Faculty of Engineering

Adaptive Markov Chain Monte Carlo for Bayesian Inference

Thesis submitted in fulfillment of the requirements for the degree of Master of Engineering: Applied Computer Science

January 15, 2018

Philip KHOURY

Promotor: Prof. Dr. Bernard Manderick (Vrije Universiteit Brussel)

VUBPRESS is an imprint of ASP nv (Academic and Scientific Publishers nv)
Ravensteingalerij 28
B-1000 Brussels
Tel. +32 (0)2 289 26 50
Fax +32 (0)2 289 26 59
E-mail: info@vubpress.be
www.vubpress.be

Abstract

Evaluating analytically intractable integrals is a limitation when implementing Bayesian inference. This thesis introduces adaptive Markov Chain Monte Carlo (MCMC) and stochastic optimization methods as techniques for evaluating said integrals. Code for these algorithms is written in Python, and has been shown to work as expected when tested against known benchmarks. Further improvements have been made to the algorithms. The thesis incorporated adaptation into the stochastic optimization methods, transforming them into samplers. Namely, Gaussian Adaptation (GaA) and Covariance Matrix Adaptation Evolution strategy (CMA-ES) optimizers have been transformed into the Metropolis-GaA, and $(1 + 1)$CMA, respectively. The latter being the contribution of this thesis. Performance is quantified using existing convergence and performance measuring tools. Results show adaptive MCMCs with better convergence, mixing, and acceptance ratios.

Acknowledgments

I want to express my sincere gratitude to my promoter Prof. Dr. Bernard Manderick for his continuous support, patience, motivation, devotion, and immense knowledge. He helped steer me in the right direction whenever I needed it. He truly made this experience an enjoyable learning process. I could not have imagined having a better promoter and mentor for my master thesis.

I want to also thank my fellow lab-mates, Nixon Kipkorir Ronoh and Edna Chelangat Milgo for their valuable contributions, stimulating discussions and critical feedback.

Contents

List of Figures

List of Tables

List of Algorithms

1 | Introduction

Machine Learning is the development of computer programs that can automatically learn and improve from data without being explicitly programmed. In this era of powerful computational abilities and data availability, Bayesian methods have undergone a great renaissance in areas such as machine learning, astronomy and genetics.

This chapter provides a broad view of the research herein. First, it outlines Bayesian inference and its challenges. It then proceeds to outline the general field of MCMC and its present-day algorithms. It then presents respectively in the form of a problem statement, research question, and contribution. An outline of the chapters follows at chapter's end.

1.1 Problem Statement

Bayesian inference assigns probability distributions to unobserved parameters and latent variables, then updates these distributions as evidence become available. A limiting difficulty is evaluating its underlying integrals, as most integrals are intractable and cannot be solved in closed-form.

Markov Chain Monte Carlo (MCMC) algorithms have become an indispensable tool for Bayesian inference. MCMC algorithms can generate samples from any probability distribution, known or unknown, and hence can evaluate most integrals, and especially useful when evaluating integrals of *complex* distributions.

The goal is to generate a Markov chain with transition kernel on a state space that has the target distribution as its unique invariant distribution. Once the chain is said to

have converged, samples generated from the Markov chain are according to the target distribution.

> **Remark 1.1: Proposal and Target Distributions**
>
> A *proposal distribution* is used to generate samples which are either accepted or rejected based on an *acceptance rule*, according to a probability distribution, coined *target distribution*.

An example is given to illustrate Bayesian inference in action, see Example 2.2.

> **Example 1.1: Coin Bias**
>
> To determine the actual bias of a coin, an agent records the outcomes of many coin flips. After each number of experiments, the agent updates their initial belief. In the end of their experiments, the agent is able to infer a probability distribution for their belief about the bias.

Metropolis-Hastings algorithm (MH) is often synonymous with MCMC methods. It generates an ergodic Markov chain based on an acceptance rule that ensures that, upon convergence, samples are generated according to the target distribution. Performance of MH depends critically on a number of factors:

- *acceptance ratio*: percentage of samples which are accepted, and;
- *mixing*: how well the state space is explored.

An active area of research is underway and focused on creating more flexible proposal distributions. A solution proposed as *adaptive MCMC* (Haario et al., 2001a; Roberts and Rosenthal, 2009a) has been successful in creating MCMC algorithms which self-adapt with each algorithm run.

The acceptance criteria of the MH algorithm ensures that samples are accepted only from high probability regions of the target distribution. These accepted samples are then used in adaptive MCMC algorithms to continuously tune proposal distribution parameters. For more information, the reader is referred to (Haario et al., 1999; Muller and Sbalzarini, 2010; Andrieu and Thoms, 2008a) for early works in adaptive MCMC.

Although current adaptive MCMC algorithms have solved the problem of non-adaptive proposal distribution in traditional MH methods, most research undertakings now focus on making adaptive MCMC methods more computationally efficient.

1.2 Research Question

This thesis was guided by the following research question:

What are the most promising present-day adaptive MCMC algorithms used for Bayesian inference problems? And how can improvements be made to their accuracy and computational efficiency?

Adaptive MCMC algorithms are contemporary algorithms for Bayesian inference. Stochastic techniques were used to improve convergence speed and accuracy. Results have been benchmarked using Geweke and Gelman Rubin measures of convergence and have shown improvements in performance of MCMC methods used.

1.3 Contribution

Guided by the research question, the following contributions have been made:

- Herein, a new approach is presented. Adaptive MCMC methods have been incorporated into a Stochastic optimization method, transforming it into a sampler. The result is the $(1+1)$CMA-ES algorithm;
- An overview of Bayesian inference has been given, along with a supporting example to its applicability;
- The underlying theorems of MCMC algorithms were explained, along with a description of the Metropolis-Hastings algorithm. This gives the reader a clear background of MCMC;
- The next step was to describe challenges faced by traditional MCMC techniques;
- This led to adaptive MCMC algorithms. An explanation ensued for tuning proposal distributions in each sampling iteration, rendering adaptive MCMC methods as self-adapting, or self-trained. Specifically, the Adaptive Metropolis (AM) algorithm was explained;
- State of the art in stochastic optimization methods was reviewed. Two important methods in this domain are Gaussian Adaptation (GaA) and Covariance Matrix Adaptation Evolutionary strategies (CMAES). These are the most preferred methods for solving black-box optimization problems.
- A number of experiments have been conducted to compare proposed samplers on test suits of Haario et al. (2001a), results were compared with other adaptive MCMC algorithms;

1.4 Outline of the Thesis

Chapter two gives an overview of Bayesian Inference and the underlying methods necessary for its computations. It shows how it differs from the frequentist approach to statistical inference. It then explains its difficulties and methods for overcoming them. Chapter 3 describes present-day MCMC algorithms and presents its underpinning algorithm, the Metropolis-Hastings (MH) algorithm. Further, it explains key concepts in MCMC sampling and explains accuracy in measurements through covariance diagnostic tools. Chapter 4, the pioneering work in adaptive MCMC, the Adaptive Metropolis Algorithm, is presented as an improvement to the MH algorithm or its flavors.

Chapter 5 explains stochastic optimization methods. And at the end of the Chapter 5, the new algorithm (1+1)CMA is presented. Chapter 6 illustrates the experimental set-ups and results. Finally, the last chapter concludes the thesis by highlighting the main takeaways.

2 | Bayesian Inference In Machine Learning

Bayesian inference is a widely used approach in machine learning. It is a method for applying Bayes rule to *update* a probability distribution as observed data (hereinafter: *evidence*) become available. It incorporates evidence in order to update the degree of prior belief about model *parameters*, and to quantify their uncertainties.

This chapter will demonstrate Bayesian inference through a toy example, *finding the bias of a coin*. It will set up the model, run experiments and illustrate results.

2.1 Bayes Rule

Bayes rule is named after Thomas Bayes but its current form can be attributed to Pierre Simon Laplace (McGrayne, 2011). It is a simple yet very powerful computational tool whose role was bolstered with the advent of computers and recent advances in MCMC methods. It is used as the *update* rule in Bayesian inference.

The mathematical formulation for Bayes rule is depicted in the following:

$$\pi(\boldsymbol{\theta}|\mathbf{x}_{1:n}) = \frac{\pi(\mathbf{x}_{1:n}|\boldsymbol{\theta})\pi(\boldsymbol{\theta})}{\int_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} \pi(\mathbf{x}_{1:n}|\boldsymbol{\theta})\pi(\boldsymbol{\theta})d\boldsymbol{\theta}} \tag{2.1}$$

where:

$\mathbf{x}_{1:n} \triangleq (\mathbf{x}_1, ..., \mathbf{x}_n)$	*evidence*, where $\mathbf{x} \in \mathcal{R}^d$, and $x_1, x_2, ..., x_d$ are scalar components of the d-dimensional vector $\mathbf{x}$
$\pi(\mathbf{x}_{1:n} \mid \boldsymbol{\theta})$	*likelihood* of data $\mathbf{x}_{1:n}$ given parameter vector $\boldsymbol{\theta} \in \mathcal{R}^k$, where $\theta_1, \theta_2, ..., \theta_k$ are scalar components of the k-dimensional vector $\boldsymbol{\theta}$
$\pi(\boldsymbol{\theta})$	*prior*; what is known about $\boldsymbol{\theta}$ before considering $\mathbf{x}$
$\pi(\boldsymbol{\theta} \mid \mathbf{x}_{1:n})$	*posterior*; the probability of model parameters ($\boldsymbol{\theta}$) given data $\mathbf{x}$
$\pi(\mathbf{x}_{1:n})$	a normalizing constant, and independent of $\boldsymbol{\theta}$

Table 2.1: Bayes Rule Components

Bayesian inference is rooted in the sound foundations of probability theory, and bolstered by technological advancements. Up until the past few decades, the update step in Bayesian inference has been the main reason for its computational complexity, rendering Bayesian inference unfeasible.

2.2 Biased Coin Example

Consider an agent who wants to estimate the bias of a coin using Bayesian inference. Let parameter θ[1] denote the unknown bias. Then the objective is to estimate θ given observed data $x_1, x_2, ..., x_n$[2] generated from n coin flips, where $x \in \{H, T\}$.

2.2.1 Statistical Model

Selecting a statistical model is an important step in Bayesian inference. Since this problem has only two outcomes for each experiment, H or T, it can be modeled as a Bernoulli distribution.

[1] θ is comprised of 1 component.
[2] x is comprised of 1 component.

Remark 2.1: Bernoulli Distribution

The Bernoulli distribution models experiments with two outcomes, $\{0, 1\}$. Hence, it is described as a probability distribution of a random variable that takes value 1 with probability p.

Let an observed experiment outcome of H be encoded as the value 1. Then a Bernoulli distribution can be used to model the bias of a coin by setting $p = \theta$, the unknown parameter.

Prior Distribution. Before experiments are performed, an initial belief about θ is set, and is referred to as the prior probability distribution, or simply the *prior*. Usually, the uniform distribution is used as the prior to encode no preference, as it encodes all outcomes with equal probability. This is $\pi(\boldsymbol{\theta})$ in Equation 2.1.

Likelihood. Conditioning on the observed data x, as though they are fixed, the parameter θ is allowed to vary to find the θ that makes it most likely to observe x. This is $\pi(\mathbf{x}_{1:n} \mid \boldsymbol{\theta})$ in Equation 2.1.

Posterior Distribution. The parameter values which maximize the chance of observing data x, while taking into account the prior. It is referred to as the posterior distribution, or simply the *posterior*, and is $\pi(\boldsymbol{\theta} \mid \mathbf{x}_{1:n})$ in Equation 2.1.

The $posterior, likelihood$ and $prior$ have the following relationship:

$$\pi(\theta \mid x) \propto \pi(x_{1:n} \mid \theta)\pi(\theta)$$

Remark 2.2: Total Probability Rule

If x_n and θ are disjoint events whose union covers the entire state space, then:

$$\pi(x) = \int_{\theta \in \Theta} \pi(x|\Theta = \theta)\pi(\theta)d\theta$$

Remark 2.3: Beta Distribution

The Beta probability distribution function, is a two parameter continuous distribution with range $[0, 1]$, and has an analytic solution, see Appendix B.1.2. The Beta pdf is given as follows:

$$\pi(\theta \mid \alpha, \beta) \stackrel{\Delta}{=} \frac{1}{B(\alpha, \beta)} \theta^{\alpha-1}(1-\theta)^{\beta-1} \tag{2.2}$$

For hyperparamters $\alpha, \beta \in \mathcal{N}^+$, as they are in this example, it has the following analytical solution:

$$\pi(\theta \mid \alpha, \beta) = \frac{(\alpha+\beta-1)!}{(\alpha-1)!(\beta-1)!} \theta^{\alpha-1}(1-\theta)^{\beta-1} \tag{2.3}$$

Remark 2.4: Conjugate Prior

A prior distribution is a conjugate prior if it belongs to the same family of probability distributions as the posterior distribution. In this biased coin example, the prior $\pi(\theta)$ is the conjugate prior for the posterior $\pi(\theta \mid x_{1:n})$.

The uniform distribution is a special case of the Beta distribution, $\pi(\theta \mid \alpha, \beta)$, when $\alpha, \beta = 1$. Hence the prior for this example is $\pi(\theta \mid 1, 1)$.

Using the Beta distribution as the prior is an important choice in this example, since the Beta distribution is the conjugate prior for the Bernoulli distribution. As a result, the posterior is a Beta distribution and has an analytic solution that can be computed in closed-form[3]. Hence, evidence from Bernoulli trials can analytically update the prior.

2.2.2 Experiments

Experiments consist of coin flips, and observed data is either H or T. Starting with an initial belief about the unknown bias θ, the current belief about θ is iteratively updated to incorporate new evidence, thereby computing the posterior. The posterior distribution is computed by summing the number of observations of H and T, then assigning them to α and β, respectively.

[3]Can be computed in a finite number of steps.

Remark 2.5: Posterior pdf of a Prior Beta distribution

Starting with a conjugate prior as the uniform distribution, it follows that the posterior of the Beta distribution is given by:

$$\pi(\theta \mid x_{1:n}) = \pi(\theta \mid \alpha + 1, \beta + 1) \qquad for \;\; \alpha, \beta \in \mathcal{N}^+ \tag{2.4}$$

where α, β are the sum of H and T, respectively; x is the observed data of a trial, and $x = \alpha + \beta$.

The values $\alpha + 1, \beta + 1$ incorporate the evidence α, β into the prior $(1, 1)$, see Equation 2.4. Specifically, they are used as arguments in Equation 2.3 to compute the posterior, i.e., an update to the Beta distribution. Observe how the Beta distribution evolves as evidence is generated and posteriors are updated, in Figure 2.1a.

For instance, in Trial: 3, the total H is 1, hence, $H = 1$ and $T = 2$. It follows from equation 2.4 that the posterior at this iteration is computed as $\pi(\theta; 2, 3)$, and its plot as given. The analytical closed-form solution for the posterior distribution at this iteration is given by Equation 2.3 as:

$$\pi(\theta \mid 2, 3) = \frac{(2 + 3 - 1)!}{(2 - 1)!(3 - 1)!}\theta^{2-1}(1 - \theta)^{3-1}$$

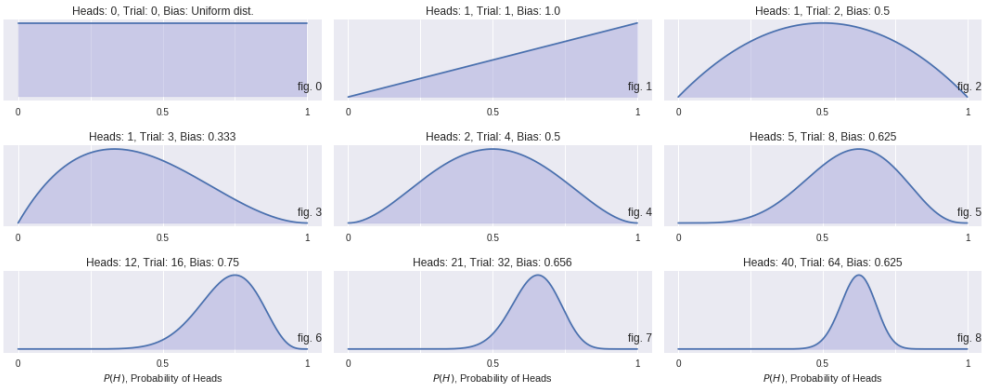
In the last trial, Trial: 64, $H = 40$ and $T = 24$. By equation 2.4, it follows that the Beta distribution at this iteration is $\pi(\theta; 41, 25)$, and its plot as given. The analytical closed-form solution for the posterior distribution at this iteration is given by Equation 2.3 as:

$$\pi(\theta \mid 41, 25) = \frac{(63)!}{(40)!(24)!}\theta^{40}(1 - \theta)^{24}$$

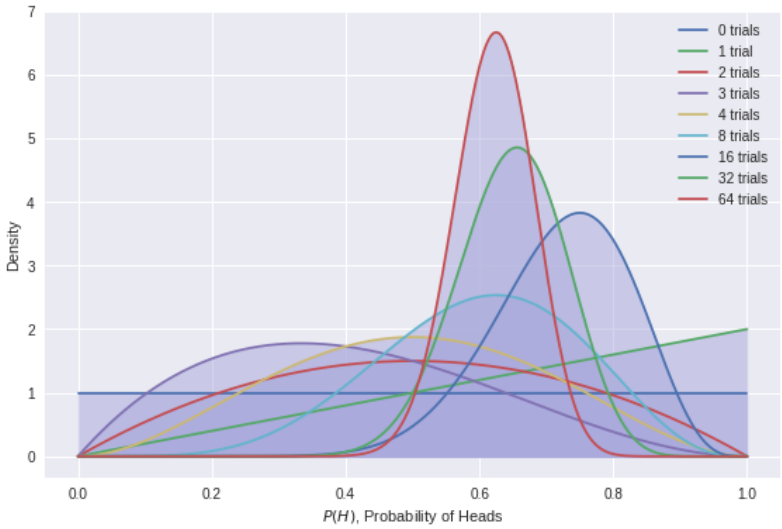
See Figure 2.1b for how, after many experiments, the posterior $\pi(x_n \mid \theta)$ takes a unimodal and peaked shape, illustrative of the final estimate for θ.

2.2.3 Computing Probability Statements

The objective from sampling consecutive probability distributions is to infer a probability statement about the unknown bias θ. For instance, the probability that the bias assumes a particular value, say θ, and belongs in the interval $[0.5, 0.7]$, is given by the area under the curve with said bounds. See Figure 2.2.



(a) Updates per iteration



(b) Consolidated Updates

Figure 2.1: Estimates of θ are given by posterior distributions modeled as the Beta distribution, and updated given evidence. Figure 1.1(a) illustrates Beta distributions which is used to model the estimate for θ. From left to right, and top to bottom, evidence from trials consecutively update the distributions to incorporate evidence. Observe that distributions take a unimodal shape and become increasingly peaked, as the number of trials increases, Figure 1.1(b) superimposes the different trials to illustrate the progress of estimating θ, indicating the value of θ to be near 0.7.

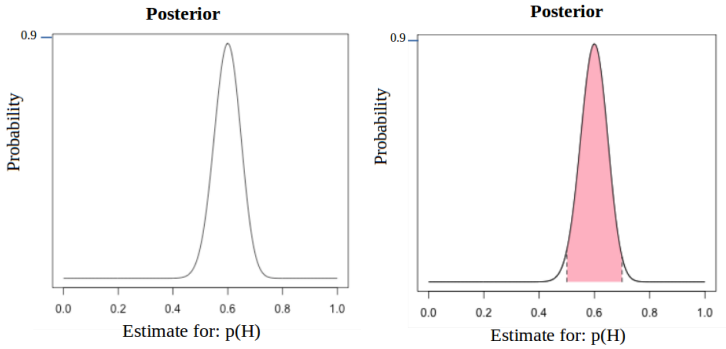


Figure 2.2: The left figure shows the posterior distribution, i.e., possible values for θ. In the right figure, interval $[0.5, 0.7]$ is chosen. Then the probability that θ belongs in this interval is given by the highlighted area under the probability curve.

Example 2.1: Probability Statement from Experiments

A probability statement for θ at Trial: 64, is inferred. Take the interval $[0.5, 0.7]$ as in Figure 2.2, then the probability statement for θ is the probability that θ belongs to this interval. It is given by:

$$\begin{aligned} P(\theta \in [0.5, 0.7]) &= \int_{0.5}^{0.7} \pi(\theta \mid 41, 25) d\theta \\ &= \pi(\theta \mid 41, 25) = \frac{(63)!}{(40)!(24)!} \theta^{40}(1-\theta)^{24} \\ &= 0.887 \end{aligned}$$

2.3 Bayesian Inference

The objective of Bayesian inference is to determine, using Bayes rule, the posterior probability distribution of unknown parameter $\boldsymbol{\theta}$. The posterior is iteratively updated as evidence becomes available, and these steps mostly require an ability to evaluate integrals.

Practical use for Bayesian inference is realized in complex problems, see Remark 2.10. In such problems, the corresponding integrals are almost always intractable. Consequently, Bayesian inference problems become integration problems.

2.3.1 Evaluating Integrals

The majority of integrals are *analytically intractable*, i.e., cannot be solved in closed-form. Numerical integration works well in low dimensional problems, under 10 dimensions as a general rule. However, the vast majority of problems within the Bayesian inference paradigm are complex, see Remark 2.10. Sampling methods offer alternatives to integral evaluation, and are the topic of this thesis.

Three methods for evaluating integrals are herein explained:

- *Analytical* methods;
- *Numerical* methods, and;
- *Sampling* methods.

Analytical Solutions

An integral can be computed analytically when it has closed-form, or exact, solutions. Otherwise, an integral is analytically intractable, i.e., no tractable exact solutions exist.

For an example of a Bayesian inference problem with analytical solutions, consider the biased coin example, see Section 2.2. At Trial: 64, the final step, the posterior was computed and a probability statement was inferred, see Example 2.1. The probability that θ belongs to the interval $0.5, 0.7$ was:

$$P(\theta \in [0.5, 0.7]) = 0.887$$

The reason this was done analytically is because the Beta distribution can be computed in closed-form.

Analytical solutions do not exist for the majority of integrals. In effect, they are rarely feasible in scientific problems. The following are two alternative solutions to computing integrals, namely: numerical methods, and sampling.

Numerical Method

When integrals cannot be computed analytically, i.e., no closed-form exists, they can be approximated using numerical methods. An example of such methods is Riemann sums.

The integral of the function $f(x)$ over its domain $\Omega = [a, b]$, see Figure 2.3 for the one dimensional case, can be approximated for large N as follows:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{N} f(x_i)\triangle x$$

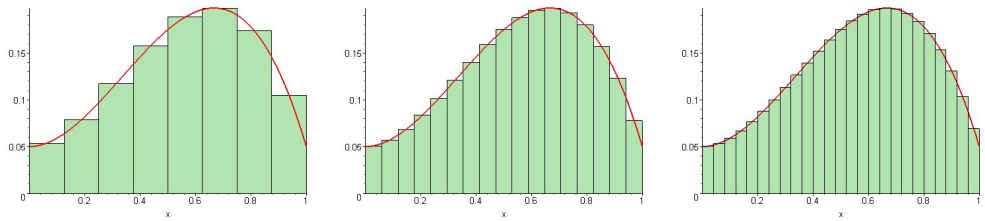


Figure 2.3: Riemann sums approximating the area under the function curve using midpoints of the interval for computing function values. Interval size from left to right, 8, 18, 24 intervals, respectively. Observe that as partition width sizes decrease, errors shrink.

In evaluating the integral, the function domain is divided into N intervals. One variation of Riemann sums is to compute the function value at the midpoint of each interval. Let these midpoints be $(x_1, x_2, ..., x_N)$ where $x_i \in \mathcal{R}$, and let the width of all intervals be equal, and denoted by $\triangle x$. Then the integral $f(x)$ over its domain $[a, b]$ is evaluated by averaging over all function values, see Figure 2.3.

Remark 2.6: Lebesgue Measure

The Lebesgue measure is being used here, where points are uniformly distributed over the function domain with interval width $\triangle x$; and where the function $f(x)$ is always positive. For this midpoint Riemann sums case, all intervals account for the same contribution towards integral approximation.

The subintervals as constructed in Figure 2.3, result in rectangular shaped areas defined by the interval width and the height given by the function value evaluated at the midpoint of the interval. An estimate for the integral is then computed by summing all these areas. This estimate converges to the true value of the integral as $\triangle \mathbf{x} \to 0$.

Errors arise as rectangles either overestimate or underestimate the area under the curve. The left figure in Figure 2.3 best illustrates the errors. Errors, $E_r f(\mathbf{x})$, given by:

$$E_r f(x) \triangleq \left| \int_a^b f(x)dx - \sum_{i=1}^{N} f(x_i)\triangle x_i \right|$$

The total sum of errors has an *exponential time complexity*, proportional to its dimensionality. The number of calculations required for the summation of all errors is proportional to N^d; $d \in \mathcal{N}$ where d represents problem dimensionality. As dimensionality increases,

the exponential time complexity becomes the limiting obstacle. This limitation is known as the *Curse of Dimensionality*.

Remark 2.7: Time Complexity for Numerical Integration

The integral for $f(x)$, where $x \in \mathcal{R}$, is given by:

$$\int_a^b f(x)dx$$

However, the integral for $f(\mathbf{x})$, where $\mathbf{x} \in \mathcal{R}^d$, is given by integrating over each of the d dimensions for each multivariate vector:

$$\int_a^b f(\mathbf{x})d\mathbf{x} = \int_{a_1}^{b_1} f(x_1)dx_1 \int_{a_2}^{b_2} f(x_2)dx_2 \cdots \int_{a_d}^{b_d} f(x_d)dx_d$$

Time complexity for numerical methods integration is in the order of $N^d; N, d \in \mathcal{N}$, where N is the number of partitions, and d the dimensionality.
This tedious computation is known as the *curse of dimensionality*.

Technological advances made possible other methods for evaluating integrals. Such are sampling methods, known as Monte Carlo (MC) methods, which overcome the curse of dimensionality.

Monte Carlo Methods

Monte Carlo methods (MC) generate N random samples according to some probability distribution π. This is in contrast to numerical methods which deterministically samples from all subintervals of a function domain.

MC methods estimate function integrals through two fundamental rules, namely:

- ***Law of Large Numbers*** *(LLN)*: as the number of samples, N, increases, the sample mean, $\bar{x}_n$, will converge to its actual mean, μ, and;
- ***Central Limit Theorem*** *(CLT)*: as the number of samples, n, increases, the *distribution* of the sample mean,$\overline{x}_n$, approaches the Gaussian distribution.

The LLN provides point-wise information about random variables but not information about their distribution. Without a distribution, no information about the rate to which $\bar{x}_n$ approaches the constant μ can be determined. Obtaining information about the distribution of $\bar{x}_n$ is by applying the *CLT*. In most cases, it provides this information regardless of the distribution of the random variable, x.

Generated samples are then used to estimate statistical moments, e.g. first and second moments, the mean and variance, respectively.

Definition 2.1: Sample Mean

Sample mean is an estimate of the population mean, and is given by:

$$\bar{\mathbf{x}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i \tag{2.5}$$

Definition 2.2: Sample Variance

The sample variance measures the dispersion of samples, and is given by:

$$s^2 = \frac{1}{N-1}\sum_{i=1}^{N}(\mathbf{x}_i - \bar{\mathbf{x}})^2 \tag{2.6}$$

The *LLN* B.2 allows MC methods to estimate an integral using the samples generated. This is given by:

$$\int_{\mathbf{\Omega}} f(\mathbf{x})\pi(\mathbf{x})d\mathbf{x} \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_i) \tag{2.7}$$

where $\pi(\mathbf{x})d\mathbf{x}$ is the *probability measure*, i.e.:

$$\int_{\Omega} \pi(\mathbf{x})d\mathbf{x} = 1$$

When using the Lebesgue measure in Equation(2.3.1), samples are uniformly generated and have equal contributions to integral evaluations. However, using the probability measure, both *values* and their *contribution* to integral estimation are considered when using the probability measure in Equation(2.7). This means that the weight of a sample $\mathbf{x}_i$ is determined by the value $\pi(\mathbf{x}_i)$.

Remark 2.8: Time Complexity of MC Methods

Time Complexity of MC methods is proportional to $\mathcal{O}(\frac{1}{\sqrt{N}})$, where N is the number of samples.

Time complexity of MC methods is proportional to $\mathcal{O}(\frac{1}{\sqrt{N}})$, i.e., independent from dimensionality. This is due to the CLT, see Section B.3. A disadvantage of MC methods is that they require large sample sizes to obtain smaller errors.

For instance, to achieve a precision of 10%, 100 independent and identically distributed $(i.i.d.)$ samples are needed, but $10,000$ i.i.d. samples are needed when the precision is limited to 1%. The cost associated with ever smaller errors renders numerical methods advantageous over sampling methods.

Two variants of MC methods are: *Vanilla* MC that generate $i.i.d.$ samples, and *Markov Chain* MC that generate correlated MC samples.

Remark 2.9: Vanilla Monte Carlo

What is often referred to as Vanilla Monte Carlo is the basic MC method from which $i.i.d.$ samples can directly be generated from some known and basic probability distribution. Such distributions are the Uniform $U[a, b]$, Beta, and Normal $\mathcal{N}(\boldsymbol{\mu}, \sigma^2)$ distributions.

Vanilla MC methods are generally used in sampling, simulations (e.g. 1-dimensional random walk, for asking *"what if"* questions), and for numerical integration. And generally $\pi(x)$ is the pdf for the uniform distribution. Vanilla Monte Carlo methods generate $i.i.d.$ samples only from a limited number of probability distributions, e.g., uniform, Beta, and Gaussian distributions.

MCMC

MCMC methods can sample from any distribution, thereby overcoming limitations when sampling from complex distributions; and are independent of dimensionality, see Chapter 3.

A sample is generated from a known distribution, and its probability (weight) is given by this pdf. It is then evaluated against its probability given by the target distribution. The proposed sample is then probabilistically accepted/rejected as a potential addition to the Markov chain.

Example 2.2: MCMC Simulation

Fig. 2.4 illustrates a simulation from an MCMC algorithm. A histogram for samples generated from the proposal distribution overlays the target distribution. These samples are generated after the burn-in period, after which all samples are generated from the target distribution with probability $p = 1$. It samples successfully from a bi-modal distribution, considered as a complex distribution.

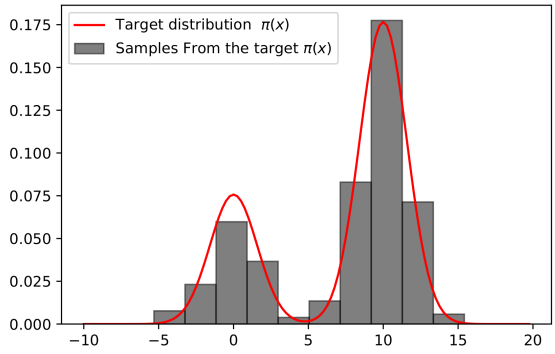


Figure 2.4: Histogram for samples generated from an MH simulation with a bimodal invariant distribution $\pi(x)$ and standard Gaussian proposal $q(x) = \mathcal{N}(0, 1)$. $\pi(x)$ has mean $= \mu_1 = 0, \mu_2 = 10$ and $\sigma_=10$

Complex Distributions

Remark 2.10: Complex Distribution

Complex distributions are generally described by at least one of the characteristics of being multi-modal, heavy-tailed, or high dimensional distributions, see Figure2.6.

Definition 2.3: Multi-modal distribution

A distribution is multi-modal if it has more than one peak.

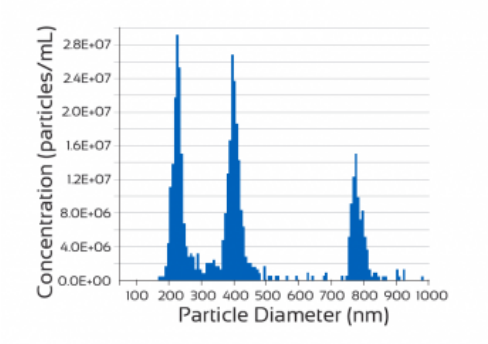


Figure 2.5: The figure illustrates a multi-modal distribution with three peaks. See here for reference.

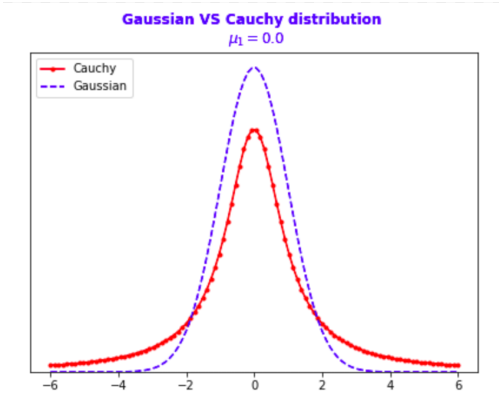


Figure 2.6: The figure illustrates the difference between the Cauchy distribution and the Gaussian as $x \to \infty$. For large x, the Cauchy distribution is proportional to polynomial decay, given by $x^{-\alpha}$ with $\alpha > 1$. While, for large x, the Gaussian distribution is proportional to exponential decay, given by e^{x}. This means that the Gaussian distribution approaches probability 0 faster

Definition 2.4: Heavy-Tailed distribution

Heavy-tailed distributions are probability distributions whose pdf decreases for large values of x, see Figure 2.6. The decay of the distribution is proportional to the polynomial distribution, i.e., proportional to $x^{-\alpha}$ with $\alpha > 1$.
If the distribution $f(x \mid \alpha)$ is heavy-tailed, then:

$$f(x \mid \alpha) \propto x^{-\alpha}; \qquad \text{for } x \in [0, \infty] \text{ and } \alpha > 1$$

Heavy-tailed distributions are probability distributions where extreme events occur somewhat frequently. This means that there is considerable mass at the tails of the distribution.

The topic of MCMC methods is taken up in the next chapter. Before doing so, a stop is due at two different paradigms, in philosophy and eventually in approach, for making probability statements, namely: the Bayesian and frequentist paradigms on inferential statistics, is the topic of the next section.

2.4 Bayesianism and Frequentism

Two proponents have been embattled silently for more than two centuries over their approaches to statistical inference(McGrayne, 2011). Until recently, the frequentist paradigm, also coined the classical approach, has dominated the field of statistics. But the Bayesian paradigm has been increasing in popularity due to its power and usability.

Computational capabilities of the past two centuries are well-suited for frequentist computational needs. Over the course of the last century, however, the Bayesian approach took-off contemporaneously with developments in sampling methods, such as MCMCs, and advancements in computational capabilities. It has since been rippling through the scientific fields.

2.4.1 Frequentist and Bayesian Models

Consider the biased coin example in Section 2.2. Recall that the observed data was $40H$ from a total of 64 coin flips. The difference between Bayesian and frequentist approaches can be attributed to a fundamental difference in their respective interpretation of probability(VanderPlas, 2014).

Frequentist Model

In a frequentist model M, the unknown parameter θ is a *constant*. A long-run of data is generated from M in each run of the algorithm, i.e., different data is generated per run. The frequentist estimates θ by measuring frequencies of events. Hence, a frequentist defines probability of an event as the frequency in which the event is *empirically* observed to occur.

In the biased coin example, the estimate for θ is computed from the descriptive statistic of the sample mean, see Definition 2.1. The sample mean at iteration 64 was computed as: $\bar{x} = 41/64 \approx 0.641$. This result is a single value for the estimate of θ.

Remark 2.11: Frequentist Probability Statement

The frequentist approach estimates θ from long-run observations from which their frequency of occurrence if computed.

Evidence about θ is used to compute its frequency of observation, from which a Confidence Interval (CI) is constructed. As new data is generated, successive CIs are constructed that eventually will converge around the true mean and with a smaller variance. This represents the frequentist probability statement about θ; a point estimate captured by the mean of the CI.

Bayesian Model

In a Bayesian model M, the unknown parameter θ is treated as a *random variable*, but the input data is *constant*. A long-run of data is generated in the initial algorithm step. Bayesian probability statements are probability distributions for different values of the parameter θ.

Remark 2.12: Bayesian Probability Statement

The Bayesian approach models θ as a *random variable* while fixing the input data. A domain interval is chosen from which the value θ will belong, with some probability.

The Bayesian approach computes the probability that θ belongs to a domain interval of the posterior distribution, on which the probability depends. In the biased coin example, it was computed that at iteration 64, θ belonged in the chosen[4] interval $[0.5, 0.7]$ with probability 0.887, see Example 2.1. In this approach, evidence for H was incorporated into the prior distribution to estimate the unknown bias of θ.

This interval is coined *Credible Region* (CR), and represents the probability statement the Bayesian set out to make. Hence, the Bayesian approach seeks to infer the value of θ through constructing a CR.

[4] Another interval could've just as likely been chosen, but it makes sense to choose an interval that is likely to contain the true mean.

Remark 2.13: Credible Region and Confidence Interval

For a Bayesian, probability represents the degree of belief about θ, given evidence. A Bayesian computes a probability distribution that assigns weights, given evidence, to each value of the unknown bias θ. Then they makes probability statements about θ belonging to a chosen interval of the probability domain. This interval is called the *Credible Region*. As this interval is chosen differently, the probability statement will change.
For a Frequentist, the probability represents a point estimate for θ. Then they construct intervals, called confidence intervals, which possibly contain θ. The frequency in which confidence intervals contain the true value for θ is called the confidence level. The frequentist conveys their probability statement for the value of θ through the confidence level.

The following table illustrates conceptual model differences of the two approaches:

	Frequentist	**Bayesian**
Data	Vary	Fixed
Parameters	Fixed	Vary

Table 2.2: Frequentist and Bayesian approaches to parameter and data variations within their respective models.

2.4.2 Models Diverge

The differences between Bayesian and frequentist approaches for solving simple models tend to reduce to philosophical discussions with minimal impact on final results. However, in complex models (see 2.10), Bayesian and frequentist results do diverge as a result of their respective interpretation of probability.

VanderPlas (2014) suggests the divergence of the two approaches can mostly be explained in two ways:

- Interpretation of uncertainty;
- Incorporation of a prior, and;
- Number of models considered.

Handling Uncertainties

Each approach quantifies uncertainties in different conceptual and computational ways. A frequentist uses *confidence intervals*, while a Bayesian uses *credible regions*.

A confidence interval is an interval over values of θ *determined* by the CLT, see Section 2.3.1. Estimates about θ are expressed by this confidence level.

A credible region is a *chosen* interval from the probability distribution domain. It represents the probability with which θ belong to this domain.

By varying the interval endpoints, different probabilities for θ are inferred. Hence, Bayesian estimates of θ are constructed around fixed endpoints.

Incorporation of a Prior

It is common that some information is known about a problem before any experiments are conducted. By using Bayes rule, a Bayesian approach can easily incorporate any information into models, especially nuisance parameters.

Number of models considered

By construction, the frequentist approach uses a single model; i.e., the probability model is chosen and the parameter θ is fixed. A Bayesian considers all possible models as θ is a random variable.

2.5 Chapter Conclusion

This chapter showed that a crucial step in Bayesian inference computation is finding the likelihood. This depends on the ability to evaluate integrals. Accordingly, various methods for evaluating integrals was provided: analytical methods, but this was found applicable only in limited cases; numerical methods which faced the curse of dimensionality; vanilla methods generate i.i.d. samples and use probability theories to evaluate the first and second statistical moments of a distribution, but generating enough samples is a challenge, and they can generate samples only from a limited number of distributions; and finally, MCMC methods which are able to overcome many said challenges and limitations.

The next chapter covers the evolution in sampling methods that led up to adaptive MCMC methods. It will be seen that MCMC methods work for all probability distributions, by constructing a Markov chain. The chapter will then conclude with adaptive MCMC methods.

3 | Markov Chain Monte Carlo

Markov Chain Monte Carlo (MCMC) methods construct a Markov chain that has as its an invariant distribution, the posterior distribution. A large number of samples are generated where each new value depends only on the current value, hence, a correlation exists between any two adjacent values. MCMC methods are used to sample from any distribution, and evaluate the otherwise intractable integrals.

This chapter explains the underlying theories and statistical principles underpinning MCMC methods. Starting with Monte Carlo (MC) sampling, see Section 2.3.1, and then explaining Markov chains, the building blocks for MCMC methods are laid out. The Metropolis Hastings algorithm (hereinafter, MH) will be presented. Review of the structure of MH ensues followed by a discussion on the challenges facing current MH samplers. The chapter concludes with detailing a number of convergence measurement tools in MCMC, commonly referred to as convergence diagnostic tools.

3.1 Markov Chain

As used in this thesis, a *Markov chain* is a stochastic process[1] of $n \in \mathbb{N}^+$ discrete-time values and continuous space values. It evolves over time, transitioning from one value to another within a given state space. Dependencies exist only between adjacent values in the chain.

[1] A stochastic process is a probabilistic model.

Remark 3.1: Markov Chain Annotation

The Markov chain is annotated in the finite discrete state space by $S \triangleq s_{1:n} = s_1, ..., s_{n-1}, s_n,$, whereas for continuous state space by $X \triangleq x_{1:n} = x_1, ..., x_{n-1}, x_n$.

Definition 3.1: Markov Chain

A sequence of random values, $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n$ is a Markov chain if the probability of observing the value $\mathbf{x}_{n+1}$ depends only on the current value, $\mathbf{x}_n$. For all $n \geq 0$, it follows that:

$$\pi(\mathbf{x}_{n+1} \mid \mathbf{x}_n, \mathbf{x}_{n-1}, ..., \mathbf{x}_0) = \pi(\mathbf{x}_{n+1} \mid \mathbf{x}_n)$$

The objective is to construct Markov chains that when simulated, result in a random walk that explores the target distribution. This means the values are stochastically distributed, and their distribution is known as the *invariant* distribution, denoted Π. This invariant distribution is the distribution from which samples must be generated.

The invariant distribution can be sampled from by constructing a stochastic matrix T, referred to as the *transition matrix*. Elements of the matrix are probabilities for transitioning between two adjacent values, hence, it specifies the transition rules.

3.1.1 Transition Matrix

Adjacent dependencies of Markov values can be captured in a probability matrix called the *transition matrix*. The matrix T is square shaped, real, and positive. Elements of T represent the probability of transitioning from one value to another, where:

$$T_{n,n+1} = T(\mathbf{x}_{n+1} \leftarrow \mathbf{x}_n)$$

Definition 3.2: Transition Matrix

For any pair of values $\mathbf{x}_n, \mathbf{x}_{n+1}$ in a Markov chain, a transition probability $\pi(\mathbf{x}_{n+1}, \mathbf{x}_n)$, specifies the likelihood of moving from value $\mathbf{x}_n$ to value $\mathbf{x}_{n+1}$.

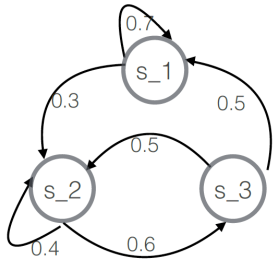
Consider a state diagram of a set of three values and the paths which connect the values. The probability of transitioning from one value to the other is placed along corresponding paths. This probability is referred to as the transition probability and constitutes entries of the transition matrix T. Figure 3.1 illustrates this transition matrix.

Example 3.1: Three Markov values and their Transition Matrix

Consider the discrete state space of s_1, s_2, s_3 as illustrated in Figure3.1. Observe from the state diagram that the probability of being in a particular value at time $n+1$ in the state space, depends only on the value at time n.

For each transition, a probability is shown, e.g $\pi(s_1|s_3) = 0.5$ and $\pi(s_1|s_1) = 0.7$. Observe that there is no self-transition for value x_3, where $\pi(x_3|x_3) = 0.0$.

The values and the probabilities for traversing between them, the transition matrix, T, is shown in the right figure.



$$T = \begin{bmatrix} 0.7 & 0.3 & 0.0 \\ 0.0 & 0.4 & 0.6 \\ 0.5 & 0.5 & 0.0 \end{bmatrix}$$

Figure 3.1: The figure to the left is a state diagram for values and their transition probabilities. And on the right is its corresponding transition probability matrix, or transition matrix.

3.1.2 Invariant Distribution

An invariant distribution is a vector in the finite discrete-time Markov chain, denoted Π. This vector represents the probability distribution of values of the chain; probability distribution of $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n$. For each value $\mathbf{x}_i$ in a Markov chain, Π satisfies $\pi(\mathbf{x}_i) > 0$ and $\sum_X \pi(\mathbf{x}) = 1$.

Definition 3.3: Invariant Distribution

For *each* value $\mathbf{x}$ in the state space $\mathbf{\Omega}$, and *every* value $\mathbf{x}_{n+1}$, the invariant distribution Π of the Markov chain holds that:

$$\sum_{\mathbf{\Omega}} T(\mathbf{x}_n, \mathbf{x}_{n+1})\pi(\mathbf{x}_n) = \pi(\mathbf{x}_{n+1}) \tag{3.1}$$

This means Π is an invariant distribution of T, and satisfies:

$$T\Pi = \Pi$$

Remark 3.2: Invariant Distribution as Eigenvector

$T\Pi = \Pi$ implies that the distribution Π is an eigenvector of transition matrix T with eigenvalue $= 1$.

A transition matrix is needed in order to sample from the invariant distribution.

Remark 3.3: Uniqueness of the Invariant Distribution

A unique invariant distribution Π of the Markov chain exists if it is *irreducible* and *aperiodic*.

Definition 3.4: Irreducibility

A Markov chain is considered irreducible if every value can eventually be reached in a finite number of steps, i.e., the values are represented in a connected graph. This means, there exists a positive integer such that for any n, $\pi(\mathbf{x}_{n+1}; \mathbf{x}_n) > 0$.

Definition 3.5: Aperiodicity

A Markov chain is aperiodic when the evolution of the chain does not become indefinitely trapped in a cycle.

3.1.3 Ergodicity

A Markov chain is ergodic if it is possible to visit any value of the domain from any other value. A chain is said to be ergodic if it is irreducible, aperiodic and positive recurrent.

Definition 3.6: Ergodicity

Every value $\mathbf{x}$ in the distribution Π is reachable with probability $\pi(\mathbf{x}) > 0$.

3.1.4 Reversibility

A transition matrix needs to be ergodic and have a unique invariant distribution in order to generate a Markov chain with the needed unique invariant distribution.

However, with a large state space, constructing T becomes intractable. Instead of building an entire transition matrix, the Markov chain is made to satisfy the *reversibility* condition (Liang et al., 2011), where its unique Π can be obtained.

Definition 3.7: Reversibility

A Markov chain which has a transition matrix T and invariant distribution Π is reversible if it satisfies the condition:

$$T(\mathbf{x}_{n+1} \leftarrow \mathbf{x}_n)\pi(\mathbf{x}_n) = T(\mathbf{x}_n \leftarrow \mathbf{x}_{n+1})\pi(\mathbf{x}_{n+1}) \tag{3.2}$$

For proof, see Proof 3.1.

Proof 3.1: Reversibility

Let Q be a known distribution which is feasible to sample from. Then:

$$\begin{aligned}
&\pi(\mathbf{x}_n)T(\mathbf{x}_{n+1} \leftarrow \mathbf{x}_n) = \pi(\mathbf{x}_n)Q(\mathbf{x}_{n+1}; \mathbf{x}_n) \min\left(1, \tfrac{\pi(\mathbf{x}_{n+1})Q(\mathbf{x}_n; \mathbf{x}_{n+1})}{\pi(\mathbf{x}_n)Q(\mathbf{x}_{n+1}; \mathbf{x}_n)}\right) \\
&= \min\left(\pi(\mathbf{x}_n)Q(\mathbf{x}_{n+1}; \mathbf{x}_n), \pi(\mathbf{x}_{n+1})Q(\mathbf{x}_n; \mathbf{x}_{n+1})\right) \\
&= \pi(\mathbf{x}_{n+1})Q(\mathbf{x}_n; \mathbf{x}_{n+1}) \min\left(1, \tfrac{\pi(\mathbf{x}_n)Q(\mathbf{x}_{n+1}; \mathbf{x}_n)}{\pi(\mathbf{x}_{n+1})Q(\mathbf{x}_n; \mathbf{x}_{n+1})}\right) \\
&= \pi(\mathbf{x}_{n+1})T(\mathbf{x}_n \leftarrow \mathbf{x}_{n+1})
\end{aligned}$$

□

Remark 3.4: Reversibility Implies the Invariant Condition

If the Markov chain satisfies reversibility, then the invariant condition is implied (MacKay, 2003).

3.1.5 Generalized Reversibility

Reversibility is a sufficient condition but not necessary. A necessary condition is generalized reversibility. If a unique invariant distribution for T exists, then the reversibility condition is generalized by using a reverse transition matrix for T, denoted Q. Together they satisfy the reversibility condition. This new condition is known as *generalized reversibility* (Murray).

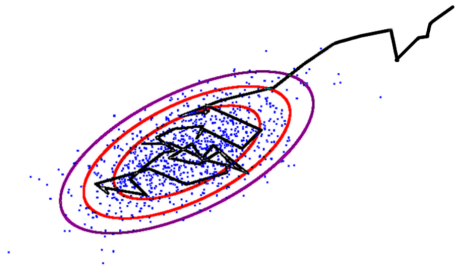


Figure 3.2: The figure illustrates the evolution of a Markov chain from one transition matrix. This is done by generating samples according to this transition matrix. The black line represents initial steps taken to demonstrate the burn-in period. The blue dots represent values of the Markov chain. The distribution of the blue dots will eventually be according to the target distribution.

Definition 3.8: Generalized Reversibility

A Markov chain which has a transition matrix T, a unique invariant distribution Π, and a reverse transition matrix Q, is reversible if it satisfies the condition:

$$T(\mathbf{x}_{n+1} \leftarrow \mathbf{x}_n)\pi(\mathbf{x}_n) = Q(\mathbf{x}_n \leftarrow \mathbf{x}_{n+1})\pi(\mathbf{x}_{n+1}) \tag{3.3}$$

3.2 Markov chain Monte Carlo

MCMC algorithms are a class of algorithms used to approximate a posterior distribution for an unknown parameter. MCMC methods construct a Markov Chain whose values belong to high probability regions of the target distribution. These values are probabilistically chosen, and form the invariant distribution of the particular Markov Chain.

For every *target distribution*, a simpler *proposal* distribution is required to guide in exploring the target distribution, which is often a complex distribution.

After a burn-in period, see Figure 3.2, a Markov chain is considered to have converged to the target distribution. Samples generated after this period are generated according to the target distribution with probability $p = 1$. Estimating this period is not a simple matter, however, see Section 3.4.3.

3.3 Metropolis-Hastings Algorithm

MCMC is synonymous with the Metropolis-Hastings algorithm (MH). The MH algorithm builds a Markov Chain $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n$ by iterating over steps that propose and accept/reject new chain values. Starting from an initial value, MH generates a proposed value, then *probabilistically* determines whether to accept or reject it as the next value in the chain, denoted $\mathbf{x}_{n+1}$.

The proposed value is also called the *proposal*, denoted $\mathbf{x}^*$, and is generated from a *proposal distribution*, denoted $q(\mathbf{x})$. A Gaussian centered at the current value, $\mathcal{N}(\mathbf{x}_n, \sigma^2)$, is an example of $q(\mathbf{x})$. The proposal, $\mathbf{x}^*$, is accepted according to the *acceptance ratio*, denoted $\alpha(\mathbf{x}_n, \mathbf{x}^*)$.

Definition 3.9: Acceptance Ratio

Let $\mathbf{x} \in X$ where X denotes the sample space; $\pi(\mathbf{x})$ the probability for $\mathbf{x}$ given by the *target* distribution π; $q(\mathbf{x})$ the probability for $\mathbf{x}$ given by the *proposal* distribution q; and $\mathbf{x}^*$ be the proposed value from X. Then the acceptance ratio is:

$$\alpha(\mathbf{x}_n, \mathbf{x}^*) = \min\left\{1, \frac{\pi(\mathbf{x}^*)q(\mathbf{x}_n; \mathbf{x}^*)}{\pi(\mathbf{x}_n)q(\mathbf{x}^*; \mathbf{x}_n)}\right\} \tag{3.4}$$

Finally, accept $\mathbf{x}_{n+1} = \mathbf{x}^*$ with probability $a(\mathbf{x}_n, \mathbf{x}^*)$, otherwise set $x_{n+1} = x_n$

Example 3.2: MCMC Simulation

Fig. 3.3 illustrates a simulation from an MH algorithm that probabilistically generates new posterior distributions, given proposed sample x^*. See Chapter on Bayesian Inference Chapter 2 for a discussion on priors, posteriors and the likelihood. The figure illustrates the decision process of new samples. It also illustrates evolution of the posterior distribution.

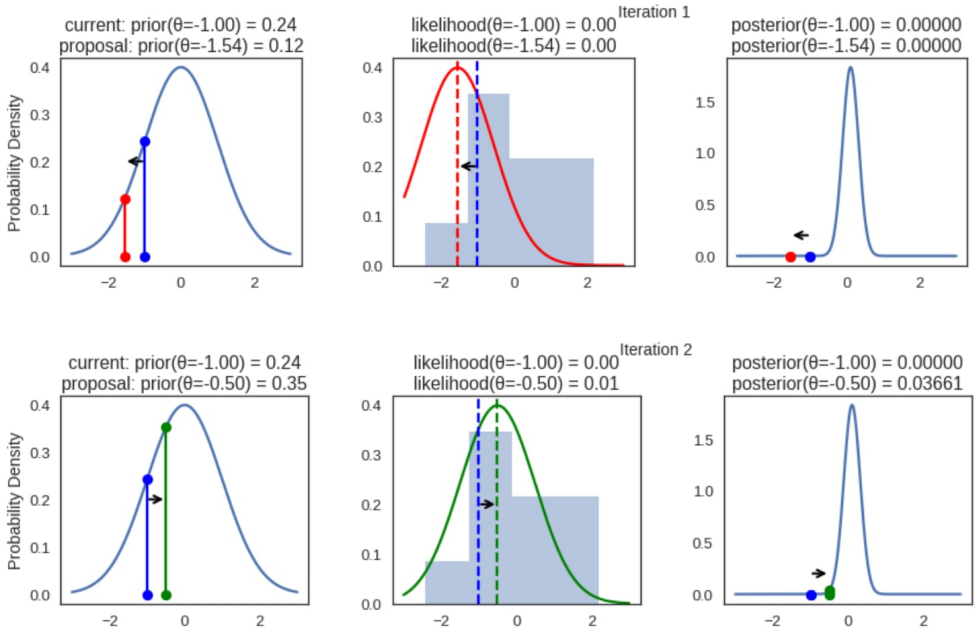


Figure 3.3: The top row shows the evaluation of a rejected proposed sample, in red, while the bottom row shows it for the accepted proposed sample. Figures in the left column are prior distributions; the belief about θ before observing any data. The blue vertical line represents the current value of θ, while the green and red vertical lines represent the accepted or rejected proposed θ, respectively.
Figures in the center column are the likelihood, and represents how well the model explains the data. The histogram represents observed data; the blue vertical lines represent the current value of θ; and the green and red dotted lines represent the accepted or rejected proposed θ, respectively. Observe that the more the likelihood overlaps with histogram data, the more likely the proposed θ will be accepted
Figures in the right column are the posterior distributions, normalized. The blue dots represent the current θ while the green and red dots represent the accepted and rejected proposed values for θ, respectively.

3.3.1 Pseudocode for the MH Algorithm

Algorithm 1: Metropolis-Hastings algorithm

input: π,$\mathbf{x}_0$, c, N

for $n = 1, 2, 3 \cdots N$ **do**

  Generate a candidate value $\mathbf{x}^* \sim q(\mathbf{x}^*|\mathbf{x}_n)$

  compute acceptance criteria:

  $\alpha(\mathbf{x}_n, \mathbf{x}^*) = min\left\{1, \frac{\pi(\mathbf{x}^*)q(\mathbf{x}_n;\mathbf{x}^*)}{\pi(\mathbf{x}_n)q(\mathbf{x}^*;\mathbf{x}_n)}\right\}$

  draw $u \sim U[0, 1]$

  if $u < \alpha(\mathbf{x}_n, x^*)$ **then**

    $\mathbf{x}_{n+1} = \mathbf{x}^*$

  else

    $\mathbf{x}_{n+1} = \mathbf{x}_n$

3.3.2 Effect of Initial Parameter Choice

The proposal distribution in 2.4 has been defined as $q(x) \sim N(0, 1)$. No preferential factor was considered, hence, the proposal distribution could just as likely have been defined as $q(x) \sim N(5, 99)$ or any $q(x) \sim B(n, p)$. Subjectivity of MH parameter decisions is a limiting factor in MH simulations.

Consider Fig. 3.4, where three proposal distributions that only differ by their step size (σ) are used to simulate the same target distribution. The resulting distributions considerably differ in their frequency of accepting new values. Hence the importance of finding an optimal value for σ, coined optimal step size; one which neither rejects at a high rate, nor accepts at a low rate.

3.3.3 Optimal Step Size

Given Gaussian proposal and target distribution, Gelman et al. (1996), Roberts and Rosenthal (2001) found the optimal step size, denoted σ_{opt}, to be $\sigma_{opt} = (2.38/\sqrt{d})$, where d is the dimension of the target distribution.

It follows that the optimal covariance of the proposal distribution $c_{opt} = (2.38/\sqrt{d})c_{target}$, where c_{target} is the covariance of the target distribution.

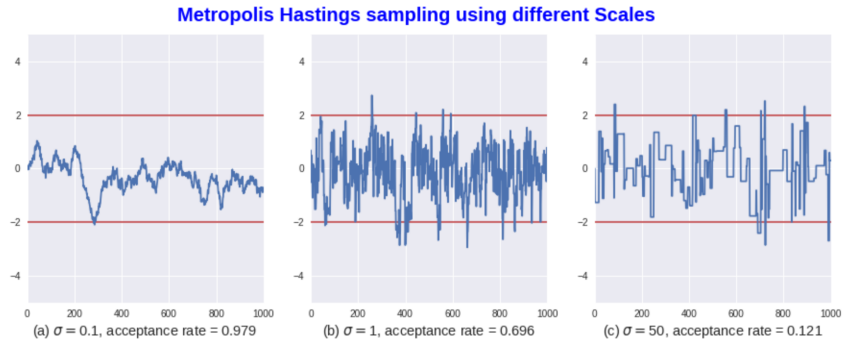


Figure 3.4: MH sampling with a bimodal invariant distribution $\pi(x)$ and standard Gaussian proposal $q(x|.) = \mathcal{N}(0, \mathbf{I})$. $\pi(x)$ has mean $= \mu_1 = 0, \mu_2 = 10$ and $\sigma_{=}10$ The samples generated are presented in a histogram

This optimal scale corresponds to an acceptance rate of 0.234 which Roberts and Rosenthal (2001) had proved optimal. They however observed that practically this acceptance rate is lower in multi-modal distributions and suggest the alternative option of tunning MH parameters.

Remark 3.5: Optimal Scale

The recommended optimal parameter settings which can generate an average optimal acceptance rate of 0.234, given a Gaussian proposal distribution, are:

- An optimal scale $\sigma_{opt} = (2.38/\sqrt{d})$
- Optimal covariance matrix $c_{opt} = (2.38/\sqrt{d})c_{target}$

In order to give reliable estimates, the chain has to *evenly* explore regions of high probability, according to the target distribution. When samples are generated from high probability regions, the chain is said to mix well, or have converged. A rapidly mixing chain has a small autocorrelation length denoted τ.

3.3.4 Burn-in

A Markov process is inherently autocorrelated, but only in the initial runs of an MCMC algorithm. Generally, as more values are generated, autocorrelation *fades* and MCMC estimates become increasingly $i.i.d.$, for an explanation on autocorrelation, see Section 3.4.

Usually the first half of samples are considered biased and therefore, not used. This is clearly visible from plotting search trajectories, see Figure 3.2. Robust burn-in period measurements can be done with tools such as Pymc3's Gelman Rubin and *ESS* tools.

Remark 3.6: Burn-in

It is standard practice to discard initial iterations of MH simulations, as they are heavily biased towards initial conditions. After this period, proposal distributions are said to have converged to their respective target distributions.

At the point when the algorithm generates samples from the target distribution, it will continue to do so with probability $= 1$.

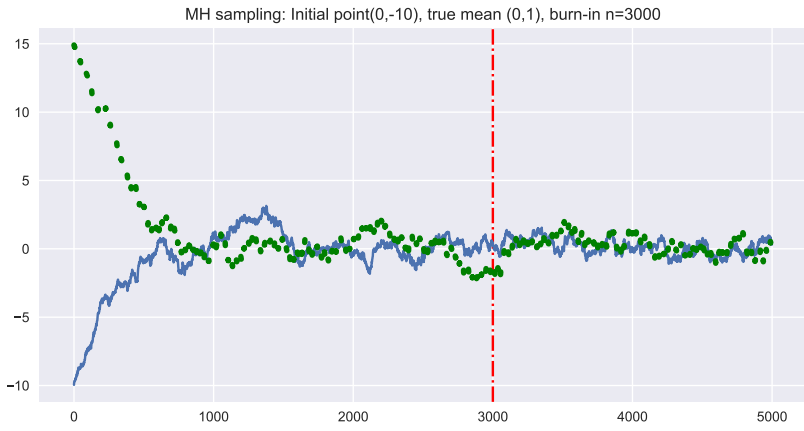


Figure 3.5: Two chains of an MH simulation which differ only in their starting points. The figure shows that the first 3000 samples can be discarded; marking the start of the burn-in period.

3.3.5 Limitations of the MH Algorithm

The MH algorithm as described is a random-walk MH algorithm, and as illustrated by Fig. 3.4, its convergence is subject to an *effective step size*.

Introduction of correlations to successive samples by the Markovian steps proved successful in overcoming the curse of dimensionality and in advancing convergence to the target distribution, see Robert (2014); Roberts and Rosenthal (2009b). However, MCMC methods face challenges when estimating a *step size* and choosing a *proposal distribution*.

Step size

Challenges when estimating a *step size*, σ; too large a step and the algorithm can potentially disregard significant distribution information, whereas too small a step can lead the search into a *local trap*.

Proposal distribution

Another crucial aspect of the MH algorithm is *choosing a proposal distribution*; a bad choice results in poor performance of the Monte Carlo estimates (Andrieu and Thoms, 2008b) because of poor convergence (Haario et al., 2001b).

3.3.6 Flexible Proposal Distributions

Finding an optimal step size and improving state space search, can be approached by tuning parameters of the MH algorithm. Tuning the *step size* of the proposal distribution, can be achieved in either one of two ways: first, *manual* tunning. But this is highly subjective, requires tedious work, and is time consuming.

Alternatively, parameters can be tuned automatically by what is termed *Adaptive MCMC*s. This is a faster and flexible alternative to manual tuning. It offers continuous improvement in exploring the search space. For details, see Chapter 4.

3.3.7 Convergence

The problem of deciding when to stop a simulation is of strategic importance for MCMC methods. Ideally, the simulation should be stopped only when convergence has been achieved. Yet, this is very difficult to determine quantitatively.

In most cases, the convergence rate of a Markov chain cannot be determined theoretically. An important point of discussion in the MCMC community is whether it is possible at all to empirically determine at which point a chain converges.

3.4 MCMC Diagnostic Tools

Estimating convergence, hence, deciding when to terminate the sampling process, is a question that is addressed in this section.

For multi-modal target distributions, a diagnostic tool must detect convergence over all regions of the target distribution. For instance, if all the starting values are located around one mode, it might become subject to the local trap. A visual aid is of great use when dealing with multi-modal distributions.

Trace plots can be used to detect incorrect convergence, e.g., to a false mode. It can also detect the lack of convergence due to inadequate simulation time or an underestimated burn-in period.

Three popular diagnostic methods are used in this thesis. These are namely: Geweke diagnostic, Effective Sample Size (ESS) and the Autocorrelation Factor (ACF).

3.4.1 Geweke Diagnostic

The Geweke diagnostic tool is a time series approach that tests for the ergodicity of an MCMC chain. The Geweke diagnostic tool takes the first 10% of the chain after the burn-in period, and the last 50%. Then, it performs a z-test for equality in the mean. If the two means are equal, the chain is ergodic, otherwise it has not converged.

Let the two means respectively be $\bar{\mathbf{x}}_a$ and $\bar{\mathbf{x}}_b$, the diagnostic is computed as:

$$Geweke = \frac{\bar{\mathbf{x}}_a - \bar{\mathbf{x}}_b}{\sqrt{\mathbf{s}_a + \mathbf{s}_b}}$$

where the denominator is the standard error of the differences, and where the test is performed on a single chain.

3.4.2 The Autocorrelation Function

The Autocorrelation function (ACF) is a test that measures the efficiency of a sampler by checking the randomness of its samples. The ACF relates to the different dimensions (usually the first two) of a random variable at different generations of a single run.

Assuming $\mathbf{x}_N$ are equally spaced, $\mathbf{v}_0$ is the initial lag and defined as:

$$\mathbf{v}_0 = \frac{1}{N} \sum_{n=1}^{N} \mathbf{x}_n^2 - \bar{\mathbf{x}}_n^2$$

where $\mathbf{v}_0$ is the variance.

The j^{th} lag, is the autocovariance measure of interdependence and defined as:

$$\mathbf{V}_j = \frac{1}{N-j} \sum_{i=1}^{N-j} \bar{\mathbf{s}}_n \bar{\mathbf{s}}_{n-j} - \boldsymbol{\mu}_n \boldsymbol{\mu}_{n-j}$$

The normalized auto-covariance is the ACF $\mathbf{r}_i$, which takes the form

$$\mathbf{r}_i = \frac{\mathbf{V}_j}{\mathbf{V}_0} \tag{3.5}$$

3.4.3 ESS

ESS is a lower-bound on the number of samples before estimates can be considered $i.i.d.$ (MacKay, 2003). These methods explore an unknown probability distribution by means of a random walk, or diffusion process, called Random walk MH (Liang et al., 2011). Once an ESS is selected, all samples thereafter are considered to be from the invariant distribution.

ESS is a unit of measure where MCMC sampler is compared to an $i.i.d$ sampler(Brooks et al., 2011). It is given by:

$$N_{ESS} = \frac{N}{1 + 2\sum_{k=1}^{\infty} \rho(k)}$$

where N is the number of samples and $\rho(k)$ is the correlation at iteration k.

If the samples are independent, the ESS equals the actual sample size; and if the correlation at k decreases slowly the ESS is zero. The infinite sum is truncated at lag k when $\rho(k) < 0.05$

3.5 Chapter Conclusion

This chapter has explained the theoretical underpinnings of MCMC which enables this class of algorithms to overcome the curse of dimensionality and be able to sample from any distribution.

The Metropolis-Hastings algorithm was explained as it is the basis for many state of the art MCMC algorithms. It showed to be effective in generating samples from regions of high probability from the target distribution. And it also showed that it is susceptible to the choice of proposal distribution parameters.

The tools used to estimate convergence and decide on the termination criteria, were explained. These tools are used to help overcome limitations in MCMC methods and realize more efficient sampling.

4 | Adaptive MCMC

Choosing a suitable proposal distribution for MCMC algorithms is a critical factor for convergence (Haario et al., 2001b). Previous research has mitigated this challenge by using methods which tune the covariance matrix and the global scale of a proposal distribution. These techniques are collectively referred to as adaptive MCMC.

This chapter will motivate the use of adaptive MCMC methods instead of the traditional non-adaptive sampling methods. Then, it will present sate-of-the-art adaptive MCMC algorithms, with flexible proposal distributions. Specifically, the Adaptive Metropolis (AM) algorithm will be described.

4.1 Motivation for Adaptive MCMC

If the proposed values are strongly correlated, it indicates that the chain only explored a small region of the distribution space. In this case, the chain exhibits poor mixing, and the algorithm takes long to achieve convergence. Also, a distribution exhibiting good mixing has a low acceptance rate, and vise versa.

To overcome correlation, methods were developed that adapt parameters of the proposal distribution as the sampling progressed. These approaches were designed with the objective of dealing with the trade-off between mixing and acceptance ratio. Accordingly, an effective method is one that yields an optimal acceptance ratio as well as the best mixing rate. The techniques became known as MCMCs.

4.2 Adaptive MCMC Algorithms

Tuning the proposal distribution markedly increases efficiency in converging to the target distribution, see (Muller, 2010; Müller and Sbalzarini, 2010; Liang et al., 2010; Haario et al., 1999).

Haario et al. (1999, 2001a) in their pioneering work on adaptive methods proposed two ways for updating the first two moments of a proposal distribution. Updates are done either using a fixed number of previous values of the Markov chain, or using all previous values of the chain (Haario et al., 1999), up to that point. These are the Adaptive Proposal (AP) and the AM, respectively. A positive definite[1] covariance matrix, c_0, is chosen as the initial covariance; the one to be iteratively updated.

The ergodicity condition must be satisfied to ensure that MH-based Markov chains converge to the invariant distribution π (Haario et al., 2006). Ergodicity requires satisfying two conditions (Roberts and Rosenthal, 2007):

- *Bounded convergence*; events in the search space occur with probability $p > 0$, and;
- *Diminishing Adaptation*: the transition matrix T becomes weakly conditioned on, or independent from, samples $\mathbf{x}$ as $N \to \infty$, where N is the number of iterations. Diminishing adaptation is a necessary condition for generating samples which are distributed according to the target distribution.

> **Remark 4.1: Convergence of Adaptive MCMC Algorithms**
>
> Adaptive MCMC algorithms that satisfy both bounded convergence and diminishing adaptation conditions are ergodic with respect to the invariant distribution, Π.

The underlying differences between adaptive methods is in the way they update their parameters. The next section explains the pivotal algorithm, the AM algorithm.

4.3 Adaptive Metropolis

The AM algorithm (Haario et al., 2001b) automatically tunes parameters of the proposal distribution during sampling. Specifically, the covariance matrix, the global scale and (sometimes) the mean vector of the Gaussian proposal distribution, are automatically tuned. Updates in the AM algorithm are according to *all* previously generated and accepted samples.

[1] A positive definite matrix is a symmetric matrix whose eigenvalues are all positive.

Algorithm 2: Adaptive Metropolis Algorithm

Input: Initial $\mathbf{x}_0, \mathbf{m}_0, c_0$, and $\boldsymbol{\sigma}_d$, β

Output: Unbiased sample from $\pi(\mathbf{x})$

for $n = 1, 2, ..., N$ **do**

  Sample $\mathbf{x}^* \sim \mathcal{N}(\mathbf{x}_n, \boldsymbol{\sigma}_{opt} c_n)$

  Apply MH criteria $\alpha(\mathbf{x}^*, \mathbf{x}_n) = \min\left\{1, \frac{\pi(\mathbf{x}^*) q(\mathbf{x}_n|\mathbf{x}^*)}{\pi(\mathbf{x}_n) q(\mathbf{x}^*|\mathbf{x}_n)}\right\}$

  if $\alpha(\mathbf{x}^*, \mathbf{x}_n)$ **then**

    $\mathbf{x}_{n+1} = \mathbf{x}^*$

    update the covariance matrix

    $c_{n+1} = c_n + \frac{1}{n+1}\left((\mathbf{x}_{n+1} - \mathbf{m}_n)(\mathbf{x}_{n+1} - \mathbf{m}_n)^\top - c_n\right)$

  else

    $\mathbf{x}_{n+1} = \mathbf{x}_n$

where $\mathbf{x}^*$ denotes the proposed sample; c_0 is the initial covariance, usually the identity matrix; m_0 is the mean and usually set at $m_0 = 0$.

Remark 4.2: An Updated AM algorithm

An amended version of the AM sampler of Haario et al. (2001a) was provided by Roberts and Rosenthal (2009a) for generating new samples. It begins with a d-dimensional target distribution $\pi(\mathbf{x})$. Then the proposed sample at iteration n is given by:

$$\mathbf{x}_{n+1} = \begin{cases} N(\mathbf{x}, \boldsymbol{\sigma}_{init} I_d) & n \leq 2d \\ (1-\beta) N(\mathbf{x}, \boldsymbol{\sigma}_{opt} c_n) + \beta N(\mathbf{x}, \boldsymbol{\sigma}_{init} I_d) & n > 2d \end{cases}$$

where: c_n is the current empirical estimate of the covariance matrix; $\boldsymbol{\sigma}_{init} = \frac{0.1^2}{d}$ and $\boldsymbol{\sigma}_{opt} = \frac{2.38^2}{d}$ are the initial and optimal scale, respectively(Gelman et al., 1996); and usually $\beta = 0.05$.

The rationale for using β is to avoid getting stuck with a singular covariance matrix. Another approach that can be used to avoid the singular matrix is regularization of the covariance matrix using its diagonal elements (Haario et al., 2001a).

The covariance matrix c_n is updated at each run of the algorithm by:

$$c_{n+1} = c_n + \frac{1}{n+1}\left(\left(\mathbf{x}_{n+1} - \mathbf{m}_n\right)\left(\mathbf{x}_{n+1} - \mathbf{m}_n\right)^\top - c_n\right)$$

where: $\mathbf{x}_{n+1}$ is the sample generated at step $n+1$; and $(n+1)^{-1}$ is often represented by γ, whose role is in diminishing adaptation and necessary for AM ergodicity.

4.4 A Practical Issue in Adaptive MCMC

Symmetry conditions cannot be satisfied in AM and therefore the chain is not Markovian. Nonetheless, Haario et al. (2001b) had established that the AM had the required ergodic properties, compared well with traditional MH algorithms, and was relatively more applicable.

Haario et al. (2001a) proved that ergodicity in adaptive MCMC can be achieved by ensuring *diminishing adaptation*. This means that adaptation can only be met if dependency on the chain history diminishes when simulations progress, i.e., as $N \to \infty$, see Section 5.3.

Remark 4.3: Adapting the Search Space

The covariance of any distribution determines the orientation of the distribution as well as the spread of the samples. The eigenvalues and eigenvectors determine search orientation.

An *isotropic* or spherical distribution $\mathcal{N}(0, I_d)$ is a standard normal distribution. It is invariant to rotation about the mean.

An axis-parallel covariance matrix $\mathcal{N}(0, c)$ is a diagonal matrix whose distribution follows $\mathcal{N}(0, diag(\sigma^2))$ where σ is a vector of the standard deviation. The principal axes of the ellipsoid are parallel to the diagonal matrix (Hansen et al., 2015). Illustration of rotations is shown in Figure 4.1.

4.5 State of the Art of Adaptive MCMC

Muller and Sbalzarini (2010) presented the 'unifying framework' for continuous optimization and sampling, based on GaA, see Section 5.2. They modified *GaA* into a Random Walk Monte Carlo (RWMC) sampler, named: *M-GaA*. They had achieved this by introducing the metropolis acceptance criterion into the optimization algorithm, thereby, converting it into a sampler.

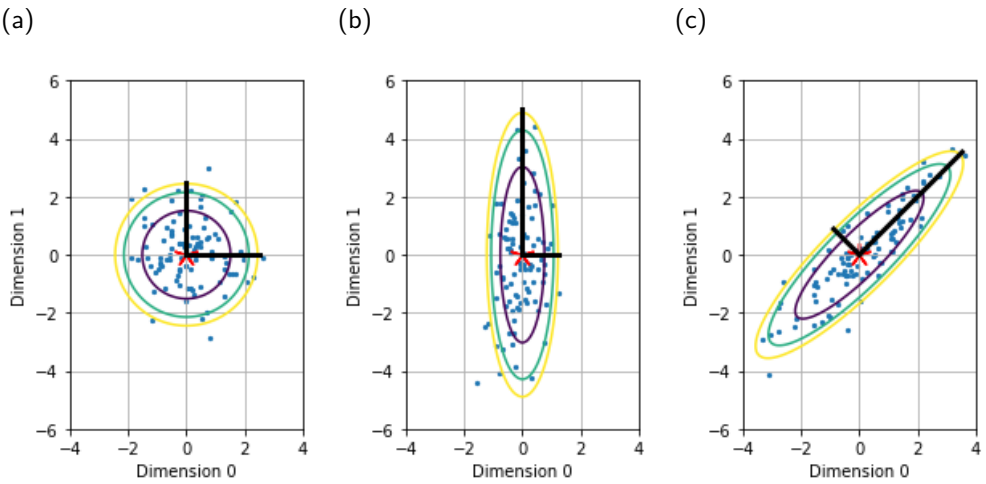


Figure 4.1: The three figures are 2-dimensional multivariate normal distributions $\mathcal{N}(0, c)$. From left to right: (a) an isotropic distribution I_d (identity matrix); (b) diagonal matrix c (an axis-parallel distribution) with eigenvalues (0.25, 4); (c) diagonal matrix ((2.125, 1.875)(1.875, 2.125)) with eigenvalues $(0.25, 4)$. The eigenvalues are shown in a black line and the centroid (mean) in a red star. Three contour lines representing the $1-\sigma$, $2-\sigma$ and 3σ for the distribution are shown

Remark 4.4: M-GaA

Just as in $(1+1)$*CMA-ES*, GaA is a one-parent one-offspring optimization algorithm that continuously adapts the mean and covariance matrix. And the M-GaA is a sampler that employs an adaptive proposal.

Glasmachers et al. (2010) discussed the *Natural Gradient (NG)* of the expected fitness, key feature of the *Natural Evolution Strategies (NES)*. NES update the proposal distribution in the direction of regions of higher probability density while taking into consideration convergence, robustness, and sample complexity, among other features. They regard the NG as a 'principled' approach towards solving real-valued or continuous Evolution Optimization Problems (EOPs).

Glasmachers et al. (2010) used Cauchy distribution as their proposal distribution. It is known to work well for heavy-tailed and multi-modal target distributions.

Wierstra et al. (2014) further analyzed NES by using both multivariate Gaussian and Cauchy distributions as proposals. They specifically chose a Gaussian multivariate distribution because it allows the Fisher information matrix, needed for the natural gradient, to be computed analytically.

Muller and Sbalzarini (2010) compared performance of GaA with MH and AM algorithms. They used the uncorrelated Gaussian (π_1) , the moderately non-linear twisted Gaussian (π_3) with $b = 0.03$, and the strongly non-linear twisted Rosenbrock function (or Rosenbrock's banana function) Gaussian $(\ \pi_4)$ with $b = 0.1$, which are similar to those used by Haario et al. (1999).

4.6 Sampler Adaptiveness

Roberts and Rosenthal (2009a) considered the Adaptive Metropolis (AM) and calculated the *sub-optimality factor* b, or *homogeneity* factor, for $b > 1$. They proposed that it is optimal to take

$$c_{prop} = c_{target}$$

where c_{prop} is the proposal distribution covariance matrix, and c_{target} is the target distribution covariance matrix (Roberts and Rosenthal, 2009a). Further, Roberts et al. suggested that for any other c_{prop}, mixing will be relative to a sub-optimality factor $b > 1$.

Finding b is first done by calculating C_b, where:

$$c_b = c_{prop}^{1/2}\ c_{target}^{-1/2} \tag{4.1}$$

When its eigenvalues are found, the the sub-optimality factor b is computed by:

$$b = d \ \frac{\sum_{i=1}^{d} \lambda_i^{-2}}{(\sum_{i=1}^{d} \lambda_i^{-1})^2} \tag{4.2}$$

where d is the dimension and λ_i , where $i = 1, ..., d$, are the d eigenvalues of the matrix c_b. As $b \to 1$, the algorithm becomes more efficient.

Experiments using the measure b led to the conclusion that the AM algorithms actually learn the covariance of the target distribution, and sample very efficiently from the target distribution π. Further, Roberts et al. used Haario's *twisted distributions* as targets, performed experiments and concluded that adaptation improved mixing up to 5000000 iterations.

4.7 Chapter Conclusion

This chapter explained the motivation behind the emergence of adaptive MCMC algorithms. It laid out the foundation of state-of-the-art techniques in mitigating limitations in MCMC brought by fixed proposal distributions. Specifically, it explained the theoretical underpinning of how adapting proposal distributions of the MH algorithm mitigates the limitations when performing MCMC sampling. As a result, both the optimal step-size and proposal distributions are no longer fixed, but vary by learning from accepted samples.

Finally, it concluded by describing a test for the adaptiveness of sampling techniques. The next Chapter will explain state of the art in stochastic optimization methods, an important technique for black-box optimization.

5 | Stochastic Optimization

This chapter explains two stochastic optimization algorithms used in machine learning, namely GaA and CMA-ES. First, it explains their parameters settings and their updates. It then discusses a more efficient covariance matrix update. The chapter then concludes by transforming two stochastic optimization techniques into samplers.

5.1 Stochastic Optimization

Stochastic optimization refers to a group of methods for finding an optimal solution[1] to an objective[2] function: $f(\mathbf{x}) : R^d \rightarrow R$. These methods introduce randomness to an objective function and its constraints. Randomness can be inherent in the system as *noise* or introduced into the optimization process through *Monte Carlo* sampling.

> **Remark 5.1: Optimization Problem**

[1] Minimum or maximum numerical solutions.

[2] Note: In the context of optimization, various terminologies will change from the usual sampling terminologies. e.g target distribution is referred to as the *objective function* in optimization; MH acceptance function is used in sampling while in optimization a threshold function is used.

Numerical optimization problems *minimize* a convex function $f(\mathbf{x})$ *with convex set constraint* $\mathbf{x} \in C$. They take the general form:

$$\mathbf{z} = \sum_{i=0}^{N} c_i \mathbf{x}_i$$

where: $z \in \mathcal{R}$, is the optimal numerical value; and c_i is the coefficient corresponding to value x_i.

In order to optimize an objective function, three of its parameters must iteratively be updated. These parameters are the *mean* $(\mathbf{m})$, and determines the center of the distribution; *covariance matrix* (C), in a multivariate distribution, and relates to the shape of states in the search distribution; and *step-size* (σ), determines the hops in the search space.

Remark 5.2: Stochastic Optimization Process Update

A stochastic optimization process update typically consists of three steps, namely:

- Generating a new sample
  e.g $\mathbf{x}_{n+1} = \mathbf{m} + \boldsymbol{\sigma}\mathcal{N}(0, c), \qquad \mathbf{x}_{n+1} \sim \mathcal{N}(\mathbf{m}_n, c_n)$
- Threshold success based on function fitness.
  e.g: accept, if: $\mathbf{x}_{n+1} \leq c_{T_n}$
- Update the **mean** and **covariance matrix**, see Remark 5.3 and 5.4

where: c is a positive definite covariance matrix, and;
$\mathbf{m}$ is mean of the search distribution of the objective function.

Remark 5.3: Mean Update

The general adaptation of the mean is given by:

$$\mathbf{m}_{n+1} = \mathbf{m}_n + \alpha \mathbf{x}_{n+1}, \qquad \alpha \in (0, 1] \tag{5.1}$$

Remark 5.4: Covariance Update

A typical update for the covariance matrix is given by:

$$c_{new} \triangleq \alpha c_{old} + \beta \mathbf{x}\mathbf{x}^T$$

and the general adaptation of the covariance matrix follows the format:

$$c_{n+1} = \alpha c_n + \beta \mathbf{x}_{n+1}\mathbf{x}_{n+1}^T \qquad \beta \in (0, 1] \tag{5.2}$$

Remark 5.5: Parent and Offspring

Parent, $\mathbf{x}_{parnet} = \mathbf{x}_n$, refers to current values, i.e., vectors at time n. Offspring, $\mathbf{x}_{offspring} = \mathbf{x}_{n+1}$, refers to values at time step $n + 1$, where:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \sigma_n \, \mathcal{N}(0, C)$$

5.2 Gaussian Adaptation for Stochastic Optimization

GaA is a maximum-entropy stochastic optimization method for discrete and continuous problems. It was first proposed in Kjellstrom and Taxen (1981) and later extended by Muller (Muller and Sbalzarini, 2010; Müller and Sbalzarini, 2011).

GaA samples one offspring from a multivariate normal distribution, and continuously adapts the first and second moments to optimize the objective function. The method either accepts or rejects samples based on a threshold, see Algorithm (3).

GaA is a search heuristic, and accordingly, accepted samples are used to align the optimization process to the orientation and *shape* of the objective function. Adaptation of these moments maximizes the entropy of search distributions under the constraint that the best points are found within a predefined *hitting probability* (Muller, 2010).

In order to minimize a real valued objective function $f(\mathbf{x})$, GaA uses a threshold value c_T, that is lowered until convergence(Muller and Sbalzarini, 2010).

Control parameters: GaA control parameters are:

Algorithm 3: Gaussian Adaptation for Optimization

Input: f, $\mathbf{x}_{init}$, $\mathbf{m}_0$, c_0, σ_0, c_T, f_e , f_c , $MaxIter$
w_T, w_m,w_c
Output: $\mathbf{x}_{best}$

while *termination criteria not met,* **do**
  generate $\mathbf{x}_{n+1} \triangleq \mathbf{m}_n + \sigma_n \mathcal{N}(0, c)$
  if $f(\mathbf{x}_{n+1}) < c_{T_n}$ **then**
    set $\mathbf{x}_{best} = \mathbf{x}_{n+1}$
    expand $\sigma_{n+1} = \sigma_n.f_e$
    update the parameters
      $\mathbf{m}_{n+1} \triangleq \mathbf{m}_n(1 - w_m) + w_m(\mathbf{x}_{n+1} - \mathbf{x}_n)$
      $c_{n+1} \triangleq c_n(1 - w_c) + (1 - w_c)((\mathbf{x}_{n+1} - \mathbf{x}_n)(\mathbf{x}_{n+1} - \mathbf{x}_n)^T)$
      $c_{T_{n+1}} \triangleq (1 - w_T)c_{T_n} + f(\mathbf{x}_{n+1})w_T$
  else
    contract $\sigma_{n+1} = \sigma_n.f_c$

- σ: step-size. It expands or contracts contour lines of the covariance matrix, i.e., its search space. By increasing σ, the search space is expanded based on a predetermined hitting probability P. When σ is small, the contour lines close-in on the centroid, and sampling from around the centroid is with higher probability;
- P: *hitting probability*, a predetermined parameters which controls the frequency of successes;
- E: Efficiency of the stochastic process, where: $E \propto -PlnP$. As a result, $P \simeq 0.37$(Kjellstrom and Taxen, 1981; Muller and Sbalzarini, 2010);
- w_m, w_c: weights in the adaptation process that control contribution from the previous mean and covariance, respectively. These are expansion parameters when $w_m, w_c > 1$ and contraction facts when $w_m, w_c < 1$. In either extremes, setting $w_m, w_c = 0$, expresses no change to the orientation or shape of the covariance matrix; while $w_m, w_c = 1$ expresses that the mean and covariance move precisely towards the accepted sample;

  The two parameter values are given as: $w_c = \frac{ln(d+1)}{(d+1)^2}$, and $w_m = e.d$ as specified in Muller and Sbalzarini (2010); where e is Euler's constant ($e = 2.71828$), d is the dimension;
- w_T: Another weighing parameter, it controls weights between the old threshold and the accepted value;

- c_{T_0}: Initial threshold value where: $c_{T_0} = f(\mathbf{m}_0)$(Muller and Sbalzarini, 2010);
- f_c, f_e: Contraction and expansion parameters, respectively. f_c shrinks the volume by a parameter f_c^{2d}, while f_e expands the volume by f_e^{2d}. Muller and Sbalzarini (2010) show that the best vales for both parameters are: $f_c = 1 - \beta P$ and $f_e = 1 + \beta(1 - P)$, where β is a small constant usually given as $\beta = w_c$, and;
- $MaxIter$: The number of algorithm iterations depend on objective function domain dimensionality, and are limited to $MaxIter = 10e^{-4}d$

Adapting the Step-Size: The *step-size*, σ, is expanded when the offspring is accepted, and contracted otherwise. Accordingly, $\sigma_{n+1} = f_e\sigma_n$ when the offspring is accepted, and $\sigma_{n+1} = f_c\sigma_n$ if rejected.

Adapting the Mean: The *mean* is updated based on $w_m < 1$. This determines the contribution of the offspring on the new mean. When $w_m = 1$, the new mean is centered on parent. Update to the mean is given by:

$$\mathbf{m}_{n+1} \triangleq (1 - w_m)\mathbf{m}_n + w_m\mathbf{x}_{n+1} \tag{5.3}$$

Adapting the Covariance Matrix: Δc_{n+1} incorporates the contribution from accepted offspring, with w_c as the weight.

$$\Delta c_{n+1} \triangleq (1 - w_c)I_d + w_c\mathbf{z}_n\mathbf{z}_n^\top \tag{5.4}$$

where I_d is the identity matrix and $\mathbf{z}_n$ is the nth sample of the multivariate standard normal distribution $\mathcal{N}(0, I_d)$. Therefore, the updated covariance is:

$$c_{n+1} \triangleq c_{n+1}c_{n+1}^T \tag{5.5}$$

As in MCMC sampling, the covariance matrix, c, is updated after a burn in period, see Figure 3.2.

Adapting the Threshold: In each iteration, the threshold value $c_{T_{n+1}}$ is updated using the objective function value of the accepted new value. This means that the threshold is accepted if and only if the search is in the direction of the objective function value. The initial search threshold c_{T_0} is usually domain dependent.

5.3 Evolution Strategy for Optimization

Evolution Strategy (hereinafter: ES) are optimization methods developed to solve real-valued nonlinear optimization problems (Igel et al., 2006b; Bäck et al., 2013). They

are applied to problems for which other techniques, such as Gradient Descent, cannot describe or solve them analytically (Bäck, 1996).

ES algorithms generate offspring vectors $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_\lambda$, where $\lambda \in N$, from parent vectors. Offspring objective function values are then evaluated to identify which best fit the model. Accordingly, $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_\mu$, where $\mu \in N$, offspring are selected as the next generation of parent vectors. Various models exist for how offspring are generated then recombined, yet they generally follow:

Let $i \in \{1, \lambda\}$, where $\mathbf{x}_i^{(g)}$ and $\mathbf{v}_i^{(g)}$ be the i^{th} offspring and multivariate random variable, respectively, at generation g, then:

$$\mathbf{x}_i^{(g+1)} \leftarrow \mathbf{x}_i^{(g)} + \mathbf{v}_i^{(g)} \tag{5.6}$$

where: $\mathbf{x}_i^{(g)}$ is offspring recombination, and; $\mathbf{v}_i^{(g)}$ is the mutation.

ES methods share the following properties(Bäck et al., 2013):

- selection is unbiased;
- selection is deterministic, and;
- mutations adapt.

The following is a brief review of each of the above ES components:

- **Selection**: According to Beyer and Schwefel (2002), this component deterministically orientates the search space in a more promising direction by pursuing offspring with best-fit values[3]. At this stage, offspring are ranked by their corresponding objective function values, in increasing order. On one side, values that minimize the objective function, and on the other, values that generate the highest errors.
  There are two forms for generating offspring in ES: *elitist* and *non-elitist* selection. Elitist selection combines parent values with the offspring, ranks them, then selects the best μ to be the next parents. Elitist selection is usually expressed as $(\mu + \lambda)$, parents and offspring, respectively. In non-elitist selection, parents are selected from the current λ offspring only. This is expressed in the form $(\mu/\mu, \lambda)$.
- **Mutation**: According to (Bäck and Schwefel, 1993), mutations perturb parent values within $\pm\delta$ from the mean. This component is the primary source of variation, i.e., the stochastic component, across consecutive generations.
  Three algorithm design principles, obtained from experiment and theoretical considerations, have been put forth(Beyer and Schwefel, 2002):
  - *reachability*: any point is reachable within a finite number of generations;

[3] As evaluated by the objective function.

  - *unbiasedness*: the algorithm uses information obtained from the parent population to freely explore the search space. Doing so without preconditions ends up being synonymous to the *maximum entropy principle* (Beyer and Schwefel, 2002) which is a quality of the *Multivariate Gaussian Distribution*(Bäck et al., 2013).
  - *scalability*: is the ability of the algorithm to *self-adapt*, depending on the mutation strength (Beyer and Schwefel, 2002), in order to explore properties of the search space.
- **Recombination**: Applicable for $\lambda > 1$. Recombination combines information from all, or a selected number of, parents to generate offspring. The aim of the recombination step is to select a new mean value.

ES algorithms vary according to the chosen parameters. There is not a one-fits-all ES algorithm, and the choice will depend on the type of optimization problem.

5.4 Covariance Matrix Adaptation Evolution Strategy

CMA-ES is a powerful ES method for continuous black-box optimization (Hansen, 2010). It ensures that offspring with a better fitness are sampled more often. Its strength lies in the fact that it adapts to the search space through mutation (and recombination when $\lambda > 1$). Isotropic mutations are achieved by adding random vectors sampled from the standard normal distribution $\mathcal{N}(0, \mathbf{I})$ to current values (Igel et al., 2006a).

Cumulative step-size: The step-size σ is updated separately from the covariance matrix to keep track of the Cumulative Step-Size (CSS) of the sampling process. And it is independent from the shape of the distribution (Hansen and Ostermeier, 2001).

> **Remark 5.6: CMA-ES Adaptation Forms**
>
> CMA-ES implements two forms for adaptation, to improve the search performance in complex objective functions: *mutation strength*, and *mutation distribution shape*. This is achieved in three ways (Suttorp et al., 2009a):
>
> - Increasing the likelihood of generating successful steps;
> - Generating future steps which are less correlated, and;
> - Sustaining the invariance properties of the covariance matrix.

Adapting mutation strength of the distribution makes steps towards higher probability regions more feasible. Hansen and Ostermeier (2001) proposed using weights to allow variation in the offspring.

Evolution Path The evolution path for the step-size and the evolution path for the covariance matrix p_c, of recently visited steps, are recorded in the ESS p_σ. CMA-ES maximizes the likelihood of an *evolution path* considered to be successful.

Hansen and Ostermeier (1996) define the *evolution path* as the path the population takes over time, and suggest using it instead of considering *single* mutation steps, see Figure 5.1.

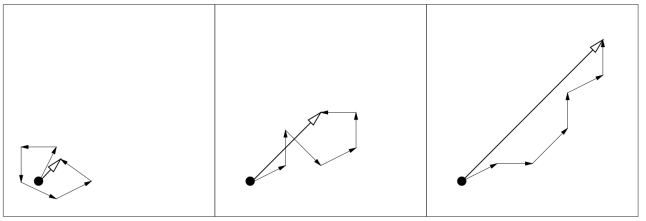


Figure 5.1: Evolution paths of six successive steps with comparable step-sizes, yet vary in the total path traversed. The left figure represents highly correlated steps, the middle figure represents less correlation, and the right figure shows little correlation.

Diminishing Adaption. Variations introduced by adaptation of the mean and covariance should ideally decrease as sampling proceeds. This is commonly referred to as *diminishing adaptation*. Andrieu and Thoms (2008a) describe a parameter denoted λ_n that ensures generating unbiased samples from the objective function. They further suggest a scale parameter r to regulate the acceptance rate. Gelman's optimal rate $r = 0.234$ is the preferred rate.

For the purpose of this thesis, only the $(1 + 1)$ CMA-ES is discussed in detail.

5.5 (1+1) CMA-ES

The foundation of Evolution Strategy (ES) algorithms, and its simplest, is the $(1 + 1)$ CMA-ES (Bäck et al., 2013), where there is one parent and one offspring (Igel et al., 2006b; Schwefel, 1995).

5.5.1 Control Parameters for (1+1) CMA-ES

CMA-ES has a number of strategy parameters that do not change during sampling. Default values are used as recommended in Igel et al. (2006a). These include the damping

Algorithm 4: $(1+1)$-CMA-ES Optimization Algorithm

Input: f, $\mathbf{x}_0$, σ_0, c_0, $\bar{p}_s = p_s^{target}$, p_s^{succ}, $\mathbf{p}_c$

for $n = 1, 2, \cdots, N$ **do**
- generate a offspring $\mathbf{x}_{offspring}$
- update σ_{n+1}
- **if** $f(\mathbf{x}_{offspring}) \leq f(\mathbf{x}_{parent})$ **then**
  - set the new parent state $\mathbf{x}_{parent} = \mathbf{x}_{offspring}$
  - update evolution path $\mathbf{p}_c$ and Covariance matrix c_{n+1}

parameter k; the target success rate p_s^{succ}; the learning parameter c_p, and the average success rate $\bar{p}_s \in [0, 1]$.

Other strategic parameters are success rate averaging parameter c_p, the covariance matrix weight w_c, the evolution path learning rate c_c and the threshold θ_p.

The average success rate $\bar{p}_s = p_s^{succ}$, where p_s^{succ} is the target success rate, p_t. The choice of the initial offspring $\mathbf{x}_0$ and the initial σ are problem dependent. The optimum initial parent vector should be: $x_{n+1} = 2\sigma(\mathbf{I})$

σ control:		
$k = 1 + \frac{d}{2}$,	$p_s^{succ} = \frac{2}{11}$,	$c_p = \frac{1}{12}$
C adaptation:		
$c_c = \frac{2}{d+2}$,	$w_c = \frac{2}{d^2+6}$	$\theta_p = 0.44$

Table 5.1: Default parameters for the $(1+1)$CMA-ES algorithm, as given by Igel et al. (2006a).

5.5.2 Parameter Update

At each iteration σ and the average success rate $\bar{p}_{succ}$ are updated depending on the accepted offspring. The update is based on the $1/5^{th}$ success rule.

> **Remark 5.7:** $1/5^{th}$ **Success Rule**
>
> If offspring have higher fitness than individuals (parents and their ancestors) of the previous five generations, they are considered successful and σ should be *increased*. The offspring, in this case, is oriented in the right direction. Otherwise, they are unsuccessful and σ is *reduced*.

There are two steps for updating parameters: first, the average success rate is updated based on success of current offspring and the past four individuals:

$$\bar{p}_{succ} = (1 - c_p)\bar{p}_{succ} + c_p\lambda_{succ} \tag{5.7}$$

where: $c_p \in [0, 1]$ is a learning rate, and $\bar{p}_{succ} \in [0, 1]$ is the average success rate over the last five generations, and

$$\lambda_{succ} \leftarrow \begin{cases} 1 & f(\mathbf{x}_{offspring}) \leq f(\mathbf{x}_{parent}) \\ 0 & otherwise \end{cases}$$

Then the update for σ is computed as:

$$\sigma_{n+1} = \sigma_n \; exp\left(\frac{1}{k}\left(\bar{p}_{succ} - \frac{p_s^{target}}{1 - p_s^{target}}(1 - \bar{p}_{succ})\right)\right) \tag{5.8}$$

where: k is the the damping parameter for updates of σ.

5.5.3 Covariance Matrix Update

The evolution point $\mathbf{x}_{n+1}$ and the covariance matrix c is updated only if $\bar{p}_s < \theta_p$. First, the evolution point $\mathbf{x}_n$ is updated, then by the update of the covariance matrix itself[4]. These updates depend on whether $\bar{p}_s < \theta_p$ or not.

The evolution point $\mathbf{x}_{n+1}$ is updated as

$$\begin{aligned} \mathbf{x}_{n+1} &= (1 - c_p)\,\mathbf{x}_n + \sqrt{c_p(2 - c_p)}\;\mathcal{N}(0, c) && \text{if } \bar{p_s} < \theta_p \\ &= (1 - c_p)\,\mathbf{x}_n && \text{otherwise} \end{aligned} \tag{5.9}$$

where c_p is the learning rate for the evolution path.

The covariance matrix c is then updated. This is applied if only the function of the new state $f(\mathbf{x}_{n+1}) \leq$ the threshold.

$$\begin{aligned} c_{n+1} &= (1 - w_c)\,c_n + w_c \mathbf{x}_{n+1}\mathbf{x}_{n+1}^\top && \text{if } \bar{p_s} < \theta_p \\ &= (1 - w_c)\,c_n + w_c\left(\mathbf{x}_{n+1}\mathbf{x}_{n+1}^\top + c_p(2 - c_p)c_n\right) && \text{otherwise} \end{aligned} \tag{5.10}$$

where w_c is the learning rate for the covariance matrix.

The identity matrix $\mathbf{I} \in \mathcal{R}^{d \times d}$ is used as the initial covariance matrix. This is to encode unbiasedness, i.e., no preference between either the positive or negative directions.

[4]Note that the covariance matrix used to generate the next offspring is $c = \sigma^2 c$

5.6 Cholesky Update

This section uses the Cholesky decomposition of the covariance matrix C, i.e., a positive definite matrix, to indirectly update it instead of a full matrix update. This method for updating C has been shown to be more computationally efficient. This method reduces the computational cost from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$, see (Krause and Igel, 2015).

Consider the covariance update equation:

$$C^{g+1} = \alpha C^g + \beta \mathbf{x}^g \mathbf{x}^{Tg} \tag{5.11}$$

The covariance matrix C can be decomposed into its Cholesky factors:

$$C = LL^T$$

Then the covariance matrix C is updated *indirectly* by updating its Cholesky factors, see (Krause and Igel, 2015):

$$L^{g+1} = \alpha L^g + \beta \mathbf{x}^g \mathbf{x}^{Tg} \tag{5.12}$$

Algorithm 5: $(1+1)$-rankOneUpdate(L, β, ν)

Input: Cholesky factor $L \in R^{d \times d}$ of C; $\alpha, \beta \in R, \nu \in R^d$
Output: Cholesky factor L' of C$+\beta\nu\nu^T$

$\omega \leftarrow \nu$
$b \leftarrow 1$
for *j = 1,...,d* **do**
  $L'_{jj} \leftarrow \sqrt{L_{jj}^2 + \frac{\beta}{b}\omega_j^2}$
  $\gamma \leftarrow L_{jj}^2 b + \beta\omega_j^2$
  for *k=j+1,...,d* **do**
    $\omega_k \leftarrow \omega_k - \frac{\omega_j}{L_{jj}} L_{kj}$
    $L'_{kj} = \frac{L'_{jj}}{L_{jj}} L_{kj} + \frac{L'_{jj}\beta\omega_j}{\gamma}\omega_k$
  $b \leftarrow b + \beta\frac{\omega_j^2}{L_{jj}^2}$

Lemma 5.1

Let C=$LL^T \in R^{d\times d}$ be symmetric positive definite and L a lower triangular matrix. Partition C and L into *blocks*:

C= $\begin{pmatrix} c_{11} & c_{21}^T \\ c_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 \\ \mathbf{l}_{21} & L_{22} \end{pmatrix} \begin{pmatrix} l_{11} & \mathbf{l}_{21}^T \\ 0 & L_{22}^T \end{pmatrix}$

with $c_{11}, l_{11} \in R, c_{21}, \mathbf{l}_{21} \in R^{d-1}$ and $C_{22}, L_{22} \in R^{d-1\times d-1}$

The objective is to compute an update of the lower triangular matrix L$\in R^{d\times d}$ to obtain the update for (C), the updates are L' and C', respectively. The update can be illustrated as follows:

$$C' = L'L'^T$$

The update for C' is given by: $C' = C + \beta\boldsymbol{\nu}\boldsymbol{\nu}^T \in R^{d\times d}$

with $\beta \in R$, and $\boldsymbol{\nu} = (\nu_1, \boldsymbol{\nu}_2) \in R^d$, where $\nu_1 \in R$, $\boldsymbol{\nu}_2 \in R^{d-1}$.

it follows that: $C' = \begin{pmatrix} l_{11} & 0 \\ \mathbf{l}_{21} & L_{22} \end{pmatrix} \begin{pmatrix} l_{11} & \mathbf{l}_{21}^T \\ 0 & L_{22}^T \end{pmatrix} + \beta(\nu_1, \boldsymbol{\nu}_2)(\nu_1, \boldsymbol{\nu}_2)^T$

the Cholesky matrix components are updated as follows:

$l'^{11} = \sqrt{l_{11}^2 + \beta\nu_1^2}$

$\mathbf{l}'_{21} = \frac{l_{11}\mathbf{l}_{21} + \beta\nu_1\boldsymbol{\nu}_2}{l'^2_{11}}$

$L'_{22}L'^T_{22} = L_{22}L_{22}^T + \beta\frac{l_{11}^2}{l'^2_{11}}\left(\boldsymbol{\nu}_2 - \frac{\nu_1}{l_{11}}\mathbf{l}_{21}\right)\left(\boldsymbol{\nu}_2 - \frac{\nu_1}{l_{11}}\mathbf{l}_{21}\right)^T$

For proof, see Proof 5.1.

Proof 5.1: Rank 1 Update

As given in(Suttorp et al., 2009b), $C' = L'L'^T$, then C' is expanded as follows:

$$C' = \begin{pmatrix} l'^2_{11} & l'_{11}\mathbf{l}'^T_{21} \\ l'_{11}\mathbf{l}'_{21} & L'_{22}L'^T_{22} + \mathbf{l}'_{21}\mathbf{l}'^T_{21} \end{pmatrix}$$

As a result, l'_{11} can be computed from c'_{11}, $\mathbf{l}'_{21}$ from C'_{21} and L'_2 from c'_{22}. It holds for l'_{11} that:

$$l'_{11} = \sqrt{c'_{11}} = \sqrt{c_{11} + \beta\nu_1^2} = \sqrt{l_{11} + \beta\nu_1^2}$$

and it holds for $\mathbf{l}'_{21}$ that:

$$\mathbf{l}'_{21} = \frac{\mathbf{C}'_{21}}{l'_{11}} = \frac{c_{21} + \beta\nu_1\boldsymbol{\nu}_2}{l'_{11}}$$

Applying the above formula to $L'_{22}L'^{T}_{22}$ gives:

$$\begin{aligned}
&L'_{22}L'^{T}_{22} = A'_{22} - \boldsymbol{l}'_{21}\boldsymbol{l}'^{T}_{21} \\
&= L_{22}L^{T}_{22} + \boldsymbol{l}_{21}\boldsymbol{l}^{T}_{21} + \beta\boldsymbol{\nu}_2\boldsymbol{\nu}^{T}_{2} \\
&- \frac{l_{11}\mathbf{l}_{21}+\beta\nu_1\boldsymbol{\nu}_2}{l'_{11}}\left(\frac{l_{11}\mathbf{l}_{21}+\beta\nu_1\boldsymbol{\nu}_2}{l'_{11}}\right)^{T} \\
&= L_{22}L^{T}_{22} + \beta\left(1 - \frac{\beta\nu_1^2}{l'^{2}_{11}}\right)\boldsymbol{\nu}_2\boldsymbol{\nu}^{T}_{2} + \left(1 - \frac{l^2_{11}}{l'^{2}_{11}}\right)\mathbf{l}_{21}\mathbf{l}^{T}_{21} \\
&- \frac{\beta\nu_1\mathbf{l}_{11}}{l'^{2}_{11}}\left(l_{21}\boldsymbol{\nu}^{T}_{2} + \boldsymbol{\nu}_2\mathbf{l}^{T}_{21}\right) \\
&= L_{22}L^{T}_{22} + \frac{\beta l^2_{11}}{l'^{2}_{11}}\boldsymbol{\nu}_2\boldsymbol{\nu}^{T}_{2} + \frac{\beta\nu_1^2}{l'^{2}_{11}}\mathbf{l}_{21}\mathbf{l}^{T}_{21} \\
&- \frac{\beta\nu_1 l_{11}}{l'^{2}_{11}}\left(\mathbf{l}_{21}\boldsymbol{\nu}^{T}_{2} + \boldsymbol{\nu}_2\mathbf{l}^{T}_{21}\right) \\
&= L_{22}L^{T}_{22} + \beta\frac{l^2_{11}}{l'^{2}_{11}}\left(\boldsymbol{\nu}_2 - \frac{\nu_1}{l_{11}}\mathbf{l}_{21}\right)\left(\boldsymbol{\nu}_2 - \frac{\nu_1}{l_{11}}\mathbf{l}_{21}\right)^{T}
\end{aligned}$$

□

5.7 Comparison of GaA and CMA-ES

Both GaA and CMA-ES are stochastic optimization methods. They are related in that they both use a threshold value for accepting an offspring.

The $(1+1)$CMA-ES is a close variant of GaA when $w_c \to \infty$ which maintains the covariance matrix as an isotropic matrix, with a fixed success rate (Müller and Sbalzarini, 2010).

However, major differences between the two methods are:

- $(1+1)$-CMA-ES uses a damped exponential function, allowing faster adaptation than in GaA (Muller and Sbalzarini, 2010);
- CMAES includes a predefined acceptance probability referred to as the target success rate p_s^{target} in $(1+1)$-CMA-ES (Igel et al., 2006a);
- CMA-ES updates the evolution path in addition to the covariance matrix, which is not the case in GaA. GaA adapts the mean vector based on the weight parameter λ_μ which determines the influence of the offspring on the position of the mean in the next iteration. If $\lambda_\mu = 1$ the new mean is centered at the parent. However, the mean in CMA-ES is always centered at the parent.

- CMA-ES is based on evolutionary strategies, hence implements additional procedures like selection, mutation and recombination.
- $(1+1)$CMA-ES (and (μ, λ)CMA-ES) parameterizes the covariance matrix based on its eigenvectors and eigenvalues.
- GaA and CMAES adapt the covariance matrix in different ways. GaA aims at increasing the entropy of the search distribution, whereas CMA-ES aims at increasing the likelihood of generating successful offspring.

5.8 Stochastic Sampling

This section turns the optimization tools discussed herein into *samplers*.

Key differences to note is that an optimization search point is controlled by consistently reducing the threshold value. This value is updated when search points have higher objective function values than previous points. This is iterated until a best solution or termination criteria is achieved.

Sampling, on the other hand, involves a sample point that is subject to a Metropolis acceptance criteria. If it is accepted, the sampler assigns it as the current value. Otherwise the old value remains the current value. This is repeated until convergence is attained.

5.8.1 GaA for Sampling

Metropolis Gaussian Adaptation (M-GaA) is obtained in two steps. First by replacing the threshold measure c_T in Algorithm 3, line 3, with the Metropolis acceptance criteria, see Algorithm 1.

Secondly the weighing factor for the mean w_m is set to 1 so that the mean is always centered at the accepted state $\mathbf{x}_n$ (Muller and Sbalzarini, 2010).

Other changes which apply to both GaA and $(1+1)$CMA-ES are setting the prior hitting probability or average success rate. See Algorithm 6.

5.8.2 $(1+1)$CMA-ES for Sampling

The $(1+1)$CMA-ES is transformed into a Metropolis-based sampler, the $(1+1)$CMA-ES sampler. Similar to GaA, this transformation is obtained in two steps. First, the Metropolis acceptance criteria in Algorithm 1 is used to evaluate the proposed value against the parent.

Second, if the the proposed value resides in a region of higher probability than the parent, then it is accepted with probability $p = 1$. Otherwise, it is accepted according to the Metropolis acceptance probability, see Algorithm 7.

Algorithm 6: Metropolis Gaussian Adaptation

Input: π, N, $\mathbf{x}_0$, $\mathbf{m}_0$, c_0, $\boldsymbol{\sigma}_0$, f_e , f_c, w_m,w_c
Output: $\mathbf{x}^{(1:N)} \in \pi$

for $n = 1, 2, 3 \cdots N$ **do**
- draw a candidate state $\mathbf{x}^* \sim \mathcal{N}(\mathbf{m}_n, c_n) = \mathbf{m}_n + \boldsymbol{\sigma}_n c_n^{\frac{1}{2}} \mathbf{z}_n$
- **compute MH acceptance criteria** $\alpha(\mathbf{x}_n|\mathbf{x}^*) = \min\left\{1, \frac{\pi(\mathbf{x}^*)}{\pi(\mathbf{x}_n)}\right\}$
- **if** $\alpha(\mathbf{x}_n|\mathbf{x}^*)$ **then**
  - $\mathbf{x}_{n+1} = \mathbf{x}^*$
  - expand $\boldsymbol{\sigma}_{n+1} = \boldsymbol{\sigma}_n.f_e$
  - update the parameters
    - $\mathbf{m}_{n+1} = \mathbf{m}_n(1 - w_m) + w_m(\mathbf{x}_{n+1} - \mathbf{x}_n)$
    - $c_{n+1} = c_n(1 - w_c) + (1 - w_c)((\mathbf{x}_{n+1} - \mathbf{x}_n)(\mathbf{x}_{n+1} - \mathbf{x}_n)^T)$
- **else**
  - $\mathbf{x}_{n+1} = \mathbf{x}_n$
  - contract $\boldsymbol{\sigma}_{n+1} = \boldsymbol{\sigma}_n.f_c$

Algorithm 7: $(1 + 1)$-CMA Sampling Algorithm

Input: f, $\mathbf{x}_0$, σ_0, c_0, $\bar{p}_s = p_s^{target}$, p_s^{succ}, $\mathbf{p}_c$

for $n = 1, 2, \cdots, N$ **do**
- generate a point $\mathbf{x}_{offspring} \sim \mathbf{x}_{parent} + \sigma_n \, \mathcal{N}(0, c)$
- update the step size
- **compute MH acceptance criteria** $\alpha(\mathbf{x}_{parent}|\mathbf{x}_{offspring}) = \min\left\{1, \frac{\pi(\mathbf{x}_{offspring})}{\pi(\mathbf{x}_{parent})}\right\}$
- draw $u \sim U[0, 1]$
- **if** $u < \alpha(\mathbf{x}_{parent}|\mathbf{x}_{offspring})$ **then**
  - parent state : $\mathbf{x}_{parent} = \mathbf{x}_{offspring}$
- **else**
  - parent state : $\mathbf{x}_{parent} = \mathbf{x}_{parent}$
- Update evolution path and covariance matrix

5.9 Comparing Parameter Updates

Adaptive samplers differ in the way the mean, covariance matrix and global scale are updated (Milgo et al., 2017).

Remark 5.8: Update of the Mean

$$\begin{aligned}
\text{AM} \qquad & \mathbf{m}_{n+1} = \frac{n}{n+1}\mathbf{m}_n + \frac{1}{n}\mathbf{x}_{n+1} \\
\text{GaA sampling} \qquad & \mathbf{m}_{n+1} = (1-w_{\mathbf{m}})\mathbf{m}_n + w_m \Delta\mathbf{x} \\
(1+1)-\text{CMA sampling} \qquad & \mathbf{p}_{n+1}^c = \begin{cases} (1-c_p)\,\mathbf{p}_n^c + \sqrt{c_p(2-c_p)}\mathbf{y} & \text{if accepted} \\ (1-c_p)\,\mathbf{p}_n^c & \text{otherwise} \end{cases}
\end{aligned}$$

Remark 5.9: Update of the Covariance Matrix

$$\begin{aligned}
\text{AM} \qquad & c_{n+1} = c_n + \frac{1}{n+1}\left(\left(\mathbf{x}_{n+1}-\mathbf{m}_n\right)\left(\mathbf{x}_{n+1}-\mathbf{m}_n\right)^\top - c_n\right) \\
\text{GaA sampling} \qquad & c_{n+1}^{\frac{1}{2}} = c_n^{\frac{1}{2}} \Delta c_{n+1}^{\frac{1}{2}} \\
(1+1)-\text{CMA sampling} \qquad & c_{n+1} = \begin{cases} (1-w_c)\,c_n + w_c \mathbf{p}_{n+1}^\top \mathbf{p}_{n+1} & \text{if accepted} \\ (1-w_c)\,c_n + w_c\left(\mathbf{p}_{n+1}^T \mathbf{p}_{n+1} + c_p(2-c_p)c_n\right) & \text{otherwise} \end{cases}
\end{aligned}$$

Remark 5.10: Update of the Global scale

AM does not update the global scale.

$$\text{AM} \qquad \boldsymbol{\sigma}_{n+1} = \boldsymbol{\sigma}_{opt}$$

$$\text{GaA sampling} \qquad \boldsymbol{\sigma}_{n+1} = \begin{cases} f_e \boldsymbol{\sigma}_n & \text{if accepted} \\ f_c \boldsymbol{\sigma}_n & \text{otherwise} \end{cases}$$

$$(1+1) - \text{CMA sampling} \qquad \boldsymbol{\sigma}_{n+1} = \boldsymbol{\sigma}_n \; exp\left(\frac{1}{k}\left(\bar{p}_s - \frac{p_s^{target}}{1 - p_s^{target}}(1 - \bar{p}_s)\right)\right)$$

5.10 Chapter Conclusion

This chapter explained two stochastic optimization methods, namely: GaA and CMA-ES. It gave an overview of their algorithms and parameter settings.

It then showed that updating the full covariance matrix comes with a high time-complexity. As a way to mitigate this computational cost, it introduced a recent method which updates the covariance matrix using its Cholesky factors, instead of updating the full matrix. This resulted in a significant drop in time-complexity. It then highlighted the main similarities and differences between GaA and CMA-ES.

Finally, this chapter transformed the two stochastic optimization techniques into samplers, namely: the Metropolis GaA and the $(1+1)$CMA samplers. The next chapter will go over the experiments conducted.

6 | Experiments

This chapter explains the experimental set-up, describes the target distributions used, the context in which each experiment is set up, and outlines the execution of algorithm runs.

The chapter then gives a summary of results from each of the algorithm runs. Most of the simulations were done in Python 3.4.

Section 6.1, describes how each run is executed, and the corresponding visualization process from the time data is stored. Performance of samplers based on various tests in Section 6.3 are shown, in addition to all their diagnostic graphs.

Finally, in Section 6.5, summary of results are provided.

6.1 Experimental Set-Up

Python code has been generated for most algorithms in this thesis. To compare these algorithms to one another, as well as diagnosing them in a meaningful way, a single experimental set-up has been developed.

6.1.1 Target distributions

Initial tests performed are the four target distributions found in Haario et al. (1999), namely:

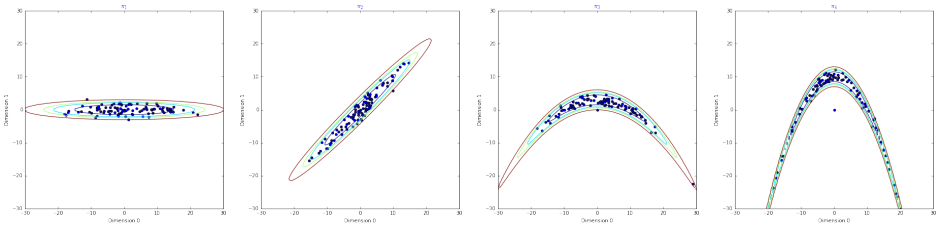
- π_1 - *uncorrelated Gaussian distribution*: is an n-dimensional multivariate normal distribution $\mathcal{N}(0, \mathbf{C_u})$, where $\mathbf{C_u} = diag(100, 1, 1, ..., 1)$. It is an uncorrelated

covariance matrix leading to a hyper-ellipsoid shape with axis-aspect ratio of 10.
$c_u = \begin{pmatrix} 100 & 0 \\ 0 & 1 \end{pmatrix}$

- π_2 - *correlated Gaussian distribution*: is the target distribution π_1 rotated such that the long semi-axis corresponds to the direction $(1, 1, ..., 1)$.
  $c_c = \begin{pmatrix} 50.5 & 49.5 \\ 49.5 & 50.5 \end{pmatrix}$
- π_3 - *moderately twisted Gaussian distribution*: is a twisted non-linear distribution. It is Rosenbrock's function and often called Rosenbrock's banana function, or simply the banana function. The density function of the twisted Gaussian distribution is given as:

$$g_b = g(\phi_b(\mathbf{x})) \tag{6.1}$$

  where $\phi_b(\mathbf{x}) = (\mathbf{x}_1, \mathbf{x}_2 + b\mathbf{x}_1^2 - 100b, \mathbf{x}3, ..., \mathbf{x}_n)$.
  b is called the twisting factor, and as it becomes larger than 1, linearity of the function g_b becomes stronger. The moderately linear target distribution function π_3 was generated by setting $b = 0.03$.
- π_4 - *highly twisted Gaussian distribution*: similar to π_3 except that the value of the twisting factor b was set to $b = 0.1$



(a) π_1 (b) π_2 (c) π_3 (d) π_4

Figure 6.1: Haario's target distributions illustrated.

These target distributions have been used in (Muller, 2010; Muller and Sbalzarini, 2010; Roberts and Rosenthal, 2009a; Sejdinovic et al., 2014) among others, to implement adaptive MCMC methods. These distributions are suitable for the experiments of this thesis, because of their complexity, and benchmarking with existing adaptive methods.

6.1.2 Context of Experiments

Each run of experiments are set-up with a *Context*. This is a named tuple containing:

- (d): *dimension* of the *State Space*;
- *TestSuite*: the test suite used, and;
- N: the number of samples to be generated.

The test-suite used herein constructs the target distributions, π_1 to π_4, from the standard Normal distribution $\mathcal{N}(0, 1)$, ordered by increasing complexity.

6.1.3 Specification of Data Runs

Data obtained from running a context is named herein as *RunData*. RunData is therefore a named tuple which contains the following:

- *Context*: the Context of the run;
- *Sampler*: the Sampler used in the run
- *TargetIdx*: index of the target used in the test suite (*TestSuite*), and;
- *RunIndex*: an index unique to the run.

> **Remark 6.1: Data Storage**
>
> Executing a run returns a named tuple that is referred to as *DataStorage*. It contains all run-related information.

> **Remark 6.2: Data for Sampler Comparisons**
>
> A single experiment for each sampler was run on a particular target distribution. This enabled performance comparisons per algorithm.

6.1.4 Visualization of a Run

Visualization, in this thesis, is restricted to scattering samples of the first two dimensions of the state space. Experimental data are kept in a variable named *DATA*, and used to visualize a run. To do so, the following parameters have to be specified:

- $datastorage$; where the experimental data are stored;

- $nbsamples$; the number of samples to be plotted;
- $plotdimensions$; in these experiments, and;
- $burninpct$; the percentage of samples considered to belong to the burn-in period.

Finally, samples are plotted and compared with $i.i.d.$ generated samples.

6.1.5 Acceptance Ratio

After N iterations, each sampler will accept sample proposals with an acceptance ratio computed at the end of *one* run. This is conventionally the ratio of the number of successful jumps to the total number of iterations. Efficiency of a sampler is illustrated by its acceptance rate.

All samplers are compared, first, on a single target distribution, say π_2. Then for each sampler, performance is tested with all the targets, with increasing order of complexity.

6.1.6 Confidence Regions

In order to compare the performance of algorithms, the number of samples were distributed relative to samples falling within 68.3%,90%,95% and 99% confidence regions, using the Chi-squared values.

Since the target distributions are multivariate, samples were visualized in contours, representing confidence levels in the Mahalanobis distance. Four contour lines between 1σ and 3σ represented standard deviations of the sample space. Then convergence diagnostic tools were applied, as described in Section 3.4.

Definition 6.1: Mahalanobis Distance

A statistical distance between two vector points $\mathbf{x}$ and $\mathbf{y}$ taking into consideration their correlation. The distance between two points $\mathbf{x} = (x_1 \cdots x_d)^T$ and $\mathbf{y} = (y_1, ..., y_d)$ in the dimension $\mathbb{R}^d$ is defined as:

$$d_M(\mathbf{x}, \mathbf{y}) = \sqrt{(\mathbf{x} - \mathbf{y})^T c^{-1} (\mathbf{x} - \mathbf{y})} \tag{6.2}$$

where c^{-1} is the precision matrix.

A positive definite covariance matrix can uniquely be identified with an ellipsoid surface of equal density to the distribution. A diagonal covariance matrix has d parameters, and $0s$ in the off-diagonal terms. A spherical or isotropic covariance matrix has one free parameter.

The principal axes of the ellipsoid correspond to the eigenvectors of c while the squared axes lengths corresponds to its eigenvalues (Hansen, 2010).

6.2 Experiments

Experiments were performed to evaluate the target function π_4 using each of 3 adaptive MCMC samplers, namely:

- Two existing Adaptive Metropolis Sampling methods
  - M-GaA, and;
  - AM
- One new method proposed in this thesis:
  - The Metropolis $(1+1)$ CMA

Experiments were performed given each of the samplers, and in turn compared with an $i.i.d.$ sampler. For consistency, the same dimensions, iterations and target distribution (π_4), were used. Figure 6.2 plots the last 350 samples generated per each of the three algorithms, and with the set-up $d = 11$ and $N = 60000$.

The figure illustrates all three algorithms to be at least just as efficient as an $i.i.d.$ sampler. Samples clearly traverse the space required, the benchmark banana function which was chosen because of its complexity as a target distribution. Additional diagnostics tests are needed to both illustrate and quantify performance, this is for the next section, Section 6.3.

The percentage errors of samples falling within four confidence regions is presented in Table 6.2. The MH algorithm was used as a benchmark.

Percentages	68.3%	90%	95%	99%
AM Fractions	0.07	0.083	0.113	0.007
$i.i.d.$ Fractions	0.055	0.093	0.108	0.142
GaA Fractions	0.061	0.1	0.11	0.142
$i.i.d.$ Fractions	0.054	0.09	0.11	0.135
$(1+1)$CMA-ES Fractions	0.028	0.073	0.093	0.105
$i.i.d.$ Fractions	0.52	0.92	0.108	0.142

Table 6.1: Comparing performance of the three algorithms by acceptance, in four confidence regions: 68.3%, 90%, 95% and 99%.

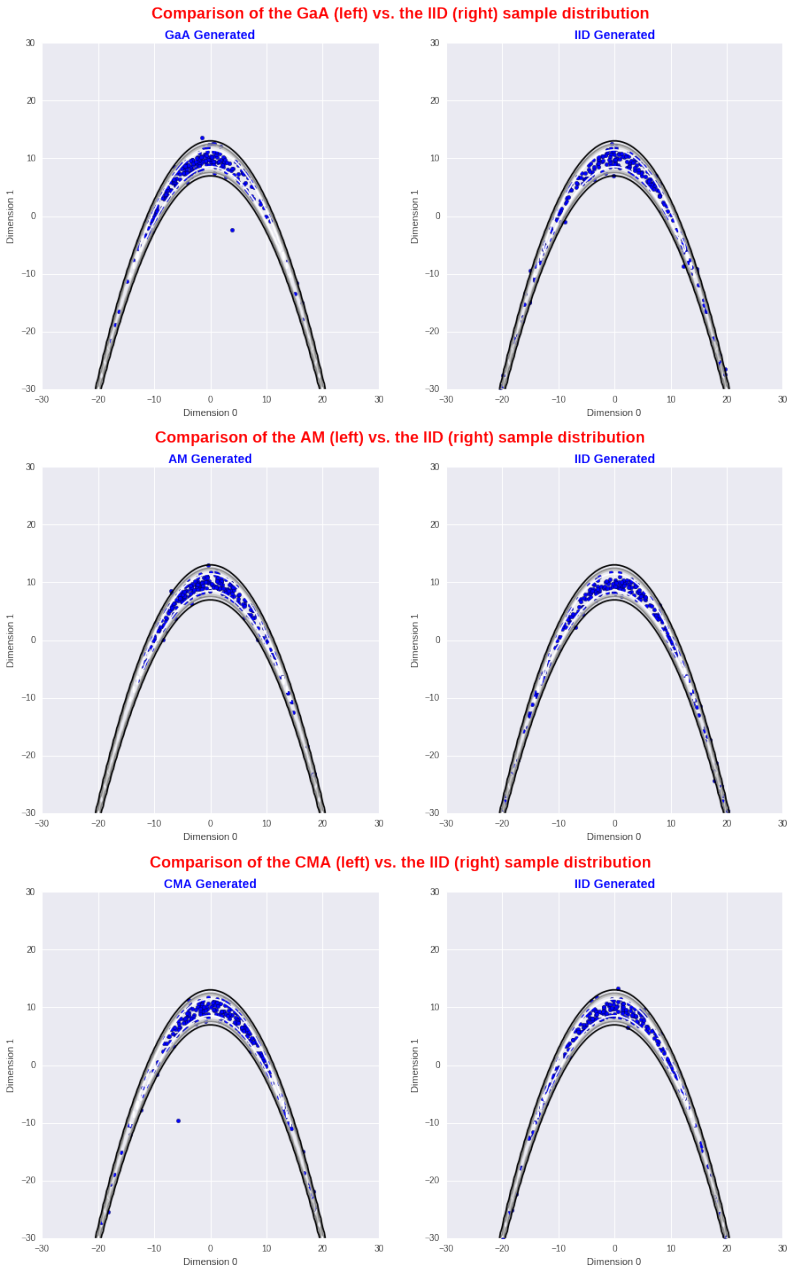


Figure 6.2: Comparison of samplers(left) vs. i.i.d. for the highly twisted Gaussian distribution, π_4

6.3 Diagnostic Tests

The key idea is to evaluate convergence and estimate when the sampling process must be terminated.

Highly correlated samples should be discouraged since they lead to slower convergence to the target distribution. $I.i.d.$ samples, for instance, usually converge faster to the target distribution. The challenge with $i.i.d.$ sampling is that it does not work well with high dimensional spaces and complex distributions.

For an explanation of the diagnostic tools used in this thesis, see Section 3.4. In this section, they are used to assess the experiments herein.

6.3.1 Autocorrelation Measure

Recall from Section 3.4 that the ACF value, denoted $\mathbf{r}_i$, points to a sampler that is performing optimally and to the samples becoming increasingly independent, if it decreases with the number of iterations.

For these experiments, the autocorrelation measure is illustrated for each of the samplers on a non-correlated target, π_4 in this case. See Figure 6.3.

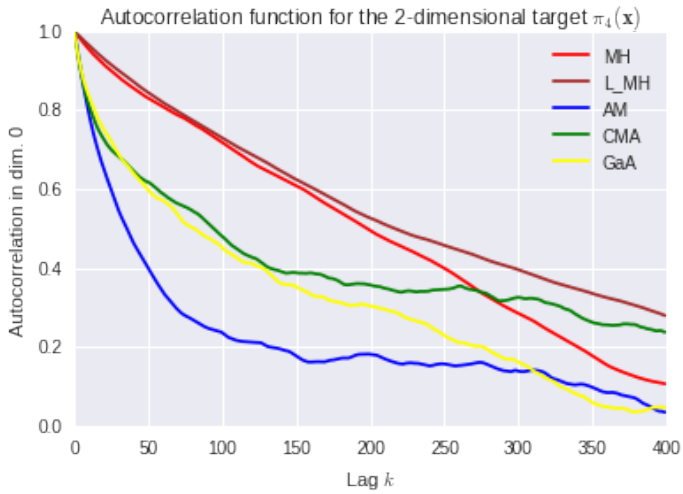


Figure 6.3: Comparison of the autocorrelation measure of MH, AM, and $(1+1)$- CMA

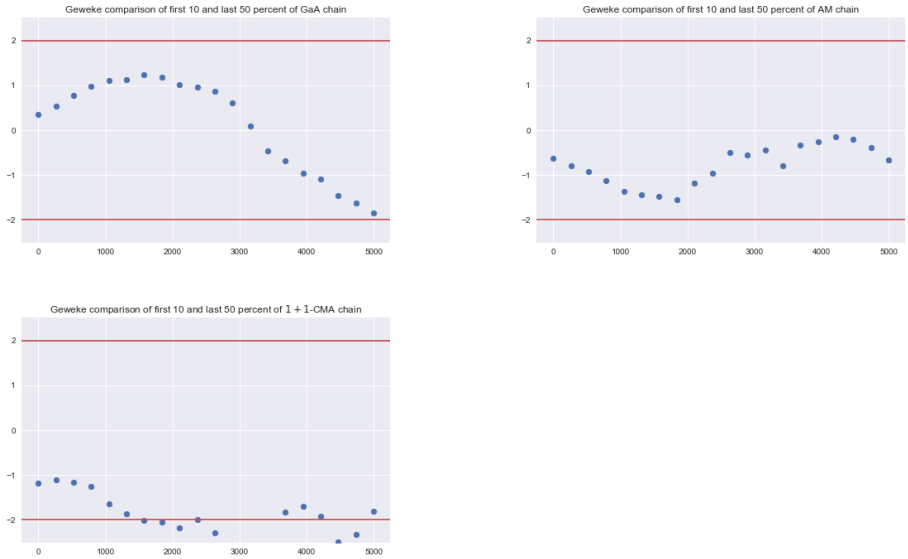


Figure 6.4: Illustration of Geweke measures for samples from three algorithms with target distribution π_3. The horizontal lines show the limits that should not be exceeded, for optimal results. The graph represents two segments of the first 10% and last 50% of samples after burn-in. The difference in means among the sample intervals in both segments should be minimal when convergence has occurred.

6.3.2 Geweke Measures

Recall from Section 3.4 that Geweke is a measure of ergodicity of the chain. Here it is used to evaluate the different experiments obtained, see Figure 6.4

6.4 Results

Results obtained from the diagnostic tests have demonstrated that the Python code is according to the techniques upon which it has been developed. Results have also shown that the code is numerically stable. This is crucial, especially for the rank-one-update algorithm. Ensuring its numerical stability allowed it to replace the full matrix update step of the covariance matrix in $(1+1)$CMA-ES with a more efficient update step. As a result, a significant decrease in its computational time complexity.

Remark 6.3: Numerical Stability

It was crucial to ensure that the covariance matrix can be computed from its Cholesky factors at any point. I.e., updates to the Cholesky factor L correspond to updates to the covariance matrix C.

It was crucial to ensure that updates to As a result, experiments have successfully simulated samples from the benchmark Rosenbrock function; the banana shaped function.

The newly presented $(1+1)$CMA-ES sampler has proven comparable in performance to the $i.i.d.$ sampler as well as the AM algorithm. Table 6.2 captures the efficiency by which the $(1+1)$CMA-ES samples from the target distribution, π_4. It outperformed each of the other algorithms, except for the 99% confidence region where the AM algorithm outperformed.

In addition, Figure 6.2 shows the $(1+1)$CMA-ES is capable in generating samples from the target distribution. It is clear, from visual inspection, that it traverses well the target distribution, it is also able to generate samples from the distribution space relatively well.

Its autocorrelation function, ACF, in Figure 6.3, has shown a steady decrease as the number of samples increases. Not showing spikes but rather a monotonic decrease means the samples are increasingly and consistently displaying $i.i.d.$ properties.

The Geweke diagnostic tool illustrates that the the $(1+1)$CMA-ES exhibits strong convergence, even outperforming convergence in the AM algorithm. See Figure 6.4. This reinforces the notion that the the $(1+1)$CMA-ES is a strong sampler, as its Geweke results show the necessary ergodic property.

In each of these experiments, the $(1+1)$CMA-ES also outperforms the GaA sampler in convergence, sampling errors, and ACF.

Although performance of the the $(1+1)$CMA-ES is demonstrated here, it is important to recall that the number of samples N and the dimensionality d are relatively small. In problems of real impact, the dimensionality is at a much larger scale, and so is the number of samples.

A next step would be to run this Python code on a supercomputer platform, and test its performance on highly complex problems.

6.5 Chapter Conclusion

This chapter has explained the experimental set-up of the experiments generated. It used a homogeneous set-up in order to allow various comparisons among algorithms. It also demonstrated the workings of the diagnostic tools and used them to further depict performance differences among the various algorithms. These tools gave deeper insight to

sampling performance, and convergence to the target function. Finally, it explained the results obtained from running the newly presented $(1 + 1)$CMA-ES sampler, and how it faired with other algorithms.

7 | Conclusion

This thesis is prefaced by stating the importance of Bayesian inference in machine learning. It explained that Bayesian inference has become popular contemporaneously with enhancements in computational capabilities. The main reason for its renaissance is the ability to evaluate intractable integrals, a requirement for most difficult, yet interesting problems. A number of techniques that attempt to evaluate said integrals have been explained, along with their shortcomings.

It was explained how MCMC methods can sample from any distribution, regardless of problem dimensionality, a factor which is often directly related to the intractability of problems. However, these methods require specific set-ups and hence, are subject to initial conditions, such as the choice of a proposal distribution or the step-size.

Contemporary MCMC methods have overcome this by adapting proposal distribution parameters, hence the name: adaptive MCMC. They start with any known distribution as a proposal distribution, usually selected is the Gaussian distribution. In each run of the algorithm, samples are used to probabilistically tune search parameters towards higher probability regions of the search space.

Stochastic optimization methods were explained herein. These are powerful and widely used numerical optimization methods for non-linear or non-convex optimization problems. Parameters are updated to become oriented towards regions of optimal values. Updating the full covariance matrix has a high time complexity. As a way to mitigate this, a more computationally efficient technique, rank-one-update of the covariance matrix, was explained. Instead of directly updating the covariance matrix, it is indirectly updated by its Cholesky factors.

Contribution of this thesis has been to devise and evaluate a new algorithm that combines two widely used and powerful computational techniques, namely stochastic optimizers and samplers. The Metropolis acceptance ratio has been incorporated into the stochastic optimizer $(1 + 1)$CMA-ES, transforming it into a sampler.

Combining the strongest properties of two powerful algorithms highlights the current tools for attempting solutions to difficult problems. Results herein have consistently shown that the Metropolis-based $(1 + 1)$CMA-ES sampler competes well with the AM algorithm, and is able to efficiently converge to the benchmark function, the Rosenbrock function. Its sampling performance is competitive when compared to other algorithms.

A suggestion for a further exercise is to tackle a difficult problem. Specifically, it would be to chose a high-dimensional and multi-modal problem to run this code on. This will inevitably require use of the higher computational power of a supercomputer. The next step from here is to share the code with the outside community, rendering it a piece of open software. The objective is to share the knowledge gained from this thesis and add it to the ever growing body of science. Furthermore, it is to help contribute to the understanding and improvement of Bayesian inference.

Bibliography

Andrieu, C. and J. Thoms
2008a. A tutorial on adaptive mcmc. *Statistics and Computing*, 18-4:343–373.

Andrieu, C. and J. Thoms
2008b. A tutorial on adaptive mcmc. *Statistics and Computing*, 18(4):343–373.

Bäck, T.
1996. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press.

Bäck, T., C. Foussette, and P. Krause
2013. *Contemporary Evolution Strategies*, Natural Computing Series. Springer Berlin Heidelberg.

Bäck, T. and H.-P. Schwefel
1993. An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation*, 1(1):1–23.

Beyer, H.-G. and H.-P. Schwefel
2002. Evolution strategies – a comprehensive introduction. *Natural Computing*, 1(1):3–52.

Brooks, S., A. Gelman, G. L. Jones, and X.-L. Meng
2011. *Handbook of Markov Chain Monte Carlo (Chapman & Hall/CRC Handbooks of Modern Statistical Methods)*. Chapman & Hall CRC.

Gelman, A., G. O. Roberts, and W. R. Gilks
1996. Efficient metropolis jumping rules. In *Bayesian Statistics*, J. M. Bernado et al., eds., volume 5, Pp. 599–607.

Glasmachers, T., T. Schaul, S. Yi, D. Wierstra, and J. Schmidhuber
2010. Exponential natural evolution strategies. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, Pp. 393–400. ACM.

Haario, H., M. Laine, A. Mira, and E. Saksman
2006. Dram: Efficient adaptive mcmc. *Statistics and Computing*, 16-4:339–354.

Haario, H., E. Saksman, and J. Tamminen
1999. Adaptive proposal distribution for random walk metropolis algorithm. *Computational Statistics*, 14-3:375–395.

Haario, H., E. Saksman, and J. Tamminen
2001a. An adaptive metropolis algorithm. *Bernoulli*, 7 no. 2:223–242.

Haario, H., E. Saksman, J. Tamminen, et al.
2001b. An adaptive metropolis algorithm. *Bernoulli*, 7(2):223–242.

Hansen, N.
2010. The cma evolution strategy: A tutorial.

Hansen, N., D. V. Arnold, and A. Auger
2015. Evolution strategies. In *Springer Handbook of Computational Intelligence*, Pp. 871–898.

Hansen, N. and A. Ostermeier
1996. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, Pp. 312–317. IEEE.

Hansen, N. and A. Ostermeier
2001. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9:159–195.

Igel, C., T. Suttorp, and N. Hansen
2006a. A computational efficient covariance matrix update and a (1+1)-cma for evolution strategies. Pp. 453–460.

Igel, C., T. Suttorp, and N. Hansen
2006b. A computational efficient covariance matrix update and a (1+1)-cma for evolution strategies. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, Pp. 453–460, New York, NY, USA. ACM.

Kjellstrom, G. and L. Taxen
1981. Stochastic optimization in system design. *Circuits and Systems, IEEE Transactions*, 28-7:702 – 715.

Krause, O. and C. Igel
2015. A more efficient rank-one covariance matrix update for evolution strategies. In *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII*, Pp. 129–136. ACM.

Liang, F., C. Liu, and R. Carroll
2010. *Advanced Markov Chain Monte Carlo Methods: Learning from Past Samples*, Wiley Series in Computational Statistics, 1 edition. New York, NY: Wiley.

Liang, F., C. Liu, and R. Carroll
2011. *Advanced Markov Chain Monte Carlo Methods: Learning from Past Samples*, Wiley Series in Computational Statistics. Wiley.

MacKay, D.
2003. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press.

McGrayne, S.
2011. *The Theory that Would Not Die: How Bayes' Rule Cracked the Enigma Code, Hunted Down Russian Submarines, & Emerged Triumphant from Two Centuries of Controversy*. Yale University Press.

Milgo, E., N. Ronoh, P. Waiganjo, and B. Manderick
2017. Adaptiveness of cma based samplers. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, Pp. 179–180, New York, NY, USA. ACM.

Muller, C. L.
2010. Exploring the common concepts of adaptive mcmc and covariance matrix adaptation schemes. In *Theory of Evolutionary Algorithms*, A. Auger, J. L. Shapiro, L. D. Whitley, and C. Witt, eds., Dagstuhl, Germany.

Muller, C. L. and I. F. Sbalzarini
2010. Gaussian adaptation as a unifying framework for continuous black-box optimization and adaptive monte carlo sampling. Pp. 1–8.

Müller, C. L. and I. F. Sbalzarini
2010. Gaussian adaptation revisited - an entropic view on covariance matrix adaptation. volume 6024, Pp. 432–441. Springer Berlin Heidelberg.

Müller, C. L. and I. F. Sbalzarini
2011. Gaussian adaptation for robust design centering. Capua, Italy.

Murray, I.
. Markov chain monte carlo.

Robert, C. P.
2014. *The Metropolis,ÄìHastings Algorithm*. John Wiley & Sons, Ltd.

Roberts, G. O. and J. S. Rosenthal
2001. Optimal scaling for various metropolis-hastings algorithms. *Statistical Science*, 16(4):351–367.

Roberts, G. O. and J. S. Rosenthal
2007. Coupling and ergodicity of adaptive markov chain monte carlo algorithms. *Journal of Applied Probability*, 44(2):458,Äì475.

Roberts, G. O. and J. S. Rosenthal
2009a. Examples of adaptive mcmc. *Journal of Computational and Graphical Statistics*, 18-2:349–367.

Roberts, G. O. and J. S. Rosenthal
2009b. Examples of adaptive mcmc. *Journal of Computational and Graphical Statistics*, 18(2):349–367.

Schwefel, H.
1995. *Evolution and Optimum Seeking*, Sixth Generation Computer Technologies. Wiley.

Sejdinovic, D., H. Strathmann, M. L. Garcia, C. Andrieu, and A. Gretton
2014. Kernel adaptive metropolis-hastings. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, Pp. 1665–1673. JMLR.org.

Suttorp, T., N. Hansen, and C. Igel
2009a. Efficient covariance matrix update for variable metric evolution strategies. *Machine Learning*, 75-2:167–197.

Suttorp, T., N. Hansen, and C. Igel
2009b. Efficient covariance matrix update for variable metric evolution strategies. *Machine Learning*, 75(2):167–197.

VanderPlas, J.
2014. Frequentism and bayesianism: A python-driven primer.

Wierstra, D., T. Schaul, T. Glasmachers, Y. Sun, J. Peters, and J. Schmidhuber
2014. Natural evolution strategies. *Journal of Machine Learning Research*, 15:949–980.

A | Linear Algebra

A.1 Matrix Algebra

Definition A.1: Square Matrix

Let $\mathbf{A}$ be a matrix whose elements are a_{ij}, where i denotes the i^{th} row and the j^{th} column. $\mathbf{A}$ is called squared if its number of rows d equal the number of columns. $\mathbf{A}$ is then said to be an $d \times d$ matrix.

Definition A.2: Matrix Transpose

This is $\mathbf{A}^\top$ whose columns are the rows of $\mathbf{A}$, and whose rows are the columns of $\mathbf{A}$. The matrix $\mathbf{A}^\top$ therefore has the elements a_{ji} and has exactly the same dimensions as $\mathbf{A}$.

Definition A.3: Symmetric Matrix

This is the square matrix $\mathbf{A}$ whose elements a_{ij} are such that $a_{ij} = a_{ji}$. i.e., $\mathbf{A} = \mathbf{A}^\top$

Definition A.4: Diagonal Matrix

This a square matrix whose only nonzero elements appear on the main diagonal. Usually denoted as $\mathbf{D}$, its elements λ_i are non zero if and only if $i = j$ for the i^{th} row and the j^{th} column.

Definition A.5: Identity Matrix

This is a scalar matrix with ones on the diagonal. This matrix is usually denoted as $\boldsymbol{I}_d$, where n is the dimension of the matrix.

Definition A.6: Triangular Matrix

This is a matrix whose elements are all zeros on either side of the major diagonal. If the zeros are above the diagonal, the matrix is called lower triangular and denoted as $\mathbf{L}$. If the zeros are below, we have the upper triangular matrix usually denoted as $\mathbf{U}$.

Definition A.7: Trace of a matrix

The trace $tr(\mathbf{A})$ of a $d \times d$ matrix $\mathbf{A}$ is the sum of its diagonal elements:

$$tr(A) = \sum_{i=1}^{d} a_i \tag{A.1}$$

A.2 The Determinant of a square matrix

The determinant of a matrix $\mathbf{A}$ is a multi-linear function of the elements a_{ij} which measures the transformation described by $\mathbf{A}$. It is normally denoted as $\det(\mathbf{A})$ or $|\mathbf{A}|$, and is computed as:

$$\det(\mathbf{A}) = \sum_{i=1}^{m} a_{ij}(-1)^{i+j} \det(A_{ij}) \tag{A.2}$$

where $i = 1, \cdots, d$ and $\mathbf{A}_{ij}$ is the matrix obtained by deleting the i_{th} row and the j_{th} column of the matrix $\mathbf{A}$.

Remark A.1

For a 2×2 matrix, it is calculated as the difference of the product of the major diagonal and the product of the elements of the minor diagonal. We have:

$$\det(\mathbf{A}) = a_{11}a_{22} - a_{21}a_{12} \tag{A.3}$$

A.3 The Inverse of a square matrix

Consider a matrix $\mathbf{A}$ whose determinant $\det(\mathbf{A}) \neq 0$.

Remark A.2

The inverse $\mathbf{A}^{-1}$ of the matrix $\mathbf{A}$ is unique, and has the property that

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \boldsymbol{I} \tag{A.4}$$

If $\mathbf{A}$ is a symmetric matrix, then $\mathbf{A}^{-1}$ is also symmetric.

Definition A.8: Cofactor Matrix

This is the matrix $\mathbf{A}_{cof}$ whose elements are $(-1)^{i+j}\det(\mathbf{A}_{ij})$, where $\mathbf{A}_{ij}$ is a matrix obtained by deleting the i^{th} row and the j^{th} column of $\mathbf{A}$.

The inverse $\mathbf{A}^{-1}$ of $\mathbf{A}$ is obtained as:

$$\mathbf{A}^{-1} = \frac{1}{\det(\mathbf{A})}\ \mathbf{A}_{cof} \tag{A.5}$$

Definition A.9: Nonsingular Matrix

A matrix is said to be nonsingular if it satisfies the necessary and sufficient condition $\det(\mathbf{A}) \neq 0$.

A.4 Eigenvalues and Eigenvectors

Consider the set of equations

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \tag{A.6}$$

The solutions to Equation (A.6) are the pairs of *eigenvectors* $\mathbf{x}$ and *eigenvalues* λ. Since multiplication by an identity matrix does not change the right hand side of (A.6), we can write

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{I}\mathbf{x} \tag{A.7}$$

which leads to $(\mathbf{A} - \lambda\mathbf{I})\mathbf{x} = 0$. This implies that

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0 \tag{A.8}$$

which leads to a polynomial of degree d in λ called the *characteristic equation* of $\mathbf{A}$.

Definition A.10: Eigenvalues

The scalar multiples λ are called the *eigenvalues* of $\mathbf{A}$.

The eigenvalues give an indication of the level to which $\mathbf{x}$ stretches or shrinks, or whether it changes when multiplied by $\mathbf{A}$. The eigenvalues of an *identity matrix* are *unit*. For this special case, $\mathbf{A}\mathbf{x} = \mathbf{x}$.

Definition A.11: Eigenvectors

These are the vectors $\mathbf{x}$ which are in the same direction as $\mathbf{A}\mathbf{x}$.

Equation (A.6) implies that $\mathbf{A}\mathbf{x}$ is λ times the original vector $\mathbf{x}$. The eigenvalues λ of a matrix $\mathbf{A}$ describe the effect of multiplying a vector $\mathbf{x}$ by $\mathbf{A}$.

The *eigenvectors* of the symmetric and positive definite matrix indicate the direction of the variance of the data, with the highest valued vector representing the direction of the largest variance.

Definition A.12: Orthogonal Vectors

Two vectors $\mathbf{x}$ and $\mathbf{y}$ are said to be orthogonal if their inner product $\mathbf{x}^\top \mathbf{y}$ is zero. This implies $\mathbf{x} \perp \mathbf{y}$

A.5 Diagonalization of the Covariance Matrix

The diagonalized expression of the covariance matrix, usually denoted as $\mathbf{C}$ is the form

$$\mathbf{C} = \mathbf{IC} = \mathbf{B}^\top \mathbf{BC} = \mathbf{B}^\top \mathbf{DB} \tag{A.9}$$

where $\mathbf{D}$ is a diagonal matrix whose elements are the eigenvalues of $\mathbf{C}$, and $\mathbf{B}$ is the basis vector such that $\mathbf{BB}^\top = \mathbf{BB}^{-1} = \mathbf{I}$. We can write Equation (A.9) therefore as

$$\mathbf{C} = \mathbf{B} \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & \lambda_d \end{bmatrix} \mathbf{B}^\top \tag{A.10}$$

If we have determined the eigenvalues and the eigenvectors of $\mathbf{C}$, we have $\mathbf{C}$ to be the matrix

$$\mathbf{D} = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & \lambda_d \end{bmatrix} \tag{A.11}$$

Equation (A.10) leads to useful definitions that relate to the determination of the square root and the inverse of $\mathbf{C}$.

Definition A.13: Matrix Square Root

The matrix square root of the covariance matrix $\mathbf{C}$ is $\mathbf{C}^{\frac{1}{2}}$ such that:

$$\mathbf{C}^{\frac{1}{2}} \triangleq \mathbf{BD}^{\frac{1}{2}} \mathbf{B}^\top \tag{A.12}$$

Equation (A.12) agrees with the usual definition of a square root since:

$$\mathbf{C}^{\frac{1}{2}} \mathbf{C}^{\frac{1}{2}} = \mathbf{BD}^{\frac{1}{2}} \mathbf{B}^\top \mathbf{BD}^{\frac{1}{2}} \mathbf{B}^\top = \mathbf{BD}^{\frac{1}{2}} \mathbf{ID}^{\frac{1}{2}} \mathbf{B}^\top = \mathbf{BD}^{\frac{1}{2}} \mathbf{D}^{\frac{1}{2}} \mathbf{B}^\top = \mathbf{BDB}^\top$$

which from Equation (A.9) simplifies to

$$\mathbf{C}^{\frac{1}{2}}\mathbf{C}^{\frac{1}{2}} = \mathbf{C}$$

Definition A.14: The Inverse of the covariance Matrix

The matrix inverse of $\mathbf{C}$ is $\mathbf{C}^{-1}$ such that:

$$\mathbf{C}^{-1} \triangleq \mathbf{B}\mathbf{D}^{-1}\mathbf{B}^{\top} \tag{A.13}$$

Equation (A.13) agrees with the usual definition of an inverse since:

$$\mathbf{C}^{-1}\mathbf{C} = \mathbf{B}\mathbf{D}^{-1}\mathbf{B}^{\top}\mathbf{B}\mathbf{D}\mathbf{B}^{\top} = \mathbf{B}\mathbf{D}^{-1}\mathbf{I}\mathbf{D}\mathbf{B}^{\top} = \mathbf{B}\mathbf{I}\mathbf{B}^{\top} = \mathbf{B}\mathbf{B}^{\top} = \mathbf{I} \tag{A.14}$$

A.6 Positive Definite Matrices

Consider a $d \times d$ square matrix $\mathbf{A}$ whose elements are a_{ij}, where $1 \leq i, j \leq d$. $\mathbf{A}$ is said to be positive definite if for all vectors $\mathbf{x} \neq 0$, the quadratic form $\mathbf{x}^{\top}\mathbf{A}\mathbf{x} > 0$. $\mathbf{A}$ is called positive semidefinite if $\mathbf{x}^{\top}\mathbf{A}\mathbf{x} \geq 0$ for all $\mathbf{x} \neq 0$

Remark A.3

If $\mathbf{A}$ is positive definite, all the eigenvalues $\lambda_1, ..., \lambda_d$ of $\mathbf{A}$ are positive, and all the sub-matrices are also positive definite.

Theorem A.1: Positive Definite Matrices

Let $\mathbf{A}$ be a $d-$dimensional symmetric matrix.

- If the eigenvalues of $\mathbf{A}$ are all positive, then the matrix $\mathbf{A}$ is *positive definite*
- If the eigenvalues of $\mathbf{A}$ are all negative, then the matrix $\mathbf{A}$ is *negative definite*
- If some of the eigenvalues are zero and all the others are positive (negative), then $\mathbf{A}$ is non-negative (non-positive) definite
- $\mathbf{A}$ is indefinite if its eigenvalues have both signs

Theorem A.2: Cholesky Factor

There exists a lower diagonal matrix factor $\mathbf{L}$ whose elements are l_{ij} such that $l_{ij} = 0$ for all $i > j$:

$$\mathbf{A} = \mathbf{L}\mathbf{L}^\top \tag{A.15}$$

The Cholesky factor $\mathbf{L}$ as defined in Equation A.15 is always positive definite. It can easily be seen that the determinant of $\mathbf{L}$ is the product of its diagonal elements, since the all the entries in the upper part of $\mathbf{L}$ are zero.

Example A.1

Let the covariance matrix $\mathbf{C}$ be $3-$ dimensional. The components of $\mathbf{L}$, its lower triangular matrix are l_{ij}, for $i, J > 0$, such that $i \geq j$. It is easy to see that the determinant of $\mathbf{L}$ is $\det(\mathbf{L}) = l_{11}l_{22}l_{33}$. If $\mathbf{L}$ is normalized, we end up with a matrix $\mathbf{A} = \frac{\mathbf{L}}{\det(\mathbf{L})}$ whose determinant is unit.

B | Probability Theory

B.1 Specific Probability Distributions

B.1.1 The Multivariate Normal Distribution

A multivariate normal distribution is fully defined by its mean $\mathbf{m}$ and covariance matrix $\mathbf{C}$. Its general form is the expression

$$P(\mathbf{x}|\mathbf{m}, \mathbf{C}) = \frac{1}{\sqrt{(2\pi)^d \det(\mathbf{C})}} e^{\frac{1}{2}\{(\mathbf{x}-\mathbf{m})^\top \ \mathbf{C}^{-1} \ (\mathbf{x}-\mathbf{m})\}} \tag{B.1}$$

Definition B.1: Mahalanobis Distance

The term $\{(\mathbf{x}-\mathbf{m})^\top \ \mathbf{C}^{-1} \ (\mathbf{x}-\mathbf{m})\}$ in Equation (B.1) represents the squared distance of $\mathbf{x}$ from the mean $\mathbf{m}$ and is referred to as the Mahalanobis distance.

Remark B.1

$\mathbf{C}^{-1}$ as defined in Equation (B.1) is the precision matrix.

B.1.2 Beta Distribution

A Beta Distribution is a family of continuous probability distributions defined by parameter θ and parameterized by α and β, where $\alpha, \beta \in \mathbb{R}$ and $\alpha > 0, \beta > 0$ θ. α, β take on values as mentioned above, where $\alpha, \beta \in \mathbb{R}$ and $\alpha > 0, \beta > 0$. α and β are termed *hyperparameters* because they are external to the model, and their values are set before any data is generated. The distribution is only defined for values of θ in the range $[0, 1]$, hence, its applications in questions concerning probabilities. It is often used to calculate a prior probability of an event because an infinite range for priors can be obtained by changing either or both the hyperparameters α and β. This flexibility in choosing a prior is certainly an advantage for using a Beta distribution.

One useful property of a Beta distribution is that it's a conjugate prior of the Binomial distribution; the posterior of a Binomial distribution is a Beta distribution. Modeled as $X \sim Bin(n, p)$, where X is the number of successful experiments (generated data); and with parameters n: the number of experiments; and p probability of successes, the Binomial distribution is a discrete probability distribution for successful events. When the probability of successes p is unknown, its prior can be modeled as $p \sim Beta(\alpha, \beta)$. This prior is then updated based on the generated data, X, resulting in its posterior distribution, $p(\theta|X)$ which is also a Beta distribution. This property is very useful for testing for bias of a coin where α represents the number of successes and β represents the number of failures. Another useful property are its strong connections to other distributions, such as the Gamma distribution, $\Gamma(\alpha, \beta)$.

The Beta Probability Density Function, pdf, is defined as:

$$\pi(\theta; \alpha, \beta) = \frac{1}{B(\alpha, \beta)} \theta^{\alpha-1}(1-\theta)^{\beta-1} \tag{B.2}$$

To simplify, set $c = \frac{1}{B(\alpha,\beta)}$, were c is the normalizing constant required for assuring the total probability integrates to 1. Then the Beta probability density function becomes:

$$p(\theta; \alpha, \beta) = c\theta^{\alpha-1}(1-\theta)^{\beta-1} \tag{B.3}$$

Note that the normalizing factor, c, can be obtained by integrating over $\theta^{\alpha-1}(1-\theta)^{\beta-1}$, on the interval $[0,1]$, in order to have the PDF integrate to 1.

$$\pi\left(\theta \mid \alpha, \beta\right) \stackrel{\Delta}{=} \frac{1}{B(\alpha,\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1} \tag{B.4}$$

Remark B.2: Beta Distribution

The Beta probability distribution function, pdf, is a two parameter continuous distribution with range $[0,1]$, and is defined as:

$$\pi\left(\theta \mid \alpha, \beta\right) \stackrel{\Delta}{=} \frac{1}{B(\alpha,\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1} \tag{B.5}$$

$B(\alpha,\beta)$ is called the Beta function, not to be confused with its distribution, and is a normalizing constant. For $\alpha,\beta \in \mathcal{N}^{+}$, as they are in this example, it is defined as:

$$B(\alpha,\beta) = \frac{(\alpha-1)!(\beta-1)!}{(\alpha+\beta-1)!}$$

As a result, the Beta distribution has a closed-form analytic solution given by:

$$\pi(\theta \mid \alpha,\beta) = \frac{(\alpha+\beta-1)!}{(\alpha-1)!(\beta-1)!}\theta^{\alpha-1}(1-\theta)^{\beta-1} \tag{B.6}$$

where $x,\theta \in \mathcal{R}$ and $\theta \in [0,1]$. The Beta distribution, see Section B.1.2, has the following pdf:

$$\pi\left(\theta \mid \alpha, \beta\right) \stackrel{\Delta}{=} \frac{1}{B(\alpha,\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1} \tag{B.7}$$

Where $\alpha,\beta > 0$, and $B(\alpha,\beta)$ is called the Beta function and is a normalizing constant ensuring the pdf has a total probability of 1. The Beta function is written as:

$$B(\alpha,\beta) = \int_0^1 \theta^{\alpha-1}\left(1-\theta\right)^{\beta-1} d\theta \tag{B.8}$$

In cases where $\alpha,\beta \in \mathcal{N}$, as in Bernoulli trials, the Beta function takes the following form:

$$B(\alpha,\beta) = \frac{(\alpha+\beta-1)!}{(\alpha-1)!(\beta-1)!}$$

As a result, the Beta pdf where $\alpha, \beta \in \mathcal{N}$, is written as:

$$\pi(\theta \mid \alpha, \beta) = \frac{(\alpha+\beta-1)!}{(\alpha-1)!(\beta-1)!}\theta^{\alpha-1}(1-\theta)^{\beta-1} \tag{B.9}$$

B.1.3 Bernoulli Distribution

The Bernoulli Distribution is for an experiment with a boolean outcome. It is assigns the probability distribution for the random variable with value 1 with probability p, and with value 0 with probability $q = 1 - p$.

B.2 Law of Large Numbers

Consider $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, ..., \mathbf{x}_n$ as independent random variables from the same underlying probability distribution. Then they are called Independent and Identically Distributed (i.i.d.), and their average is defined by:

$$\overline{\mathbf{x}}_n = \frac{\mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 + ... + \mathbf{x}_n}{n} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i \qquad , n \to \infty$$

Definition B.2: Law of Large Numbers

As the number of samples, n, increases, the sample mean, $\overline{\mathbf{x}}_n$, will converge to the actual mean, μ.

Formal statement of the Law of Large Numbers:

Theorem B.1: Law of Large Numbers

Suppose $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, ..., \mathbf{x}_n$ are i.i.d. with mean μ and with finite variance, where $\sigma^2 \in \mathbb{R}^+$. And let $\overline{\mathbf{x}}_n$ denote the sample average.Then as $n \to \infty$:
$\overline{\mathbf{x}}_n \to \mu$, with probability 1.

Remark B.3

$\overline{\mathbf{x}}_n$ is also a random number.

B.3 Central Limit Theorem

Definition B.3: Central Limit Theorem

For any distribution with a finite *variance*, σ^2, for a large sample size n, the *distribution* of $\overline{\mathbf{x}}_n$ converges to the Gaussian distribution.

Formal statement of the Central Limit Theorem:

Theorem B.2: Central Limit Theorem

Suppose $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, ..., \mathbf{x}_n$ are i.i.d. with mean μ and with finite variance. And let $\overline{\mathbf{x}}_n$ denote the sample average. Then as $n \to \infty$:

$$\overline{\mathbf{x}}_n \sim \mathcal{N}(\mu, \sigma^2) \tag{B.10}$$

C | Code

Experiments

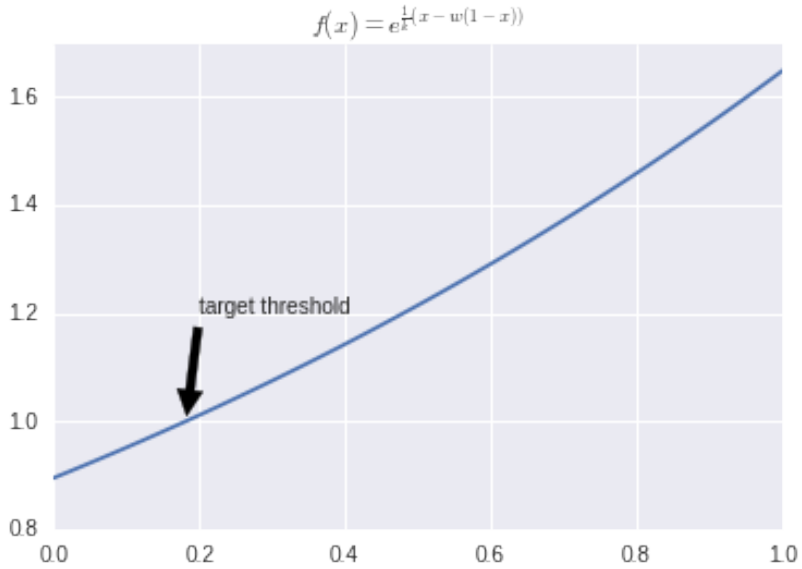
January 15, 2018

In [1]:
```
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
import seaborn as sns
import scipy as sp
import scipy.stats as ss
import math
import random
import numba
from collections import namedtuple
```

In [2]:
```
plt.style.use('classic') #'seaborn-whitegrid'
sns.set()
%precision 4
%matplotlib inline
# To reload modules
import importlib
```

In [3]:
```
from FileHandling import save_state, inspect_state, save_run_data, save_comparison
```

In [4]:
```
from MH_Sampling import acceptance_decision, MH_sampler, L_MH_sampler, init_MH_pars
#from SG_MH_Sampling import SG_MH_sampler, init_SG_MH_pars
from AM_Sampling import AM_sampler, init_AM_pars
from CMA_Sampling import CMA_sampler, init_CMA_pars
from GaA_Sampling import C_GaA_sampler, Q_GaA_sampler, init_GaA_pars
```




```
In [5]: from TestSuite import generate_state_space, generate_iid_samples, \
            generate_initial_states, generate_Gaussian, get_distribution, get_samples
```

1 The Samplers

```
In [6]: SAMPLERS = {'MH':dict(Name='MH', Function=MH_sampler,
                               InitParameters=init_MH_pars,
                               Color='red'),
                   'L_MH':dict(Name='L_MH', Function=L_MH_sampler,
                               InitParameters=init_MH_pars,
                               Color='brown'),
                   'AM':dict(Name='AM',
                             Function=AM_sampler,
                             InitParameters=init_AM_pars,
                             Color='blue'),
                   'CMA':dict(Name='CMA',
                              Function=CMA_sampler,
                              InitParameters=init_CMA_pars,
                              Color='green'),
                   'GaA':dict(Name='GaA',
                              Function=Q_GaA_sampler,
                              InitParameters=init_GaA_pars,
                              Color='yellow')}
```

```
        #'SG_MH':dict(Name='SG_MH', Function=SG_MH_sampler, InitParameters=init_SG_MH_pars, Colo
```

```
In [7]: def get_sampler(name):
            global SAMPLERS
            return SAMPLERS[name]
```

```
In [8]: def create_data_store(dim, N):
            return dict(States=np.zeros(shape=(N, dim)),
                        Densities=np.zeros(N),
                        Accepted_p=np.zeros(N, dtype=bool),
                        Means=np.zeros(shape=(N, dim)),
                        # Covariance matrix or its Cholesky decomposition or square root
                        Covariances=np.zeros((N, dim, dim)),
                        # The parameter r in Gaussian Adaption or sigma in CMA-ES is stored in
                        Scales=np.zeros(N),
                        # The parameter theta in Gaussian Adaption is stored in
                        Thresholds=np.zeros(N),
                        # Done is false when created and becomes true when the run is finished
                        Done=False,
                        # ChainLength is 0 when created and is set to the effective chain length
                        # when the run is finished.
                        ChainLength=0)
```

2 Run *Specification*

A **run** is specified by the * state space **sp**, * number of samples **N** we want to generate, * **name** of the **target distribution**, and * **sampler's name**, * **index** of that run.

```
In [9]: RunData = namedtuple('RunData',
                             ['StateSpace', 'N', 'Target', 'Sampler', 'RunIdx', 'DataStore'])
```

```
In [10]: def specify_run(dim, N, name_target, name_sampler, run_idx):
             sp = generate_state_space(dim=dim)
             generate_iid_samples(sp=sp, N=N)
             target = get_distribution(sp=sp, name=name_target)
             sampler = get_sampler(name=name_sampler)
             data_store = create_data_store(dim=dim, N=N)
             return RunData(StateSpace=sp,
                            N=N,
                            Target=target,
                            Sampler=sampler,
                            RunIdx=run_idx,
                            DataStore=data_store)
```

```
In [11]: def get_states(run_data):
             return run_data.DataStore['States']
```

```
def get_contour_function_and_levels(run_data):
    target = run_data.Target
    return target['Contour Function'], target['Contour Levels']
```

3 Run *Execution*

```
In [12]: def execute_run(run_data):
             sp, sampler, N, run_idx = \
                 run_data.StateSpace, run_data.Sampler, run_data.N, run_data.RunIdx
             initial_states = sp['Initial States']
             return sampler['Function'](pars=sampler['InitParameters'](sp=sp),
                                        target=run_data.Target,
                                        initial_state=initial_states[run_idx],
                                        run_data=run_data)
```

3.1 This is how you execute a run

3.1.1 Specification

- Specify the dimension of the state space, i.e. the value of the parameter *dim*
- Specify the total number N of samples to be generated
- Give the name of the target, i.e. the value of the parameter *name_target*
- Give the name of the sampler, i.e. the value of the parameter *name_sampler*
- The argument *run_idx* refers to the index of this run

3.1.2 Execution

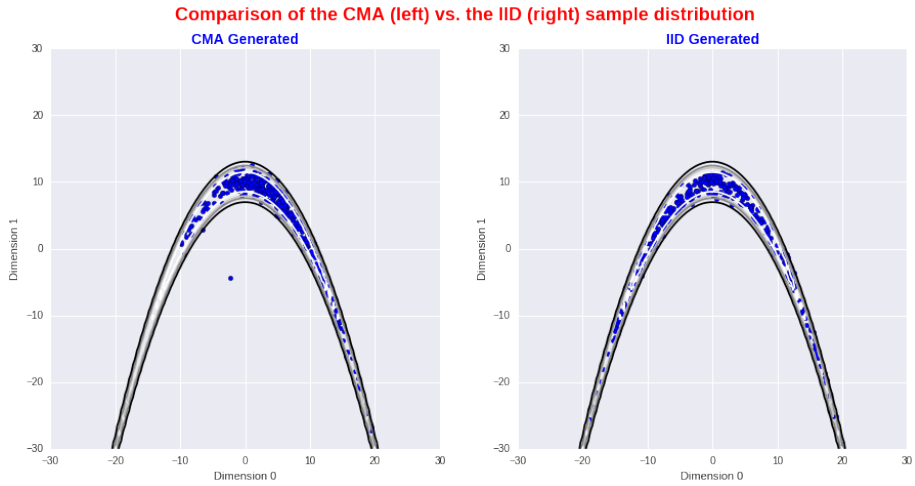
- Execute a run using the given specification. It returns a datastore that contains all data related to that run and is used in the visualization of that run.

```
In [13]: SPEC = specify_run(dim=11, N=60000, name_target='Pi_4', name_sampler='CMA', run_idx=0)
         DATA = execute_run(SPEC)
```

4 Visualization of a run

```
In [14]: from Visualization import GRID, plot_samples, compare_to_iid_samples, plot_contour_line

In [15]: compare_to_iid_samples(run_data=DATA, dim1=0, dim2=1, nb_samples=350, burnin_pct=50)
```

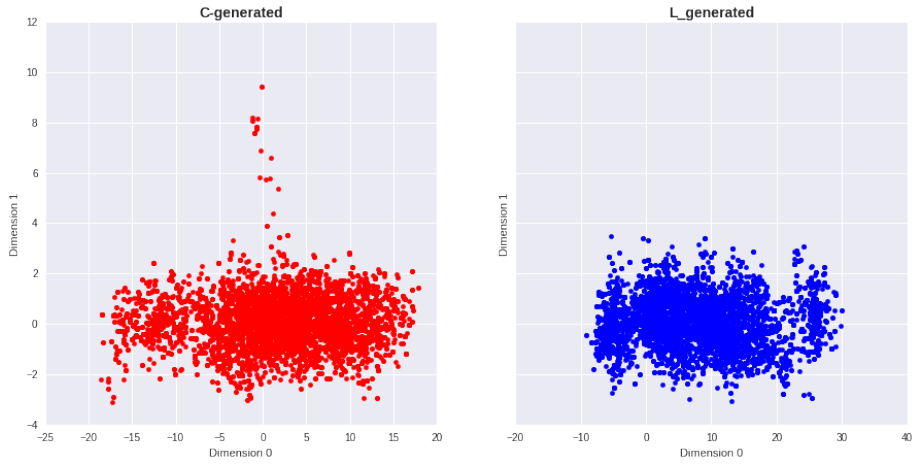
```
Burn in used is 50 percent of the generated samples.
CMA Fractions: 0.00003, 0.00230, 0.00583, and 0.03387
IID Fractions: 0.00003, 0.00327, 0.00920, and 0.03667
```



5 Time to Test

New we compare two different implementations of Metropolis-Hastings: one that uses the covariance matrix C to generate the candidates and one that uses the Cholesky factor L to genereate the candidates. The results should be similar when visualized.

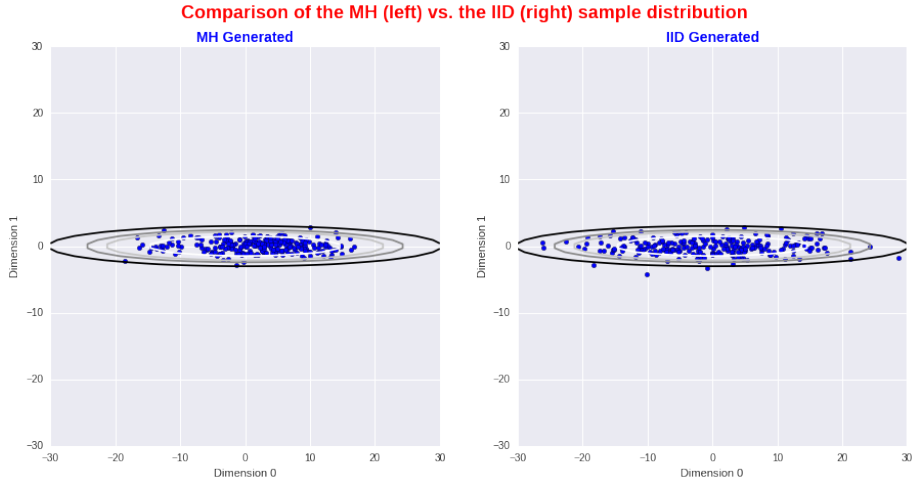
```
In [16]: def test_L_variant(dim, N, name_target, name_sampler, run_idx=0):
             '''Compares the C- and L-variant of the sampler named s_name on target with name t_
             dim is the dimension of the state space and N is the number of samples generated.''
             # Specify the run for both variants.
             sampler_spec = specify_run(dim, N, name_target, name_sampler, run_idx)
             l_sampler_spec = specify_run(dim, N, name_target, 'L_'+name_sampler, run_idx)

             #Execute with the given specifications
             sampler_data, l_sampler_data = execute_run(sampler_spec), execute_run(l_sampler_spe

             #Get the samples
             samples, l_samples = sampler_data.DataStore['States'], l_sampler_data.DataStore['St
             return sampler_data, l_sampler_data

In [17]: MH_DATA, L_MH_DATA = test_L_variant(dim=10, N=10000, name_target='Pi_1', name_sampler='

In [18]: compare_sample_spread(dim1=0, dim2=1,
                               list_of_samples=[get_states(MH_DATA), get_states(L_MH_DATA)],
                               colors=['red', 'blue'], titles=['C-generated', 'L_generated'])
```



5.1 Plot MCMC samples and compare them to i.i.d samples

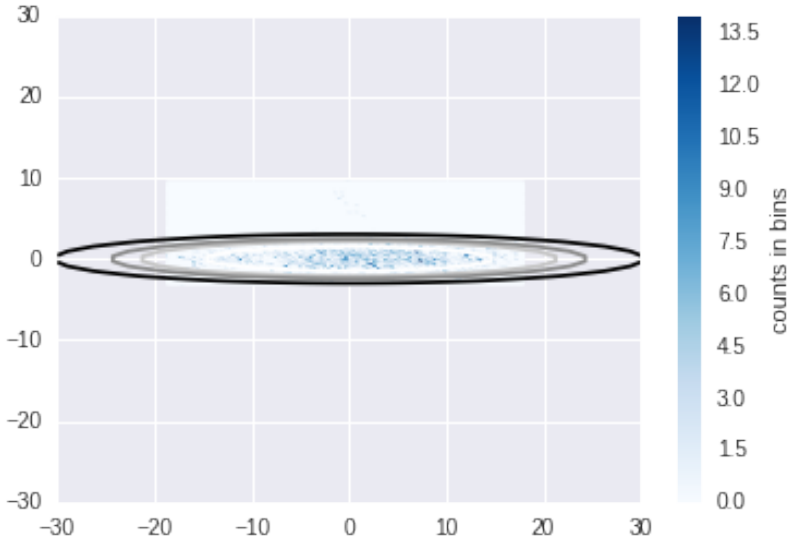
```
In [19]: compare_to_iid_samples(run_data=MH_DATA, dim1=0, dim2=1, nb_samples=250, burnin_pct=50)
```

```
Burn in used is 50 percent of the generated samples.
MH Fractions: 0.00400, 0.10440, 0.22580, and 0.54940
IID Fractions: 0.00420, 0.08260, 0.17420, and 0.47920
```



```
In [20]: def heath_map(run_data, dim1=0, dim2=1):
             global GRID
             states = run_data.DataStore['States']
             x_components, y_components = states[:,dim1], states[:,dim2]
             contour_function, levels = get_contour_function_and_levels(run_data)
             X, Y = GRID.X, GRID.Y
             Z = contour_function(X, Y)
             plt.hexbin(x_components, y_components, bins=15, cmap='Blues')
             cb = plt.colorbar()
             cb.set_label('counts in bins');
             plt.contour(X, Y, Z, levels)
```

```
In [21]: heath_map(MH_DATA)
```



6 Plot several *traces*

The following **traces** are plotted

1. target values,
2. acceptance ratio
3. trace of one of the components of the state, and
4. in the future, the condition number of the 'covariances'

```
In [22]: import numba
```

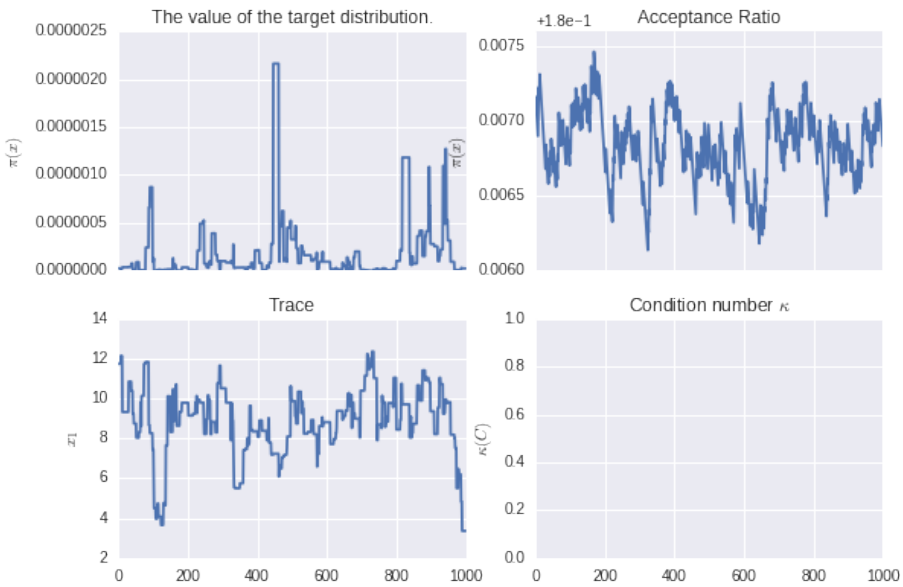
In [23]:
```
@numba.jit
def Euclidean_distance(x_1, x_2):
    return np.sqrt(np.sum((x_1 - x_2)**2))
```

In [24]:
```
def get_life_data(run_data):
    ds = run_data.DataStore
    return ds['States'], ds['Densities'], ds['Accepted_p'], ds['Covariances']
```

In [25]:
```
def running_avg(seq):
    return np.cumsum(seq)/np.arange(1, len(seq)+1)
```

In [26]:
```
def plot_traces(run_data, dim=0, resolution=1, start=0, stop=None, step=1):
    # The information to visualize
    ds = run_data.DataStore
    states, values, accepted_p, covariances = get_life_data(run_data)
    #fig, axes = plt.subplots(2, 2, sharex=True)
    #fig = plt.figure(figsize=(40, 40))
    #fig.subplots_adjust(hspace=10.0)

    fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, sharex=True, figsize=(9, 6))

    ax1.plot(values[start:stop:step])
    ax1.set_title('The value of the target distribution.')
    ax1.set_ylabel('$\pi(x)$')

    ax2.plot(running_avg(accepted_p)[start:stop:step])
    ax2.set_title('Acceptance Ratio')
    ax2.set_ylabel('$\pi(x)$')

    ax3.plot(states[start:stop:step, dim])
    ax3.set_title('Trace')
    ax3.set_ylabel('$x_{}$'.format(dim))

    #ax4.plot(la.cond(covariances)[start:stop:step])
    ax4.set_title('Condition number $\kappa$')
    ax4.set_ylabel('$\kappa(C)$')

    fig.subplots_adjust(left=0.125, right=0.9, bottom=0.1,
                        top=0.9, wspace=0.2, hspace=0.2)

    #fig.tight_layout()
```

In [27]:
```
plot_traces(run_data=DATA, start=5000, stop=6000, dim=1)
```



7 Next we can *compare all samplers*

```
In [28]: def compare_samplers(name_target, dim, N, run_idx=0):
             global SAMPLERS
             data = []
             for name_sampler, sampler in SAMPLERS.items():
                 dim, N, name_target, name_sampler, run_idx
                 run_data = specify_run(dim, N, name_target, name_sampler, run_idx)
                 data.append(execute_run(run_data))
             return data
```

```
In [29]: def plot_comparison(samplers_data):
             [plot_samples(run_data=data, nb_samples=100, dim1=0, dim2=1, burnin_pct=50)
              for data in combined_data]
```

```
In [30]: SAMPLERS_DATA = compare_samplers(name_target='Pi_4', dim=2, N=10000)
```

```
---------------------------------------------------------------------------

LinAlgError                               Traceback (most recent call last)

<ipython-input-30-6e0b38864000> in <module>()
----> 1 SAMPLERS_DATA = compare_samplers(name_target='Pi_4', dim=2, N=10000)
```

```
    <ipython-input-28-2f280fd1ad49> in compare_samplers(name_target, dim, N, run_idx)
      5         dim, N, name_target, name_sampler, run_idx
      6         run_data = specify_run(dim, N, name_target, name_sampler, run_idx)
----> 7         data.append(execute_run(run_data))
      8     return data


    <ipython-input-12-f5824f803765> in execute_run(run_data)
      5                                 target=run_data.Target,
      6                                 initial_state=initial_states[run_idx],
----> 7                                 run_data=run_data)


    /home/philip/Desktop/Adaptive_Sampling/code/4PK/4PK/AM_Sampling.py in AM_sampler(pars, t
    161                     state=current, value=target_pdf(current),
    162                     mean=mean, covariance=M2, accepted_p=accepted)
--> 163         candidate = generate_AM_candidate(current=current, M2=M2, n=n, pars=pars)
    164         accepted = acceptance_decision(current=current, proposed=candidate, pdf=targ
    165         if accepted:


    /home/philip/Desktop/Adaptive_Sampling/code/4PK/4PK/AM_Sampling.py in generate_AM_candid
     98 def generate_AM_candidate(current, M2, n, pars):
     99     prop_cov = get_proposal_cov(M2, n, pars)
--> 100     candidate = ss.multivariate_normal(mean=current, cov=prop_cov).rvs()
    101     return candidate
    102


    /home/philip/anaconda3/lib/python3.6/site-packages/scipy/stats/_multivariate.py in __cal
    349         return multivariate_normal_frozen(mean, cov,
    350                                           allow_singular=allow_singular,
--> 351                                           seed=seed)
    352
    353     def _process_parameters(self, dim, mean, cov):


    /home/philip/anaconda3/lib/python3.6/site-packages/scipy/stats/_multivariate.py in __ini
    599         self.dim, self.mean, self.cov = self._dist._process_parameters(
    600                                                             None, mean, cov)
--> 601         self.cov_info = _PSD(self.cov, allow_singular=allow_singular)
    602
    603     def logpdf(self, x):


    /home/philip/anaconda3/lib/python3.6/site-packages/scipy/stats/_multivariate.py in __ini
```

```
    155         d = s[s > eps]
    156         if len(d) < len(s) and not allow_singular:
--> 157             raise np.linalg.LinAlgError('singular matrix')
    158         s_pinv = _pinv_1d(s, eps)
    159         U = np.multiply(u, np.sqrt(s_pinv))


    LinAlgError: singular matrix
```

8 Compare the *autocorrelation* functions

```
In [ ]: def autocorrelation(series, max_lag=1000):
            x = series - np.mean(series)
            xnorm = np.sum(x**2)
            autocor = np.correlate(x, x, "same")/xnorm
            # use only second half
            origin = int(np.floor(len(autocor)/2))
            return autocor[origin:origin+max_lag]

In [ ]: def plot_autocorrelation(autocorrelations, color, label):
            plt.plot(autocorrelations, color=color, label=label)

In [ ]: def compare_autocorrelations(samplers_data, dim):
            for run_data in samplers_data:
                series = run_data.DataStore['States'][:, dim]
                autocor = autocorrelation(series, max_lag=400)
                s_name, color = run_data.Sampler['Name'], run_data.Sampler['Color']
                plt.plot(autocor, color=color, label=s_name)
            #finally plot the iid samples.
            #idd_samples = combined_data[0].Target['Samples'][:, dim]
            d = samplers_data[1].StateSpace['dim']
            #autocor = autocorrelation(idd_samples, max_lag=400)
            #plt.plot(autocor, 'black', label='I.D.D')

            # Complete the graph
            plt.title('Autocorrelation function for the %s-dimensional target $\pi_4(\mathbf{x})
            plt.xlabel('Lag $k$')
            plt.ylabel('Autocorrelation in dim. %s'%(dim))
            plt.legend()
            #plt.legend(('MH', 'AM','CMA', 'GaA', 'I.I.D.'), loc='best', prop={'size':10})

In [ ]: compare_autocorrelations(samplers_data=SAMPLERS_DATA, dim=0)

In [ ]: #plot_comparison(SAMPLERS_DATA)
```

CMA_Sampling

January 15, 2018

1 CMA *Sampling*

```
In [ ]: import numpy as np
        import numpy.linalg as la
        import matplotlib.pyplot as plt
        import scipy as sp
        import scipy.stats as ss
        import math
        import random
        from collections import namedtuple
        #%precision 4
        #%matplotlib inline
```

from importlib import reload reload(util)

```
In [ ]: from MH_Sampling import acceptance_decision
        from FileHandling import save_state
        from TestSuite import generate_state_space, generate_iid_samples, get_samples
```

2 CMA Sampler

Both the CMA-ES optimization and sampling algorithm have a number of **strategy parameters** that do not change during the execution of the algorithm. We are using the default values as recommended in the paper *C. Igel, T. Suttorp, and N. Hansen, A Computational Efficient Covariance Matrix Update and a* $(1+1)$*-CMA for Evolution Strategies*. henceforth called *the paper*.

In the update of the **global scale** σ, the following parameters with given initial values are used

- **damping parameter**

  $$k = 1 + \frac{d}{2}$$

  where d is the dimension of the state space. (**Note**: In the paper, d is used for the damping parameter instead of k and n for dimension instead of d)
- **target success rate**

  $$p_s^{succ} = \frac{2}{11}$$

- **learning rate**

$$\lambda_p = \frac{1}{12}$$

used in the update of the average success rate $\bar{p}_s \in [0, 1]$, cf. the procedure *update_scale* below for more information.

For the **covariance matrix adaptation**, they are

- **evolution point weight**

$$\lambda_{\mathbf{p}} = \frac{2}{d+2}$$

- **covariance matrix weight**

$$\lambda_{\mathbf{C}} = \frac{2}{d^2+6}$$

- **threshold**

$$\theta_p$$

for *average success rate* $\bar{p}_s$. The update of the evolution point $\mathbf{p}_c$ and the covariance matrix $\mathbf{C}$ depend on the test $\bar{p}_s < \theta_p$, cf. the procedure *update_cov* below for more information.

Note: in the CMA ES literature, step size is used instead of global scale. In order to be consistent with the MCMC literature we prefer and use the latter.

3 Initial values

The initial values are set as follows

- **average success rate** $\bar{p}_s = p_s^{succ}$ where p_s^{succ} is the *target success rate*.
- **evolution point** $\mathbf{p}_c = \mathbf{0}$
- **covariance matrix** $\mathbf{C} = \mathbb{1}_d$

The choice of the initial candidate $\mathbf{x}_0$ and the initial global scale σ are problem dependent. Here, we initialize $\mathbf{x}_0$ with a random point in a hypercube centered in the origin. Its side can vary. And $\sigma = 1$.

```
In [ ]: CMA_Parameters = namedtuple('CMA_Parameters',
                                    ['z_samples',
                                    #Parameters used in the global scale control
                                     's', 'k', 't_succ', 'c_p',
                                     #Parameters used in the covariance adaptation
                                     'c_c', 'c_cov', 'p_thres'])

        def init_CMA_pars(sp):
            dim = sp['dim']
            return CMA_Parameters(z_samples=get_samples(sp=sp, name='Z'),
                                  s=1,
                                  k=1+dim/2,
                                  t_succ=2/11,
```

GaA_Sampling

January 15, 2018

1 Gaussian Adaptation *Sampling*

```
In [ ]: import numpy as np
        import numpy.linalg as la
        import matplotlib.pyplot as plt
        import scipy as sp
        import scipy.stats as ss
        import math
        import random
        from collections import namedtuple
        #%precision 4
        #%matplotlib inline
```

```
In [ ]: from MH_Sampling import acceptance_decision
        from FileHandling import save_state
        from TestSuite import generate_state_space, generate_iid_samples, get_distribution, get_
```

To make sure that we have the module Utilities after we have *changed* it, we have to *reload* it.
from importlib import reload reload(ut)

2 *Maximum Entropy* Principle

To be done.

The entropy of the multivariate normal distribution $N(\mathbf{m}, \mathbf{C})$ with mean $\mathbf{m}$ and covariance matrix $\mathbf{C}$ is

$$H(N) = \ln \sqrt{(2\pi e)^n \det \mathbf{C}}$$

```
In [ ]: def entropy(cov):
            dim1, dim2 = cov.shape
            assert dim1==dim2
            return math.log(np.sqrt((2* math.pi * math.e)**dim1 * la.det(cov)))
```

2.1 Check Covariance Matrix

```
In [ ]: def analyse_cov(cov):
            eigenvals, eigenvecs = la.eig(cov)
            print('Covariance Matrix:\n', cov)
            print('Determinant:', la.det(cov))
            print('Eigenvalues:', eigenvals)
            print('Eigenvectors:', eigenvecs)
            #print('Inner Product between eigenvectors:', eigenvec_1.T @ eigenvec_2)
            print('Symmetric:', np.allclose(cov, cov.T))
```

3 Gaussian Adaptation according to Mueller's Matlab code

This notebook is based on Mueller's Matlab code and the paper *"Gaussian Adaptation as a unifying framework for black-box optimization and adaptive Monte Carlo sampling"* by *Christian I. Muellen* and *Ivo F. Sbalzarini*.

3.1 Strategy Parameters

The **strategy parameters** are: - the **acceptance ratio** P - the **expansion** $f_e > 1$ and **contracton factor** $f_c < 1$ used to update the global scale σ - the **weigths** $\lambda_{\mathbf{m}}$, $\lambda_{\mathbf{C}}$, and λ_θ used to update the mean $\mathbf{m}$, the covariance matrix $\mathbf{C}$, and the threshold θ, respectively.

They are **initialized** as follows, cf. p.2 of the MATLAB code of Mueller:

- **acceptance ratio** $P = \frac{1}{e}$
- **expansion factor** $f_e = 1 + \beta(1 - P)$ and **contraction factor** $f_c = 1 - \beta P$ where $\beta = \lambda_{\mathbf{C}}$
- **weights** are initialized as follows
- $\lambda_{\mathbf{C}} = \frac{\ln(d+1)}{(d+1)^2}$
- $\lambda_{\mathbf{m}} = \frac{1}{ed}$
- $\lambda_\theta = \frac{1}{ed}$ without restart, cf. the end of Section II.B of the paper what to do in case of restart.

Here, d is the dimension of the **search space** in case of **optimization** or the **state space** in case of **sampling**.

3.1.1 Initializing the *strategy parameters*

Cf. above for their initial values.

```
In [ ]: GaA_Pars = namedtuple('GaA_Pars',
                              ['l_C', 'l_m', 'b', 'P',
                               'f_e', 'f_c', 'max_scale', 'max_cond',
                               'Origin', 'Id'])
```

```
In [ ]: def init_GaA_pars(sp):
            D, origin, identity = sp['dim'], sp['Origin'], sp['Id']
            tmp_l_c = math.log(D+1)/(D + 1)**2
            tmp_P = 1/math.e
            return GaA_Pars(l_C=tmp_l_c,
                            l_m=1/(math.e*D),
                            b=tmp_l_c,
                            P=tmp_P,
                            f_e=1 + tmp_l_c*(1-tmp_P),
                            f_c=1 - tmp_l_c*tmp_P,
                            max_scale=1000,
                            max_cond=80, # he value used by Mueller is 1e6*D but this results in
                            Origin=origin,
                            Id=identity)
```

```
In [ ]: def display_parameters(pars):
            str_1 = "l_C: {:1.4f}\nl_m: {:1.4f}\nb: {:1.4f}\nP: {:1.4f}"
            str_2 = "\nf_e: {:1.4f}\nf_c: {:1.4f}\nmax_scale: {:1.4f}\nmax_cond: {:1.4f}"
            pars_info_1 = str_1.format(pars.l_C, pars.l_m, pars.b, pars.P)
            pars_info_2 = str_2.format(pars.f_e, pars.f_c, pars.max_scale, pars.max_cond)
            print( pars_info_1,  pars_info_2)
```

4 Generate next sample using Q

The new state $\mathbf{x}_{n+1}$ is generated as follows

$$\mathbf{x}_{n+1} = \mathbf{m}_n + \sigma_n \mathbf{Q}_n \mathbf{z}_n$$

where σ_n is the global scale, $\mathbf{Q}_n$ is the "square root" of the covariance matrix $\mathbf{C}_n$ as defined below, and $\mathbf{z}_n$ is a sample of the multivariate standard normal distribution $N(0,1)$ generated at step n.

```
In [ ]: def Q_generate_GaA_candidate(mean, scale, Q, z_sample):
            # This function uses the normalized sqrt Q of the covariance matrix C.
            # cf. p.7 of the MATLAB code of Mueller
            x = mean + scale*(Q @ z_sample.T)
            return x
```

5 Generate next sample using C

```
In [ ]: def C_generate_GaA_candidate(mean, C, z_sample):
            return mean + C @ z_sample.T
```

6 Updates of the *scale* σ, the *mean* $\mathbf{m}$, and the *threshold* θ

6.1 Update of the scale σ

The **scale** is *updated at each step*:

When the new sample is **accepted** then the scale is **increased**:

$$\sigma_{n+1} = f_e \sigma_n$$

where $f_e > 1$ is the *expansion factor*, one of the strategy parameters of Gaussion Adaptation.

When the sample is **rejected** then the scale is **decreased**:

$$\sigma_{n+1} = f_c \sigma_n$$

where $f_c < 1$ is the *contraction factor*, another strategy parameter.

```
In [ ]: def contract(scale, pars):
            return pars.f_c*scale

        def expand(scale, pars):
            # cf. p.10 of the MATLAB code of Mueller
            f_e, max_scale = pars.f_e, pars.max_scale
            next_scale = f_e*scale
            if next_scale <= max_scale:
                return next_scale
            else:
                return max_scale
```

6.2 Update of the *mean* m

These are

The **mean** is **only updated** when the new sample $\mathbf{x}_{n+1}$ is **accepted**. Th new mean is

$$\mathbf{m}_{n+1} = (1 - \lambda_{\mathbf{m}})\mathbf{m}_n + \lambda_{\mathbf{m}}\mathbf{x}_n$$

Here, $\lambda_{\mathbf{m}}$ and $\lambda_{\mathbf{C}}$ are *strategy parameters* of *Gaussian Adaptation*.

```
In [ ]: # This code is for global optimization, NOT sampling.
        def GaA_mean_update_2(mean, sample, pars):
            l_m = pars.l_m
            return (1-l_m)*mean + l_m*sample
```

```
In [ ]: # In case of sampling l_m = 1, in other words the new sample becomes the next mean.
        def GaA_mean_update(mean, sample, pars):
            return sample
```

7 Update of the covariance matrices C and Q

C and **Q** are covariance matrices and therefore positive definite and symmetric. Symmetry might get lost due to rounding off errors in the update process. After each update we make sure that the result is still symmetric.

The first way to do this uses the *Numpy*-function *triu* that returns the upper triangle part of a matrix. The second one uses the *transpose* of a matrix. Recall that $\mathbf{S} = \mathbf{S}^\top$ for a symmetric matrix **S**.

```
In [ ]: def trui_enforce_symmetry(cov):
            dim1, dim2 = cov.shape
            assert dim1==dim2
            return np.triu(cov,0)+np.triu(cov,1).T

        def transpose_enforce_symmetry(cov):
            dim1, dim2 = cov.shape
            assert dim1==dim2
            return 1/2*(cov+cov.T)
```

8 Update of the "*square root*" Q of the *covariance* matrix C

First, we calculate $\Delta\mathbf{C}_n$ as follows

$$\Delta\mathbf{C}_{n+1} = (1-\lambda_\mathbf{C})\mathbb{1}_d + \lambda_\mathbf{C}\mathbf{z}_n\mathbf{z}_n^\top$$

where $\mathbb{1}_d$ is the identity matrix, $\mathbf{z}_n$ is the nth sample of the multivariate standard Gaussian distribution, and $\lambda_\mathbf{C}$ is the strategy parameter used in the update of the covariance matrix $\mathbf{C}$.

```
In [ ]: def delta_C(z_sample, pars):
            l_C = pars.l_C
            identity = pars.Id
            deltaC = (1-l_C)*identity + l_C*np.outer(z_sample, z_sample)
            #return enforce_symmetry(deltaC)
            return deltaC
```

Next, we define $\Delta\mathbf{Q}_{n+1}$ as

$$\Delta\mathbf{Q}_{n+1} \triangleq \sqrt{\Delta\mathbf{C}_{n+1}}$$

```
In [ ]: def sqrtm(cov):
            D, B = la.eigh(cov)
            sqrtD = np.diag(np.sqrt(D))
            # Return the sqrt Q of the matrix C
            return B @ sqrtD @ B.T
```

Finally, we calculate $\mathbf{Q}_{n+1}$ as

$$\mathbf{Q}_{n+1} = \mathbf{Q}_n\Delta\mathbf{Q}_{n+1}$$

```
In [ ]: def normalize(cov):
            D, _ = cov.shape
            normalization_constant = la.det(cov)**(1/D)
            normalized_cov = cov/normalization_constant
            #det = la.det(normalized_cov)
            #np.testing.assert_almost_equal(det, 1.0)
            return normalized_cov
```

```
In [ ]: def GaA_Q_update(z_sample, Q, pars):
            max_cond = pars.max_cond
            deltaC = delta_C(z_sample, pars)
            deltaQ = sqrtm(deltaC)
            Q_next = normalize(transpose_enforce_symmetry(Q @ deltaQ))
            if la.cond(Q_next) <=  max_cond:
                return Q_next
            else:
                return Q
```

9 Update of the *covariance* matrix C

```
In [ ]: def GaA_C_update(C, mean, sample, pars):
            # Cf. p.10 of the MATLAB code of Mueller
            l_C, max_cond = pars.l_C, pars.max_cond
            delta = mean - sample
            C_next = (1 - l_C)*C + l_C*np.outer(delta, delta)
            if la.cond(C_next) <= max_cond:
                return C_next
            else:
                return C
```

10 Gaussian Adaptation Sampling

```
In [ ]: def Q_GaA_sampler(pars, target, initial_state, run_data):
            target_pdf, sp = target['pdf'], target['State Space']
            Origin, Id = sp['Origin'], sp['Id']

            ds, N = run_data.DataStore, run_data.N
            z_samples = get_samples(sp=sp, name='Z')

            #Set up and save the initial state
            m = x_current = initial_state
            sigma = 1
            Q = Id
            save_state(data_store=ds,
                       step=0,
                       state=x_current,
                       value=target_pdf(x_current),
                       accepted_p=True,
                       mean=m,
                       covariance=Q,
                       scale=sigma,
                       threshold=None)

            #Sample and save state
            for n in range(1, N):
```

```
            z_sample = z_samples[n]
            x_proposed = Q_generate_GaA_candidate(mean=x_current,
                                                  scale=sigma,
                                                  Q=Q,
                                                  z_sample=z_sample)
            accepted = acceptance_decision(x_current, x_proposed, target_pdf)
            if accepted:
                x_current = x_proposed
                sigma = expand(sigma, pars=pars)
                m = GaA_mean_update(mean=m, sample=x_proposed, pars=pars)
                Q = GaA_Q_update(Q=Q, z_sample=z_sample, pars=pars)
            else:
                sigma = contract(sigma, pars=pars)
            save_state(data_store=ds,
                       step=n,
                       state=x_current,
                       value=target_pdf(x_current),
                       accepted_p=accepted,
                       mean=m,
                       covariance=Q,
                       scale=sigma,
                       threshold=None)
        return run_data
```

In [ ]:
```
def C_GaA_sampler(pars, target, initial_state, run_data):
    target_pdf, sp = target['pdf'], target['State Space']
    Origin, Id = sp['Origin'], sp['Id']

    ds, N = run_data.DataStore, run_data.N
    z_samples = get_samples(sp=sp, name='Z')

    #Set up and save the initial state
    m = x_current = initial_state
    sigma = 1
    C = Id

    save_state(data_store=ds,
               step=0,
               state=x_current,
               value=target_pdf(x_current),
               accepted_p=True,
               mean=m,
               covariance=C,
               scale=sigma,
               threshold=None)

    #Sample and save state
    for n in range(1, N):
```

```
            z_sample = z_samples[n]
            x_proposed = C_generate_GaA_candidate(mean=x_current,
                                                  C=C,
                                                  z_sample=z_sample)
            accepted = acceptance_decision(x_current, x_proposed, target_pdf)
            if accepted:
                x_current = x_proposed
                sigma = expand(sigma, pars=pars)
                m = GaA_mean_update(mean=m, sample=x_proposed, pars=pars)
                C = GaA_C_update(C=C, mean=m, sample=x_proposed, pars=pars)
            else:
                sigma = contract(sigma, pars=pars)
            save_state(data_store=ds,
                       step=n,
                       state=x_current,
                       value=target_pdf(x_current),
                       accepted_p=accepted,
                       mean=m,
                       covariance=C,
                       scale=sigma,
                       threshold=None)
        return run_data
```

11 Time to Test

def tGaA_sampler(pars, target, initial_state, N): target_pdf, sp = target['pdf'], target['State Space'] Origin, Id = sp['Origin'], sp['Id'] z_samples = get_samples(sp=sp, name='Z')

```
#Set up and save the initial state
m = x_current = initial_state
sigma = 1
Q = Id

#Sample and save state
for n in range(1, N):
    z_sample = z_samples[n]
    x_proposed = Q_generate_GaA_candidate(mean=x_current, scale=sigma, Q=Q, z_sample=z_sample)
    accepted = acceptance_decision(x_current, x_proposed, target_pdf)
    if accepted:
        x_current = x_proposed
        sigma = expand(sigma, pars=pars)
        m = GaA_mean_update(mean=m, sample=x_proposed, pars=pars)
        Q = GaA_Q_update(Q=Q, z_sample=z_sample, pars=pars)
    else:
        sigma = contract(sigma, pars=pars)
return N
```

12 Test Section

N=10000 SP = generate_state_space(dim=5) generate_iid_samples(SP, N=N) tGaA_sampler(pars=init_GaA_pars(sp=SP), target=SP['Test Suite']['Pi_4'], initial_state=SP['Initial States'][0], N=N)

13 Extra Plotting Utilities

```
In [ ]: def plot(series):
            plt.plot(series)

        def distance_2_minimum(states, minimum):
            distances = np.linalg.norm(states - minimum, axis=1)
            return distances

        def scatter(states, means):
            plt.scatter(states[:,0], states[:,1], color='r')
            plt.scatter(means[:,0], means[:,1], color='b')
```

14 Utility to *rescue code* that was accidentally deleted.

```
In [ ]: def rescue_code(function):
            import inspect
            get_ipython().set_next_input("".join(inspect.getsourcelines(function)[0]))
```

CholeskyUpdatev2

January 15, 2018

```
In [ ]: import numpy as np
        import numpy.linalg as la
        import matplotlib.pyplot as plt
        import scipy as sp
        import scipy.stats as ss
        import math
        import random
        from collections import namedtuple
        %precision 4
        %matplotlib inline
```

1 Code optimization

In order to optimize the code import *numba*

```
In [ ]: import numba
```

2 Cholesky Update

Input: the lower triangular Cholesky factor L of the covariance matrix $\mathbf{\Sigma} \in \mathbb{R}^{n \times n}$, the vector $\boldsymbol{x} \in \mathbb{R}^n$, and their weights w_1 and w_2.

Output: the lower triangular Cholesky factor L' of $\mathbf{\Sigma}' = w_1\mathbf{\Sigma} + w_2\boldsymbol{x}^\top\boldsymbol{x}$

```
In [ ]: @numba.jit(nopython=True)
        def rank_1_update(L, u, alpha, beta):
            #assert alpha > 0, 'Argument alpha should be positive'
            #assert beta > 0, 'Argument beta should be positive'
            d = len(u)
            L = np.sqrt(alpha)*L  #Added
            b = 1
            nL = np.zeros_like(L)
            v = np.copy(u)  #Added
            for j in np.arange(d):
                nL[j,j] = np.sqrt(L[j,j]**2 + (beta/b)*(v[j]**2))
                gamma = b*L[j,j]**2 + beta*v[j]**2
                for k in range(j+1, d):
```

```
                v[k] = v[k] - (v[j]/L[j,j])*L[k,j]
                nL[k,j] = (nL[j,j]/L[j,j])*L[k,j] + (nL[j,j]*beta*v[j]/gamma)*v[k]
            b = b + beta*(v[j]**2/L[j,j]**2)
        return nL
```

2.1 Test rank 1 update

In [ ]:
```
def random_Gaussian(d):
    mean = np.random.random(size=d)
    X = np.random.random(size=(10*d, d))
    cov = X.T@X
    return mean, cov

def samples_random_Gaussian(d, N):
    m, C = random_Gaussian(d=d)
    samples = ss.multivariate_normal.rvs(mean=m, cov=C, size=N)
    return samples, m, C
```

In [ ]:
```
def check_rank_1_update(d, alpha, beta):
    # Create a random covariance matrix, i.e. C is symmetric and positive definite.
    m, C = random_Gaussian(d)
    L = la.cholesky(C)

    # Create a random vector
    v = np.random.random(size=d)

    # Update the covariance matrix
    uC = alpha*C + beta*np.outer(v,v)

    # Calculate its Cholesky factor
    uL = la.cholesky(uC)

    # Update the Cholesky factor of the initial covariance
    nL = rank_1_update(L=L, u=v, alpha=alpha, beta=beta)

    # The two return statements should be equivalent.
    return np.allclose(uC, nL@nL.T) and np.allclose(uL, nL)
```

In [ ]:
```
check_rank_1_update(d=100, alpha=100, beta=100)
```

In [ ]:
```
def test_rank_1_update(dimensions, alphas, betas, nbtrials=10):
    for d in dimensions:
        for alpha in alphas:
            for beta in betas:
                for n in range(nbtrials):
                    if not check_rank_1_update(d=d, alpha=alpha, beta=beta):
                        print('d: ', d, 'n: ', n)
    else:
        print('Pass')
```

```
In [ ]: base = np.array([2, 10, 100, 1000, 10000, 100000, 1000000, 1000000, 10000000])
        Alphas = np.array([(n-1)/n for n in base])
        Betas =  np.array([n/(n+1) for n in base])

In [ ]: test_rank_1_update(dimensions=[200],
                          alphas=Alphas,
                          betas=Betas)
```

3 Incrementally update the first two moments

```
In [ ]: def update_moments(mean, M2, sample, n):
            w = 1/(n+1)
            new_mean = mean + w*(sample - mean)
            delta_bf, delta_af = sample - mean, sample - new_mean
            new_M2 = M2 + np.outer(delta_bf, delta_af)
            return new_mean, new_M2

In [ ]: def calculate_moments(samples):
            N, d = samples.shape
            mean, M2 = np.zeros(d), np.zeros(shape=(d, d))
            for n in range(N):
                z_sample = samples[n]
                mean, M2 = update_moments(mean=mean, M2=M2, sample=z_sample, n=n)
            return mean, M2/(N-1)
```

3.1 Test incrementally update

The algorithms above have been tested up to dimension $d = 200$ of the statespace and $N = 1,000,000$ samples and each time they gave the same result, i.e. np.allclose returned True.

```
In [ ]: def test_moments(d, N):
            z_samples = ss.multivariate_normal.rvs(mean=np.zeros(d), cov=np.eye(d,d), size=N)
            moments_mean, moments_cov = calculate_moments(z_samples)
            emp_mean, emp_cov = np.mean(z_samples, axis=0), np.cov(z_samples, rowvar=False)
            return np.allclose(moments_mean, emp_mean) and np.allclose(moments_cov, emp_cov)

In [ ]: test_moments(d=200, N=100)
```

4 Compare incremental update of C and L

In case of the Cholesky update we - First, calculate the empirical covariance C of the first $2d$ generated samples - Next, we calculate its Cholesky factor L - From then on, L is updated.

```
In [ ]: @numba.jit
        def update_mean(samples):
            N, d = samples.shape
            initial_period = 2*d
            initial_cov = np.cov(samples[:initial_period], rowvar=False)
```

```
            initial_mean = np.mean(samples[:initial_period], axis=0)
            mean = initial_mean
            for n in range(initial_period, len(samples)):
                sample = samples[n]
                w = 1/(n+1)
                mean = (1-w)*mean + w*sample
            return mean
```

In [ ]:
```
@numba.jit
def update_L(samples):
    N, d = samples.shape
    initial_period = 2*d
    initial_cov = np.cov(samples[:initial_period], rowvar=False)
    initial_mean = np.mean(samples[:initial_period], axis=0)
    C = initial_cov
    L = la.cholesky(initial_cov)
    mean = initial_mean
    for n in range(initial_period, len(samples)):
        sample = samples[n]
        w = 1/(n+1)
        L = rank_1_update(L, sample-mean, alpha=(n-1)/n, beta=w)
        mean = (1-w)*mean + w*sample
    return L@L.T
```

5 Time to Test

In [ ]:
```
def test_Cholesky_update(d, N):
    # Generate samples of a Gaussian with random mean and covariance
    samples, mean, cov = samples_random_Gaussian(d=d, N=N)

    # Use numpy functions to calculate the sample mean and variance
    sample_mean, sample_cov = np.mean(samples, axis=0), np.cov(samples, rowvar=False)

    # Use the update functions 'updated_mean' and 'updated_cov' to calculate again the s
    updated_mean, updated_cov = update_mean(samples=samples), update_L(samples=samples)

    # Check whether they give the same results
    return np.allclose(sample_mean, updated_mean) and np.allclose(sample_cov, updated_co
```

In [ ]:
```
test_Cholesky_update(d=100, N=1000)
```

AM_Sampling

January 14, 2018

1 *Adaptive* MH

See the 1999 and 2001 papers of Haario et al.

```
In [ ]: import numpy as np
        import numpy.linalg as la
        import matplotlib.pyplot as plt
        import scipy as sp
        import scipy.stats as ss
        import math
        import random
        from collections import namedtuple
        #%precision 4
        #%matplotlib inline
```

from importlib import reload reload(ut)

```
In [ ]: from MH_Sampling import acceptance_decision
        from FileHandling import save_state
        from TestSuite import generate_state_space, generate_iid_samples, get_distribution, get_
```

```
In [ ]: AM_Pars = namedtuple('AM_Pars',
                            ['Origin', 'Id',
                             'sigma_0', 'sigma_opt',
                             'C_0', 'C_opt',
                             'z_samples'])

        def init_AM_pars(sp):
            dim, origin, idty, = sp['dim'], sp['Origin'], sp['Id'],
            sigma_0, sigma_opt = 0.1/np.sqrt(dim), sp['sigma_opt']
            cov_0, cov_opt = sigma_0**2*idty, sigma_opt**2*idty
            return AM_Pars(Origin=origin, Id=idty,
                           sigma_0=sigma_0, sigma_opt=sigma_opt,
                           C_0=cov_0, C_opt=cov_opt,
                           z_samples=get_samples(sp=sp, name='Z'))
```

2 Adaptive MH algorithm *AM*

2.1 Generate the candidate next sample

We consider a version of the *Adaptive Metropolis* (*AM*) sampler of Haario et al. (2001). We want to sample from the d-dimensional target distribution $\pi(\mathbf{x})$.

We perform a Metropolis algorithm with covariance matrix $\boldsymbol{Q}_n$ at iteration n given by

$$\mathbf{Q}_n(\mathbf{x},) = N(\mathbf{x}, \sigma_0^2 \mathbb{1}_d)$$

for $n \leq 2d$, while for $n > 2d$

$$\mathbf{Q}_n(\mathbf{x},) = (1\beta)N(\mathbf{x}, \sigma_{opt}^2 \mathbf{C}_n) + \beta N(\mathbf{x}, \sigma_0^2 \mathbb{1}_d)$$

where $\mathbf{C}_n$ is the current empirical estimate of the covariance of the target distribution based on the samples so far, $\sigma_0^2 = \frac{0.1^2}{d}$ and $\sigma_{opt}^2 = \frac{2.38^2}{d}$ are the initial and optimal scale, respectively, and β is a small positive constant, we use $\beta = 0.05$.

In other words, the next candidate is sampled from

$$\mathbf{x}^* \sim \mathbf{Q}_n(\mathbf{x},)$$

The text above is adapted from Section 2 of Gareth O. Roberts and Jeffrey S. Rosenthal (2008) *Examples of Adaptive MCMC*.

2.2 Random covariance matrix M from the above paper.

```
In [ ]: def get_proposal_cov(M2, n, pars, beta=0.05):
            d, _ = M2.shape
            init_period = 2*d
            s_0, s_opt, C_0 = pars.sigma_0, pars.sigma_opt, pars.C_0
            if np.random.rand()<=beta or n<= init_period:
                return C_0
            else:
                # We can always divide M2 by n-1 since n > init_period
                return (s_opt/(n - 1))*M2
```

```
In [ ]: def generate_AM_candidate(current, M2, n, pars):
            prop_cov = get_proposal_cov(M2, n, pars)
            candidate = ss.multivariate_normal(mean=current, cov=prop_cov).rvs()
            return candidate
```

2.3 Update the mean m and the the covariance C

In the *AM*-algorithm, the **mean** is updated as

$$\mathbf{m}_{n+1} = \frac{n}{n+1}\mathbf{m}_n + \frac{1}{n+1}\left(\mathbf{x}_{n+1} - \mathbf{m}_n\right)$$

and the **covariance** as

$$\mathbf{C}_{n+1} = \frac{n}{n+1}\mathbf{C}_n + \frac{1}{n+1}\left(\left(\mathbf{x}_{n+1} - \mathbf{m}_n\right)\left(\mathbf{x}_{n+1} - \mathbf{m}_n\right)^\top - \mathbf{C}_n\right)$$

where $\mathbf{x}_{n+1}$ is the sample generated at step $n+1$.
In the Welford algorithm,

$$M_n \triangleq \sum_{i=1}^{n} (x_i - \overline{x}_n)^2$$

or in other words

$$s_n^2 = \frac{M_n}{n-1}$$

It is easier to update M_n in a numerical stable way,

$$M_n = M_{n-1} + (x_n - \overline{x}_{n+1})(x_n - \overline{x}_n)^\top$$

```
In [ ]: def update_moments(mean, M2, sample, n):
            next_n = n + 1
            w = 1/next_n
            new_mean = mean + w*(sample - mean)
            delta_bf, delta_af = sample - mean, sample - new_mean
            new_M2 = M2 + np.outer(delta_bf, delta_af)
            return new_mean, new_M2, next_n
```

```
In [ ]: def multiple_of_10000(n):
            return n%10000 == 0
```

```
In [ ]: def AM_sampler(pars, target, initial_state, run_data):
            ds, N = run_data.DataStore, run_data.N
            target_pdf = target['pdf']

            current = initial_state
            mean, M2 = pars.Origin, np.zeros_like(pars.Id)
            accepted = True

            for n in range(0, N):
                save_state(data_store=ds, step=n,
                           state=current, value=target_pdf(current),
                           mean=mean, covariance=M2, accepted_p=accepted)
                candidate = generate_AM_candidate(current=current, M2=M2, n=n, pars=pars)
                accepted = acceptance_decision(current=current, proposed=candidate, pdf=target_p
                if accepted:
                    current = candidate
                # We always update M2, where S^2 = M2/n-1
                # whether the proposed samples are accepted or not
                mean, M2, n = update_moments(mean, M2, current, n)
            return run_data
```

3 Time to Test

def my_generate_candidate(current, cov): delta = ss.multivariate_normal(cov=cov).rvs() candidate = current + delta return candidate

def choose_cov(C_0, C_current): # We use beta = 0.05 for the bias of the Bernoulli distribution if np.random.binomial(n=1, p=0.05, size=None): return C_0 else: return C_current

def tAM_sampler(pars, target, initial_state, N): target_pdf = target['pdf'] current = initial_state Origin, Id = pars.Origin, pars.Id mean, M2 = Origin, np.zeros_like(Id) C_0, s_opt = pars.C_0, pars.sigma_opt d = len(initial_state) init_period = 2*d

```
for n in range(init_period):
    candidate = my_generate_candidate(current=current, cov=C_0)
    accepted = MH_decision(current=current, proposed=candidate, pdf=target_pdf)
    if accepted:
        current = candidate
    else:
        current = current
    mean, M2, n = update_moments(mean, M2, current, n)
for n in range(init_period, N):
    if multiple_of_10000(n):
        print('n:', n)
    C = (s_opt/(n - 1))*M2
    candidate = my_generate_candidate(current=current,
                                      cov=choose_cov(C_0=C_0, C_current=C))
    accepted = MH_decision(current=current, proposed=candidate, pdf=target_pdf)
    if accepted:
        current = candidate
    else:
        current = current
    mean, M2, n = update_moments(mean, M2, current, n)
return mean, M2/(n-1), n
```

d = 2 N = 10000 SP = generate_state_space(dim=d) generate_iid_samples(sp=SP, N=N)

mean, C, n = tAM_sampler(pars=init_AM_pars(sp=SP), target=get_distribution(sp=SP, name='Pi_2'), initial_state=SP['Initial States'][0], N=N)

MH_Sampling

January 15, 2018

1 *Metropolis-Hastings* Sampling

```
In [ ]: import numpy as np
        import numpy.linalg as la
        import matplotlib.pyplot as plt
        import scipy as sp
        import scipy.stats as ss
        import math
        import random
        from collections import namedtuple
        import numba
        #%precision 4
        #%matplotlib inline
```

```
In [ ]: from FileHandling import save_state
        from TestSuite import get_samples
```

To make sure that we have the module Utilities after we have *changed* it, we have to *reload* it.
from importlib import reload reload(ut)

```
In [ ]: def acceptance_decision(current, proposed, pdf):
            # Remark: 'accepted_p' includes the case where p_proposed > p_current
            # since u, a random number between 0 and 1, is then
            # always less than the ratio p_proposed/p_current
            # But for readability we make a distinction between the
            # between cases below.

            p_current, p_proposed = pdf(current), pdf(proposed)
            if p_current <= p_proposed:
                return True
            else:
                u = np.random.rand()
                return u <= p_proposed/p_current
```

2 Metropolis-Hasting decision when log-likelihood is used

compute a=loglikelihood(cc)+logprior(cc) draw pp from N(c;ji)N(c;ji) compute b==loglikelihood(pp)+logprior(pp) let H=min(1,exp(b-a)) and draw r from U(0,1) a=loglikelihood(cc)+logprior(cc) and b==loglikelihood(pp)+logprior(pp), it follows that exp(ba)=likelihood(p)Œprior(p)likelihood(c)Œprior(c),

```
In [ ]: def compose2(f, g):
            return lambda x: f(g(x))

        def likelihood_acceptance_decision(current, proposed, log_pdf):
            # Remark: 'accepted_p' includes the case where p_proposed > p_current
            # since u, a random number between 0 and 1, is then
            # always less than the ratio p_proposed/p_current
            # But for readability we make a distinction in the code below between the
            # two cases.

            p_current, p_proposed = log_pdf(current), log_pdf(proposed)
            if p_current <= p_proposed:
                return True
            else:
                u = np.random.rand()
                return u <= p_proposed/p_current
```

2.1 Proposal Distribution

Proposal_Distr is a named tuple *Distribution*, cf. the notebook *Utilities*, and the field *Samples* contains N samples of the proposal distribution. They are generated when a run is initialized.

In order to generate the candidate at step n we add the n-th sample $\Delta\mathbf{x}$ of the proposal distribution to the current state $\mathbf{x}_n$, i.e.

$$\mathbf{x}^* = \mathbf{x}_n + \Delta\mathbf{x}$$

Next, we use the function *MH_decision* to decide whether $\mathbf{x}^*$ is accepted or not.

```
In [ ]: MH_Pars = namedtuple('MH_Pars', ['Proposal'])

        def init_MH_pars(sp):
            proposal = sp['Test Suite']['Proposal']
            return MH_Pars(Proposal=proposal)
```

```
In [ ]: def generate_candidate(center, delta):
            return center + delta
```

3 Metropolis-Hastings algorithm

```
In [ ]: def MH_sampler(pars, target, initial_state, run_data, C_generation=False, likelihood=Tru
            ds, N = run_data.DataStore, run_data.N
```

```
    target_pdf = target['pdf']
    proposal_samples = pars.Proposal['Samples']

    current = initial_state
    accepted = True

    #The integration of the C- and L-variant still has to be done.
    #if C_generation:
    #    generation_function = generate_candidate
    #else:
    #    generation_fuction = L_generate_candidate

    if likelihood:
        decision_function, comparison_function = likelihood_acceptance_decision, compose
    else:
        decision_function, comparison_function = acceptance_decision, target['pdf']

    for n in range(1, N):
        save_state(data_store=ds, step=n,
                   state=current, value=target_pdf(current),
                   accepted_p=accepted)
        proposed = generate_candidate(center=current, delta=proposal_samples[n])
        accepted = decision_function(current, proposed, target_pdf)
        if accepted:
            current = proposed
        else:# The else clause is redundant but added for readability.
            current = current
    return run_data
```

4 Metropolis-Hastings using L instead of C

In [ ]:
```
#@numba.jit
def L_generate_candidate(center, L, scale, z_sample):
    return center + scale*L@z_sample
```

In [ ]:
```
#@numba.jit
def L_MH_sampler(pars, target, initial_state, run_data, likelihood=True):
    ds, N = run_data.DataStore, run_data.N
    sp = target['State Space']
    opt_scale, L = sp['sigma_opt'], sp['Id']

    if likelihood:
        decision_function, comparison_function = likelihood_acceptance_decision, compose
    else:
        decision_function, comparison_function = acceptance_decision, target['pdf']

    target_pdf = target['pdf']
```

```
current = initial_state
accepted = True

z_samples = get_samples(sp=sp, name='Z')
for n in range(1, N):
    save_state(data_store=ds, step=n,
               state=current, value=target_pdf(current),
               accepted_p=accepted)
    proposed = L_generate_candidate(center=current,
                                    L=L, scale=opt_scale,
                                    z_sample=z_samples[n])
    accepted = decision_function(current, proposed, target_pdf)
    if accepted:
        current = proposed
    else:# The else clause is redundant but added for readability.
        current = current
return run_data
```

L_AM_Sampling

January 15, 2018

1 *Adaptive* MH using *Cholesky decomposition* of the covariance

See the 1999 and 2001 papers of Haario et al.

```
In [ ]: import numpy as np
        import numpy.linalg as la
        import matplotlib.pyplot as plt
        import scipy as sp
        import scipy.stats as ss
        import math
        import random
        import numba
        from collections import namedtuple
        %precision 4
        %matplotlib inline
```

from importlib import reload reload(ut)

```
In [ ]: from MH_Sampling import MH_decision
        from FileHandling import save_state
        from TestSuite import generate_state_space, generate_iid_samples, get_standard_normal_sa
```

```
In [ ]: AM_Pars = namedtuple('AM_Pars',
                            ['Origin', 'Id',
                             'sigma_0', 'sigma_opt', 'L_0',
                             'z_samples'])

        def init_AM_pars(sp):
            dim, origin, idty = sp['dim'], sp['Origin'], sp['Id']
            sigma_0, sigma_opt = 0.1/np.sqrt(dim), sp['sigma_opt']
            L_0 = idty
            return AM_Pars(Origin=origin, Id=idty,
                           sigma_0=sigma_0, sigma_opt=sigma_opt, L_0=L_0,
                           z_samples=get_standard_normal_samples(sp))
```

2 Adaptive MH algorithm *AM*

2.1 Generate the candidate next sample

We consider a version of the *Adaptive Metropolis* (*AM*) sampler of Haario et al. (2001). We want to sample from the d-dimensional target distribution $\pi(\mathbf{x})$.

We perform a Metropolis algorithm with covariance matrix $\boldsymbol{Q}_n$ at iteration n given by

$$\mathbf{Q}_n(\mathbf{x},) = N(\mathbf{x}, \sigma_0^2 \mathbb{1}_d)$$

for $n \leq 2d$, while for $n > 2d$

$$\mathbf{Q}_n(\mathbf{x},) = (1\beta)N(\mathbf{x}, \sigma_{opt}^2 \mathbf{C}_n) + \beta N(\mathbf{x}, \sigma_0^2 \mathbb{1}_d)$$

where $\mathbf{C}_n$ is the current empirical estimate of the covariance of the target distribution based on the samples so far, $\sigma_0^2 = \frac{0.1^2}{d}$ and $\sigma_{opt}^2 = \frac{2.38^2}{d}$ are the initial and optimal scale, respectively, and β is a small positive constant, we use $\beta = 0.05$.

In other words, the next candidate is sampled from

$$\mathbf{x}^* \sim \mathbf{Q}_n(\mathbf{x},)$$

The text above is adapted from Section 2 of Gareth O. Roberts and Jeffrey S. Rosenthal (2008) *Examples of Adaptive MCMC*.

2.2 Random covariance matrix M from the above paper.

```
In [ ]: Factors = namedtuple('Factors',
                              ['Chol', 'Scale'])
```

```
In [ ]: def get_prop_data(L, n, pars):
            beta = 0.05
            d, _ = L.shape
            sigma_0, sigma_opt, L_0 = pars.sigma_0, pars.sigma_opt, pars.L_0
            init, current = Factors(Chol=L_0, Scale=sigma_0), Factors(Chol=L, Scale=sigma_opt)
            init_period = 2*d
            if n <= init_period:
                return init
            else:
                return current if np.random.binomial(n=1, p=1-beta) else init
```

3 Generation of candidate

If the proposal distribution is the d-dimensional multivariate normal distribution $N(\boldsymbol{m}, \boldsymbol{C})$ then the next candidate $\boldsymbol{x}^*$ is generated according to that distribution, i.e.

$$\boldsymbol{x}^* \sim N(\boldsymbol{m}, \boldsymbol{C})$$

If L is the lower Cholesky factor of C, i.e. $C = LL^\top$ this can be rewritten as

$$\boldsymbol{x}^* = \boldsymbol{m} + L\boldsymbol{z}$$

where $\boldsymbol{z} \sim N(\boldsymbol{0}, \mathbb{1}_d)$ is a sample of the d-dimensional standard normal distribution.
In case of

$$\boldsymbol{x}^* \sim N(\boldsymbol{m}, \sigma^2 \boldsymbol{C})$$

this becomes

$$\boldsymbol{x}^* = \boldsymbol{m} + \sigma L \boldsymbol{z}$$

```
In [ ]: def C_generate_candidate(m, C, s):
            return

In [ ]: def L_generate_candidate(m, L, s, z):
            return m + s*L@z

In [ ]:

In [ ]: L_SAMPLES = np.array([L_generate_candidate(m, L, s, z=z_sample) for z_sample in Z_sample

In [ ]: def L_generate_samples(m, L, s,):
            ,

In [ ]: @numba.jit(nopython=True)
        def rank_1_update(L, u, alpha, beta):
            assert alpha > 0, 'Argument alpha should be positive'
            assert beta > 0, 'Argument beta should be positive'
            d = len(u)
            L = np.sqrt(alpha)*L  #Added
            b = 1
            nL = np.zeros_like(L)
            v = np.copy(u)  #Added
            for j in np.arange(d):
                nL[j,j] = np.sqrt(L[j,j]**2 + (beta/b)*(v[j]**2))
                gamma = b*L[j,j]**2 + beta*v[j]**2
                for k in range(j+1, d):
                    v[k] = v[k] - (v[j]/L[j,j])*L[k,j]
                    nL[k,j] = (nL[j,j]/L[j,j])*L[k,j] + (nL[j,j]*beta*v[j]/gamma)*v[k]
                b = b + beta*(v[j]**2/L[j,j]**2)
            return nL

In [ ]: def update_moments(mean, L, sample, n):
            next_n = n + 1
            w = 1/next_n
            new_mean = mean + w*(sample - mean)
            new_L = rank_1_update(L=L, u=sample, alpha=1-w, beta=w)
            return new_mean, new_L, next_n

In [ ]: @numba.jit
        def update_L(samples):
            N, d = samples.shape
            initial_period = 2*d
```

```
            initial_cov = np.cov(samples[:initial_period], rowvar=False)
            initial_mean = np.mean(samples[:initial_period], axis=0)
            C = initial_cov
            L = la.cholesky(initial_cov)
            mean = initial_mean
            for n in range(initial_period, len(samples)):
                sample = samples[n]
                w = 1/(n+1)
                L = rank_1_update(L, sample-mean, alpha=(n-1)/n, beta=w)
                mean = (1-w)*mean + w*sample
            return L@L.T
```

In [ ]:
```
def AM_sampler(pars, target, initial_state, run_data):
    ds, N = run_data.DataStore, run_data.N


    target_pdf = target['pdf']
    z_samples = pars.z_samples

    current = initial_state
    mean, L, sigma_0 = pars.Origin, pars.L_0, pars.sigma_0
    accepted = True
    d = len(initial_state)
    init_period = 2*d
    samples=[]
    for n in range(init_period):
        save_state(data_store=ds, step=n,
                   state=current, value=target_pdf(current),
                   mean=mean, covariance=L, accepted_p=accepted)
        candidate = L_generate_candidate(m=current, L=L, s=sigma_0, z=z_samples[n])
        accepted = MH_decision(current=current, proposed=candidate, pdf=target_pdf)
        if accepted:
            current = candidate
        else:
            current = current
        samples.append(current)
    # Calculate the first two moments at the end of initial period.
    initial_cov = np.cov(samples, rowvar=False)
    initial_mean = np.mean(samples, axis=0)
    C = initial_cov
    L = la.cholesky(initial_cov)
    mean = initial_mean


    # Once the initial period is finished we start to adapt.
    for n in range(init_period, N):
        #if n%1000 == 0:
        #    print('n:', n)
```

```
            save_state(data_store=ds, step=n,
                       state=current, value=target_pdf(current),
                       mean=mean, covariance=L, accepted_p=accepted)

            p_L, p_sigma = get_prop_data(L=L, n=n, pars=pars)
            candidate = L_generate_candidate(m=current, L=p_L, s=p_sigma, z=z_samples[n])
            accepted = MH_decision(current=current, proposed=candidate, pdf=target_pdf)
            if accepted:
                current = candidate
            mean, L, n = update_moments(mean, L, current, n)
        return run_data
```

4 Test Section

5 @numba.jit

def tAM_sampler(pars, target, initial_state, N): target_pdf = target['pdf'] z_samples = pars.z_samples

```
current = initial_state
mean, L, sigma_0 = pars.Origin, pars.L_0, pars.sigma_0
accepted = True

d = len(initial_state)
init_period = 2*d
samples = []
for n in range(init_period):
    candidate = generate_AM_candidate(current=current, L=L, sigma=sigma_0, z_sample=z_samples[n]
    accepted = MH_decision(current=current, proposed=candidate, pdf=target_pdf)
    if accepted:
        current = candidate
    else:
        current = current
    samples.append(current)
# Calculate the first two moments at the end of initial period.
initial_cov = np.cov(samples, rowvar=False)
initial_mean = np.mean(samples, axis=0)
C = initial_cov
L = la.cholesky(initial_cov)
mean = initial_mean

# Once the initial period is finished we start to adapt.
for n in range(init_period, N):
    #if n%1000 == 0:
    #    print('n:', n)
    p_L, p_sigma = get_prop_data(L=L, n=n, pars=pars)
    candidate = generate_AM_candidate(current=current, L=p_L, sigma=p_sigma,  z_sample=z_samples
```

```
    accepted = MH_decision(current=current, proposed=candidate, pdf=target_pdf)
    if accepted:
        current = candidate
    mean, L, n = update_moments(mean, L, current, n)
return mean, L, n
```

M = np.random.normal(size=(d,d)) rnd_C = M@M.T rnd_L = la.cholesky(rnd_C)

mean = np.zeros(d) L = PARS.L_0 n = 0 z_sample = PARS.z_samples[n] sample = generate_AM_candidate(current=mean, L=L, sigma=PARS.sigma_0, z_sample=z_sample) mean, L, n, z_sample

d = 100 N = 100000 SP = generate_state_space(dim=d) generate_iid_samples(sp=SP, N=N)

mean, L, n = tAM_sampler(pars=init_AM_pars(sp=SP), target=SP['Test Suite']['Pi_rnd'], initial_state=SP['Initial States'][0], N=N)

la.eigvals(L@L.T)

Candidate Generation

January 15, 2018

1 Generation of candidates

1.1 Basic properties of the *covariance*

All MH-algorithms that we consider use a multivariate Gaussian distribution as proposal distribution centered in some state $\mathbf{x} \in \Omega$ where Ω is the state space.

Often $\mathbf{x}$ is the current state $\mathbf{x}_n$ but this is not necessary. In general, the covariance of the proposal distribution is written as $\sigma^2\mathbf{C}$ where σ is called the *global scale* and $\mathbf{C}$ is a *positive definite matrix*. Adaptive MH algorithms update σ and/or $\mathbf{C}$.

Some important properties of a positive definite matrix $\mathbf{C}$ are

- the *eigenvectors* $\mathbf{e}_i$ for $i = 1, 2, \cdots, d$ of $\mathbf{C}$ are pairwise orthogonal, and
- its *eigenvalues* λ_i are real and positive: $\lambda_i > 0$ for all i
- In the basis consisting of the normalized eigenvectors the matrix $\mathbf{C}$ becomes diagonal. In other words, the matrix $\mathbf{B}$ whose rows are the normalized eigenvectors of $\mathbf{C}$ represents a *orthogonal transformation*, i.e.

  $$\mathbf{B}\mathbf{B}^\top = \mathbb{1}_d = \mathbf{B}^\top\mathbf{B}$$

  and

  $$\mathbf{C} = \mathbf{B}\mathbf{D}\mathbf{B}^\top$$

  where $\mathbf{D}$ is the diagonal matrix of the eigenvalues of $\mathbf{C}$

  $$\mathbf{D} = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_d \end{bmatrix}$$

Another important property is that a positive definite $\mathbf{C}$ has a *unique Cholesky decomposition*, i.e. $\mathbf{C} = \mathbf{L}\mathbf{L}^\top$ where $\mathbf{L}$ is a lower triangle matrix with positive elements on the diagonal, i.e. all $l_{ii} > 0$.

$$\mathbf{L} = \begin{bmatrix} l_{11} & & & \\ l_{12} & l_{22} & & \\ \cdots & & & \\ \cdots & & & \\ l_{1d} & l_{2d} & l_{3d} \cdots & l_{dd} \end{bmatrix}$$

In case $\mathbf{C}$ is diagonal then $\mathbf{L} = \mathbf{D}^{1/2}$.

1.2 Transformation of sampes from the standard Gaussian distribution

To generate a sample $\mathbf{x}^*$ from the multivariate Gaussian distribution with mean $\mathbf{x}$ and covariance $\sigma^2\mathbf{C}$, i.e.

$$\mathbf{x}^* \sim N(\mathbf{x}, \sigma^2\mathbf{C})$$

is equivalent to generate a sample $\mathbf{z}$ from the *standard Gaussian* distribution $N(\mathbf{0}, \mathbb{1}_d)$ and to transform it as

$$\mathbf{x}^* = \mathbf{x} + \sigma\mathbf{L}\mathbf{z}$$

Here, $\mathbf{x}$ is the mean of the proposal distribution and $\mathbf{L}$ is the Cholesky factor of $\mathbf{C}$.
If we can express the update of $\mathbf{C}$ in terms of $\mathbf{L}$ then we do not need $\mathbf{C}$ anymore.
This has a number of advantages:

- E.g. updating L is numerical more stable than updating $\mathbf{C}$, i.e. $\mathbf{C}$ might become **singular**
- We only need to generate samples from the standard Gaussian distribution $N(\mathbf{0}, \mathbb{1}_d)$.

```
In [ ]: import numpy as np
        import numpy.linalg as la
        import matplotlib.pyplot as plt
        import scipy as sp
        import scipy.stats as ss
        import seaborn as sns
        import math
        import random
        import numba
        from collections import namedtuple
```

```
In [ ]: plt.style.use('classic') #'seaborn-whitegrid'
        sns.set()
        %precision 4
        %matplotlib inline
        # To reload modules
```

```
In [ ]: import importlib
```

```
In [ ]: def generate_sample(center, L, scale, z):
            return center + scale*L@z
```

2 Time to Test

2.1 Test Procedure

- First, we generate samples from a multivariate Gaussian distribution with random mean $\mathbf{m}$ and covariance $\mathbf{C}$
- Second, we factorize $\mathbf{C} = \sigma^2\mathbf{C}_n$ where $\det(\mathbf{C}_n) = 1$
- Third, we obtain the Cholesky factor $\mathbf{L}$ of $\mathbf{C}_n$

- Fourth, we use σ, $\mathbf{L}$ and $\mathbf{z} \sim N(\mathbf{0}, \mathbb{1}_d)$ to obtain the samples

$$\mathbf{m} + \sigma \mathbf{L} \mathbf{z}$$

- We scatter the samples obtained in step one and four and the spread should be similar.

```
In [ ]: def random_Gaussian(d):
            # mean = np.random.random(size=d)
            # X = np.random.random(size=(d, d))
            mean = np.random.uniform(low=-10, high=10, size=d)
            X = np.random.uniform(low=-10, high=10, size=(d, d))
            cov = X.T@X
            return mean, cov

        def samples_random_Gaussian(d, N):
            m, C = random_Gaussian(d=d)
            samples = ss.multivariate_normal.rvs(mean=m, cov=C, size=N)
            return samples, m, C
```

```
In [ ]: def test_generate_candidate(d, N):
            # Generate samples from a Gaussian with random mean and covariance
            samples, mean, cov = samples_random_Gaussian(d=d, N=N)

            # Calculate the normalize Cholesky factor L
            scale = la.det(cov)
            n_cov = cov/scale**2
            n_L = la.cholesky(n_cov)

            # Generate samples from the standard normal procedure
            z_samples = ss.multivariate_normal.rvs(mean=np.zeros(d), cov=np.eye(d), size=N)

            # Transform the z_samples
            L_generated_samples = np.zeros_like(z_samples)
            for idx, z_sample in enumerate(z_samples):
                L_generated_samples[idx] = generate_sample(center=mean, L=n_L, scale=scale, z=z_
            return samples, L_generated_samples, mean, cov
```

```
In [ ]: SAMPLES, L_GENERATED_SAMPLES, MEAN, COV = test_generate_candidate(d=5, N=10000)
```

```
In [ ]: from Visualization import compare_sample_spread
```

```
In [ ]: compare_sample_spread(dim1=0, dim2=1,
                              list_of_samples=[SAMPLES, L_GENERATED_SAMPLES],
                              colors=['red', 'blue'], titles=['C-generated', 'L_generated'])
```

```
In [ ]: MEAN, COV
```

FileHandling

January 15, 2018

1 Save and Inspect the state

```
In [ ]: import os
        import numpy as np
```

```
In [ ]: def save_state(data_store, step, state, value, accepted_p,
                           mean=None, covariance=None, scale=None, threshold=None, C_evol_pt=None):
            data_store['States'][step] = state
            data_store['Densities'][step] = value
            data_store['Accepted_p'][step] = accepted_p
            data_store['Means'][step] = mean
            data_store['Covariances'][step] = covariance
            data_store['Scales'][step] = scale
            data_store['Thresholds'][step] = threshold

        def inspect_state(data_store, step):
            state = data_store['States'][step]
            value = data_store['Densities'][step]
            accepted_p = data_store['Accepted_p'][step]
            mean = data_store['Means'][step]
            covariance = data_store['Covariances'][step]
            scale = data_store['Scales'][step]
            threshold = data_store['Thresholds'][step]
            print("State:", state, "R: ", scale, "\nThreshold: ", threshold, "\nState: ", state,
                  "\nIt's value: ", value, "\nMean: ", mean,
                  "\nCovariance: ", covariance)
```

2 Save in the file format used by *PyMC3*

2.1 Structure of the data directory

The directory structure of 'Data' is as follows

```
1. for each dimension $d$ of the state space, 'Data' contains a folder 'Dim d'
2. for each target, 'Dim d' contains a folder 'Target k' where $k$ is the index of
that targets in the test suite
3. for each sampler, 'Target k' contains a folder named after that sampler
```

4. for each run given the dimension of the state space, the target and the sampler, a file 'chain_i' is generated where i is the index of the run.

The global variable PARENT_FOLDER contains the parent folder, i.e. the folder where the experimental data will be store, e.g.

```
PARENT_FOLDER = '/home/philip/Desktop/Adaptive_Sampling/ipynb_code'
```

The functions below assume that the parent folder is correctly set.

```
In [ ]: def relative_path_to_chain(dim, t_name, s_name):
            data_folder = 'Data'
            dim_folder = 'Dimension_{}'.format(dim)
            target_folder = t_name
            sampler_folder = s_name
            return './'+'/'.join([data_folder, dim_folder, target_folder, sampler_folder])

        class ChDir(object):
            """
            Step into a directory temporarily.
            """
            def __init__(self, path):
                self.old_dir = os.getcwd()
                self.new_dir = path

            def __enter__(self):
                os.chdir(self.new_dir)

            def __exit__(self, *args):
                os.chdir(self.old_dir)

        def save_chain(chain, idx, individual_components_p=True):
            """Save a single-chain trace with index 'idx'. PyMC3 uses the labels x__0, x__1, x__
            for a vector when are regarded as COMPONENTS of that vector.
            If we want to treat them INDIVIDUALLY the labels x_0, x_1, x_2, etc. have to be used
            This is, we use double versus single underscore.
            """
            chain_name = 'chain-{}.csv'.format(idx)
            _, nbcols = chain.shape
            underscore = '_' if individual_components_p else '__'
            varnames = ['x{}{}'.format(underscore, index) for index in range(nbcols)]
            header = ','.join(varnames)
            np.savetxt(fname=chain_name, X=chain, header=header, comments='', delimiter=',')

        def save_run_data(run_data, parent_folder):
            warning = 'Parent Folder \'%s\' does NOT exist'%(parent_folder)
            if not os.path.exists(parent_folder):
                return warning
            chain = run_data.DataStore['States']
```

```
            chain_folder = relative_path_to_chain(dim=run_data.StateSpace['dim'],
                                                  t_name=run_data.Target['Name'] ,
                                                  s_name=run_data.Sampler['Name'])
            if not os.path.exists(chain_folder):
                os.makedirs(chain_folder)
            with ChDir(chain_folder):
                nbfiles = len(os.listdir())
                save_chain(chain=chain, idx=nbfiles)

        def save_comparison(combined_data, parent_folder):
            for i, run_data in enumerate(combined_data):
                save_run_data(run_data, parent_folder)
```

In [ ]:
```
def read_states(f_name, dim, t_name, s_name):
    chains_folder = relative_path_to_chain(dim=dim, t_name=t_name, s_name=s_name)
    with ChDir(chains_folder):
        return np.loadtxt(fname=f_name, skiprows=1, delimiter=',')
```

TestSuite

January 15, 2018

```
In [ ]: import numpy as np
        import numpy.linalg as la
        import matplotlib.pyplot as plt
        import scipy as sp
        import scipy.stats as ss
        import math
        import random
        from collections import namedtuple
        #get_ipython().magic('precision 4')
        #get_ipython().magic('matplotlib inline')
```

1 State Space

The dictionary *State Space* contains its dimension, the origin and identity matrix, and the optimal σ_{opt} scale according to Gelman et al.: $\sigma_{opt} = 2.38/\sqrt{d}$, where d is the dimension of the state space and σ_{opt} is the scale of the isotropic proposal distribution used in the Metropolis-Hastings sampler.

Later on we will add the target distributions of the test suite used in our experiments.

```
In [ ]: def calculate_optimal_sigma(dim):
            return 2.38/np.sqrt(dim)

        def state_space(dim):
            return {'dim': dim, 'Origin': np.zeros(dim), 'Id': np.eye(dim),
                    'sigma_opt':calculate_optimal_sigma(dim)}
```

1.1 A random state of the state space

```
In [ ]: def generate_random_state(sp, min_range=-10, max_range=10):
            """Generates a random state in the state space that fits in the area to be plotted.
            """
            return np.random.uniform(low=min_range, high=max_range, size=sp['dim'])
```

```
In [ ]: def generate_initial_states(sp, nb_runs):
            initial_states = {i:generate_random_state(sp) for i in np.arange(nb_runs)}
            # Only update if the key does not exist yet. Check out how to do this.
            sp.update({'Initial States':initial_states})
```

2 Testsuite of Target Distributions

2.1 Uncorrelated and Correlated Gaussian Distributions

π_1 is the uncorrelated Gaussian distribution with covariance matrix

$$C_u = \begin{pmatrix} 100 & 0 \\ 0 & 1 \end{pmatrix}$$

and π_2 is the correlated Gaussion distribution with covariance matrix

$$C_c = \begin{pmatrix} 50.5 & 49.5 \\ 49.5 & 50.5 \end{pmatrix}$$

3 Covariance Matrix

```
In [ ]: def generate_rotation_matrix(theta):
            # Rotation matrix is 2-dimensional
            return np.array([[np.cos(theta), -np.sin(theta)],
                             [np.sin(theta), np.cos(theta)]])

        def generate_correlated_cov(uncorrelated_cov, theta):
            correlated_cov = np.copy(uncorrelated_cov)
            R = generate_rotation_matrix(theta)
            R_inv = la.inv(R)
            # Rotate the first 2 dimensions only and leave the other dimensions
            # of the covariance matrix intact.
            correlated_cov[:2, :2] = R @ uncorrelated_cov[:2,:2] @ R_inv
            return correlated_cov
```

3.0.1 We could also use the fact that the transpose of a rotation is also its inverse.

def alt_generate_correlated_cov(uncorrelated_cov, theta): # Here we use the fact that the transpose of a rotation is also its inverse. correlated_cov = np.copy(uncorrelated_cov) R = generate_rotation_matrix(theta) correlated_cov[:2, :2] = R @ uncorrelated_cov[:2,:2] @ R.T return correlated_cov

3.1 Contour Functions corresponding with the Target Distributions

3.1.1 Standard Ellipse and General Ellipse

When the ellips with equation

$$\left(\frac{x_1}{a}\right)^2 + \left(\frac{x_2}{b}\right)^2 = 1$$

is rotated over an angle θ then the equation of that ellips becomes

$$\underbrace{\left(\frac{\cos^2\theta}{a^2} + \frac{\sin^2\theta}{b^2}\right)}_{\text{A}} x_1^2 + \underbrace{\left(\frac{\sin^2\theta}{a^2} + \frac{\cos^2\theta}{b^2}\right)}_{\text{C}} x_2^2 + 2\underbrace{\cos\theta\sin\theta\left(\frac{1}{a^2} - \frac{1}{b^2}\right)}_{\text{B}} x_1 x_2 = 1$$

or

$$Ax_1^2 + 2Bx_1x_2 + Cx_2^2 = 1$$

where

$$B^2 - AC < 0$$

actually

$$B^2 - AC = -1/(ab)^2$$

```
In [ ]: def get_ellipse_parameters(cov):

            """Get the first 2 eigenvalues and their angle of covariance matrix.
            The eigenvalues are returned in descending order together with
            the angle of rotation (in radians). The eigenvalues correspond with
            half the length, a and b, of these two main axes of
            the general ellipse.
            If the angle is small enough, meaning that the covariance matrix
            can be considered diagonal, 0.0 is returned."""

            e, v = la.eig(cov)
            e_1, e_2, *_ = e
            a, b = np.sqrt(e_1), np.sqrt(e_2)
            v_a, v_b, *_ = v
            # a must be at least b
            if a < b:
                a, b = b, a
                v_a, v_b = v_b, v_a
            cos, *_ = v_a
            theta = np.arccos(cos)
            if np.isclose(theta, 0):
                theta = 0.0
            return a, b, theta
```

```
In [ ]: def calculate_ellipse_coefficients(a, b, theta):
            sin, cos = np.sin(theta), np.cos(theta)
            cos_sqd, sin_sqd = cos**2, sin**2
            a_sqd, b_sqd = a**2, b**2
            A = cos_sqd/a_sqd + sin_sqd/b_sqd
            C = sin_sqd/a_sqd + cos_sqd/b_sqd
            B = (1/a_sqd - 1/b_sqd)*sin*cos
            return A, B, C
```

```
In [ ]: def get_Gaussian_contour(cov):
            a, b, theta = get_ellipse_parameters(cov)
            A, B, C = calculate_ellipse_coefficients(a, b, theta)
            return lambda x1, x2: A*x1**2 + 2*B*x1*x2 + C*x2**2
```

4 Distribution

We have three kind of distributions in the test suite 1. Gaussian distributions 2. mixture of Gaussians 3. transformed Gaussians, the so called twist distributions

The second kind is not implemented yet.

The dictionary *Gaussian* contains the following fields * its *Name* * the *State Space* on which the probability distribution is defined * its *probability density function* or *pdf* * *Samples* that are *independent and identically distributed*. These samples will be compared to the samples generated by the MCMC samplers studied. These samples are added at run time. * the *Contour Function* used to plot the * *Contour Levels* corresponding to the preset confidence levels, cfr. the global variable CONFIDENCE_LEVELS for the values used. The values of the χ^2 distribution corresponding to the confidence levels used: 67, 90, 95 and 99 percent.

Additionarly to the fields of Gaussian dictionary, *non_Gaussian* contains the additional fields *Transformation*, this is the function that will generated its i.i.d. samples using the samples of generating Gaussian.

4.1 Gaussian Distributions in the Test Suite

4.1.1 Draw the contour lines corresponding to preset *confidence levels*

```
In [ ]: def get_chi2s(df, confidence_levels=[0.67, 0.90, 0.95, 0.99]):
            """ppf stands for the percent point function (inverse of cdf  percentiles)."""
            #contour_levels = {conf:ss.chi2.ppf(conf, df) for conf in confidence_levels}
            contour_levels = [ss.chi2.ppf(conf, df) for conf in confidence_levels]
            return contour_levels
```

4.1.2 Generate the Gaussians given their covariances

```
In [ ]: def generate_Gaussian(sp, name, mean, cov):
            d = sp['dim']
            rv = ss.multivariate_normal(mean=mean, cov=cov)
            return {'Name':name,
                    'State Space':sp,
                    'pdf':rv.pdf,
                    'Mean':mean,
                    'Covariance':cov,
                    'Contour Function':get_Gaussian_contour(cov),
                    'Contour Levels':get_chi2s(df=2)
                    #'Samples':None,
                   }
```

```
In [ ]: def generate_covs(sp):
            # Standard Normal Z has the identity matrix as covariance
            identity = sp['Id']

            # The optimal isotropic proposal is $\sigma_{opt} * Id$
            var_opt = sp['sigma_opt']**2
            prop_cov = var_opt*identity
```

```
        # P1_2
        Pi_1_cov = np.copy(identity)
        Pi_1_cov[0, 0] = 100

        # Pi_2
        Pi_2_cov = generate_correlated_cov(Pi_1_cov, np.pi/4)

        # Pi_rnd
        d = sp['dim']
        M = np.random.normal(size=(d,d))
        Pi_rnd = M@M.T
        return {'Z':identity, 'Proposal':prop_cov, 'Pi_1':Pi_1_cov,
                'Pi_2':Pi_2_cov, 'Pi_rnd':Pi_rnd}


    def generate_all_Gaussians(sp):
        named_covs = generate_covs(sp)
        gaussians = {name:generate_Gaussian(sp=sp, name=name, mean=sp['Origin'], cov=cov)
                     for name, cov in named_covs.items()}
        return gaussians
```

4.2 Proposal Generator

The **radial basis** or **isotropic** proposal generator used by the Metropolis-Hastings sampler. Its *mean* is the origin and the *spread* is σ.

```
In [ ]: def generate_isotropic_Gaussian(sp, sigma):
            origin, identity = sp['Origin'], sp['Id']
            diagonal = sigma**2 * identity
            return generate_Gaussian(sp=sp, name='Isotropic', mean=origin, cov=diagonal)
```

```
In [ ]: def generate_random_Gaussian(sp):
            d, origin = sp['dim'], sp['Origin']
            M = np.random.normal(size=(d,d))
            random_cov = M@M.T
            return generate_Gaussian(sp=sp, name='Random', mean=origin, cov=random_cov)
```

4.3 Twisted Distributions in the Test Suite

```
In [ ]: def f_twist(b):
            def phi_b(x):
                """Argument and the value returned are d-dimensional numpy arrays."""
                y = np.copy(x)
                x1, x2 = x[:2]
                y[0], y[1] = x1, x2 + b*x1**2 - 100*b
                return y

            def phi_b_inv(y):
```

```
            """Argument and the value returned are d-dimensional numpy arrays."""
            x = np.copy(y)
            y1, y2 = y[:2]
            x[0], x[1] = y1, y2 - b*y1**2 + 100*b
            return x
        return phi_b, phi_b_inv

    def compose2(f, g):
        return lambda x: f(g(x))
```

In [ ]:
```
def apply_to(transformation, pts):
    """Used to generate samples of a twist distribution given samples of a Gaussian one.
    The argument transformation, e.g. phi_b(x1, x2) = (y1, y2) is a 2-dimensional
    transformation of the vectors in pts. The result is an array of the transformed poin
    """
    transformed_pts = np.zeros_like(pts)
    for i, pt in enumerate(pts):
        transformed_pts[i] = transformation(pt)
    return transformed_pts
```

In [ ]:
```
def apply(transformation):
    return lambda pts: apply_to(transformation, pts)
```

In [ ]:
```
def get_twisted_contour(gaussian, b):
    cov = gaussian['Covariance']
    f = get_Gaussian_contour(cov)
    return lambda x1, x2: f(x1, x2 + b*x1**2 - 100*b)
```

In [ ]:
```
def generate_twist(gaussian, b, name):
    # The twisted distribution is a transformation of
    # the uncorrelated Gaussian distribution 'gaussian'
    transformed_distr = gaussian.copy()
    transformed_function, inverse_twist_function = f_twist(b=b)
    transformed_pdf = compose2(gaussian['pdf'], transformed_function)
    contour_function = get_twisted_contour(gaussian=gaussian, b=b)
    transformed_distr.update({'Name':name,
                              'Generator':gaussian,
                              'pdf':transformed_pdf,
                              'Contour Function':contour_function})
    transformed_distr.update({'Transformation':apply(inverse_twist_function)})
    return transformed_distr
```

In [ ]:
```
def generate_all_twists(gaussian, b_values, names):
    twists ={name:generate_twist(gaussian, b, name)
             for b, name in zip(b_values, names)}
    return twists
```

In [ ]:
```
def generate_test_suite(sp):
    gaussians = generate_all_Gaussians(sp)
```

```
            twists = generate_all_twists(gaussian=gaussians['Pi_1'],
                                         b_values=[0.03, 0.1],
                                         names=['Pi_3', 'Pi_4'])
            sp.update({'Test Suite':{**gaussians, **twists}})
```

```
In [ ]: def generate_state_space(dim, nb_runs=100, N=None):
            sp = state_space(dim=dim)
            generate_test_suite(sp)
            generate_initial_states(sp=sp, nb_runs=nb_runs)
            return sp
```

4.3.1 Generate independent and identically distributed or i.i.d. samples

These samples will be generated when we initialize a run. They are compared to the correlated samples generated by a MCMC sampler.

```
In [ ]: def iid_samples_Gaussian(gaussian, N):
            mean, cov = gaussian['Mean'], gaussian['Covariance']
            rv = ss.multivariate_normal(mean=mean, cov=cov)
            samples = rv.rvs(size=N)
            gaussian.update({'Samples':samples})
```

4.3.2 Generate i.i.d. samples of an transformed Gaussian distribution.

These samples will be generated when we initialize a run. They are compared to the correlated samples generated by a MCMC sampler.

```
In [ ]: def iid_samples_transformed_Gaussian(distr, N):
            #Samples are generated by transforming the random samples of
            #the generating Gaussian distribution.
            generator = distr['Generator']
            transformation = distr['Transformation']
            if not 'Samples' in generator:
                iid_samples_Gaussian(generator, N)
            transformed_samples = transformation(generator['Samples'])
            distr.update({'Samples':transformed_samples})
```

4.4 Generate i.i.d. samples for the whole Test Suite

```
In [ ]: def generate_iid_samples(sp, N):
            test_suite = sp['Test Suite']
            for name, distr in test_suite.items():
                if 'Generator' not in distr:
                    iid_samples_Gaussian(gaussian=distr, N=N)
                else:
                    iid_samples_transformed_Gaussian(distr=distr, N=N)
```

4.5 Getter functions for the samples of a distribution

```
In [ ]: def get_distribution(sp, name):
            return sp['Test Suite'][name]

        def get_samples(sp, name):
            return get_distribution(sp, name)['Samples']
```

5 Time to test

```
In [ ]: def inspect(sp, field):
            test_suite = sp['Test Suite']
            for key, distr in test_suite.items():
                print(key, distr[field])

        #inspect(SP, 'Covariance')
```

```
In [ ]: def inspect_Gaussian(sp, name_gaussian):
            gaussian = sp['Test Suite'][name_gaussian]
            print(gaussian['Name'])
            print(gaussian['Mean'])
            print(gaussian['Covariance'])
            print(gaussian['Samples'][:5])

        def inspect_transformed_Gaussian(sp, name_distr):
            distr = sp['Test Suite'][name_distr]
            print(distr['Name'])
            print(distr['Mean'])
            print(distr['Covariance'])
            inspect_Gaussian(sp, distr['Generator']['Name'])
            print(distr['Samples'][:5])

        #inspect_transformed_Gaussian(SP, 'Pi_4')
```

SP = generate_state_space(dim=2, nb_runs=10) generate_iid_samples(SP, N=1000) TEST-SUITE = SP['Test Suite']

Z_samples = get_samples(SP, name='Z')

prop = SP['Test Suite']['Proposal'] prop_cov = prop['Covariance'] prop_samples = prop['Samples'] samples = Z_samples @ prop_cov

samples[:10], prop_samples[:10]

Visualization

January 15, 2018

```
In [ ]: import numpy as np
        import numpy.linalg as la
        import matplotlib.pyplot as plt
        import scipy as sp
        import scipy.stats as ss
        import math
        import random
        from collections import namedtuple
        #get_ipython().magic('precision 4')
        #get_ipython().magic('matplotlib inline')
```

1 *Visualize* the results of an *MCMC* run

1.1 Set Up the Grid

The values of $x_{min}, x_{max}, nb_x, y_{min}, y_{max}$, and nb_y depend on where the **probability mass** of the **target distribution** is located, i.e. where the probability density function is sufficiently 'large'.

```
In [ ]: Grid = namedtuple('Grid', ['x_min', 'x_max', 'y_min', 'y_max', 'X', 'Y'])

        def make_grid(x_min=-30.0, x_max=30.0, nb_x =100,
                      y_min=-30.0, y_max=30.0, nb_y =100):
            x_list = np.linspace(x_min, x_max, nb_x)
            y_list = np.linspace(y_min, y_max, nb_y)
            x, y = np.meshgrid(x_list, y_list)
            return Grid(x_min=x_min, x_max=x_max, y_min=y_min, y_max=y_max, X=x, Y=y)

        GRID = make_grid()
```

```
In [ ]: def Mahalanobis_distance(mean, point, precision):
            # The precision matrix is the inverse of the covariance matrix.
            delta = mean - point
            return np.sqrt(delta @ precision @ delta.T)

        def squared_Mahalanobis_distance(point, precision):
            # The precision matrix is the inverse of the covariance matrix.
            delta = mean - point
            return delta @ precision @ delta.T
```

```
        def Mahalanobis_norm(point, precision):
            # The precision matrix is the inverse of the covariance matrix.
            return np.sqrt(point @ precision @ point.T)

        def squared_Mahalanobis_norm(point, precision):
            # The precision matrix is the inverse of the covariance matrix.
            return point @ precision @ point.T

        def calculate_fractions(distribution, samples, burnin_pct=0):
            precision = la.inv(distribution['Covariance'])
            end_burnin = burnin_pct*len(samples)//100
            samples_after_convergence = samples[end_burnin:]
            nb_samples = len(samples_after_convergence)
            norm_sq = [squared_Mahalanobis_norm(sample, precision)
                       for sample in samples_at_equilibrium]
            return [sum(norm_sq <= contour_level)/nb_samples
                    for contour_level in distribution['Contour Levels']]
```

In [ ]:
```
        # # the histogram of the distances
        # n, bins, patches = plt.hist(Distances, 50, normed=1, facecolor='green', alpha=0.75)
        #
        # plt.xlabel('Distance to the Mean')
        # plt.ylabel('Relative Frequency')
        # plt.title(r'$\mathrm{Histogram\ of\ Sample\ Distance\ to\ the\ Mean}$')
        # plt.grid(True);

        # # cumulative distribution of the distances
        # values, base = np.histogram(Distances, bins=100)
        # # evaluate the cumulative
        # cumulative = np.cumsum(values)
        # # plot the cumulative function
        # plt.plot(base[:-1], cumulative, c='blue');

        # fig = plt.figure("i.i.d.", figsize=(7, 7))
        # ax = fig.add_subplot(1, 1, 1)
        # subplot(ax, Pi_2, Pi_2.Samples[::1000], dim1=0, dim2=1,
        #         title='Distribution of i.i.d. generated samples.')

        # ## Contour Lines corresponding with given Confidence Levels
        #
        # Next we plot the contour lines corresponding with 10, 90, 95 and 99 percent confidence
        # use the corresponding values of $\chi^2$-distribution. In case of a bivariate distribu
```

In [ ]:
```
        def plot_contour_lines(ax, distribution, dim1, dim2):
            global GRID
            X, Y = GRID.X, GRID.Y
            # Plot the contour lines
```

```
        contour_function = distribution['Contour Function']
        # Since we project and a 2-dimensional subspace we will use 2 degrees of freedom
        # instead of the dimension of the statespace as we did before.
        contour_levels = distribution['Contour Levels']
        Z = contour_function(X, Y)
        ax.contour(X, Y, Z, contour_levels)

    def scatter_samples(ax, samples, dim1, dim2):
        ax.scatter(samples[:, dim1], samples[:, dim2])

    def subplot(ax, distribution, samples, dim1, dim2, title, fraction_str=None):
        ax.set_title(title, fontweight='bold', color='blue', fontsize=14)
        ax.axis([GRID.x_min, GRID.x_max, GRID.y_min, GRID.y_max])
        ax.set_xlabel('Dimension ' + str(dim1))
        ax.set_ylabel('Dimension ' + str(dim2))
        plot_contour_lines(ax, distribution, dim1, dim2)
        scatter_samples(ax, samples, dim1, dim2)
```

In [ ]:
```
    def compare_to_iid_samples(run_data, nb_samples, dim1=0, dim2=1, burnin_pct=50):
        global GRID
        fig, ((ax_left, ax_right)) = plt.subplots(nrows=1, ncols=2, figsize=(15,7))
        target = run_data.Target
        # Data to be plotted.
        step = run_data.N//nb_samples
        mcmc_samples = run_data.DataStore['States']
        iid_samples = target['Samples']
        mcmc_samples_2_display = mcmc_samples[::step]
        iid_samples_2_display = iid_samples[::step]
        mcmc_fractions = calculate_fractions(target, mcmc_samples, burnin_pct)
        iid_fractions = calculate_fractions(target, iid_samples, burnin_pct)

        # Information to be shown.
        s_name = run_data.Sampler['Name']
        title_str = 'Distribution of samples generated by {:s}'
        title_info = title_str.format(s_name)
        burnin_str = 'Burn in used is {:d} percent of the generated samples.'
        burnin_info = burnin_str.format(burnin_pct)
        mcmc_str = '{:s} Fractions: {:1.5f}, {:1.5f}, {:1.5f}, and {:1.5f}'
        mcmc_info = mcmc_str.format(s_name, *mcmc_fractions)
        iid_str = 'i.i.d. Fractions: {:1.5f}, {:1.5f}, {:1.5f}, and {:1.5f}'
        iid_info = iid_str.format(*iid_fractions)
        title_mcmc = '{:s} Generated'.format(s_name)
        title_idd = 'i.i.d. Generated'
        suptitle_str = 'Comparison of the {:s} (left) vs. the IID (right) sample distributio
        suptitle = suptitle_str.format(s_name)

        # Display everything.
        print(burnin_info)
```

```
            print(mcmc_info)
            print(iid_info)
            fig.suptitle(suptitle, fontweight='bold', color='red', fontsize=28)
            subplot(ax_left, target, mcmc_samples_2_display, dim1, dim2, title=title_mcmc)
            subplot(ax_right, target, iid_samples_2_display, dim1, dim2, title=title_idd)
```

In [ ]:
```
def plot_samples(run_data, nb_samples, dim1=0, dim2=1, burnin_pct=50):
    global GRID
    # New figure window for the current sampling method
    s_name = run_data.Sampler['Name']
    fig = plt.figure(s_name, figsize=(10, 10))
    ax = fig.add_subplot(1, 1, 1)
    # Data to be plotted.
    target = run_data.Target
    # Data to be plotted.
    step = run_data.N//nb_samples
    mcmc_samples_2_display = run_data.DataStore['States'][::step]
    # Information to be shown.
    fig_title_str = 'Distribution of samples generated by {:s}'
    fig_title =  fig_title_str.format(s_name)
    #Plot everything.
    subplot(ax, target, mcmc_samples_2_display, dim1, dim2, title=fig_title)
```

In [ ]:
```
def subplot_2(ax, samples, dim1, dim2, title, color):
    ax.set_title(title, fontweight='bold', fontsize=22)
    ax.set_xlabel('Dimension ' + str(dim1), fontsize=24)
    ax.set_ylabel('Dimension ' + str(dim2), fontsize=24)
    ax.scatter(samples[:, dim1], samples[:, dim2], color=color)

def compare_sample_spread(dim1, dim2, list_of_samples, titles, colors):
    # Ensure that dim1 and dim2 are less than the dimension of the state space.
    _, dim = list_of_samples[0].shape
    assert dim1 < dim, "dim1 should be less then %r" % dim
    assert dim2 < dim, "dim2 should be less then %r" % dim

    #Generate the supplots.
    fig, (axes) = plt.subplots(nrows=1, ncols=2, figsize=(15,7), sharex='col', sharey='r
    for ax, samples, title, color in zip(axes, list_of_samples, titles, colors):
        subplot_2(ax=ax, samples=samples, dim1=dim1, dim2=dim2, title=title, color=color
```