In both cases we restrict the sample to exclusively a single direction. Note that we can also implement this algorithm with the use of one-sided distributions for proposals, like the gamma distribution, instead of half space Gaussian distributions.

We need to take into account one last aspect of this algorithm which is how often do we change direction $\boldsymbol{y}$? To overcome this we periodically pick a random direction to move in. This is to ensure that the algorithm covers all dimensions of the state space. There is no defined period one should re-sample this variable for. To find this period it is usually done in ad-hoc fashion, and needs to tuned for the particular problem. Putting this all together we can define the **N-Directional I-Jump sampler**. Below is the pseudo code for this algorithm. *For a Python implementation of this algorithm refer to Appendix B.*

Algorithm 3: N-Directional I-Jump sampler

```
1  randomly pick $\boldsymbol{y} \in \mathbb{R}^n$;
2  randomly pick $x_1$ such that $\pi(x_1) > 0$;
3  for $i = 1$ to $N_{iter}$ do
4      periodically randomly sample $\boldsymbol{y} \in \mathbb{R}^n$;
5      if $\boldsymbol{z} > 0$ then
6          sample $x\prime \sim f(x\prime|x_i, \boldsymbol{y})$;
7          $\alpha(x\prime|x_i) = \min\left\{1, \frac{\pi(x\prime)}{\pi(x_i)} \frac{g(x_i|x\prime,\boldsymbol{y})}{f(x\prime|x_i,\boldsymbol{y})}\right\}$;
8      else
9          sample $x\prime \sim g(x\prime|x_i, \boldsymbol{y})$;
10         $\alpha(x\prime|x_i) = \min\left\{1, \frac{\pi(x\prime)}{\pi(x_i)} \frac{f(x_i|x\prime,\boldsymbol{y})}{g(x\prime|x_i,\boldsymbol{y})}\right\}$;
11     sample $u \sim U[0, 1]$;
12     if $u < \alpha(x\prime|x_i)$ then $\boldsymbol{z} = \boldsymbol{z}$ else $\boldsymbol{z} = -\boldsymbol{z}$;
13     if $u < \alpha(x\prime|x_i)$ then $x_{i+1} = x\prime$ else $x_{i+1} = x_i$;
```

Notice that this algorithm is the very similar to Algorithm 2, where the differences are due to the introduction of the auxiliary variable $\boldsymbol{y}$ and the need to periodically change $\boldsymbol{y}$ in line **4**. Furthermore, we make use of the new proposals $f(x_i|x\prime, \boldsymbol{y})$ and $g(x_i|x\prime, \boldsymbol{y})$ defined in equations 5.4 and 5.5 respectively. Finally, its worth mentioning, that it can be seen that in both settings, that is with one-sided distributions and half-space Gaussian, "there is always a direction of exploration, $\boldsymbol{y}$, that enjoys most of the benefits from irreversibility. However, in multiple dimensions, a favorable direction of exploration is often not clear." [25], as we shall this will pose to be a problem in high dimensions. I have decided to not walk through the algorithm again in depth since it is very similar to Algorithm 2 please refer to subsection 5.4.1 for a more detailed explanation of each step of the algorithm.

To conclude, in this section we have introduced non reversible samplers highlighting the advantages of using these methods and showing the simplicity of implementing these algorithms. In the next chapter, we will run experiments to show how these algorithms preform against reversible samplers on different problems. We will also show how to tune the MH algorithm and how to tackle some of the challenges faced the by algorithm, as discussed in section 4.6.5.

Chapter 6

Exploring Markov Chain Monte Carlo Methods

In this chapter we will explore Markov Chain Monte Carlo Methods (MCMC). We will first inspect the Metropolis Hastings (MH) algorithm to highlight important aspects of MCMC methods such as; appropriately tuning parameters, finding a burn-in phase or deciding which proposal density to use. Subsequently, we will introduce the N-Directional I-Jump sampler. Finally, we will then compare the performance of the MH algorithm against the N-Directional I-Jump sampler on a range of target distributions.

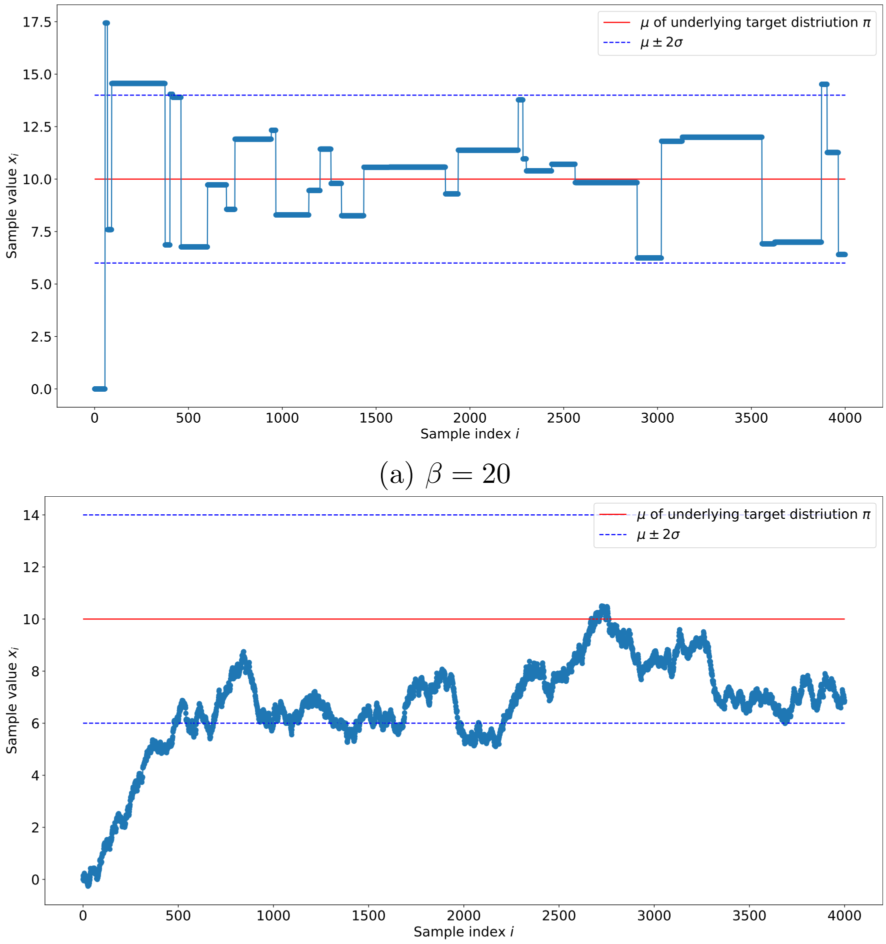
Before going further, I want to emphasize, that in this chapter, we will be plotting the *normalized autocorrelation function* (referred to as ACF) $\varrho_{\varphi}(\tau)$, given by definition 4.5.2, against different time lags τ. We interested in the **rate** at which $\varrho_{\varphi}(\tau)$ decays. Recall, that better preforming algorithms will have a greater rate of decay. Throughout this chapter, φ will simply be the identity I since we are interested in the samples produced. More details are in section 4.5.

6.1 Metropolis Hastings Algorithm

We will begin by using the Metropolis Hastings (MH) algorithm with a Random Walk proposal density to illustrate the following. First, how to appropriately tune β for this problem. Secondly, how to find an optimal *burn-in* value for this problem. Thirdly, to show that the samples are indeed distributed according to the target distribution $\pi(x)$. Finally, we will show how to pick a proposal density for the algorithm. Throughout this section the **target distribution** $\pi(x)$ will be $\mathcal{N}(x|10, 2)$ distribution.

6.1.1 Tuning Beta

To recall, the Random Walk proposal density, given by equation 4.26, proposes samples in the following form $x\prime = x_i + \beta\epsilon$ where $\epsilon \sim N(0, I)$. Notice that β controls the step size of the algorithm. As mentioned in subsection 4.6.3 we need to tune this parameter for the particular problem. Both small and large values of β will pose to be a problem to explore the state space appropriately.



(a) $\beta = 20$

(b) $\beta = 0.5$

Figure 6.1: Figures show how the chain evolves over time for a small and large value for β. Using the MH algorithm with a Random Walk proposal to sample from $\mathcal{N}(x|10, 2)$. The x-axis is the sample index, this is the index of the sample at iteration i. The y-axis is the value which the sample x_i takes.

We begin by comparing the chains produced by small and large values of β as seen in Figure 6.1. We can see that when β is large, Figure 6.1 (a), there are various occurrences of *horizontal lines*. This is telling us that the algorithm is staying at the same sample for a certain number of iterations. But why is this case? We can explain this by looking at the equation for the acceptance ratio α:

$$\alpha(x\prime|x_i) = \min\left\{1, \frac{\pi(x\prime)}{\pi(x_i)}\right\} \tag{6.1}$$

In this case the acceptance ratio is given in this form since the proposal density is *symmetric*. Notice, that when we have a **large** step size then in many case $\pi(x\prime)$ is close to 0 because we take a large step away from the target distribution, thus moving to space of less density as a result $\frac{\pi(x\prime)}{\pi(x_i)}$ is also close to 0, this causes $\alpha(x\prime|x_i)$ to be small and as a result we more likely to reject samples, thus remaining at the current sample x_i, resulting in horizontal lines.

Similarly, if we focus on Figure 6.1 (b) where β is small, we can see that there are no *horizontal lines*. To explain this we focus our attention again on the acceptance ratio equation 6.1. In many cases $\pi(x\prime)$ is large since we are taking small step, from a sample which has been accepted, so we are likely to move towards areas of high density, in turn the acceptance ratio is large, so we are more likely to accept samples. So we see no *horizontal lines*. Furthermore, we can see that at the start the chain is indeed always moving towards the target distribution $\pi(x)$. Again, this can be explained by looking at the acceptance ratio 6.1. In this example, the chain starts at point $x_0 = 0$, when it propose samples $x\prime$ such that they move towards towards $\pi(x)$ we have that $\pi(x\prime) > \pi(x_i)$ (since we are starting at $x_0 = 0$) and so $\frac{\pi(x\prime)}{\pi(x_i)}$ is greater than 1. As a result, we are more likely to accept samples that move towards $\pi(x)$ rather than away from it.

Recall, that the main goal is to sample from the target distribution $\pi(x)$ where $\pi(x)$ is illustrated in Figure 6.1 (a) and (b) by the red line indicating the mean of the distribution in this case 10. The blue dotted lines indicate two standard deviations away from the mean. We can see that both step sizes do indeed converge to the target distribution. However, what is important to note is the **speed** at which they converge. We can see that the large value of β converges quicker since it can take larger step size in the state space in contrast to the small β which converges slower since it accepts many proposals. Therefore, there is a need to find a balance between accepting proposals and quick convergence.

As a result, this motivates us to find ways of appropriately tuning this parameter β in order to ensure that our samples are not correlated and that we convergence rapidly to the target distribution $\pi(x)$. Since choosing an arbitrary value, as we have seen above, can lead to convergence and correlation issues. To tune the parameter β we can look into the theory.

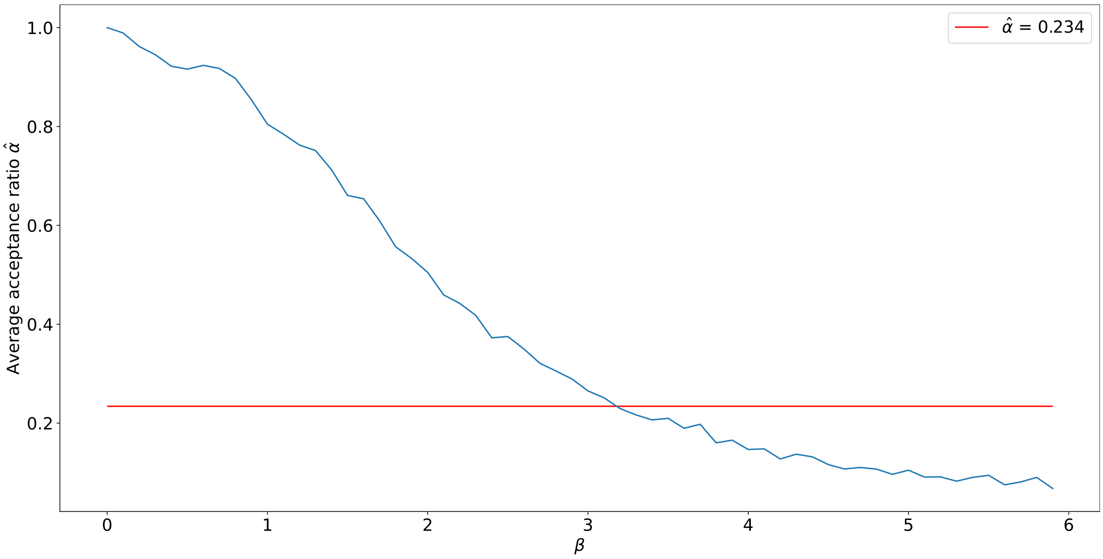


Figure 6.2: Finding β for $\mathcal{N}(10, 2)$ target distribution, by plotting average acceptance ratio $\hat{\alpha}$ against β. In this case we tried 60 different values of β ranging from 0 to 6, and for each we ran the MH algorithm for 1000 iterations.

A general rule of thumb is to tune β such that the *average acceptance ratio* $\hat{\alpha} \approx 0.234$ [16], in order to achieve small asymptotic variance when using a Random Walk proposal. To find this value, we will run a series of simulations with different values of β to find the value of β such that $\hat{\alpha}$ is closest to 0.234.

Figure 6.2 shows us how $\hat{\alpha}$ changes for different values of β. In this case the red horizontal line shows us $\hat{\alpha} = 0.234$. We can see that when $\beta \approx 3.2$ it is closest to the red line, thus we will use $\beta = 3.2$ as our tuned parameter. This step size finds a balance between accepting proposals and exploring the target distribution. Below I have illustrate the differences at which the rate of the ACF decays for when beta is tuned (3.2), small (0.3) and large (20).

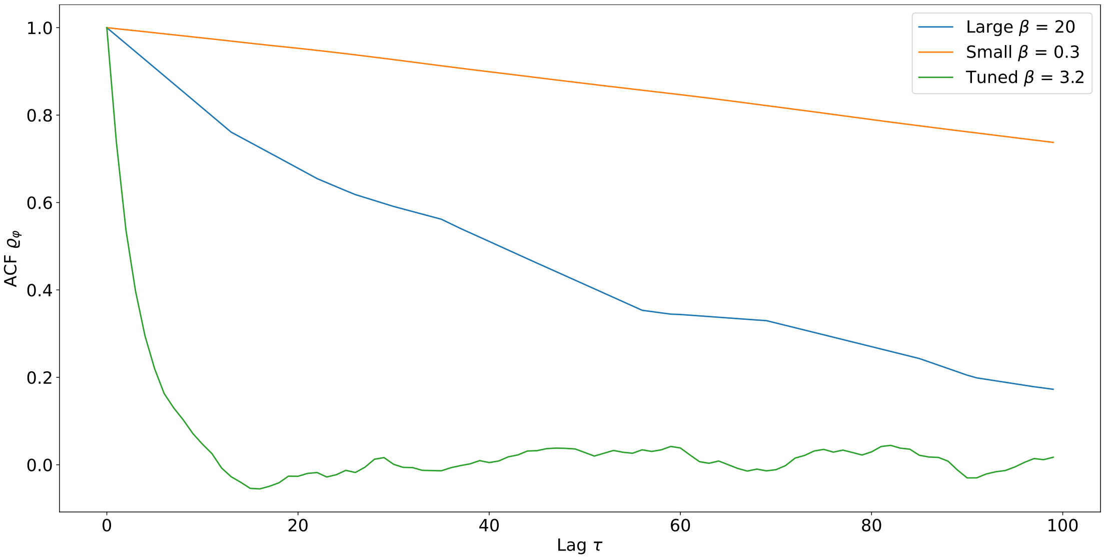


Figure 6.3: Plotting ACF ϱ_φ against lag τ, for small, large and tuned β values.

Figure 6.3 shows the rate at which ACF decreases for the three different β parameter values. It is clear that when the parameter is appropriately tuned we can see a high rate of decay. In fact, by lag $\tau = 20$, the algorithm is producing samples with very small correlation. This is what we would like since the variance of the estimator we are trying to calculate will be smallest for the tuned β. Recall that the variance is $\frac{\sigma_\varphi^2}{N}$ where σ_φ^2 is the asymptotic variance 4.4.1. As a result, if we produce less correlated samples then σ_φ^2 is smaller.

What is interesting is the rate at which the ACF decreases is greater for large step sizes than for small step sizes, *why is this the case?* By looking at Figures 6.1 (a) and (b) they can give us an idea why. For large β the chain is more likely to remain at the same sample, as seen by the *horizontal lines*, which means the correlation between elements for **small** time lags is large. However as soon as we accept a new proposal $x\prime$ it moves to a very different value and remains there for a certain number of steps, thus for larger lags the correlation between the samples decreases. In contrast to the small step size we can see that the samples are very close to one another. In fact we can see that the chain does not move far from the current position after a large number of iterations. In Figure 6.1 (b) $x_0 = 0$ and $x_{500} \approx 5$ this means that in 500 iterations the value of the samples has only varied between 0 and 5 thus the samples highly correlated. This example has highlighted the necessity of appropriately tuning β for exploration of the state space and the reduction of correlation between of samples.

6.1.2 Burn-in Phase

We will show how to find the burn-in phase. In particular we use two methods to find it. Furthermore, in this example we will let $\beta = 0.3$, a small step size, in order for slower convergence to the target distribution $\pi(x)$. By using a small step size the algorithm will require a larger number of step to reach the desired target distribution, therefore a burn-in phase is vital. The first method to find the burn-in phase will be by using the following heuristic: *discard all the samples up to the first sample that falls within two standard deviations from the mean.*

From Figure 6.4 we can see that the first sample that falls within two standard deviations of the mean, is x_{546}, as indicated by the green vertical line. As a result, we will discard the first 545 samples from the chain, and use the remaining values to estimate the target distribution. We will now compare the estimated target distribution with and without the burn-in phase.

In Figure 6.5 (a) we have no burn-in. We can see a concentration of density on the left tail. In contrast, when we discard the first 545 samples, the estimated target distribution has less density between 0 and 3.5, as seen in Figure 6.5 (b). This is desirable, because the analytical target distribution, $\mathcal{N}(x|10, 2)$, which we are trying to sample from has low probability density in the tails, so discarding them is appropriate.

However, we can still see that this method of finding the burn-in period for the samples is not sufficient to produce a good estimate for the target distribution $\pi(x)$. We can see that the left tail still has a high density, and as a result we need to find alternatives ways of taking this into account.

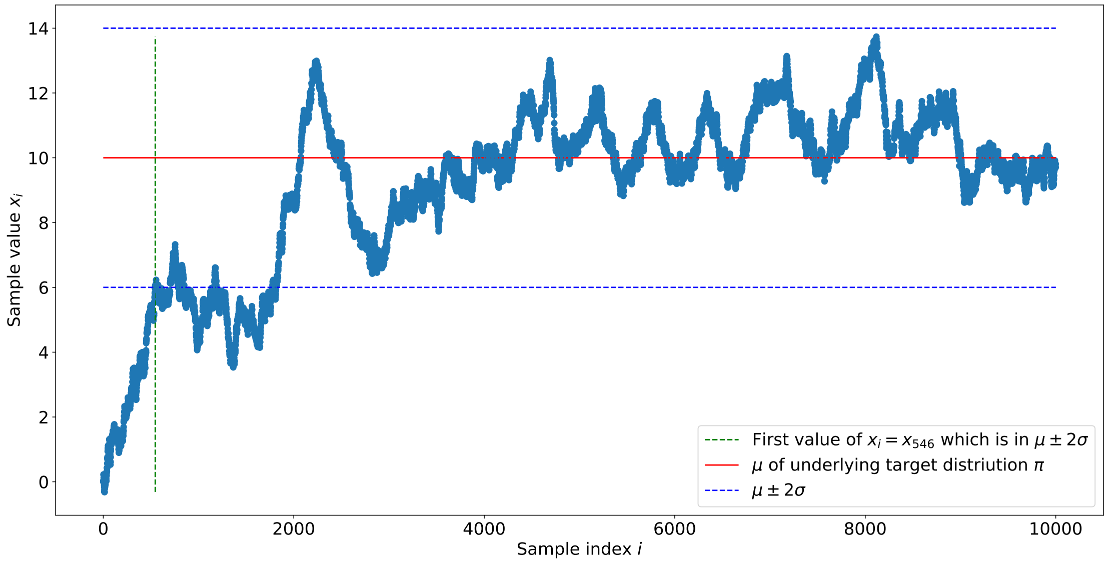
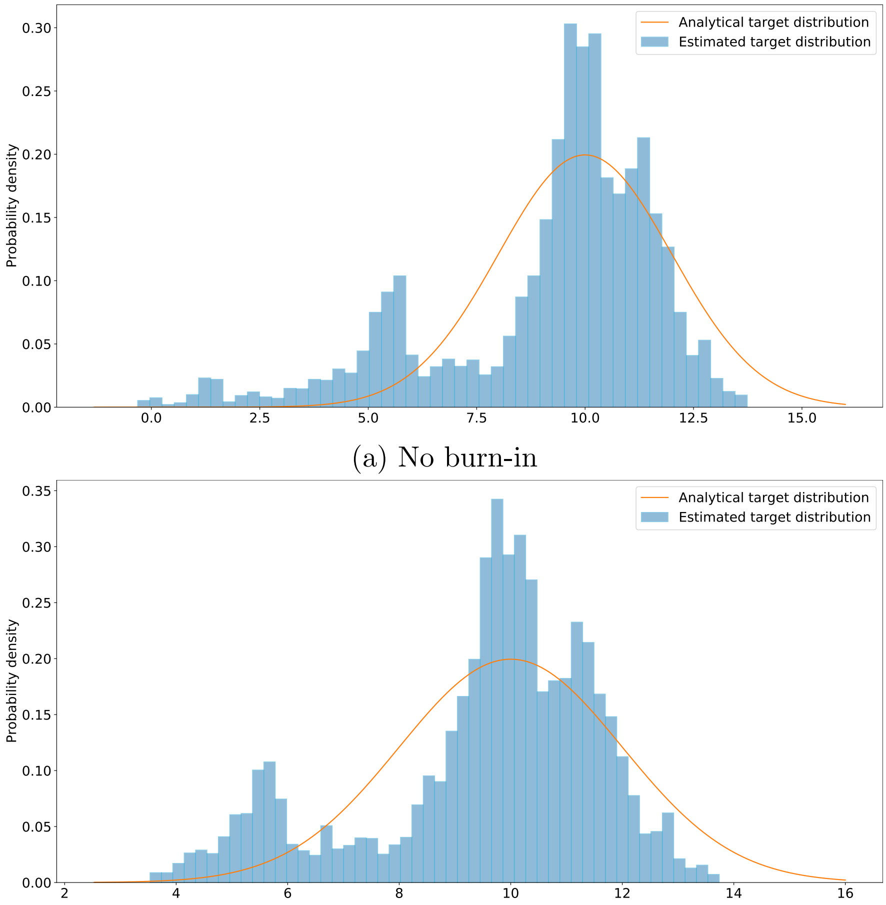


Figure 6.4: Using the MH algorithm with a Random Walk to sample from $\mathcal{N}(10, 2)$ when $\beta = 0.3$, the algorithm ran for 10000 iterations. This figure shows how the chain evolves over time.



(a) No burn-in

(b) Burn-in

Figure 6.5: Estimated target distribution for $\mathcal{N}(10, 2)$ when $\beta = 0.3$, when using no burn-in (a) and burn-in (b).

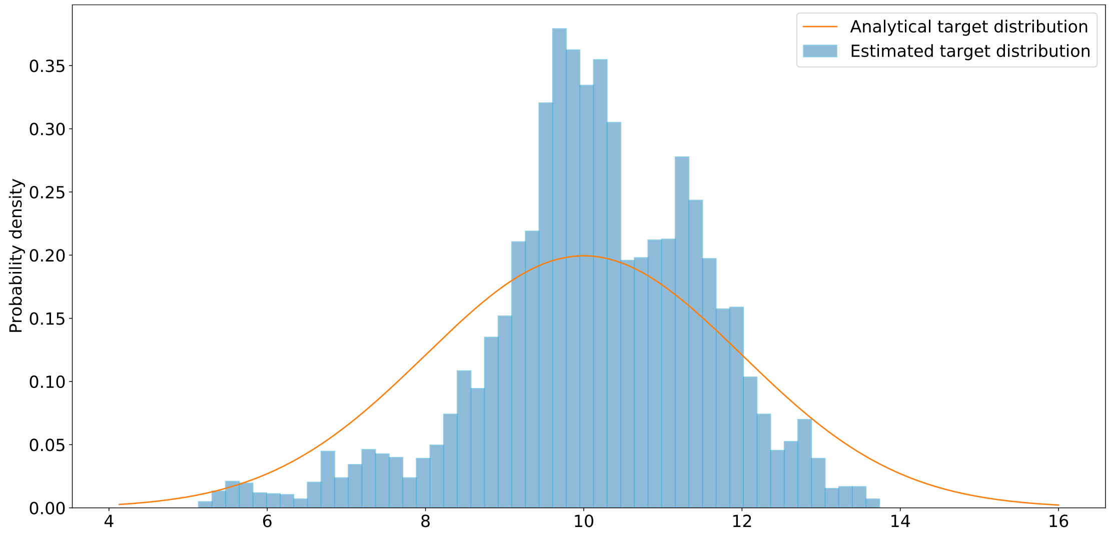


Figure 6.6: Estimated target distribution from $\mathcal{N}(10, 2)$ when $\beta = 0.3$, by inspection. In this case we discarded the first 1800 samples.

To do this, we can find the burn-in phase by taking this one step further and inspecting the chain. We can see in Figure 6.4, that even though the chain enters two standard deviations of the mean at sample index 546, at around sample 2000 it still moving outside of blue dotted line. As a result, we would like eliminate more of the samples on the left tail. By inspection we increase the burn-in phase and discard the first 1800 samples from the chain. In Figure 6.6 we can see that estimated target distribution has a better approximation to our target distribution $\pi(x)$, when we eliminate 1800 samples. Now the samples are centered around the mean as desired and the tails have less density. However, since we are taking small step sizes the density of the estimated distribution is concentrated in the centre of the distribution and does not explore the tails. Again, this highlights the importance of discarding samples but also tuning an appropriate parameter β.

In this section, we have shown the difficulty of finding an appropriate burn-in phase. By introducing two methods to find such a phase. However, as we have seen no one method works for all problems. Therefore, it is vital to examine the chain produced on a case by cases basis.

6.1.3 Convergence

In this section, we will illustrate the convergence of the chain by showing that as we increase the number of iterations our samples convergence to the target distribution $\pi(x)$ and we are in fact sampling from it. This is to show that Theorem 4.3.1 holds and that the stationary distribution of the chain created is in fact the target distribution $\pi(x)$ for the MH algorithm with a Random Walk proposal density. I will use the tuned MH algorithm where $\beta = 3.2$.

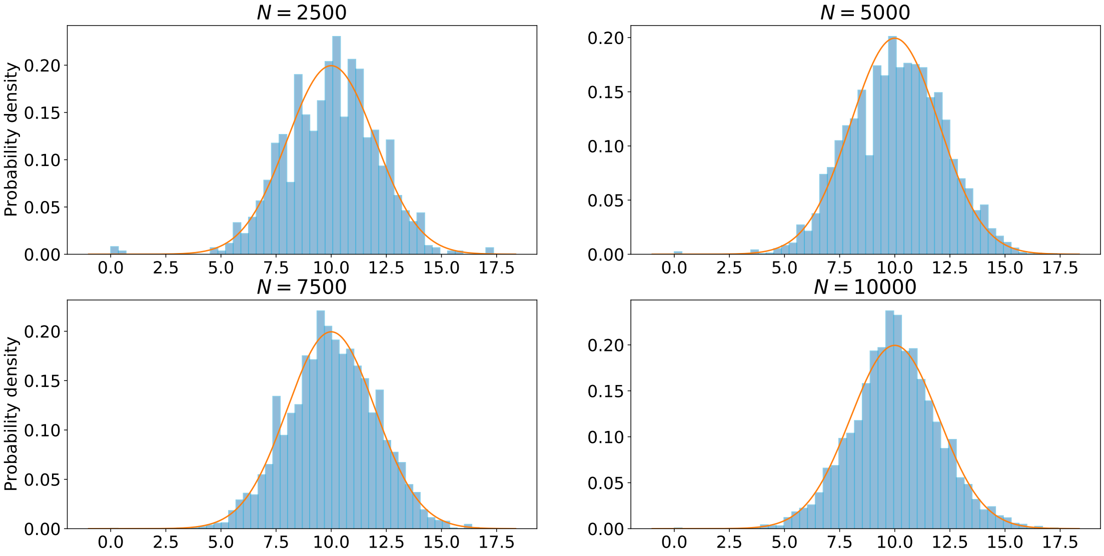


Figure 6.7: Estimated target distribution for $\mathcal{N}(x|10, 2)$, when $\beta = 3.2$. We vary N, the number of iterations, to illustrate the convergence to the target distribution.

We can from Figure 6.7 see that as N, the number of iterations, goes to infinity we have that the estimated target distribution tends to analytical target distribution $\pi(x)$. As a result, we can conclude that the MH algorithm with a Random Walk proposal, converges to the desired distribution. This is what we expect to see since we know by Theorem 4.3.1, that as the number of iterations tends to infinity the samples are distributed according to the stationary distribution of the chain.

6.1.4 Pre-conditioned Crank-Nicolson

In this section we will compare the performance of the Random Walk proposal against the Pre-conditioned Crank-Nicolson (pCN). To decided which proposal to pick for this problem. I will use the tuned MH algorithm with $\beta = 3.2$, we found in subsection 6.1.1. We will take it one step further by finding the burn-in phase for the tuned parameter. To do this we will use the same heuristic in subsection 6.1.2, *discard all the samples up to the first sample that falls within two standard deviations from the mean.*

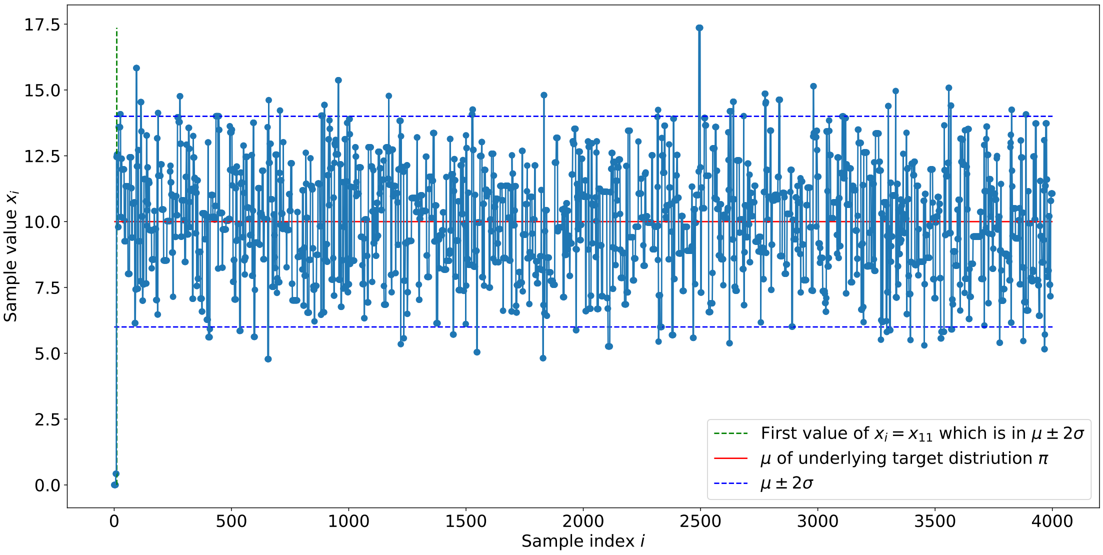


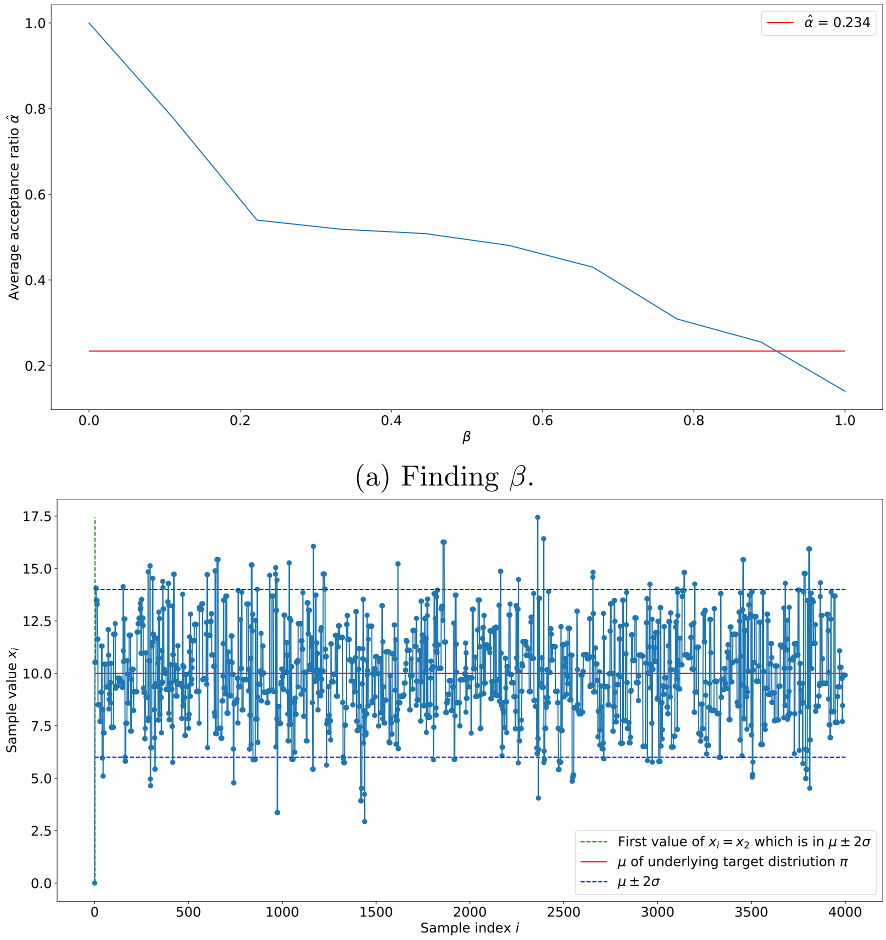
Figure 6.8: Using MH algorithm with a Random Walk proposal to sample from $\mathcal{N}(x|10, 2)$ when $\beta = 3.2$. This plot helps us find the burn-in phase for the algorithm. The algorithm ran for 4000 iterations.

We can see form Figure 6.8 that the first sample that is within two standard deviations of the mean is x_{11}. As a result, we will discard the first 10 samples when we compare these samples with the ones produced with the pCN proposal.

We will now tune the parameters for the pCN proposal density before being able to compare them. Recall that when using the pCN proposal density, the proposals are in the form $x\prime = \sqrt{1-\beta^2}x_i + \beta\epsilon$ where $\epsilon \sim \mathcal{N}(0, C_0)$. Again, we need to tune β.

Furthermore, we let C_0 be equal to 8. We will let this value be large, so the variance of proposal distribution is too. Thus, allowing allow for larger step sizes. Similar to before we tune β such that the average acceptance ratio $\hat{\alpha} \approx 0.234$ [36]. As mentioned in subsection 4.6.3, β can only take values between 0 and 1.

We can see from Figure 6.9 (a) that β between 0.8 and 1.0 is closest to the desired $\hat{\alpha}$ of 0.234. This is consistent with the theory where we would expect $\beta \approx 1$ [16]. What is happening for the proposal $x\prime = \sqrt{1-\beta^2}x_i + \beta\epsilon$, is as β gets closer to 1 then $\sqrt{1-\beta^2}x_i$ goes to zero. Therefore $x\prime$ depends on the ϵ variable, thus C_0 has more influence over the step size. We pick β to be equal to 0.9, for the remaining analysis. Figure 6.9 (b) shows the chain produced by using the MH algorithm with pCN proposal density.



(a) Finding β.

(b) Chain produced with pCN proposal for 4000 iterations.

Figure 6.9: These are the figures produced when using the pCN proposal. (a) shows how to tune β and (b) is the chain produced by using the MH algorithm with pCN to sample from $\mathcal{N}(x|10,2)$ when $\beta = 0.9$.

From Figure 6.9 (b), can see that the first sample that is within two standard deviations of the mean is x_2. Therefore, in this case we will only discard the first sample.

Comparison

Let $\{x_i\}_{RW}$ be the chain produced by the Random Walk proposal density and let $\{x_i\}_{pcn}$ be the chain made using the pre-conditioned Crank-Nicolson proposal density.

To compare them I will first use a simple measure, **percentage error**, of the mean of the samples in the chain with respect to actual mean. We know that by Theorem 4.4.1 that the Ergodic average is asymptotically normally distributed with mean of the target distribution $\pi(x)$, so we should expect both of these values close to 10. Function of interest φ is the identity map. We first discard the values from the burn-in phase. Then we can calculate the mean for each method. We get the following results for each chain (4 s.f):

$$E[\{x_i\}_{RW}] = 10.04,\ \ E[\{x_i\}_{pcn}] = 9.922 \tag{6.2}$$

With the respective percentage error as:

$$\%\text{ error } E[\{x_i\}_{RW}] = 0.387,\ \ \%\text{ error } E[\{x_i\}_{pcn}] = 0.784 \tag{6.3}$$

We can see that both methods are extremely accurate, both within 1% of the actual mean, when the parameters are properly tuned and taking into account the burn-in phase.

What is left to see is how correlated the samples are for each of the different methods. For this we will plot the ACF against lag τ, like we did in subsection 6.1.1.

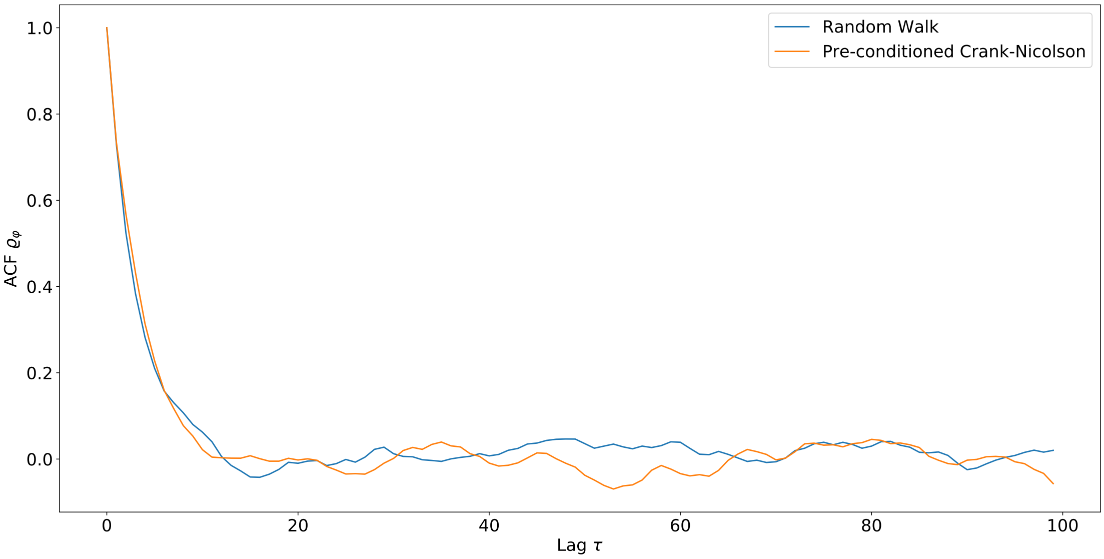


Figure 6.10: Plotting ACF ϱ_φ against lag τ, for tuned pCN and RW proposals.

We can see from Figure 6.10 that both methods behave similarly. The rate of decrease is the similar for both proposal densities for this problem. This is expected since the benefits of pCN become more prominent when dealing with higher dimensions [18]. Since the problem is one-dimensional both preform equally well. Furthermore, the convergence speed for both algorithms is also similar since both produce samples from the target distribution within 10 samples. As a result, in this case, we can pick either proposal for this problem, since it is relatively simple. Note that this analysis can be generalized for different problems when deciding on which proposal to pick.

We have now covered the Metropolis Hastings algorithm we will now move on to the One-Directional I-Jump sampler.

6.2 One-Directional I-Jump Sampler

Very much like the Metropolis Hastings (MH) algorithm, the One-Directional I-Jump sampler also has parameter which need to be tuned. In this section we will discuss how to tune the parameters mentioned in subsection 5.4.1. I will propose several different ways to do this. Once the parameters have been chosen we will compare the performance of this algorithm against the MH algorithm on the target distribution $\pi(x) = \mathcal{N}(x|10, 2)$.

6.2.1 Tuning Parameters

Parameter tuning for MCMC algorithms is vital to for good exploration. As we have seen before, these parameters vary depending on the algorithm. We will show how to tune these parameters for the One-Directional I-Jump sampler.

Before finding these parameters we need to understand *which* parameters we are tuning. Recall, that proposals are given by $x\prime = x_i + \xi$ or $x\prime = x_i - \xi$ where $\xi \sim \Gamma(a, b)$. Therefore, we need to tune the parameters a and b. a is know as shape parameter and b as the rate parameter, both strictly greater than 0 when using this characterisation of the gamma distribution, the mean is equal to $\frac{a}{b}$. This will come in useful when analysing the step size of the algorithm.

We will fix a to be equal to 1. In order to only have to tune one variable, b. We will start by using the rule of thumb for the MH algorithm, aiming for b such that the average acceptance ratio $\hat{\alpha} \approx 0.234$, as a starting point. However, as we shall see, we challenge this value for non reversible samplers.

From Figure 6.11, we can see that for large values of b the $\hat{\alpha}$ increases. Values of b between 0 and 0.2 are closest to the desired $\hat{\alpha} \approx 0.234$. Therefore, if we let $b \approx 0.13$, then the average step size is $\frac{1}{0.13} \approx 7.69$, since it is the expected value of ξ.

To understand the Figure 6.11 we need to look at the step size. We know that the mean of the random variable ξ is $\frac{a}{b}$. Since a is fixed to 1, the step size is given by $\frac{1}{b}$. Thus, larger values of b will result in smaller step sizes, so we are more likely to accept the proposal sample $x\prime$. Therefore, as b increases $\hat{\alpha}$ should too, as seen by Figure 6.11.

We can now compare the this step size with the step size taken by the MH algorithm using a Random Walk proposal density. To find the step size, we need to first look at the expression of the proposal sample $x\prime$. Recall that $x\prime$ is given by $x_i + \beta\epsilon$ where $\epsilon \sim \mathcal{N}(0, I)$. Therefore, the step is given by $S = \beta\epsilon$ where $S \sim \mathcal{N}(0, \beta^2 I)$. To find the step size of the random variable S we look at the *folded normal distribution.* Since we would like to find the absolute value of each of our steps.

Given a normally distributed random variable X with mean μ and variance σ^2, the random variable $Y = |X|$ has a folded normal distribution [49]. The

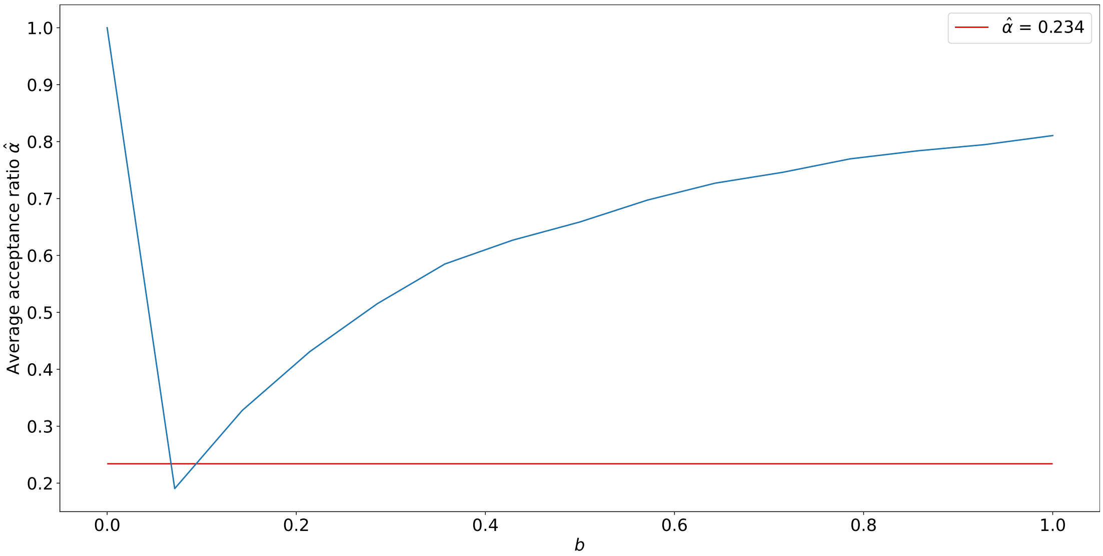


Figure 6.11: Plot of average acceptance ratio $\hat{\alpha}$ against b. This is used to the parameter b for $\mathcal{N}(x|10,2)$ target distribution, using One-Directional I-Jump sampler. We considered 20 different values of b from 0 to 1, and for each using a One-Directional I-Jump sampler with 1000 iterations.

expected value for the folded normal distribution is given by [49]:

$$E_Y = \sqrt{\frac{2}{\pi}}\sigma e^{-\frac{\mu^2}{2\sigma^2}} + \mu\left[1 - 2\Phi\left(-\frac{\mu}{\sigma}\right)\right] \tag{6.4}$$

Where, Φ is the normal cumulative distribution function. This is exactly what we are interested in. In our case the random variable $X = S$ where S has $\mu = 0$ and $\sigma = \beta$. If we substitute our values for μ and σ into 6.4. We can simplify the expression to:

$$E_Y = \sqrt{\frac{2}{\pi}}\beta e^{-\frac{0^2}{2\sigma^2}} + 0\left[1 - 2\Phi\left(-\frac{\mu}{\sigma}\right)\right] = \sqrt{\frac{2}{\pi}}\beta \tag{6.5}$$

Therefore, the step size for the MH algorithm when using a Random Walk proposal density is equal to $\sqrt{\frac{2}{\pi}}\beta$. So, when $\beta = 3.2$ its step size is $\sqrt{\frac{2}{\pi}}(3.2) \approx 2.55$.

When comparing the step sizes for the MH algorithm (2.55) against the One-Directional I-Jump sampler (7.69) we can see that the step size is almost 3 time larger than the MH algorithm. This raises the question, is $\hat{\alpha} \approx 0.234$ appropriate for non reversible samplers?

To answer this question we can investigate three different values for b.

1. $b = 0.13$, this is the b value such that $\hat{\alpha} \approx 0.234$, with step size $\frac{1}{0.13} \approx 7.69$.

2. $b = 0.4$, this value ensures that both samplers have the **same** step size at 2.55. To calculate this value we simply let the two expected step sizes be the same, that is $\sqrt{\frac{2}{\pi}}(3.2) = \frac{1}{b}$ therefore, $b = \frac{1}{\sqrt{\frac{2}{\pi}}(3.2)} \approx 0.39$.

3. $b = 0.6$, this is to get another step size which is smaller than 1. and 2. in this case the step size is $\frac{1}{0.6} \approx 1.67$.

To compare the different step sizes, and as a result different b values, we plot the ACF against time lags τ, for each of these values of b.

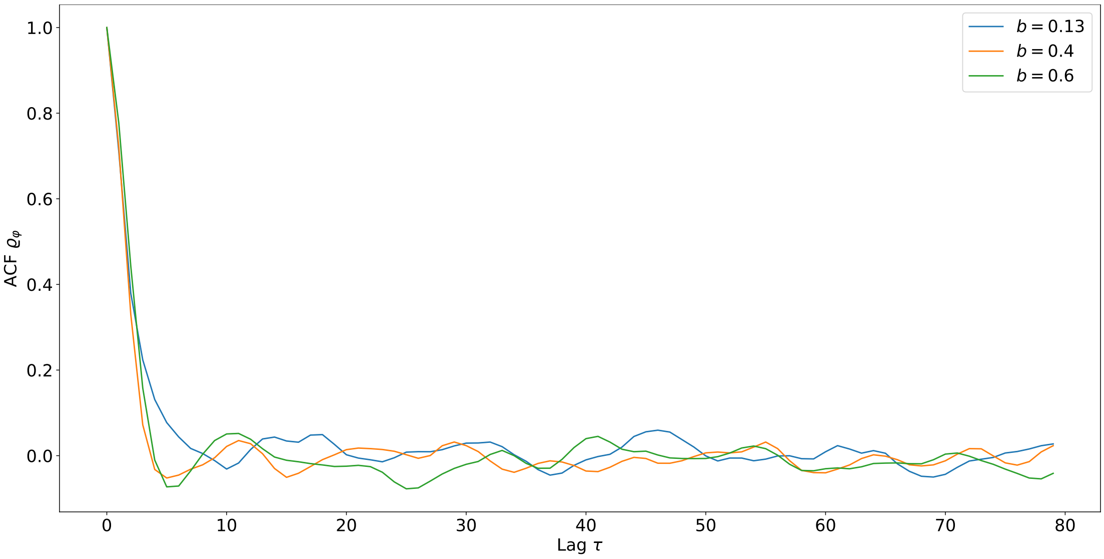


Figure 6.12: Plotting ACF ϱ_φ against lag τ, for different b values for the One-Directional I-Jump sampler.

From Figure 6.12 we can see that all the chains produced with different b values decay in a similar fashion. In fact, when $b = 0.4$ and $b = 0.6$ decayed quicker than $b = 0.13$ between lags 5 and 10. Therefore, for this problem we can conclude that for the One-Directional I-Jump sampler, a smaller step size works as well or better than a large step size, when we compare the rate at which the ACF decays. Therefore, one would prefer to use a smaller step size.

Thus, we suggest that $\hat{\alpha} > 0.234$ should be greater for the One-Directional I-Jump sampler, which in turn will result in smaller step sizes. We have see the rate of decay of the ACF for smaller step sizes is greater than for larger step sizes, which is desirable. To justify a greater value of $\hat{\alpha}$ we need to look at the algorithm. **Algorithm 2** only proposes candidate solutions that are greater or less than the current sample depending on the sign of the indicator variable $\boldsymbol{z}$. The sign of $\boldsymbol{z}$ only changes upon **rejection**, this is the key point. If the $\hat{\alpha}$ is low then on average we are rejecting samples more often. Thus, we are alternating between the directions from more frequently. But, in fact we would like longer chains in one direction, so we need to decrease the frequency at which the sign of $\boldsymbol{z}$ changes. To achieve this we let the $\hat{\alpha}$ be larger which will allow for more samples to be accepted. Therefore, to achieve a larger value of $\hat{\alpha}$ we can do this by increasing b, thus decreasing the step size, as seen in Figure 6.11.

6.2.2 Comparison

To conclude we will compare the One-Directional I-Jump sampler with parameters $a = 1$ and $b = 0.4$. I have chosen $b = 0.4$ so both samplers have the same step size refer to subsection 6.2.1, against the MH algorithm with Random Walk proposal with parameter $\beta = 3.2$.

From Figure 6.13 we can immediately see the benefits of using a non reversible sampler, due to the rate at which the ACF decays. We can see that for small lags, the ACF decreases quicker than for reversible samplers. This is consistent with the advantages we covered in section 5.2, since this means that the asymptotic variance for the non reversible sampler should be smaller than the reversible sampler on this problem. In the next sections we will compare these two algorithms on a range of different target distributions in the hope that we can highlight the benefits of the non reversible sampler.

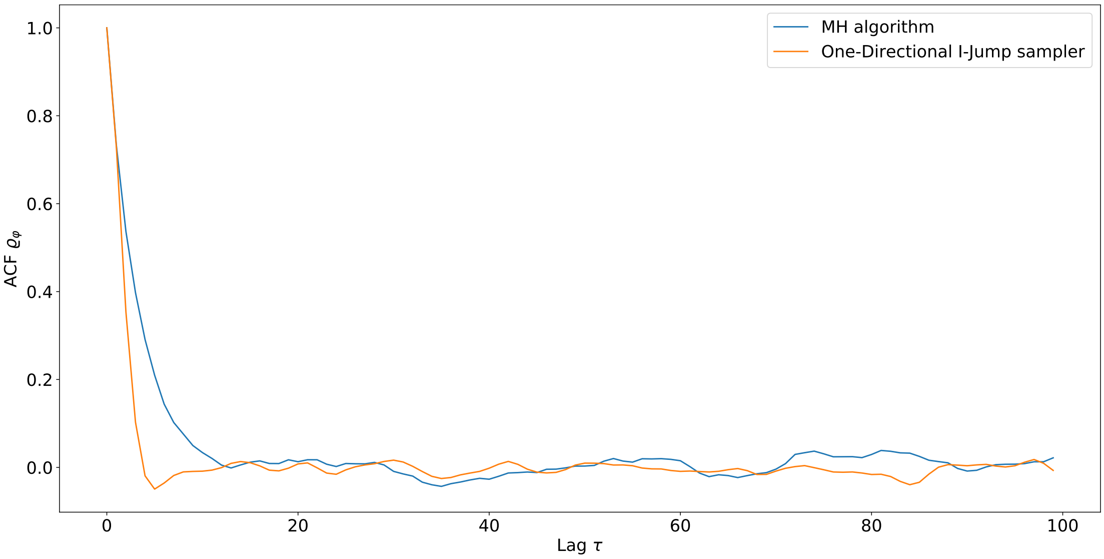


Figure 6.13: Plotting ACF ϱ_φ against lag τ, using MH algorithm and One-Dimensional I-Jump sampler. For $\mathcal{N}(x|10, 2)$ target distribution.

6.3 Exploring Target Distributions

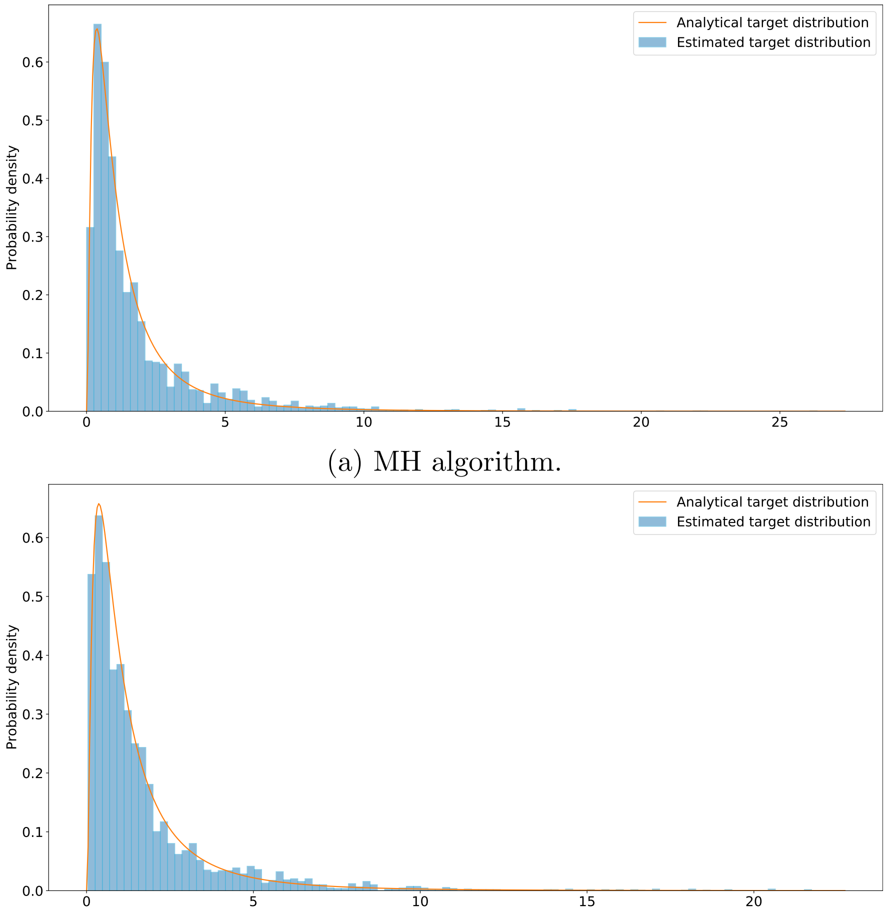
In the previous sections we have been focusing on one target distribution, to illustrate how different algorithms are tuned, finding burn-in phases and showing the convergence of these methods. In this section, we will shift our focus to compare the performance of reversible and non reversible on different **one** and **two** dimensional target distributions. For consistency we will tune the algorithms such that they have the **same** acceptance ratio, in particular $\hat{\alpha} \approx 0.234$.

6.3.1 Log Normal Distribution

The first distribution we will try to sample from is the **Log Normal distribution**. In particular, we say Y has a log normal distribution if $X = exp(Y)$ is normally distributed. The log normal distribution has the same two parameters that the normal distribution has. We need to define the mean μ and standard deviation σ. In this case the target distribution $\pi(x) = LogNormal(x|0, 1)$, where $\mu = 0$ and $\sigma = 1$.

The difficulty in sampling from this distribution is due to its long tail. We have tuned these samplers in the same fashion as in sections 6.1.1 and 6.2.1. For the MH algorithm with a Random Walk proposal the tuned β value which was found was $\beta = 3.2$. Similarly, for the One-Directional I-Jump sampler the parameters we found were $b = 0.2$ and $a = 1$. Notice that in this case the step size for the One-Directional I-Jump sampler is larger than the MH algorithm, recall section 6.2.1, where we discussed step sizes.

For Figure 6.14, at first glance, both samplers seem to produce similar posterior estimates of the $LogNormal(x|0, 1)$ distribution. We need to take a closer look at the chain produced.

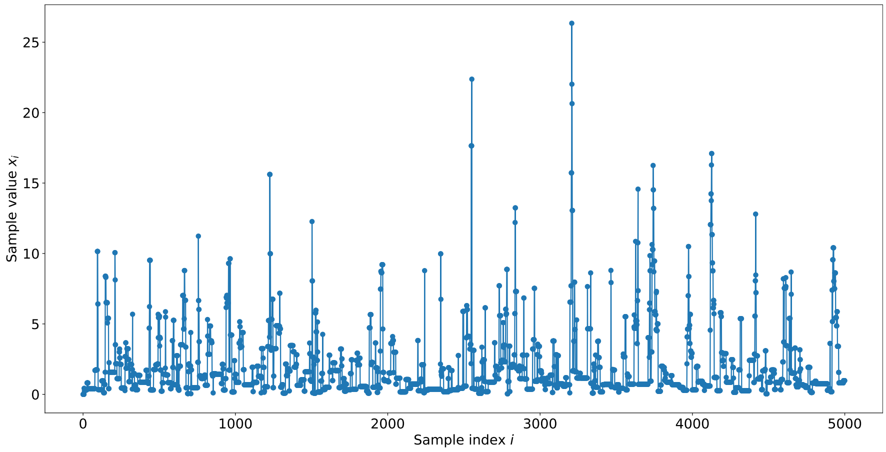


(a) MH algorithm.

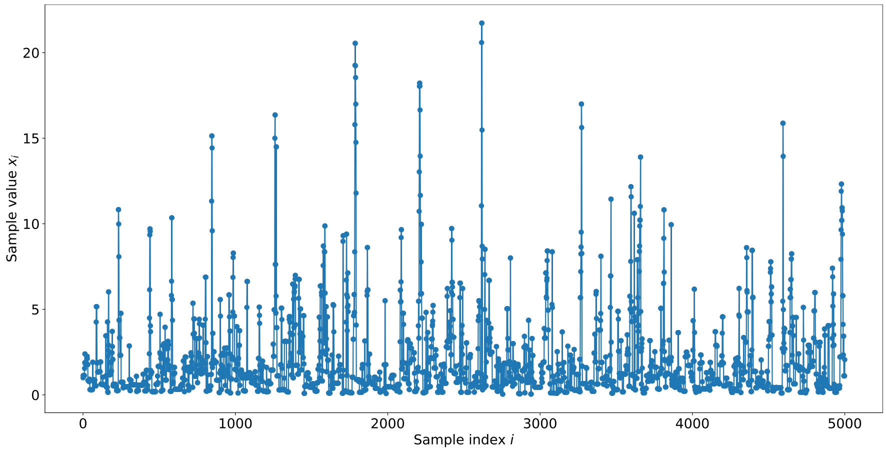
(b) One-Directional I-Jump sampler

Figure 6.14: These figures show the estimated target distribution after the samplers ran for 5000 iterations compared to the analytical target distribution $LogNormal(x|0, 1)$.

In Figure 6.15 we can see that both plots produce *spikes*. These spikes indicate when the sampler is in the tail for the target distribution. In Figure 6.15 (a) we can see that the One-Directional I-Jump as more samples along these spikes. The reason for this is in the nature of the algorithm. We force movement in a certain direction. In many cases, we force the chain to move right as a result the chain is forced up the tail. Since we have this forcing behaviour we sample more points



(a) MH algorithm.



(b) One-Directional I-Jump sampler.

Figure 6.15: These are the chains produced by both samplers, for 5000 iterations when trying to sample from the target distribution $LogNormal(x|0, 1)$.

from this area and as a result we are more likely to accept samples from these lower probability regions.

Again, we will now compare performance between the samplers for each algorithm, by using the ACF.

From Figure 6.16 , we can see that the One-Directional I-Jump sampler decays at a higher rate than the MH algorithm. We know that the target distribution $LogNormal(x|0, 1)$ has density 0 for values of x less than 0. This is the key point. Suppose, we are using the One-Directional I-Jump sampler and we propose a sample $x\prime = x_i - \xi$, that is we are proposing a sample to the left of the current sample. It could be the case that this new value $x\prime < 0$. Then since $\pi(x\prime) = 0$, where $\pi(x)$ is the probability density function of $LogNormal(x|0, 1)$, then the acceptance ratio $\alpha(x\prime|x_i)$ is immediately 0. Since $\frac{\pi(x\prime)}{\pi(x_i)} = 0$ (this is an expression inside $\alpha(x\prime|x_i)$, which can be found in Algorithm 2). Therefore, the sampler **rejects** $x\prime$ and changes the value of the indicator variable $\boldsymbol{z}$, as a result it is **always** the case that the next sample will be in the form of $x\prime = x_i + \xi$ that is we move to the right, towards an area of high target probability since we know

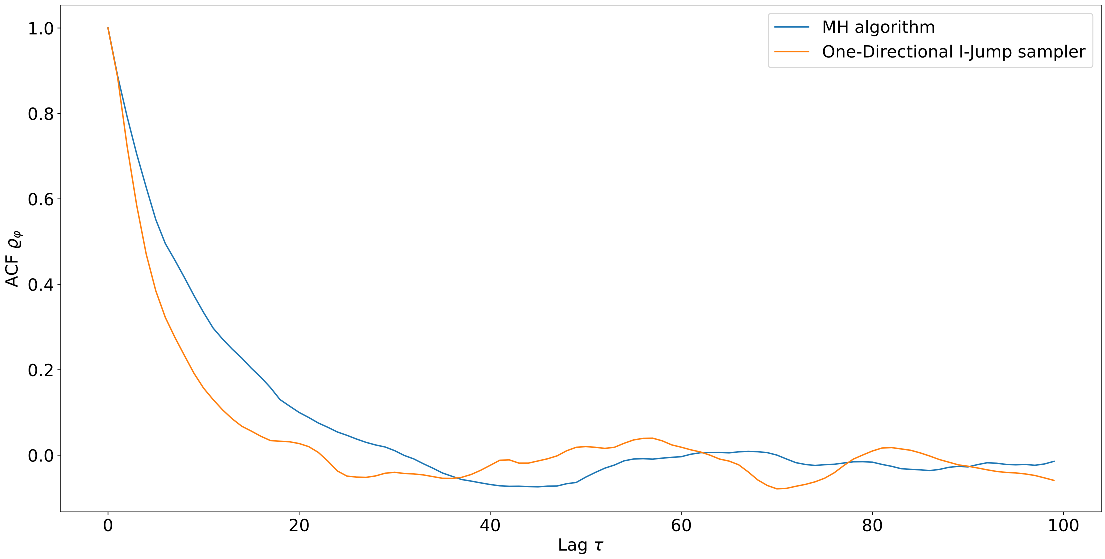


Figure 6.16: Plotting ACF ϱ_φ against lag τ, using MH algorithm and One-Dimensional I-Jump sampler, target distribution is $LogNormal(x|0,1)$.

from the previous proposal that moving left results in $\pi(x\prime) = 0$. Therefore, we are more likely to accept the next sample.

This is the key difference with respect to the MH algorithm. Recall, the samples are in the form of $x\prime = x_i + \beta\epsilon$ where $\epsilon \sim N(0, I)$. Therefore, if $x\prime < 0$, then the acceptance ratio $\alpha(x\prime|x_i)$ is also 0. However, unlike with the One-Directional I-Jump sampler where the next sample is forced in the opposite direction. In this case the next sample $x\prime$ takes the same form. As a result, we are equal likely to reject the proposal since we are sampling ϵ form a $N(0, I)$ distribution. In turn, we remain in the **same** position more often. This means that for small time lags τ the samples are more correlated for the MH algorithm than for the One-Directional I-Jump sampler. In fact, if we observe Figure 6.15 (a) we can see that the chain produced by the MH algorithm has more points concentrated near the sample value 0 while the samples for the One-Directional I-Jump sampler are more spread out between 0 and 5, which supports this claim.

6.3.2 Gaussian Mixture Model

Gaussian Mixture Model can be thought as a probability distribution which is made up of several Gaussian distributions added together, each with an associated weighted. The probability density function $\pi(x)$ is given by [37]:

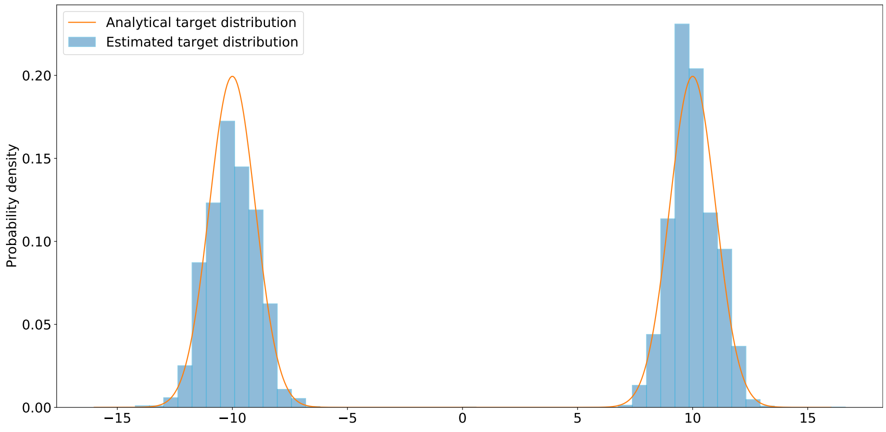
$$\pi(x) = \sum_{i=1}^{K} w_i \mathcal{N}(x| \ \mu_i, \sigma_i) \tag{6.6}$$

Where $\mathcal{N}(x|\mu_i, \sigma_i)$ is the probability density function for the ith Gaussian distribution. Furthermore, w_i is the mixture component weights, where $\sum_{i=1}^{K} w_i = 1$.

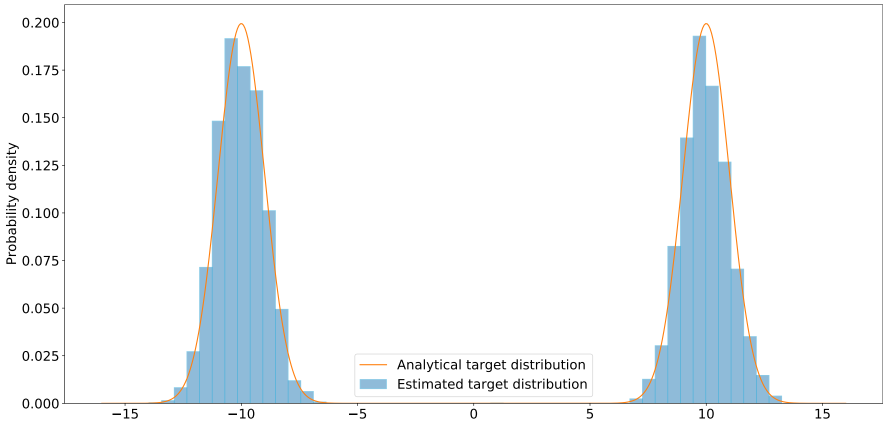
In this example, the target distribution is given by:

$$\pi(x) = \frac{1}{2}\mathcal{N}(x|\text{-}10, 1) + \frac{1}{2}\mathcal{N}(x|10, 1) \tag{6.7}$$

That is, two equal weighted Gaussian distributions with means at -10 and 10 respectively, both with standard deviation 1. We use this distribution since it is spread apart. As a result, this distribution will pose a difficultly for samplers which are not appropriately tuned. Again, we have tuned the samplers in the same fashion as in sections 6.1.1 and 6.2.1. In particular, we use the MH algorithm with a Random Walk proposal where $\beta = 3.2$. Similarly, for the One-Directional I-Jump sampler, where $a = 1$ and $b = 0.2$.



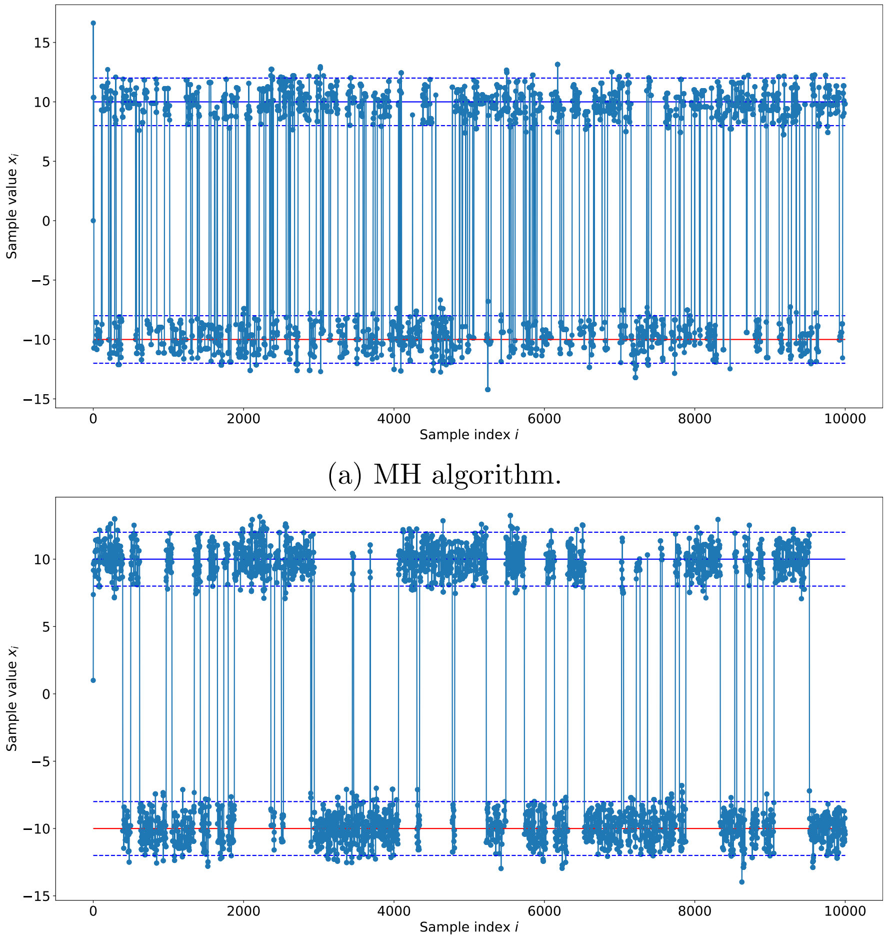
(a) MH algorithm.



(b) One-Directional I-Jump sampler

Figure 6.17: These figures show the estimated target distribution after the samplers ran for 10000 iterations compared to the analytical target distribution $\frac{1}{2}\mathcal{N}(x|\text{-}10, 1) + \frac{1}{2}\mathcal{N}(x|10, 1)$.

We can see from the histograms in Figure 6.17 that both of the samplers, when tuned, are able to explore both distributions in the mixture model. However, from Figure 6.18 we can see the chains produced are different. For the MH algorithm, we can see that there is a high frequency of switching between both areas of high density, while for the One-Directional I-Jump sampler, we can see that the frequency in which it moves between the areas of high density is at a slower rate.



(a) MH algorithm.

(b) One-Directional I-Jump sampler.

Figure 6.18: These are the chains produced by both samplers, for 10000 iterations. For the target distribution $\pi(x) = \frac{1}{2}\mathcal{N}(x|\text{-}10, 1) + \frac{1}{2}\mathcal{N}(x|10, 1)$.

To explain this we can compare the nature of the algorithms. The One-Directional I-Jump sampler proposes samples that move in exactly one direction until it rejects a sample. This result in longer periods of time where the chain moves in one direction. Thus, the chain is sampling exclusively from one of the two areas of high density. In contrast to the MH algorithm where the chain is produced by a random walk. It is more likely to move between the two areas of high probability since at every step the algorithm can propose any direction to move in. As a result, we a high frequency of movement between the areas of high density.

Surprisingly, from Figure 6.19, we can see that the rate at which ACF decays for the MH algorithm is greater than the rate of decay for the One-Directional I-Jump sampler. To understand why this is the case we can go back to Figure 6.18, as we mentioned the chains produced are different. The MH algorithm, due to the Random Walk proposal, can jump between both areas of high density more often, which inevitably result in smaller correlation between samples for smaller time lags. Thus, the rate at which the ACF decreases is greater. In contrast to the One-Directional I-Jump Sample where the chains remain in exclusively one

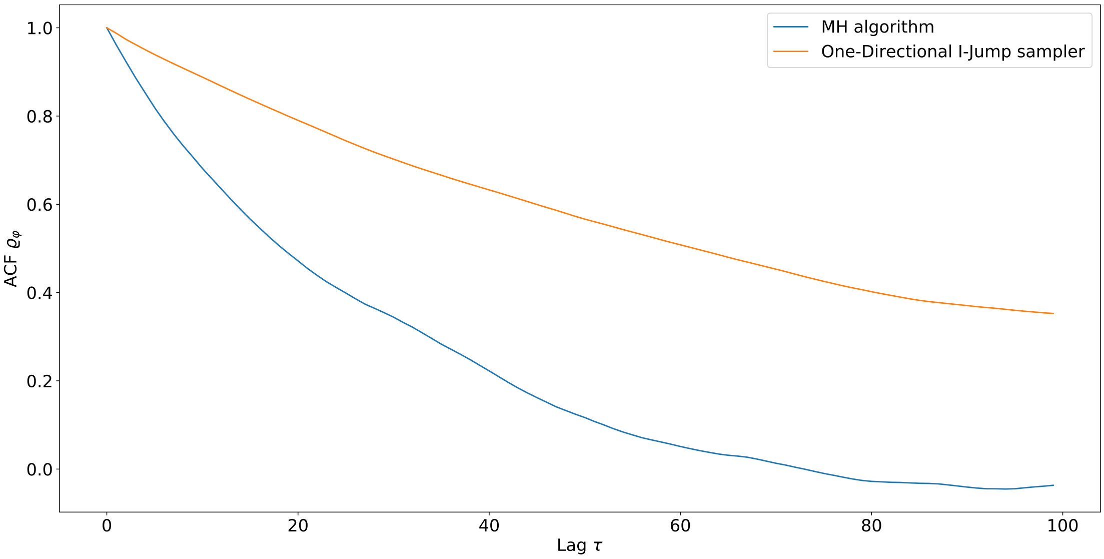


Figure 6.19: Plotting ACF ϱ_φ against lag τ, using MH algorithm and One-Dimensional I-Jump sampler. For the $\pi(x) = \frac{1}{2}\mathcal{N}(x|\text{-}10, 1) + \frac{1}{2}\mathcal{N}(x|10, 1)$ target distribution.

of the two areas of high probability density for longer, as a result the samples for small lags will often be in the same area which results in the correlation between the samples being larger. This happens because we force a certain direction of movement. Finally, I want to illustrate the problem with small step sizes with this target distribution. Again, we will use the MH algorithm with $\beta = 0.5$.

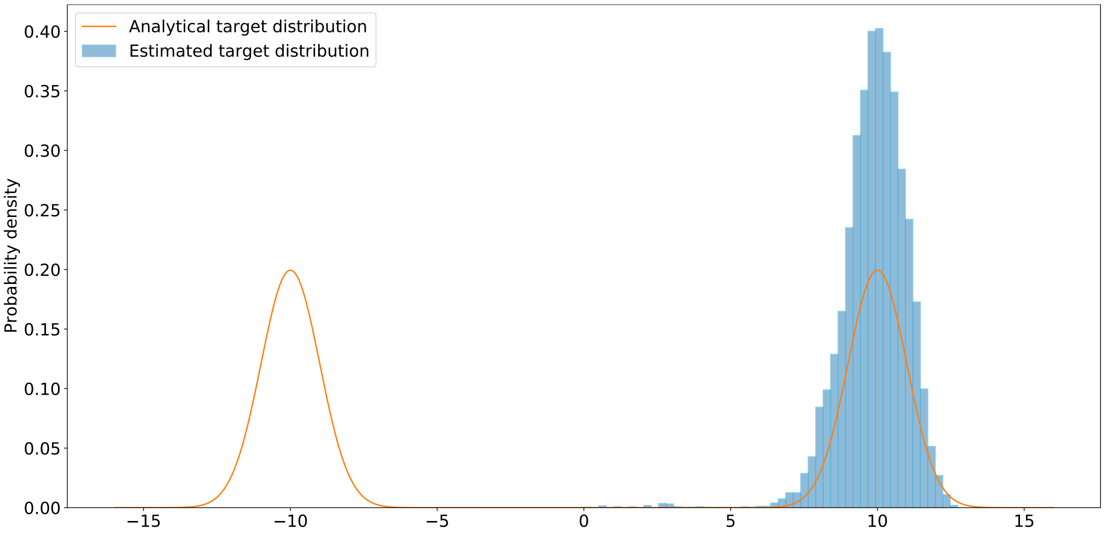


Figure 6.20: Estimated target distribution for a Gaussian Mixture Model, using the MH algorithm with $\beta = 0.5$, a small step size. With 10000 iterations.

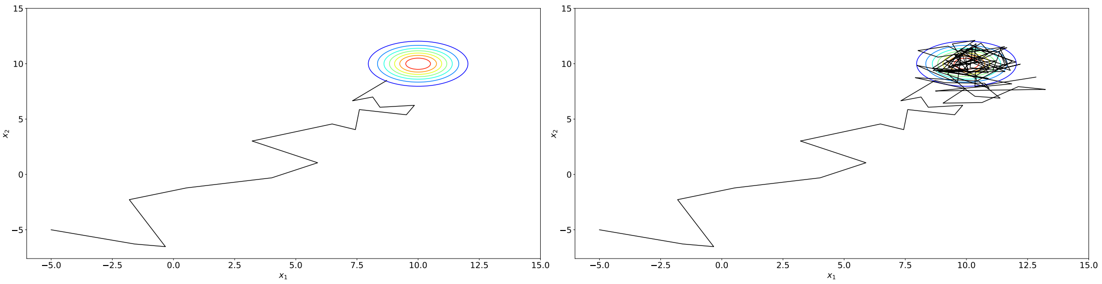
We can see in Figure 6.20 the chain gets trapped on the right side of distribution. Since the areas of high density in the distribution are far apart, the chain will not be able to explore the other side of the distribution by taking small steps. Recall subsection 4.6.2, that the proposal density is symmetric then the acceptance ratio is given by $\alpha(x\prime|x_i) = \min\left\{1, \frac{\pi(x\prime)}{\pi(x_i)}\right\}$, as a result $\frac{\pi(x\prime)}{\pi(x_i)}$ will be close to 0, between -5 and 5, when it moves towards the left side of distribution. Since it takes a small step size it cannot propose a sample within the area of high probability. As a result, the chain does not explore the other side of the distribution. In fact, as the chain remains on the right side, the estimated target distribution for this chain is the exactly $\mathcal{N}(x|10, 1)$ target distribution.

6.3.3 Multivariate Gaussian Distribution

We will now shift our attention to two dimensional target distributions. In this section we will explore the well know **Multivariate Gaussian distribution**. This is simply a generalization of the one dimensional normal distribution to higher dimensions. We will use the notation $X \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ to refer that a random variable X comes from a multivariate Gaussian distribution with d-dimensional mean vector $\boldsymbol{\mu} \in \boldsymbol{R}^d$ and with a covariance matrix $\boldsymbol{\Sigma} \in \boldsymbol{R}^{d\times d}$. The probability density function is given by:

$$p(x|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = (2\pi)^{-\frac{d}{2}} \det(\boldsymbol{\Sigma})^{-\frac{1}{2}} \, e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathsf{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})}, \tag{6.8}$$

Note, that this exists only when $\boldsymbol{\Sigma}$ is positive-definite. In this example, we will focus on a simple multivariate Gaussian distribution to illustrate these algorithms working in a two dimensional space. In particular, the target distribution we will be $\pi(x|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ where $\boldsymbol{\mu} = \begin{bmatrix}5\\5\end{bmatrix}$ and $\boldsymbol{\Sigma} = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$. Furthermore, the seed we have set makes all the chains start at position (-5, -5). We being by showing how the chains move throughout the space to sample from the target distribution.



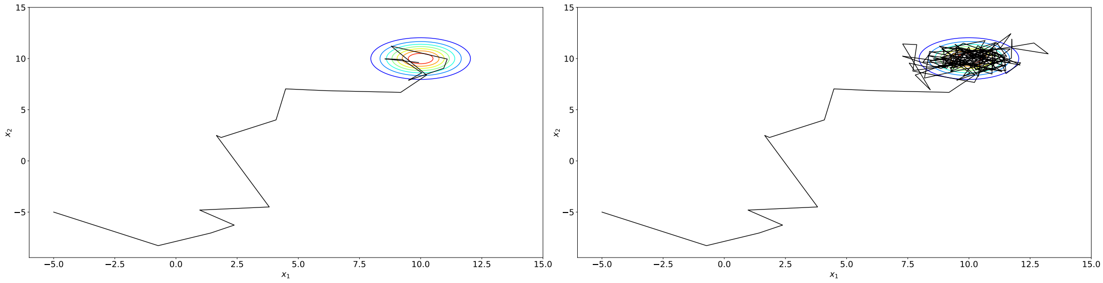
(a) N = 50

(b) N = 500

Figure 6.21: Trajectory of the MH algorithm using Random Walk proposals where $\beta = 2$. (a) First 50 steps (b) First 500 steps. For the multivariate Gaussian target distribution $\pi(x|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ where $\boldsymbol{\mu} = \begin{bmatrix}5\\5\end{bmatrix}$ and $\boldsymbol{\Sigma} = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$.

We can see form Figure 6.21 (a) that for the first 50 steps of the MH algorithm with $\beta = 2$ still has not reached the area of high density that we want to sample from. However, after 500 steps the algorithm begins to sample points from the distribution. Notice that it is not concentrated has difficulties with staying within this distribution. We will now see how this trajectory compares to the Two-Directional I-Jump sampler.

Before comparing the two algorithms we need to understand what parameters are tuned for the N-Directional I-Jump sampler, in this case $N = 2$, so we will refer to this algorithm as the Two-Directional I-Jump sampler. If we reference back to **Algorithm 3** we can see that the variable we have control over is $\eta \sim \mathcal{N}(0, \sigma^2 I)$ which is used for proposals. In this case we let $\sigma = 2$ in order for $\hat{\alpha} \approx 0.234$, tuning it in a similar fashion as in subsection 6.2.1. We also have control over the period at which we change the direction vector $\boldsymbol{y}$. We found that the for this problem periodically changing the direction every 2 steps yielded the best results.



(a) N = 50

(b) N = 500

Figure 6.22: Trajectory of the 2-Dimensional I-Jump sampler, where $\sigma = 2.5$ and the period is equal to 2. (a) First 50 steps (b) First 500 steps. For a multivariate Gaussian target distribution $\pi(x|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ where $\boldsymbol{\mu} = \begin{bmatrix} 5 \\ 5 \end{bmatrix}$ and $\boldsymbol{\Sigma} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$.

We can see form figure 6.22 (a) that for the first 50 steps of the algorithm has reached the area of high density that we want to sample from, this again is consistent with theory where we can see faster convergence to the target distribution than reversible samplers, covered in subsection 5.2. Furthermore, after 500 steps the algorithm begins to sample points from the distribution. We can see these are in fact more concentrated within the areas of high density, in contrast to the MH algorithm on the same problem.

Perhaps what is more interesting to consider is the value of the period used in this case, that is the direction of exploration for the non reversible sampler. Since the period is equal to 2 this suggests that the algorithm preformed the best when it was changing the direction a high frequency. This can be explained since we are working in a two dimensional space the sampler works best when it moves in all directions of the state space and not just focusing on one.

Finally, to compare these two algorithm we will focus on the rate of decay of the ACF, like we have been doing in the previous sections.

We can see from Figure 6.23 that the rate at which the Two-Directional I-Jump sampler decays is greater than that of the MH algorithm. This is consistent with theory since we expected the asymptotic variance to be smaller for non reversible samplers. This means that the correlation between the samples for the non reversible chain to be smaller. As a result, we see that the rate at which the which ACF decreases is greater than the MH algorithm on the same problem. This highlights one of advantages of using non reversible samplers in two

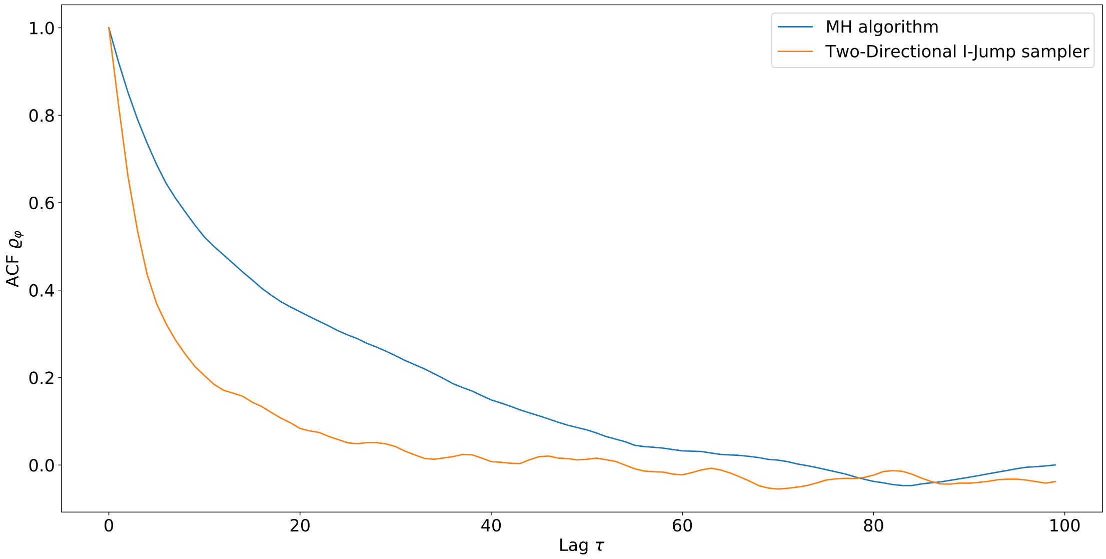


Figure 6.23: Plotting ACF ϱ_φ against lag τ, using MH algorithm and Two-Directional I-Jump sampler. For the Multivariate Gaussian distribution target $\pi(x|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ where $\boldsymbol{\mu} = \begin{bmatrix} 5 \\ 5 \end{bmatrix}$ and $\boldsymbol{\Sigma} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$.

dimensional. We can already see a significant decay in the correlation between the samples on a more complicated problem.

6.3.4 Rosenbrock Function

In this section we explore the **Rosenbrock function**. It is a notoriously difficult function to explore. The function is concentrated inside a parabolic shaped valley. Therefore, when trying to sample from this function it will be difficult to reach areas of high density. The function is defined by:

$$f(x, y) = (a - x)^2 + b(y - x^2)^2 \tag{6.9}$$

Typical values used for this function are $a = 1$ and $b = 100$. These are the values we will use. Furthermore, we will follow [33], where they re-scale equation 6.9 so that the distribution takes the shape of a curved narrow ridge, in this case we will re scale by 1/5. Putting this all together we can get an expression for our target target probability distribution:

$$\pi(x_1, x_2) = exp(-[(1 - x_1)^2 + 100(x_2 - x_1^2)^2]/5) \tag{6.10}$$

Figure 6.24 shows the target distribution. This function is usually quite hard for MCMC algorithms to explore. Furthermore, the seed we have set makes all the chains start at position (-5, -5) in order to show how the chains move thought the space and reach the desired distribution. Figure 6.25 shows the trajectories we get from running the MH algorithm.

We can see form Figure 6.25 (a) that for the first 50 steps of the algorithm has reached the target distribution. However, we can see the steps it has taken

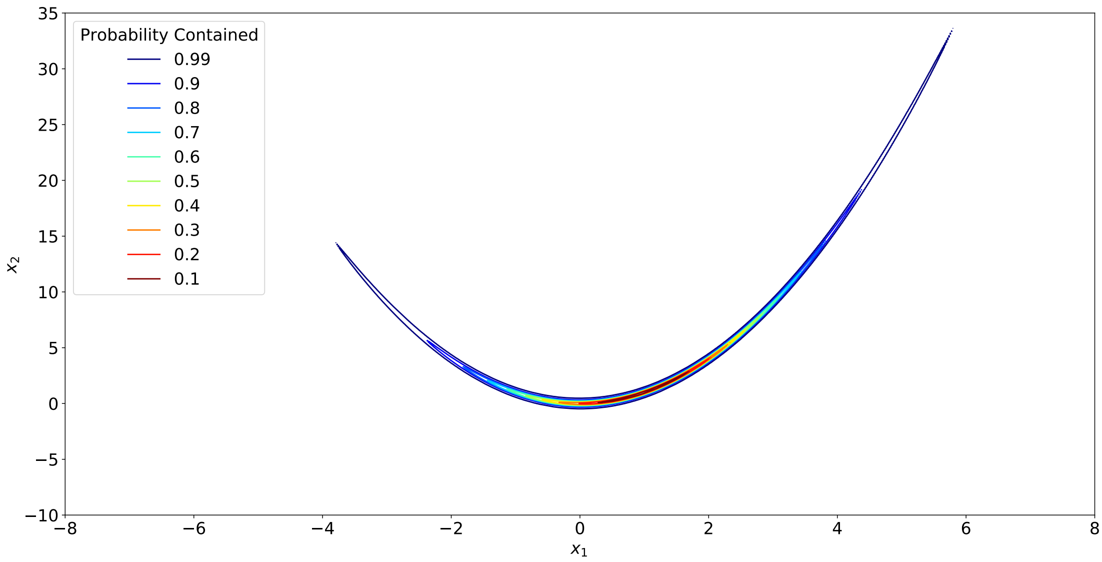
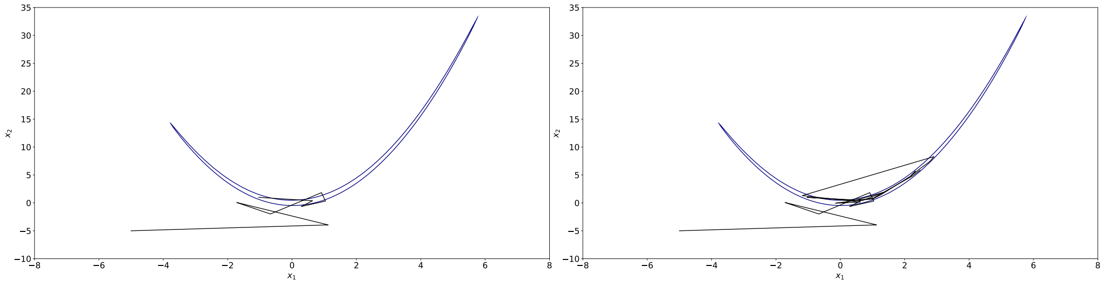


Figure 6.24: Contour plot of the 2D-Rosenbrock density as described in Equation 6.10.



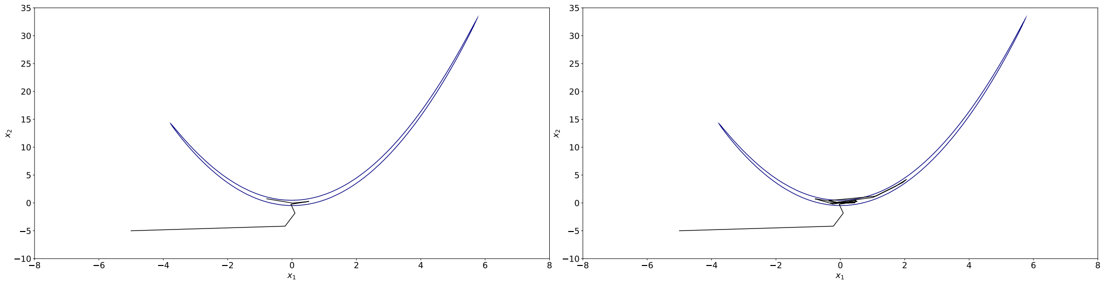
(a) N = 50

(b) N = 500

Figure 6.25: Trajectory of the MH algorithm using Random Walk proposals where $\beta = 2$. (a) First 50 steps (b) First 500 steps. For the Rosenbrock target distribution 6.10.

to reach it are irregular and the algorithm finds it difficult to remain within the areas of high density. Furthermore, this becomes more evident when we look at (b) we can see that the chain moves in all directions jumping over to the other side of the ridge, suggesting that the Random Walk proposal is not suitable for this target distribution. We will now see how this trajectory compares to the Two-Dimensional I-Jump sampler.

In this case we let $\sigma = 2.5$ in order for $\hat{\alpha} \approx 0.234$. We also have control over the period at which we change the direction vector $\boldsymbol{y}$. We found that the for this problem periodically changing the direction every 2 steps yielded the best results, this the same as the Multivariate Gaussian Distribution. We can see form Figure 6.26 (a) that for the first 50 steps of the algorithm has reached the area of high density that we want to sample from. Interestingly, after 500 steps the algorithm begins to sample points from the distribution and seems to concentrate in the areas of high density, in contrast to the MH algorithm on the same problem, where it found it hard to remain within this area.



(a) N = 50

(b) N = 500

Figure 6.26: Trajectory of the 2-Dimensional I-Jump sampler, where $\sigma = 2.5$ and the period is equal to 2. (a) First 50 steps (b) First 500 steps. For the Rosenbrock target distribution 6.10.

Finally, to compare these two algorithm we will focus on the rate of decay of the ACF, like we have been doing in the previous sections.

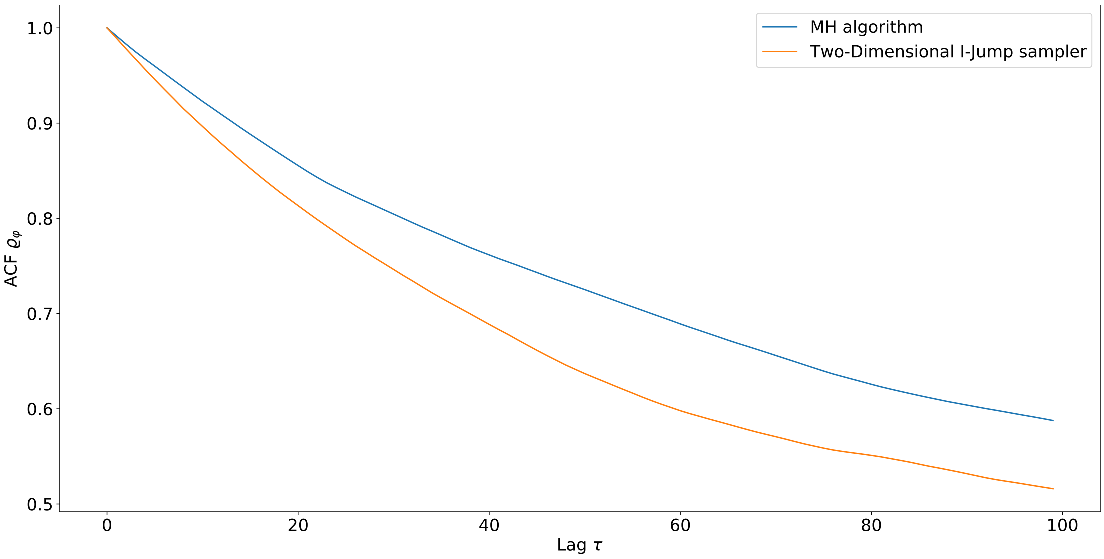


Figure 6.27: Plotting ACF ϱ_φ against lag τ, using MH algorithm and Two-Directional I-Jump sampler. For the Rosenbrock target distribution 6.10.

We can see from Figure 6.27 the rate at which the Two-Directional I-Jump sampler decays is greater than that of the MH algorithm. However, when we compare this to previous example of the Multivariate Gaussian the rate of decay is not as steep. This highlights the difficulty of sampling form this distribution. Since the area which the density is high is very narrow many of the samples will be close together and subsequently correlated. Tuning the parameters for a larger acceptance rate could be beneficial for both samplers since they have more flexibility to accept samples beyond the narrow ridge. Furthermore, we can see the immediate benefits of using a non reversible sampler. We have shown that for a more complicated target distributions the non reversible sampler preforms better.

6.4 Future Work

In section 6.3 we have seen the performance of both MH algorithm and the N-Directional I-Jump sampler on different target distributions. In particular we have made evident the substantial benefits of using non reversible samplers, in particular by highlighting the rate at which the normalized autocorrelation function decreased. In most cases the N-Directional I-Jump sampler decayed quicker, which in turn reduced the asymptotic variance.

However, there are still many undiscovered areas that can be researched for non reversible samplers. In particular, finding an appropriate average acceptance ratio $\hat{\alpha}$. We suggested in subsection 6.2.1 that a larger $\hat{\alpha}$ can yield better results for these samplers. It is remarkable, that even though, throughout the comparisons we used $\hat{\alpha} \approx 0.234$, there still could be room for improvement if we can find an $\hat{\alpha}$ to appropriately tune the I-Jump samplers.

Another area of research is to methodically find methods to choose the d-dimensional auxiliary variable $\boldsymbol{y}$, rather than randomly, when using the N- Directional I-Jump sampler. In many cases picking an appropriate direction, like the MALA proposal density which incorporates the gradient, could be of interest here. That is, the algorithm can periodically pick a direction which moves towards areas of higher density for a more focused search on the state space. This will be further highlighted in the next chapter where we will be working with higher dimensional problems.

These are some of the many areas that can be improved when working with the N-Directional I-Jump sampler. Work on these areas will further increase the power of sampling algorithms.

To conclude, in this chapter we have showed how to appropriately tune the MH algorithm as well as how to find the burn in phase. We showed how to pick different proposal density. Then we shifted our attention to the One-Directional I-Jump sampler and tuning its parameters. Finally, we sampled from various different target distributions, comparing the performances between reversible and non reversible samplers. In many cases we saw the advantages of non reversible sampler. Finally, we proposed future research areas that can be investigated in order to increase the power of non reversible samplers. In the next chapter, we will continue the comparison between reversible and non reversible sampler but applied to the context of inverse problems in imaging.

Note that all plots were implemented from scratch in Python using Matplotlib, NumPy , Pandas and SciPy libraries, these function can be found in Appendix C.

Chapter 7

Inverse Problems In Imaging

In this chapter we will put together all we have covered in Chapters 2 through 6 to solve inverse problems in imaging. We will begin by motivating the reader as to why they would be interested in solving such problems. Then we will introduce the framework, using the Bayesian inference approach, which will allow us reformulate imaging problems as inverse problems. Finally, we will use MCMC methods to sample from the posterior distributions in the hope to solve these problems.

7.1 Motivation

An important question we need to ask ourselves is *why* are we interested in solving such problems and how are they relevant to us?

In recent years, there has been a growth of new imaging technologies, such as modern telescopes and medical scanners. These collect data and then put them together to form images with a computer. In many cases, this process involves solving an inverse problem. The unknown image u is reconstructed from indirect observations y of the object of interest. It comes to no surprise that these imaging technologies appear in a variety of real world areas, for example biomedical imaging techniques such as computed tomography (CAT) and magnetic resonance imaging (MRI) [5]. As a result, this highlights the importance of researching and advancing these techniques, due to their wide range of applications.

Furthermore, in many cases, imaging problems are typically ill-posed 2.2.1. Therefore, it is vital to develop methods and techniques to deal with this instability.

7.2 Background

In this section give a background to understand how image de-blurring and image de-noising can be formulated as inverse problems. These are the problems we will attempt to solve. We will begin by explaining how images are interpreted by computers. A quick note, the original image Figure 7.1(a), which we will use throughout this chapter, was provided by my supervisor and does not have a known author. Other images have been made by me.

7.2.1 Fundamentals of Image Processing

Before we can formulate imaging as an inverse problem we first need to understand how an image can be interpreted by a computer. The most basic element of an image is a **pixel**. Pixel's are the individual points that when put together form an image. One can visualize an image as a two dimensional array of pixels when we are given an an image u then we can refer to a pixel as $u_{n,m}$. This will be the pixel in the nth row and mth column. For example, when we refer to pixel $u_{0,0}$ we will be referring to the pixel in the first column and row. It is also worth mentioning that the number of rows n and the number of columns m does not need to be the same. However, in this project we will be working with n by n images.

Finally, I would like to briefly mention color. In this project we will be focusing on greyscale images. The value of each pixel is a single sample representing the amount of light is present. The values for each pixel range from 0 to 256. Black at 0 to white at 256. However, images many take a variety of color, these can be achieved by using the RGB color model.

7.2.2 Image De-Noising and De-Blurring

Image de-noising and de-blurring are problems that can be reformulated as an inverse problem. Suppose you have an image with added noise. This can arise through a range of different reasons, for example, through measurement inaccuracies in the instrument used to capture the image. The goal is to reconstruct the underlying unknown image u from the observed image y, where:

$$y = A(u) + \eta \tag{7.1}$$

We need to convert an image into a tangible mathematical formulation. To do this we will work with individual pixels. Each pixel of the observed image is given by $y_{i,j} = A(u_{i,j}) + \eta$, where in this case η is the observational noise, $\eta \sim \mathcal{N}(0, \sigma^2)$. Each pixel of the unknown image $u_{i,j}$ is perturbed by some blurring operator A, a **linear** operator. In many cases one needs to know the blurring operator A before hand or at be able to approximate it, in order to solve these problems.

The blurring operator A is applied to each pixel $u_{i,j}$ of the unknown image independently. Each of the elements in the matrix has an associated weight to it, know as the importance weights. To illustrate this suppose we have the following blurring operator A:

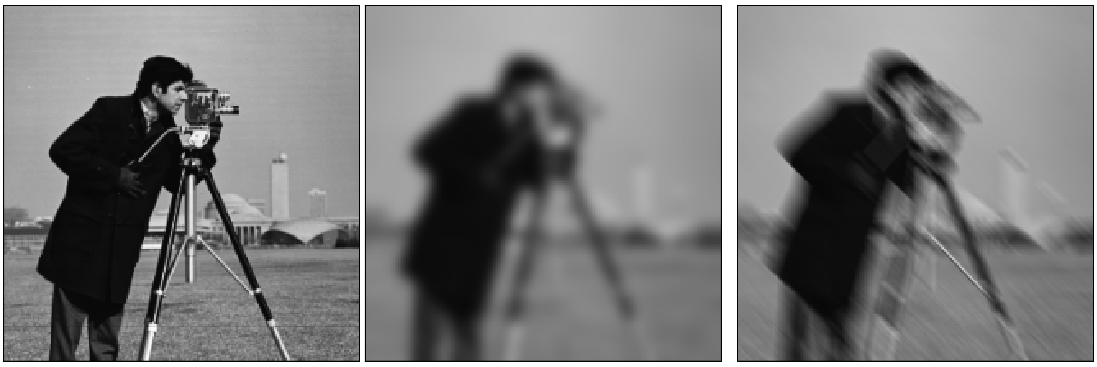
$$A = \begin{bmatrix} 0 & w_1 & 0 \\ w_2 & w_3 & w_4 \\ 0 & w_5 & 0 \end{bmatrix} \tag{7.2}$$

We can apply this operator on some pixel $u_{i,j}$ then observed pixel once it undergoes the transformation is given by $y_{i,j} = w_1 u_{i+1,j} + w_2 u_{i,j-1} + w_3 u_{i,j} + w_4 u_{i,j+1} + w_5 u_{i-1,j}$. What is happening is that all the surrounding pixels influence the current pixel with a certain weight w_i. Notice that this is just an example, this can be generalized to:

$$y_{i,j} = \sum_{a \in N(i,j)} w_a u_a \tag{7.3}$$

Where $N(i,j)$ are the neighbourhood of pixel $u_{i,j}$, that is the pixels around $u_{i,j}$.

The key difference between image de-blurring and image de-noising is that when we deal with image de-noising the blurring operator A is the identity thus the observation is in the form of $y = u+\eta$. In Figure 7.1, we can see the difference highlighted by the subfigures (b) and (c). We can see that blurring and adding noise to an image are very different. Intuitively, the noise can be thought as a camera not in focus while blurring could be thought as the camera moving.



(a) Original Image (b) Noisy Image (c) Blurred Image

Figure 7.1: Comparison between original image (a), noisy image with added noise to each pixel where $\eta \sim \mathcal{N}(0,5)$ (b) and image blurred with no noise where the blurring operator $A = \frac{1}{15} I^{15\times 15}$, that is the 15 by 15 identity matrix times $\frac{1}{15}$ (c).

7.3 Framework

In this section we will discuss how to formulate image de-blurring and de-noising using the Bayesian inference approach, explained in Chapter 3. Recall, from Theorem 3.2.1, we need to derive an expression for the posterior distribution $\pi(u|y)$ which is proportional to $L(y|u)\pi_{pr}(u)$ where $L(y|u)$ is the likelihood function and $\pi_{pr}(u)$ is the prior.

Prior

We will being by defining the prior. We mentioned in subsection 3.2.1 that in order to make appropriate prior one usually needs have knowledge of the domain. In this case we since we are working with images we use the **Total Variation (TV) prior** [19, p. 97]. We will use this prior for both image de-noising and de-blurring.

The Total Variation (TV) prior is a measure of the "accumulated size of the edges in the image" [19, p. 98]. Intuitively, what this prior is telling us is that for sharper images, that is images with greater difference between the pixels, are more likely to be undisturbed images thus will be given higher probability. The total variation of the image u is given by:

$$\pi_{pr}^{TV}(u) = \sum_{i=1}^{m}\sum_{j=1}^{n}((uD_{1,n}^T)_{i,j}^2 + (D_{1,m}u)_{i,j}^2)^{\frac{1}{2}} \tag{7.4}$$

"For periodic boundary conditions the circulant matrix $D_{1,m}$ can be used to compute first derivatives" [19, p. 99]. This ensures that when u is the constant image, then $\pi_{pr}^{TV}(u) = 0$. The circulant matrix $D_{1,m}$ is given by:

$$D_{1,m} = \begin{bmatrix} -1 & 1 & 0 & \dots & 0 \\ -1 & 0 & 1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 0 & 1 \\ 0 & \dots & 0 & -1 & 1 \end{bmatrix} \tag{7.5}$$

Likelihood

What is left to define is the likelihood function $L(y|u)$. We assume that the noise for each pixel is given by $\eta \sim \mathcal{N}(0, \sigma^2)$. We know that each pixel is given in the following form $y_{i,j} = A(u_{i,j}) + \eta$. Then $y_{i,j} \sim \mathcal{N}(A(u_{i,j}), \sigma^2)$ since it is a random variable centred at $A(u_{i,j})$ with the same standard deviation as the noise. As result we have explicit expression for the likelihood for each pixel, given by:

$$L(y_{i,j}|u_{i,j}) \sim \mathcal{N}(A(u_{i,j}), \sigma^2) \tag{7.6}$$

Note that in the case of image de-noisng the blurring operator $A = I$, therefore we have that the likelihood will be given by $L(y_{i,j}|u_{i,j}) \sim \mathcal{N}(u_{i,j}, \sigma^2)$.

We have found an expression for the likelihood for each pixel. We need to generalize this for the whole image y by using a **multivariate Gaussian distribution**. We know that we will be working with $n \times n$ images. Therefore, since each pixel $A(u_{i,j})$ is the mean of the Gaussian distribution given in equation 7.6. For some image u we can apply the blurring operator A on each pixel and then flatten it. This will result in a n^2 dimensional vector which will be the mean of the multivariate Gaussian defined by $\mu = [A(u_{0,0}), ..., A(u_{n,n})]$. Furthermore, we let $\sigma^2 I$ be the $n \times n$ covariance matrix, since we know that each pixel has the same standard deviation. Therefore, the likelihood for the whole image is given by:

$$L(y|u) \sim \mathcal{N}(\mu, \sigma^2 I) \tag{7.7}$$

where $\mathcal{N}$ in this case is a multivariate Gaussian distribution. Similarly like before, when we will let $A = I$ when dealing with image de-noising.

As a result, when we put equations (7.4) and (7.7) together we get an expression for the posterior distribution:

$$\pi(u|y) \propto L(y|u)\pi_{pr}^{TV}(u) \tag{7.8}$$

This is when MCMC methods come into play. We need methods to explore this posterior distribution. We can use MCMC methods, covered in Chapters 4 and 5, to sample from $\pi(u|y)$ directly, since we have found an expression which is **proportional** to it. In this particular problem we are interested in reconstructing the unknown image u. Therefore, a point estimate, such as MAP or CM, would be a *sufficient* solution to the inverse problem. Note that in order to calculate the CM we use equation 3.13. The CM can be thought as the average image created over all the images sampled. We can find the average value for each pixel and reform them into an image. Likewise, to calculate the MAP estimate we will evaluate all the samples we have produced at the posterior distribution $\pi(u|y)$ and return the sample with highest value as our point estimate.

7.4 Implementation

In this section we will implement MCMC methods to sample from the posterior distribution given in equation 7.8. Once we have these samples we will calculate point estimates to attempt to reconstruct the unknown image u from the observed image y, and compare the performances of reversible and non reversible samplers on these problem.

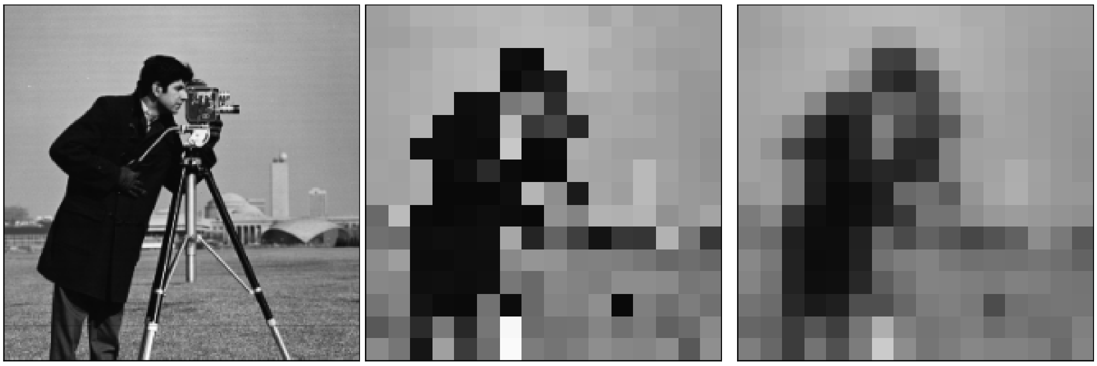
7.4.1 Solution to Image De-Noising

We will first work with image de-noising. In particular the observed image y has added Gaussian noise in the form of $\eta \sim \mathcal{N}(0, \frac{3}{5}I)$. We will fist reshape our image to a smaller dimensional one. To be able to preform more and quicker runs. We will be working with the scaled image. The unknown scaled image u is given by Figure 7.2 (b) and the observed image y by Figure 7.2 (c) with Gaussian noise added to each pixel. As a result, the inverse problem is to reconstruct image u given image y.

Furthermore, when working with images, we will flatten them as explained in section 7.3. As a result, the MCMC methods will be sampling from a high dimensional space. In this case in a $16 \times 16 = 256$ dimensional space, since the images are scaled down to be 16 by 16 in size. This will be a major challenge for both reversible and non reversible samplers, as we shall see.

Since we have we wish to sample from $\pi(u|y)$ we define the target distribution to be $L(y|u)\pi_{VTpr}(u)$. We now need to tune the MCMC methods. In particular we will use the Metropolis Hastings (MH) algorithm with Random Walk proposal and the N-Directional I-Jump sampler. We will tune both algorithms such that average acceptance ratio $\hat{\alpha} \approx 0.234$, for consistency. It is important to mention, that since this problem is high dimensional, we initialize the samplers to start from the given observed image y, Figure 7.2 (c). In order to incorporate as much information we can.

Suppose, we initialize the chain at some random point in the sample space, due to the *curse of dimensionality* this point will likely have close to zero density under the target distribution $L(y|u)\pi_{VTpr}(u)$. As a result, we will be too far from the desired distribution. In fact, this can be seen from Figure 7.3 where the $\hat{\alpha}$



(a) Unknown Image. (b) Unknown Scaled Image. (c) Scaled Image with noise.

Figure 7.2: Scaling our unknown image (a) into one with smaller dimensions (b). In this case we have scaled the unknown image by $\frac{1}{16}$. The unknown image (a) is an 256 by 256 pixels the scaled one is now 16 by 16 pixels. Finally we add Gaussian noise from a $\mathcal{N}(0, \frac{3}{5}I)$ to the scaled image u (b) to get the observed image y (c).

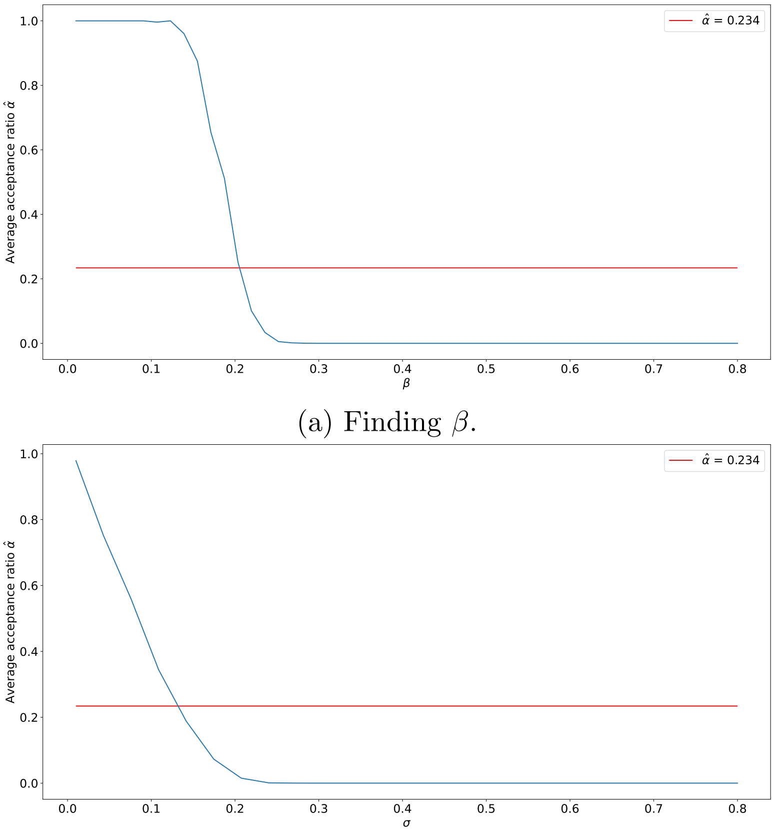
fell to zero when both algorithm started to take larger step sizes, suggesting a low target probability, when taking large step sizes.

We begin by tuning the MH algorithm, we can see that from Figure 7.3 (a) $\beta \approx 0.2$. This is a very small step size, as a result this will pose a problem to explore the state space. We can see that for a larger step size, where $\beta > 0.3$ then $\hat{\alpha} \approx 0$. This means that most of the samples are rejected when we start to take larger step sizes. Also notice that when $\beta < 0.15$ then $\hat{\alpha} \approx 1$, this is because the steps are so small that we in fact accept lots of proposals $x\prime$.

Furthermore, we also need tune the parameter σ and the periodicity for the N-Directional I-Jump sampler. After running various different combinations of periodicity and σ, we can see from Figure 7.3 (b), that $\sigma \approx 0.15$ and periodicity of 10 works best for this problem, again suggesting a small step size. It is worth mentioning that, finding an appropriate periodicity to change the direction $\boldsymbol{y}$ poses to be a problem. We know there is a direction of movement $\boldsymbol{y}$, benefits most from in non reversible chains [25]. Therefore, as we shall see it is vital to propose ways of finding an appropriate direction $\boldsymbol{y}$.

Furthermore, Figure 7.3 (b) differs from (a), since we decide in what direction $\boldsymbol{y}$ we sample from. In many cases even with very small step sizes, we do not accept the proposals in certain directions $\boldsymbol{y}$, since we could be moving to low density areas of the state space. So, we do not see the horizontal line for $\sigma < 0.15$.

Once we have found these parameters we can sample from the target distribution $\pi(y|u)$. In this example, we will run the MCMC algorithms for 10000 iterations. Furthermore, to see how good these estimates are we will calculate the **mean square error** (MSE) between the unknown image u and the point estimate produced. The mean squared error simply measures the squared difference between each pixel, if this measure is small then the images are more alike. The MSE between the unknown image scaled image u and observed image y, is **543.92**, to 2 decimal places. Therefore, smaller values than this suggests a better estimate. In Figure 7.4 shows the CM and MAP point estimates produced.



(a) Finding β.

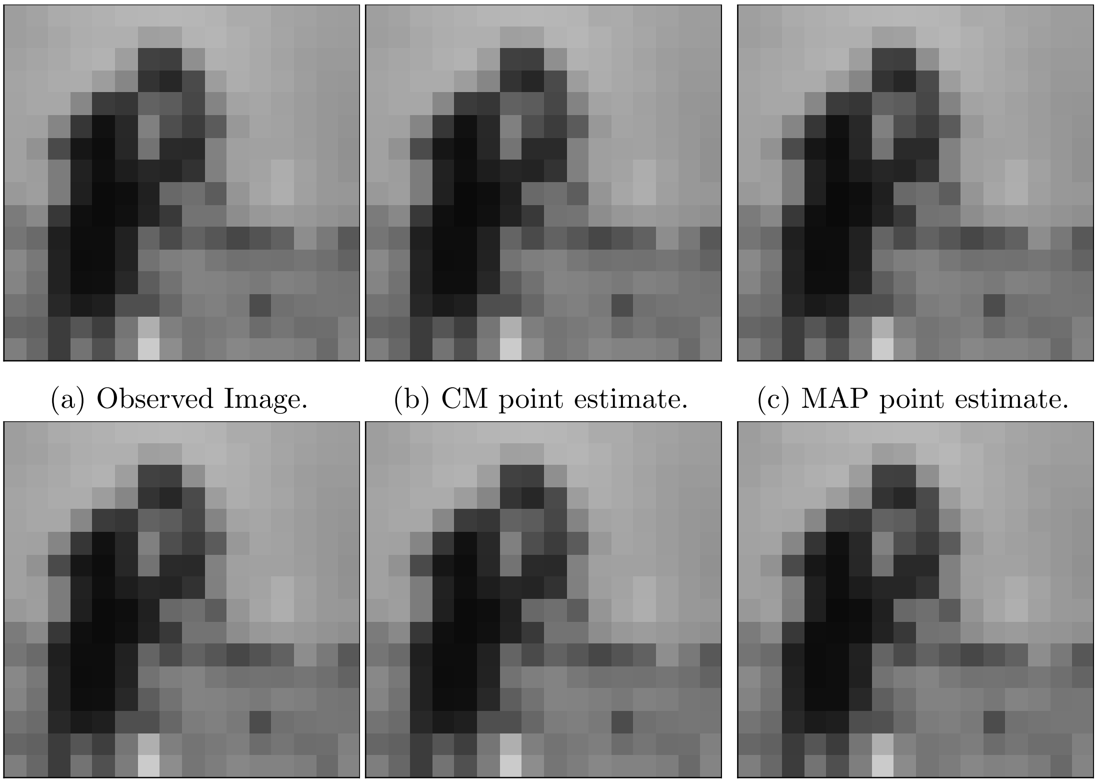
(b) Finding σ.

Figure 7.3: $\hat{\alpha}$ against different parameter values. Tuning parameters for (a) MH algorithm with Random Walk proposal and (b) N-Directional I-Jump sampler for $\pi(u|y)$ target distribution. We used 50 different values of β and σ between 0 and 0.8, for each parameter we ran the algorithms for 1000 iterations.

	CM	**MAP**
MH algorithm with Gaussian proposal	544.04	543.92
N-Directional I-Jump sampler	542.40	542.22

Table 7.1: Comparison between MH algorithm and N-Directional I-Jump sampler. MSE for each of the point estimates with respect to the unknown scaled image u, Figure 7.2 (b). For image de-noising.

We can see form Table 7.1 that MH algorithm has a larger MSE value for the CM point estimate and the same MAP compared to the observed image y. The MAP has the same value as the observed image y since it was the position from where the algorithm started. This tells us that the MH algorithm in fact moved away from our unknown image u, thus producing estimates $u\prime$ that where less similar to the unknown image u. In contrast, we can see that the N-Directional I-Jump sampler in fact produced smaller MSE values for both the point estimates compared to the observed image y. We can therefore conclude that the N-Dimensional sampler indeed moved towards the direction of the unknown image u. We will discuss this in depth in section 7.5. What is important to note is that all the point estimates produced by the samplers are similar to the observed image y. This can be seen in Figure 7.4. We know from Figure 7.3 that the step sizes are small for both algorithm and so they cannot explore the state space well thus remaining close to the initial image.



(a) Observed Image.

(b) CM point estimate.

(c) MAP point estimate.

(d) Observed Image.

(e) CM point estimate.

(f) MAP point estimate.

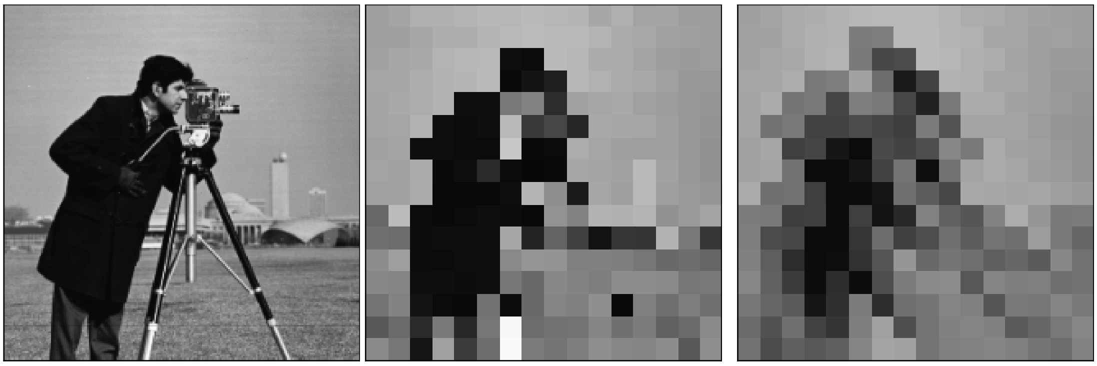
Figure 7.4: Point estimates when produced by MCMC methods for image denoising. *Top row:* Comparing the observed image (a) to the point estimates CM (b) and MAP (c) found using MH algorithm with Random Walk proposal. *Bottom row:* Comparing the observed image (d) to the point estimates CM (e) and MAP (f) found using the N-Directional I-Jump sampler.

7.4.2 Solution to Image De-Blurring

We will now work with image de-blurring. We first need to define the blurring operator A we will use in this example. In this case the blurring operator is:

$$A = \begin{bmatrix} \frac{1}{3} & 0 & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & 0 & \frac{1}{3} \end{bmatrix} \tag{7.9}$$

Like before, once we have scaled down our unknown image we then can apply the blurring operator A to each pixel to create the observed image y. Notice that in this case there is no additive noise. As a result, the inverse problem is to recreate the unknown scaled image u, Figure 7.5 (b) given the observed image y, Figure 7.5 (c). Like before we initialize the samplers to start from the given observed image y.



(a) Unknown Image. (b) Unknown Scaled Image. (c) Observed Image.

Figure 7.5: Scaling our unknown image (a) into one with smaller dimensions (b). In this case we have scaled the unknown image by $\frac{1}{16}$. The unknown image is an 256 by 256 pixels the scaled one is now 16 by 16 pixels. Finally the observed image y (c) is the scaled unknown image u (b) after applying the blurring operator A, defined in equation 7.9.

Again, we tune the parameters such that the average acceptance ratio $\hat{\alpha} \approx 0.234$, for consistency. We have decided to omit the graphs in this section since for both imaging problems we found that the parameters where the same for both problems. In particular, for the MH algorithm $\beta = 0.2$ and for the N-Directional I-Jump sampler we found $\sigma \approx 0.15$ and the period to be 10. This is expected due to the similarities between the problems.

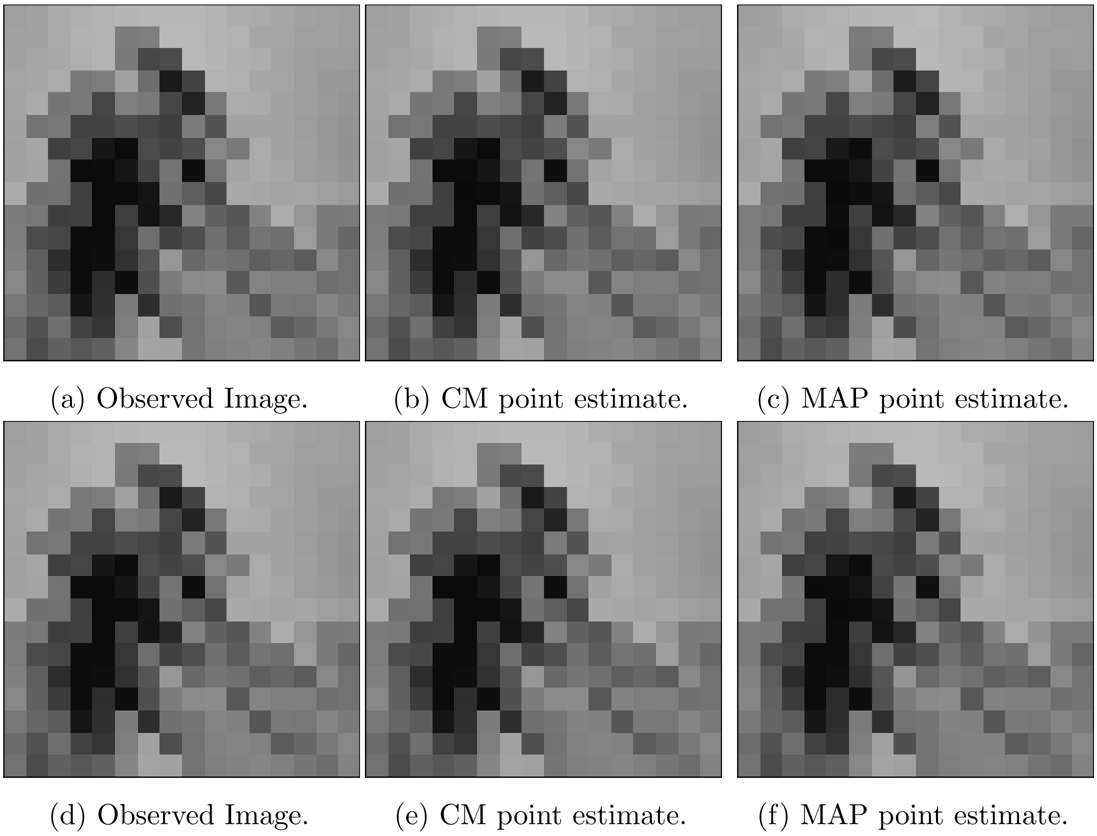
Furthermore, the MSE between the unknown image u and observed image y is **1207.97**, to 2 decimal places. This means that if we can get a MSE value smaller than **1207.97** we are moving towards the unknown image u.

	CM	**MAP**
MH algorithm with Gaussian proposal	1208.23	1207.97
N-Directional I-Jump sampler	1206.58	1202.02

Table 7.2: Comparison between MH algorithm and N-Directional I-Jump sampler. In particular we calculate the MSE for each of the point estimates with respect to the unknown scaled image, Figure 7.5 (b). For image de-blurring.

We can see form Table 7.2 that, like before, the MH algorithm has a larger MSE value for CM estimate and the MAP estimate is the same as the MSE of the observed image y and the unknown image u. This suggests that the MH algorithm in fact moved away from our target image, thus producing estimates that where less similar to the unknown image u. Furthermore, the MAP was the same in fact the observed image y since we initialize the algorithm form this position thus all other samples proposed where in fact worse MAP estimates.

Furthermore, we can see that the N-Directional I-Jump sampler in fact produced smaller MSE values for both the point estimates compared to the observed image. Therefore the N-Directional sampler indeed moved towards the direction of the unknown image u. This was consistent for both types of problems. Suggesting that in fact the non reversible sampler in fact preformed better, but to what extent? Again, we need to highlight that all the point estimates produced for both samplers where similar to the observed image y, given in Figure 7.6.


(a) Observed Image. (b) CM point estimate. (c) MAP point estimate.

(d) Observed Image. (e) CM point estimate. (f) MAP point estimate.

Figure 7.6: Point estimates when produced by MCMC methods for image deblurring. *Top row:* Comparing the observed image (a) to the point estimates CM (b) and MAP (c) found using MH algorithm with Random Walk proposal. *Bottom row:* Comparing the observed image (a) to the point estimates CM (b) and MAP (c) found using the N-Directional I-Jump sampler.

7.5 Takeaways

This problem has highlighted the difficulty of sampling form high dimensional target distributions. The point estimates produced for both problems remained close to the observed image y. We know that by the Theorem 4.3.1, we would expect to sample from the target distribution. However, we need to ask ourselves if we could preform more iterations, will this yield better solutions or should we be looking else where to solve these types of problems? Let us begin with the first question: *will preforming more iterations will result in better solutions?*

Recall that both algorithms had a **small** step size when tuned. However, this is not ideal, since we would like to explore the state space, thus in both cases we remained within a close range of the observed image y, since this is where we initialized the algorithms from. For the MH algorithm with a Random Walk proposal, we know that for any value of β we have that $\hat{\alpha} \to 0$ as dimension $d_x \to \infty$ [12], thus always remains close to the observed image y. In fact, this explains the results from tables 7.1 and 7.2. As a result, this rules out the MH algorithm, since we know it will not produce better results with more iterations.

We can then shift our focus to the N-Directional I-Jump sampler. Notice that the step size is small, again due to the *curse of dimensionality* it is difficult to take large step sizes, as seen by Figure 7.3 (b). So lets us discuss results from tables 7.1 and 7.2. In both case the N-Directional I-Jump sampler produced smaller MSE values for the point estimates than the original image y. This is due to the fact that the algorithm randomly picks a d-direction auxiliary variable $\boldsymbol{y}$. So, it was in fact fortunate to pick $\boldsymbol{y}$ such that it proposed samples in areas of higher density, resulting in better estimates. This motivates us to methodically pick $\boldsymbol{y}$ rather than randomly. In many cases, moving in a certain direction will not produce better samples. Thus the algorithm spends time proposing samples that will be rejected. This explains why in this example periodically changing $\boldsymbol{y}$ every 10 iterations worked best. Since it allowed the algorithm to explore many different directions in this space.

As a result, I suggest an alternative heuristic for changing $\boldsymbol{y}$. We dynamically increase or decrease the periodicity when the algorithm is accepting more samples, in a given direction $\boldsymbol{y}$. This will allow it to continue moving in this direction for a longer period of time. In turn, converging towards the stationary distribution of the chain. Since we know that “there is always a direction of exploration, $\boldsymbol{y}$, that enjoys most of the benefits from irreversibility” [25]. An alternative, is to incorporate gradients information such that we move towards the target distribution, in a similar fashion in which the MALA proposal works.

Therefore, more iterations could produce better estimates if we can appropriately explore the state space. However, there could be better methods to solve imaging problems since these problems are generally high dimensional spaces, and sampling methods struggle to explore them. This takes us to the second question. *Should we be looking else where to solve these types of problems?* The optimisation approach has been shown to solve such problems. In [43], the Bayesian framework is used in union with the optimisation approach where they aim to minimize the negative log posterior probability density function $\pi(u|y) = L(y|u)\pi_{pr}^{TV}(u)$. They incorporate the prior knowledge as a the regularisation term, in the paper they used various different priors, including the Total Variation prior. Solutions of the where found by solving:

$$\hat{u} = \text{arg min}_u \frac{1}{2}||y - Au||_2^2 + \lambda(\pi_{VTpr}(u)) \tag{7.10}$$

where $\hat{u}$ is the reconstructed image and λ is the regularization parameter. Therefore, even though there are several advantages of using the Bayesian inference approach to solve inverse problems, we need to make sure to choose most appropriate approach for the problem. In this case the optimisation approach is suitable since we require a point estimate, the restored image.

Note that all plots and functions were implemented from scratch in Python. As well as the images. These functions can be found in Appendix D.

Chapter 8

Conclusion

We have come to the end of the project. We began by introducing inverse problems. Throughout the project we highlighted different methods of solving inverse problems, then shifted our attention to MCMC methods, where we distinguished between reversible and non reversible samplers. We then explored a range of different target distributions comparing performances between the two types of samplers. Finally, we put all of these concepts together to solve inverses problems in imaging. As a result, we can look back at our project objectives to see how we have completed them:

1. In Chapters 1 and 2 we introduced inverse problems are and how we can solve them. We dove into reformulating inverse problems such that they could be solved with the Bayesian inference approach.

2. In Chapter 3, we motivated the reader to understand why one would want to use MCMC methods and described in depth what they are and how they work, linking this back to inverse problems.

3. In Chapter 3 and 6 we introduced the Metropolis Hastings algorithm and walked through several detailed examples to show the importance of tuning its parameters.

4. In Chapter 4, we introduced non reversible samplers and motivated the reader to use them. We then gave details on how to implement them. In Chapter 6, we illustrated the benefits of these algorithms through several examples. We also indicated areas of future research for non reversible samplers, which could be beneficial for the larger field.

5. Finally, in Chapter 7 we showed how we can apply all we have learnt throughout the project to solve inverse problems in imaging, motivating the reader to understand why these are important problems to solve.

It is important to understand that the techniques covered in this project are only the surface of two extensive fields. Perhaps, more importantly, is the significance of these methods. We have seen that they appear in a range of different disciplines where they attempt to solve real world problems.

Bibliography

[1] M. Allaby, *A dictionary of geology and earth sciences*, Oxford University Press, 2013, p. 229.

[2] T. Antal, *Stochastic modelling.* University Lecture Notes, 2019.

[3] P. Argoul, *Overview of inverse problems*, PhD thesis, Modes, 2012.

[4] J. M. Bardsley and J. Kaipio, *Gaussian markov random field priors for inverse problems*, Inverse Problems & Imaging, 7 (2013), pp. 397–416.

[5] M. Bertero and M. Piana, *Inverse problems in biomedical imaging: modeling and methods of solution*, in Complex Systems in Biomedicine, Springer, 2006, pp. 1–33.

[6] J. Bierkens, *Non-reversible metropolis-hastings*, Statistics and Computing, 26 (2016), pp. 1213–1228.

[7] I. Bilionis and N. Zabaras, *Solution of inverse problems with limited forward solver evaluations: a bayesian perspective*, Inverse Problems, 30 (2013), pp. 32–64.

[8] J. M. Bioucas-Dias and M. A. Figueiredo, *Multiplicative noise removal using variable splitting and constrained optimization*, IEEE Transactions on Image Processing, 19 (2010), pp. 1720–1730.

[9] T. Bocky, *What is autocorrelation?* https://www.displayr.com/autocorrelation/, 2020. Website.

[10] W. M. Bolstad and J. M. Curran, *Introduction to statistical science*, in Introduction to Bayesian statistics, John Wiley & Sons, 2016, pp. 6–7.

[11] G. Casella, C. P. Robert, and M. T. Wells, *Generalized accept-reject sampling schemes*, Lecture Notes-Monograph Series, (2004), pp. 342–347.

[12] S. L. Cotter, G. O. Roberts, A. M. Stuart, and D. White, *Mcmc methods for functions: modifying old algorithms to make them faster*, Statistical Science, (2013), pp. 424–446.

[13] B. N. Datta, *Stability, conditioning, and accuracy*, in Numerical linear algebra and applications, 2010.

[14] P. Diaconis, S. Holmes, and R. M. Neal, *Analysis of a non reversible markov chain sampler*, Annals of Applied Probability, (2000), pp. 726–752.

[15] H. W. Engl, M. Hanke, and A. Neubauer, *Regularisation operators*, in Regularization of inverse problems, Springer, 1996, pp. 49–55.

[16] A. Gelman, W. R. Gilks, and G. O. Roberts, *Weak convergence and optimal scaling of random walk metropolis algorithms*, The annals of applied probability, 7 (1997), pp. 110–120.

[17] A. Gelman, D. B. Rubin, et al., *Inference from iterative simulation using multiple sequences*, Statistical science, 7 (1992), pp. 457–472.

[18] M. Hairer, A. M. Stuart, S. J. Vollmer, et al., *Spectral gaps for a metropolis–hastings algorithm in infinite dimensions*, Annals of Applied Probability, 24 (2014), pp. 2455–2490.

[19] P. C. Hansen, J. G. Nagy, and D. P. O'leary, *Deblurring images: matrices, spectra, and filtering*, SIAM, 2006.

[20] W. K. Hastings, *Monte carlo sampling methods using markov chains and their applications*, (1970).

[21] D. J. Higham, *Condition numbers and their condition numbers*, Linear Algebra and its Applications, 214 (1995), pp. 193–213.

[22] J. Kaipio, *Modeling of uncertainties in statistical inverse problems*, in Journal of Physics: Conference Series, vol. 135, IOP Publishing, 2008, p. 012107.

[23] J. Kaipio and E. Somersalo, *Statistical and computational inverse problems*, Springer, 2006.

[24] S. S. Lee, *Markov chains on continuous state space.* University Lecture Notes, 2019.

[25] Y.-A. Ma, E. B. Fox, T. Chen, and L. Wu, *Irreversible samplers from jump and continuous markov processes*, Statistics and Computing, 29 (2019), pp. 177–202.

[26] V. Maz'ya and T. Shaposhnikova, *Jacques hadamard, a universal mathematician*, Bulletin American Mathematical Society, 36 (1999), pp. 95–97.

[27] 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 (1953), pp. 1087–1092.

[28] S. P. Meyn, R. L. Tweedie, et al., *Computable bounds for geometric convergence rates of markov chains*, The Annals of Applied Probability, 4 (1994), pp. 981–1011.

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

[30] J. Neyman, *Outline of a theory of statistical estimation based on the classical theory of probability*, Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences, 236 (1937), pp. 333–380.

[31] M. O'Searcoid, *Uniform continuity*, in Metric spaces, Springer, 2006, pp. 147–164.

[32] F. O'Sullivan, *A statistical perspective on ill-posed inverse problems*, Statistical science, 1 (1986), pp. 502–518.

[33] F. Pagani, M. Wiegand, and S. Nadarajah, *An n-dimensional rosenbrock distribution for mcmc testing*, arXiv preprint arXiv:1903.09556, (2019).

[34] Z. Pizlo, *Perception viewed as an inverse problem*, Vision research, 41 (2001), pp. 3145–3161.

[35] M. Razavy, *An Introduction to Inverse Problems in Physics*, World Scientific, 2020.

[36] S. Reuschen, F. Jobst, and W. Nowak, *Sequential pcn-mcmc, an efficient mcmc method for bayesian inversion of high-dimensional multi-gaussian priors*, arXiv preprint arXiv:2103.13385, (2021).

[37] D. A. Reynolds, *Gaussian mixture models*, Encyclopedia of biometrics, 741 (2009), pp. 659–663.

[38] C. Robert and G. Casella, *The metropolis—hastings algorithm*, in Monte Carlo Statistical Methods, Springer, 2013, pp. 231–283.

[39] ——, *Monte Carlo statistical methods*, Springer, 2013.

[40] G. O. Roberts and J. S. Rosenthal, *Harris recurrence of metropolis-within-gibbs and trans-dimensional markov chains*, The Annals of Applied Probability, (2006), pp. 2123–2139.

[41] G. O. Roberts, J. S. Rosenthal, et al., *General state space markov chains and mcmc algorithms*, Probability surveys, 1 (2004), pp. 20–71.

[42] J. S. Rosenthal, *Optimal proposal distributions and adaptive mcmc*, in Handbook of Markov Chain Monte Carlo, vol. 4, pp. 93–111.

[43] K. Sitara and S. Remya, *Image deblurring in bayesian framework using template based blur estimation*, The International Journal of Multimedia & Its Applications, 4 (2012), pp. 137–153.

[44] A. Sokal, *Monte carlo methods in statistical mechanics: foundations and new algorithms*, in Functional integration, Springer, 1997, pp. 131–192.

[45] A. R. Syversveen, *Noninformative bayesian priors. interpretation and problems with construction and applications*, Preprint statistics, 3 (1998), pp. 1–11.

[46] A. Teckentrup, *Introduction to the bayesian approach to inverse problems.* Lecture, 2020.

[47] ——, *Numerical linear algebra.* University Notes, 2020.

[48] L. Tierney, *Markov chains for exploring posterior distributions*, the Annals of Statistics, (1994), pp. 1701–1728.

[49] M. Tsagris, C. Beneki, and H. Hassani, *On the folded normal distribution*, Mathematics, 2 (2014), pp. 12–28.

[50] K. S. Turitsyn, M. Chertkov, and M. Vucelja, *Irreversible monte carlo algorithms for efficient sampling*, Physica D: Nonlinear Phenomena, 240 (2011), pp. 410–414.

[51] H. Xu, C. Caramanis, and S. Mannor, *Statistical optimization in high dimensions*, in Artificial Intelligence and Statistics, vol. 1, PMLR, 2012, pp. 1332–1340.

[52] N. Ye, F. Roosta-Khorasani, and T. Cui, *Optimization methods for inverse problems*, in 2017 MATRIX Annals, Springer, 2019, pp. 121–140.

Appendix A

Derivations

Derivation of example in section 3.4

We measure u directly with some additive noise $e \in E$. In this case the inverse problem is in the form of $y = u + e$. We want to determine the value of $u \in U$ from observed data $y \in Y$. We assume that $U \sim \mathcal{N}(0,1)$ and $E \sim \mathcal{N}(0,\ \sigma^2)$. The prior denoted as $\pi_{pr}(u)$ which has the form of a $\mathcal{N}(0,1)$ and the noise denoted as $\pi_{noise}(e)$ which has the form of a $\mathcal{N}(0,\ \sigma^2)$. We can get an explicit formula for the posterior distribution.

$$\begin{aligned}
\pi\,(u|y) &\propto \pi(y|u)\pi_{pr}(u) \\
&\propto \pi_{noise}(y-u)\pi_{pr}(u) \\
&= \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{y-u}{\sigma}\right)^2\right) \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}(u-0)^2\right) \\
&\propto \exp\left(-\frac{1}{2}\left(\frac{y-u}{\sigma}\right)^2\right) \exp\left(-\frac{1}{2}u^2\right) \\
\pi\,(u|y) &= \exp\left(-\frac{1}{2\sigma^2}(y-u)^2 - \frac{1}{2}u^2\right) \qquad (1)
\end{aligned}$$

Now we want to rewrite (1) such that it takes form of the probability density function of a normal distribution, so we have explicit expression for it. That is we want to find constants μ, K, and σ', such that:

$$\pi\,(u|y) = \exp\left(-\frac{1}{2(\sigma')^2}(u-\mu)^2 + K\right) \qquad (2)$$

To do this we focus on expression within the exponential of equation (1):

$$\begin{aligned}
-\frac{1}{2\sigma^2}(y-u)^2-\frac{1}{2}u^2 &= -\frac{1}{2\sigma^2}(y^2-2uy-u^2)-\frac{1}{2}u^2\\
&= -\frac{1}{2}\left(\frac{1}{\sigma^2}(y^2)-\frac{1}{\sigma^2}(2uy)+\frac{1}{\sigma^2}(u^2)+u^2\right)\\
&= -\frac{1}{2}\left(u^2\left(\frac{1}{\sigma^2}+1\right)-\frac{1}{\sigma^2}(2uy)+y^2\frac{1}{\sigma^2}\right)\\
-\frac{1}{2\sigma^2}(y-u)^2-\frac{1}{2}u^2 &= -\frac{1}{2}\left(u^2a-2bu+c\right) \qquad (3)
\end{aligned}$$

$$\text{Where we have that } a=\frac{1}{\sigma^2}+1, b=\frac{y}{\sigma^2}, c=y^2\frac{1}{\sigma^2}$$

To achieve the form we want (2) we can complete the square on the expression $(u^2a-2bu+c)$ given in equation (3).

$$\left(u^2a-2bu+c\right)=a(u^2-\frac{2bu}{a}+\frac{c}{a})=a(u-\frac{b}{a})^2+c-\frac{b^2}{a} \qquad (4)$$

We can now substitute the above expression (4) into the original expression (1) such that we get.

$$\pi\left(u|y\right)=\exp\left(-\frac{1}{2}\left(a(u-\frac{b}{a})^2+c-\frac{b^2}{a}\right)\right)=\exp\left(-\frac{a}{2}(u-\frac{b}{a})^2-\frac{1}{2}(c-\frac{b^2}{a})\right) \qquad (5)$$

The last expression (5) is exactly in the form we want it in. We can no compare it to the expression of probability distribution of a normal distribution, in order to get the explicit parameters we are interested in. In particular, the mean μ and variance $(\sigma')^2$ of the new function. To do this we need to find the constants μ, σ' and K. Recall that: $a=\frac{1}{\sigma^2}+1, b=\frac{y}{\sigma^2}, c=y^2\frac{1}{\sigma^2}$. First we find μ:

$$\mu=\frac{b}{a}=\frac{\frac{y}{\sigma^2}}{\frac{1}{\sigma^2}+1}=\frac{y}{\sigma^2}\frac{\sigma^2}{1+\sigma^2}=\frac{y}{1+\sigma^2} \qquad (6)$$

Now we can find $(\sigma')^2$ of the new function, notice that $-\frac{1}{2(\sigma')^2}=-\frac{a}{2}$ if we rearrange this equation and substitute a we get:

$$(\sigma')^2=\frac{1}{a}=\frac{1}{\frac{1}{\sigma^2}+1}=\frac{1}{\frac{1+\sigma^2}{\sigma^2}}=\frac{\sigma^2}{1+\sigma^2} \qquad (7)$$

Finally, we can find K:

$$\begin{aligned} K &= -\frac{1}{2}\left(c - \frac{b^2}{a}\right) \\ &= -\frac{1}{2}\left(y^2\frac{1}{\sigma^2} - \frac{(\frac{y}{\sigma^2})^2}{\frac{1}{\sigma^2}+1}\right) \\ &= -\frac{1}{2}\left(\frac{y^2}{\sigma^2} - (\frac{y}{\sigma^2})^2\frac{\sigma^2}{1+\sigma^2}\right) \\ K &= -\frac{1}{2}\left(\frac{y^2}{\sigma^2} - \frac{y^2\sigma^2}{\sigma^4+\sigma^6}\right) \end{aligned} \tag{8}$$

We can now completely define the new posterior distribution. This is because we know the values for the mean and standard deviation of this new function. We have that the posterior distribution can now be written as:

$$\begin{aligned} \pi\left(u|y\right) &= \exp\left(-\frac{a}{2}(u-\frac{b}{a})^2 - \frac{1}{2}(c-\frac{b^2}{a})\right) \\ &= \exp\left(-\frac{1}{2(\sigma')^2}(u-\mu)^2 + K\right) \\ \pi\left(u|y\right) &\propto \exp\left(-\frac{1}{2}\left(\frac{u-\mu}{\sigma'}\right)^2\right) \text{ where } \mu = \frac{y}{1+\sigma^2} \text{ and } (\sigma')^2 = \frac{\sigma^2}{1+\sigma^2} \end{aligned} \tag{9}$$

As a result we have completed our derivation where we have found an expression for $\pi\left(u|y\right)$ given in equation (9). We do not need to worry about the constant term K since it does not depend on u.

Appendix B

Markov chain Monte Carlo Methods

Python implementation of the Metropolis Hastings Algorithm in one and higher dimensions.

```python
import numpy as np
import pandas as pd
from scipy import stats
import random

class Reversible_Sampler():
    """
    Class definition for the reversible sampler.
    In both one and two dimensions.
    """

    def __init__(self, N, dimension, target_distribution,
   proposal):
        #Beta values which we get from the proposal
        self.beta = proposal.beta
        #N is the number of iterations the algorithm runs (int)
        self.N = N
        #dimension of the state space (int)
        self.dimension = dimension
        #proposal density defined using Class Proposal.
        self.proposal = proposal
        #target_distribution defined using the Class definitions
        #for target distributions.
        self.target_distribution = target_distribution

    def Reversible_Metropolis_Hastings_Algo(self):
        """
        Implementation of the Metropolis Hastings Algorithm.

        Output:
        -samples: List of samples which are integers or tuples
        -list_acceptance_probaility: List of acceptance
        probabilities used for tuning the algorithm.
        """

        # initialize in this case we set the initial sample
        # between 1 and 10 so has positive probability
        # can be any value with positive probability.
        if self.dimension == 1:
            x_0 = np.random.randint(1,10, size=1)[0]
            x_t = x_0
        # Initialise for higher dimension between 0 and 1
        # Again this can be any position.
        else:
            x_0 = list(np.random.rand(1,self.dimension)[0])
            x_t = x_0

        #list of samples generated by the algorithm
        samples = [x_0]
        #list of acceptance ratio at each iteration
        of the algorithm
        list_acceptance_probaility = []

        #We define two new variables such that they are the
        #Proposal and posterior distirbution of the problem
        posterior_distribution = self.target_distribution
        proposal = self.proposal

        #Begin iterations
        for t in range(self.N):

            #Sample a proposal x_prime from proposal at x_t
            x_prime = proposal.get_random_variable(x_t)

            #Proposal density at x_prime and x_t
            proposal_density_prime = proposal.get_dist(x_prime)
            proposal_density_t = proposal.get_dist(x_t)

            #Evaluation of the numerator of acceptance ratio
            numerator = posterior_distribution.pdf(x_prime)*float
(proposal_density_prime.pdf(x_t))
            #Evaluation of the denomiantor of acceptance ratio
            denominator = posterior_distribution.pdf(x_t)*float(
proposal_density_t.pdf(x_prime))

            #Calculate acceptance ratio
            if numerator == 0:
                acceptance_probaility = 0
            elif denominator == 0:
                acceptance_probaility = 1
            else:
                acceptance_probaility = float(min([1,numerator/
denominator]))

            #Add acceptance ratio to
            #list_acceptance_probaility
            list_acceptance_probaility.append(
acceptance_probaility)

            #Sample a random variable
            rv = np.random.uniform(0,1,1)

            #Accept or reject step
            if rv <= acceptance_probaility:
                #Accept: we update x_t to x_prime
                x_t = u_prime
                samples.append(x_t)
            else:
                #Reject: we remain at x_t
                x_t = x_t
                samples.append(x_t)

        #Return the samples and all the acceptance probabilities
        return samples, list_acceptance_probaility
```

Python implementation of the proposal densities used by the Metropolis Hastings algorithm.

```
import numpy as np
import pandas as pd
from scipy import stats
import random

class Proposal():
    """
    This class defines the proposals can be used for
    the reversible sampler, the Metropolis Hastings Algorithm.
    These need to be defined since they will be  used in the
    algorithm of choice.

    In particular we have 4 possible proposals:
    -Random walk
    -Random walk two dimensions
    -Random walk n dimensions
    -Pre-Condtioned Crank Nicolson
    """

    def __init__(self, beta, pick):
        """
        To initialize we need two parameters, first the parameter
        for the algorithm beta  (int) and we need to decide which
        proposal to pick (str).
        The pick variable can take 4 values:
        -Random walk: "random_walk"
        -Random walk two dimensions: "random_walk_2d"
        -Random walk n dimensions: "random_walk_nd"
        -Pre-Condtioned Crank Nicolson: "pCN"
        """
        self.beta = beta
        self.pick = pick

    def get_random_variable(self, x_t):
        """
        Depending on the proposal it returns a
        random variable given sample x_t.
        """
        if self.pick == "random_walk":
            return self.random_walk(x_t).rvs()
        if self.pick == "pCN":
            return self.pCN(x_t).rvs()
        if self.pick == "random_walk_2d":
            return self.random_walk_2d(x_t).rvs()
        if self.pick == "random_walk_nd":
            return self.random_walk_nd(x_t).rvs()

    def get_dist(self, u_t):
        """
        Depending on the proposal it returns the
        distribution of the proposal.
        """
        if self.pick == "random_walk":
            return self.random_walk(u_t)
        if self.pick == "pCN":
            return self.pCN(u_t)
        if self.pick == "random_walk_2d":
            return self.random_walk_2d(u_t)
        if self.pick == "random_walk_nd":
            return self.random_walk_nd(u_t)

    def random_walk(self, x_t):
        """
        Definition of random walk distribution
        centred at sample x_t.
        """
        return stats.norm([x_t],[self.beta**2])

    def pCN(self, x_t):
        """
        Definition of pCN distribution
        centred at sample x_t.
        """
        return stats.norm([np.sqrt(1-self.beta**2)*x_t],[8*self.beta**2])

    def random_walk_2d(self, x_t):
        """
        Definition of random walk two dimesnional
        distribution centred at sample x_t.
        """
        return stats.multivariate_normal([int(x_t[0]),int(x_t[1])],[[self.beta**2,0],[0,self.beta**2]])

    def random_walk_nd(self, x_t):
        """
        Definition of random walk n dimensional
        distribution centred at sample x_t.
        """
        n = len(x_t)
        return stats.multivariate_normal(x_t, np.eye(n)*(self.beta**2))
```

Python implementation of the One-Directional I-Jump sampler and N-Directional I-Jump sampler.

```
import numpy as np
import pandas as pd
```

```python
from scipy import stats
import random

class Non_Reversible_Sampler():
    """
    Class definition for the non reverisble sampler.
    Both in one dimension and n dimensions.
    """
    def __init__(self, alpha, beta, N, dimension, target_distribution):

        #Alpha for the gamma distrbution (a)
        self.alpha = alpha
        #Beta for the gamma distrbution (b) / also used as sigma for n dimensional
        self.beta = beta
        #N is the number of iterations the algorithm runs (int)
        self.N = N
        #dimension is the dimension of the state space.
        self.dimension = dimension
        #target_distribution defined using the Class definitions
        #for target distributions.
        self.target_distribution = target_distribution

    def One_Directional_I_Jump_Sampler(self):
        """
        Implementation of the One-Directional I-Jump sampler.

        Output:
        -samples: List of samples which are integers.
        -list_acceptance_probaility: List of acceptance
        probabilities used for tuning the algorithm.
        """
        # initialize in this case we set the initial sample
        # between 1 and 10 so has positive probability
        # can be any value with positive probability.
        x_0 = np.random.randint(1,10, size=1)[0]
        x_t = x_0

        #list of samples generated by the algorithm
        samples = [x_t]
        #list of acceptance ratio at each iteration of the
        #algorithm
        list_acceptance_probaility = []

        #We define variables such that they are the
        #posterior distribution of the problem
        posterior_distribution = self.target_distribution

        #We initialize the z indicator random variable
        rv_z = np.random.uniform(0,1,1)
        z = 0
        if rv_z < 0.5:
            z = 1
        else:
            z = -1

        #defining the gamma distirbution used for the proposal of
        #samples.
        gamma_distribution = stats.gamma(a=self.alpha, scale=1/self.beta)

        #Begin iterations
        for t in range(self.N):

            #Indicator variable direction to move in.
            if z > 0:

                #Sample a proposal x_prime from proposal at x_t
                #(Move to the right)
                x_prime = x_t + gamma_distribution.rvs()

                # numerator of acceptance ratio
                numerator = posterior_distribution.pdf(x_prime)*gamma_distribution.pdf(x_prime - x_t)

                #denominator of acceptance ratio
                denominator = posterior_distribution.pdf(x_t)*gamma_distribution.pdf(x_prime - x_t)

                #Calculate acceptance proability
                if numerator == 0:
                    acceptance_probaility = 0
                elif denominator == 0:
                    acceptance_probaility = 1
                else:
                    acceptance_probaility = float(min([1, numerator/denominator]))

            else:

                # sample a proposal x_prime from proposal at x_t
                # (Move to the left)
                x_prime = x_t - gamma_distribution.rvs()

                #numerator of acceptance ratio
                numerator = posterior_distribution.pdf(x_prime)*gamma_distribution.pdf(x_t - x_prime)

                #denominator of acceptance ratio
                denominator = posterior_distribution.pdf(x_t)*gamma_distribution.pdf(x_t - x_prime)

                # Calculate acceptance proability
                if numerator == 0:
                    acceptance_probaility = 0
                elif denominator == 0:
                    acceptance_probaility = 1
                else:
                    acceptance_probaility = float(min([1, numerator/denominator]))

```

```
         #Add acceptance ratio to
         #list_acceptance_probaility
         list_acceptance_probaility.append(
acceptance_probaility)

         #Sample a random variable [0,1]
         rv = np.random.uniform(0,1,1)

         #Accept or reject step
         if rv <= acceptance_probaility:
             #Accept update x_t to x_prime
             x_t = x_prime
             samples.append(x_t)
             #leave z unchanged
             z = z
         else:
             #Reject remain at x_t
             x_t = x_t
             samples.append(x_t)
             #update the indicator z
             z = -1*z

     #Return the samples and all the acceptance probabilities
     return samples, list_acceptance_probaility

 def N_Directional_I_Jump_Sampler(self, period):
     """
     Implementation of the N-Directional I-Jump sampler.

     Input:
     -period: periodicity to change the direction of auxiliary
     variable y_p. (int)

     Output:
     -samples: List of samples which are integers.
     -list_acceptance_probaility: List of acceptance
     probabilities used for tuning the algorithm.
     """

     #Initialize at a random position
     x_0 = list(np.random.rand(1,self.dimension)[0])
     x_t = x_0

     #Define the possible directions we can move in
     direction_vectors = np.eye(self.dimension)
     #Pick a random direction for auxiliary variable y_p
     y_p = direction_vectors[np.random.randint(self.dimension)
]

     #list of samples generated by the algorithm
     samples = [x_t]
     #list of acceptance ratio at each
     #iteration of the algorithm
     list_acceptance_probaility = []

     #We define variables such that they are the
```

```
        #posterior distribution of the problem
        posterior_distribution = self.target_distribution

        #We initialize the z indicator random variable
        rv_z = np.random.uniform(0,1,1)
        z = 0
        if rv_z < 0.5:
            z = 1
        else:
            z = -1

        #Define sgn function which calculates
        #the dot product of 2 vectors
        #returns 1 greater than 0 or -1 otherwise
        def sgn(eta,y_p):
            x = np.dot(eta,y_p)
            if x >= 0:
                return 1
            else:
                return -1

        #Define a multivariate normal distribution
        #used to propose samples
        multivariate_normal = stats.multivariate_normal(np.zeros(self.dimension),self.beta**2*np.eye(self.dimension))

        #Define a half space multivariate Gaussian distribution
        #f for one direction
        def f_pdf(z, y, y_p):
            x = np.dot(y_p,z-y)
            if x >= 0:
                return (2/(2*np.pi*self.beta)**(self.dimension/2)*np.exp((-1/2)*np.linalg.norm(z-y)**2))
            else:
                return 0

        #Define a half space multivariate Gaussian distribution
        #g for other direction
        def g_pdf(z, y, y_p):
            x = np.dot(y_p,z-y)
            if x >= 0:
                return 0
            else:
                return (2/(2*np.pi*self.beta)**(self.dimension/2)*np.exp((-1/2)*np.linalg.norm(z-y)**2))

        #Begin iterations
        for t in range(self.N):

            # Change direction y_p periodically
            if t%period == 0:
                y_p = direction_vectors[np.random.randint(self.dimension)]

            #Indicator variable telling us in
            #what direction to move in.
            if z > 0:

                #Create a multivariate Gaussian distribution
                #random variable
                eta = multivariate_normal.rvs()
                #Sample a proposal x_prime from
                #proposal at x_t with f.
                x_prime = x_t + sgn(eta,y_p)*eta

                # numerator of acceptance ratio
                numerator = posterior_distribution.pdf(x_prime)*g_pdf(x_t, x_prime, y_p)

                # denominator of acceptance ratio
                denominator = posterior_distribution.pdf(x_t)*f_pdf(x_prime, x_t, y_p)

                #Calculate acceptance proability
                if numerator == 0:
                    acceptance_probaility = 0
                elif denominator == 0:
                    acceptance_probaility = 1
                else:
                    acceptance_probaility = float(min([1, numerator/denominator]))

            else:

                #Create a normal distribution random variable
                eta = multivariate_normal.rvs()
                #Sample a proposal x_prime from
                #proposal at x_t with g.
                x_prime = x_t - sgn(eta,y_p)*eta

                #numerator of acceptance ratio
                numerator = posterior_distribution.pdf(x_prime)*f_pdf(x_t,x_prime, y_p)

                #denominator of acceptance ratio
                denominator = posterior_distribution.pdf(x_t)*g_pdf(x_prime, x_t, y_p)

                #Calculate acceptance proability
                if numerator == 0:
                    acceptance_probaility = 0
                elif denominator == 0:
                    acceptance_probaility = 1
                else:
                    acceptance_probaility = float(min([1, numerator/denominator]))

            #Add acceptance ratio to
            #list_acceptance_probaility
            list_acceptance_probaility.append(acceptance_probaility)

```

```
            #Sample a random variable [0,1]
            rv = np.random.uniform(0,1,1)

            #Accept or reject step
            if rv <= acceptance_probaility:
                #Accept
                x_t = x_prime
                samples.append(x_t)
                #leave z unchanged
                z = z
            else:
                #Accept update x_t to x_prime
                x_t = x_t
                samples.append(x_t)
                #update the indicator z
                z = -1*z

        #return the samples and all the acceptance probabilities
        return samples, list_acceptance_probaility
```

Python target distributions used by both Reversible and Non Reversible Samplers.

```
import numpy as np
import pandas as pd
from scipy import stats
import random

class Normal_Distribution():

    def __init__(self, mu, sigma):
        """
        mu = mean of distribution (int)
        sigma = standard deviation of distribution (int)
        """
        self.mu = mu
        self.sigma = sigma

    def pdf(self, x):
        """
        Returns pdf value at x. Where x is the
        value we want to evaluate
        """
        from scipy import stats
        return stats.norm([self.mu],[self.sigma]).pdf(x)[0]

class Log_Normal_Distribution():

    def __init__(self, mu, sigma):
        """
        mu = mean of distribution (int)
        theta = standard deviation of distribution (int)
        """
        self.mu = mu
        self.sigma = sigma

    def pdf(self, x):
        """
        Returns pdf value at x. Where x is the
        value we want to evaluate.
        """
        from scipy import stats
        return stats.lognorm([self.sigma],loc=self.mu).pdf(x)[0]

class Gaussian_Mixture():

    def __init__(self, pi_s):
        """
        pi_s = pi_s is a list of normal
        distributions made with stats.norm([mu],[sigma])
        """
        self.pi_s = pi_s

    def pdf(self, x):
        """
        Returns pdf value at x. Where x is the
        value we want to evaluate.
        """
        return sum([pi.pdf(x) for pi in self.pi_s])

class Multivariate_Gaussian():

    def __init__(self, mu, covariance):
        """
        mu = mean of distribution (list of int). Example: [10,5]
        covariance = covariance matrix (list of lists of int).
        Example: [[1,0],[0,1]]
        """
        self.mu = mu
        self.covariance = covariance

    def pdf(self, x):
        """
        Returns pdf value at x. Where x is
        the value we want to evaluate.
        """
        from scipy import stats
        return stats.multivariate_normal(self.mu,self.covariance)
   .pdf(x)

class Rosenbrook_2D_Function():

    def __init__(self, coef_1):
        """
        coeff_1 = Scaling variable (int)
        """
        self.coef_1 = coef_1

    def pdf(self, x):
        """
        Returns pdf value at x. Where x is the
        value we want to evaluate.
        """
        return np.exp(-((100*(x[1] - x[0]**2)**2 + (1 - x[0])**2)
    /self.coef_1))
```

Functions used to be able to calculate the normalized autocorrelation which we later plotted in Chapter 6.

```
class Variance_Analysis():
    """
    Class used for the variance analysis. In particular
    this class lets us calculate:
    -the normalized autocorrelation function for
    different time lags
    -asymptotic variance
    -integrated autocorrelation
    -variance of estimator
    """

    def __init__(self, list_samples):
        """
        We need list_samples. This is a list of samples
        created by an algorithm.
        """
        self.list_samples = list_samples
        #Number of samples
        self.N = len(self.list_samples)

    def c_tau_0(self):
        """
        Calculates the variance of the samples, that is the
        covariance at lag 0.
        Output: variance of samples (int)
        """
        #Mean of samples
        u_mean = np.mean(self.list_samples)
        #Number of samples
        N = self.N
        #Calculate the sum of the covariance between the samples.
        sum_covariance = sum([np.dot((self.list_samples[i]-u_mean
    ),(self.list_samples[i]-u_mean)) for i in range(0,N)])

        return sum_covariance/N

    def c_tau(self, tau):
        """
        Calculates the covariance of the samples, at any time lag
    .
        Input: tau (int) covariance at the given time lag
        Output: covariance of samples (int)
        """
        #Mean of samples
        u_mean = np.mean(self.list_samples)
        #Normalizing constant c(0)
        c_0 = self.c_tau_0()
        #Number of samples
        N = self.N
        #Calculate the sum of the covariance between the samples.
        sum_covariance = sum([np.dot((self.list_samples[i]-u_mean
),(self.list_samples[i+tau]-u_mean)) for i in range(0,self.N-tau)])

        return (sum_covariance/(self.N-tau))/c_0

    def integrated_auto_correlation(self, M):
        """
        Calculates integrated autocorrelation time for a given M.
        """
        #Normalizing constant c(0)
        c_0 = self.c_tau_0()
        #Returns the sum up to M
        return 1 + 2*sum([self.c_tau(i)/c_0 for i in range(1,M+1)])

    def asymptotic_variance(self, M):
        """
        Calculates asymptotic variance for a given M.
        """
        #Calculates variance of samples
        u_variance = np.var(self.list_samples)
        #Returns the sum up to M
        return u_variance + 2*sum([self.c_tau(i) for i in range(1,M+1)])

    def variance_of_estimator(self, M):
        #Calculates estimate of variance of the estimator.
        return self.asymptotic_variance(M)/self.N
```

Appendix C

Plotting Functions

Functions and Classes used to produce the graphs in Chapter 6.

```python
import numpy as np
import pandas as pd
from scipy import stats
import random

class Graph_Maker():
    """
    This class is used to make the graphs for a given set of
    samples. In particular it can:
    -plot 1d samples on a graph.
    -plot 1d samples that come form a Gaussian distribution.
    -plot 2d samples that come form a 2d Gaussian distribution.
    -plot samples from a multivariate Gaussian distribution.
    -plot Rosebrock functions, with and without points
    -finally it can produce the histograms for the given
    distribution the samples come from.

    When using the functions one needs to pick the most
    appropriate for the samples they have. If one has 1d
    sample from a Gaussian distribution you should use the
    functions that are appropriate for this type of sample.
    """

    def __init__(self, samples):
        #To initalzie we need the samples we wish to plot.
        self.samples = samples
        self.N = len(samples)

    def plot_samples(self, file_name):
        """
        This function scatter the samples so one can
        visualize the graph created. Appropriate for
        any 1 dimensional set of samples.
        Input:
        -file_name: name of the file we wish to save (string)
        Outputs:
        Plots from the samples given.
        """
        #Sample index for the x-axis
        sample_index = [i for i in range(self.N)]

        #Create the plots
        fig, ax = plt.subplots(figsize=(20, 10))
        #Plot and scatter the points
        ax.plot(sample_index,self.samples)
        ax.scatter(sample_index,self.samples)

        #Set the lables of the axis
        ax.set_ylabel(r'Sample value $x_i$')
        ax.set_xlabel(r'Sample index $i$')

        #Save the figure with the name given
        plt.savefig('{}.pdf'.format(file_name), bbox_inches = 'tight', pad_inches = 0)

    def plot_1D_gaussian(self, mean, sd, burn_in, file_name):
        """
        This function plots the samples from a one dimensional
        Gaussian distribution.

        Inputs:
        -mean: mean of distribution we are sampling from (float)
        -sd: standard deviation of distribution we are
        sampling from (float)
        -burn_in: indicates if we want to highlight first point
        which is within 2 standard deviations of the mean (bool)
        -file_name: name of the file we wish to save (string)
        Outputs:
        Plots from the samples given.
        """
        #If true we show burn-in analysis
        if burn_in == True:

            #Sample index for the x-axis
            sample_index = [i for i in range(self.N)]

            #Create the plots
            fig, ax = plt.subplots(figsize=(20, 10))
            #Plot and scatter the points
            ax.plot(sample_index,self.samples)
            ax.scatter(sample_index,self.samples)

            #Finding index of fist value within 2sd of the mean
            index = 0
            for u in self.samples:
                index+=1
                if(u > mean-2*sd):
                    break

            #Vertical line to show the index of
            #fist value within 2sd of the mean
            ax.vlines(index, min(self.samples), max(self.samples), "g",'--',label = r"First value of $x_t = x_{{{}}}$ which is in $\mu \pm 2\sigma$".format(index))

            #Horizontal lines showing the mean,
            #and +- 2sd from the mean
            ax.hlines(mean, 0, self.N, "r",label = r"$\mu$ of underlying target distriution $\pi$")
            ax.hlines(mean-2*sd, 0, self.N, "b",'--',label = r"$\mu \pm 2\sigma$")
            ax.hlines(mean+2*sd, 0, self.N, "b",'--')

            #Set the lables of the axis
            ax.set_ylabel(r'Sample value $x_i$')
            ax.set_xlabel(r'Sample index $i$')
            #Add a legend
            ax.legend(loc="lower right")

            #Save the figure with the name given
            plt.savefig('{}.pdf'.format(file_name), bbox_inches = 'tight', pad_inches = 0)

        #We do not show the burn-in analysis
        else:

            #Sample index for the x-axis
            sample_index = [i for i in range(self.N)]

            #Create the plots
            fig, ax = plt.subplots(figsize=(20, 10))
            #Plot and scatter the points
            ax.plot(sample_index,self.samples)
            ax.scatter(sample_index,self.samples)

            #Horizontal lines showing the mean,
            #and +- 2sd from the mean
            ax.hlines(mean, 0, len(self.samples), "r",label = r"$\mu$ of underlying target distriution $\pi$")
            ax.hlines(mean-2*sd, 0, self.N, "b",'--',label = r"$\mu \pm 2\sigma$")
            ax.hlines(mean+2*sd, 0, self.N, "b",'--')

            #Set the lables of the axis
            ax.set_ylabel(r'Sample value $x_i$')
            ax.set_xlabel(r'Sample index $i$')
            #Add a legend
            ax.legend()

            #Save the figure with the name given
            plt.savefig('{}.pdf'.format(file_name), bbox_inches = 'tight', pad_inches = 0)

    def plot_gaussian_mixture_models(self, mean, sd, file_name):
        """
        This function plots the samples from a Gaussian
        mixture models. With 2 Gaussian.

        Inputs:
        -mean: mean of distribution we are sampling
        from (list of floats)
        -sd: standard deviation of distribution we are
        sampling from (list of floats)
        -file_name: name of the file we wish to save (string)
        Outputs:
        Plots from the samples given.
        """
        #Sample index for the x-axis
        sample_index = [i for i in range(self.N)]

        #Create the plots
        fig, ax = plt.subplots(figsize=(20, 10))
        #Plot and scatter the points
        ax.plot(sample_index,self.samples)
        ax.scatter(sample_index,self.samples)

        #Horizontal lines showing the mean,
        #and +- 2sd from the mean for first gaussian
        ax.hlines(mean[0], 0, self.N, "r",label = r"$\mu$ of underlying target distriution $\pi_1$")
        ax.hlines(mean[0]-2*sd[0], 0, self.N, "b",'--',label = r"$\mu \pm 2\sigma$")
        ax.hlines(mean[0]+2*sd[0], 0, self.N, "b",'--')

        #Horizontal lines showing the mean,
        #and +- 2sd from the mean for second gaussian
        ax.hlines(mean[1], 0, self.N, "b",label = r"$\mu$ of underlying target distriution $\pi_2$")
        ax.hlines(mean[1]-2*sd[1], 0, self.N, "b",'--')
        ax.hlines(mean[1]+2*sd[1], 0, self.N, "b",'--')

        #Set the labels of the axis
        ax.set_ylabel(r'Sample value $x_i$')
        ax.set_xlabel(r'Sample index $i$')

        #Save the figure with the name given
        plt.savefig('{}.pdf'.format(file_name), bbox_inches = 'tight', pad_inches = 0)

    def plot_histogram_mix(self, pi_s, file_name):
        """
        This function plots the histograms for Gaussian
        mixture models.

        Inputs:
        -pi_s: are defined by using a stats.norm
        function and the put into a list.
        Example:
            pi_1 = stats.norm([-10],[1])
            pi_2 = stats.norm([10],[1])
            pi_s = [pi_1, pi_2]
        -file_name: name of the file we wish to save (string)
        Outputs:
        Plots from the samples given.
        """

        #How generate samples for the analytical distribution
        xs = np.linspace(pi_s[0].mean()[0] - 6*pi_s[0].std()[0], pi_s[1].mean()[0] + 6*pi_s[1].std()[0], 500)

        #Calculate the analytical distribution need
        #to use the Gaussian_Mixture Class.
        posterior_analytical = [Gaussian_Mixture(pi_s).pdf(x)/2 for x in xs]

        #Create the plot
        fig, ax = plt.subplots(figsize=(20, 10))
        #Plot the histogram
        plt.hist(self.samples,density=True,bins=50, label = "Estimated target distribution", alpha = 0.5, edgecolor='#169acf')

        #Plot the analytical distribution
        plt.plot(xs, posterior_analytical, label = "Analytical target distribution")

        #Set the labels of the axis
        ax.set_ylabel('Probability density')
        #Adding the legend
        ax.legend()
        #Save the figure with the name given
        plt.savefig('{}.pdf'.format(file_name), bbox_inches = 'tight', pad_inches = 0)


    def plot_2D_gaussian(self, mu, sigma, file_name, plot_type):
        """
        This function plots the samples from a 2d Gaussian distribution.

        Inputs:
        -mu: vector of means of distribution (list of floats)
        -sigma: matrix of standard deviations of
        distribution (list of lists floats)
        -file_name: name of the file we wish to save (string)
        -plot_type: if 1 we plot the points otherwise we
        create a scatter plot(int)
        -file_name: name of the file we wish to save (string)
        Outputs:
        Plots from the samples given.
        """
        #change from lists ot np arrays.
        mu = np.array(mu)
        Sigma = np.array(sigma)

        #x and y values of our samples
        x = [s[0] for s in self.samples]
        y = [s[1] for s in self.samples]
        #samples indexes
        sample_chosen_index = [i for i in range(self.N)]

        # Our 2-dimensional distribution will be
        #over variables X and Y
        #This is used for the mesh grid to be
        #able to plot contours
        N = 1000
        X = np.linspace(mu[0]-5*Sigma[0][0], mu[0]+5*Sigma[0][0],
    N)
        Y = np.linspace(mu[1]-5*Sigma[1][1], mu[1]+5*Sigma[1][1],
    N)
        X, Y = np.meshgrid(X, Y)

        # Pack X and Y into a single 3-dimensional array
        pos = np.empty(X.shape + (2,))
        pos[:, :, 0] = X
        pos[:, :, 1] = Y

        def multivariate_gaussian(pos, mu, Sigma):
            """
            Use to define the contour lines.
            Return the multivariate Gaussian distribution.

            pos is an array constructed by packing the
            meshed arrays of variables
            x_1, x_2, x_3, ..., x_k into
            its _last_ dimension.
            """
            n = mu.shape[0]
            Sigma_det = np.linalg.det(Sigma)
            Sigma_inv = np.linalg.inv(Sigma)
            N = np.sqrt((2*np.pi)**n * Sigma_det)
            # This einsum call calculates (x-mu)T.Sigma-1.(x-mu)
            #in a vectorized way across all the input variables.
            fac = np.einsum('...k,kl,...l->...', pos-mu,
   Sigma_inv, pos-mu)

            return np.exp(-fac / 2) / N


        #Gaussian distribution which is used for
        the contour plots
        Z = multivariate_gaussian(pos, mu, Sigma)

        #Create the plots
        fig, ax = plt.subplots(figsize=(20, 10))

        #Set the labels of the axis
        ax.set_xlabel('$x_1$')
        ax.set_ylabel('$x_2$')

        #Define the type of plot
        #Plot and scatter the points
        if plot_type == 1:
            ax.plot(x,y, color="black")
        else:
            ax.scatter(x, y, c=sample_chosen_index)

        #Add the contour plot to the figure
        ax.contour(X, Y, Z, cmap = 'jet')

        #Save the figure with the name given
        plt.savefig('{}.pdf'.format(file_name), bbox_inches = 'tight', pad_inches = 0)

    def plot_2D_rosenbrock(self, scale, file_name, plot_type, show_c):
        """
        This function plots the samples from a 2D Rosenbrock function.

        Inputs:
        -scale: how much we want to scale the Rosenbrock
        function by (float)
        -file_name: name of the file we wish to save (string)
        -plot_type: if 1 we plot the points otherwise we
        create a scatter plot(int)
        -show_c: if true we show the contour lines of
        the Rosenbrock function.
        Outputs:
        Plots from the samples given.
        """
        #Define the Rosenbrock pdf
        def Rosenbrock(x,y):
            return np.exp(-((100*(y - x**2)**2 + (1 - x)**2)/scale))

        #x and y values of our samples
        xs = [s[0] for s in self.samples]
        ys = [s[1] for s in self.samples]
        sample_chosen_index = [i for i in range(self.N)]

        #used to define the meshgrid such
        #that we can plot the contour lines
        x = np.linspace(-10,10,1000)
        y = np.linspace(-10,50,1000)
        X, Y = np.meshgrid(x, y)
        Z = Rosenbrock(X, Y)

        #Create the plots
        fig, ax = plt.subplots(figsize=(20, 10))

        #Set the labels of the axis
        ax.set_xlabel('$x_1$')
        ax.set_ylabel('$x_2$')

        #Define what type of plot you want
        #Plot and scatter the points
        if plot_type == 1:
            ax.plot(xs, ys, color="black")
        else:
            ax.scatter(xs, ys, c=sample_chosen_index)

        #This is used to define the levels
        #we want for the contour lines
        levels =  [0.01,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
        probability_contained = np.around(np.repeat(1, len(levels))-levels, decimals=1)

        #Define if want to use see the countor lines.
        if show_c == True:
            cntr = ax.contour(X, Y, Z, levels = levels, cmap="jet")
        else:
            cntr = ax.contour(X, Y, Z, levels = [0.01], cmap="jet")

        #Set the limits of the axis
        plt.xlim([-8,8])
        plt.ylim([-10,35])
        #Save the figure with the name given
        plt.savefig('{}.pdf'.format(file_name), bbox_inches = 'tight', pad_inches = 0)

    def plot_2D_rosenbrock_no_points(self, scale, file_name):
        """
        This function plots a 2D Rosenbrock function.

        Inputs:
        -scale: hwo much we want to scale the Rosenbrock f
        unction by (float)
        -file_name: name of the file we wish to save (string)
        Outputs:
        Plots from the samples given.
        """
        #Define the Rosenbrock pdf
        def Rosenbrock(x,y):
            return np.exp(-((100*(y - x**2)**2 + (1 - x)**2)/scale))

        #used to define the meshgrid such that we can
        #plot the contour lines
        x = np.linspace(-10,10,1000)
        y = np.linspace(-10,50,1000)
        X, Y = np.meshgrid(x, y)
        Z = Rosenbrock(X, Y)

        #Create the plots
        fig, ax = plt.subplots(figsize=(20, 10))

        #Set the labels of the axis
        ax.set_xlabel('$x_1$')
        ax.set_ylabel('$x_2$')

        #This is used to define the levels
        #we want for the contour lines
        levels =  [0.01,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
        probability_contained = np.around(np.repeat(1, len(levels))-levels, decimals=2)
        #Define if want to use see the countor lines.
        cntr = ax.contour(X, Y, Z, levels = levels, cmap = 'jet')
        #Defining the legend for the plot
        h,l = cntr.legend_elements("Z")
        ax.legend(h, probability_contained, title="Probability Contained",loc='upper left')

        #Set the limits of the axis
        plt.xlim([-8,8])
        plt.ylim([-10,35])

        #Save the figure with the name given
        plt.savefig('{}.pdf'.format(file_name), bbox_inches = 'tight', pad_inches = 0)

    def plot_histogram_gaussian(self, mean, sd, file_name):
        """
        This function plots the histograms for a one dimensional Gaussian distribution.

        Inputs:
        -mean: mean of distribution we are
        sampling from (float)
        -sd: standard deviation of distribution
        we are sampling from (float)
        -file_name: name of the file we wish to save (string)
        Outputs:
        Plots from the samples given.
        """
        #Which values of x to generate samples for
        #the analytical distribution
        if(max(self.samples)+1 > mean+3*sd):
            xs = np.linspace(min(self.samples)-1, max(self.samples)+1, 500)
        else:
            xs = np.linspace(min(self.samples)-1, mean+3*sd, 500)

        #Calculate the analytical distribution need
        #to use the Normal_Distribution Class.
        posterior_analytical = [Normal_Distribution(mean, sd).pdf(x) for x in xs]

        #Create the plots
        fig, ax = plt.subplots(figsize=(20, 10))
        #Plot the histogram
        plt.hist(self.samples,density=True,bins=50, label = "Estimated target distribution", alpha = 0.5, edgecolor='#169acf')
        #Plot the analytical distribution
        plt.plot(xs, posterior_analytical, label = "Analytical target distribution")

        #Set the labels of the axis
        ax.set_ylabel('Probability density')
        #Adding the legend
        ax.legend()

        #Save the figure with the name given
        plt.savefig('{}.pdf'.format(file_name), bbox_inches = 'tight', pad_inches = 0)


    def plot_histogram_gaussian_multiple(self, mean, sd, sample1, sample2, sample3, file_name):
        """
        This function plots the histograms for 4 one dimensional
        Gaussian distribution. This is used to show the evolution
        for different samples on the same target distribution.

        Inputs:
        -mean: mean of distribution we are
        sampling from (float)
        -sd: standard deviation of distribution we are
        sampling from (float)
        -sample_1, 2 and 3: list of a samples you
        want to compare to (list int)
        -file_name: name of the file we wish to save (string)

        Outputs:
        Plots from the samples given.
        """
        #How generate samples for the analytical distribution
        if(max(self.samples)+1 > mean+3*sd):
            xs = np.linspace(min(self.samples)-1, max(self.samples)+1, 500)
        else:
            xs = np.linspace(min(self.samples)-1, mean+3*sd, 500)

        #Calculate the analytical distribution need to
        #use the Normal_Distribution Class.
        posterior_analytical = [Normal_Distribution(mean, sd).pdf(x) for x in xs]

        #Create the plots (2 by 2)
        fig, ax = plt.subplots(2,2, figsize=(20, 10))

        #Populate each subplot with the given samples.
        #first with the samples we used to define the function.
        ax[0, 0].hist(self.samples,density=True,bins=50, label = "Estimated target distribution", alpha = 0.5, edgecolor='#169acf')
        ax[0, 0].plot(xs, posterior_analytical, label = "Analytical target distribution")
        ax[0, 0].set_ylabel('Probability density')
        ax[0, 0].set_title(r"$N = {}$".format(self.N))

        #Populate each subplot with the given by samples1.
        ax[0, 1].hist(sample1,density=True,bins=50, label = "Estimated target distribution", alpha = 0.5, edgecolor='#169acf')
        ax[0, 1].plot(xs, posterior_analytical, label = "Analytical target distribution")
        ax[0, 1].set_title(r"$N = {}$".format(len(sample1)))

        #Populate each subplot with the given by samples2.
        ax[1, 0].hist(sample2,density=True,bins=50, label = "Estimated target distribution", alpha = 0.5, edgecolor='#169acf')
        ax[1, 0].plot(xs, posterior_analytical, label = "Analytical target distribution")
        ax[1, 0].set_ylabel('Probability density')
        ax[1, 0].set_title(r"$N = {}$".format(len(sample2)))

        #Populate each subplot with the given by samples3
        ax[1, 1].hist(sample3,density=True,bins=50, label = "Estimated target distribution", alpha = 0.5, edgecolor='#169acf')
        ax[1, 1].plot(xs, posterior_analytical, label = "Analytical target distribution")
        ax[1, 1].set_title(r"$N = {}$".format(len(sample3)))

        #Save the figure with the name given
        plt.savefig('{}.pdf'.format(file_name), bbox_inches = 'tight', pad_inches = 0)

    def plot_histogram_log(self, mean, sd, file_name):
        """
        This function plots the histograms for a
        log normal distribution.

        Inputs:
        -mean: mean of distribution we
        are sampling from (float)
        -sd: standard deviation of distribution we
        are sampling from (float)
        -file_name: name of the file we wish to save (string)
        Outputs:
        Plots from the samples given.
        """
        #How generate samples for the analytical distribution
        xs = np.linspace(0, max(self.samples)+1, 500)

        #Calculate the analytical distribution need to
        #use the Log_Normal_Distribution Class.
        posterior_analytical = [Log_Normal_Distribution(mean, sd).pdf(x) for x in xs]

        #Create the plot
        fig, ax = plt.subplots(figsize=(20, 10))
        #Plot the histogram
        plt.hist(self.samples,density=True,bins=100, label = "Estimated target distribution", alpha = 0.5, edgecolor='#169acf')
        #Plot the analytical distribution
        plt.plot(xs, posterior_analytical, label = "Analytical target distribution")

        #Set the labels of the axis
        ax.set_ylabel('Probability density')
        #Adding the legend
        ax.legend()

        #Save the figure with the name given
        plt.savefig('{}.pdf'.format(file_name), bbox_inches = '
   tight', pad_inches = 0)
```

Functions used to tune the parameters of the reversible and non reversible samplers.

```
import numpy as np
import pandas as pd
from scipy import stats
import random

def tuning_parameter_reversible(beta_s, target_dist, proposal, N,
    dim):
    """
    Function to find the average acceptance ratio for
    different values of beta. The output of this function is then
    used in the plot_finding_beta function. This is used in the
    analysis when tuning beta for reversible samplers.

    Input:
    -beta_s: list of beta values you are testing (list of float)
        Example:
        beta_s = [x/10 for x in range(60)]
    -target_dist: one of the Classes of target distributions
    we introduced.
        Example:
        target_dist = Normal_Distribution(10,2)
    -proposal: proposal density to use one of the
    ones from the Proposal Class.
    -N: number of iteration each chain should run for (int)
    -dim: dimension of the problem (int)

    Output:
    -list_a_p: a list of average acceptance
    probability for each beta value
    """
    #Empty list to keep all average acceptance ratio
    list_a_p = []

    #Iterate over the beta values
    for beta in beta_s:
        #Define the proposal for the given beta
        proposal = Proposal(beta = beta, pick = proposal)
        #Define the sampler for the given proposal
        sampler = Reversible_Sampler(N = N, dimension = dim,
                                     target_distribution=
   target_dist, proposal=proposal)
        #Get the acceptance probabilities
        _, a = sampler.Reversible_Metropolis_Hastings_Algo()
        #Calculate the average acceptance ratio
        average_acceptance_probabity = np.mean(a)
        #Add this value to the end of the list.
        list_a_p.append(average_acceptance_probabity)

    #Return the all average acceptance probabilities.
    return list_a_p

def plot_finding_beta(beta_s, list_average_acceptance_probabity, file_name):
        """
        Function used to plot the average acceptance
        probability against beta. This is used in the analysis
        when tuning beta for reversible samplers.
        Inputs:
        -beta_s: list of beta values you
        are testing (list of float)
        -list_average_acceptance_probabity: list of average acceptance
        probability (list of float)
        -file_name: name of the file we wish to save (string)
        """

        #Create the plot
        fig, ax = plt.subplots(figsize=(20, 10))
        #Plot the values
        plt.plot(beta_s,list_average_acceptance_probabity)
        #Plot the horizontal for desired average acceptance probabilty
        ax.hlines(.234, beta_s[0], beta_s[-1], "r",label = r"$\alpha$ = 0.234")
        #Set the labels of the axis
        ax.set_ylabel(r'average acceptance ratio $\alpha$')
        ax.set_xlabel(r'$\sigma$')
        #Adding the legend
        ax.legend()

        #Save the figure with the name given
        plt.savefig('{}.pdf'.format(file_name), bbox_inches = 'tight', pad_inches = 0)

def tuning_parameter_non_reversible(b_s, period, target_dist, N, dim):
    """
    Function to find the average acceptance ratio
    for different values of b. The output of this function
    is then used in the plot_finding_b function.
    This is used in the analysis when tuning beta for
    non reversible samplers.

    Input:
    -b_s: list of b values you are testing (list of float)
        Example:
        b_s = [x/10 for x in range(60)]
    -period: period to update the auxiliary variable
    in higher dimensions.
    -target_dist: one of the Classes of target
    distributions we introduced.
        Example:
        target_dist = Normal_Distribution(10,2)
    -N: number of iteration each chain should run for (int)
    -dim: dimension of the problem (int)
    Output:
    -list_a_p: a list of average acceptance ratio for
    each b value
    """
    #Empty list to keep all average acceptance ratio
    list_a_p = []

    #One dimensional case
    if dim == 1:
        #Iterate over the beta values
        for b in b_s:
            #Define the sampler for the given proposal
            sampler = Non_Reversible_Sampler(alpha= 1, beta = b,
   N = N, dimension = dim, target_distribution=target_dist)
            #Get the acceptance probabilities
            _, a = sampler.One_Directional_I_Jump_Sampler()
            #Calculate the average acceptance ratio
            average_acceptance_probabity = np.mean(a)
            #Add this value to the end of the list.
            list_a_p.append(average_acceptance_probabity)

    #High dimensional target distirbution
    else:
        #Iterate over the beta values
        for b in b_s:
            #Define the sampler for the given proposal
            sampler =  Non_Reversible_Sampler(alpha= 1, beta = b,
    N = N, dimension = dim, target_distribution=target_dist)
            #Get the acceptance probabilities
            _, a = sampler.N_Directional_I_Jump_Sampler(period)
            #Calculate the average acceptance ratio
            average_acceptance_probabity = np.mean(a)
            #Add this value to the end of the list.
            list_a_p.append(average_acceptance_probabity)

    #Return the all average acceptance probabilities.
    return list_a_p


def plot_finding_b(b_s, list_average_acceptance_probabity,
   file_name):
        """
        Function used to plot the average acceptance ratio
        against b. This is used in the analysis when tuning b
        for non reversible samplers.
        Inputs:
        -b_s = list of b values you are testing (list of float)
        -list_average_acceptance_probabity = list of average
        acceptance ratio (list of float)
        -file_name: name of the file we wish to save (string)
        """
        #Create the plot
        fig, ax = plt.subplots(figsize=(20, 10))
        #Plot the values
        plt.plot(b_s,list_average_acceptance_probabity)
```

```python
        #Plot the horizontal for desired average acceptance ratio
        ax.hlines(.234, b_s[0], b_s[-1], "r",label = r"$\alpha$ = 0.234")
        #Set the labels of the axis
        ax.set_ylabel(r'average acceptance ratio $\alpha$')
        ax.set_xlabel(r'$b$')

        #Adding the legend
        ax.legend()
        #Save the figure with the name given
        plt.savefig('{}.pdf'.format(file_name), bbox_inches = 'tight', pad_inches = 0)
```

Functions used to plot normalized autocorrelation function against lags.

```python
import numpy as np
import pandas as pd
from scipy import stats
import random

class Compare_Covariance():
    """
    Class used to plot the normalized autocorrelation function
    between samples for different samples.
    Makes use of the Variance_Analysis class.
    """

    def __init__(self, samples_1, samples_2):
        #Define the two set of samples we want to compare
        self.samples_1 = samples_1
        self.samples_2 = samples_2

    def compare_lags(self, max_lag, label_1, label_2, file_name):
        """
        This function plots normalized autocorrelation
        function against lag.
        Input:
        -max_lag: to what lag we want to calculate this for (int)
        -label_1: name of sampler which produced samples_1 (str)
        -label_2: name of sampler which produced samples_2 (str)
        -file_name: name of the file we wish to save (string)
        """
        #Create Variance_Analysis objects to be able to use
        #the functions within them
        variance_analysis_r = Variance_Analysis(self.samples_1)
        variance_analysis_non_r = Variance_Analysis(self.samples_2)

        #Calculate normalized autocorrelation for different
        #lags up to max_lag
        covariance_reversible = [variance_analysis_r.c_tau(i) for i in range(max_lag)]
        covariance_non_reversible = [variance_analysis_non_r.c_tau(i) for i in range(max_lag)]

        #Create a list for lags used
        lag_reversible = [i for i in range(max_lag)]
        lag_non_reversible = [i for i in range(max_lag)]

        #Create the plot
        fig, ax = plt.subplots(figsize=(20, 10))
        #Plot ACF against lag for first set of samples
        plt.plot(lag_reversible,covariance_reversible,label = label_1)
        #Plot ACF against lag for second set of samples
        plt.plot(lag_non_reversible,covariance_non_reversible, label = label_2)

        #Set the labels of the axis
        ax.set_ylabel(r'ACF ${\varrho}_{\varphi}$')
        ax.set_xlabel(r'Lag $\tau$')
        #Adding the legend
        ax.legend()

        #Save the figure with the name given
        plt.savefig('{}.pdf'.format(file_name), bbox_inches = 'tight', pad_inches = 0)

    def compare_lags_three(self, max_lag, label_1, label_2, label_3, sample_3, file_name):
        """
        This function plots normalized autocorrelation function
        against lag.For 3 different set of samples where the
        third needs to be defined in the function call.

        Input:
        -max_lag: to what lag we want to calculate this for (int)
        -label_1: name of sampler which produced samples_1 (str)
        -label_2: name of sampler which produced samples_2 (str)
        -label_3: name of sampler which produced samples_3 (str)
        -sample_3: third set of samples we want to compare
        -file_name: name of the file we wish to save (string)
        """
        #Create Variance_Analysis objects to be able to use the
        #functions within them
        variance_analysis_1 = Variance_Analysis(self.samples_1)
        variance_analysis_2 = Variance_Analysis(self.samples_2)
        variance_analysis_3 = Variance_Analysis(samples_3)

        #Calculate normalized autocorrelation for different
        #lags up to max_lag
        ACF_1 = [variance_analysis_1.c_tau(i) for i in range(max_lag)]
        ACF_2 = [variance_analysis_2.c_tau(i) for i in range(max_lag)]
        ACF_3 = [variance_analysis_3.c_tau(i) for i in range(max_lag)]

        #Create a list for lags used
        lag = [i for i in range(max_lag)]

        #Create the plot
        fig, ax = plt.subplots(figsize=(20, 10))

        #Plot lag agaisnt ACF for all set of samples
        plt.plot(lag, ACF_1,label = label_1)
        plt.plot(lag, ACF_2,label = label_2)
        plt.plot(lag, ACF_3,label = label_3)

        #Set the labels of the axis
        ax.set_ylabel(r'ACF ${\varrho}_{\varphi}$')
        ax.set_xlabel(r'Lag $\tau$')
        #Adding the legend
        ax.legend()
        #Save the figure with the name given
        plt.savefig('{}.pdf'.format(file_name), bbox_inches = 'tight', pad_inches = 0)
```

Appendix D

Imaging Algorithms

Python implementation of the Metropolis Hastings algorithm and the N-Directional I-Jump sampler with modifications for imaging problems.

```python
import numpy as np
import pandas as pd
from scipy import stats
import random

class Sampler_Imaging():
    """
    This class defines reversible and non reverisible samplers
    such that they can be used with images.
    """

    def __init__(self, beta, N, dimension, image, proposal):
        #This defines the step size for both reversible and non
        #reverisble samplers
        self.beta = beta
        #N is the number of iterations the algorithm runs (int)
        self.N = N
        #dimension is the dimension of the state space (int)
        self.dimension = dimension
        #Proposal used for the reversible sampler do not need to
        define one for non reverisble samplers
        self.proposal = proposal
        #The target distribution is defined by the image and uses
        #the Class_Target, to define the pdf of this distribution
        self.target_distribution = Image_Target(image)
        #This is the image in raw form, under no transformation.
        self.image = image

    def Reversible_Metropolis_Hastings_Algo(self):
        """
        Implementation of Metropolis Hastings Algorithm.
        For imaging.

        Output:
        -samples: List of samples which proposal images
        -list_acceptance_probaility: List of acceptance
        probabilities used for tuning the algorithm.
        """

        #We initalzie the first sample at the image flattened.
        u_0 = np.around(self.image.flatten())
        u_t = u_0

        #list of samples generated by the algorithm
        samples = [u_0]
        #list of acceptance ratio at each iteration
        #of the algorithm
        list_acceptance_probaility = []

        #Proposal of the problem
        proposal = self.proposal

        #Begin iterations
        for t in range(self.N):

            #Sample a proposal x_prime from proposal at x_t
            u_prime = np.around(proposal.get_random_variable(u_t))

            #Evaluation of the numerator of acceptance ratio
            numerator = self.target_distribution.pdf(u_prime)
            #Evaluation of the denomiantor of acceptance ratio
            denominator = self.target_distribution.pdf(u_t)

            #Calculate acceptance proability
            if numerator == 0:
                acceptance_probaility = 0
            elif denominator == 0:
                acceptance_probaility = 1
            else:
                acceptance_probaility = float(min([1,numerator/denominator]))

            #Add acceptance ratio to
            #list_acceptance_probaility
            list_acceptance_probaility.append(acceptance_probaility)
            rv = np.random.uniform(0,1,1)
            #Accept or reject step
            if rv <= acceptance_probaility:
                #Accept: we update x_t to x_prime
                u_t = u_prime
                samples.append(u_t)
            else:
                #Reject: we remain at x_t
                u_t = u_t
                samples.append(u_t)

        #return the samples and all the acceptance probabilities
        return samples, list_acceptance_probaility

    def Non_Reversible_Metropolis_Hastings_High_Dimensions(self, period):
        """
        Implementation of the N-Directional I-Jump sampler
        for imaging.

        Input:
        -period: periodicity to change the direction of
        auxiliary variable y_p. (int)
        Output:
        -samples: List of samples which are integers.
        -list_acceptance_probaility: List of acceptance
        probabilities used for tuning the algorithm.
        """

        #We initialize the first sample at the image flattened.
        x_0 = np.around(self.image.flatten())
        x_t = x_0

        #Define the possible directions we can move in
        direction_vectors = np.eye(self.dimension)
        #Pick a random direction for auxiliary variable y_p
        y_p = direction_vectors[np.random.randint(self.dimension)]

        #list of samples generated by the algorithm
        samples = [x_t]
        #list of acceptance probabilities
        list_acceptance_probaility = []

        #We define variables such that they are the
        #posterior distirbution of the problem
        posterior_distribution = self.target_distribution

        #We inizalize the z indicator random varibale
        rv_z = np.random.uniform(0,1,1)
        z = 0
        if rv_z < 0.5:
            z = 1
        else:
            z = -1

        #Define sgn function which calculates
        #the dot product of 2 vectors
        #And returns 1 greater than 0 or -1 otherwise
        def sgn(eta,y_p):
            x = np.dot(eta,y_p)
            if x >= 0:
                return 1
            else:
                return -1

        #Define a multivariate normal distribution
        #used to propose samples
        multivariate_normal = stats.multivariate_normal(np.zeros(self.dimension),self.beta**2*np.eye(self.dimension))

        #Define a half space multivariate Gaussian distribution,
        f for one direction
        def f_pdf(z, y, y_p):
            x = np.dot(y_p,z-y)
            if x >= 0:
                return (2/(2*np.pi*self.beta)**(self.dimension/2)*np.exp((-1/2)*np.linalg.norm(z-y)**2))
            else:
                return 0

        #Define a half space multivariate Gaussian distribution,
        #g for other direction
        def g_pdf(z, y, y_p):
            x = np.dot(y_p,z-y)
            if x >= 0:
                return 0
            else:
                return (2/(2*np.pi*self.beta)**(self.dimension/2)*np.exp((-1/2)*np.linalg.norm(z-y)**2))

        #Begin iterations
        for t in range(self.N):

            # Change direction y_p periodically
            if t%period == 0:
                y_p = direction_vectors[np.random.randint(self.dimension)]

            #Indicator variable telling us in what
            #direction to move in.
            if z > 0:

                #Create a normal distribution random variable
                eta = multivariate_normal.rvs()
                #Sample a proposal x_prime from proposal
                #at x_t with f.
                x_prime = x_t + sgn(eta,y_p)*eta

                #numerator of acceptance ratio
                numerator = posterior_distribution.pdf(x_prime)*g_pdf(x_t, x_prime, y_p)

                #denominator of acceptance ratio
                denominator = posterior_distribution.pdf(x_t)*f_pdf(x_prime, x_t, y_p)

                #Calculate acceptance ratio
                if numerator == 0:
                    acceptance_probaility = 0
                elif denominator == 0:
                    acceptance_probaility = 1
                else:
                    acceptance_probaility = float(min([1, numerator/denominator]))

            else:
                #Create a normal distribution random variable
                eta = multivariate_normal.rvs()

                #Sample a proposal x_prime from proposal
                #at x_t with g.
                x_prime = x_t - sgn(eta,y_p)*eta

                #numerator of acceptance ratio
                numerator = posterior_distribution.pdf(x_prime)*f_pdf(x_t,x_prime, y_p)

                # denominator of acceptance ratio
                denominator = posterior_distribution.pdf(x_t)*g_pdf(x_prime, x_t, y_p)

                #Calculate acceptance ratio
                if numerator == 0:
                    acceptance_probaility = 0
                elif denominator == 0:
                    acceptance_probaility = 1
                else:
                    acceptance_probaility = float(min([1,numerator/denominator]))

            #Add acceptance ratio to
            #list_acceptance_probaility
            list_acceptance_probaility.append(acceptance_probaility)

            #Sample a random variable [0,1]
            rv = np.random.uniform(0,1,1)

            #Accept or reject step
            if rv <= acceptance_probaility:
                #Accept
                x_t = x_prime
                samples.append(x_t)
                #leave z unchanged
                z = z
            else:
                #Accept update x_t to x_prime
                x_t = x_t
                samples.append(x_t)
                #update the indicator z
                z = -1*z
        #We return the samples and all the
        #acceptance probabilities
        return samples, list_acceptance_probaility
```

Python target distributions the Metropolis Hastings algorithm and the N-Directional I-Jump sampler for imaging problems. In particular for the posterior distribution defined in equation 7.8.

```
import numpy as np
import pandas as pd
from scipy import stats
import random

class Image_Target():
    """
    This is the target distribution for imaging problems.
    We need to define the prior and likelihood to be
    able to use this as the posterior distribution.
    """
    def __init__(self, image):
        """
        To initialize we need to pass an image through this,
        that is an nxn np.array. With values between 0 to 264.
        """
        self.image = image
        #n is the dimension of our problem
        self.n = image.shape[0]
        #we flatten the image to be the mean of the likelihood
        self.flatten_image = image.flatten()

    def prior_J_TV(self, x):
        """
        We are defining the Total Variation (TV) prior in
        this function for a given image x.
        input:
        -x:is a flattened n by n image (n by n np.array)
        output:
        -prior: the value of the prior for the image x (int)
        """
        #This is the dimension of the image
        n = self.n
        #Reshape the image
        X = x.reshape((self.n,self.n))

        #Here we define the circulant matrix D
        first = np.array([-1]+[1]+[0 for i in range(n-2)])
        last = np.array([0 for i in range(n-2)]+[-1]+[1])
        D = first
        for i in range(n-2):
            D_i = np.array([0 for i in range(i)]+[-1,0,1]+[0 for
   i in range(n-3-i)])
            D = np.vstack((D,D_i))

        D = np.vstack((D,last))

        #get the number of rows and collumns of matrix D
        n, m = D.shape
        #We calulcate the expression for the TV prior
        #Multiply image X by D
        X_D_t = X@D.T
        #Multiply D by X
        D_X = D@X

        #We need to iterate over these matrices. Square and
        #add these values together, by definition of TV prior.
        prior = 0
        #range over the columns
        for i in range(m):
            #range over the rows
            for j in range(n):
                #Sum each individual entry of X_D_t and
                #D_X together
                prior = prior + np.sqrt(((X_D_t)[i,j])**2 + ((D_X)[i,j])**2)

        #Return the prior value
        return prior

    def pdf(self, x):
        """
        Posterior distribution of image. Evaluates pdf of an
        image x.
        """
        #The likelihood is defined in
        #the later part of the expression
        return self.prior_J_TV(x)*(np.exp(-1/2*(np.linalg.norm(self.flatten_image-x))**2))
```

Functions to calculate Conditional Mean, Maximum A-Posteriori Estimate and Mean Square error for images.

```

def MAP(images, target_distribution):
    """
    Calculates the Maximum A-Posteriori Estimate of a target
    of interest given samples in this particular case for images.
    Input:
    -images: images to be evaluated produced
    by an MCMC method (list)
    -target_distribution: where these
    images are evaluated at.
    Output:
    -best_image: image with highest
    target density (image)
    """
    #set max value to 0
    max_value = 0
    #let the best image be the first sample
    best_image = images[0]

    #iterate over the images
    for image in images:
        #if a new sample has higher target probability
        if target_distribution.pdf(image) > max_value:
            #update the max value
            max_value = target_distribution.pdf(image)
            #update best_image
            best_image = image

    return best_image

def CM(images):
    """
    Estimates the Condtional Mean of a set of sample.
    Input:
    -images: images to be evaluated produced
    by an MCMC method (list)
    Output:
    -CM of the images.
    """
    #get the first image
    image_1 = images[0]
    #iterate over the other images
    for image in images[1:]:
        #add current image with the next
        image_1 = image_1 + image
    #return the mean of all the images together
    return (1/len(images))*(image_1)

def MSE(actual, pred):
    """
    Calculates the MSE between the actual vector and the
    predicted vector. In imaging this is simply
    the difference between the original and the estimate.
    """
    actual, pred = np.array(actual), np.array(pred)
    return np.square(np.subtract(actual,pred)).mean()
```

Helper functions for image processing provided by Dr Aretha Teckentrup.

```
import PIL.Image
import matplotlib.pyplot as plt
import numpy as np
import scipy.ndimage
from scipy import stats

__all__ = \
    [
        "empty_true_color",
        "gaussian_filter",
        "get_bit_plane",
        "imhist",
        "imread",
        "imscale",
        "imshow",
        "imshow_reduced_color_depth",
        "imwrite",
        "linear_filter",
        "median_filter",
        "rank_filter",
        "zero_bit_plane"
    ]


def _img_a_cast(img_a, dtype, true_color=False):
    img_a = np.maximum(img_a, 0)
    img_a = np.minimum(img_a, 255)
    img_a = np.round(img_a, 0)
    img_a = np.array(img_a + 1.0e-6, dtype=dtype)
    if len(img_a.shape) == 2:
        if true_color:
            img_a_gs = np.zeros((img_a.shape[0], img_a.shape[1],
   3),
                                dtype=dtype)
            for k in range(3):
                img_a_gs[:, :, k] = img_a
            return img_a_gs
        else:
            return img_a
    else:
        if len(img_a.shape) != 3 or img_a.shape[2] != 3:
            raise RuntimeError("Unexpected image type")
        return img_a

def imread(filename):
    img = PIL.Image.open(filename, "r")
    return np.array(img, dtype=np.int64)

def imscale(img_a, factor, interpolation="nearest"):
    """
    Scale an image.
    """
    M, N = img_a.shape[:2]
    M_scaled = int(max(round(M * factor, 0), 1) + 1.0e-6)
    N_scaled = int(max(round(N * factor, 0), 1) + 1.0e-6)

    img_a = _img_a_cast(img_a, dtype=np.uint8)
    img = PIL.Image.fromarray(img_a)
    img = img.resize((M_scaled, N_scaled),
                     resample={"nearest": PIL.Image.NEAREST,
                               "bilinear": PIL.Image.BILINEAR,
                               "bicubic": PIL.Image.BICUBIC,
                               "lanczos": PIL.Image.LANCZOS}[
   interpolation])

    return np.array(img, dtype=np.int64)

def imshow(img_a, new_figure=True):
    img_a = _img_a_cast(img_a, dtype=np.uint8, true_color=True)

    if new_figure:
        plt.figure()
    plt.imshow(img_a)
    plt.xticks([])
    plt.yticks([])

def imsave(img_a, file_name, new_figure=True):
    """
    Save an image.
    """
    img_a = _img_a_cast(img_a, dtype=np.uint8, true_color=True)

    if new_figure:
        plt.figure()
    plt.imshow(img_a)
    plt.xticks([])
    plt.yticks([])
    plt.savefig('{}.pdf'.format(file_name), bbox_inches = 'tight'
   , pad_inches = 0)

```

```python

def linear_filter(img_a, W, **kwargs):
    """
    Adds a linear filter W to the img_a.
    """
    img_a = _img_a_cast(img_a, dtype=np.int64)
    W = np.fliplr(np.flipud(W))

    if len(img_a.shape) == 2:
        img_filtered_a = scipy.ndimage.convolve(img_a, W, **kwargs)
    else:
        assert len(img_a.shape) == 3
        assert img_a.shape[2] == 3
        img_filtered_a = np.zeros_like(img_a)
        for k in range(3):
            img_filtered_a[:, :, k] = scipy.ndimage.convolve(
                img_a[:, :, k], W, **kwargs)

    return _img_a_cast(img_filtered_a, dtype=np.int64)

def gaussian_filter(img_a, sigma, **kwargs):
    """
    Adds a gaussian filter variance sigma to the img_a with variance.
    """
    img_a = _img_a_cast(img_a, dtype=np.int64)

    if len(img_a.shape) == 2:
        img_filtered_a = scipy.ndimage.gaussian_filter(
            img_a, sigma, **kwargs)
    else:
        assert len(img_a.shape) == 3
        assert img_a.shape[2] == 3
        img_filtered_a = np.zeros_like(img_a)
        for k in range(3):
            img_filtered_a[:, :, k] = scipy.ndimage.gaussian_filter(
                img_a[:, :, k], sigma, **kwargs)

    return _img_a_cast(img_filtered_a, dtype=np.int64)
```