Department of Applied Mathematics

Faculty of Mathematical Sciences

University of Khartoum

Bayesian Inference for Dynamic Models using the Metropolis-Hastings Algorithm

A dissertation submitted to the Faculty of Mathematical Sciences, University of Khartoum in partial fulfilment of the requirements for the degree of Msc in Applied Mathematics

By: **Mohanned Mahgoup Khairy**

Supervisor:

Dr. Abdoelnaser Degoot

June, 2022

ACKNOWLEDGEMENTS

I would like to thank Dr Abdoelnaser Degoot, whose expertise was invaluable in conducting this research. Your insightful feedback pushed me to sharpen my thinking and brought this work to a higher level.

Also, I would like to thank my wife and daughter for their continuous support and encouragement.

DEDICATION

To my mother.I love you.

Abstract

Bayesian inference is a type of statistical reasoning that combines probability distributions to create useful and meaningful probability distributions. It is also one of the most useful techniques for parameter inference. In this thesis, we employed a Bayesian inference technique, namely the Metropolis-Hasting algorithm, to estimate the four parameters of the Lotka-Volterra equations. These equations are well-known in ecology and have several other applications. We adopted a well-known Lynx-Hare population problem from the literature, and the key challenge was that the data collected was frequently unreliable or partial, necessitating the use of another approach to estimate the parameters. We obtained an average estimate for each parameter using four chains of the Metropolis-Hasting algorithm and compared it to the original data set, as well as predicted the population dynamic for a few cycles in the future.

Contents

List of Figures

Source Code

List of Tables

Chapter 1

Introduction

1.1 Bayesian Inference

The word model originated from the Latin word modulus, which indicates measure or standard. A conceptual model is an illustration of a system a human-designed representation with a specific goal in mind. A mathematical model is an explanation of phenomena using mathematical languages, Models can be static or dynamic with the sole purpose to explain, study and predict the behavior of the system of interest. In recent years, many dynamical systems studied in the physical, life, social sciences, and engineering are modeled by ordinary, delay, or stochastic differential equations. However, for the vast majority of systems, we lack reliable data about the exact values of the parameters governing these systems. Moreover, the available data are often scarce, noisy, and incomplete, and the likelihood of large models is complex. The analysis of such dynamical systems with such parameters, therefore, requires more representative, quantifiable, and predictive models. Bayesian inference is a special form of statistical inference based on combing probability distributions to obtain meaningful and useful probability distributions. Bayesian inference offers a concise approach for parameter estimations that enables accounting for inherent systematic errors and large statistical biases. Its also associated with predicting results based on prior knowledge and reasoning. Computationally intensive algorithms, such as the Markov chain Monte Carlo (MCMC), is then used to integrate the uncertainty associated with the missing or unobserved data [10].

1.1.1 Bayes Theorem

Thomas Bayes was a reverend in the 18th century, his mathematical knowledge let him think of the problem of God exciting from a probability point of view, and hence he developed the famous formula for updating beliefs to achieve some probability about any unknown problem. Bayes's formula was discovered after his death and the famous Pierre-Simon Laplace refined Bayes' work and gave it the name "Bayes' theorem " [12]. Mathematically, Suppose a number of examples $x_1, x_2, ..., x_n \in X$ are observed, the goal is to find a parametrized density function $f(x|\theta)$ which best fit to the distribution of X. It is equivalent to infer the unknown parameters θ based on the observation. Using Bayes rule, the posterior probability for θ given X is given by:

$$P(\theta|X) = \frac{P(X|\theta)P(\theta)}{P(X)}\,. \tag{1.1}$$

The components of this equation are:

- $P(\theta|X)$ - the posterior, the most probable model parameters can be estimated by maximizing the posterior distribution.
- $P(\theta)$ - the prior, which is what we think in advance the parameters would be. If we have enough data, it turns out the influence of the prior is rather insignificant.
- $P(X|\theta)$ - the likelihood of observing X given model parameters, is given by the model function $f(x|\theta)$ The choice of the model function can be separated from the estimation of model parameters.
- $P(X)$ - the evidence or marginal likelihood, it is a normalizing term that is generally intractable, and it is a constant, independent of θ.

Another way of formulating Bayes' Theorem is

$$\begin{aligned} P(X,\theta) &= P(\theta|X)P(X) \\ &= P(X|\theta)P(\theta)\,, \end{aligned} \tag{1.2}$$

where $P(X,\theta)$ is the joint probability of the data and parameters. This approach to modeling uncertainty is particularly useful when:

- Data limitations.
- The curse of Over-fitting.
- The believe that some facts are more likely than others.
- Interested in precisely knowing how likely certain facts are.

1.1.2 Modelling Steps

These are modeling steps for Bayesian analysis [13]:

- Identify the data relevant to the research questions.
- What are the measurement scales of the data?
- Define a descriptive model for the relevant data.
- The mathematical form and its parameters should be meaningful and appropriate for the theoretical purposes of the analysis.
- Specify a prior distribution on the parameters.
- Use Bayesian inference to redistribute the integrity of parameter values. Interpret the posterior distribution concerning theoretically meaningful issues.
- The posterior predictions represent the data with reasonable accuracy.

In more mathematical terms: suppose that we have a model with an unknown parameter, θ. Most of the time we have some knowledge about θ. If we don't, then we should probably question whether we have an appropriate model or not. Whatever existing knowledge we have about θ is encapsulated in the prior probability distribution, $P(\theta)$. What we are aiming to derive is the posterior probability distribution, $P(\theta|X)$, which reflects our knowledge about θ conditional on some observations X. It should be simple to calculate the probability of the observations for a given model with a specific value of θ. This is known as the likelihood, $P(X|\theta)$. Since we know the observed data, X. If we think of the model as a generative process and we can simulate the data for all possible values of θ, then we are looking for the value of θ where the simulated data look most similar to the observed data [4].

1.1.3 Evidence

The evidence is given by:

$$P(X) = \int P(X|\theta)P(\theta)\,\mathrm{d}\theta\,. \tag{1.3}$$

This integrates the likelihood over all possible parameter values. It's the probability that the data were generated by the chosen model. This is the computationally challenging component of Bayesian analysis, but, there are efficient techniques that allow us to circumvent the evaluation of the evidence, including:

- In cases with conjugate priors, we can get closed-form solutions.
- Numerical integration.
- Approximate the functions used to calculate the posterior with simpler functions and show that the resulting approximate posterior is close to the true posterior (variational Bayes [19]).
- Monte Carlo methods, of which the most important is Markov Chain Monte Carlo (MCMC) and its variants.

1.1.4 Prior

The prior captures the existing knowledge about the distribution of θ before seeing the observations. The prior comes from the domain knowledge which is what we know about the problem before we examine the observations. There's (almost) always some information available that allows you to formulate a reasonable prior [4]. The prior can also help you apply constraints to parameters. For example, if we know that a parameter can only assume positive values then you can enforce this by choosing a prior distribution that only has positive support. The time required for the posterior to converge can depend on a suitable choice for the prior. The better the guess for the shape of the prior, the sooner you'll get good results. Although the prior can be an arbitrary function of θ, it's more common to use a standard distribution. As an example, for the prior, we will be using the truncated normal distribution, which is the probability distribution derived from that

of a normally distributed random variable by bounding the random variable from either below or above (or both) [23]. Suppose X has a normal distribution with mean μ and variance σ^2 and lies within the interval (a, b) with $-\infty \leq a < b \leq \infty$. Then X conditional on $a < X < b$ has a truncated normal distribution with a probability density function:

$$f(x; \mu, \sigma, a, b) = \frac{1}{\sigma} \frac{\phi\left(\frac{x-u}{\sigma}\right)}{\Phi\left(\frac{b-u}{\sigma}\right) - \Phi\left(\frac{a-u}{\sigma}\right)}. \tag{1.4}$$

1.1.5 Conjugate Prior

A closed-form expression for the posterior cannot be achieved without a conjugate prior; otherwise, numerical integration may be necessary which is an algebraic inconvenience, Further, conjugate priors may give intuition, by more transparently showing how a likelihood function updates a prior distribution [24]. If the posterior distribution $p(\theta|x)$ is in the same probability distribution family as the prior probability distribution $p(\theta)$, the prior and posterior are then called conjugate distributions, and the prior is called a conjugate prior for the likelihood function $p(x|\theta)$.

Example 1.1.1. *If we used a beta distribution as a prior over a binomial likelihood function the posterior will also end up being a beta distribution.*

Proof. From Bayes theorem

$$P(\theta|X) = \frac{P(X|\theta)P(\theta)}{P(X)},$$

where $P(X) = \int P(X|\theta)P(\theta)\,\mathrm{d}\theta.$, we can use the basic definitions of the beta and the binomial distributions: $g(\theta) = \frac{1}{B(\alpha,\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1}$, Beta distribution $f(x) = \binom{n}{x}\,\theta^x(1-\theta)^{n-x}$, Binomial distribution by substituting in the general bayes equation:

$$\begin{aligned} P(\theta|X) &= \frac{\binom{n}{x}\theta^x(1-\theta)^{n-x}.\frac{1}{B(\alpha,\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1}}{\int_0^1 \binom{n}{x}\theta^x(1-\theta)^{n-x}.\frac{1}{B(\alpha,\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1}\,\mathrm{d}\theta}, \\ P(\theta|X) &= \frac{\frac{nCx}{B(\alpha,\beta)}\theta^{x+\alpha-1}(1-\theta)^{n-x+\beta-1}}{\frac{nCx}{B(\alpha,\beta)}\int_0^1 \theta^{x+\alpha-1}(1-\theta)^{n-x+\beta-1}\,\mathrm{d}\theta}, \end{aligned} \tag{1.5}$$

and finally:

$$P(\theta|X) = \frac{\theta^{x+\alpha-1}(1-\theta)^{n-x+\beta-1}}{B(x+\alpha-1, n-x+\beta-1)}\,, \tag{1.6}$$

Which indicate that the posterior is also a beta distribution with the parameters $(x+\alpha, n-x+\beta)$ by simply thinking about what the normalizing factor for the numerator must be and it could be written as $P(\theta|x+\alpha, n-x+\beta)$. □

Example 1.1.2. *Let's consider the following two-parameter Bayesian model[25]:*

$$\begin{aligned} y_i &\sim N(\mu, \tau^{-1}) \quad i = 1, ..., n \\ \mu &\sim N(0, 1) \\ \tau &\sim G(2, 1)\,, \end{aligned} \tag{1.7}$$

Where $N(a, b)$ is a normal distribution with mean a and variance b, and $Ga(a, b)$ is a gamma distribution with mean $\frac{a}{b}$ and variance $\frac{a}{b^2}$. Here we assume the y_i are conditionally independent given μ and τ. The joint distribution of y, μ, and τ is:

$$\begin{aligned} P(y, \mu, \tau) &= \prod_{i=1}^{n} P(y_i|\mu, \tau)P(\mu)P(\tau) \\ &= (2\pi)^{-\frac{n+1}{2}}\tau^{\frac{n}{2}}e^{\left\{-\frac{\tau}{2}\Sigma(y_i-\mu)^2\right\}}e^{\left\{-\frac{1}{2}\mu^2\right\}}\tau e^{-\tau}\,, \end{aligned} \tag{1.8}$$

and the joint distribution of μ and τ is:

$$\pi(\mu, \tau) = P(\mu, \tau|y) = \frac{P(y, \mu, \tau)}{\int P(y, \mu, \tau)d\mu d\tau}\,, \tag{1.9}$$

and the full conditional of μ is:

$$\begin{aligned} \pi(\mu, \tau) &= \frac{P(\mu, \tau|y)}{P(\tau, y)} \\ &= \frac{P(y, \mu, \tau)}{P(y, \tau)} \\ &\propto P(y, \mu, \tau)\,. \end{aligned} \tag{1.10}$$

This proportionality follows because $\pi(\mu, \tau)$ is a distribution for μ, Thus the full distribution will be constructed from (1.8) by picking only terms containing μ as follows [25]:

$$\begin{aligned}
\pi(\mu,\tau) &\propto e^{\left\{-\frac{\tau}{2}\Sigma(y_i-\mu)^2\right\}} e^{\left\{-\frac{1}{2}\mu^2\right\}} \\
&\propto e^{\left\{-\frac{\tau}{2}\Sigma(y_i-\mu)^2-\frac{1}{2}\mu^2\right\}} \\
&\propto e^{\left\{-\frac{\tau}{2}(\Sigma {y_i}^2-2\Sigma y_i\mu+n\mu^2)-\frac{1}{2}\mu^2\right\}} \\
&\propto e^{\left\{-\frac{\tau}{2}\Sigma {y_i}^2+\tau\Sigma y_i\mu-n\frac{\tau}{2}\mu^2-\frac{1}{2}\mu^2\right\}} \\
&\propto e^{\left\{-\frac{\tau}{2}\Sigma {y_i}^2+\tau\Sigma y_i\mu-\frac{\mu^2}{2}(n\tau+1)\right\}} \\
&\propto e^{\left\{-\frac{n\tau+1}{2}(\frac{\tau}{n\tau+1}\Sigma {y_i}^2-\frac{\tau}{n\tau+1}\Sigma y_i\mu+\mu^2)\right\}} \\
&\propto e^{\left\{-\frac{n\tau+1}{2}(\mu-\frac{\tau\Sigma y_i}{n\tau+1})^2\right\}} .
\end{aligned} \tag{1.11}$$

Finally, the full conditional for μ *is a normal distribution with mean* $\frac{\tau}{n\tau+1}\Sigma y_i$ *and variance* $\frac{1}{n\tau+1}$*. By the same way, the conditional distribution of* τ *:*

$$\begin{aligned}
\pi(\tau,\mu) &\propto \tau^{\frac{n}{2}} e^{\left\{-\frac{\tau}{2}\Sigma(y_i-\mu)^2\right\}} \tau e^{-\tau} \\
&= \tau^{1+\frac{n}{2}} e^{\left\{-\tau(1+\frac{1}{2}\Sigma(y_i-\mu)^2)\right\}} ,
\end{aligned} \tag{1.12}$$

which is a gamma distribution kernel with index $2+\frac{n}{2}$ *and a scale of* $1+\frac{1}{2}\Sigma(y_i-\mu)^2$*.*

In these two simple examples, the prior distributions were conjugate to the likelihood, so full conditionals reduce analytically to closed-form distributions. Table 1.1 below demonstrates several priors and their conjugates [25]. This is the main intuition behind the wide usage of Beta, Gamma, and the

Prior	**likelihood**	**posterior**
Beta	Bernoulli	Beta
Beta	Binomial	Beta
Beta	Negative Binomial	Beta
Beta	Geometric	Beta
Gamma	Poisson	Gamma
Gamma	Exponential	Gamma
Normal	Normal	Normal

Table 1.1: Some example of conjugate priors and their likelihood.

normal distributions as prior. In summary, it mainly depends on the prior

selection that when we multiply it with the likelihood function we get a posterior distribution that resembles the prior. The below example of estimating the bias of a coin given a sample consisting of n tosses is used to illustrate a few of the approaches.

Example 1.1.3. *In this example, we will be estimating the bias of a coin given a sample consisting of n tosses, If we use a beta distribution as the prior, then the posterior distribution has a closed-form solution. the maximum likelihood estimation (MLE) and the maximum a posteriori (MAP) were shown also [6].*

```
n = 100     # Number of tosses
h = 61      # Number of heads
p = h/n     # Heads Probability
a, b = 10, 10
rv = stats.binom(n, p)  # to create the normal distribution for the
    heads probability
mu = rv.mean()
thetas = np.linspace(0, 1, 200)
prior = stats.beta(a, b)
post = stats.beta(h+a, n-h+b)   # the posterior is a close form on the
    beta distribution
likelihood = n*stats.binom(n, thetas).pmf(h)
ci = post.interval(0.95)
plt.figure(figsize=(9, 5))
plt.plot(thetas, prior.pdf(thetas), label='Prior', c='blue')
plt.plot(thetas, post.pdf(thetas), label='Posterior', c='red')
plt.plot(thetas,likelihood , label='Likelihood', c='green')
plt.axvline((h+a-1)/(n+a+b-2), c='red', linestyle='dashed', alpha=0.4,
    label='MAP')
plt.axvline(mu/n, c='green', linestyle='dashed', alpha=0.4, label='MLE')
plt.xlim([0, 1])
plt.axhline(0.3, ci[0], ci[1], c='black', linewidth=2, label='95% CI');
plt.xlabel(r'θ', fontsize=14)
plt.ylabel('Density', fontsize=16)
plt.legend();
```

Source Code 1.1: A coin tossing problem using Bayesian statistics

the figure below 1.1 shows the results of the above code.

1.2 MCMC Sampling

The term "Markov Chain Monte Carlo" refers to two concepts [25]:

- The Monte Carlo technique uses repeated random samples to estimate a numerical result.

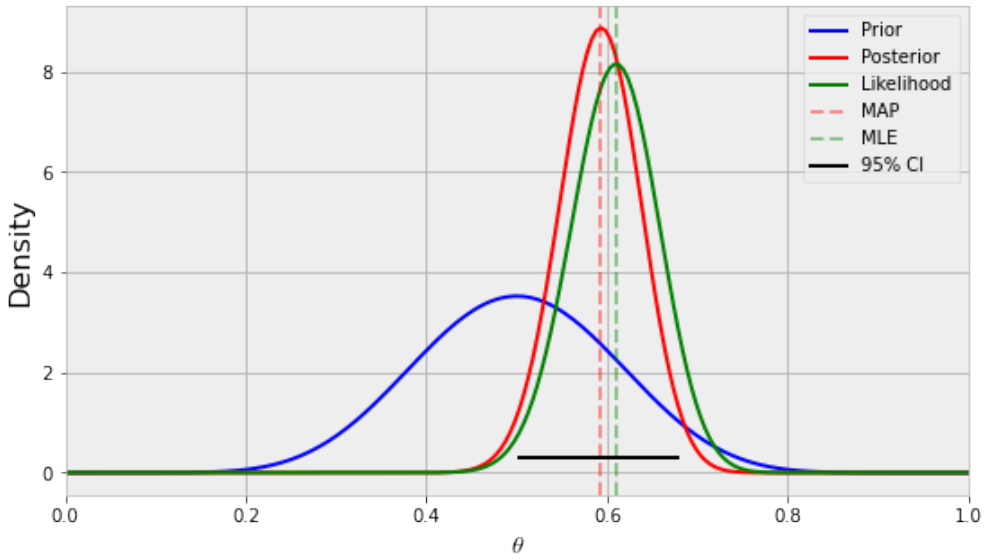


Figure 1.1: Prior, Likelihood, and Posterior estimation

- A Markov Chain which is a process where the next state depends only on the current state

Many algorithms can be used as Universal Inference Engines. Probably the most widely adopted and powerful is the family of Markov chain Monte Carlo methods (MCMC). At a very high level, all MCMC methods approximate the posterior distribution using samples. The samples from the posterior distribution are generated by accepting or rejecting samples from a different distribution called the proposal distribution. The main goal here is to draw samples from the posterior distribution. We could attempt to do this with normal Monte Carlo techniques, but that would require us to generate independent samples from the posterior distribution. To do that we would need to invert the distribution [3], and this is the most challenging part as we stated earlier in section 1.1.

The idea behind MCMC is to perform an intelligent search of the space, this is mainly done by returning samples from the posterior distribution, not the distribution itself. The algorithm will do this search and hopefully will converge towards the areas of high posterior probability after many iterations, the algorithm does this by exploring nearby positions and moving into areas with higher probability, these accepted samples are cumulatively called the traces.

1.2.1 The Central Limit Theorem and the Law of Large Numbers

Let Z_i be N independent samples from some probability distribution. According to the Law of Large numbers, so long as the expected value $E[Z]$ is finite, the following holds,

$$\frac{1}{N}\sum_{i=1}^{N} Z_i \rightarrow E[Z], \quad N \rightarrow \infty\,. \tag{1.13}$$

In words, the average of a sequence of random variables from the same distribution converges to the expected value of that distribution. This may seem like a boring result, but it will be the most useful tool we will use. As a cool intro to the sampling issues, the central limit theorem is intuition from the law of the large number, the Central Limit Theorem states that the sampling distribution of the sample means approaches a Normal Distribution as the sample size gets larger, regardless of the shape of the population distribution [12].

1.2.2 Monte-Carlo Simulation

The first modern Monte Carlo experiments were carried out in the latter decades of the nineteenth century. Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. Their essential idea is to use randomness to solve problems that might be deterministic in principle. They are most useful when it is difficult or impossible to use other approaches. In principle, Monte Carlo methods can be used to solve any problem having a probabilistic interpretation. for example, Let x be a real random variable with a probability distribution functions (PDF), $p(x)$, and a corresponding cumulative distribution function [12].

$$F_x(x) = \int_{-\infty}^{x} p(\tau)d(\tau)\,. \tag{1.14}$$

Monte Carlo integration evaluates $E[f(X)]$ by drawing samples $X_t, t = 1, ..., n$ from π and then approximating So the population mean of $f(X)$ is estimated by a sample mean. When the samples X_t are independent, laws of large numbers ensure that the approximation can be made as accurate as desired by

increasing the sample size n. Note that here n is under the control of the analyst: it is not the size of a fixed data sample. In general, drawing samples X_t independently from π is not feasible, since π can be quite non-standard. However, the X_t need not necessarily be independent.

It is known from probability theory that the random variable, u, defined as $u = F_x(x)$ is uniformly distributed in the interval $0 \leqslant u \leqslant 1$ irrespective of the nature of $p(x)$ If, in addition, we assume that the function F_x has an inverse, F_x^{-1}, then:

$$x = F_x^{-1}(u)\,. \tag{1.15}$$

Thus, following the reverse arguments, samples from p(x) can be generated by first generating samples from the uniform distribution, $U(u|0,1)$, and then applying to them the inverse function, F_x^{-1}. This method works well provided that the function F_x has an inverse that can be easily computed. However, only a few PDFs can have inverses that can be expressed in an analytical form. Besides computational reasons, likely the strongest reason for returning samples is that we can easily use the Law of Large Numbers to solve otherwise intractable problems. With the thousands of samples, we can reconstruct the posterior surface by organizing them in a histogram.

The basis of a Monte Carlo simulation is that the probability of varying outcomes cannot be determined because of random variable interference. Therefore, a Monte Carlo simulation focuses on constantly repeating random samples to achieve certain results.

A Monte Carlo simulation takes the variable that has uncertainty and assigns it a random value. The model is then run and a result is provided. This process is repeated again and again while assigning the variable in question with many different values. Once the simulation is complete, the results are averaged together to provide an estimate [12]. The following code shows an example of using Monte-Carlo simulation for a population problem.

```
p_size = 2.3e5
population = 1/np.logspace(np.log10(0.001), np.log10(10),int(p_size))
fig,ax = plt.subplots(2,1, figsize = (6,6))
ax[0].plot(population[::int(1e3)], 'o',label = 'Population') # to plot
    every 1000 point
ax[0].legend()
#to create randomness, the data will be shuffled
np.random.shuffle(population)
ax[1].plot(population[::int(1e3)], 'o',label ='Shuffled Population') #
    to plot every 1000 point but shuffled
```

```
ax[1].legend()
s_size = 50 # to generate 50 sample randomly for the population
n = 500  # numbers of samples
true_mean = np.mean(population)
# Calculate the mean for each of the 500 samples andcompare to the true
    mean of the population
samplemean = np.zeros(n)
for i in range(n):
    sample = np.random.choice(population,size= s_size) # sample from the
    population with replacement is True
    samplemean[i] = np.mean(sample)
plt.plot(samplemean,'ko',label ='Samples Mean')
plt.plot([0,n],[true_mean,true_mean],'r',label = 'True Mean', linewidth
    = 3)
plt.xlabel('Sample Number')
plt.ylabel('Mean')
plt.legend()
plt.show()
average = np.cumsum(samplemean)/ np.arange(1,n+1)
plt.plot(average,'ko',label ='cumulative Average Mean')
plt.plot([0,n],[true_mean,true_mean],'r',label = 'True Mean')
plt.xlabel('Sample Number')
plt.ylabel('Cumulative Average Mean')
plt.legend()
plt.show()
```

Source Code 1.2: Monte-Carlo Simulation Population Problem.

the result of the above code shows how the samples will look likes randomly and how the average cumulative will be moving towered the mean as we increase the samples number.

1.3 Metropolis-Hastings Algorithm

The Metropolis-Hastings algorithm is an algorithm that produces samples from distributions that may otherwise be difficult to sample from. To sample from a distribution θ, which is known as the target distribution [10].

We assume that θ is a one-dimensional distribution, although it is easy to extend to more than one dimension [17]. The MH algorithm works by simulating a Markov Chain, whose stationary distribution is θ. This means that, in the long run, the samples from the Markov chain look like the samples from θ.

As we will see, the algorithm is incredibly simple and flexible. Its main limitation is that, for difficult problems, "in the long run" may converge

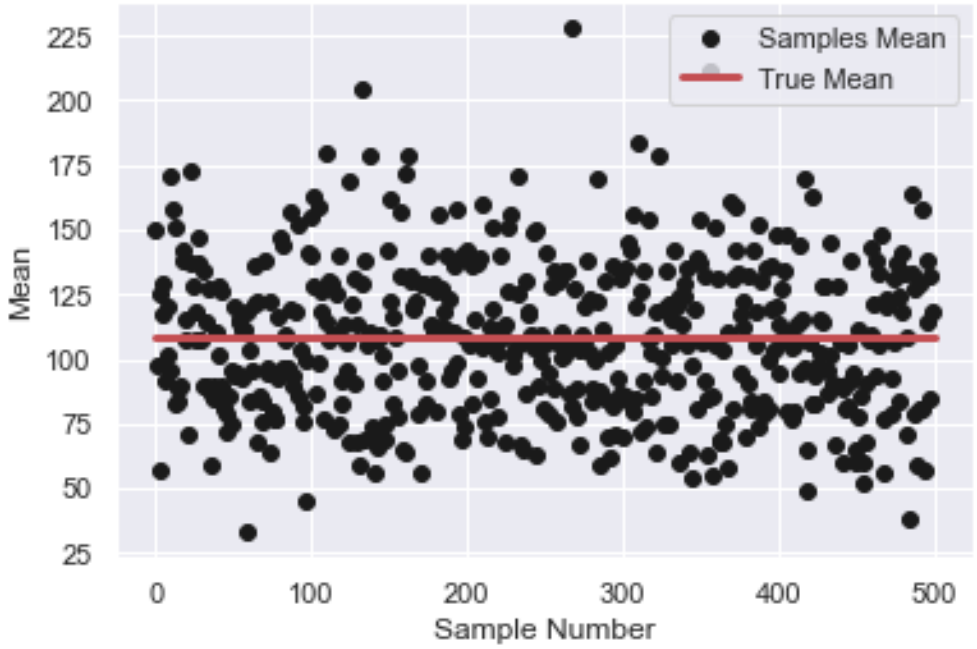


Figure 1.2: Samples mean of a population and their distribution around the true mean.

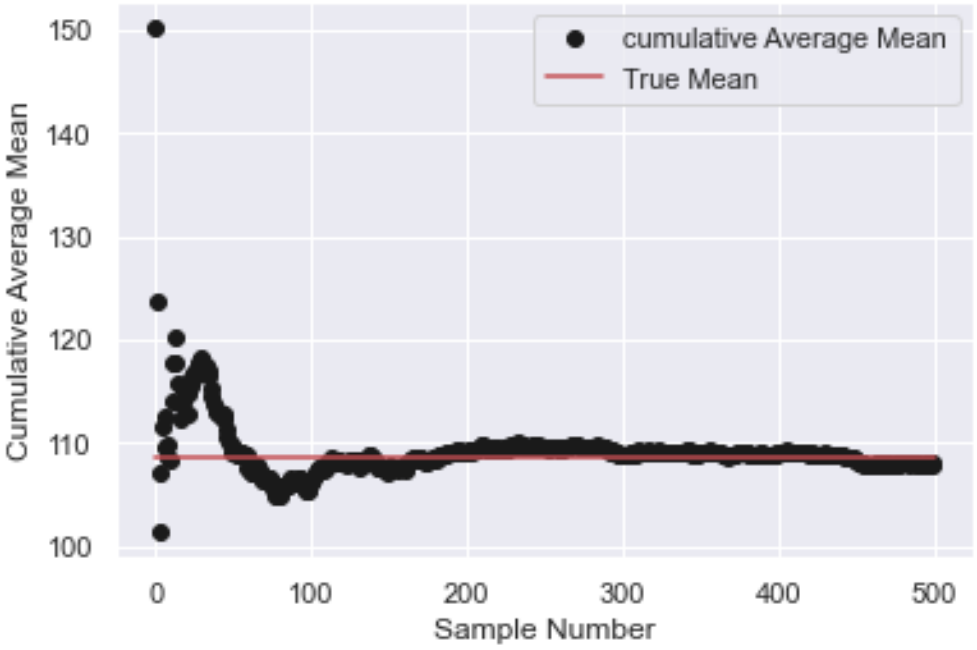


Figure 1.3: Cumulative average of samples mean and its approach toward the true mean.

after a very long time. However, for simple problems, the algorithm can work well [6].

To implement the MH algorithm, we must provide a transition kernel, Q. A

transition kernel is simply a way of moving, randomly, to a new position in space y given a current position x. Q is a distribution on y given x, and it will be written as $Q(y|x)$. In many applications Q will be a continuous distribution, in which case $Q(y|x)$ will be a density on y, and so

$$\int Q(y|x)dy = 1 \qquad \forall x\,. \tag{1.16}$$

For example, a very simple way to generate a new position y from a current position x is to add an $N(0,1)$ random number to x. That is:

$$y = x + N(0,1)\,, \tag{1.17}$$

or

$$(y|x) = N(x,1)\,, \tag{1.18}$$

so

$$Q(y|x) = \frac{1}{\sqrt{2\theta}} e^{[-0.5(y-x)^2]}\,. \tag{1.19}$$

This kind of kernel, which adds some random number to the current position x to obtain y, is often used in practice and is called a random walk kernel. Because of the role Q plays in the MH algorithm, it is also sometimes called the proposal distribution. The example given above would be called a random walk proposal [14].

Burn In period:

the burn-in period is the time it takes for the chain to stabilize so that it's not drifting up or down over time, this will decrease the chaining dependency of the first value and will increase the convergence of the algorithm.

The Algorithm:

We know the current state x_n, and we want to generate x_{n+1}. This is a two steps process [10]. The first step is to generate a candidate x^*, which is generated from a proposal distribution $Q(X^*|x_n)$ which depends on the current state of the Markov chain x_n. We need a distribution that is centred on x_n, so can use something like:

$$x^*|x_n \sim Normal(x_n, \sigma^2)\,, \tag{1.20}$$

Where we include our own σ, the second step is the accept-reject step. We calculate an acceptance probability $A(x_n \to x^*)$ which is given by:

$$A(x_n \to x^*) = min(1, \frac{P(x^*)}{P(x_n)} \frac{Q(x_n|x^*)}{Q(x^*|x_n)}) \,. \tag{1.21}$$

$\frac{P(x^*)}{P(x_n)}$ is easy to compute, as we just plug the value of x^* and x_n in the numerator above. Looking at, $\frac{Q(x_n|x^*)}{Q(x^*|x_n)}$ $Q(x^*|x_n)$ tells us that the probability of generating x^* as the candidate given the current state x_n), which is what is happening now. The numerator tells us exactly the opposite. If the distribution is symmetric, then these two values will be the same and it will just become 1, this is the Metropolis algorithm, and it's a special case of the Metropolis-Hastings algorithm [15].

We now have a candidate x^* and calculated the acceptance probability $A(x_n \to x^*$, we have to either accept or reject this value. If we reject then $x_{n+1} = x_n$. This is easy. We just generate a random number from a uniformly distributed distribution between 0 and 1 which we call u [15]:

$$x_{n+1} = \begin{cases} x^* & \text{if} \quad u \leq A(x_n \to x^*) \\ x_n & \text{if} \quad u > A(x_n \to x^*) \end{cases} . \tag{1.22}$$

We can summarize the above steps in this algorithm [6].

Algorithm 1 An algorithm with caption

- Generate a random starting location x_n.
- Iterate the following for $n = 1, .., N$:
  - Propose a new location from the distribution: $x_{n+1} \sim Q(x_{n+1}|x_n)$.
  - Calculate the ratio:
    $$r = min(1, \frac{P(x^*)}{P(x_n)} \frac{Q(x_n|x^*)}{Q(x^*|x_n)})$$
  - Compare r with a uniformly-distributed number u between 0 and 1.
  - If $r \geq u \implies$ we move.
  - Otherwise, we remain at our current position.

Example 1.3.4. *we used the same data from the previous example to apply the concept of the Metropolis-Hastings algorithm. The code below describes a*

10000 steps iteration to achieve the required distribution and report the results as a chain.

```
def target(lik, prior, n, h, theta):
    if theta < 0 or theta > 1:
        return 0
    else:
        return lik(n, theta).pmf(h)*prior.pdf(theta)
n = 100    # Number of trials
h = 61     # Number of heads
a = 10
b = 10
lik = stats.binom
prior = stats.beta(a, b)
sigma = 0.3
naccept = 0
theta = 0.1
iter = 10000
samples = np.zeros(iter+1)
samples[0] = theta
for i in range(iter):
    theta_p = theta + stats.norm(0, sigma).rvs()
    rho = min(1, target(lik, prior, n, h, theta_p)/target(lik, prior, n,
    h, theta ))
    u = np.random.rand()
    if u < rho:
        naccept += 1
        theta = theta_p
    samples[i+1] = theta
burn_in = len(samples)//2
print ("Efficiency = ", naccept/iter)
```

Source Code 1.3: MH Algorithm Source Code for the Coin Example

This will result in 18 % acceptance rate for MH algorithm, we can visualize the results as follows in figure 1.4 : we can also check how the posterior and prior looks like:

```
post = stats.beta(h+a, n-h+b)
plt.figure(figsize=(12, 9))
plt.hist(samples[burn_in:],40,histtype='step',density=True,linewidth=1,
    label='Distribution of prior samples');
plt.hist(prior.rvs(burn_in), 40, histtype='step',density= True,
    linewidth=1, label='Distribution of posterior samples');
plt.plot(thetas, post.pdf(thetas), c='red', linestyle='--', alpha=0.5,
    label='True posterior')
plt.xlim([0,1]);
plt.legend(loc='best');
```

Source Code 1.4: MH Algorithm Source Code for the Coin Example, the Posterior Distribution

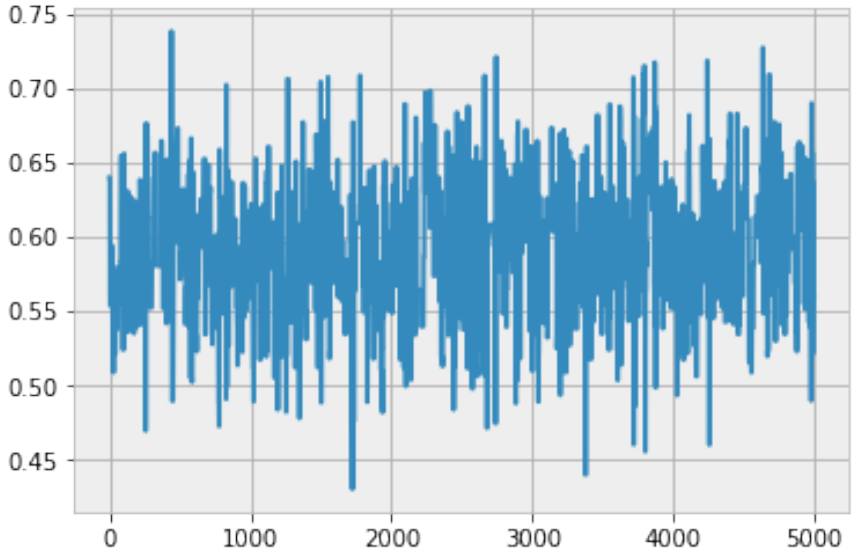


Figure 1.4: Chain results of a MH simulation run without the Burn-in

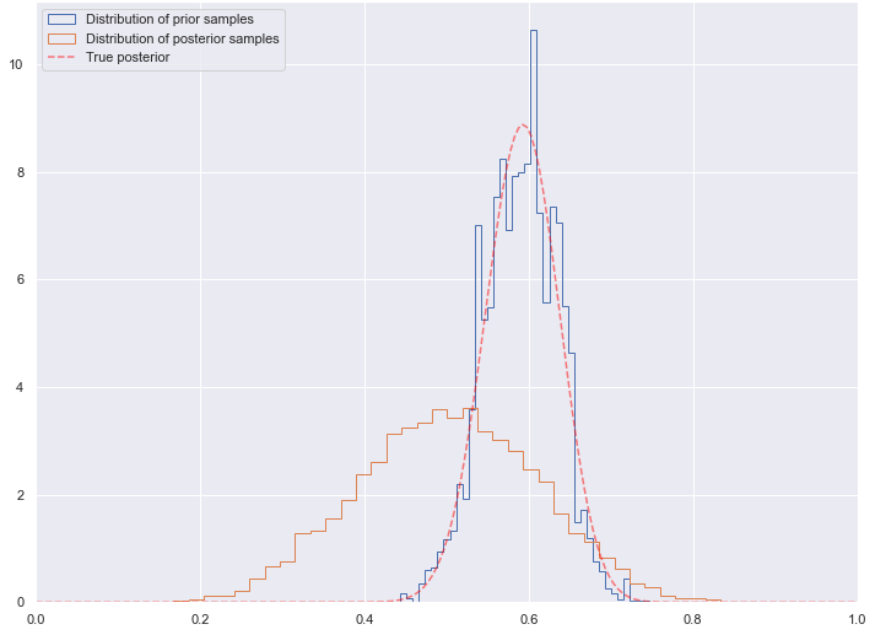


Figure 1.5: Prior and posterior estimation of a coin flip using MH algorithm

Chapter 2

Lotka-Volterra System

2.1 Introduction

Lotka [16] and Volterra [22] formulated parametric differential equations that characterize the oscillating populations of predators and prey. The equations are a pair of first-order non-linear differential equations, frequently used to describe the dynamics of biological systems in which predators and prey interact as seen in figure 2.1.

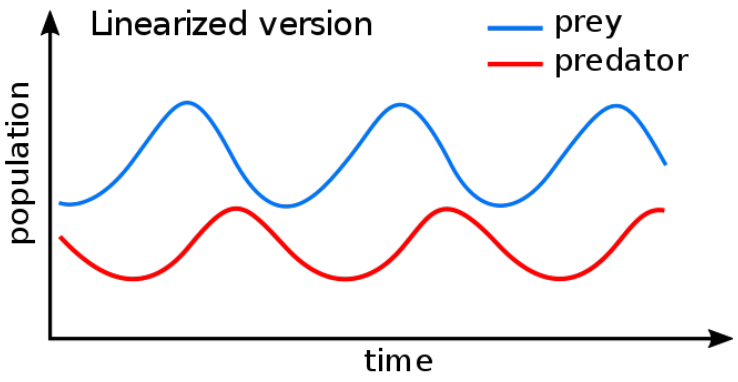


Figure 2.1: Oscillation nature of the Lotka-Volterra system

The equations expressed mathematically as:

$$\begin{aligned} \frac{dx}{dt} &= \alpha x - \beta xy \\ \frac{dy}{dt} &= -\gamma y + \delta xy \,. \end{aligned} \tag{2.1}$$

where:

- $x(t) \geqslant 0$ is the population size of the prey species at time t, and
- $y(t) \geqslant 0$ is the population size of the predator species.
- $\frac{dx}{dt}$ and $\frac{dy}{dt}$ represent the instantaneous growth rates of the two populations.
- $\alpha, \beta, \gamma, \ and \ \delta$ are positive real parameters describing the interaction of the two species.

The parameter γ represents the predator's mortality rate in the absence of prey, whereas α represents the exponential growth rate of the prey without predators [7]. Commonly within the field of theoretical biology, the rate at which two species encounter is proportional to the product of their abundance. Hence the rate at which the predator and prey encounter each other is the product of x and y. The parameter β represents the attack rate of the predators, whereas δ encompasses the rate at which the predator population grows as a result of the abundance of prey. The values of these parameters are not fixed and can differ depending on the modeling of predator-prey interaction. The Lotka–Volterra model makes several assumptions, not necessarily realizable, about the environment and evolution of the predator and prey populations:

- The prey food supply is limitless.
- The predator food supply depends on the size of the prey population.
- The rate of change of population is proportional to its size.
- During the process, the environment does not change in favor of one species, and genetic adaptation is not significant.
- Predators have a limitless desire for their prey.

The prey growth rate population is proportional to the size of its population, leading to exponential growth if not controlled. The prey population simultaneously shrinks at a rate proportional to the size of the product of the prey and predator populations. For the predator species, the direction of growth is reversed. The predator population shrinks at a rate proportional to its size and grows at a rate proportional to the product of its size and

the prey population's size. Together, these dynamics lead to a cycle of rising and falling populations. deterministic and continuous nature of the differential equations solution implies that the generations of both the species are continually coinciding [24].

Although a population following the Lotka-Volterra dynamics will never become extinct before the population became extinct, it may become arbitrarily small. In reality, random events such as accidents can cause extinction events as can not so random events such as extensive hunting. Therefore, the Lotka-Volterra model should not be expected to adequately model the dynamics of small populations.

2.2 Applications of the Lotka-Volterra Equations

Nonetheless, these equations are not restricted to predator-prey type dynamics. The Lotka–Volterra model has found applications in several fields of science such as biology, chemistry, physics, and neuroscience. In each case, the equations were modified to adopt the studied phenomena mathematically and conceptually [18].

The Lotka–Volterra equations were applied to laser physics to describe population inversion and the number of emitted photons [1]. Given that predation and stimulated emissions are analogous processes, two rate equations were obtained by finding suitable parameter transformations for a three-level laser. This resulted in a set of differential equations that are isomorphic to several laser models under accurate parameter identification.

A sales forecasting model that can analyze the interaction effects of two retail competing formats (convenience-oriented vs. budget-oriented formats) [11].

A traditional approach to making such a forecast is based on the Lotka–Volterra equations (also called the Lotka–Volterra model). The Lotka–Volterra model assumes that the population of each species is affected by its self-growth, internal interaction within the species, and external interaction with other species. Most prior studies in business applications directly use sales data as input to the Lotka–Volterra. The prior approach may result in misleading conclusions when sales data are embedded with seasonal variation

because this variation is not addressed in the original development of the Lotka–Volterra model.

The Lotka–Volterra equations have a long history of use in economic theory; their initial application is commonly credited to Richard Goodwin in 1965 or 1967 [24]. In economics, links are between many if not all industries; a proposed way to model the dynamics of various industries has been by introducing trophic functions between various sectors and ignoring smaller sectors by considering the interactions of only two industrial sectors.

2.3 Solution of the System

The equations have periodic solutions, these solutions do not have a simple expression in terms of the usual trigonometric functions, although they are quite tractable. In the model system, the predators thrive when there is plentiful prey but, eventually, consume their food supply and decline. As the predator population is low the prey population will increase again. These dynamics continue in a cycle of growth and decline. Population equilibrium occurs in the model when neither of the population levels is changing, i.e. when both of the derivatives are equal to 0.

$$\begin{aligned} 0 &= \alpha x - \beta xy \\ 0 &= -\gamma y + \delta xy \,, \end{aligned} \tag{2.2}$$

when solved for x and y the above system of equations yields

$$\{y = 0, x = 0\} \,, \tag{2.3}$$

and

$$\left\{ y = \frac{\alpha}{\beta}, x = \frac{\gamma}{\delta} \right\} . \tag{2.4}$$

Hence, there are two equilibria. The first solution effectively represents the extinction of both species. If both populations are at 0, then they will continue to be so indefinitely. The second solution represents a fixed point at which both populations sustain their current, non-zero numbers, and, in the simplified model, do so indefinitely. The levels of the population at which this equilibrium is achieved depending on the chosen values of the parameters, $\alpha, \beta, \gamma, and\ \delta$.

2.4 Example Problem: Population Dynamics for Baboons and Cheetahs

Example 2.4.1. *Suppose there are two species of animals, a baboon (prey) and a cheetah (predator) in figure 2.2 [24]. If the initial conditions are 10 baboons and 10 cheetahs, one can plot the progression of the two species over time; given the parameters that the growth and death rates of baboons are 1.1 and 0.4 while that of Cheetahs are 0.4 and 0.1, respectively. The choice of the time interval is arbitrary. The below code will illustrate how the problem*

(a) Cheetahs

(b) Baboons

Figure 2.2: Cheetahs (predator) and Baboons (prey)

will be solved analytically using the python ODEINT library.

```
from scipy.integrate import odeint
def LotkaVolterra(y,t,p):
    x,y = y
    dxdt = alpha*x-beta*x*y
    dydt = -gamma*y +delta *x*y
    return [dxdt,dydt]
parameters = [1.1,.4,.4,.1,10,10]
alpha,beta,gamma,delta,Xt0,Yt0 = [z for z in parameters]
t = np.linspace(0,100,1000)
values = odeint(LotkaVolterra,[Xt0,Yt0],t,(parameters,))
plt.plot(values[:,0],label = 'baboon')
plt.plot(values[:,1],'.', label = 'cheetah')
plt.xlabel("Time")
```

```
plt.ylabel("Counts")
plt.title("baboon Vs cheetah Change over time");
plt.plot(values[:,0],values[:,1],label = 'baboon, cheetah Cycle')
plt.xlabel("baboon")
plt.ylabel("cheetah")
plt.plot(values[:,0],values[:,1],label = 'baboon, cheetah Cycle')
plt.xlabel("baboon")
plt.ylabel("cheetah")
plt.legend()
for i in range(1,10):
    values = odeint(LotkaVolterra,[i,Yt0],t,(parameters,))
    plt.plot(values[:,0],values[:,1],label=r"$_1$="+"{0:.2f}".format(i))
    plt.xlabel("baboon")
    plt.ylabel("cheetah")
    plt.title('baboon, cheetah Cycle with diff initial baboons')
    plt.legend()
```

Source Code 2.1: Solving the Lotka-Volterra using ODEINT.

The first output is the change in the population counts with time,

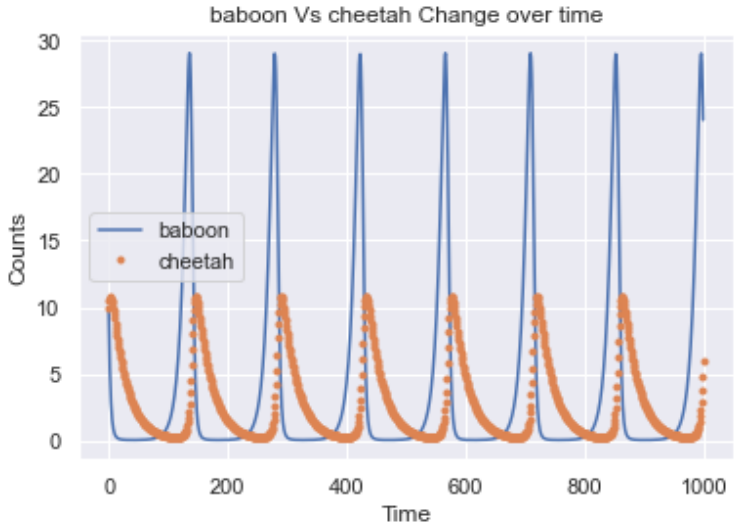


Figure 2.3: Baboons and Cheetahs Count Change with Time as it predicted by the Solution

And also the population dynamic where the graph show how the population change simultaneously:

These graphs illustrate a serious problem with this biological model: in each cycle, the baboon the population is reduced to extremely low numbers yet recovers (while the cheetah population remains sizeable at the lowest baboon density). With chance fluctuations, discrete numbers of individuals, and the family structure and the life cycle of baboons, the Baboons actually could go extinct, and consequence the cheetahs as well.

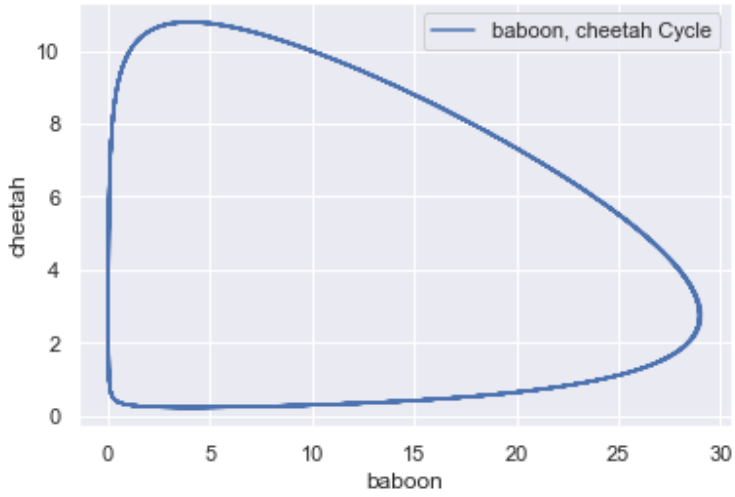
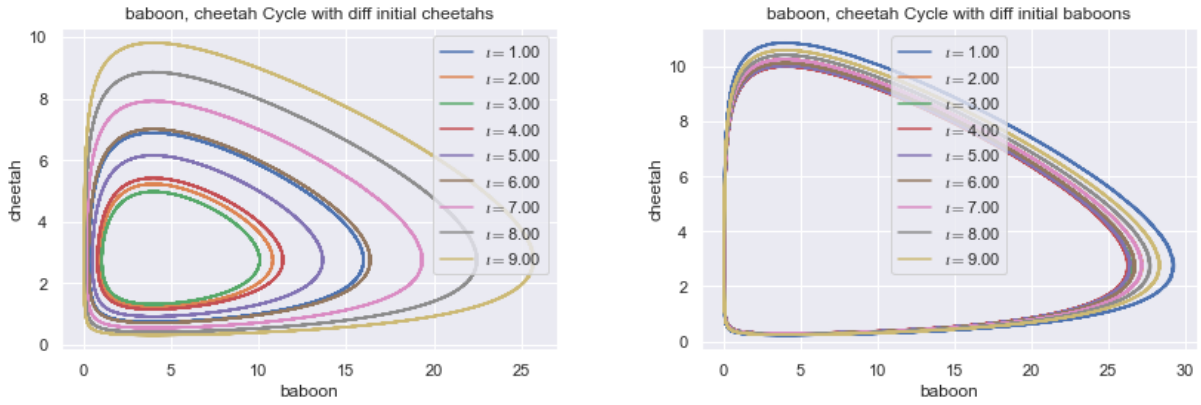


Figure 2.4: Simultaneous Change of Baboons and Cheetahs



(a) Cheetahs change

(b) Baboons change

Figure 2.5: Phase-space plot for the Baboons and Cheetahs with different initials populations count for Cheetahs left, and Baboons right

2.5 Bayesian Inference on Lotka-Volterra System parameters

The parametric inference is a special case of the statistical inference where it is assumed that the functional form of the joint distribution of the random vector is fixed up to the value of the parameter vector living in some parameter space. The distribution of the data is written as the conditional distribution of the data given the parameter, this is because in Bayesian inference the parameter is considered a random variable. This means that

inference about the distribution of the data is reduced to finding out the distribution of the unknown parameter. This simplifies the inference process significantly because we can limit ourselves to the vector spaces instead of the function spaces [20].

There are several studies in recent years focusing on solving the Lotka-Volterra system by using new approaches, A Bayesian approach was developed by [9] to estimate the parameters of ordinary differential equations (ODE) from the observed noisy data. Their method does not need to solve ODE directly and they replaced the ODE constraint with a probability expression and combine it with the non-parametric data fitting procedure into a joint likelihood framework. the main advantage of the proposed method is that for some ODE systems, in particular, they used the Lotka-Volterra system as an example, it can obtain closed-form conditional posterior distributions for all variables which substantially reduces the computational cost and facilitate the convergence process. They used a hybrid Monte Carlo scheme to generate samples for variables whose conditional posterior distribution cannot be written in terms of closed-form. The proposed method was demonstrated through applications to both simulated and real data.

In some cases [19], Approximate Bayesian computation (ABC) methods were used to evaluate posterior distributions without having to calculate likelihoods. In their paper, they discuss and applied an ABC method based on sequential Monte Carlo (SMC) to estimate parameters of Lotka-Volterra system. They found that ABC SMC provides information about the infer-ability of parameters and model sensitivity to changes in parameters, and tends to perform better than other ABC approaches.

The work of [2], introduced gradient matching to the parameter inference problem, instead of quantifying how well the solutions of the ODEs match the data, they quantified how well the derivatives predicted by the ODEs match the derivatives obtained from an interpolate to the data. These methods aim to more efficiently infer the parameters of the equations, but inherent in these procedures is an introduction of bias to the learning problem as we no longer sample based on the exact likelihood function.

In this thesis we will use the traditional MCMC using the Metropolis-Hastings algorithm on the Lotka-Volterra system to infer the parameters of the equa-

tions, we will derive the likelihood function first from simulated data and then from actual data to validate the methodology.

Chapter 3

Parameters Estimation of the Lotka-Volterra System

3.1 Introduction

The Lotka-Volterra model will predict population dynamics into the future and the past for a given accurate data. But given noisy data about population dynamics, how do we solve the inverse problem, that of inferring the values of model parameters consistent with the data?. The general approach in Bayesian statistics is somewhat counter-intuitive, as it involves formulating the forward model and then using general principles to solve the inverse problem [5].

Several functions have been developed to address this issue as follows:

- Create a forward solution to the problem, this will generate a data frame containing simulated data to be used for the likelihood function.
- A likelihood function using log-normal distribution.
- A prior function with truncated normal distribution and the target function to generate the target distribution for the Metropolis-Hastings algorithm.
- The main simulation, this function will wrap up and collect the data from above and generate chains using the Metropolis-Hastings engine.

- The final step is to analyze the results and collect the average parameters for the final inference.

The described steps above will be used to study a specific problem [5].

3.2 Solving the forward problem

As we discussed in 2.4.1 the forward solution will generate the population data as predicted by the numerical solver and store it in a data frame. In Bayesian analysis it is often useful to generate simulated data using samples from the posterior distribution, and compare to the original data.

```
def LotkaVolterra(y,t, alpha, beta,gamma,delta):
    x,y = y
    dxdt = alpha*x-beta*x*y
    dydt = -gamma*y +delta *x*y
    return [dxdt,dydt]
# let us define a function that generate data for the forward version
   problem given set of parameter values
def forwardProblem(time_interval =25, time_points =  25, parms =
   [0.55,0.028,0.84,0.026, 33, 6]):
  alpha,beta,gamma,delta,Xt_0,Yt_0 = [z for z in parms]
  t = np.linspace(0,time_interval,time_points)
  forward_sol = odeint(LotkaVolterra,[Xt_0,Yt_0],t,args=(alpha,beta,
   gamma,delta))
  df = pd.DataFrame({'time': t, 'X': forward_sol[:, 0], 'Y': forward_sol
   [:, 1]})
  # for now we use fixed values for the hyperparameters
  sigma_X= np.std(df.X)
  sigma_Y= np.std(df.Y)
  return df, sigma_X, sigma_Y
# Let us generate some data using some fixed parameter values
df, sigma_X, sigma_Y = forwardProblem()
print("Variance of X=", sigma_X, " and  varaince of Y=", sigma_Y)
```

Source Code 3.1: Forward Solution of the Lotka-Volterra System.

3.3 Model Setup

3.3.1 The likelihood Function

Like in a simple linear regression, we will proceed by treating the underlying deterministic model as providing an expected population value around which

there will be variation due to both measurement error and simplifications in the scientific model. the log-likelihood function will be in the form :

$$L(\mu) = \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} exp\left(-\frac{(x_i-\mu)^2}{2\sigma^2}\right)$$
$$log[L(\mu)] = -nlog(\sigma\sqrt{2\pi}) - \Sigma_{i=1}^{n}\frac{(x_i-\mu)^2}{2\sigma^2}. \tag{3.1}$$

For numerical stability we used the log-transform to avoid negative values we added a minus sign, and we normalized the function by the length of the data set. Also, we focused only on the predator's data y of the model to account for the possibility of missing or incomplete data. This is described in the code below.

```
# compute the average negative likelihood of the data given parameters vector
def computeLikelihood(data, parms):
    alpha,beta,gamma,delta,Xt_0,Yt_0, sigma_X, sigma_Y  = parms
    sol, _,_  = forwardProblem(time_interval= len(data), time_points= len(data), parms = [ alpha,beta,gamma,delta, Xt_0,Yt_0])
    lik_x = np.sum([stats.norm.logpdf(data.X[i], sol.X[i]) for i in range(len(data))])
    lik_y = np.sum([stats.norm.logpdf(data.Y[i], sol.Y[i]) for i in range(len(data))])
    lik = lik_y
    return -lik/len(data)
```

Source Code 3.2: Log likelihood Function

3.3.2 The Prior

A truncated normal distribution was used for the prior with parameters that describe our knowledge about the data. The distribution was used directly for the python SCIPY STATS [21] library with parameters describing the initial set of the model, the target function was adjusted to account for the minus log likelihood and the log prior as well as to be separated for each individual parameter, the function are shown in the code below.

```
alpha_prior = stats.truncnorm(0.0, 1.0, 0.55, 0.1)
beta_prior =  stats.truncnorm(0.0, 0.1, 0.028, 0.01)
gamma_prior = stats.truncnorm(0.5, 1.5, 0.8, 0.2)
delta_prior = stats.truncnorm(0.0, 0.1, 0.026, 0.01)

def alphaTarget(data, parms):
  a = parms[0]
```

```
  if a < 0 or a > 1:
    posterior = 0
  else:
    posterior = computeLikelihood(data, parms) - alpha_prior.logpdf(a)
  return posterior
def betaTarget(data, parms):
  a = parms[1]
  if a < 0 or a > 1:
    posterior = 0
  else:
    posterior = computeLikelihood(data, parms) - beta_prior.logpdf(a)
  return posterior
def deltaTarget(data, parms):
  a = parms[2]
  if a < 0 or a > 1:
    posterior = 0
  else:
    posterior = computeLikelihood(data, parms) - delta_prior.logpdf(a)
  return posterior
def gammaTarget(data, parms):
  a = parms[3]
  if a < 0 or a > 1:
    posterior = 0
  else:
    posterior = computeLikelihood(data, parms) - gamma_prior.logpdf(a)
  return posterior
```

Source Code 3.3: Prior and Target Functions.

3.4 Running the Simulation

This is the main function of this study, the implementation of the Metropolis-Hastings algorithm by utilizing the previously mentioned prior and likelihood functions. The prior data will be generated randomly from the prior function and noise in the form of random values with a certain time step will simulate the random walk effect of the algorithm.

The output here is the accepted values out of the chain for each variable and also the acceptance rate of the algorithm. The average results of the chain are the estimated result of the parameters, several chains will also be simulated together to fully understand the results and the convergence of the algorithm.

```
# Now put all together
def runSimulation(data, niters =100, step_size = 0.01):
  burn_in = niters//2
  sigma_x = 0.25 # varaince of the predator's data
```

```
sigma_y = 0.25 # varaince of the pray's data
alpha = alpha_prior.rvs()
beta = beta_prior.rvs()
gamma = gamma_prior.rvs()
delta = delta_prior.rvs()
# initail  random value for alpha parameter, and likewise for the
 remaining parameters
init_parms = [alpha, beta,gamma,delta]
hyper_parms = [ 33, 6, sigma_x, sigma_y]
chain = np.zeros((niters+1, 4)) # create a container to keep the
 samples
chain[0,:] = init_parms
acceptance_vector = len(init_parms)*[0]
parms = copy.deepcopy(chain[0,:])
for i in range(niters):
  for j in range(4): #
    current_parms = copy.deepcopy(chain[i,:])
    if j == 0:
      #for alpha
      alpha_new = current_parms[j] + step_size*stats.norm.rvs()
      new_parms = copy.deepcopy(current_parms)
      new_parms[j] = alpha_new
      parms = list(current_parms) + list(hyper_parms)
      parms2 =  list(new_parms) + list(hyper_parms)
      p1 = alphaTarget(data,  parms)
      p2 = alphaTarget(data,  parms2)
      if p1 != 0 and p2 != 0:
        rho = min(1, p2/p1)
        if np.random.uniform() < rho:
          acceptance_vector[j] += 1
          chain[i+1,j] = alpha_new
        else:
          chain[i+1,j] = current_parms[j]
      else:
        chain[i+1,j] = current_parms[j]
    else:
      chain[i+1,j] = init_parms[j]
      ##########################################
      if j == 1:
        #for betta
        beta_new = current_parms[j] + step_size*stats.norm.rvs()
        new_parms = copy.deepcopy(current_parms)
        new_parms[j] = beta_new
        parms = list(current_parms) + list(hyper_parms)
        parms2 =  list(new_parms) + list(hyper_parms)
        p1 = betaTarget(data,  parms)
        p2 = betaTarget(data,  parms2)
        if p1 != 0 and p2 != 0:
          rho = min(1, p2/p1)
          if np.random.uniform() < rho:
            acceptance_vector[j] += 1
            chain[i+1,j] =  beta_new
          else:
            chain[i+1,j] = current_parms[j]
```

```
          else:
            chain[i+1,j] = current_parms[j]
        else:
          chain[i+1,j] = init_parms[j]
        ##########################################
          if j == 2:
            #for gamma
            gamma_new = current_parms[j] + step_size*stats.norm.rvs()
            new_parms = copy.deepcopy(current_parms)
            new_parms[j] = gamma_new
            parms = list(current_parms) + list(hyper_parms)
            parms2 =  list(new_parms) + list(hyper_parms)
            p1 = gammaTarget(data,  parms)
            p2 = gammaTarget(data,  parms2)
            if p1 != 0 and p2 != 0:
              rho = min(1, p2/p1)
              if np.random.uniform() < rho:
                acceptance_vector[j] += 1
                chain[i+1,j] = gamma_new
              else:
                chain[i+1,j] = current_parms[j]
            else:
                chain[i+1,j] = current_parms[j]
          else:
            chain[i+1,j] = init_parms[j]
          ##########################################
            if j == 3:
                #for delta
                delta_new=current_parms[j]+step_size*stats.norm.rvs()
                new_parms=copy.deepcopy(current_parms)
                new_parms[j] =  delta_new
                parms = list(current_parms) + list(hyper_parms)
                parms2 =list(new_parms) + list(hyper_parms)
                p1 = deltaTarget(data,  parms)
                p2 = deltaTarget(data,  parms2)
                if p1 != 0 and p2 != 0:
                  rho = min(1, p2/p1)
                  if np.random.uniform() < rho:
                    acceptance_vector[j] += 1
                    chain[i+1,j] =delta_new
                  else:
                    chain[i+1,j] = current_parms[j]
                else:
                  chain[i+1,j] = current_parms[j]
            else:
              chain[i+1,j] = init_parms[j]
clean_chain = chain[-0:burn_in, :]
print ("Efficiency = ", np.array(acceptance_vector)/niters)
chain_df = pd.DataFrame( {'iterations': np.arange(0, clean_chain.shape
 [0]),'alpha': clean_chain[:,0] , 'beta' : clean_chain[:,1],'gamma' :
 clean_chain[:,2],'delta' : clean_chain[:,3]})
return clean_chain,chain_df
```

Source Code 3.4: Running The Simulation.

3.5 Simulated Data Results

Several Data point were generated using this data $[0.55, 0.028, 0.84, 0.026, 33, 6]$ as for $\alpha, \beta, \gamma, \delta, X_0,$ *and* Y_0. We then ran the simulation several times to test

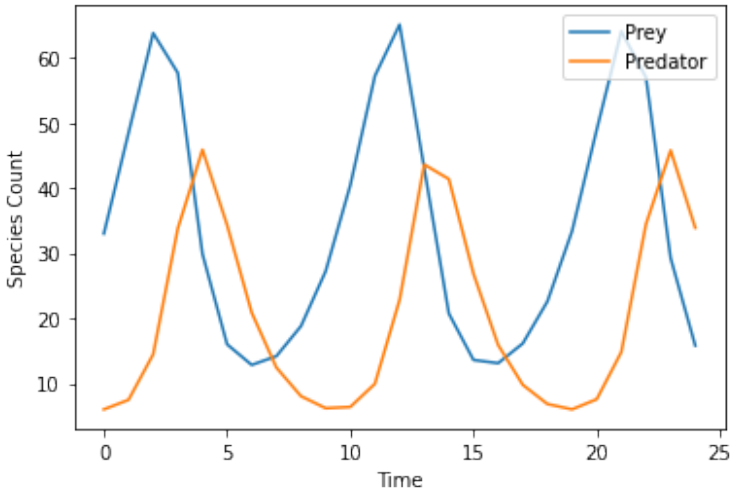


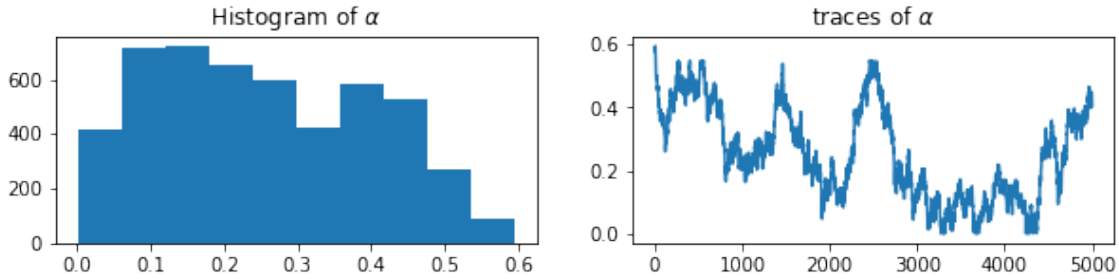
Figure 3.1: Simulated Data.

different time steps, 0.01 was found more suitable and gave accepted results. Figure 3.2 illustrate the single-chain traces and histogram results for $10,000$ iterations for each parameter. We can also show the relationships between the parameters to see if there is any correlations among them. Figure 3.3 describe the relationships between the estimated values of the four parameters and the kernel density function (kde) for each one of them.

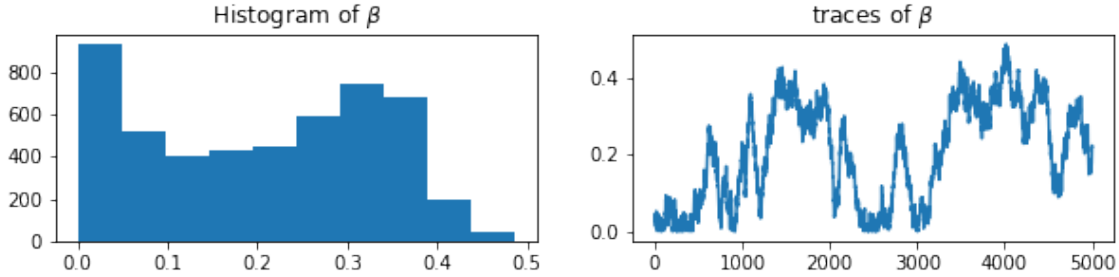
the final result is then obtained as a comparison between the actual values and the simulated values.

Results	α	β	γ	δ
Actual	0.55	0.028	0.84	0.026
Estimated	0.47	0.020	1.26	0.12
Error %	17	29	33	78

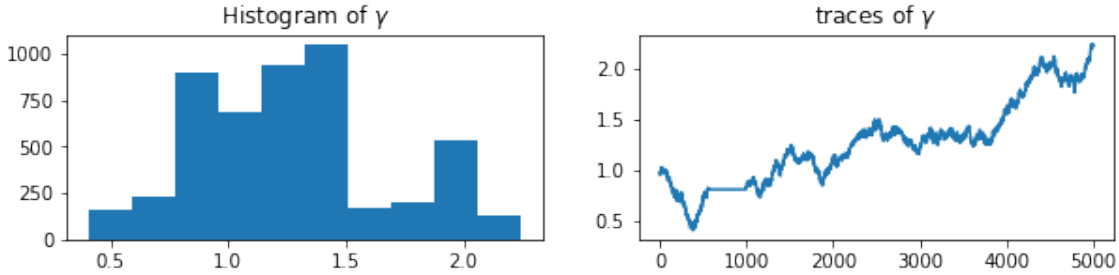
Table 3.1: Simulated Data Results and the Percentage Error.



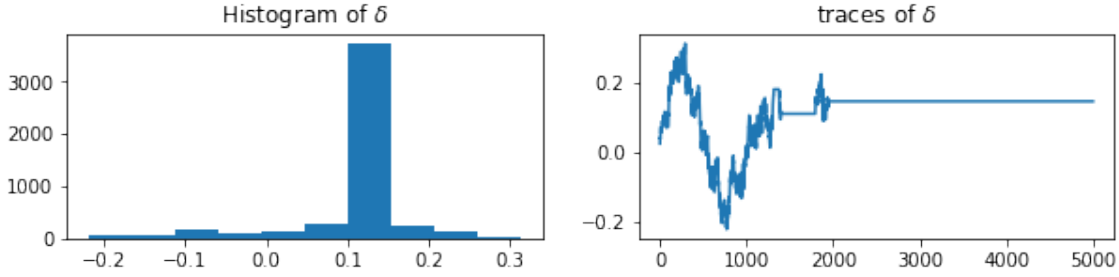
(a) α Chain and Histogram



(b) β Chain and Histogram



(c) γ Chain and Histogram



(d) δ Chain and Histogram

Figure 3.2: Chain Results for Simulated Data

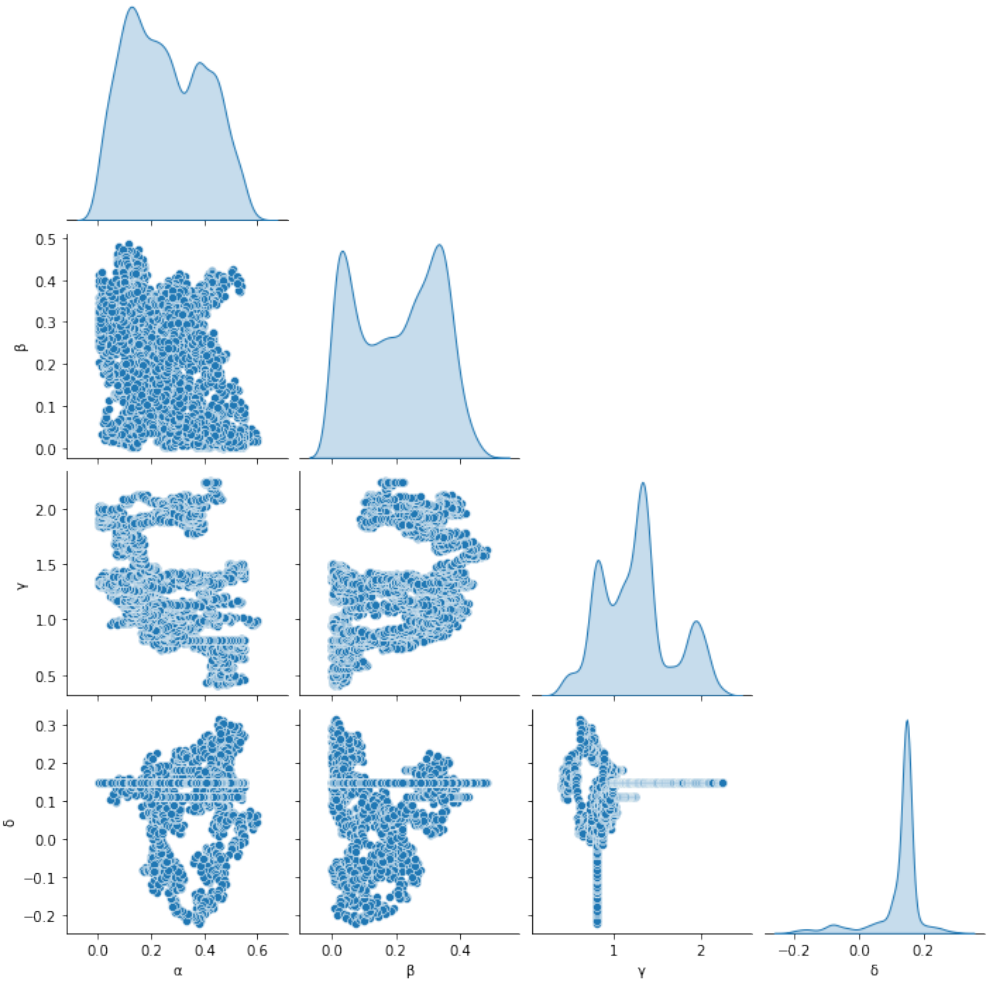


Figure 3.3: Parameters Relationships.

3.6 Model Results: Lynx and Hare Data

The species of interest in this case study are snowshoe hares and Canadian lynxes, a predator whose main feeding is to consume the snowshoe hares.

The main CSV file containing the data was collected from [5], while the original study is discussed by [8] where they provided plots of the number of pelts collected by the Hudson's Bay Company, the largest fur trapper in Canada, between the years 1821 and 1914. The discussion ranged over many natural factors affecting the population sizes, such as plagues, migrations, and weather-related events. They even discussed measurement confounders such as the fact that lynx are easier to trap when they are hungry and weak, which is correlated with a relative decline in the hare population. These factors will

(a) Hare (b) Lynx

Figure 3.4: Prey-Predator under study

not be considered since the model we are discussing here is just for illustration purposes. We will use the same strategy we used for the simulated data, but

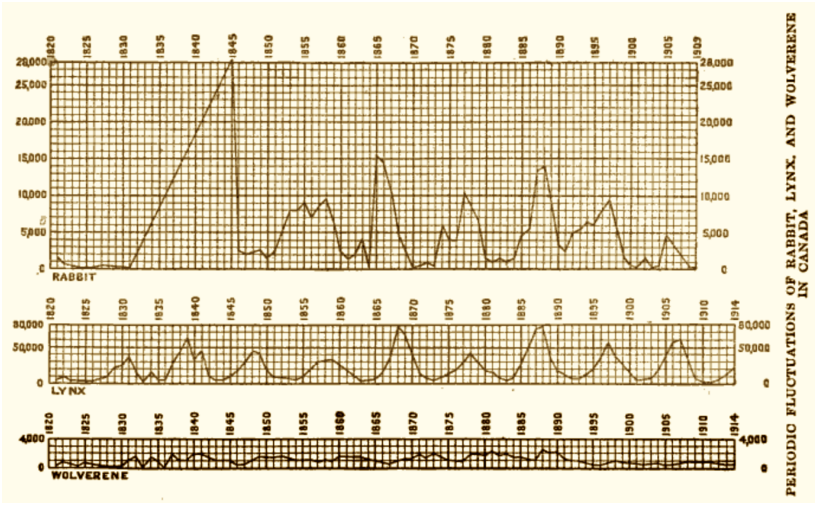


Figure 3.5: Lynx-Hare original Data.

instead of using one chain, several chains will be generated at once for a $10,000$ iterations.

```
data = pd.DataFrame(pd.read_csv('lynxhare.csv'))
# rename the data frame columns to match the LH function and get initial
    parameters :
data.rename(columns = {'Hare':'X', 'Lynx':'Y', 'Year' : 'time'}, inplace
    = True)
Xt0 = data.X[0]
Yt0 = data.Y[0]
sigma_X, sigma_Y = np.std(data.X), np.std(data.Y)
parms = [0.55,0.028,0.84,0.026, Xt0, Yt0,sigma_X, sigma_Y]
## to run several chains at the same time :
```

```
chain1,chains_df1 = runSimulation(data = data, niters = 10000, step_size
    = 0.01)
chains_df1['chain_idx'] = '1'
chain2,chains_df2 = runSimulation(data = data, niters = 10000, step_size
    = 0.01)
chains_df2['chain_idx'] = '2'
chain3,chains_df3 = runSimulation(data = data, niters = 10000, step_size
    = 0.01)
chains_df3['chain_idx'] = '3'
chain4,chains_df4 = runSimulation(data = data, niters = 10000, step_size
    = 0.01)
chains_df4['chain_idx'] = '4'
chains_df = pd.concat([chains_df1,chains_df2,chains_df3,chains_df4])
init_parms = ['\aplha', '\beta','\gamma', '\delta']
for i in range(4):
        fig, (ax1, ax2) = plt.subplots(1,2)
        fig.set_size_inches(10, 2)
        name = init_parms[i]
        ax1.hist(chain1[:,i])
        ax1.hist(chain2[:,i])
        ax1.hist(chain2[:,i])
        ax1.hist(chain3[:,i])
        ax1.set_title("Histogram of (name))
        ax2.plot(chain1[:,i])
        ax2.plot(chain2[:,i])
        ax2.plot(chain3[:,i])
        ax2.plot(chain4[:,i])
        ax2.set_title("traces of (name))
        plt.show()
```

Source Code 3.5: Generating Chains.

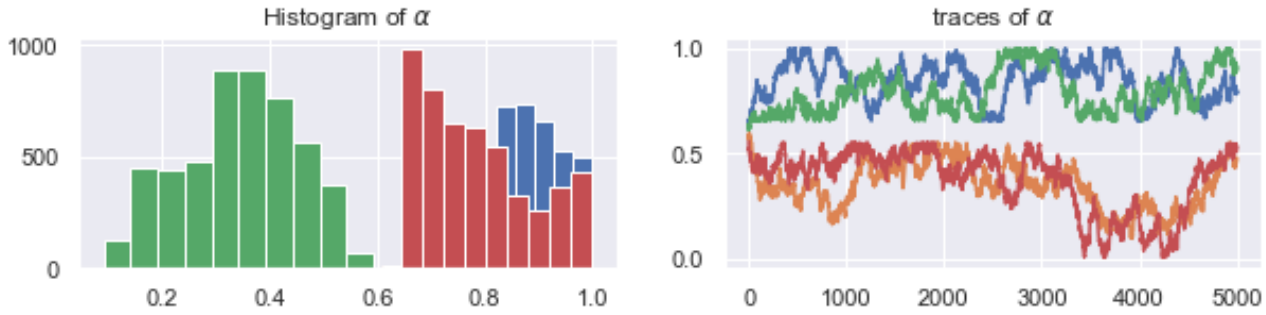
Figure 3.6 show each parameter chains results and its histogram, the chains convergence is some time not acceptable for one or two chains but we to some extent we can extrapolate some results out of it, this problem and the run time are the main weak point of this code which can be improved in the future.

Table 3.2 shows how the algorithm efficiency is changed through the chains.

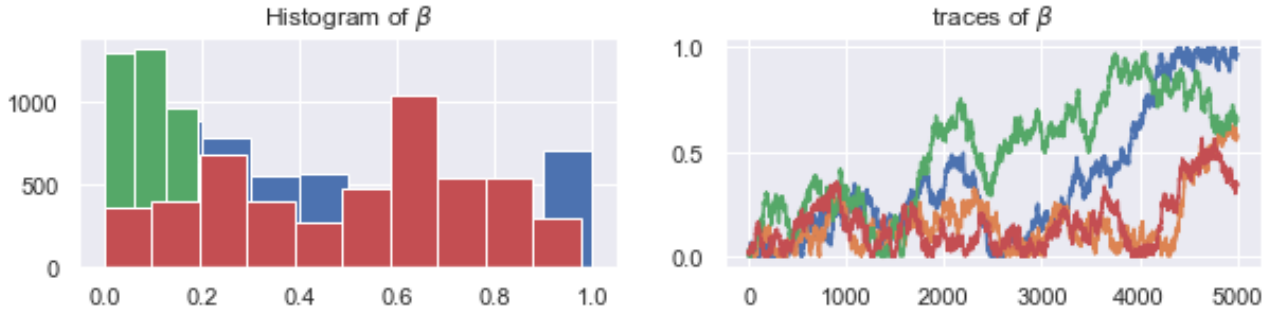
Efficiency	α	β	γ	δ
Chain 1	0.9789	0.9817	0.9293	0.0904
Chain 2	0.99	0.991	0.0379	0.2001
Chain 3	0.976	0.9919	0.9947	0.1495
Chain 4	0.9764	0.9899	0.9976	0.3151

Table 3.2: Chain Efficiency for Each Parameter.

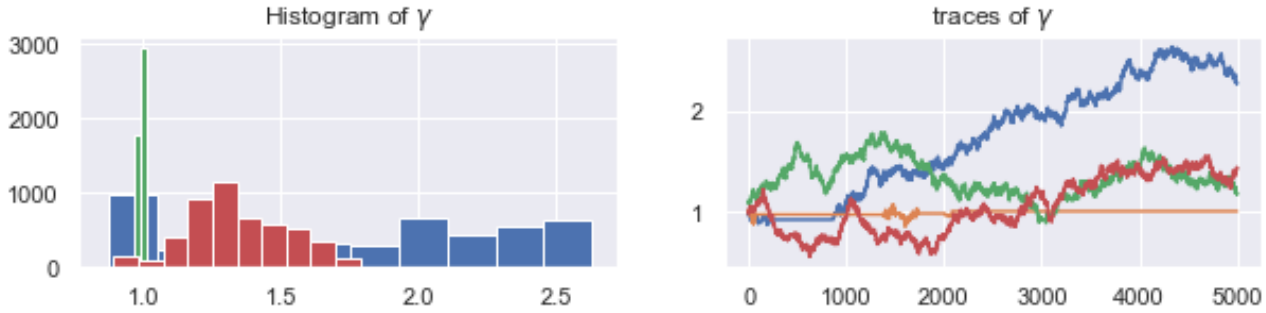
And we can get a sense of how are average values for the parameters are from



(a) α Chain and Histogram



(b) β Chain and Histogram



(c) γ Chain and Histogram

(d) δ Chain and Histogram

Figure 3.6: Chain Results for Lynx-Hare Data

the accepted results of chains.

```
# Average results  [0.55,0.028,0.84,0.026]
chains = [chain1,chain2,chain3,chain4]
for i in chains:
      alpha_average = np.round(np.average(i[:,0]),2)
      beta_average =np.round(np.average(i[:,1]),2)
      gamma_average = np.round(np.average(i[:,2]),2)
      delta_average =np.round(np.average(i[:,3]),2)
print("average values of alpha,beta,gamma and delta are  : ",
      [alpha_average,beta_average,gamma_average,delta_average])
```

Source Code 3.6: Calculating the Average Results.

The average result for each is as in table 3.3.

Efficiency	α	β	γ	δ
Average	0.37	0.016	1.06	0.020

Table 3.3: Average Result for Each Parameter.

Finally, we can use these results for estimating the species interaction and compare it with the original data we used for the inference. The code below used the average result for each parameter which we inferenced to generate a solution for the forward problem.

```
## using the average results :
result,_,_ = forwardProblem(time_interval =91, time_points = 91, parms =
    [alpha_average,beta_average,gamma_average,delta_average,Xt0, Yt0])
## Plot the result vs the actual data set
plt.plot(result.time,data.X, label = 'actual prey')
plt.plot(result.time,data.Y, label = 'actual predator')
plt.plot(result.time,result.X,'--',label = 'Inferenced prey')
plt.plot(result.time,result.Y,'--',label = 'Inferenced predator')
plt.rcParams["figure.figsize"] = (10,8)
plt.legend()
```

Source Code 3.7: Simulated vs. Real Data.

To some extent the comparison result is acceptable as in figure 3.7, the results were used also to predict the population change for a few years to the future.

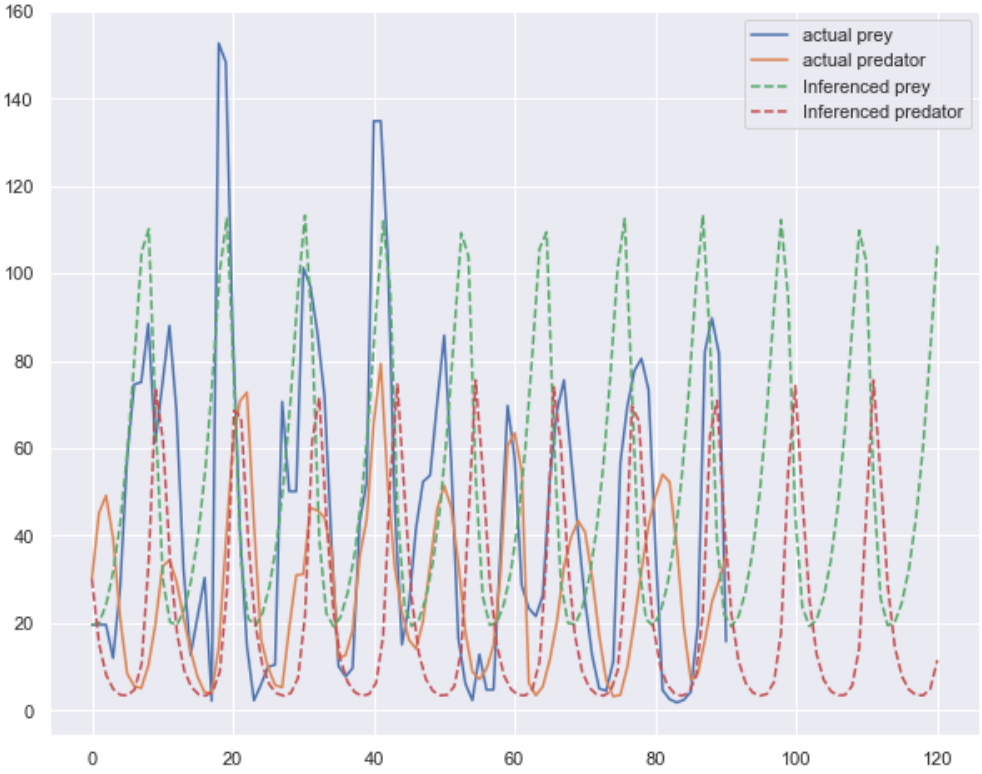


Figure 3.7: Lynx-Hare Original vs. Inferenced Results.

Conclusion

The inability to find closed-form solutions for dynamical systems makes parameter inference in non-linear ordinary differential equations a challenging task. We addressed the parameter estimation problem of the Lotka-Volterra dynamic system in this thesis. The Bayesian framework, specifically the Metropolis-Hastings Algorithm, was used to demonstrate the capabilities of the Markov chain Monte-Carlo in terms of sampling a distribution that will mimic the posterior distribution and eventually converge to give acceptable results. We used the Lotka-Volterra population dynamics equations, which are a pair of first-order nonlinear differential equations commonly used to explain the dynamics of biological systems in which predators and prey interact. The equations describe the fluctuating predator and prey populations.

We numerically solved the system by using the Python ODEINT module to obtain a form of a forward solution using a well-known problem of Cheetahs and Baboons. The framework that specifies how to get insights into the system parameters was then developed using a Bayesian Inference technique. We began by developing many functions and testing them with simulated data to validate the suggested technique. In addition, the Lynx-Hare problem, a well-known data set, was used to estimate the relevant Lotka-Volterra system parameters, and we generated several chains of the Metropolis-Hastings algorithm to gain a better understanding of how it will act on a complete or partial data set, and its efficiency was found to be fairly acceptable. Finally, we compared and predicted the equation's behavior over several time periods after the initial time data found in the data set.

Despite the accepted results, Metropolis-Hastings MCMC is susceptible to problems caused by the nature of the ODE likelihood function, where stiffness and computational efficiency mean that large samples require immense computation time, which is primarily a concern with the Python computa-

tional scheme. Many well-known Bayesian framework programs that employ the methodology first convert their code to C or C++ languages for quicker, reduced computation.

References

[1] Vicente Aboites, Jorge Francisco Bravo-Avilés, Juan Hugo García-López, Rider Jaimes-Reategui, and Guillermo Huerta-Cuellar. Interpretation and Dynamics of the Lotka–Volterra Model in the Description of a Three-Level Laser. *Photonics*, 9(1), 2022. doi:10.3390/photonics9010016.

[2] Alan Lazarus. *Using Gradient Matching to Accelerate Parameter Inference in Nonlinear Ordinary Di erential Equations*. Msc thesis, University of Glasgow, 2018. URL: https://theses.gla.ac.uk/30784/.

[3] Johnathan M. Bardsley and Tiangang Cui. A Metropolis-Hastings-Within-Gibbs Sampler for Nonlinear Hierarchical-Bayesian Inverse Problems. *Springer International Publishing*, pages 3–12, 2019. doi:10.1007/978-3-030-04161-8_1.

[4] Stephen Brooks, A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin. *Bayesian Data Analysis.*, volume 45. 1996. doi:10.2307/2988417.

[5] Bob Carpenter. Predator-Prey Population Dynamics : the Lotka-Volterra model in Stan, 2018. URL: https://mc-stan.org/users/documentation/case-studies/lotka-volterra-predator-prey.html.

[6] Janice McCarthy Cliburn Chan. Computational Statistics in Python, 2022. URL: http://people.duke.edu/$\sim$ccc14/sta-663/index.html.

[7] André M De Roos and Lennart Persson. Size-dependent life-history traits promote catastrophic collapses of top predators. *Proceedings of the National Academy of Sciences*, 99(20):12907–12912, 2002.

[8] C. G. Hewitt. The Conservation of the Wild Life of Canada. *Canadian Historical Review*, 2(4):401–403, 1921. doi:10.3138/chr-02-04-br16.

[9] Hanwen Huang, Andreas Handel, and Xiao Song. A Bayesian approach to estimate parameters of ordinary differential equation. *Computational Statistics*, 35(3):1481–1499, sep 2020. URL: http://link.springer.com/10.1007/s00180-020-00962-8, doi:10.1007/s00180-020-00962-8.

[10] C Huber. Stata. Introduction to Bayesian statistics, part 2: MCMC and the Metropolis-Hastings algorithm, 2016. URL: https://www.youtube.com/watch?v=OTO1DygELpY&feature=youtu.be.

[11] Hui Chih Hung, Yu Chih Chiu, Huang Chen Huang, and Muh Cherng Wu. An enhanced application of Lotka–Volterra model to forecast the sales of two competing retail formats. *Computers and Industrial Engineering*, 109:325–334, jul 2017. URL: https://linkinghub.elsevier.com/retrieve/pii/S0360835217302255, doi:10.1016/j.cie.2017.05.022.

[12] David Johnson, Robert D. Mason, and Douglas A. Lind. *Statistical Techniques in Business and Economics (8th Edition).*, volume 45. McGrawHill, 1994. doi:10.2307/2584153.

[13] Lasse Koskinen. *A First Course in Bayesian Statistical Methods by Peter D. Hoff*, volume 78. Springer Dordrecht Heidelberg London New York, 2010. doi:10.1111/j.1751-5823.2010.00109_17.x.

[14] John K. Kruschke. *Doing Bayesian data analysis: A tutorial with R, JAGS, and Stan, second edition.* Elsevier Inc., 2 edition, 2014. URL: http://dx.doi.org/10.1016/B978-0-12-405888-0.09999-2, doi:10.1016/B978-0-12-405888-0.09999-2.

[15] John K. Kruschke. Doing Bayesian data analysis: A tutorial with R, JAGS, and Stan, second edition. *Doing Bayesian Data Analysis: A Tutorial with R, JAGS, and Stan, Second Edition*, (ISBN: 978-0-12-405888-0):1–759, 2014. doi:10.1016/B978-0-12-405888-0.09999-2.

[16] A.J. Lotka. Contribution to the Theory of Periodic Reaction. *J. Phys. Chem*, 14(3):271–274, 1910.

[17] Brian J. Reich and Sujit K. Ghosh. *Bayesian statistical methods.* CRC Press Taylor-Francis Group, 2019. doi:10.1201/9780429202292.

[18] Monika Sharma and Praveen Kumar. Chemical oscillations 2. mathematical modelling. *Resonance*, 11:61, 02 2006. doi:10.1007/BF02837274.

[19] Tina Toni, David Welch, Natalja Strelkowa, Andreas Ipsen, and Michael P.H Stumpf. Approximate Bayesian computation scheme for parameter inference and model selection in dynamical systems. *Journal of The Royal Society Interface*, 6(31):187–202, feb 2009. URL: https://royalsocietypublishing.org/doi/10.1098/rsif.2008.0172, doi:10.1098/rsif.2008.0172.

[20] Ville Hyvonen and Topias Tolonen. *Bayesian Inference 2019*. Lecture Notes, 2019. URL: https://vioshyvo.github.io/Bayesian_inference/.

[21] Pauli Virtanen and Gommers. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi:10.1038/s41592-019-0686-2.

[22] V Volterra. Variazioni e fluttuazioni del numero d'individui in specie animali conviventi. *Memoria della regia accademia nazionale del lincei ser.*, 62:31–113, 1926.

[23] Wikipedia. Truncated Normal Distribution. URL: https://en.wikipedia.org/wiki/Truncated_normal_distribution.

[24] Wikipedia. Lotka – Volterra equations, 2022. URL: https://en.wikipedia.org/wiki/Lotka\OT1\textendashVolterra_equations.

[25] D.J. Spiegelhalter W.R. Gilks, S. Richardson. *Markov Chain Monte Carlo in Practice*, volume 1. 1996. doi:10.1007/978-1-4899-4485-6_9.