Master Thesis

Markov chain Monte Carlo methods in biological mechanistic models

Author(s):
Kuzmanovska, Irena

Publication Date:
2012

Permanent Link:
https://doi.org/10.3929/ethz-a-007343293

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
Markov Chain Monte Carlo Methods in Biological Mechanistic Models

Master Thesis

IRENA KUZMANOVSKA
ETH Zurich
irenak@student.ethz.ch

January 2, 2012 - July 2, 2012

Supervised by:
Sotiris Dimopoulos
Andreas Milias-Argeitis
Prof. Dr. Jörg Stelling
Abstract

ODE models are the most common way of mathematically representing deterministic biochemical networks. The values of the parameters of a model are, however, rarely known with high certainty; thus methods of parameter identification are of great importance. A probabilistic approach based on Bayesian statistics offers a natural way of dealing with such parameter uncertainty [6]. It requires efficient estimation of probability distributions over the parameter values, which is a very challenging task especially in high dimensional spaces where the probability is typically concentrated in regions whose volume is a tiny fraction of the total [11, 10]. To generate points drawn from such distributions with efficiency, the sampling procedure must search for these relevant regions. A good candidate for this task is the class of Markov Chain Monte Carlo (MCMC) sampling algorithms.

In this work five different MCMC algorithms are evaluated: Metropolis-Hastings (MH), Hamiltonian Monte Carlo (HMC), Metropolis Adjusted Langevin Equation (MALA) and two more sophisticated MCMC that have been developed recently by Girolami and Calderhead [5] which are defined on a Riemann manifold: manifold Metropolis Adjusted Langevin Equation (mMALA) and Riemann Manifold Hamiltonian Monte Carlo (RMHMC). When used to sample from the posterior distribution of a dynamical ODE model, these methods require information about the states of the model and their first- and/or second-order sensitivities for specific points of time. Except for very small linear ODE models, the equations for the states and sensitivities can not be solved analytically; thus specialized ODE solvers are needed. In this work an automatic framework has been developed in which the MCMC samplers are connected and interact with specialized ODE solvers. The framework takes as an input an ODE model in SBML format and after a number of sampler iterations, the sampled variables are returned.

The samplers have been applied on three common benchmark test distributions, and also on the posterior distributions of two ODE models (one small and one medium-sized). The algorithms were compared in terms of their convergence properties and speed.
Acknowledgments

I would like to sincerely thank Prof. Dr. Jörg Stelling for his guidance and supervision during my master studies and for letting me do my master thesis in his group.

I would also like to thank Sotiris Dimopoulos and Andreas Milias-Argeitis for their assistance, patience and encouragement throughout the entire master thesis period. Without their support and advice, the work done for this master thesis wouldn’t have been possible.
# Contents

1 Introduction .......................................................... 6  
1.1 ODE Models .......................................................... 6  
1.2 Parameter Estimation ............................................... 6  
  1.2.1 Important Concepts ............................................. 7  
  1.2.2 Methods for Parameter Estimation ......................... 7  

2 Goal ................................................................. 9  

3 Markov Chain Monte Carlo ........................................... 11  
  3.1 General ............................................................ 11  
  3.2 Description of the algorithms used ......................... 11  
    3.2.1 Metropolis - Hastings (MH) .............................. 11  
    3.2.2 Hamiltonian Monte Carlo (HMC) ....................... 12  
    3.2.3 Metropolis Adjusted Langevin Algorithm (MALA) .... 15  
    3.2.4 Manifold Metropolis Adjusted Langevin Algorithm (mMALA) 16  
    3.2.5 Riemann Manifold Hamiltonian Monte Carlo (RMHMC) .. 17  
  3.3 Performance Criteria ............................................ 18  

4 Embedding the MCMC Samplers into an Automatic Framework ... 21  
  4.1 Sensitivity Analysis ............................................. 21  
  4.2 Embedding the Sampler into a Framework .................... 21  
    4.2.1 Computing the States ...................................... 22  
    4.2.2 Computing the 1st Order Sensitivities ................. 22  
    4.2.3 Computing 2nd Order Sensitivities ..................... 24  
    4.2.4 Overview of the Framework ............................... 25  

5 Results ............................................................. 27  
  5.1 Test Distributions .............................................. 27  
    5.1.1 Correlated Gaussian ...................................... 27  
    5.1.2 Banana-shaped Distribution ............................. 29  
    5.1.3 Mixture of two Gaussians ............................... 31  
  5.2 Dynamical ODE Models .......................................... 33  
    5.2.1 Small ODE Model ......................................... 33  
    5.2.2 Medium-sized ODE Model ............................... 39  

6 Discussion and Conclusions ......................................... 46  
  6.1 Conclusions .................................................... 46  
  6.2 Future Work .................................................... 47  

A Code for MCMC Samplers ........................................... 49  
  A.1 Metropolis-Hastings (MH) ...................................... 49  
  A.2 Hamiltonian Monte Carlo (HMC) ............................. 49  
  A.3 Metropolis Adjusted Langevin Algorithm (MALA) ........... 50  
  A.4 Manifold Metropolis Adjusted Langevin Algorithm (mMALA) 51  
  A.5 Riemann Manifold Hamiltonian Monte Carlo (RMHMC) ...... 52
B Main Function and Configuration File
  B.1 Main Function ........................................... 54
  B.2 Configuration File ....................................... 55

C Other Core Functions ......................................... 56
  C.1 Calculation of the likelihood ................................. 56
  C.2 Calculation of the gradient of the log-likelihood .......... 56
  C.3 Calculation of G ........................................ 56
  C.4 Calculation of the derivative of G ......................... 57
  C.5 Other functions needed for the Riemannian methods ...... 57
  C.6 Function that calls CVODES ............................... 58
Chapter 1

Introduction

1.1 ODE Models

Mathematical models of biochemical systems help us to better understand observed biochemical processes and to be able to make reasonable predictions about the future behavior of the system [3]. Most commonly, such mathematical models are formalized as a system of ordinary differential equations (ODEs), in which the time evolution of the concentration of each species of the system is modelled by an ODE which captures its interactions with other species. System solutions give us an insight of how the system behaves at different time points and thus enable us to predict its future behavior. The ODE formalism of biochemical models is valid only for homogeneous, well-stirred system, in which the number of molecules is big enough and thus their behaviour is deterministic.

The general form of such an ODE system is

\[ \frac{dx}{dt} = f(x, \theta, t) \]  

(1.1)

where \( x \in \mathbb{R}^N \) is a vector of the states of the system, \( f = (f_1, \ldots, f_N) \) is a vector of \( N \) ODEs (one for each species), \( \theta \in \mathbb{R}^P \) is a vector of the kinetic parameters. Part of the ODE definition are also the initial conditions for the states \( x(t_0) = x_0, (x_0 \in \mathbb{R}^N) \), which together with the set of the ODEs define the so called initial value problem (IVP). In this context, the solution of the system equations for \( T \) time points can be presented in a \( T \times N \) matrix in which each row \( i, (i = 1, \ldots, T) \) contains the solutions for all the \( N \) states at a particular time point \( t_i \), i.e. concentrations of all the species at time \( t_i \).

A biochemical ODE system usually has also a number of observables (biologically observed states) \( y \in \mathbb{R}^M \), such that \( y = g(x) \). Throughout this thesis it will be referred to the observed states as “observables”, “measurements”, “experimental data”, “observed states” interchangeably. Not always all the states \( x \) involved in the model can be observed. In fact, very rarely \( M = N \), especially for bigger systems. Most commonly the observables are combinations of two or more states (ex. sum of the concentrations of two species). Again, if time measurements of the observables are made at \( T \) time points, a matrix \( T \times M \) stores the time measurements for all the observables at different time points.

1.2 Parameter Estimation

The model parameters are very important components of the model definition as they can alter model behavior to great extent. A fraction of the kinetic parameters of a model can be directly measured experimentally. For many other parameters simple and easy ways of measurement do
not exist or the existing measurement techniques lead to incoherent and noisy results, which gives rise to a high degree of uncertainty about their values. In such cases, these parameters values have to rely on assumptions, approximations and, most importantly, fitting.

1.2.1 Important Concepts

Every well defined ODE model of a biochemical system should come with the following components:

1. ODE structure
2. Set of experimental data (measurements)
3. Noise model (for the data) e.g. Gaussian
4. Prior distribution of the parameters (needed for Bayesian inference) \( \pi(\theta) \)

For a particular time point \( t_i \), the observables of the process \( y(t_i) \) are usually contaminated with some measurement error, such that \( y(t_i) = x(t_i) + \xi(t_i) \), where \( \xi(t_i) \) defines an appropriate multivariate noise process [3]. In the process of parameter estimation, it is very important to know how the noise of the data looks like. The experimental data can come from real biological measurements or, for the purpose of testing the model structure or parameter estimation methods, they can be generated artificially, by adding noise \( \xi \) to the outputs obtained by simulating the model with nominal parameter values.

The distribution of the noise defines a very important feature in parameter estimation: the likelihood of the parameters \( P(y|\theta) \) which is the probability of obtaining the set of observed data, under a given parameter value \( \theta \) [10].

The last component of an ODE structure, which is important in Bayesian parameter estimation, is the prior distribution of the parameters \( \pi(\theta) \). This distribution reflects our prior beliefs or knowledge about the parameters even before we have received the data [10]. This helps us to have some initial bias of the region of parameter space and to constrain our search in this region. If we do not have such prior knowledge about the parameters, then we can construct an appropriate prior that embodies our supposed ignorance, for example a uniform prior over the range of parameters [10].

1.2.2 Methods for Parameter Estimation

As the model parameters can greatly affect the model outputs, it is very important that in our model we incorporate parameter estimates that are as close as possible to their true values. Parameter estimation is a crucial yet non-trivial part of modeling of biochemical networks. Usually, unknown parameter values are obtained by fitting the model outputs to a set of experimentally measured data [3]. Under this scope, there are two main methods of parameter estimation [3]:

- Single-point estimation (Frequentist approach)
- Distribution-based inference (Bayesian inference)

In single-point estimation only a single parameter set is found (e.g. maximum likelihood, maximum a posteriori estimation). By assigning a confidence interval to this parameter set (for example by using the Fisher Information matrix), a parameter range of interest is obtained. All of the model predictions are later based only on that parameter range and no other parts of the parameter space are considered. There might, however, exist parameters that do not lie close to the chosen value, but are good enough and might be more suitable for certain cases. The concern
when using single-point estimation is proper interpretation of the results in the case of highly correlated and multimodal parameter distribution landscapes.

In **distribution-based inference** (usually carried out in a Bayesian framework), on the other hand, we are interested in the entire distribution over parameter values. (e.g. likelihood distribution, posterior distribution). This gives us an overview of what the model is capable of by knowing which parameter sets are more plausible and preferred and which should be avoided. Usually, the parameters that are considered to be “good” are those whose probability falls above certain probability threshold. In high-dimensional parameter spaces, this might include several modes (peaks of the distribution) with equally good parameter values. Whereas in the case of single-point estimation, the information about the rest of the modes is usually lost. With distribution-based inference, the capabilities of the model are explored in a more systematic way and its behaviour can be predicted in a more holistic manner.

In the Bayesian inference framework, we want to know the posterior distribution of model parameters, i.e. the distribution that summarizes our state of knowledge about the system parameters after taking into account the observed data [3]. According to the *[Bayes’ rule](#)*, this is proportional to the product of the likelihood and the prior [10].

\[
P(\theta | y) \propto p(y | \theta)\pi(\theta)
\]  

(1.2)

When we have more measurements, we usually assume that the measurement noise samples are i.i.d. (independent, identically distributed), so the likelihood given all the data points is equal to the product of the likelihoods for individual data points. If we consider for simplicity only one observable \(y\), measured at \(T\) distinct time points \((y_{t_1}, y_{t_2}, \ldots, y_{t_T})\), then the posterior is given by [3]:

\[
P(\theta | y_{t_1}, \ldots, y_{t_T}) \propto \pi(\theta) \prod_{i=1}^{T} P(y_{t_i} | \theta)
\]  

(1.3)

Working with products is sometimes difficult, especially when small quantities as probabilities have to be multiplied. This is the reason why most often these operations are done with log-probabilities instead of the probabilities themselves. Once logarithms are used, the multiplication in Eq. 1.3 becomes addition.

If more observables measured at more time points are considered, then the measurement noise for all of them is assumed to be independent and for the likelihood calculation in Eq. 1.3, a double multiplication over all the observables and all the time points is required.

The shape of the posterior parameter distribution of ODE models is usually not known. However we are frequently able to evaluate point wise a function that is proportional to it. Therefore, we can make use of Monte Carlo sampling methods to draw samples from this distribution and thus be able to approximate its form. Markov Chain Monte Carlo (MCMC) samplers are the most popular class of algorithms for carrying out this task and will be presented more analytically in the next section.

**Note:** If we assume additive Gaussian noise on the measurements, finding the maximum log-likelihood is equivalent to minimizing the sum of squared residuals between the outputs obtained by the simulations and the experimental data at specific point in time [3].

\[
logP(\theta | y_{t_1}, \ldots, y_{t_T}) \propto log\pi(\theta) + C \cdot \sum_{i=1}^{T} -\frac{1}{2}(y_{t_i} - \mu_{t_i})^T \Sigma^{-1}(y_{t_i} - \mu_{t_i})
\]  

(1.4)

where \(\Sigma\) is the covariance matrix of the Gaussian distribution, \(\mu_{t_i}\) is the model output at \(t_i\) and \(C\) is a constant.
Chapter 2

Goal

The problem of distribution-based parameter inference is a hard problem. It requires efficient estimation of probability distributions over the parameter values, which is a very challenging task especially in high-dimensional spaces where the probability is typically concentrated in regions whose volume is a tiny fraction of the total \([11, 10]\). To generate points drawn from such distributions with efficiency, the sampling procedure must search for these relevant regions. Sampling methods based on Markov chains incorporate the required search aspect in a framework where it can be proved that the correct distribution is generated, at least in the limit as the length of the chain grows. However, simple MCMC methods perform poorly on high-dimensional distributions which have strong correlations between the parameters. In the past years several more sophisticated MCMC methods have been developed that can traverse the parameter space more effectively and draw fast independent samples from the distribution \([5]\).

When used for Bayesian inference of parameter values of an ODE-based system (given some data), the MCMC methods require knowledge of the model outputs at specific points of time. Some of the methods require also knowledge of first- and second-order state sensitivities. Since, in most of the cases, the system’s equations can not be solved analytically, numerical integration is typically employed to estimate the state solutions and their sensitivities and for this specialized ODE solvers are needed (see Chapter 4).

The goal of this master thesis is:

1. to evaluate the performance of different MCMC sampling algorithms (Chapter 3) when sampling from the posterior parameter distribution of small and medium-sized ODE models. The evaluation is done in terms of convergence properties of the algorithms and also of the computational cost they have.

2. to develop an automatic framework which can sample the parameter space of any biochemical model with any of the different samplers being evaluated. The samplers are connected to specialized ODE solvers which can also calculate the state sensitivities (Chapter 4). A flowchart of the automatic framework is shown in Figure 2.1. The framework is explained in more details in Section 4.2.4.
Goal

Figure 2.1: Simplified version of the automatic framework for sampling parameter spaces of ODE models
Chapter 3

Markov Chain Monte Carlo

3.1 General

Markov Chain Monte Carlo (MCMC) sampling methods are based on the construction of a Markov chain whose equilibrium distribution is the target distribution of interest [10]. In all cases, we first construct a transition kernel of an ergodic Markov chain with the desired invariant distribution, and then simulate the chain for many steps, so that it reaches equilibrium. The states sampled after the chain has converged will be then distributed according to the target distribution of interest.

3.2 Description of the algorithms used

In this work, five different MCMC sampling algorithms were used to sample points from a target distribution $P(\theta)$ which can be evaluated point wise for any point: Metropolis-Hastings (MH), Hamiltonian Monte Carlo (HMC), Metropolis Adjusted Langevin Algorithm (MALA) and two recently proposed algorithms that exploit the Riemannian geometry of the space of parametrized distribution functions: Manifold Metropolis Adjusted Langevin Algorithm (mMALA) and Riemann Manifold Hamiltonian Monte Carlo (RMHMC).

3.2.1 Metropolis - Hastings (MH)

Throughout the thesis, the variables of interest (that need to be sampled) will be denoted by $\theta$ and the target distribution by $P(\theta)$. For a certain iteration of the sampling algorithm, the current sample of the Markov chain will be denoted by $\theta^n$ and the potential successive (candidate) sample by $\theta^*$. The simplest algorithm, Metropolis-Hastings (MH), as a proposal mechanism uses a probability density $Q(\theta^* \mid \theta^n)$ (called the proposal density) which depends on the current point $\theta^n$ [10]. The most widely used proposal density (which is also used in this thesis) is a (n-dimensional) Gaussian density centered at the current point with a covariance matrix $\Sigma$. After a candidate point is generated according to this proposal density, the values for the probabilities for the current and for the candidate point are calculated ($P(\theta^n)$ and $P(\theta^*)$ respectively). To decide whether to accept the candidate point, we compute the quantity [10]:

$$a = \frac{P(\theta^*) \cdot Q(\theta^n \mid \theta^*)}{P(\theta^n) \cdot Q(\theta^* \mid \theta^n)}$$ (3.1)

In case of a Gaussian proposal density $Q(\theta^* \mid \theta^n)$ and $Q(\theta^n \mid \theta^*)$ are equal and can be omitted from the calculation, since they will cancel out in 3.1.
Finally, the new state is accepted with probability $\min\{1, a\}$ [10]. In case of rejection the new state $\theta^*$ is set equal to the previous state $\theta^n$. This process is repeated for several steps, until the generated Markov chain converges.

When a Gaussian proposal density is used, the parameters that have to be properly tuned are the values for the elements in the covariance matrix $\Sigma$. If too small variances are used, then the sampler will move very slowly and sample only tiny local region around the starting position [17]. If the variances are too big, on the other hand, the sampler will propose points which are far from the high density regions of the distribution resulting in high rejection rate [17]. Furthermore, the target distribution might stretch in one dimension more than the other. If this information is known, it needs to be considered when choosing the variances for the different dimensions of the Gaussian proposal density. Therefore, it is very important to tune the $\Sigma$ matrix carefully.

### 3.2.2 Hamiltonian Monte Carlo (HMC)

The Hamiltonian Monte Carlo (HMC) algorithm, as the name suggests, uses Hamiltonian dynamics as a basic proposal mechanism. In addition to the points being sampled that will be referred to as position variables $\theta$, random momentum variables $p$ are generated at every iteration, which also have their own distribution (Gaussian for instance) [17, 11]. A Hamiltonian function is defined that depends on these two variables, which is a sum of potential ($E(\theta)$) and kinetic ($K(p)$) energy terms [11].

$$H(\theta, p) = E(\theta) + K(p)$$  \hspace{1cm} (3.2)

According to the concept of a canonical distribution from statistical mechanics, the relationship between the joint density of the position and momentum variables $P(\theta, p)$ and the Hamiltonian function is given by

$$P(\theta, p) = \frac{1}{Z_H} e^{-H(\theta, p)} = \frac{1}{Z_H} e^{-E(\theta)} e^{-K(p)}$$  \hspace{1cm} (3.3)

in which $Z_H$ is a normalizing constant [11].

When we sample with HMC, we actually draw points from the joint density $P(\theta, p)$. However, since this density is separable, the marginal density of the position variables $\theta$ is our density of interest [11].

$$P(\theta) = \frac{1}{Z_E} e^{-E(\theta)}$$  \hspace{1cm} (3.4)

It follows that the potential energy is equal to:

$$E(\theta) = -\log P(\theta) - \log Z_E$$  \hspace{1cm} (3.5)

For the kinetic energy function usually quadratic form is considered; however, other forms of energy functions can be also used [17]:

$$K(p) = \frac{p^T M^{-1} p}{2}$$  \hspace{1cm} (3.6)

Here $M$ is a symmetric, positive-definite mass matrix [17]. If $M$ is diagonal, we can interpret its elements as the relative “stiffness” or “mass” associated to different directions in the state space.

The HMC algorithm is summarized with a simple MATLAB code in Listing 3.1. (note that in the code the position variables are denoted by “$q$” and the momentum variables by “$p$”).

In each iteration of the HMC algorithm, two proposals are made [17, 10]:

- Random proposal for the momentum - always accepted (line 5 in the code)
- Deterministic proposal for the position - accepted according to the standard M-H accept-reject rule
The deterministic proposal for the position variable is done by solving the Hamiltonian equations for the momentum and position variables given by [17]:

\[
\frac{d\theta_i}{dt} = \frac{\partial H(\theta, p)}{\partial p_i} = [M^{-1}p]_i \tag{3.7}
\]

\[
\frac{dp_i}{dt} = \frac{\partial H(\theta, p)}{\partial \theta_i} = -\frac{\partial E(\theta)}{\partial \theta_i} \tag{3.8}
\]

The equations are solved by numerical integration (most often by using the leapfrog algorithm for \(L\) steps, with step size \(\epsilon\)). For this, see lines 14 - 35 in the code; or more precisely \(L\) iterations of the algorithm are performed in lines 24 - 31. During these \(L\) iterations the Hamiltonian equations for the position (line 25) and momentum (line 29) variables are solved alternately. For more details about the leapfrog algorithm see [11] and [17]. The proposed position variable is the final position value that is obtained after the \(L\) steps of the leapfrog algorithm (i.e. at the end of the simulated Hamiltonian trajectory).

If the Hamiltonian trajectory is simulated perfectly, then the final position value is accepted with acceptance probability of 1 [11]. In that case, the trajectory moves on a hypersurface of constant probability density (isocontour of the joint probability density) [11]. However, since the numerical integration is not perfect, to account for the errors made while integration, the proposed position variable is accepted according to the standard M-H accept-reject rule (lines 44 - 51 in the code). If the step size is too large or the trajectory is too long, the chances of error are higher. Occasional rejections make sure that points are sampled from the correct distribution [11].

The equation for the momentum involves calculation of the gradient of the potential energy function with respect to the position variables (line 26). The advantage of HMC lies in its ability, by using Hamiltonian dynamics and the gradient of the potential energy function, to travel long distances in the parameter space and still have a high acceptance probability [11]. This long distance travel reduces the random walk behaviour. However, in order to show its best performance behaviour, the HMC algorithm first needs to be properly tuned. The three parameters that require tuning are the step size \(\epsilon\), the number of leapfrog integration steps \(L\) and the values in the mass matrix \(M\), which mimic choosing different step sizes for different dimensions of the parameter space. This matrix is very hard to tune and this is usually done by trial-and-error procedure, especially for high-dimensional distributions whose shape is not known. The smaller the step size, the smaller the integration error and thus higher the acceptance rate. In order to compensate for the short moves due to the small step size, more leapfrog integrations (big \(L\)) need to be used, so that at the end of the Hamiltonian trajectory we can move further away. The increase of the step size is accompanied by increase in the integration error and in that case smaller \(L\) should be used. Once the step size gets too big and crosses a certain threshold, the Hamiltonian trajectory becomes unstable [11, 17].

Figure 3.1(a) presents combinations of \(\epsilon\) and \(L\) values that result in good sampling (judged by a certain efficiency measure) of a certain known target distribution. The lower the efficiency measure, the better the sampling properties of the algorithm. In the figure, for example, it can be seen that very small \(\epsilon\) values require big \(L\) and very big \(\epsilon\) values need smaller \(L\). There is also a range of \(\epsilon\) values in the middle, that can combine well with any \(L\) and still produce good results.
Listing 3.1: MATLAB code for HMC

function [new_q, new_P] = HMC(current_q, current_P)

% generate random momentum variables
p = randn(1,length(current_q));

% Calculate E(q) and K(p)
current_E = -log(current_L);
current_K = sum((current_p*inv(M)*current_p')/2);

% calculates the gradient of E(q)
current_gradE = calculateGrad(current_E)

% _____________________ LEAPFROG ALGORITHM _____________________

% at the begining of the trajectory
proposed_q = current_q;

% Make a half step for the momentum at the begining
p = p - (epsilon * current_gradE /2);

% _______ Iterate L times _______
for i= 1:L
    proposed_q = proposed_q + epsilon * p*inv(M);
    proposed_gradE = calculateGrad(proposed_q);
    if i ~= conf.L
        p = p - (epsilon * proposed_gradE);
    end;
end;

%Make a half step for the momentum at the end
p = p - (epsilon * proposed_gradE/2);

% evaluate the probability function at the proposed position variable
proposed_P = calculateP(proposed_q);

proposed_E = -log(proposed_P);
proposed_K = sum((p*inv(M)*p')/2);

%accept_reject
a = min(1, exp(- proposed_E - proposed_K + current_E + current_K ));
if a >= rand;
    new_q = proposed_q; % accept
    new_P = proposed_P;
else
    new_q = current_q; % reject
    new_P = current_p
end;
%
};label{lst:code}
Figure 3.1: (a) Combinations of $\epsilon$ and $L$ and the corresponding efficiency measure. Combinations associated with low values of the efficiency measure (KL divergence), result in good sampling properties. (b) Five iterations of one dimensional HMC in $\theta - p$ space. The red square marks the starting position. The blue crosses on the x-axis show the sampled position points.

Figure 3.1(b) shows ten iterations of 1D HMC in $\theta - p$ space, each having ten leapfrog steps with a step size of 0.1. The target distribution of the position variables is 1D Gaussian with zero mean and a variance of 2. The blue crosses lying along the x-axes are the accepted (thus sampled) position variables. The starting point is at $\theta = 3$ (marked with a red square). Random jumps (vertical lines) can be noticed from one to another isocontour of the probability density at the beginning of each iteration due to the randomization of the momenta.

After each random jump Hamiltonian trajectory follows (the long arcs that lie on the same isocontour of the probability density). It is noticeable how the first trajectory ends up very far away from its starting point. The proposed position variable at the end of this trajectory (around $-3$) is accepted and is very distant from its predecessor, thus autocorrelation is avoided and good mixing is enhanced, which is the main strength of HMC. The piecewise deterministic nature of the algorithm is clearly visible in this figure.

Let’s quickly come back to equation 3.8. What is interesting in this equation is the derivative of the potential energy function with respect to the position variables $\partial E(\theta)/\partial \theta_i$. When the target distribution is the likelihood distribution for an ODE model, the position variables are the model parameters and the derivative of the potential energy (following from Eq. 3.5) is the derivative of the negative log-likelihood function with respect to the parameters. Since the log-likelihood is dependent only on the states $x$ and the states are dependent on the parameters $\theta$, the gradient of the negative log-likelihood with respect to the parameters would be a product of its gradient with respect to the states and the derivative of the states with respect to the parameters, which is nothing more than the state sensitivities $\partial x/\partial \theta$.

$$\frac{\partial(- \log P(y \mid \theta))}{\partial \theta} = \frac{\partial (- \log P(y \mid \theta))}{\partial x} \frac{\partial x}{\partial \theta} \quad (3.9)$$

The calculation of the sensitivities is not trivial, since it involves symbolic differentiation and numerical computation (see Chapter 4), and for their computation specialized ODE solvers are preferred.

### 3.2.3 Metropolis Adjusted Langevin Algorithm (MALA)

The Metropolis Adjusted Langevin Algorithm - MALA makes proposals according to the Langevin equation [5], which has a deterministic term (drift) which consists of the gradient of the target
distribution and a stochastic term (diffusion) which adds a noise sample to the deterministic movement:

\[ \theta^* = \theta^n + \frac{\epsilon^2}{2} \frac{\partial P(\theta^n)}{\partial \theta^n} + \epsilon \cdot z^n \]  

(3.10)

where \( z \sim \mathcal{N}(z \mid 0, I) \) and \( \epsilon \) is the integration step size [5]. The proposal density in this case is:

\[ Q(\theta^* \mid \theta^n) = \mathcal{N}(\{\theta^* \mid \mu(\theta^n, \epsilon), \epsilon^2 I\}) \]  

(3.11)

with

\[ \mu(\theta^n, \epsilon) = \theta^n + \frac{\epsilon^2}{2} \frac{\partial P(\theta^n)}{\partial \theta^n} \]  

(3.12)

After this, the following steps of the MALA algorithm are exactly the same as in MH.

Taking into consideration the fact that noise is added every time when a proposal is made, MALA does not have that big advantage due to the gradient information as HMC. Same as MH, MALA may also exhibit random walk behavior.

### 3.2.4 Manifold Metropolis Adjusted Langevin Algorithm (MALA)

The next two samplers are a bit more advanced:

The formal definition of distance between two parametrized density functions \( p(y; \theta) \) and \( p(y; \theta + \delta \theta) \) was defined by Rao(1945) as \( \delta \theta^T G(\theta) \delta \theta \), with \( G(\theta) \) being the expected Fisher information matrix [5, 15]:

\[ G(\theta) = -E_{y\mid \theta} \left[ \frac{\partial^2}{\partial \theta^2} \log P(y \mid \theta) \right] = \text{cov} \left[ \frac{\partial}{\partial \theta} \log P(y \mid \theta) \right] \]  

(3.13)

Rao noted that the Fisher information matrix \( G(\theta) \) is positive definite by definition and is a position-specific metric of a Riemann manifold, which leads to the conclusion that the space of parametrized probability density functions is endowed with a natural Riemann geometry [5]. He therefore noted further that expressions for the curvature of the manifold and geodesics on the manifold between two densities could be derived [5, 15].

Girolammi and Calderhead used this idea to propose two novel MCMC sampling algorithms which exploit the properties of the Riemann geometry to make effective proposals: manifold Metropolis Adjusted Langevin Algorithm (mMALA) and Riemann Manifold Hamiltonian Monte Carlo (RMHMC) [5].

In case the desired distribution is the posterior distribution of the parameters, the prior needs to be also included in the calculations of the metric tensor. This is done by adding the negative Hessian of the log-prior to the Fisher information matrix:

\[ -E_{y\mid \theta} \left[ \frac{\partial^2}{\partial \theta^2} \log P(y, \theta) \right] = -E_{y\mid \theta} \left[ \frac{\partial^2}{\partial \theta^2} \log \{ P(y \mid \theta) \pi(\theta) \} \right] = G - \frac{\partial^2}{\partial \theta^2} \log \pi(\theta) \]  

(3.14)

The Fisher information matrix defines a local distance measure effectively allowing the MCMC proposals to be based on the curvature of the manifold, which is defined by the parameter sensitivities [6]. The matrix functions as a mass matrix that varies at each point; it adapts itself according to the local structure.

If we consider a Gaussian noise model with variance \( \sigma_n^2 \) for each of the \( N \) observed states, and measurements made at \( T \) distinct time points, using the expected Fisher information matrix we can obtain analytical expressions for the metric tensor and its derivatives in terms of the first-and second-order sensitivities [5]:

\[ G(\theta)_{ij} = \sum_{n=1}^N S_{i,n} S_{j,n}^{-1} S_{i,n} \]  

(3.15)
\[
\frac{\partial G(\theta)^{ij}}{\partial \theta_k} = \sum_{n=1}^{N} \frac{\partial S^i_n}{\partial \theta_k} \Sigma_n^{-1} S_n^{ij} + S^{i,j}_n \Sigma_n^{-1} \frac{\partial S^j_n}{\partial \theta_k}
\]  
(3.16)

In this equation \(G(\theta)^{ij}\) denotes the \((i,j)\)th entry of the Fisher information matrix, \(S^i_n = \partial y_n / \partial \theta_i\), represents T-dimensional vector of first-order sensitivities for the \(n\)th component of the observed state relative to its \(i\)th parameter given for \(T\) time points, \(\partial S^i_n / \partial \theta_k\) represents a derivative of \(S^i_n\) w.r.t. the parameter \(\theta_k\) (i.e. second-order sensitivities at \(T\) time points) and \(\Sigma_n = I_T \sigma_n^2\) [5].

Thus, Riemann Manifold MCMC methods via the Fisher information matrix use local sensitivity information of the system in order to design proposal mechanisms.

Manifold Metropolis Adjusted Langevin Algorithm (mMALA) is very similar to the normal MALA, with the Langevin diffusion being defined on a Riemann manifold [5]. The proposal mechanism is as following [5]:

\[
\theta^*_i = \theta^i + \frac{\epsilon^2}{2} \left( G^{-1}(\theta^0) \triangledown \theta \mathcal{L}(\theta^0) \right)_i - \epsilon^2 \sum_{j=1}^{D} \left( G^{-1}(\theta^0) \frac{\partial G(\theta^0)}{\partial \theta_j} G^{-1}(\theta^0) \right)_i
\]

\[
+ \frac{\epsilon^2}{2} \sum_{j=1}^{D} \left( G^{-1}(\theta^0) \right)_i \text{tr} \left( G^{-1}(\theta^0) \frac{\partial G(\theta^0)}{\partial \theta_j} \right) + \{ \epsilon \sqrt{G^{-1}(\theta^0)} z^n \}_i = \mu(\theta^0, \epsilon)_i + \{ \epsilon \sqrt{G^{-1}(\theta^0)} z^n \}_i
\]

(3.17)

with proposal density [5]:

\[
Q(\theta^* | \theta^0) = \mathcal{N}(\theta^* | \mu(\theta^0, \epsilon), \epsilon^2 G^{-1}(\theta^0))
\]

(3.18)

In Eq. 3.17 \(\mathcal{L} \equiv \log P(\theta)\) and \(\sqrt{G^{-1}}\) can be obtained by diagonalization of \(G^{-1}\) or via Cholesky decomposition such that \(\sqrt{G^{-1}} = U U^T\) and \(\sqrt{G^{-1}} = U\) [5].

The steps that follow in this algorithm are same as in MH and MALA.

### 3.2.5 Riemann Manifold Hamiltonian Monte Carlo (RMHMC)

Riemann manifold Hamiltonian Monte Carlo (RMHMC) algorithm follows similar concepts as in HMC, but the Hamiltonian function explained in HMC previously is defined on a Riemann manifold instead [5]:

\[
H(\theta, p) = E(\theta) + K(p, \theta) = -\mathcal{L}(\theta) + \frac{1}{2} \log \{(2\pi)^D | G(\theta) \} + \frac{1}{2} p^T G(\theta)^{-1} p
\]

(3.19)

Note here that the normalization constant for the potential energy \(\frac{1}{2} \log \{(2\pi)^D | G(\theta) \}\) is also included in the Hamiltonian function, since it depends upon the position variables \(\theta\). The kinetic energy \(K(p, \theta)\) depends not only on the \(p\), but also on \(\theta\).

Therefore, the Hamiltonian dynamics is defined by the following equations for the position and momentum [5]:

\[
\frac{d\theta^i}{d\tau} = \frac{\partial H}{\partial p^i} = \{ G(\theta)^{-1} p \}_i
\]

(3.20)

\[
\frac{dp^i}{d\tau} = -\frac{\partial H(\theta)}{\partial \theta^i} = -\frac{\partial \mathcal{L}(\theta)}{\partial \theta^i} - \frac{1}{2} \text{tr} \left( G(\theta)^{-1} \frac{\partial G(\theta)}{\partial \theta^i} \right) + \frac{1}{2} p^T G(\theta)^{-1} \frac{\partial G(\theta)}{\partial \theta^i} G(\theta)^{-1} p
\]

(3.21)

For numerically integrating the above equations, the leapfrog algorithm used in HMC can not be used, since it may violate some of the properties of MCMC chains [5]. Suitable algorithm for this scheme would be the “generalized leapfrog algorithm” whose equations are given as following [5]:

\[
p(\tau + \frac{\epsilon}{2}) = p(\tau) - \frac{\epsilon}{2} \nabla \theta \nabla \theta H(\theta(\tau), p(\tau + \frac{\epsilon}{2}))
\]

(3.22)

\[
\theta(\tau + \epsilon) = \theta(\tau) + \frac{\epsilon}{2} [\nabla_p H(\theta(\tau), p(\tau + \frac{\epsilon}{2})) + \nabla_p H(\theta(\tau + \epsilon), p(\tau + \frac{\epsilon}{2}))]
\]

(3.23)
\begin{equation}
    p(\tau + \epsilon) = p(\tau + \frac{\epsilon}{2}) - \frac{\epsilon}{2} \nabla_{\theta} H\{\theta(\tau + \epsilon), p(\tau + \frac{\epsilon}{2})\} \tag{3.24}
\end{equation}

In this context, equations 3.22 and 3.23 require fixed point iterations to be solved until convergence, since the same expressions appear on the left and on the right hand side of the equation \[5\]. Every fixed point iteration for the position variables 3.23 requires calculation of the first- and second-order state sensitivities to be used for the Fisher information matrix and its derivative calculation. This adds to the computational complexity of the algorithm.

A brief summary of the so far described algorithms is given in the following table:

<table>
<thead>
<tr>
<th>MCMC sampler</th>
<th>Parameters to be tuned</th>
<th>Sensitivity information used</th>
</tr>
</thead>
<tbody>
<tr>
<td>MH</td>
<td>$\Sigma$</td>
<td>no sensitivity information</td>
</tr>
<tr>
<td>HMC</td>
<td>$\epsilon, L, M$</td>
<td>$1^{st}$ order sensitivities</td>
</tr>
<tr>
<td>MALA</td>
<td>$\epsilon$</td>
<td>$1^{st}$ order sensitivities</td>
</tr>
<tr>
<td>mMALA</td>
<td>$\epsilon$</td>
<td>$1^{st}$ and $2^{nd}$ order sensitivities</td>
</tr>
<tr>
<td>RMHMC</td>
<td>$\epsilon, L$, number of fixed point iterations</td>
<td>$1^{st}$ and $2^{nd}$ order sensitivities</td>
</tr>
</tbody>
</table>

### 3.3 Performance Criteria

The implementation of different sampling algorithms requires performance comparison between them. When evaluating the performance of these algorithms it is important to consider two things: 1. whether the Markov chain has reached stationarity and 2. how fast.

A good sampler is one that quickly converges to the target distribution. When the target distribution is known, the correspondence of the sampled distribution to it can be verified. However, in many cases we do not have information about the shape of the target distribution and the correctness of the samples could be inferred only by examining the convergence properties of the chain.

If we want to verify that the chain converges to an invariant distribution, it is advisable to run several chains from different starting points and check whether they always converge to the same mean values.

It is also important that the drawn samples are minimally correlated, i.e. that the chain mixes fast. To avoid biasing the results, it is a common practice to discard the initial iterations as they are too strongly influenced by starting values and do not provide good information about the target distribution [17]. The number of discarded iterations is referred to as “burn-in” period [17].

For the comparison of the samplers used, their general convergence properties were examined and additionally, when the target distribution was known, their divergence from the true distribution was examined as well. As the computer time needed for the generation of the samples is also important, it was also taken into an account when considering some of the criteria.

As a general indicator of the speed of convergence, we examined the qualitative behaviour of the traces of the variables (whether they mix well or not), their running averages (how does the average value of the variables changes as the chain evolves), 2D scatter plots of the sampled variables (when possible), 3D histograms (when possible). Quantitative measures were also considered such as autocorrelation and effective sample size.

The autocorrelation measures how correlated the samples are and in an ideal case we expect them to be independent from their predecessors. In order to define the autocorrelation quantitatively, several other terms need to be introduced first.
Let’s suppose that \( x_1, x_2, \ldots, x_n \) is stationary Markov chain. The quantity
\[
\gamma_k = \text{cov}\{x_i, x_{i+k}\}
\] (3.25)
is called the lag-\( k \)-autocovariance of the samples \( x_1, x_2, \ldots \). The variance of the chain with respect to their mean value is given by [17]:
\[
\sigma^2 = \text{var}\{x_i\} + 2 \sum_{k=1}^{\infty} \text{cov}\{x_i, x_{i+k}\} = \gamma_0 + 2 \sum_{k=1}^{\infty} \gamma_k
\] (3.26)
The function \( k \mapsto \gamma_k \) is called the autocovariance function of the samples \( x_1, x_2, \ldots \), and the function \( k \mapsto \gamma_k / \gamma_0 \) is their autocorrelation function [17].

The natural estimator of the autocovariance function is [17]:
\[
\hat{\gamma}_k = \frac{1}{n} \sum_{i=1}^{n-k} [x_i - \hat{\mu}_n][x_{i+k} - \hat{\mu}_n]
\] (3.27)
where \( \hat{\mu}_n \) is the estimated mean value of the sampled variables:
\[
\hat{\mu}_n = \frac{1}{n} \sum_{i=1}^{n} x_i
\] (3.28)

For ideally independent samples, as the sample chain grows their autocorrelation should drop to zero.

A nice way of measuring independence is to measure effective sample size (ESS), which measures the approximate number of independent samples in a group of dependent ones.
\[
\text{ESS} = \frac{N}{1 + 2 \sum_k \gamma(k)}
\] (3.29)
where \( N \) is the number of samples and \( \sum_k \gamma(k) \) is the sum of the \( K \) monotone sample autocorrelations, estimated by the initial monotone sequence estimator [5, 4].

Later, this value can be normalized relative to the CPU time needed for the completion of a given number of iterations to obtain a value for the CPU time per independent sample [5]. The smaller this value, the faster the algorithm converges.

Sometimes, autocorrelation and other convergence properties do not indicate that the chain has converged to the target distribution. The chain might converge, but be stuck in a local peak of the distribution. As a measure of how different the sampled distribution \( Q(\theta) \) from the target distribution \( P(\theta) \) is, Kullback-Leibler divergence (\( D_{KL} \)) [8] and Hellinger distance (\( H \)) measures were used:
\[
D_{KL}(P||Q) = \sum_i P(i) \ln \frac{P(i)}{Q(i)}
\] (3.30)
\[
H^2(P, Q) = 1 - \int \sqrt{P(x)Q(x)} \, dx
\] (3.31)
It should be noted that these can be calculated only when the target distribution is known. However, the target distributions of the parameter spaces of high-dimensional models is very often not known. As mentioned previously, in this case we can verify that the chain has converged to the target distribution by running several chains in parallel starting from different initial points and checking if they converge to the same mean. If this is not the case, the target distribution is probably complex and multimodal and the chains probably converge only to a local mode. The problem of multimodality and sampling of only a single mode can be tackled by so called population MCMC methods.

The criteria used to visualize and examine the MCMC outputs are summarized below:
• **Convergence properties**
  Qualitative
  * Traces of the sampled variables
  * Running averages
  * Scatter plots/histograms of sampled variables
  Quantitative
  * Autocorrelation
  * Effective sample size

• **Divergence from the target distribution** (only when the entire target distribution is known).
  Kullback-Leibler divergence
  Hellinger distance
Chapter 4

Embedding the MCMC Samplers into an Automatic Framework

4.1 Sensitivity Analysis

As discussed earlier, many kinetic parameters in a model are associated with a degree of uncertainty. Sensitivity analysis is a method to determine the effect of uncertain parameters on system solutions [2]. A sensitivity of a state $x$ with respect to a certain parameter $\theta$ is a partial derivative of $x$ with respect to that parameter. It tells us how the state concentration is affected by changes in the parameter. A parameter is considered to be sensitive if a small change in its value can make big change in the outputs [2].

Let’s consider a system of ODEs, having solutions $x$ that are dependent upon a certain single rate parameter $\theta$, and initial conditions $x(t_0) = x_0$:

$$\frac{\partial x_i}{\partial t} = f_i(x_1, x_2, \ldots, x_n, t, \theta), \quad i = 1, \ldots, n$$  \hspace{1cm} (4.1)

The state sensitivities are then given by:

$$S_i = \frac{\partial x_i}{\partial \theta}$$  \hspace{1cm} (4.2)

The second-order state sensitivities represent the rate of change of the first-order sensitivities:

$$\dot{S}_i = \frac{\partial^2 x_i}{\partial \theta^2}$$  \hspace{1cm} (4.3)

Methods of solving the $1^{st}$ and $2^{nd}$ order sensitivities are discussed below.

4.2 Embedding the Sampler into a Framework

As it was described in Chapter 1.2, model parameters can be best estimated by fitting the results obtained by model simulations to the experimental data. Equation 1.4 describes how the likelihood of the parameters is calculated, given a particular data set with an additive Gaussian noise, which corresponds to the method of finding least squared residuals between the experimentally observed data and the state outputs obtained by solving the system of ODEs with the particular parameter set at $T$ time measurements.

Likelihood calculations are required by all of the described MCMC algorithms. Furthermore, all algorithms except MH need to calculate the gradient of the log-likelihood and for that calculation
of the state sensitivities is required (refer to Eq. 3.9). The Riemann manifold methods require the sensitivities also for the calculation of the $G$ matrix (Eq. 3.15) and its derivative (Eq. 3.16) along with the 2\textsuperscript{nd} order sensitivities. Next sections discuss how the states and their sensitivities are calculated and how those calculations are integrated within a framework together with the MCMC samplers used.

### 4.2.1 Computing the States

The change of the state concentrations in time is governed by their ODEs (see Section 1.1). An ODE system as in Eq. 4.1 can be written in a matrix notation as:

$$\frac{dx}{dt} = A \cdot x, \quad x_0 = \sum_{i=1}^{N} \alpha_i v_i \quad (4.4)$$

where $A$ is referred to as “system matrix”.

For linear and homogeneous ODE systems, the state solutions for different time points can be then derived analytically by the formula [3]

$$x(t) = \sum_{i=1}^{N} \alpha_i v_i e^{\lambda_i t} \quad (4.5)$$

where $v_i$ is the eigenvectors of the system matrix $A$ and $\lambda_i$ are their corresponding eigenvalues [3]. This gives us the general solution for $x(t)$. A particular solution specifying the coefficients $\alpha_i$ can be found by considering the initial conditions [3].

For bigger, non-linear systems, however, a closed-form formula for the system’s solution is hard to compute and in that case, the state concentrations at different time points need to be found by numerical integration, with the help of ODE solvers such as \texttt{ode15s}, \texttt{ode45} embedded in the MATLAB framework, or other specialized solvers like CVODE which is part of the SUNDIALS software suite [16], or \texttt{odeSD} [13].

Table 4.1 compares three different solvers in terms of the CPU time they needed to calculate state concentrations of a small ODE model (described in more details in Section 5.2.1) at 33 time points at $t = [0.3 : 0.3 : 9.9]$ . Column 2 of the table shows the total CPU time for 100 solver runs, and column 3 the average CPU time over the same 100 runs.

<table>
<thead>
<tr>
<th>Solver</th>
<th>total CPU for 100 runs</th>
<th>average CPU over 100 runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVODE</td>
<td>0.7557</td>
<td>0.0076</td>
</tr>
<tr>
<td>\texttt{ode15s}</td>
<td>4.4806</td>
<td>0.0448</td>
</tr>
<tr>
<td>\texttt{odeSD}</td>
<td>7.1092</td>
<td>0.0711</td>
</tr>
</tbody>
</table>

The fastest solver among these three is CVODE and thus it was most frequently used in this work whenever state calculations were needed.

### 4.2.2 Computing the 1\textsuperscript{st} Order Sensitivities

Along with the states, for HMC, MALA, mMALA and RMHMC the 1\textsuperscript{st} order sensitivities are needed for the calculation of the gradient of the log-likelihood and for mMALA and RMHMC, additionally, for the calculation of the $G$ matrix and its partial derivatives.

There are several methods for calculation of the 1\textsuperscript{st} order sensitivities:

- Finite Differences Approximation
Embedding the MCMC Samplers into an Automatic Framework

- Direct Method (Forward Method)
- other methods (not considered in this work)

The simplest method of finding the sensitivities is approximation by finite differences [7]. But since it is only an approximation of the true solution, it is not very accurate and it should be used with caution. A partial derivative of a function with respect to some parameter can be approximated by the differences of the outputs of the function when two different, but very similar parameter values are used [7]:

\[
\frac{\partial x(t, \theta)}{\partial \theta} \approx \frac{x(t, \theta + h) - x(t, \theta)}{h} \quad (4.6)
\]

Here, \(h\) is a small increment of the parameter value \(\theta\). So, the sensitivities can be found by calculating the states obtained with parameters \(\theta\) and \(\theta + h\), taking their difference and dividing by \(h\).

The most popular and mathematically sound way of calculating sensitivities is the Direct Method [2]. By this method, the sensitivities at specific point of time are found by solving their equations. As the state concentration changes in time governed by its ODE, the sensitivities of this state also change in time, governed by their own ODE equations with the following general form [2]:

\[
\frac{dS_i}{dt} = \frac{d}{dt} \frac{\partial x}{\partial \theta} = \frac{\partial f_n(x(t), \theta, x_0)}{\partial \theta} = \sum_{l=1}^{N} \frac{\partial f_{l,n}}{\partial x} S_{l,t} + \frac{df_{l,n}}{\partial \theta} \quad (4.7)
\]

In matrix notation this can be written as [2]:

\[
\dot{S} = J_x S + J_p \quad (4.8)
\]

\(J_x\) is the Jacobian matrix of the system with respect to the states and \(J_p\) is the Jacobian with respect to the parameters.

The most straightforward way of solving the sensitivity ODEs is by augmenting the original ODE system with the sensitivities equations (as in 4.7) and solving for the original and augmented equations simultaneously [2] (for instance with some of the solvers discussed in 4.2.1). The sensitivities will be then solutions to a new set of \(n\) differential equations (if only one parameter is considered) [2].

However, except for very small ODE models, it is very difficult for these equations to be derived analytically and also, the augmentation is computationally very expensive.

Specialized ODE solvers have been developed which can solve the 1st order sensitivities ODEs faster and more efficient than numerically integrating the augmented system. Two such specialized solvers used in this work are CVODES and odeSD.

CVODES (a C-language package) is a part of the SUNDIALS suite. It has sensitivity analysis capabilities which are a superset of the capabilities of CVODE) [16]. It solves the IVP by one of two numerical methods: the backward differentiation formula (BDF) and the Adams-Moulton formula [16] (not discussed in details here).

The solver odeSD solves the IVP by using second-derivative integrator [13]. The higher accuracy second-derivative rule enables, in combination with a better error estimate, to take larger steps, which allows us to decrease the number of otherwise expensive sensitivity calculations [13].

odeSD is implemented in Matlab as well as in the C programming language (a faster version called odeSDmex) [13]. The ODE systems that we considered were too stiff for odeSDmex, thus giving integration errors. Therefore the C version of odeSD could not be used in this work.

The Table 4.2 shows comparison of four methods for calculating the 1st order sensitivities in terms of their running time: finite differences, CVODES, odeSD and augmenting the original system with sensitivity ODE equations and solving with ode15s. Again, 100 runs were done on the same small system.
Since the finite difference method in not stable, that method was excluded from our possible alternatives from the start. The other three methods are sufficiently accurate and the fastest of them turned out to be CVODES, which was further used for the computation of the 1st order sensitivities, whenever needed.

<table>
<thead>
<tr>
<th>Methods</th>
<th>total CPU 100 runs</th>
<th>average CPU 100 runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVODES 1st</td>
<td>1.7733</td>
<td>0.0177</td>
</tr>
<tr>
<td>ode15s 1st order sensitivities</td>
<td>4.6909</td>
<td>0.0469</td>
</tr>
<tr>
<td>finite differences 1st order sensitivities</td>
<td>2.0264</td>
<td>0.0203</td>
</tr>
<tr>
<td>odeSD 1st order sensitivities</td>
<td>11.7179</td>
<td>0.1172</td>
</tr>
</tbody>
</table>

### 4.2.3 Computing 2nd Order Sensitivities

When sampling with mMALA and RMHMC, second-order sensitivity information is needed for the calculation of the derivative of the $G$ matrix (3.16).

So far, efficient methods for calculating 2nd order sensitivities have not been designed. Neither can any of the specialized ODE solvers mentioned before compute 2nd order sensitivities of the states directly.

There are some existing methods, however, that can compute 2nd order derivatives of ODE-embedded functionals. The Adjoint Method for example (which is implemented in CVODES), doesn’t directly compute the state sensitivities, but it computes derivatives of some functional that depends on the states with respect to the parameters instead [16], i.e. it calculates the gradient $dG/d\theta$ of [16]:

$$G(\theta) = \int_{t_0}^{t_f} g(t, x, \theta) dt$$  \hspace{1cm} (4.9)

In the context of sampling, such functional of the states is the likelihood function.

This is not what is required in the samplers though. We need 2nd order sensitivities of individual states with respect to individual parameters separately, so that as such, they can be included in the calculation of the partial derivative of the $G$ matrix in Eq. 3.16.

We tackled this issue by augmenting the original ODE system with its 1st order sensitivities equations (Eq. 4.7) and letting CVODES to find the 1st order sensitivities of the augmented system by the Direct Method. The 1st order sensitivities of the augmented 1st order sensitivity equations are in fact the required 2nd order sensitivities of the states. The same trick was applied for odeSD.

Alternative way of 2nd order sensitivity calculation is naturally, finite differences method, which was not considered due to its inaccuracy and instability.

Moreover, the 2nd order sensitivities can be also found, similarly as the 1st order ones, with the Direct Method by augmenting the original system of ODEs with the ODEs for the 2nd order sensitivities, and then again using one of the solvers in 4.2.1 to solve for the states and 1st and 2nd order sensitivities simultaneously. The general form of the ODEs for the 2nd order sensitivities is as following [5]:

$$\frac{\partial S_{i,n}^k}{\partial \theta_l} = \sum_{l=1}^{N} \{ (\sum_{m=1}^{N} \frac{\partial^2 f_{l,n}}{\partial x_l \partial x_m} S_{m}^k + \frac{\partial^2 f_{l,n}}{\partial \theta_l \partial \theta_k} S_{l}^i) S_{l,t}^i + \frac{\partial f_{l,n}}{\partial x_l} \frac{\partial S_{l,t}^i}{\partial \theta_k} \} + \sum_{l=1}^{N} \frac{\partial f_{l,n}}{\partial \theta_l} S_{l,t}^i + \frac{\partial^2 f_{l,n}}{\partial \theta_l \partial \theta_k}$$  \hspace{1cm} (4.10)

The table below shows comparison of the different methods for 2nd order sensitivity calculation mentioned previously.
Table 4.3: Comparison between different methods for 2nd order sensitivity calculation

<table>
<thead>
<tr>
<th>Methods</th>
<th>total CPU 100 runs</th>
<th>average CPU 100 runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVODES 1st + 2nd order sensitivities</td>
<td>1.9665</td>
<td>0.0197</td>
</tr>
<tr>
<td>finite differences 2nd order sensitivities</td>
<td>5.2475</td>
<td>0.0525</td>
</tr>
<tr>
<td>ode15s 1st + 2nd order sensitivities</td>
<td>5.2893</td>
<td>0.0529</td>
</tr>
<tr>
<td>odeSD 1st + 2nd order sensitivities</td>
<td>13.2282</td>
<td>0.1323</td>
</tr>
</tbody>
</table>

Similarly to the results in Table 4.3, the fastest method for the 2nd order sensitivity calculation is with CVODES and it was therefore used for that purpose when necessary.

4.2.4 Overview of the Framework

As described previously, when the MCMC algorithms are used to sample from parameter distributions, system simulations are necessary. For this purpose an automatic framework was developed which connects the sampling algorithms with specialized ODE solvers. The framework can take as an input any ODE model and give as an output a distribution (posterior or the likelihood) of parameter values (after a number of iterations). The automatic framework is described in details below and a flowchart of it is presented in Figure 4.1:

![Flowchart of the automatic framework](image)

As an input to the framework any ODE model can be taken in standard SMBL format, which is parsed to extract some information and produce some files which are later needed by the ODE solver.

The produced files (which are necessary for the computation of the states and their sensitivities) include the right-hand sides of the ODEs for the states and sensitivities, as well as the state and parameter Jacobians of the system. These expressions are found by using the Matlab Symbolic Toolbox.

Together with the ODE model, naturally, as an input to the framework a set of experimental data (observables) measured at different time points should be given.
At the beginning of the sampling procedure, one of the five MCMC samplers should be specified that is to be used for the sampling, together with its specific tuned parameters. Also, one of the two solvers should be specified (CVODES or odeSD) to be used.

At every iteration of the sampling algorithm, after a new sample has been proposed (a new parameter set), the sampler interacts with the solver and the system is simulated with this parameter set producing new outputs (by using the previously produced files). The new outputs are then used together with the experimental data to calculate the likelihood for the proposed parameter set. The calculated likelihood is later included in the acceptance probability and then an accept-reject step is performed. For some of the samplers, together with the state outputs, 1\textsuperscript{st} and 2\textsuperscript{nd} order sensitivities are also calculated which are later used in the proposal mechanism for the next parameter set. Whether on not the 1\textsuperscript{st} and/or the 2\textsuperscript{nd} order sensitivities will be also calculated along with the states depends on the choice of the sampler, which is specified at the beginning. The entire procedure is repeated for many iterations until at the end, a distribution of sampled parameter values is obtained.
Chapter 5

Results

Setup: All the algorithms and functions needed for this work were implemented in MATLAB. The ODE solver used for producing all the results was CVODE/S. The performance measurements were run on a 64-bit Windows 7 Enterprise machine with an Intel(R) Core(TM) i5 processor (2.67GHz) and 8 GB RAM.

The codes for the five MCMC samplers (for the small ODE model) are given in Appendix A. The codes for the main function and the configuration file are given in Appendix B and some important core functions used within the Riemannian manifold samplers are given in Appendix C. The code for the rest of the functions used is available on request.

5.1 Test Distributions

The sampling algorithms were first tested on three common benchmark 2D distributions whose density function was precisely known:

- Correlated Gaussian
- Banana-shaped distribution
- Mixture of two Gaussians

Since these distributions are 2D it was very easy to visualize how the Markov chain moves around the coordinate space following its probability density. This could give an initial feeling about the correctness of the implementation of the algorithms.

5.1.1 Correlated Gaussian

The easiest test distribution on which the algorithms were tested was a strongly correlated bivariate Gaussian distribution $N(\mu, \Sigma)$, with $\mu = [2, 5]$ and $\Sigma = \begin{bmatrix} 1 & 0.95 \\ 0.95 & 1 \end{bmatrix}$. The samplers used were MH with proposal covariance matrix $\Sigma = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.2 \end{bmatrix}$, HMC with $\epsilon = 0.3, L = 7, M = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and MALA with $\epsilon = 0.35$. The sampler parameters were obtained after a fine tuning process, to ensure good sampler performance. The samplers were run for 5000 iterations with starting points within the center of the distributions. No burn-in was considered.

The following figures show trace plots, running averages and autocorrelation plots of the sampled variables by the different samplers.
Figure 5.1: Gaussian distribution: Traces of the sampled variables (in both dimensions) for different samplers. From left to right: MH, HMC, MALA. On the x-axis the number of the sampler iteration is shown, on the y-axis the values of the sampled variables ($x_1$ on the upper plots, $x_2$ on the lower plots).

Figure 5.2: Gaussian distribution: Running averages of the sampled variables (in both dimensions) for different samplers. From left to right: MH, HMC, MALA.

Figure 5.3: Gaussian distribution: Autocorrelation of the sampled variables (in both dimensions) for different samplers. From left to right: MH, HMC, MALA.

The shape of this distribution is simple, therefore all samplers performed well, as expected. Figure 5.1 shows the traces of the variables, i.e. their values at every sampler iteration. The movement of the MH and MALA Markov chains resembles random walk. Every next sample is close to its predecessor and there is no good mixing. The consecutive samples drawn by HMC on the other hand, are not necessarily close to each other and mix well.
This is supported by the autocorrelation plots of the variables (see Figure 5.3). Due to the random walk of MH and MALA, their samples are more correlated than the samples produced by HMC, whose autocorrelation immediately drops to zero.

Even though the mixing and the autocorrelation of the samples is different for different algorithms, all of the sampled distributions converge to the same mean value (see Figure 5.2). The convergence is slower and less stable in the case of MH in MALA than in the case of HMC. MH and MALA need much more iterations to achieve what HMC has achieved with 5000 iterations.

5.1.2 Banana-shaped Distribution

The function for the banana-shaped distribution is the following:

\[
f(x) \propto \exp\left(-\frac{x_1^2}{200} - 0.5(x_2 + Bx_1^2 - 100B)^2\right)
\] (5.1)

with \(B = 0.01\) and \(B = 0.1\). The greater the \(B\), the more curved (the more “banana-like”) the distribution becomes. Below, the results only for the case when \(B = 0.1\) are shown.

This distribution is tricky to sample. It spreads along huge range and the chain needs to cross from one side of the distribution to the other. For the sampling, the covariance matrix for the proposal distribution for MH used was \(\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 8 \end{bmatrix}\), the step size for HMC used was \(\epsilon = 0.6\) with \(L = 10\) and the identity matrix as mass matrix. The step size for MALA used was \(\epsilon = 1.3\). In this case, each sampler was run for 10000 iterations, since the distribution is spread over larger space range.

Figure 5.4 shows the points sampled (in black) by MH, HMC and MALA in 2D (\(x_1\) on the x-axis, \(x_2\) on the y-axis) overlaid on a contour plot of the probability density (blue lines). We expect a good sampler to sample points from the entire arc. MH (left) and MALA (right) perform poorly on the banana - shaped distribution, since by random walk they can not shift easily from one side of the distribution to the other in 10000 iterations. They sample primarily from the central part (where the starting point was) and the tails of the distribution are poorly sampled. HMC (middle) samples from the entire distribution, including the tails as well.

![Figure 5.4](image)

The covariance matrix for MH proposal density was adjusted in such way that the variance of the \(x\) coordinate is much larger than the variance of the \(y\) coordinate. This kind of proposal distribution is good for sampling of the central part of the distribution, but not good for the tails. For the tails, bigger variance in \(y\) dimension than in \(x\) dimension would lead to better performance. In this case, a MH algorithm that automatically adjusts its proposal matrix might be appropriate.
Such class of samplers is called “Adaptive Metropolis-Hastings algorithms”; however, they were not implemented in this work.

With this distribution the advantage of the HMC algorithm over the other two can be more easily seen. Its chain can move easily from one side of the distribution to the other due to the Hamiltonian dynamics and thus mixes better. This is visible from the plots of the traces of the sampled variables (Figure 5.5). HMC exhibits fast mixing on contrary to MH and MALA.

Figure 5.5: Traces of the sampled variables from the banana - shaped distribution. From left to right: MH, HMC, MALA

Again, same as in the case for Gaussian distribution, MH and MALA algorithms exhibit higher correlation than HMC, whose correlation drops immediately to zero (Figure 5.7). The running averages of HMC converge much faster and more stably than those of the other two samplers (Figure 5.6).

Figure 5.6: Running averages of the sampled variables from the banana - shaped distribution. From left to right: MH, HMC, MALA
5.1.3 Mixture of two Gaussians

The third distribution the samplers were tested on was a mixture of bivariate Gaussians ($\mu_1 = [2 \ 5], \mu_2 = [3 \ 6], \Sigma_1 = \begin{bmatrix} 1 & 0.95 \\ 0.95 & 1 \end{bmatrix}, \Sigma_2 = \begin{bmatrix} 1.5 & -1.48 \\ -1.48 & 1.5 \end{bmatrix}$), each contributing 50% to the mixture. Below are shown 2D plots of the sampled points (8000 samples) overlaid on a contour plot of the density function:

The mixture of two Gaussian distributions can be tricky, since there are several turns the chain needs to make (where the distributions cross each other). In 8000 iterations, as can be seen from Figure 5.8 all of the chains have converged to the target distribution, however, similar to the case of the previous distributions, the fastest convergence was noticed with HMC. The plots of the traces, autocorrelation and running averages can be seen below.
Figure 5.9: Traces of the sampled variables from the mixture of Gaussians. From left to right: MH, HMC, MALA

Figure 5.10: Running averages of the sampled variables from the mixture of Gaussians. From left to right: MH, HMC, MALA

Figure 5.11: Autocorrelation of the sampled variables from the mixture of Gaussians. From left to right: MH, HMC, MALA

Remark: The mMALA and RMHMC algorithms use the expected Fisher information matrix $G$ that is defined with the likelihood of the parameters (eq. 3.13). Both algorithms use characteristics that are defined on a Riemann manifold, which is a property of parametrized density functions. Therefore it is difficult for these methods to be applied on non-parameter distributions. Their implementation was tried on the test distributions by using the Hessian of the distribution function instead, which is equivalent to the observed Fisher information matrix. However, the algorithms implemented in this way could not do effective proposals. The logarithm of the banana-shaped distribution and of the mixture of Gaussians is not always convex. This creates problems with the Hessian as it is not always positive definite, which leads to poor sampler performance. In
such cases, the undesirable matrix properties were remedied by adding small increments to its components, until the point it became positive-definite, but this was not always helpful. In the case of the mixture of Gaussians, the region where the two distributions cross, is a saddle point of the log-density. There, the surface of the log-likelihood is flat and this creates problems when calculating the $G$ matrix, which becomes singular in many cases.

5.2 Dynamical ODE Models

In order to further test the samplers and to construct the ODE and sensitivity solver framework, the parameter posteriors of two ODE models (one small and one larger) were sampled with the different MCMC methods.

5.2.1 Small ODE Model

The first dynamical ODE model used was a small linear model with only two states $(x_1, x_2)$ and two parameters $(\theta_1, \theta_2)$. The model was created manually and the eigenvalues of the system matrix were chosen such that the states oscillate. The ODEs for the two states are given below:

$$\frac{dx_1}{dt} = \theta_1 x_1 - 7.191 x_2$$

$$\frac{dx_2}{dt} = -\theta_1 x_1 + \theta_2 x_2$$

The nominal values of the parameters are $\theta_1 = -3.561$ and $\theta_2 = 2.191$.

Experimental data $y$ were generated artificially by adding a Gaussian noise to the simulated outputs of the two state variables (when the nominal parameters were used), with variance of 2.25 for every dimension. The prior used was uniform within a box of the parameter space (range of $[-4 \ -2]$ for $x_1$ and $[1 \ 3]$ for $x_2$).

Since the system is small and linear, it was solved analytically for different parameter values on a regular grid and the exact shape of the posterior parameter distribution was obtained (Figure 5.12). This gave us an insight later whether the samplers were sampling from the right density.
The samplers were run for 10000 iterations (no burn-in period was considered) with the following sampler parameters: \( \Sigma = \begin{bmatrix} 0.001 & 0 \\ 0 & 0.001 \end{bmatrix} \) for MH, \( \epsilon = 0.05, L = 5 \) and identity mass matrix for HMC, \( \epsilon = 0.05 \) for MALA, \( \epsilon = 1 \) for mMALA and \( \epsilon = 0.2, L = 5 \) (three fixed point iterations each) for RMHMC. The starting points for all of the chains were the nominal parameter values.

Figures 5.13 - 5.17 show the results obtained from the sampling of the parameter space by different samplers. In each figure, on the top left there is a histogram of the sampled variables, on the top right trace plots, on the bottom left plots of the running averages and bottom right autocorrelation of the sampled variables. All histograms have very similar shape as the shape of the posterior distribution obtained analytically (for the particular experimental data set). This tells us that the samplers managed to sample from the right density. In all cases, at the beginning of the chain there are fluctuation in the average, which stabilizes after the chain has reached stationarity. This stabilization is stronger for some samplers (HMC, mMALA, RMHMC) and less strong for others (MH, MALA).

In terms of convergence the MH and MALA samplers, as expected, performed slightly worse that the rest three, due to their random walk nature. HMC showed performance similar as in the case of the test distributions. The performance of the Riemann methods was also very good. This is because the Fisher information matrix adapts their movement and tuning according to the local curvature of the parameter space.
Figure 5.13: Results obtained by MH sampler for the small ODE model

Figure 5.14: Results obtained by HMC sampler for the small ODE model
Figure 5.15: Results obtained by MALA sampler for the small ODE model

Figure 5.16: Results obtained by mMALA sampler for the small ODE model
Since the target distribution was known, the divergence from the sampled distributions to the target distribution after 10000 iterations was calculated by using the Kullback-Leibler divergence and Hellinger distance measures at every 100\textsuperscript{th} iteration of the sampler (Figure 5.18(a) and 5.18(b) respectively). This is a way of incorporating the question of “how fast” the algorithms operate in the performance evaluation (for a good algorithm, the KL divergence and the Hellinger distance are expected to quickly and smoothly drop to zero).

According to Figure 5.18, the converging algorithms can be divided into two groups: those with fast and those with slow convergence. The first category includes HMC, mMALA and RHMHC. In the figure the curves for their divergence vs. iterations are lower than the other two curves, which means that they converge faster. At the beginning of the evolution of their Markov chain the divergence is very high since the number of samples is still very low. As the chain grows and starts converging, the divergence starts decreasing. After a certain point, when the chain has converged, the divergence converges as well. The other category includes MH and MALA, whose divergence starts converging later and does not drop that smoothly (their curves experience some uphills).

It has to be mentioned that these results are obtained by running only one chain of each algorithm. Different runs might produce slightly different results; however, the trends are expected to be similar. Moreover, further tuning might also improve the performance of the algorithms.

Even though the algorithms show different convergence properties the KL divergence and Hellinger distance drop very low for all of them (as the chain reaches stationarity), indicating that eventually all the chains have converged to the right density.

Figure 5.17: Results obtained by RHMHC sampler for the small ODE model
Figure 5.18: The divergence of the sampled distributions from the target distribution versus the iteration number

Figure 5.19: CPU time taken for different samplers versus iteration number

On the next Figure 5.19, the CPU time that algorithms need for 10000 iterations vs. number of iterations is plotted. The CPU time that RMHMC takes is much higher than the rest since the sampler needs to calculate 1st and 2nd order sensitivities for a number of leapfrog steps and for a number of fixed point iterations within one leapfrog step. HMC is also slow due to the number of leapfrog steps, yet faster than RMHMC since it calculates only 1st order sensitivities and do not require fixed point iterations. The mMALA algorithm follows, which calculates both 1st and 2nd order sensitivities, but since it does it only once per iteration (no leapfrog steps involved) it is much faster. MALA is slightly faster than mMALA since it calculates only 1st order sensitivities and as expected MH is the fastest of all, since it does not do any sensitivity calculation.

The convergence properties of each of the algorithms were combined further with the computational time it takes for them to run in a quantity defined as CPU/ESS. This ratio tells us the CPU time needed for the sampler to produce one effectively independent sample. The smaller this number, the more efficient the algorithm. Table 5.1 summarizes the CPU/ESS results for both $\theta_1$ and $\theta_2$ (2nd and 3rd column respectively).

Slightly surprisingly, the worst CPU/ESS ratio is obtained by RMHMC (average CPU/ESS of
Table 5.1: CPU/ESS

<table>
<thead>
<tr>
<th>MCMC sampler</th>
<th>$\theta_1$</th>
<th>$\theta_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMHMC</td>
<td>2.0695</td>
<td>2.1437</td>
</tr>
<tr>
<td>MALA</td>
<td>0.6553</td>
<td>0.6860</td>
</tr>
<tr>
<td>MH</td>
<td>0.4181</td>
<td>0.4162</td>
</tr>
<tr>
<td>HMC</td>
<td>0.1527</td>
<td>0.1558</td>
</tr>
<tr>
<td>mMALA</td>
<td>0.0759</td>
<td>0.0774</td>
</tr>
</tbody>
</table>

Even though it draws independent samples even at the beginning of the chain, it takes very long time to do so (revisit Figure 5.19) because of the previously mentioned computational requirements (the 1$^{st}$ and 2$^{nd}$ order sensitivity calculation in a combination with fixed point iteration steps per leapfrog iteration). However, there is still some room for further optimization of its implementation which might decrease the running time of the algorithm. RMHMC is followed by MALA (average CPU/ESS of 0.6707) and MH (0.4172). Better performance was obtained by HMC (0.1543) as expected, since due to the avoidance of the random walk behaviour it produces less correlated samples. The best performing algorithm in this scope is mMALA with average CPU/ESS of 0.0767. Even though it needs to calculate both the 1$^{st}$ and 2$^{nd}$ order sensitivities, it profits extremely by drawing well-mixed, uncorrelated samples.

5.2.2 Medium-sized ODE Model

The final test was performed on a bigger and more complex ODE model, namely a six-state cell signaling model introduced by Becker et al. [1], which describes the nonlinear dynamic behaviour of the erythropoietin (Epo) receptor in response to changes in the ligand Epo concentrations outside the cell [1]. The model has eight parameters, two of which were fixed (as in [6]), so the dimensionality of the parameter space was six. Despite having six states ($x_1 \ldots x_6$), the model has only two observables $y_1 = x_1 + x_6$ and $y_2 = x_3$ [1, 6]. The first one is a sum of two states ($x_1$ and $x_6$) and the second observable is the state $x_3$. Therefore, the likelihood calculation and the Fisher information Matrix calculation were dependent only on these two observables, i.e. only on three states. For the calculation of the Fisher information Matrix, the sensitivities of the observables are needed. The sensitivity of the 1$^{st}$ observable $y_1$ is simply sum of the sensitivities for $x_1$ and $x_6$.

Figure 5.20 represents evolution of the six states concentration over time:

Figure 5.20: Time course of the concentrations of the six states of the system
In order to explore a wider range of the parameter space, the sampling was done in the log scale, as in [6]. The model itself was however in linear scale, so effective conversions from linear to log scale and vice versa were needed continuously during the sampling process. The relationship between the parameters in linear and in log scale is as follows:

\[ \theta = \theta_0 \cdot 10^k \]  

(5.2)

where \( \theta \) is the parameter in linear scale, \( k \) is the parameter in log scale and \( \theta_0 \) is the nominal parameter value in linear scale. The nominal parameter values in the linear scale are:

- \( \theta_{on} = 8.096e - 5 \)
- \( \theta_{ex} = 0.0036 \)
- \( \theta_{e} = 0.0175 \)
- \( \theta_{d} = 0.0665 \)
- \( \theta_{de} = 0.0019 \)
- \( \theta_{le} = 0.0131 \) [6]. Since the sampling is done in log scale, it is necessary for all the partial derivatives with respect to the parameters (sensitivities of the observables) to be calculated with respect to the parameters in log scale \( k \), which according to the chain rule produces the following formulas:

\[ \frac{\partial y}{\partial k} = \frac{\partial y}{\partial \theta} \cdot \frac{\partial \theta}{\partial k} \]  

(5.3)

The sensitivities for parameters in linear scale are obtained from the solver; they just need to be multiplied by \( \frac{\partial \theta}{\partial k} \) which from Equation 5.2 is given by:

\[ \frac{\partial \theta}{\partial k} = 10^k \theta_0 \log 10 \]  

(5.4)

The conversion of the second-order sensitivities of the states with respect to parameters in log scale requires more computation and is as following:

\[ \frac{\partial}{\partial k_2} \frac{\partial y}{\partial k_1} = \frac{\partial}{\partial \theta_2} \frac{\partial y}{\partial \theta_1} \cdot \frac{\partial \theta_2}{\partial k_2} \cdot \frac{\partial \theta_1}{\partial k_1} + A \]  

(5.5)

where

\[ A = \begin{cases} 
\frac{\partial y}{\partial \theta_1} \cdot \frac{\partial \theta_2}{\partial k_1} & \text{if } k_1 = k_2, \\
0 & \text{otherwise}
\end{cases} \]  

(5.6)

and

\[ \frac{\partial}{\partial k_2} \frac{\partial \theta_1}{\partial k_1} = 10^{k_1} \theta_0 (\log 10)^2 \]  

(5.7)

Note the appearance of the first- and second- order sensitivities w.r.t. \( \theta \) in the above expressions which are obtained directly from the solver.

The sampling was done in log scale, i.e. the sampled variables were the exponents \( k \) from Eq. 5.2. The proposed parameters were then converted into linear scale and like that passed to the solver for simulation of the ODE equations. The sensitivities returned by the solver were then again converted to log scale to be used in the proposal mechanism for the next parameter set.

The measurement data for the two observables were generated artificially and followed Gaussian distribution with standard deviation equal to 4% of the amplitude of the observable output signals. The prior in this case was a multivariate Gaussian \( N(-2, 2) \) as in [6].

The sampling was done with all of the five MCMC samplers for 15000 iterations (the first 5000 of which were considered as burn-in period and thus thrown away), with starting values \( k_0 = [0 \ 0 \ 0 \ 0 \ 0 \ 0] \). The CPU times needed for completion of all the iterations were stored.

The figures that follow show some convergence properties of the algorithms. The MH sampler was not able to sample properly unless very small variances for its proposal covariance matrix were used. With a covariance matrix with diagonal entries \( 10^{-4} \cdot [0.5 \ 0.5 \ 0.5 \ 0.5 \ 0.5 \ 0.5] \), the obtained results are shown in Figure 5.21. Figure 5.21(a) shows the traces of the sampled variables, followed by the running averages (Fig.5.21(b)) and autocorrelation plots (Fig.5.21(c)). The algorithm shows similar properties as for the small ODE model. Due to its small elements of the \( \Sigma \) matrix, it doesn’t move much around the parameter space and produces highly correlated samples. Consequently, it can not converge to its mean fast enough.
MALA algorithm doesn’t require much tuning, but it does not perform very well in terms of convergence. With $\epsilon = 0.006$, the results obtained from that sampler are given in Figure 5.23.

The HMC sampler required delicate tuning of the mass matrix $M$ because due to the high-dimensionality and correlations between the parameters, different leapfrog step sizes are required for the different dimensions. Results obtained with the diagonal mass matrix $M = \text{diag}([1 \ 5 \ 0.1 \ 8 \ 4 \ 4])$, $\epsilon = 0.003$ and $L = 4$ are reported in Figure 5.22. There is no systematic way of tuning the values of the mass matrix for high-dimensional systems; thus their tuning was based on a trial-and-error process. The reported values turned out to give the best convergence results among many other runs with different values; however, might not be optimal.

Figure 5.21: Convergence results obtained with MH for the medium-sized ODE model.
Figure 5.22: Convergence results obtained with HMC for the medium-sized ODE model

Figure 5.23: Convergence results obtained with MALA for the medium-sized ODE model
The Riemannian methods have the advantage over HMC since they use mass matrix $G$ which depends on the local curvature and adapts itself as the sampler moves around the parameter space. The only parameter that needs to be tuned in mMALA is the step size $\epsilon$ and yet it performs very well, which makes the algorithm very powerful. Figure 5.24 shows convergence plots obtained by running mMALA with $\epsilon = 0.5$. The mixing of the variables is better than the previous algorithms and also the running averages converge immediately in comparison to the running averages of the previous three samplers. The autocorrelations of the variables drop rapidly close to zero and the fluctuations around it are much smaller than the fluctuations of the previous algorithms.

RMHMC was run with $\epsilon = 0.1$, $L = 4$ with three fixed point iterations per Leapfrog step. It also gives much better convergence results than the first three algorithms (see Fig. 5.25). It doesn’t require tuning of the mass matrix as HMC does, but however, it needs more tuning than mMALA (the parameters that need to be tuned are $\epsilon$, $L$ and number of fixed point iterations), which becomes slow and difficult due to the very high computational cost of the algorithm. Therefore, the parameters for RMHMC used here, might not be the most optimal ones. Even if RMHMC shows best converging properties when it is used with its best parameter values, the time it takes to run for 15000 iterations might be equal to (or even longer) than letting a less powerful algorithm run for longer time until proper convergence (ex. HMC).

![Convergence results obtained with mMALA for the medium-sized ODE model](image)
Table 5.2: Average value of the sampled variables in log scale after 15000 iterations (burn-in period of 5000 iterations)

<table>
<thead>
<tr>
<th>MCMC sampler</th>
<th>$k_1$</th>
<th>$k_2$</th>
<th>$k_3$</th>
<th>$k_4$</th>
<th>$k_5$</th>
<th>$k_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>MH</td>
<td>0.0087</td>
<td>0.0260</td>
<td>-0.0175</td>
<td>0.0018</td>
<td>-0.1125</td>
<td>-0.0226</td>
</tr>
<tr>
<td>MALA</td>
<td>0.0322</td>
<td>0.1334</td>
<td>0.0097</td>
<td>0.0193</td>
<td>0.0470</td>
<td>0.0153</td>
</tr>
<tr>
<td>HMC</td>
<td>0.0214</td>
<td>0.0445</td>
<td>0.0011</td>
<td>0.0029</td>
<td>0.0285</td>
<td>0.0101</td>
</tr>
<tr>
<td>mMALA</td>
<td>0.0144</td>
<td>0.0344</td>
<td>0.0008</td>
<td>0.0038</td>
<td>0.0022</td>
<td>0.0048</td>
</tr>
<tr>
<td>RMHMC</td>
<td>0.0272</td>
<td>0.0351</td>
<td>-0.0023</td>
<td>0.0022</td>
<td>0.0015</td>
<td>0.0047</td>
</tr>
</tbody>
</table>

Table 5.2 summarizes the average values of the sampled variables (in log scale) after 15000 iterations for all of the five samplers. Even though the converged averages are not identical, their difference is not big in log scale, which makes it even smaller when translated to linear scale (Table 5.3). All of the converged averages are close to the nominal values indicating that all the samplers are sampling from the right density.
Table 5.3: Average value of the sampled variables in linear scale after 15000 iterations (burn-in period of 5000 iterations)

<table>
<thead>
<tr>
<th>sampler</th>
<th>$\theta_1$</th>
<th>$\theta_2$</th>
<th>$\theta_3$</th>
<th>$\theta_4$</th>
<th>$\theta_5$</th>
<th>$\theta_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>MH</td>
<td>8.2731e-005</td>
<td>3.7931e-003</td>
<td>1.6767e-002</td>
<td>6.6796e-002</td>
<td>1.4373e-003</td>
<td>1.2399e-002</td>
</tr>
<tr>
<td>MALA</td>
<td>8.7339e-005</td>
<td>4.8577e-003</td>
<td>1.7854e-002</td>
<td>6.9557e-002</td>
<td>2.0749e-003</td>
<td>1.3530e-002</td>
</tr>
<tr>
<td>HMC</td>
<td>8.5186e-005</td>
<td>3.9587e-003</td>
<td>1.7504e-002</td>
<td>6.6968e-002</td>
<td>1.9883e-003</td>
<td>1.3368e-002</td>
</tr>
<tr>
<td>mMALA</td>
<td>8.3822e-005</td>
<td>3.8674e-003</td>
<td>1.7491e-002</td>
<td>6.7114e-002</td>
<td>1.8715e-003</td>
<td>1.3207e-002</td>
</tr>
<tr>
<td>RMHMC</td>
<td>8.6347e-005</td>
<td>3.8738e-003</td>
<td>1.7365e-002</td>
<td>6.6860e-002</td>
<td>1.8686e-003</td>
<td>1.3204e-002</td>
</tr>
<tr>
<td>nominal</td>
<td>8.1069e-005</td>
<td>3.5727e-003</td>
<td>1.7458e-002</td>
<td>6.6527e-002</td>
<td>1.8621e-003</td>
<td>1.3062e-002</td>
</tr>
</tbody>
</table>

Finally, ESS related quantities are reported in Table 5.4 for the five different samplers: average ESS over all the six variables ($2^{nd}$ column), CPU time taken from after the burn-in period until the end of the 15000 iterations ($3^{rd}$ column) and average CPU/ESS ($4^{th}$ column). Big ESS values indicate good convergence properties of the algorithm. Small CPU/ESS values, on the other hand, indicate efficient algorithms (when the running time is considered). A good algorithm both in terms of convergence and speed should have big ESS value corresponding to a small CPU/ESS ratio.

Table 5.4: ESS related quantities for the five different samplers

<table>
<thead>
<tr>
<th>MCMC sampler</th>
<th>average ESS</th>
<th>CPU</th>
<th>average CPU/ESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MH</td>
<td>10.627</td>
<td>45.674</td>
<td>6.006</td>
</tr>
<tr>
<td>mMALA</td>
<td>327.880</td>
<td>2405.100</td>
<td>7.362</td>
</tr>
<tr>
<td>MALA</td>
<td>12.390</td>
<td>197.330</td>
<td>18.059</td>
</tr>
<tr>
<td>HMC</td>
<td>31.350</td>
<td>620.040</td>
<td>52.487</td>
</tr>
<tr>
<td>RMHMC</td>
<td>194.240</td>
<td>14506.000</td>
<td>85.994</td>
</tr>
</tbody>
</table>

The smallest ESS is obtained by MH, as expected. However, due to the very short CPU time it takes to run, it shows the best CPU/ESS value. Slightly bigger ESS is obtained by MALA, but its CPU/ESS value is three times bigger than that of MH, due to its higher computational complexity. As expected, the ESS obtained by HMC is bigger (twice as big as the ESS obtained by MALA), but the CPU/ESS quantity is also very big (3 times bigger than that of MALA) as a consequence of the long CPU time it takes. RMHMC produces much higher ESS (six times higher than that of HMC). However, its CPU time is longer as well (around 1.5 times longer than HMC), which gives rise to a worst CPU/ESS quantity. The highest ESS is obtained by mMALA. Its CPU time is also long (not as long as RMHMC, but almost 4 times longer than that of HMC) and yet, its CPU/ESS ratio is extremely good (just slightly higher than MH).
Chapter 6

Discussion and Conclusions

6.1 Conclusions

In this work five different MCMC sampling methods have been evaluated: Metropolis-Hastings (MH), Hamiltonian Monte Carlo (HMC), Metropolis Adjusted Langevin Algorithm (MALA), manifold Metropolis Adjusted Langevin Algorithm (mMALA) and Riemann manifold Hamiltonian Monte Carlo (RMHMC).

When used to sample from parameter distributions of ODE models, the sampling algorithms require information about the states, 1st and 2nd order state sensitivities at specific points of time. For complex, nonlinear systems it is hard for these quantities to be found analytically and thus, specialized ODE solvers are required. For this purpose an automatic framework was developed in which the different samplers were connected with specialized ODE solvers (CVODE/S) which calculate the states and their sensitivities. The input for the framework can be any ODE model in SMBL format and the output are the sampled variables.

The samplers were tested on three common benchmark 2D distributions whose target density was known. Later they were used to sample the parameter space of a small, linear, oscillatory ODE model (two states, two parameters), whose posterior distribution could be analytically derived. Finally, they were used to sample from the posterior of a medium-sized ODE model (six states, six parameters). For their evaluation, their convergence properties have been inspected (qualitatively: traces of the variables, running averages, scatter plot etc. as well as quantitatively: autocorrelation, effective sample size). Furthermore, the divergence of the sampled distributions from the target distribution was calculated (only in case when the target distribution was known). The running time of the algorithms was also taken into consideration. The results obtained can be summarized below as following:

In terms of convergence MH and MALA perform poorly, especially when high-dimensional problems are considered. They are not very good in producing independent samples, mainly due to their random walk behaviour. These algorithms are however very fast (especially MH), which is a big advantage for simpler distributions. If the distribution landscape is not very complex, running them for long enough might give very good convergence to the target distribution. After proper tuning they might be sometimes even more efficient than the other algorithms in case of simple probability density landscapes. However, when more complex distributions are considered, they might experience convergence difficulties. In addition, MH requires careful tuning of the covariance matrix for the proposal density.

The performance of HMC is mediocre. It performs very well on the test distributions and on simpler parameter distributions. However, if high-dimensional, correlated distributions are considered, HMC needs very precise tuning of the mass matrix, especially if the target distribution has different spread in different dimensions. Moreover, HMC takes more CPU time than MH and MALA, but less than the Riemannian methods (for more complex models).
The Riemann manifold MCMC methods could not be applied on the test distributions since they require a manifold to operate, but performed well in the case of ODE models, in terms of convergence. They both converge fast and produce least correlated samples. They are very good choice when high-dimensional and highly correlated target distributions are considered. RMHMC however, is extremely slow, which decreases its overall efficiency. Furthermore, it also requires proper tuning. Even though it might produce the most uncorrelated samples and the fastest (in terms of number of iterations) converging mean, due to the extremely long running time, it is not advantageous. Last, since its implementation is most complex than the rest, there is a room for further optimization, which might decrease the running time of the algorithm.

On the other hand, mMALA performs extremely well. It converges fast and it takes far less CPU time than RMHMC, which in overall outperforms all the other algorithms.

It is difficult to say any precise order of performance of these algorithms. The boundaries of their performance classifications are soft, since there is always a trade-off between computational time required vs. convergence properties of the algorithms and even divergence from the true distribution (in case it is known). Also, when accessing performance, the tuning time and difficulty should be taken into account. A very good and fast algorithm that works very well with strictly one specific set of sampler parameters should not be preferred if it needs extremely long time to be tuned. Furthermore, different target distributions can give rise to different results, depending on their dimensionality, correlation, modality etc. This should be always considered and the samplers should be tuned accordingly.

However, even when all of these facts are considered, the clear winner among all MCMC samplers is mMALA especially when used to sample from more difficult distributions. It is relatively fast (despite the fact that it needs to compute 1st and 2nd order sensitivities), it produces very well-mixed, uncorrelated samples and the only tuning that it requires is the step size $\epsilon$.

### 6.2 Future Work

Single-chain MCMC methods showed relatively good performance on the tested simpler parameter distributions. However, it is very likely that bigger ODE models have complex, highly correlated probability landscapes, which have more than one disconnected modes. In this case the MCMC methods used in this work will fail because if the modes are not connected, they will always sample from the mode where they started, not being able to foresee the existence of other more distant modes.

Population-based MCMC methods combine existing MCMC methods and global search algorithms. Instead of a single chain, more chains are run in parallel, all having different properties. The individual samplers are adaptive and use information from other samplers in the population in order to adjust their sampling distributions [9]. Variants of these methods are “Population Monte Carlo Algorithms [12], Sequential Monte Carlo Samplers [14] etc.

The next step in this work would be to integrate the best performing MCMC sampler (which is mMALA) within a bigger framework of population-based MCMC methods and use it to sample from more complex target distributions.
Bibliography


Appendix A

Code for MCMC Samplers

A.1 Metropolis-Hastings (MH)

function [output] = MH(input)
    conf = config();
    current_x = input.parameters;
    current_likelihood = input.likelihood;
    %generate a random point from a bivariate gaussian around the current point
    candidate_x = mvnrnd(current_x, conf.proposal_sigma, 1);
    %Calculate the "transition" probabilities
    Q_n_c = mvnpdf(candidate_x, current_x, conf.proposal_sigma);
    Q_c_n = mvnpdf(current_x, candidate_x, conf.proposal_sigma);
    %The current likelihood is passed as an argument
    L_current = current_likelihood;
    [varargout{1}] = simulate_cvodes(candidate_x, 'cvode');
    xout = varargout{1};
    %The likelihood at the candidate position (with the new xout)
    L_candidate = calculate_likelihood_simulations(xout);
    %calculate the acceptance probability
    a = (L_candidate*Q_c_n)/(L_current*Q_n_c);
    %---------accept-reject step--------------%
    %If it is out of bounds - reject
    if candidate_x(1) < conf.bound_x(1) || candidate_x(1) > conf.bound_x(2) || candidate_x(2) < conf.bound_y(1) || candidate_x(2) > conf.bound_y(2)
        next_sample = current_x;
        likelihood = L_current;
        % disp('rejected')
    else % If it within the bounds, do the normal accept-reject step
        if a >= 1
            next_sample = candidate_x;
            likelihood = L_candidate; % if accepted return the likelihood of the accepted sample
        else
            u = rand;
            if u > a
                next_sample = candidate_x;
                likelihood = L_candidate; % if accepted return the likelihood of the accepted sample
            else
                next_sample = current_x;
                likelihood = L_current; % if rejected return the likelihood of the current sample
            end;
        end;
    end;
    output{1} = next_sample(1);
    output{2} = next_sample(2);
    output{3} = likelihood;

A.2 Hamiltonian Monte Carlo (HMC)

function [output] = HMC(input)
    conf = config();
    current_q = input.parameters;
    current_likelihood = input.likelihood;
    current_grad = input.gradE;
    %The current position
    q = current_q;
    %Generate random momentum variables
    p = randn(length(current_q));
    current_p = p;
% Calculate E(q) and K(p)
current_E = -log(current_likelihood);
% E = -log(Px), in our case since Px = likelihood, then E = -log(likelihood) = -logL
current_K = sum(current_p.^2/2*conf.m);
%------------------LEAPFROG ALGORITHM------------------
% Make a half step for the momentum at the beginning
p = p - (conf.epsilon * current_gradE/2);
% Iterate L times
for i=1:conf.L
q = q + conf.epsilon * p/conf.m;
[varargout{1:2}] = simulate_cvodes(q,'cvodes');
out = varargout(1);
sensitivities = varargout(2);
gradE = grad_E_parameters(out,sensitivities); % with the new sensitivities
if i ~= conf.L
p = p - (conf.epsilon * gradE);
end;
end;
% Make a half step for the momentum at the end
p = p - (conf.epsilon * gradE/2);
%----------------------------------------------------
p = -p;
new_likelihood = calculate_likelihood_simulations(out);
proposed_E = -log(new_likelihood);
proposed_K = sum(p.^2/2*conf.m);
% accept - reject
if q(1) < conf.bound_x(1) || q(1) > conf.bound_x(2) || q(2) < conf.bound_y(1) || q(2) > conf.bound_y(2) % If it is out of bounds - reject
new_q = current_q;
likelihood = current_likelihood;
grad = current_gradE;
else
a = min(1, exp(- proposed_E - proposed_K + current_E + current_K ));
u = rand;
if a >= u;
new_q = q; % accept
likelihood = new_likelihood;
grad = gradE;
else
new_q = current_q; % reject
likelihood = current_likelihood;
grad = current_gradE;
end;
end;
output{1} = new_q(1);
output{2} = new_q(2);
output{3} = likelihood;
output{4} = grad;

A.3 Metropolis Adjusted Langevin Algorithm (MALA)

function [output] = MALA(input)
conf = config();
current_x1 = input.parameters(1);
current_x2 = input.parameters(2);
current_x = [current_x1 current_x2];
current_likelihood = input.likelihood;
current_gradL = input.gradL;
current_x1_drift = current_x1 + (conf.epsilon_MALA^2)*current_gradL(1)/2;
current_x2_drift = current_x2 + (conf.epsilon_MALA^2)*current_gradL(2)/2 ;
z = normrnd(0,1);
w = normrnd(0,1);
x1_diffussion = conf.epsilon_MALA*z;
x2_diffussion = conf.epsilon_MALA*w;
% The PROPOSED PARAMETERS are drift + diffusion
candidate_x1 = current_x1_drift + x1_diffussion;
candidate_x2 = current_x2_drift + x2_diffussion;
candidate_x = [candidate_x1 candidate_x2];
L_current = current_likelihood;
[varargout{1:2}] = simulate_cvodes(candidate_x,'cvodes');
xout = varargout{1};
sensitivities = varargout{2};
L_candidate = calculate_likelihood_simulations(xout); % The likelihood at the candidate position
gradL_proposed = - grad_E_parameters(out,sensitivities);
candidate_x1_drift = candidate_x1 + (conf.epsilon_MALA^2)*gradL_proposed(1)/2;
candidate_x2_drift = candidate_x2 + (conf.epsilon_MALA^2)*gradL_proposed(2)/2 ;
% calculate q(theta*|theta) and q(theta|theta*)
c_q = mvnpdf(candidate_x, [current_x1_drift, current_x2_drift], conf.epsilon_MALA^2*['1 0;0 1']);
c_q = mvnpdf(current_x, [current_x1_drift, current_x2_drift], conf.epsilon_MALA^2*['1 0;0 1']);
% calculate the acceptance probability
a = L_candidate*c_q/n/L_current*c_q;
% === accept - reject step ======
% If it is out of bounds - reject
if candidate_x(1) < conf.bound_x(1) || candidate_x(1) > conf.bound_x(2) || candidate_x(2) < conf.bound_y(1) || candidate_x(2) > conf.bound_y(2)
next_sample = current_x;
else
a = min(1, exp(- proposed_E - proposed_K + current_E + current_K ));
u = rand;
if a >= u;
next_sample = candidate_x;
else
next_sample = current_x;
end;
end;
function [output] = mMALA(input)
conf = config();
current_x1 = input.parameters(1);
current_x2 = input.parameters(2);
current_x = [current_x1 current_x2];
current_likelihood = input.likelihood;
current_gradL = input.gradL;
current_invG = input.invG;
current_dGdTheta = input.dGdTheta;

% THE DRIFT ------------------------------
current_drift = calculate_drift_mMALA(current_x,current_dGdTheta,current_invG,current_gradL); %get the mean of the current x i.e. the drift

% THE DIFFUSION -------------------------
RMdiffussion = randn(1,2)*conf.epsilon_mMALA*chol(current_invG);

%THE PROPOSED PARAMETERS are drift + diffusion
candidate_x = current_drift + RMdiffussion;

%Calculate P(theta) and P(theta*)
L_current = current_likelihood;
[varargout{1:2}] = simulate_cvodes(candidate_x,'cvodes_aug');
xout = varargout{1};
sensitivities = varargout{2};
L_candidate = calculate_likelihood_simulations(xout); %The likelihood at the candidate position

%get the gradient and the drift for the proposed parameters, so that
%the drift can be used in q(theta*|theta) and q(theta|theta*)
gradL_proposed = - grad_E_parameters(xout,sensitivities);
new_G = calculate_G(sensitivities);
new_invG = inv(new_G);
new_dGdTheta = calculate_dG_dTheta(sensitivities);

%calculate q(theta*|theta) and q(theta|theta*)
Q_n_c = mvnpdf(candidate_x, current_drift, conf.epsilon_mMALA^2*current_invG*[1 0;0 1]);
Q_c_n = mvnpdf(current_x, candidate_drift, conf.epsilon_mMALA^2*new_invG*[1 0;0 1]);

%calculate the acceptance probability
a = log(L_candidate) + log(Q_c_n) - log(L_current) - log(Q_n_c);

% ==== accept - reject step ========
if a >= 1
next_sample = candidate_x;
likelihood = L_candidate;
gradL = gradL_proposed;
end;
end;
end;
end;
output(1) = next_sample(1);
output(2) = next_sample(2);
output(3) = likelihood;
output(4) = gradL;

A.4 Manifold Metropolis Adjusted Langevin Algorithm (mMALA)

Code for MCMC Samplers - Manifold Metropolis Adjusted Langevin Algorithm (mMALA) 51
% disp('rejected')
end;
end;
end;
output{1} = next_sample(1);
output{2} = next_sample(2);
output{3} = likelihood;
output{4} = gradL;
output{5} = invG;
output{6} = dGdTheta;

A.5 Riemann Manifold Hamiltonian Monte Carlo (RMHMC)

function [output] = RMHMC(input)

cnf = config();
current_q = input.parameters;
current_likelihood = input.likelihood;
current_gradL = input.gradL;
current_G = input.G;
current_invG = input.invG;
current_dGdTheta = input.dGdTheta;

%the current position
q = current_q;
%generate random momentum variables
p = randn(1,2)*chol(current_G);
current_p = p;

%Calculate E(q) and K(p)
current_E = calculate_E_RMHMC(current_G,current_likelihood); % Potential energy (-logLikelihood + normalizing term)
current_K = 1/2*(current_p*current_invG*current_p'); % Kinetic energy

%-------------------LEAPFROG ALGORITHM-------------------
%Iterate L times
for i=1:cnf.L

%Make a half step for the momentum at the beginning
for j = 1:cnf.FPI

    grad_Hq = grad_Hq_RMHMC(tmp_p,current_invG, current_gradL, current_dGdTheta);
    tmp_p = p - (cnf.epsilon_RMHMC/2)*grad_Hq;
end;
p = tmp_p;

%Make a full step for the position

for j = 1:cnf.FPI

    grad_Hp1 = grad_Hp_RMHMC(p,current_invG);
    for j = 1:cnf.FPI
        grad_Hp2 = grad_Hp_RMHMC(p,new_invG);
        tmp_q = q + (cnf.epsilon_RMHMC/2)*(grad_Hp1' + grad_Hp2');
        varargout = {};
        [varargout{1:2}] = simulate_cvodes(tmp_q,'cvodes_aug');
        xout = varargout{1};
        sensitivities = varargout{2};
        new_G = calculate_G(sensitivities);
        new_invG = inv(new_G);
    end;
end;
p = -p; % Negate momentum, but I don't understand why

%Calculate the likelihood of the new parameter values
new_likelihood = calculate_likelihood_simulations(xout);
proposed_E = calculate_E_RMHMC(new_G,new_likelihood);
proposed_K = 1/2*(p*new_invG*p');

%accept_reject
if q(1) < conf.bound_x(1) || q(1) > conf.bound_x(2) || q(2) < conf.bound_y(1) || q(2) > conf.bound_y(2) % If it is out of bounds - reject
new_q = current_q;
likelihood = current_likelihood;
gradL = current_gradL;
G = current_G;
invG = current_invG;
dGdTheta = current_dGdTheta;
else
    a = min(1, exp(- proposed_E - proposed_K + current_E + current_K ));
    if a > rand,
        new_q = q;
        likelihood = current_likelihood;
        gradL = current_gradL;
        G = current_G;
        invG = current_invG;
dGdTheta = current_dGdTheta;
        else
new_q = current_q;
likelihood = current_likelihood;
gradL = current_gradL;
G = current_G;
invG = current_invG;
dGdTheta = current_dGdTheta;
        end;
end;

end;
end;
end;
end;
end;
end;
end;
end;
end;
end;
end;
end;
end;
end;
end;
dGdTheta = current_dGdTheta;
ed;
output(1) = new_q(1);
output(2) = new_q(2);
output(3) = likelihood;
output(4) = gradL;
output(5) = invG;
output(7) = dGdTheta;
Appendix B

Main Function and Configuration File

B.1 Main Function

```matlab
% ------------ DYNAMIC MODEL ------------
% This program samples from the posterior of the dynamical model
conf = config();
% We can choose btw 'MH', 'HMC', 'MALA', 'mMALA' or 'RMHMC'
sampler = 'RMHMC';
[x1, x2, x1p1, x1p2, x2p1, x2p2] = parseFile();
% parses the log file obtained after processing of the SMBL file and gets the
% number of the state that represents the original and the augmented states(sensitivities)
states = [x1, x2, x1p1, x1p2, x2p1, x2p2];

%% Sampling of the parameter space
x0 = [-3.56, 2.191]; % nominal

% Sampling of the parameter space
x0 = [-3.56, 2.191]; % nominal

switch sampler
    case {'RMHMC' 'mMALA'}
        varargout = {};
        [varargout{1:2}] = simulate_cvodes(x0,'cvodes_aug');
        xout = varargout{1};
        sensitivities = varargout{2};
    otherwise
        varargout = {};
        [varargout{1:2}] = simulate_cvodes(x0,'cvodes');
        xout = varargout{1};
        sensitivities = varargout{2};
end;

gradE = grad_E_parameters(xout,sensitivities);
gradL = -gradE;
G = calculate_G(sensitivities);
invG = inv(G);
if (strcmp(sampler,'RMHMC') == 1) || (strcmp(sampler,'mMALA') == 1)
    dGdTheta = calculate_dG_dTheta(sensitivities);
end;
likelihood = zeros(conf.trajectories, 1);
likelihood(1) = calculate_likelihood_simulations(xout);
%store the coordinates here
coord1 = zeros(conf.trajectories,1); coord2 = zeros(conf.trajectories,1);
coord1(1) = x0(1); coord2(1) = x0(2);
current_likelihood = likelihood(1);
if (strcmp(sampler,'RMHMC') == 1) || (strcmp(sampler,'mMALA') == 1)
else
    input = struct('parameters', [coord1(1), coord2(1)], 'likelihood',current_likelihood,'gradE',gradE, 'gradL', gradL,'invG' ,invG, 'states', states);
end;
% --------------------------------------
% THE SAMPLING ALGORITHM
tic
for i=2:conf.trajectories
    %Perform the sampling algorithm
    switch sampler
        case 'MH'
            output = {};
            output = MH(input); % call the MH function
            clear input
            output = struct('parameters', output(1), output(2), 'likelihood',output(3));
        case 'HMC'
            output = {};
            output = HMC(input); % call the HCM function
        case 'MALA'
            output = {};
            output = MALA(input); % call the MALA function
        case 'mMALA'
            output = {};
            output = mMALA(input); % call the mMALA function
        case 'RMHMC'
            output = {};
            output = RMHMC(input); % call the RMHMC function
        otherwise
            output = {};
            output = RMHMC(input); % call the RMHMC function
end;
likelihood(i) = calculate_likelihood_simulations(output);
end;
%store the coordinates here
```
clear input;
input = struct('parameters', [output{1}, output{2}], 'likelihood',output{3},'gradE',output{4});
\n% MALA
case 'MALA'
output = {};
output = MALA(input); % call the MALA function
clear input;
input = struct('parameters', [output{1}, output{2}], 'likelihood',output{3},'gradL',output{4});
\n% mMALA
case 'mMALA'
output = {};
output = mMALA(input); % call the mMALA function
clear input;
input = struct('parameters', [output{1}, output{2}], 'likelihood',output{3},'gradL',output{4},'invG',output{5}, 'dGdTheta', output{6});
\n% RMHMC
case 'RMHMC'
output = {};
output = RMHMC(input); % call the RMHMC function
clear input;
input = struct('parameters', [output{1}, output{2}], 'likelihood',output{3},'gradL',output{4}, 'G', output{5}, 'invG', output{6}, 'dGdTheta', output{7});
end

% stores the returned samples in the coord array
coord1(i) = output{1};
coord2(i) = output{2};
likelihood(i) = output{3};
end;

cpu_time = toc;

B.2 Configuration File

function con = config()
con = struct('sigma', 2.25, 
'x_noise', 
[-16.5056, -13.5027, 2.2540, 10.5065, 5.3543, -5.7311, -4.7520, -1.5777, -0.3616, 0.0509, 1.6314, -0.4456, -0.2286, ...
-2.7465, -1.4342, 3.3877, -0.1246, -0.9216, -0.8340, 2.4409, -0.6899, -0.1136, 2.9566, -1.0579, -0.5693, 2.5196, 0.6777, 1.1191, 0.1774, 1.0547, ...
14.7982, -0.3374, -4.7296, -2.4177, 2.5291, 4.9250, 2.5201, -0.2017, -2.8041, 1.5758, 3.6951, -0.3093, -0.03755, 0.9664, 1.0820, 2.7702, ...
0.1044, 9.9123, 3.9386, 0.8341, 0.0222, -0.1081, -1.4073, -0.2165, 2.1694, -1.3791, 0.7949, 2.0404, 0.0101, -0.4945, 1.7229, 2.8988, -0.2222, ...
'time', [0.3, 0.6, 0.9, 1.2, 1.5, 1.8, 2.1, 2.4, 2.7, 3.0, 3.3, 3.6, 3.9, 4.2, 4.5, 4.8, 5.1, 5.4, 5.7, 6.0, 6.3, 6.6, 6.9, 7.2, 7.5, 7.8, 8.1, 8.4, 8.7, 9, 9.3, 9.6, 9.9], ...
'bound_x', [-4, -2], ...
'bound_y', [1, 3], ...
'trajectories', 10000,...
'epsilon_MALA', 0.07, ...
'epsilon_mMALA', 0.07, ...
'epsilon_RMHMC', 0.2, ...
'FPI', 3, ...
'm', 1, ...
'proposal_sigma', [0.007 0; 0 0.007], ...
'O', 2, ...
'S', 2, ...
'model', 'ToyModel7');
Appendix C

Other Core Functions

C.1 Calculation of the likelihood

```matlab
function LS = calculate_likelihood_simulations(xout)

global x1; global x2; global x1p1; global x1p2; global x2p1; global x2p2;
conf = config();

% Least squares method for calculation of the likelihood
LS = sum( ((conf.x_noise(1,:))' - xout(:,x1)).^2 + ((conf.x_noise(2,:))' - xout(:,x2)).^2 );

% Multiply by a normalization constant to account for the variance used to
% generate the noisy datapoints
ln_LS = (-1/(2*conf.sigma))*LS;

% Add the normalization constant at the end
C = length(conf.x_noise)*log(1/(2*pi*conf.sigma));
ln_LS = C + ln_LS;

% At the end, I have to convert the log likelihood into likelihood (by exponentiating it)
LS = exp(ln_LS);

% artificially added norm
norm = 30.8138;
LS = 1/norm * LS;
```

C.2 Calculation of the gradient of the log-likelihood

```matlab
function gradE = grad_E_parameters(xout,sensitivities)

global x1; global x2;
conf = config();

% E = -log(Px) and since Px is the likelihood in this case, then
% E = -log(likelihood) = -logL
% gradient_p1 = dE/dp1 = d(-logL)/dp1
% gradient_p2 = dE/dp2 = d(-logL)/dp2
% By chain rule (since x and y depend on the parameters p1 and p2):
% gradient_p1 = (d(-logL)/dx1)*dx1/dp1 + (d(-logL)/dx2)*dx2/dp1
% gradient_p2 = (d(-logL)/dx1)*dx1/dp2 + (d(-logL)/dx2)*dx2/dp2
gradient_x1 = -((1/conf.sigma)*((conf.x_noise(x1,:))' - xout(:,x1)));
gradient_x2 = -((1/conf.sigma)*((conf.x_noise(x2,:))' - xout(:,x2)));
gradient_p1 = sum (gradient_x1(:).*squeeze(sensitivities(x1,1,:)) + gradient_x2(:).*squeeze(sensitivities(x2,1,:)));
gradient_p2 = sum (gradient_x1(:).*squeeze(sensitivities(x1,2,:)) + gradient_x2(:).*squeeze(sensitivities(x2,2,:)));
gradE = [gradient_p1 gradient_p2];
```

C.3 Calculation of G

```matlab
function G = calculate_G(sensitivities)

global x1; global x2; global x1p1; global x1p2; global x2p1; global x2p2;
conf = config();

sigma_matrix = eye(length(sensitivities));
sigma_matrix = sigma_matrix*conf.sigma;
invSigma = inv(sigma_matrix);
G = zeros(2);
% Calculate matrix G
for p1 = 1:2
    for p2 = p1:2
        gradient_p1 = squeeze(sensitivities(x1,p1,:))'*invSigma*squeeze(sensitivities(x1,p2,:));
        gradient_p2 = squeeze(sensitivities(x2,p1,:))'*invSigma*squeeze(sensitivities(x2,p2,:));
        G(p1,p2) = gradient_p1 + gradient_p2;
    end;
end;
```
end;
for p1 = 1:2
  for p2 = p1:2
    G(p2,p1) = G(p1,p2);
  end
end;

C.4 Calculation of the derivative of G

function dGdTheta = calculate_dGdTheta(sensitivities)
  global x1; global x2; global x1p1; global x1p2; global x2p1; global x2p2;
  conf = config();
  sigma_matrix = eye(length(sensitivities));
  sigma_matrix = sigma_matrix*conf.sigma;
  invSigma = inv(sigma_matrix);
  second_order1 = zeros([2,2,length(sensitivities)]);
  second_order2 = zeros([2,2,length(sensitivities)]);
  dGTheta1 = zeros([2]);
  dGTheta2 = zeros([2]);
  for i = 1:length(conf.time)
    second_order1(1,1,i) = sensitivities(x1p1,1,i); % second_order1(s,p,i) gives the second order of the s-th state, p-th parameter, with respect to the 1st parameter
    second_order1(1,2,i) = sensitivities(x1p2,1,i);
    second_order1(2,1,i) = sensitivities(x2p1,1,i);
    second_order1(2,2,i) = sensitivities(x2p2,1,i);
    second_order2(1,1,i) = sensitivities(x1p1,2,i);
    second_order2(1,2,i) = sensitivities(x1p2,2,i);
    second_order2(2,1,i) = sensitivities(x2p1,2,i);
    second_order2(2,2,i) = sensitivities(x2p2,2,i);
  end;
  for p1 = 1:2
    for p2 = p1:2
      for s = 1:2
        dGTheta1(p1,p2) = dGTheta1(p1,p2) + squeeze(second_order1(s,p1,:))'*invSigma*squeeze(sensitivities(s,p2,:)) + ...
        squeeze(sensitivities(s,p1,:))'*invSigma*squeeze(second_order1(s,p2,:));
        dGTheta2(p1,p2) = dGTheta2(p1,p2) + squeeze(second_order2(s,p1,:))'*invSigma*squeeze(sensitivities(s,p2,:)) + ...
        squeeze(sensitivities(s,p1,:))'*invSigma*squeeze(second_order2(s,p2,:));
      end;
    end;
  end;
  for p1 = 1:2
    for p2 = p1:2
      for s = 1:2
        dGTheta1(p2,p1) = dGTheta1(p1,p2);
        dGTheta2(p2,p1) = dGTheta2(p1,p2);
      end;
    end;
  end;
  dGTheta = [dGTheta1 dGTheta2];

C.5 Other functions needed for the Riemannian methods

function drift = calculate_drift_mMALA(x,dGTheta,invG,gradL)
  conf = config();
  RMgrad = invG*gradL';
  a = invG*dGTheta(:,1:2)*invG;
  b = invG*dGTheta(:,3:4)*invG;
  drift1 = x(1) + ...
    (conf.epsilon_mMALA^2/2)*RMgrad(1) - ...
    conf.epsilon_mMALA^2*(a(1,1) + b(1,2)) + ...
    (conf.epsilon_mMALA^2/2)*(invG(1,1)*trace(invG*dGTheta(:,1:2)) + invG(1,2)*trace(invG*dGTheta(:,3:4)));
  drift2 = x(2) + ...
    (conf.epsilon_mMALA^2/2)*RMgrad(2) - ...
    conf.epsilon_mMALA^2*(a(2,1) + b(2,2)) + ...
    (conf.epsilon_mMALA^2/2)*(invG(2,1)*trace(invG*dGTheta(:,1:2)) + invG(2,2)*trace(invG*dGTheta(:,3:4)));
  drift = [drift1 drift2];

function E = calculate_E_RMHMC(G,likelihood)
  conf = config();
  logZ = -1/2*log(((2*pi)^conf.D)*det(G));
  E = -log(likelihood) - logZ;

function grad_Hp = grad_Hp_RMHMC(p,invG)
  grad_Hp = invG*p';

function grad_Hq = grad_Hq_RMHMC(p,invG, gradL, dGTheta)
  gradHq_x1 = -gradL(1) + 1/2*trace(invG*dGTheta(:,1:2)) - 1/2*p*invG*dGTheta(:,1:2)*invG*p';
  gradHq_x2 = -gradL(2) + 1/2*trace(invG*dGTheta(:,3:4)) - 1/2*p*invG*dGTheta(:,3:4)*invG*p';
  grad_Hq = [gradHq_x1 gradHq_x2];
C.6 Function that calls CVODES

```matlab
function [varargout] = simulate_cvodes(param, solver)
switch solver
    case 'cvode'
        t0 = 0.0; tspan = [0:0.3:9.9];
        y0 = [10; 10];
        data.p = param;
        options = CVodeSetOptions('UserData', data,'RelTol',1.e-4, 'AbsTol', [1.e-4; 1.e-4], 'JacobianFn', @djacfn);
        CVodeInit(@rhsfn, 'BDF', 'Newton', t0, y0, options);
        [status, tout, xout] = CVode(tspan, 'Normal');
        CVodeFree;
        varargout{1} = xout';
    case 'cvodes'
        t0 = 0.0; tspan = [0:0.3:9.9];
        Ns = 2; dim = 2;
        yS0 = zeros(dim, Ns);
        y0 = [10; 10];
        data.p = param;
        options = CVodeSetOptions('UserData', data,'RelTol',1.e-4, 'AbsTol', [1.e-4; 1.e-4], 'JacobianFn', @djacfn);
        FSAoptions = CVodeSensSetOptions('method','Simultaneous','ErrControl',true,'ParamScales', [param(1); param(2)]);
        CVodeInit(@rhsfn, 'BDF', 'Newton', t0, y0, options);
        CVodeSensInit(Ns, @rhsSfn, yS0, FSAoptions);
        [status, tout, xout, sensitivities] = CVode(tspan, 'Normal');
        CVodeFree;
        varargout{1} = xout';
        varargout{2} = sensitivities;
    case 'cvodes_aug'
        t0 = 0.0; tspan = [0:0.3:9.9];
        Ns = 2; dim = 6;
        yS0 = zeros(dim, Ns);
        y0 = [10; 10; 0; 0; 0; 0];
        data.p = param;
        options = CVodeSetOptions('UserData', data,'RelTol',1.e-4, 'AbsTol', [1.e-4; 1.e-4; 1.e-4; 1.e-4; 1.e-4; 1.e-4], 'JacobianFn', @djacfn_a);
        FSAoptions = CVodeSensSetOptions('method','Simultaneous','ErrControl',true,'ParamScales', [param(1); param(2)]);
        CVodeInit(@rhsfn_a, 'BDF', 'Newton', t0, y0, options);
        CVodeSensInit(Ns, @rhsSfn_a, yS0, FSAoptions);
        [status, tout, xout, sensitivities] = CVode(tspan, 'Normal');
        CVodeFree;
        varargout{1} = xout';
        varargout{2} = sensitivities;
end;
[varargout];
```