Markov Chain Monte Carlo and Probabilistic Programming Using Monads

A Proof of Concept

Søren Emil Schmidt, s144285
Thomas M. Pethick, s144448

Kongens Lyngby 2017

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 45 25 30 31
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

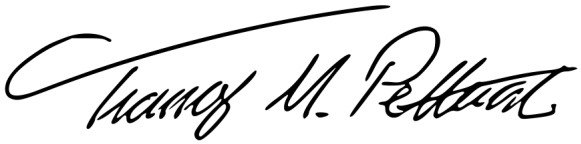
This paper supplies the necessary theoretical knowledge to understand the Markov Chain Monte Carlo (MCMC) method by using the Metropolis-Hastings algorithm to do Bayesian Inference.

In addition, it presents the tools needed to model a Probabilistic Programming Language in JavaScript using generators. It does so by first exploring the modeling of an embedded domain specific language in Haskell using monads. As there are, at the time of writing, no other libraries using generators to do MCMC in JavaScript, this paper can be seen as a proof of concept.

With the aim of making Bayesian Inference more approachable, this work combines the theory of Bayesian Inference with the programming tool-set needed to build a working application.

To do this the subjects of Markov Chains, Monte Carlo sampling, Bayesian Networks, Functional Programming and Monads are all presented, as this provides a thorough foundation in both the theoretical and the applied parts of probabilistic programming.

Preface



Søren Emil Schmidt, s144285

Thomas M. Pethick, s144448

5th June 2017

Contents

Chapter 1

Introduction

One class of probabilistic modeling, is Bayesian Modeling. One seeks to describe the world using probability distributions. Bayes' Theorem is employed in order to use observed data to update these distributions, thus determining the posterior distribution, a procedure also known as Bayesian Inference,

$$P(\theta|X) = \frac{1}{Z}P(X|\theta)P(\theta),$$

where θ refers to the parameters of the model, X is the observed data, $P(X|\theta)$ is the likelihood of X given the parameters of the model, $P(\theta)$ is the prior belief in the parameters of the model, and $\frac{1}{Z}$ is a normalization factor in order to ensure that the posterior $P(\theta|X)$ is a probability distribution.

The problem is that the normalization constant is often intractable, creating the need for methods to handle this problem, and allowing for Bayesian Inference with regard to the posterior distribution. One method is using Markov Chain Monte Carlo, which is what will be discussed in this paper.

In addition, since Bayesian Inference is a powerful tool allowing for complex probabilistic models to be made and worked with, this tool is made available to non-experts, through an implementation of a probabilistic programming language (PPL) in JavaScript.

1.1 Theoretical Foundation

This project aims to introduce the reader to the theory behind Bayesian Modeling. To do this, the theory of Markov Chains is described in depth, in order to make the reader comfortable with the subject.

Subsequently, sampling and simulation as a tool to solve different problems is introduced. Here the Metropolis-Hastings algorithm is described in great detail, in order for the reader to understand what happens inside a probabilistic programming language.

This discussion is used to motivate Bayesian Network, and the problems that can be described by them. This is done to give the reader the theoretical understanding of probabilistic modeling needed,

in order to fully understand the problem that probabilistic programming aims to solve. Namely to make probabilistic programming available to researchers without expert knowledge within probability.

All of this will be done using examples drawing parallels to the real world, in the hope of making the subject more approachable for the reader. With the main takeaway being how Bayesian Inference is done in probabilistic models, in order to motivate the simplicity that probabilistic programming offers.

1.2 Practical Foundation

The main focus is to build a probabilistic programming language, based on the strong theoretical foundation created for the approximate inference algorithm used. We further show how problems can be modeled as a Bayesian Inference problem using Bayesian Networks and then run using the PPL.

The implementation is inspired by the work done in *Probabilistic Programming with Monads* [42] which uses a free monad to built a PPL embedded in Haskell. We explore how a probabilistic programming language can be describe in Haskell using monads. A PPL in JavaScript is implemented using methods inspired by Haskell combined with the lightweight Metropolis-Hastings algorithm by Wingate et al. [50]. The main contribution is using the realization that generators can mimic Haskells do-notation to create an embedded domain specific language in JavaScript for probabilistic programming.

To the best of our knowledge this is the first implementation of a probabilistic programming language in JavaScript using generators.

1.3 Structure of the paper

We start with the foundation for the approximate inference algorithm by covering Markov Chains in Chapter 2 and subsequently combining it with sampling in Chapter 3. In Chapter 4 we describe how this relates to Bayesian Networks and show how it presents a good framework for modeling problems. This is done in order to get a firm mathematical foundation for the theory behind the algorithm. We then construct the probabilistic programming language.

To do this, Haskell is introduced and an embedded domain specific language (DSL) is modeled using the free monad which is then transferred to JavaScript in Chapter 5. In Chapter 6 a Metropolis-Hastings algorithm suitable for PPLs is described and it is used as the interpreter for the DSL. How to use the language is then shown and evaluated in Chapter 7. Finally, we discuss improvements, performance and related languages in Chapter 8, followed by a conclusion in Chapter 9.

Chapter 2

Stochastic Processes

A stochastic process, $\mathbf{X} = \{X_t, t \in T\}$, is a set of random variables. This means that for $t \in T$, where T is the index set, X_t is a random variable taking values in some state space S. One example that helps understanding this concept, is the simple random walk. Let $\mathbb{Z}$ be the state space, and $X_t \in \mathbb{Z}$ be the state of the random walk at time t. The walk then takes a step up ($X_{t+1} = X_t + 1$) with probability p, and a step down ($X_{t+1} = X_t - 1$) with probability $q = 1 - p$. Compactly this can be written as follows:

$$X_{t+1} = \begin{cases} X_t + 1, & \text{prob: } p \\ X_t - 1, & \text{prob: } q = 1 - p. \end{cases}$$

If $p = \frac{1}{2}$, the walk is said to be symmetric. Such a walk, with $X_0 = 0$ and $1,000$ steps, can be seen on Figure 2.1.

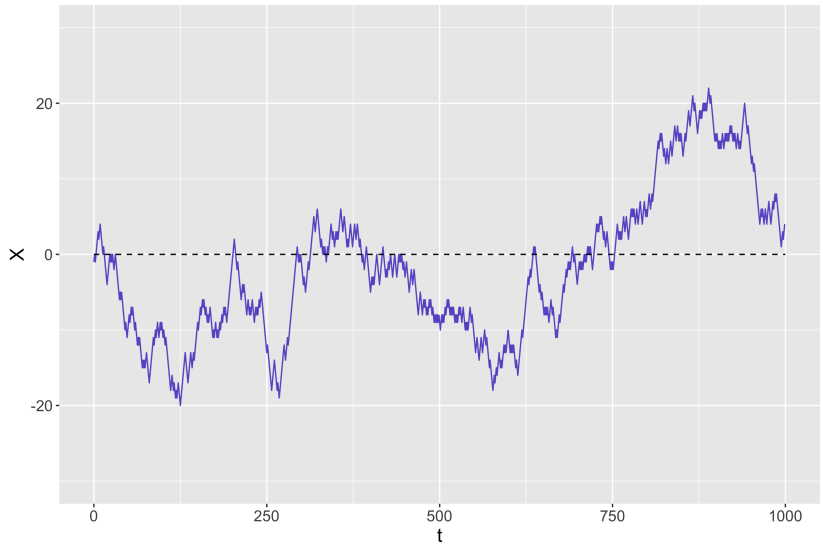


Figure 2.1: A symmetric random walk

As T is a countable space in this example, the process is said to be a discrete-time stochastic process. Had T been a continuum, then the process would be a continuous-time stochastic process.

2.1 Markov Chains

An interesting subclass of stochastic processes is the class of Markov Chains. These are stochastic processes where given the present, the future does not depend on the past. Let S be a system which is observed at discrete moments of time $t = 0, 1, 2, \ldots$, and let $X_t \in S$ be a random variable, then this can formally be written as follows

$$\mathbb{P}(X_{t+1} = j | X_t = i, X_{t-1} = i_{t-1}, \ldots, X_1 = i_1, X_0 = i_0) = \mathbb{P}(X_{t+1} = j | X_t = i). \tag{2.1}$$

From this it is seen that the random walk is in fact a Markov Chain, with discrete time indexing. Furthermore, this property also defines continuous Markov Chains, however the mathematical definition has been left out here.

Something that Markov Chains are often used for, is modelling the transitioning between different states in a system. In a state space S which is countable and discrete, this can be modelled using a transition matrix, P, where $P_{ij,t}$ is the probability of moving from state i, to state j at time t. That is

$$P_{ij,t} = \mathbb{P}(X_{t+1} = j | X_t = i).$$

If $P_{ij,t}$ is the same for all $t \in T$, then the Markov Chain is said to be time homogeneous. This indicates that the transition probabilities does not vary with time, and as such the transition matrix P does not change with time. Moving forward the Markov Chains are all time homogeneous, and therefore the parameter t will be left out.

As P_{ij} indicates the probability of going from state i to state j, and since the state must be updated with the time, the following properties hold

$$P_{ij} \geq 0, \quad i, j \geq 0; \qquad \sum_{j=0}^{\infty} P_{ij} = 1, \quad i = 0, 1, \ldots \quad [38]. \tag{2.2}$$

From this, it is possible to define the one-step transition matrix P, here shown for a countable state space

$$P = \begin{bmatrix} P_{00} & \ldots & P_{0j} & \ldots \\ \vdots & & \vdots & \\ P_{i0} & \ldots & P_{ij} & \ldots \\ \vdots & & \vdots & \end{bmatrix}.$$

Having defined the one-step transition probability, this can be extended to look at the n-step transition probability. Let this be defined as

$$P_{ij}^n = \mathbb{P}(X_{t+n} = j | X_t = i)$$

Using the *Chapman-Kolmogorov equations* [38, p. 167], the n-step transition probabilities can be

calculated as follows

$$\begin{aligned} P_{ij}^{n+m} &= \mathbb{P}(X_{n+m} = j | X_0 = i) \\ &= \sum_{k=0}^{\infty} \mathbb{P}(X_{n+m} = j, X_n = k | X_0 = i) \\ &= \sum_{k=0}^{\infty} \mathbb{P}(X_{n+m} = j | X_n = k, X_0 = i) \mathbb{P}(X_n = k | X_0 = i) \\ &= \sum_{k=0}^{\infty} P_{kj}^m P_{ik}^n = \sum_{k=0}^{\infty} P_{ik}^n P_{kj}^m \end{aligned} \tag{2.3}$$

Let $P^{(n)}$ denote the matrix consisting of n-step transition probabilities P_{ij}^n

$$P^{(n)} = \begin{bmatrix} P_{00}^n & \dots & P_{0j}^n & \dots \\ \vdots & & \vdots & \\ P_{i0}^n & \dots & P_{ij}^n & \dots \\ \vdots & & \vdots & \end{bmatrix}. \tag{2.4}$$

From (2.3) it then follows that

$$P^{(n+m)} = P^{(n)} \cdot P^{(m)}, \tag{2.5}$$

which, due to P being a stochastic matrix, is also defined for the infinite dimensional case.

As $P^{(1)} = P$ trivially, (2.5) implies

$$P^{(n)} = P \cdot P^{(n-1)} = P^2 \cdot P^{(n-2)} = \dots = P^n. \tag{2.6}$$

If it is possible to get from state i to j in a finite amount of steps, then j is said to be accessible from i, formally written as $(i \to j)$. Using the n-step notation, this means that $\exists n \geq 0$ such that $P_{ij}^n > 0$, as this indicates a non-zero probability of going from i to j in n steps.

Further, if i is also accessible from j, then the two are said to be communicating, formally written as $(i \leftrightarrow j)$. If two states communicate, they are said to be part of the same class, and if all states communicate, that is if it is possible to get to any state from any starting point, then the Markov Chain is said to be irreducible and contains only one class.

If the return to a given state i, has to happen in a multiple of d steps, then the state i is said to have period d, which is defined as follows

$$d = \gcd \{ n \in \mathbb{N} \mid \mathbb{P}(X_n = i | X_0 = i) > 0 \}.$$

If $d = 1$, then the state is said to be aperiodic. From Proposition 4.2.2 in [38, p. 169] it is known that periodicity is a class property. This means that if the Markov Chain is irreducible and one of the states is aperiodic, then all of the states are aperiodic.

Now let f_{ij}^n be defined as the first passage probability of going from state i to j in n steps, then by definition

$$\begin{aligned} f_{ij}^0 &= 0 \\ f_{ij}^n &= \mathbb{P}(X_n = j, X_k \neq j, k = 1, \dots, n-1 | X_0 = i). \end{aligned}$$

Then define f_{ij} as the sum of f_{ij}^n, $\forall n \in \mathbb{N}$

$$f_{ij} = \sum_{n=1}^{\infty} f_{ij}^n.$$

Using this, the term recurrence can be defined. A state i is said to be recurrent if $f_{ii} = 1$. This means that starting in state i, the probability of returning is 1. If this is not the case, then the state is said to be transient. Moreover, from Corollary 4.2.4 [38, p. 171] it is known that recurrence, like periodicity, is a class property.

Let $\boldsymbol{\mu}_{jj}$ be the expected number of steps necessary in order to return to state j, given that the process starts in j, then from [38, p. 173]

$$\boldsymbol{\mu}_{jj} = \begin{cases} \infty & \text{if } j \text{ is transient} \\ \sum_{n=1}^{\infty} n f_{jj}^{n} & \text{if } j \text{ is recurrent.} \end{cases}$$

Now let j be a recurrent state. It is then called positive recurrent if $\boldsymbol{\mu}_{jj} < \infty$. In other words, if the expected number of transitions to return is finite, then the state is said to be positive recurrent. Positive recurrence can also be determined by examining the properties of the Markov Chain's stationary distribution, however first this concept has to be defined.

Let $\boldsymbol{\pi}_j$, $j \geq 0$ be a probability distribution describing a Markov Chain, it is then said to be stationary if the following equality holds

$$\boldsymbol{\pi}_j = \sum_{i=0}^{\infty} \boldsymbol{\pi}_i P_{ij}, \quad j \geq 0 \quad \text{[38, p. 174].} \tag{2.7}$$

Using induction this definition can be used to show that the stationary distribution does not change over time. Let X_0 follow a stationary distribution, such that $\boldsymbol{\pi}_j = \mathbb{P}(X_0 = j)$ for $j \geq 0$. Then it follows from (2.7), that

$$\begin{aligned} \mathbb{P}(X_1 = j) &= \sum_{i=0}^{\infty} \mathbb{P}(X_1 = j | X_0 = i) P(X_0 = i) \\ &= \sum_{i=0}^{\infty} P_{ij} \boldsymbol{\pi}_i \\ &= \boldsymbol{\pi}_j. \end{aligned}$$

Assume that $\mathbb{P}(X_{n-1} = i) = \boldsymbol{\pi}_i$, then it can be shown to hold for any successive n, completing the induction proof

$$\begin{aligned} \mathbb{P}(X_n = j) &= \sum_{i=0}^{\infty} \mathbb{P}(X_n = j | X_{n-1} = i) \mathbb{P}(X_{n-1} = i) \\ &= \sum_{i=0}^{\infty} P_{ij} \boldsymbol{\pi}_i \\ &= \boldsymbol{\pi}_j. \end{aligned}$$

It can be seen that for a stationary distribution, the distribution does not change, and X_n will follow the same distribution for all n [38, p. 174].

This can also be seen in the context of the transition matrix. Let $\boldsymbol{\pi}$ be the stationary distribution's probability vector, then

$$\boldsymbol{\pi} P = \boldsymbol{\pi}. \tag{2.8}$$

That is, the probability distribution does not change by application of the transition matrix.

All of this leads to Theorem 4.3.3 from [38, p. 175], which states that for an irreducible aperiodic Markov Chain in which all states are positive recurrent, the stationary distribution $\boldsymbol{\pi}$ is unique, and is found in the limit of the transition matrix P

$$\boldsymbol{\pi}_j = \lim_{n \to \infty} P_{ij}^{n}, \quad j = 0, 1, 2, \ldots \quad . \tag{2.9}$$

This is an important point as it means that the stationary distribution can be estimated by looking at the limit of the n^{th} step transition matrix.

Finally, if a Markov Chain is positive recurrent, irreducible and aperiodic, then it is called an ergodic Markov Chain.

2.1.1 Practical Markov Chain Example

To give an example of the uses of Markov Chains, the weather is modelled using an ergodic Markov Chain. Living in Denmark, it is categorized into 3 different states: *Rainy*, *Cloudy*, and *Sunny*. Making assumptions regarding the transition probabilities, it is possible to define the transition matrix P. This can be seen in (2.10).

$$P = \begin{array}{c} \\ \textit{Rainy} \\ \textit{Cloudy} \\ \textit{Sunny} \end{array} \begin{array}{c} \begin{array}{ccc} \textit{Rainy} & \textit{Cloudy} & \textit{Sunny} \end{array} \\ \begin{bmatrix} 0.2 & 0.2 & 0.6 \\ 0.3 & 0.4 & 0.3 \\ 0.4 & 0.3 & 0.3 \end{bmatrix} \end{array}. \tag{2.10}$$

This shows the assumption that there is a 20% risk that a *rainy* day follows a *rainy* day, whereas there is a 60% chance that a *sunny* day follows, etc.

Let X_0 be the state of the weather and τ_0 the probability vector denoting the probability of the weather at step $n = 0$. For example, $\mathbb{P}(X_0 = \textit{Rainy}) = 1 \Leftrightarrow \tau_0 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$. Then the probability of the different types of weather the next day, at step $n = 1$, can be found by simply multiplying τ_0 onto the transition matrix.

$$\tau_1 = \tau_0 P = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0.2 & 0.2 & 0.6 \\ 0.3 & 0.4 & 0.3 \\ 0.4 & 0.3 & 0.3 \end{bmatrix} = \begin{bmatrix} 0.2 & 0.2 & 0.6 \end{bmatrix}.$$

Doing this it is seen how the distribution of the weather has changed

$$\begin{aligned} \mathbb{P}(X_1 = \textit{Rainy} \mid X_0 = \textit{Rainy}) &= 0.2 \\ \mathbb{P}(X_1 = \textit{Cloudy} \mid X_0 = \textit{Rainy}) &= 0.2 \\ \mathbb{P}(X_1 = \textit{Sunny} \mid X_0 = \textit{Rainy}) &= 0.6. \end{aligned}$$

When calculating τ_2, it is seen how the Markovian property from (2.1) works, in that the future is independent of the past, given the present. It does not matter if it rained the first day, all that matters is knowing that it is *sunny* at $n = 1$

$$\mathbb{P}(X_2 = \textit{Cloudy} \mid X_1 = \textit{Sunny}, X_0 = \textit{Rainy}) = \mathbb{P}(X_2 = \textit{Cloudy} \mid X_1 = \textit{Sunny}) = 0.3.$$

In order to calculate the entire probability vector two steps into the future, we can simply use the n-step transition matrix as defined in (2.4) due to the property from (2.6) to show the following equality

$$\tau_2 = \tau_1 P = (\tau_0 P) P = \tau_0 P^2.$$

Extending this to a general amount of steps n, this allows for the following simplification

$$\tau_n = \tau_0 P^n.$$

Thus the probability of the weather n steps into the future can easily be calculated by simply applying the n-step transition matrix to the initial distribution.

Utilizing this, the long run average of the different states, can be examined. This is done by checking how much time is spent in each state, as $n \to \infty$

$$\lim_{n\to\infty} P^n = \begin{array}{c} \\ \textit{Rainy} \\ \textit{Cloudy} \\ \textit{Sunny} \end{array} \begin{array}{c} \begin{array}{ccc} \textit{Rainy} & \textit{Cloudy} & \textit{Sunny} \end{array} \\ \begin{bmatrix} 0.31 & 0.30 & 0.39 \\ 0.31 & 0.30 & 0.39 \\ 0.31 & 0.30 & 0.39 \end{bmatrix} \end{array}. \tag{2.11}$$

Here it is seen that the weather, in the long run, is independent of the initial weather and has the following distribution

$$\left.\begin{array}{ll} \mathbb{P}(X_n = \textit{Rainy}) & = 0.31 \\ \mathbb{P}(X_n = \textit{Cloudy}) & = 0.30 \\ \mathbb{P}(X_n = \textit{Sunny}) & = 0.39 \end{array}\right\} \quad n \to \infty$$

As the Markov Chain defined by (2.10) is an ergodic chain, it converges towards what is known as its stationary distribution, $\boldsymbol{\pi}$, as seen in (2.9). Therefore it should satisfy the property seen in (2.8)

$$\boldsymbol{\pi} P = \begin{bmatrix} 0.31 & 0.30 & 0.39 \end{bmatrix} \begin{bmatrix} 0.2 & 0.2 & 0.6 \\ 0.3 & 0.4 & 0.3 \\ 0.4 & 0.3 & 0.3 \end{bmatrix} = \begin{bmatrix} 0.31 & 0.30 & 0.39 \end{bmatrix} = \boldsymbol{\pi}$$

This indicates that it was possible to determine the stationary distribution, by looking at the limit of the transition matrix. The implications of this is that if the Markov Chain has a stationary distribution, then it can be estimated simply by initializing a state, moving around in the state space for a sufficiently long time, and then looking at the distribution of the states visited. Here sufficiently is a vague term, which will be developed upon in the following section.

Furthermore, as the Markov Chain described by the transition matrix is aperiodic, irreducible and positive recurrent, and as such ergodic, the stationary distribution is unique. Therefore the initial state, in this case $X_0 = \textit{Rainy}$, is irrelevant, as any initial state will converge towards the same unique stationary distribution, as seen in (2.9). This is also evident from the fact that all of the rows are equal in (2.11).

Chapter 3

Sampling

What is the chance that a random dealing of a given solitaire is solvable? While this might initially seem like a simple probability question to answer that it is not necessarily the case. In fact, this question is what inspired Stanislaw Ulam to develop the Monte Carlo method. The thought behind it was that instead of trying to solve the problem using combinatorial calculations and abstract thinking, the solution could be estimated by simply laying out the given solitaire a lot of times, and observing how often it was solvable [9].

Simple as it sounds, this is the essence of the Monte Carlo method - Instead of solving a problem analytically, it is more efficient to simply model the problem using a probabilistic interpretation, and then have a computer evaluate the model thousands of times. A method that has become increasingly useful, as computers allow for faster and faster calculations - What would typically take hours to do by hand, like playing a hundred games of solitaire, can be done in seconds on a modern computer.

This is also applicable to more complex probabilistic models, such as Bayesian Networks, where Bayesian Inference is used. Say that some data X have been observed, then it is of interest to determine the set of parameters θ that are most likely to result in the observed data. This is done by using the observed data to update the posterior $P(\theta|X)$. However, direct application of Bayes' Theorem as in (3.1), is often intractable due to the nature of $P(X)$,

$$P(\theta|X) = \frac{P(X|\theta)P(\theta)}{P(X)}. \tag{3.1}$$

However, there exists methods that make it possible to sample from $P(\theta|X)$, in order to get information about the distribution. One of these is the Markov Chain Monte Carlo method. Before it can be described however, the Monte Carlo method itself must be defined.

3.1 The Monte Carlo Method

The Monte Carlo method is simply the use of random sampling to get a numerical estimate of a desired value. This is typically done by modelling the problem using random variables that follow distributions which can be sampled from using a pseudo random number generator. This covers

distributions such as $\mathcal{N}(\mu, \sigma^2)$, $\mathcal{U}(a, b)$, Exp(n, p), Pois(λ) etc. In Probabilistic Programming, these are known as elementary random primitives (ERP), which will be described in more detail in Chapter 6.

This has a range of applications, from risk analysis where different scenarios can be simulated, to numerical estimates of values. In essence, if the problem has a probabilistic interpretation using probability distributions from which it is possible to sample effectively, then a solution can be estimated using the Monte Carlo method.

One example is the analysis of a series of random events that are dependent on each other. This is done by modelling the phenomenon of interest, and then running the scenario multiple times, while observing the different outcomes. This can for example be used to determine the optimal number of operators at a call center, or to calculate the most valuable properties in Monopoly. For the Monopoly example, this could be calculated analytically, taking into account the distribution of two fair dice, as well as all of the chance cards with their various effects. However, it can also be solved by simply simulating a large amount of games, and seeing which properties are most often visited, thus giving an estimate of the most valuable properties in the game.

3.1.1 Practical Monte Carlo Method Example

Simulation can also be used to estimate specific values of interest. One such example is the estimation of π. There are many different ways of estimating π, including infinite series and geometric proofs. However, one easy and quick way is through the use of the Monte Carlo method.

Given a square with width w, and a circle inscribed, as seen on Figure 3.1, it is possible to estimate π.

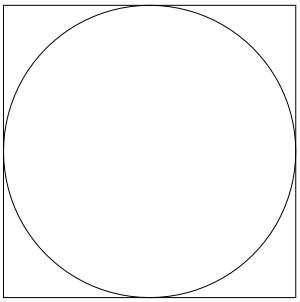


Figure 3.1: Square with circle inscribed

This is done by calculating the ratio between the area of the two figures. Let A_c be the area of the circle, and A_s be the area of the square

$$\frac{A_c}{A_s} = \frac{\left(\frac{w}{2}\right)^2 \pi}{w^2} = \frac{\frac{w^2}{4}\pi}{w^2} = \frac{\pi}{4}$$

From this it is seen that the problem of estimating π, has turned into determining the ratio between the area of the square, and the circle. Using the Monte Carlo method, this can be done by sampling uniformly within the square. As the ratio between the area of the square and circle can then be estimated by simply taking the ratio of the number of points within the circle and those within the square.

While this would be a very time consuming experiment to physically carry out, it can easily, and

quickly, be simulated. This can be seen on Figure 3.2, where the points within the circle have been colored red, and those outside of it have been colored blue.

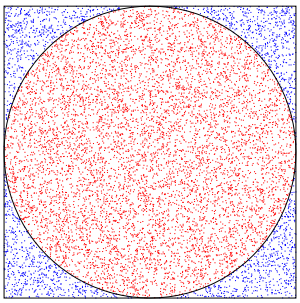


Figure 3.2: Using $N = 10,000$, $\tilde{\pi} = 3.1356$

Sampling $10,000$ points uniformly within the square, yielded an estimate of $\tilde{\pi} = 3.1356$. Rounding this to two digits, this gives the standard $\tilde{\pi} = 3.14$. Thus the problem of estimating π, was solved by interpreting it as a probabilistic question.

This was a rather simple problem, with a very simple solution. However, this approach can be extended to also cover more complex tasks, with one such extension being Markov Chain Monte Carlo. Markov Chain Monte Carlo is interesting as it does not simply estimate a certain value, like the example with π, but allows for a way to estimate intractable distributions, where the normalization factors are not known like the one mentioned in the introduction.

3.2 Markov Chain Monte Carlo

As mentioned in Chapter 2, one of the properties of ergodic Markov Chains, is that their distribution converges towards a unique stationary probability distribution. Utilizing this, a Markov Chain can be constructed such that the limit of the distribution it follows, is the target distribution from which information is wanted. Once such a Markov Chain is defined, it is then simply a question of letting the distribution it follows converge, and then sampling from the Markov Chain, as the samples will then follow the target distribution. As the samples are the states in a Markov Chain, they are a realization of a Markov Chain themselves, and since they are sampled, it is a use of the Monte Carlo method, which is why it is called Markov Chain Monte Carlo (MCMC).

One caveat is that the samples are correlated. This can be handled in different ways, however, that is not the focus of this project.

3.2.1 The Metropolis-Hastings algorithm

The problem then becomes to create a Markov Chain with the desired stationary distribution. One algorithm for doing this, is the Metropolis-Hastings algorithm. The way it achieves this, is through the use of two distributions. The first proposing a change of state, and the second determining the probability of accepting the change of state. As such Metropolis-Hastings works by first initializing a position in the sample space. Then a move to a random position in the state space is proposed, where the distribution used for the proposal is problem dependent, with the best proposal distributions having attributes similar to the target distribution. Having proposed a new position, the probability of accepting this move is then calculated, with the expression for this probability being problem specific.

In practice this means that Metropolis-Hastings makes use of a proposal distribution $Q(x, y)$ which proposes a move from x to y. Whether to accept this move or not, is then decided by utilizing an acceptance ratio $A(x, y)$. This means that the next state given the present, is independent of the past, which is the defining property of a Markov Chain.

In order for the Metropolis-Hastings algorithm to work, a probability function proportional to the distribution of interest must be known. This is good news, as this is exactly what Bayes' Theorem provides as seen in (3.1)

$$P(\theta|X) \propto P(X|\theta)P(\theta)$$

As such it is only necessary to know $P(X|\theta)P(\theta)$, in order to determine the posterior using Metropolis-Hastings. This has to do with the acceptance ratio, which will be covered in the proof of the algorithm.

Once the distribution of the Markov Chain has converged, the samples are then the states the Markov Chain visits, which can be seen as a sampling from the distribution, thus making the Metropolis-Hastings algorithm a use of the Markov Chain Monte Carlo method.

3.2.1.1 Proof of Metropolis-Hastings Convergence

As mentioned the idea behind Metropolis-Hastings is to create a Markov Chain such that its probability distribution converges towards the desired distribution function. In order to do this, the concept of detailed balance is utilized. Detailed balance ensures that if the following condition is satisfied

$$\boldsymbol{\pi}(x)P(x, y) = \boldsymbol{\pi}(y)P(y, x), \tag{3.2}$$

where $P(x, y) = \mathbb{P}(X_{t+1} = y|X_t = x)$. Then $\boldsymbol{\pi}$ is the stationary distribution for the Markov Chain defined by P. For the discrete case with state space $\mathbb{N}$, this is easily proven by using the property of discrete transition matrices defined in (2.2)

$$\begin{aligned}\sum_{y=0}^{\infty} \boldsymbol{\pi}(x)P(x, y) = \sum_{y=0}^{\infty} \boldsymbol{\pi}(y)P(y, x) \quad &\Leftrightarrow \\ \boldsymbol{\pi}(x) = \sum_{y=0}^{\infty} \boldsymbol{\pi}(y)P(y, x).&\end{aligned}$$

This is the definition of a stationary probability distribution seen in (2.7), and thus $\boldsymbol{\pi}$ is a stationary distribution.

This indicates that the distribution of the Markov Chain defined by the transition matrix P, will converge towards the stationary distribution $\boldsymbol{\pi}$. With the existence of a Markov Chain described by P being ensured by Kolmogorov's Consistency Criterion [34].

Since the target distribution, $\boldsymbol{\pi}$, is known, it is simply a question of determining a transition distribution such that (3.2) is satisfied. This is the motivation behind the use of a proposal and acceptance distribution in the Metropolis-Hastings algorithm.

Let S be the domain of the distribution of interest, and $x \in S$ be the current state. Then, as mentioned, the Metropolis-Hastings algorithm has two parts that ensures it works. First there is the proposal distribution, which is used to move around in S, with a move from x to y denoted by $Q(x, y)$. Second there is the acceptance distribution, which determines the probability of accepting the move. This is needed in order to not have the samples simply be a random walk according to the proposal distribution, and also ensure that the Markov Chain does not only move towards the most probable

place in the distribution. This acceptance probability is defined to be

$$A(x,y) = \min\left(1, \frac{Q(y,x)\boldsymbol{\pi}(y)}{Q(x,y)\boldsymbol{\pi}(x)}\right), \tag{3.3}$$

where $\boldsymbol{\pi}$ is the desired stationary distribution.

Having defined the probability of accepting the move, the transition distribution utilized by Metropolis-Hastings can now be defined

$$P(x,y) = Q(x,y)A(x,y) = Q(x,y)\min\left(1, \frac{Q(y,x)\boldsymbol{\pi}(y)}{Q(x,y)\boldsymbol{\pi}(x)}\right). \tag{3.4}$$

Inserting (3.4) into (3.2), it is seen that the detailed balance criteria is satisfied.

$$\begin{aligned}
\boldsymbol{\pi}(x)P(x,y) &= \boldsymbol{\pi}(x)Q(x,y)\min\left(1, \frac{Q(y,x)\boldsymbol{\pi}(y)}{Q(x,y)\boldsymbol{\pi}(x)}\right) \\
&= \boldsymbol{\pi}(x)\min\left(Q(x,y), \frac{Q(y,x)\boldsymbol{\pi}(y)}{\boldsymbol{\pi}(x)}\right) \\
&= \boldsymbol{\pi}(x)\frac{1}{\boldsymbol{\pi}(x)}\min\left(Q(x,y)\boldsymbol{\pi}(x), Q(y,x)\boldsymbol{\pi}(y)\right) \\
&= \boldsymbol{\pi}(y)\frac{1}{\boldsymbol{\pi}(y)}\min\left(Q(y,x)\boldsymbol{\pi}(y), Q(x,y)\boldsymbol{\pi}(x)\right) \\
&= \boldsymbol{\pi}(y)\min\left(Q(y,x), \frac{Q(x,y)\boldsymbol{\pi}(x)}{\boldsymbol{\pi}(y)}\right) \\
&= \boldsymbol{\pi}(y)Q(y,x)\min\left(1, \frac{Q(x,y)\boldsymbol{\pi}(x)}{Q(y,x)\boldsymbol{\pi}(y)}\right) \\
&= \boldsymbol{\pi}(y)P(y,x).
\end{aligned}$$

From this it is seen that the acceptance probability is the reason the distribution of the Markov Chain, which is sampled from using the Metropolis-Hastings algorithm, converges towards the proper distribution. Therefore it is interesting to examine why $A(x,y)$ is chosen as it is.

First (3.2) is rewritten:

$$\frac{P(x,y)}{P(y,x)} = \frac{\boldsymbol{\pi}(y)}{\boldsymbol{\pi}(x)},$$

Then (3.4) is inserted

$$\begin{aligned}
\frac{Q(x,y)A(x,y)}{Q(y,x)A(y,x)} &= \frac{\boldsymbol{\pi}(y)}{\boldsymbol{\pi}(x)} \qquad \Leftrightarrow \\
\frac{A(x,y)}{A(y,x)} &= \frac{Q(y,x)\boldsymbol{\pi}(y)}{Q(x,y)\boldsymbol{\pi}(x)}.
\end{aligned} \tag{3.5}$$

Now it is simply a question of determining an acceptance criteria which satisfies (3.5). One suitable choice is (3.3), and with that choice, detailed balance is achieved.

The way $A(x,y)$ is defined, is also why it is only necessary to know a distribution proportional to the one of interest. As the acceptance probability is calculated using a ratio of the target distribution, any constant factor, such as the normalizing factor, simply cancels out. Let X be the observed data, and x, y states in the sample space, with $P(x|X)$ being the desired posterior distribution, then for the problem at hand

$$\frac{\boldsymbol{\pi}(y)}{\boldsymbol{\pi}(x)} = \frac{P(y|X)}{P(x|X)} = \frac{\frac{P(X|y)P(y)}{P(X)}}{\frac{P(X|x)P(x)}{P(X)}} = \frac{P(X|y)P(y)}{P(X|x)P(x)},$$

thus eliminating the need to determine the normalizing factor $P(X)$.

It is worth noting that the proposal distribution, $Q(x, y)$ does not necessarily have to be symmetric, but in the cases where it is, the acceptance probability reduces to

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

In fact, this was the original acceptance probability proposed in the original Metropolis algorithm, therefore the original algorithm had the constraint that the proposal distribution needed to be symmetric. Hastings later generalized this to account for non-symmetric proposal distributions, thus creating the Metropolis-Hastings algorithm.

3.2.1.2 Practical Example with Analytical Comparison

Having proved why the Metropolis-Hastings algorithm works, it can now be tested on a case where the analytical solution is known. This can be done by looking at an example where the distributions are all Gaussian as these are self-conjugate. A simple example of this, can be seen by determining the distribution describing the students at DTU's mean height.

Assume the mean, μ, of the heights follows a normal distribution. Then let X be the heights of a new class of students. Given this information, the posterior distribution of μ, $P(\mu|X)$, has to be determined. This is known as Bayesian Inference, as Bayes' Theorem is an essential tool for doing this. From Bayes' Theorem the following proportionality is given

$$f(\mu|X) \propto f(X|\mu)f(\mu). \tag{3.6}$$

Here the normalization constant is omitted, as this is not needed for the Metropolis-Hastings algorithm to work.

Now say historic records show $\mu \sim \mathcal{N}(\mu_p, \sigma_p^2)$ over the past years, giving the prior distribution for μ

$$f(\mu) = \frac{1}{\sqrt{2\pi\sigma_p^2}} \exp\left\{-\frac{(\mu - \mu_p)^2}{2\sigma_p^2}\right\}. \tag{3.7}$$

Now the heights of N new students are observed and stored in X. Assuming that they are normally distributed, the likelihood is then given by

$$f(X|\mu) = \prod_{n=0}^{N} \frac{1}{\sqrt{2\pi\sigma_{obs}^2}} \exp\left\{-\frac{(x_n - \mu)^2}{2\sigma_{obs}^2}\right\}. \tag{3.8}$$

Now (3.7) and (3.8) can be inserted in (3.6)

$$\begin{aligned}
f(\mu|X) &\propto f(X|\mu)f(\mu) \\
&= \prod_{n=0}^{N}\left(\frac{1}{\sqrt{2\pi\sigma_{obs}^2}} \exp\left\{-\frac{(x_n - \mu)^2}{2\sigma_{obs}^2}\right\}\right) \cdot \frac{1}{\sqrt{2\pi\sigma_p^2}} \exp\left\{-\frac{(\mu - \mu_p)^2}{2\sigma_p^2}\right\} \\
&= \left(\frac{1}{\sqrt{2\pi\sigma_{obs}^2}}\right)^N \cdot \exp\left\{-\frac{\sum_{n=0}^{N}(x_n - \mu)^2}{2\sigma_{obs}^2}\right\} \cdot \frac{1}{\sqrt{2\pi\sigma_p^2}} \cdot \exp\left\{-\frac{(\mu - \mu_p)^2}{2\sigma_p^2}\right\} \\
&= \left(\frac{1}{\sqrt{2\pi\sigma_{obs}^2}}\right)^N \cdot \frac{1}{\sqrt{2\pi\sigma_p^2}} \cdot \exp\left\{-\frac{\sum_{n=0}^{N}(x_n - \mu)^2}{2\sigma_{obs}^2} - \frac{(\mu - \mu_p)^2}{2\sigma_p^2}\right\}.
\end{aligned} \tag{3.9}$$

As the requirement is simply that the distribution is proportional, and the variable of interest is μ, the normalizing terms in front of the exponential, can be disregarded. This results in the following proportionality

$$f(\mu|X) \propto \exp\left\{-\frac{\sum_{n=0}^{N}(x_n-\mu)^2}{2\sigma_{obs}^2} - \frac{(\mu-\mu_p)^2}{2\sigma_p^2}\right\}. \tag{3.10}$$

Here it should be noted that the right hand side is what will be used as the desired stationary distribution for the Metropolis-Hastings algorithm, $\boldsymbol{\pi}(\mu) = \exp\left\{-\frac{\sum_{n=0}^{N}(x_n-\mu)^2}{2\sigma_{obs}^2} - \frac{(\mu-\mu_p)^2}{2\sigma_p^2}\right\}$, as this can be evaluated at μ, and is proportional to the distribution of interest.

Examining the exponential function, this can be rewritten to reveal that it is a normal distribution, due to the fact that Gaussian distributions are self-conjugate. To simplify the calculations, any term not including μ is seen as a normalization constant, and thus discarded.

$$\begin{aligned}
\exp\left\{-\frac{\sum_{n=0}^{N}(x_n-\mu)^2}{2\sigma_{obs}^2} - \frac{(\mu-\mu_p)^2}{2\sigma_p^2}\right\} &= \exp\left\{-\frac{1}{2}\left(\frac{\sum_{n=0}^{N}(x_n-\mu)^2}{\sigma_{obs}^2} + \frac{(\mu-\mu_p)^2}{\sigma_p^2}\right)\right\} \\
&= \exp\left\{-\frac{1}{2}\left(\frac{\sum_{n=0}^{N}x_n^2 - 2\bar{x}N\mu + N\mu^2}{\sigma_{obs}^2} + \frac{(\mu^2 - 2\mu\mu_p + \mu_p^2}{\sigma_p^2}\right)\right\} \\
&\propto \exp\left\{-\frac{1}{2}\left(\frac{-2\bar{x}N\mu + N\mu^2}{\sigma_{obs}^2} + \frac{\mu^2 - 2\mu\mu_p}{\sigma_p^2}\right)\right\} \\
&= \exp\left\{-\frac{1}{2}\left(\frac{N\mu^2\sigma_p^2 + \mu^2\sigma_{obs}^2 - 2\bar{x}N\mu\sigma_p^2 - 2\mu\mu_p\sigma_{obs}^2}{\sigma_{obs}^2\sigma_p^2}\right)\right\} \\
&= \exp\left\{-\frac{1}{2}\left(\frac{\mu^2(N\sigma_p^2 + \sigma_{obs}^2) - 2\mu(\bar{x}N\sigma_p^2 + \mu_p\sigma_{obs}^2)}{\sigma_{obs}^2\sigma_p^2}\right)\right\} \\
&= \exp\left\{-\frac{1}{2}\left(\frac{\mu^2 - 2\mu\frac{\bar{x}N\sigma_p^2 + \mu_p\sigma_{obs}^2}{N\sigma_p^2 + \sigma_{obs}^2}}{\frac{\sigma_{obs}^2\sigma_p^2}{N\sigma_p^2 + \sigma_{obs}^2}}\right)\right\} \\
&\propto \exp\left\{-\frac{1}{2}\left(\frac{\mu^2 - 2\mu\frac{\bar{x}N\sigma_p^2 + \mu_p\sigma_{obs}^2}{N\sigma_p^2 + \sigma_{obs}^2} + \left(\frac{\bar{x}N\sigma_p^2 + \mu_p\sigma_{obs}^2}{N\sigma_p^2 + \sigma_{obs}^2}\right)^2}{\frac{\sigma_{obs}^2\sigma_p^2}{N\sigma_p^2 + \sigma_{obs}^2}}\right)\right\} \\
&= \exp\left\{-\frac{1}{2}\left(\frac{\left(\mu - \frac{\bar{x}N\sigma_p^2 + \mu_p\sigma_{obs}^2}{N\sigma_p^2 + \sigma_{obs}^2}\right)^2}{\frac{\sigma_{obs}^2\sigma_p^2}{N\sigma_p^2 + \sigma_{obs}^2}}\right)\right\}.
\end{aligned}$$

From this it is seen that the density function $f(\mu|X)$ describes a random variable which follows $\mathcal{N}\left(\frac{\bar{x}N\sigma_p^2 + \mu_p\sigma_{obs}^2}{N\sigma_p^2 + \sigma_{obs}^2}, \frac{\sigma_{obs}^2\sigma_p^2}{N\sigma_p^2 + \sigma_{obs}^2}\right)$.

Rewriting the expression for the mean gives an insight into how the posterior mean can be interpreted

$$\begin{aligned}
\mu_{posterior} = \frac{\bar{x}N\sigma_p^2 + \mu_p\sigma_{obs}^2}{N\sigma_p^2 + \sigma_{obs}^2} &= \frac{N\sigma_p^2}{N\sigma_p^2 + \sigma_{obs}^2}\bar{x} + \frac{\sigma_{obs}^2}{N\sigma_p^2 + \sigma_{obs}^2}\mu_p \\
&= \frac{\frac{N}{\sigma_{obs}^2}}{\frac{N}{\sigma_{obs}^2} + \frac{1}{\sigma_p^2}}\bar{x} + \frac{\frac{1}{\sigma_p^2}}{\frac{N}{\sigma_{obs}^2} + \frac{1}{\sigma_p^2}}\mu_p.
\end{aligned}$$

Here it is seen that the posterior mean is a weighted average of the prior mean μ_p, and the mean of the observed data, $\bar{x}$. With the weight on μ_p being inversely proportional with the prior variance, σ_p^2, and the weight on $\bar{x}$ being proportional with $\frac{N}{\sigma_{obs}^2}$. Intuitively this makes a lot of sense, as this indicates

a higher belief in the data if there is a lot of data, or if the variance of the data is small. Likewise it reflects how a higher belief in the prior, corresponding to a low prior variance σ_p^2, pulls the posterior mean more towards the prior mean.

Having found the analytical distribution of the posterior, this can then be compared to the distribution found by using the Metropolis-Hastings algorithm.

Let X be an observation of the heights of $N = 30$ students, with an observed mean $\bar{x} = 184.9$ and standard deviation of $\sigma_{obs} = 12.43$. Further let the prior distribution of $\mu \sim \mathcal{N}(170.0, 10^2)$. The analytical posterior is thus

$$f(\mu|X) \sim \mathcal{N}\left(184.2, 2.21^2\right).$$

Here the posterior mean is very close to the observed mean. This is because the prior standard deviation, $\sigma_p = 10$, was high, indicating a low belief in the prior mean. The observed data also had a high standard deviation $\sigma_{obs} = 12.43$, however because the weight on $\bar{x}$ also takes into account the data $N = 30$, the posterior mean gets weighted more towards the observed mean.

Running Markov Chain Monte Carlo using the Metropolis-Hastings algorithm with $9,500$ samples after a burn-in of 500 and an initialization of $\mu_0 = 170$, the distribution found can be seen on Figure 3.3, alongside the analytical distribution $f(\mu|X)$.

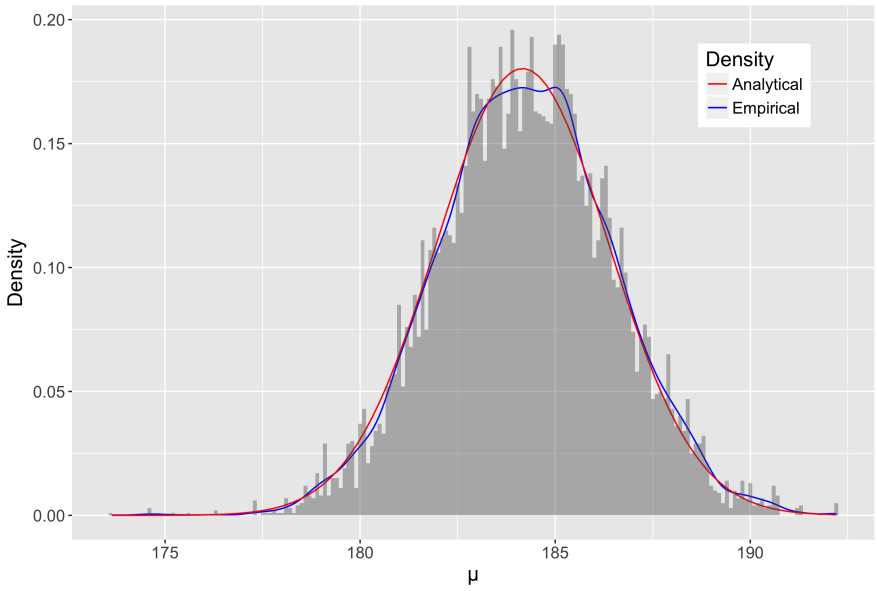


Figure 3.3: Sample and analytical density describing μ, for $9,500$ samples

From this it is seen that the estimated distribution is very close to the analytical distribution, indicating that the Metropolis-Hastings method worked.

How this problem was modelled will be described in Section 4.2, with the application of Metropolis-Hastings being described in Chapter 7.

Chapter 4

Bayesian Networks

In Chapter 3, the idea of probabilistic models were introduced. These are models where the different parameters are random variables, reflecting the stochastic nature of life. As this can seem quite abstract, it often helps to visualize the model at hand. One tool for doing this, is Bayesian Networks (BNs).

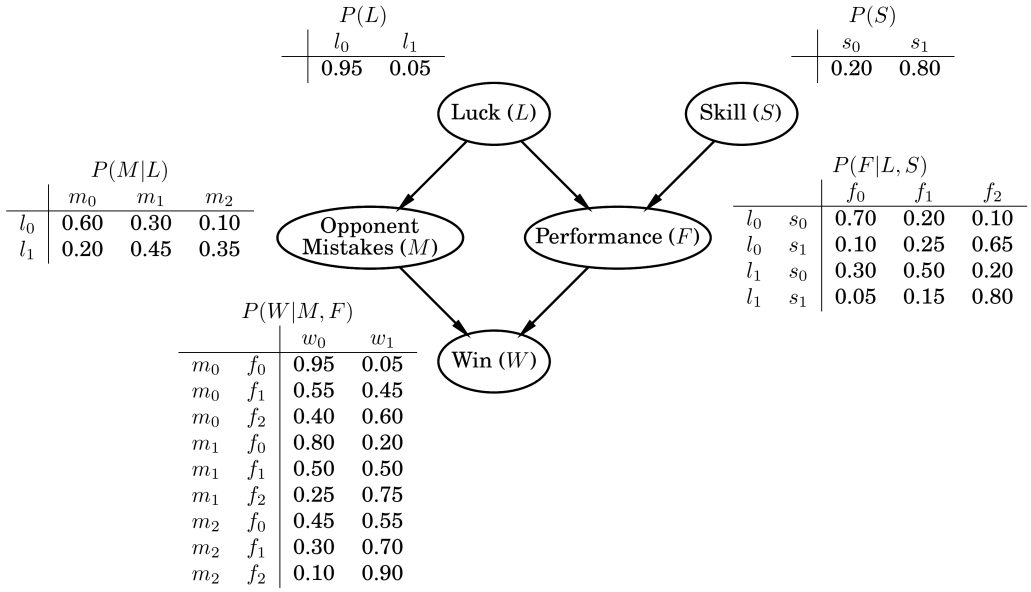
In essence a Bayesian Network is a collection of random variables organized in a directed acyclic graph (DAG), where every node represents a random variable, and every edge indicates a conditional dependency between two nodes. This allows for visual inference with regards to conditional dependencies and independencies, which further gives insight into how the factorization of the joint distribution can be made.

To work with Bayesian Networks, it is therefore necessary to know the conditional distribution functions for the random variables. However, once these have been determined, either by extraction from data, or from prior knowledge, it is possible to do Bayesian Inference on the model at hand.

4.1 Sampling from a Bayesian Network

Let Figure 4.1 be a Bayesian Network that describes the elements of some game in the discrete case, with a conditional probability table describing the distribution of each random variable.

The assumption is that a player's amount of *Luck* influences *Performance* and amount of *Opponent Mistakes*; that the amount of *Skill* influences *Performance*; and that a player's *Performance* and the number of *Opponent Mistakes* both influence the probability of getting a *Win*. Further let *Luck*, *Skill* and *Win* be binary, with 0 indicating a lack of the attribute, and let low, medium and high, denoted by 0, 1 and 2 respectively, be the different levels of *Opponent Mistakes* and *Performance*.



$P(L)$

l_0	l_1
0.95	0.05

$P(S)$

s_0	s_1
0.20	0.80

$P(M|L)$

	m_0	m_1	m_2
l_0	0.60	0.30	0.10
l_1	0.20	0.45	0.35

$P(F|L,S)$

		f_0	f_1	f_2
l_0	s_0	0.70	0.20	0.10
l_0	s_1	0.10	0.25	0.65
l_1	s_0	0.30	0.50	0.20
l_1	s_1	0.05	0.15	0.80

$P(W|M,F)$

		w_0	w_1
m_0	f_0	0.95	0.05
m_0	f_1	0.55	0.45
m_0	f_2	0.40	0.60
m_1	f_0	0.80	0.20
m_1	f_1	0.50	0.50
m_1	f_2	0.25	0.75
m_2	f_0	0.45	0.55
m_2	f_1	0.30	0.70
m_2	f_2	0.10	0.90

Figure 4.1: Simple Bayesian Network describing a competition

Having defined the Bayesian Network visually, the joint distribution can easily be factorized, as the expression can be seen directly from the structure of the BN.

$$P(L,S,M,F,W) = P(L)P(S)P(M|L)P(F|L,S)P(W|M,F) \tag{4.1}$$

Further, as this is a discrete BN, and the conditional probability distributions (CPDs) are known, forward sampling can be applied.

Forward sampling works by initially sampling from an appropriate unconditioned distribution corresponding to the parent nodes, which in this case are the nodes L and S. In practice for L this would mean flipping a coin with probability 0.95 of getting heads, and letting $L = l_0$ if the coin flip turns up heads. Likewise a value for S is sampled.

Once L and S have been sampled, the appropriate distribution for M and F can be found using the CPDs, as it is simply a matter of looking at the row corresponding to the sampled values of L and S. Using this approach, the entire BN can be sampled, by moving through it in a topological order. This is illustrated on Figure 4.2, where initially L is sampled, as denoted by L being grayed out. Then at the next time-step S is sampled, etc. Once the entire BN has been sampled, the algorithm starts over, and finds a new sample for L, S, M, etc.

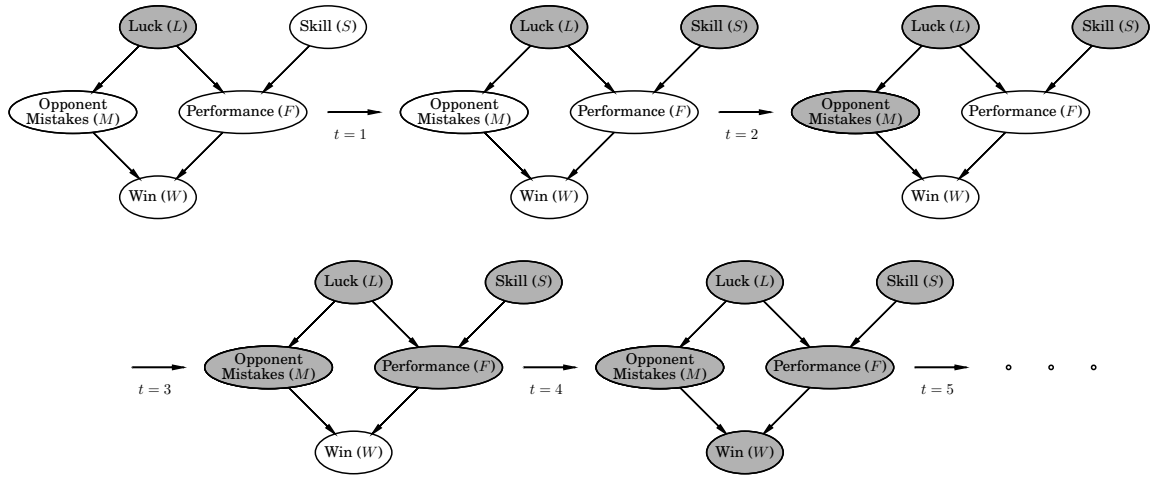


Figure 4.2: First 5 steps of forward sampling in a discrete BN

The problem with using forward sampling for inference is that it only works when it is possible to sample values from a distribution which satisfies the constraints set up by the evidence. For discrete random variables, there is a general approach for generating samples from a CPD once it is known, but for continuous CPDs the problem is a lot more complex [26, p. 489].

Although the factorization seen in (4.1) is the same for the continuous distributions, it is not certain that multiplying the factors yields a distribution from which sampling can easily be done. In fact, even if it is possible to sample from each of the continuous distributions given their parents in the BN, then using forward sampling for inference would still not be applicable generally, as evidenced by the following example.

Let $X \to Y$ be a BN describing continuous random variables X and Y. If there is evidence that $Y = y$, then using forward sampling for inference would generate a sample set (x_t, y_t), and reject the samples where $y_t \neq y$. However, because Y is a continuous variable, $\mathbb{P}(y_t = y) = 0$, resulting in all of the samples being rejected [26, p. 643].

Therefore it is necessary to use a different approach, such as the Metropolis-Hastings MCMC method described in Chapter 3.

4.2 Bayesian Networks and Metropolis-Hastings algorithm

Remember from Chapter 3 that the Metropolis-Hastings algorithm can be used when a distribution $\boldsymbol{\pi}$ is known, even if its normalization factor is not. What this translates to in terms of a BN, is the ability to calculate the likelihood of a realization for each random variable given its parents, and evaluate the prior at a point for the parent nodes. This corresponds to writing out and calculating the factorized joint distribution, without normalizing it.

For an application of this, remember the example from Section 3.2.1.2, where the goal was to determine the distribution describing the mean heights of the students at DTU, $f(\mu|X)$. This model can now be built as a BN, as seen on Figure 4.3.

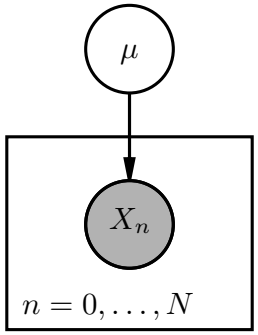


Figure 4.3: Student Heights Example Using BN

From the BN, it is seen that X_n is an observed random variable, which is conditioned on μ. It was assumed that $\mu \sim \mathcal{N}(\mu_p, \sigma_p^2)$ with $f(X_n|\mu)$ being the density of a random variable distributed according to $\mathcal{N}(\mu_{obs}, \sigma_{obs}^2)$. Therefore the factorization of the BN yields

$$f(X, \mu) = f(X|\mu)f(\mu) = \prod_{n=0}^{N} \left(\frac{1}{\sqrt{2\pi\sigma_{obs}^2}} \exp\left\{ -\frac{(x_n - \mu)^2}{2\sigma_{obs}^2} \right\} \right) \cdot \frac{1}{\sqrt{2\pi\sigma_p^2}} \exp\left\{ -\frac{(\mu - \mu_p)^2}{2\sigma_p^2} \right\},$$

which is the same as seen in (3.9), where the example was worked through analytically. Thus it is proportional to the target distribution $f(\mu|X)$, and therefore the factorization of the BN can be used as the desired stationary distribution $\boldsymbol{\pi}(\mu)$ in the Metropolis-Hastings algorithm. What this means, is that an expression for the desired stationary distribution can be found easily, once the problem has been modelled using a BN, as it is simply a matter of writing out the factorized joint distribution as defined by the BN.

4.3 Modelling Using Bayesian Networks

Having determined that it is possible to sample from the conditional distributions of a BN using the Metropolis-Hastings algorithm, it is interesting to look at how Bayesian modeling is done. One intuitive way of doing Bayesian modeling is to reverse the process, and think about how the observed data might be generated. This can be done by asking a series of questions:

1. Which distribution best describes the data?
2. What parameters are needed for the above chosen distribution?
3. Are these parameters known?
4. Is there prior belief regarding these parameters?

This process continues until the parents in the Bayesian Network, and thus prior distributions, have been found.

4.3.1 Linear Regression with Bayesian Networks

Some of the first models that are taught in statistics, are often linear regression models. These are simply models where the output, Y, is modelled as a linear combination of the input, X. Assuming

that X is one-dimensional, and that Y has been observed with some noise ϵ, this can be written as

$$Y = \alpha X + \beta + \epsilon. \tag{4.2}$$

This is a well known model where there are methods, such as Least Squares and Maximum Likelihood, that can be used to determine the parameters α and β [28]. However, using these methods only a single value for each of the parameters is found, and there is no possibility of adding prior knowledge to the model. Therefore it can be of interest to model it using Bayesian modeling, as this yields a distribution for each of the parameters, while making it possible to add knowledge regarding the parameters, as well as the noise. Rewriting (4.2) to be a probabilistic model, it then becomes

$$Y \sim \mathcal{N}(\alpha X + \beta, \sigma^2). \tag{4.3}$$

Here the first steps of probabilistic modeling has been done, as it is assumed that Y is best described by a normal distribution, with independent variable X and parameters α, β and σ. The next step is then to determine if these parameters follow a distribution themselves, or if they should be fixed. As α, β and σ are the parameters of interest, these should all be modelled with a suitable prior distribution.

In order for the prior distributions to be suitable, it is necessary to account for the range of the parameters they model. In fact, ensuring this, is the most important modeling part, as the estimated distribution will converge towards the target distributions, as long as the prior distributions maps to the proper range. As σ is the standard deviation of a normal distribution, this parameter cannot be negative. Therefore the prior distribution for σ should only map to $\mathbb{R}_+$. The slope, α, and intercept, β, are defined on $\mathbb{R}$, and therefore that is the range which the prior distributions for these parameters should map to. As there is no prior belief in what influences the parameters of the distributions that describe α, β and σ, the modeling stops here, and as such the parent nodes have been found.

As the realizations of X have been observed, this parameter is fixed. In order to model this, it is necessary to extend the capabilities of BNs. So far they have only consisted of random variables, of which some might be observed. However, now it is necessary to account for observations that are not stochastic, but still influence one of the random variables. This is done by introducing fixed observations.

Let $(\mathbf{x}, Y)$ be an observed pair of values, where $\mathbf{x} = \{x_0, x_1, \ldots, x_N\}$ are realizations of the random variable X, also known as fixed observations and $Y = \{Y_0, Y_1, \ldots, Y_N\}$ is a random variable. Then the probabilistic linear regression model from (4.3) can be visualized using a BN, as seen on Figure 4.4.

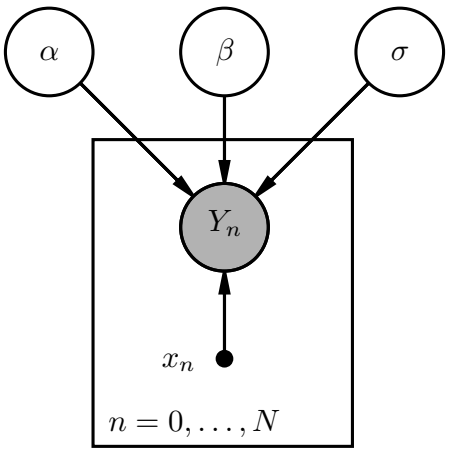


Figure 4.4: Linear Regression Using BN

How this is modelled using probabilistic programming, including the choice of prior distributions, will be shown in Chapter 7, once the implementation of Metropolis-Hastings and the syntax has been described in Chapter 6.

4.3.2 Poisson Changepoint with Bayesian Networks

Another interesting probabilistic problem, is determining a changepoint between two rates for a Poisson process. It can be assumed that the amount of accidents at a worksite follows a Poisson process. Therefore being able to determine if the rate of the process has changed, allows for inference with regards to the safety of the workers. This can be used to determine whether a safety measure implemented to reduce the number of accidents worked, as that would result in a decrease in the rate of the Poisson process describing the amount of accidents.

As this example deals with data from a Poisson process, the modeling part consists of choosing the appropriate parameters for the Poisson distribution, as well as the prior distributions describing these parameters.

The idea is to determine a switch between two rates in a Poisson process, which gives rise to the following three parameters; the first rate of the Poisson process λ_1, the second rate λ_2 and the time at which the rate changes τ, also known as the changepoint.

Like σ in the linear regression example, both λ_1 and λ_2 must be positive, as these are the rates of a Poisson distribution. Therefore it is necessary to choose a prior distribution that maps to $\mathbb{R}_+$. For τ it is important to ensure that the proposed values matches the indexing of the observations. This means that if the observations are indexed using a discrete time-step, then the prior distribution for τ has to map to $\mathbb{N}$.

Modelling this using a BN, the three parameters λ_1, λ_2 and τ become the parent nodes with corresponding prior distributions, all of which influence the observed data X_n. Where X_n is the amount of events, at the n^{th} time-step. For the worksite example, this would be the amount of accidents at time n. A visualization of this can be seen on Figure 4.5.

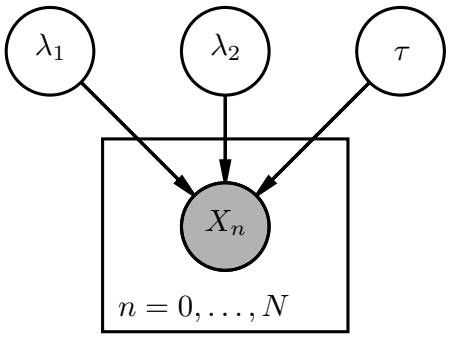


Figure 4.5: Model of Mining Disasters Using BN

Like the linear regression model, how this is modelled using probabilistic programming, as well as the choice of prior distributions, will be shown in Chapter 7.

Chapter 5

Pure Functional Programming

This work builds on that done with Haskell in *Probabilistic Programming with Monads* [42]. This paper aims to port the implementation to JavaScript by exploiting the similarity between do-notation in Haskell and generators in JavaScript. It first explores how a probabilistic programming language could be built in Haskell. Using similar techniques a proof of concept implementation is then done in JavaScript.

It will further expand on the original paper with the introduction of a Free Monad to allow for inference with the Metropolis-Hastings algorithm proposed by Wingate [50] which has an implementation in a wide range of popular language due to its simplicity. It will do so through the following structure of the chapter: 1) Introduce Haskell's syntax and type system, 2) Introduce the necessary Monoids and Monads, 3) Provide a mapping from Haskell to JavaScript, 4) Model probability distributions as a Monad

Part of the objective is to explore the use of pure functional programming design patterns for probabilistic programming. What this chapter aims to do is give the necessary background to understand how these are used. No former background in functional programming is expected.

5.1 Haskell

This section is intended to provide a fundamental understanding of Haskell. This will be used later to build up an embedded probabilistic programming language using monads. Haskell was specifically chosen for its concise notation and strong type system which provides a good environment for understanding how category theory concepts like monads operates.

The chapter is divided into three sections that provide the necessary context for understanding Monads: 1) Types, 2) Functions, and 3) Type classes.

5.1.1 Functional Programming

Functional programming is a programming paradigm where the program is build up by means of function applications to arguments [18, p. 2]. To illustrate the differences from imperative programming we look at the how the sum of integers between 1 and $n > 1$ are computed in the two paradigms[1]. In an imperative language this would usually be done by keeping some state and mutating this to update the score as expressed in the following pseudo code.

```
1: count := 0
2: total := 0
3: repeat
4:     count := count + 1
5:     total := total + count
6: until count = n
```

In Haskell this computation can be expressed by two function applications, `[x..y]` to generate the list of integers between x and y, and `sum` used to calculate the sum of a list:

```
sum [1..n]
```

This program computes the sum by applying a sequence of functions. These are *pure functions* that given an input, always return the same output. We say that they do not have any *side effects* since they do not change the state outside the function scope.

Let us look at how `sum` might be defined to compare the two approaches properly. It can be defined by two *equations* in Haskell:

```
sum [] = 0
sum (x:xs) = x + sum xs
```

This definition is recursive with the first equation expressing the base case that the sum of an empty list is 0.[2] The second equation expresses that the sum of a non-empty list is calculated by adding the head of the list and `sum` applied to the rest of the list.

This differs from the imperative implementation by declaring how the computation is structured instead of describing the procedure. This function will be used as a running example throughout the introduction to Haskell where the syntax will be covered in more details. In general this guide will focus on examples to provide an intuition of the syntax beyond a formal introduction. For a formal definition we refer to the Haskell Language Report [29].

5.1.2 Notation

When Haskell code is provided lines starting with `>>>` will refer to a line executed in interactive mode[3] which interacts with IO to display the output e.g.:

```
f n = n + 1
>>> f 1
```

[1] The `sum` example is extracted from [18, p. 2]

[2] Note that 0 is the identity in that $0 + x = x$ and $x + 0 = x$ for any integer x. This will later be formalised in Section 5.1.6

[3] Interactive mode is possible with e.g. the GCHi `https://downloads.haskell.org/~ghc/5.04/docs/html/users_guide/ghci.html`

```
2
```

The consecutive line to >>> will be the output produced.

5.1.3 Types

Haskell consist of *expressions* (like the function call `f 1`) that yields *values* (in this case `2`). Every value has a *type* associated with it which makes it possible to group similar values into a category. An example of this is the type `Boolean` which contains the two values `False` and `True`. Types can be used to describe functions as well e.g. the function `not` that negates a boolean is of type `Boolean -> Boolean` which is the type containing all function mapping a boolean to a boolean. The notation `v :: T` is used to note the value v is in type T, which reads, *v has type T* [18, p. 17]. Similarly the same notation can be used to express that the yielded value of an expression is expected to have a specific type. This notation can be used to explicitly specify a type. Most of the time this is not needed however due to Haskells *type inference*. The type checking happens at compile-time (which makes it a *statically typed language*) and ensures that the program is *type safe*. Expressions such as `not 1` would not type check since an argument of type `Int` is passed to a function requiring `Boolean`.

Every type associated with a value is called a *concrete type*. These are best seen in contrast to *type constructors* like the list type constructor `[]` that takes a concrete type to produce another concrete type.

5.1.3.1 Polymorphism

Polymorphic types are types that are universally quantified over all types. This is expressed by including a *type variable* (recognized by an initial lowercase character) in the type expression which can be substituted by any type. As an example this could be the type signature `[a] -> a` expressing a mapping from a list of some concrete type to a value of similar type. A function with this type signature would be applicable on e.g. a value of type `[Int]` and return a value of type `Int`.

5.1.3.2 User-Defined Types

Haskell provides a way to define custom types through the `data` declaration. If we were to recreate the `Boolean` type:

```
data Boolean = True | False
```

`Boolean` is called a *type constructor* and `True` and `False` are called *value constructors*. Neither types of constructors take any arguments in this case. It is also possible to have polymorphic data types where the type constructor takes arguments:

```
data Point a = Point a a
```

The type constructor takes a concrete type and returns a concrete type which can also be stated by its *kind* `* -> *` where `*` represents a concrete type. By creating a point with the value constructor, say `Point 1 1`, the concrete type `Point Int` is inferred. Even though the two constructors have the same name, note the difference between applying a *data constructor* to yield a *value* and yielding a *type* by applying a *type constructor*.

Another way to create a type is *newtype* used in the following example:

```
newtype Wrap a = Wrap { getWrap :: () -> a }
```

This creates a value constructor `Wrap` taking a function of type `() -> a` as an argument. Additionally a function `getWrap` is created that "extracts" the wrapped value for any type `Wrap a` where the wrapped value in this case is a function returning type `a`. There is much more to both `newtype` and `data` and the relationship between them but this is sufficient for our use. In Section 5.1.8.4 this is used to define the *state monad*.

5.1.3.3 Type Synonyms

A type synonym is a convenient way to refer to a type using an alias.

```
type Count = Int
type Points a = [Point a]
```

As seen it is possible for the type to be polymorphic.

5.1.4 Functions

Functions are *first-class citizens* in Haskell meaning they can be passed around like any other value. We will explore their properties in this section.

5.1.4.1 Lambda Abstractions

So far functions has been defined using equations. Lambda abstractions provides a way to define a function *anonymously*[29, p. 18]. The previously defined f can be written using a lambda abstraction:

```
f = \x -> x + 1
```

This concept will be useful in understanding *currying*.

5.1.4.2 Curry

A function in Haskell is a mapping from a single argument to a single return value. When a function that takes multiple arguments $x_1, x_2, ..., x_n$ is defined as being *curried* so that it takes only one argument x_1. The return value is then another function taking x_2 as a parameter. We say that the function is *partially applied* if not all arguments have been applied. Let us consider a simple function `add` taking two parameters:

```
add :: Int -> Int -> Int
add x y = x + y
```

The type signature is made explicit to show that arguments are applied one at a time. It can be understood in terms of Lambda Abstractions:

```
add :: Int -> Int -> Int
add = \x -> (\y -> x + y)
```

The signature associates right so that `Int -> Int -> Int` is equivalent to `Int -> (Int -> Int))`.

5.1.4.3 Function Application

Function application in Haskell is denoted by *juxtaposition* e.g. the function f applied to the argument x is expressed by `f x`. This function application operator has the highest precedence 10 so that `f 2 * f 2` applies the function before multiplying. It is possible to override the precedence with parentheses. Additionally two operators exist that can be used to control the application order:

1. The right-associated operator `$` with lowest precedence 0 exists so that all other operation on both sides are evaluated before `$` is applied.
2. The operator `.` used for function composition[29, p. 51].

Effectively, they can be used to avoid parentheses e.g. with `f . f $ 1` that composes the functions and apply them to the argument.

5.1.4.4 Infix Operator

We have already seen examples of *infix operators*, both `+` and `.` are such. They are just functions and can be defined with equations:

```
(++)        :: [a] -> [a] -> [a]
[]     ++ ys =  ys
(x:xs) ++ ys =  x : (xs ++ ys)
```

One can then use this infix operator to concatenate two lists `[1, 2] ++ [2, 3] == [1, 2, 3, 4]`.

An infix operator can be partially applied like any other function and is then called a *section*. Both arguments can be partially applied, for an operator $\oplus$ we have:

```
(x ⊕) = \y -> x ⊕ y
(⊕ y) = \x -> x ⊕ y
(⊕)   = \x -> (\y -> x ⊕ y)
```

The final line defines a way to convert an infix operator to a regular function, `(+) 1 2 == 3`.

5.1.4.5 Pattern Matching

Pattern matching has already been used by the equations defining `sum`. It provides a way to choose between a sequence of results by *matching* against a sequence of syntactic expressions called *patterns*[18, p. 32]. A formal definition of supported patterns will not be provided but will become clear from the use cases throughout the paper. It is more important that the reader can recognize when pattern matching is used. Pattern matching can also be used outside of equation definition using case expressions. The previously defined `sum` can be expressed using this:

```
sum x = case x of
     [] = 0
     (x:xs) = x + sum xs
```

As has already been seen, they are, among other purposes, useful for defining recursive functions in a compact way.

5.1.5 Type Classes

The operator `+` is applicable to not only `Int` but also `Double` (including others). The operator is said to be *overloaded* since a separate implementation is provided for each type for which the operator is defined. This restriction is made precise by the type system with a *class contraint*:

```
(+) :: Num a => a -> a -> a
```

Overloading is also called *ad-hoc polymorphism* whereas the polymorphism covered so far has been *parametric polymorphism*. In Haskell, *type classes* provide structure for implementing ad-hoc polymorphism. A type can be made an *instance* of a type class in which it defines the required overloaded operations for that type class. The syntax will be explained gradually in the following sections on some of the type classes used, *Monoids*, *Functors* and *Monads*.

5.1.6 Monoids

A monoid is defined as a triple $(M, e, \oplus)$, where M is some set, $\oplus$ is an associative operator, and e is an identity element such that $x = e \oplus x = x \oplus e$[24]. This will prove useful for any kind of type used to accumulate by providing a standardized interface. With slight abuse of notation by incorporating Haskell types examples of monoids are: $(Integer, 0, +)$, $([a], [], ++)$. In Haskell a monoid can be expressed as an instance of the type class Monoid.

```
class Monoid a where
    mempty :: a
    mappend :: a -> a -> a
```

This states, that to be an instance of the type class Monoid, the type a should implement two operators with the specified signature. As an example the following will make the list type an instance of Monoid:

```
instance Monoid [a] where
    mempty  = []
    mappend = (++)
```

5.1.7 Functor

Functors provides a way to map a function of type `a -> b` to that of type `f a -> f b`[40] which is given by its type class [29, p. 80]:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Note how the `kind` of `f` is inferred from the fact that it takes an `a` in the argument for the `fmap`-operator. Recall that a parameter has to be a concrete type so both `f a` and `a` are concrete types. It follows that `f` is a type constructor returning a concrete type (`* -> *`). Finally, `a` and `b` are uses of parametric polymorphism and thus allow for any concrete type. This allows `f` to wrap two different types. Making no restriction on what types `f` wraps is called *higher-order polymorphism*[21, p. 6].

The list type is an instance of `Functor` providing an interface for applying a method on every element in the list, effectively mapping a list of one type to a list of another type: `fmap (== 2) [1,2,3] == [False, True, False]`.

5.1.8 Monads

Both monoids and functors have roots in category theory and the same applies to monads. The concept of a monad was introduced into programming in 1989 by Moggi[32] and turned out to be useful in a lot of different cases including I/O, keeping state, and error handling.

We will not describe the origins in category theory and instead introduce it in terms of its type class[29, p. 80]:

```
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    return ::   a               -> m a
```

Monads also have to conform to three laws but these cannot be captured with the Haskell type system. So they will be covered later after a thorough explanation of the type class above. Similarly to the `Functor` type class, it uses *higher-kinded polymorphism*. The interface will now be discussed to understand how it behaves.

Coming from an imperative background it is important to note that `return` does not end function execution. It is simply a function taking a value and putting it in some *context* `m`.

The more interesting function is `>>=` which is called `bind` (or `flatMap` in other languages e.g. Scala). The official Haskell Monad type class does have more operators[29, p. 80] but `>>=` with `return` is sufficient for defining and implementing the rest of the monads interface[48]. It takes a value with context and a function mapping that value to another value with context. In this way it provides a way of chaining computations together sequentially. There will be refered to `>>=` with `bind` and `>>=` interchangeably.

To understand how these function can be applied, a concrete monad will be used.

5.1.8.1 The Maybe Monad

One of the simplest monads is the Maybe monad. It provides a way to avoid `null` pointer errors by short circuiting[49, p. 142] the computations if something returns `null`. The type definition for the maybe monad is provided below.

```
data Maybe a = Just a | Nothing
```

Recall that with this definition we get two value constructors. `bind` is then implemented using pattern matching on these two constructors.

```
instance Monad Maybe where
    (Just x) >>= k = k x
    Nothing  >>= k = Nothing
    return = Just
```

It is clear from this Monad instance that if you `bind` over `Nothing`, by induction any number of additional `bind`s would also return `Nothing`. So the `bind` operator allows us to chain computations, as in the following:

```
>>> Just 1 >>= \x -> Just (x + 1)
Just 2
>>> Nothing >>= \x -> Just (x + 1)
Nothing
```

It simply wraps the value and those values are unwrapped using `bind`.

5.1.8.2 The Three Monad Laws

In addition to being an instance of the `Monad` type class a monad has to satisfy three laws:

```
return a >>= f  = f a
m >>= return    = m
(m >>= f) >>= g = m >>= (\x -> f x >>= g)
```

For each instance these rules should be proved. However, we only use already built-in monads in Haskell for which proofs already exists[29, p. 81]. The rules are listed here nonetheless, since they allow for safe refactoring that we will later use.

5.1.8.3 *Do*-Notation

Haskell has some syntactic sugar called do-notation for working with monads. This is useful for deeply nested `bind` structures. Desugaring the do-notation can be done recursively to get the equivalent `bind` structure:

```
do { a <- f ; m } ≡ f >>= \a -> do m
do { f ; m }      ≡ f >> do { m }
do { m }          ≡ m
```

The operator `>>` has not been mentioned before since it can be implemented in terms of `bind`.

```
m >> k = m >>= \_ -> k
```

It simply ignores the value passed from `m` to the bound function.

Omitting curly brackets and semicolons which were only used for brevity, it follows that these two expressions are equivalent:

```
do
    a <- return 1
    b <- return (a + 2)
```

```
    return (a, b)

return 1 >>= \a ->
    return (a + 2) >>= \b ->
        return (a, b)
```

The usefulness of this syntax will become clear with the introduction of e.g. the state monad. It will be possible to implicitly pass context between monads, which will be the foundation for the domain specific language for probabilistic programming which is the goal.

5.1.8.4 State Monad

The state monad will turn out to be very useful for implementation of the probability monad. It is different from the maybe monad in that it wraps a computation. The computation is a function that manipulates a state of type `s` and returns a result of type `a` along with the modified state. The type is made explicit in the following `newtype` definition:

```
newtype State s a = State { runState :: s -> (a, s) }
```

To understand how it works, let us look at a minimal implementation of it[22, p. 40]:

```
instance Monad (State s) where
    return x = State $ \s -> (x,s)
    (State h) >>= f = State $ \s -> let (a, s') = h s
                                        (State c) = f a
                                    in  g s'
```

First `return` creates a state that leaves the state unchanged but returns the wrapped value.

Note the critical difference between the type constructor and the value constructor in the instance declaration. Where `s` refers to the specific type of the state `h` refers to the function provided in the value constructor. Recalling how `newtype` works is helpful to get a better intuition of how it operates. The following example will be used:

```
let s = return 1 >>= \s -> return (s + 1)
runState s []
```

The only thing `runState` does is unwrap the function that we passed to the value constructor `State`. This could be achieve equivalently with pattern matching as follows: `let (State x) = s in x []`. The function that we unwrap is that produced by the `bind`. This function first runs the function `h` wrapped in `return 1` with the original state. It then runs the function wrapped in the state monad produced by `f`. The procedure might become clearer if `runState` is used instead of pattern matching:

```
pr >>= f = state \$ \ st ->
    let (x, st') = runState pr st -- Running the first processor on st.
    in runState (f x) st'         -- Running the second processor on st'.
```

this is equivalent to the original `bind`.

The state monad is not very useful if the state cannot be modified. To do this wrappers around the value constructor have been created.

```
get = State $ \_ -> (s,s)
put x = State $ \_ -> ((), x)
```

More specialized ones can be created in the case where to state type `s` is known e.g.:

```
pop :: State [a] a
pop = State $ \(x:s) -> (x, s)

push :: a -> State [a] ()
push a = State $ \s -> ((), a:s)
```

It should be noted that the state is not truly modified. The state is still immutable and the functions pure, but the state behaves in the do-notation as if it were modified.

This way of implementing state will prove very useful for a pure implementation of random number generation. There is slightly more to the real implementation of `State` in Haskell but we leave this out for clarity.

5.1.8.5 Lifting

A term which will be used a lot is *lifting*. The simplest form of lifting has already been used with `return` which puts a value of any type into a specific monadic context. Both `fmap` and `bind` also goes under the term lifting where a function is lifted to operate in a certain context.

5.1.8.6 Monad Transformer

Each monad introduced so far has been specialized to do only one thing. For more complex programs it is necessary to combine several monads. It is a type that wraps a monad in another monad effectively providing the behavior of both.

`MaybeT` is an example of this which has the following `newtype` definition:

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
```

Notice that it wraps the type parameter in a `Maybe` monad. This provides a way to terminate early without having the pattern match the output of the monad. The pattern matching is instead done once in the monad instance definition for `MaybeT m`.

Building on the previous state example we can build a monad that terminates when the list is empty (see Appendix A.1).

```
pop :: MaybeT (State [a]) a
pop = do
    s <- lift get
    case s of
        []     -> MaybeT $ return Nothing
        (r:sx) -> do
            lift $ put sx
            return r
```

```
push :: a -> MaybeT (State [a]) ()
push x = do
    s <- lift get
    lift . put $ x:s
    return ()

program1 = pop >> pop >> pop
program2 = pop >> push 3 >> pop
>>> runState (runMaybeT program1) []
(Nothing,[])
>>> runState (runMaybeT program2) [1, 2]
(Just 3,[2])
```

The interesting part happens in the pattern matching in `pop` where an empty list maps to `Nothing` (specifically a state monad returning `Nothing`). This will short-circuit `bind` for `MaybeT`.

Note how `get` is lifted. `lift` is a overloaded method that in this case transforms a value of type `State [a] a` into that of type `MaybeT (State [a]) a`. This is necessary since it has to be of type `MaybeT (State [a]) a`.

This provides a way to combine multiple monad types and work on each on of them using `lift`.

5.1.8.7 Free Monad

The free monad is a constructor that lifts any functor into a monad. We will introduce it from an operational perspective. Assume only the functor instance of `State` is implementation.

```
newtype State s a = State { runState :: s -> (a, s) }

instance Functor (State s) where
    fmap f (State g) = State (\st -> let (x, st') = g st
                                     in (f x, st'))
```

Following the work of [25] it is possible to lift this into a monad using the free monad.

```
data Free f r = Impure (f (Free f r)) | Pure r

liftF :: Functor f => f a -> Free f a
liftF = Impure . fmap Pure
```

`Impure` is called `Free` in the official implementation but is replaced here to make the distinction clear between the type constructor and the value constructor.

`Free f` is an instance of `Monad` and its definition can be found in the Appendix A.2. We refer to [47] for proof of the three monad laws. State can be lifted into a monad with `Free`[45]:

```
type FState s = Free (State s)

-- Using definitions for ‘get‘ and ‘put‘ from earlier
getF :: FState s s
getF = liftF get
```

```
putF :: s -> FState s ()
putF = liftF . put
```

This allows us to write an interpreter, and thereby pushing the definition of `bind` and `return` from the monad instance definition for the state monad to the function `runFState`.

```
runFState :: FState s a -> s -> (a,s)
runFState (Pure x) s   = (x,s)
runFState (Impure m) s = let (m',s') = runState m s in runFState m' s'
```

To clarify, this does not help us avoid specifying the monad behavior for the state monad. What it does is to separate the syntax for State from the semantics.

Much more sophisticated implementations exists[5][23] than the version included in Appendix A.2 but they are conceptually similar.

5.1.8.8 Interpreter for Free Monad

The free monad can be used for the interpreter pattern specified in [7] by building up a description of our program with a domain specific language (DSL) which is later interpreted. The functor `f` usually makes up the basic operation that the resulting DSL includes[3].

An example of this can found in Appendix A.3 but this is in many ways similar to the DSL for the probabilistic programming language which is defined in the section on *Monads for probability*.

5.2 Mapping Haskell to JavaScript

The JavaScript implementation draws on many of the concept from Haskell covered in Section 5.1. JavaScript is on many levels already suitable for a functional programming style since it treats functions as first-class citizens[14]. However, it does not have the same type safety built-in as Haskell. Furthermore concepts from category theory like monads are not widely used.

The main objective is to implement a domain specific language for probabilistic programming. To make this possible two critical parts needs to be ported from Haskell, namely do-notation and the free monad. This chapter will explain how.

5.2.1 *Do*-notation in JavaScript

There is no direct equivalent to do-notation in JavaScript. There is a way to use *Generators* introduced in JavaScript ECMAScript 6[13] to achieve similar functionality however. Details on generators will not be provided but in essence they allow for stepping through the execution of a function.

```
const gen = function* () {
    yield 1;
    const a = yield 2;
    yield a + 3;
};
const g = gen();
```

```
g.next();           // => { value: 1, done: false }
const a = g.next(); // => { value: 2, done: false }
g.next(a);          // => { value: 5, done: true }
```

This example captures the power of generators. For every `yield` keyword inside the function it is possible to decide when to execute it and what value it should take on in the rest of the computation.

This is sufficient for mimicking the do-notation. It has one limitation though. The monad returned by the do-notation cannot be evaluated (with e.g. `runState`) multiple times. This fails because generators in JavaScript are state based and mutable so the second time the monad is evaluated `g.next()` will not be at the beginning of the function execution as expected. A workaround for this is using immutable generators. It does have some performance implications that will be discussed in Chapter 8 with the proposal of alternatives.

5.2.2 Free Monad

Fantasy Land[11] has been used as the basis for the free monad. It is an attempt at standardizing the interface specification in JavaScript for *algebraic data types* (which includes monads and monoids). Many libraries that provides monads also implements support for the fantasy-land specification[39][37][33][2]. The specification makes it simpler to switch implementation in the future.

Amongst these libraries the official *Fantasy Land* implementation of `Free` was chosen since this closely matches the Haskell syntax introduced. This makes it conveniently easy to first built up the structure in Haskell.

5.3 Monads for Probability Distributions

The objective is to compose a probability distribution from primitive distributions as defined in Chapter 4. An elegant way of doing this is using do-notation e.g.:

```
let dist do =
        a <- uniform 2 5
        b <- normal a 1
        return (abs b)
```

This example motivates representing probability distributions as a monad. Recall that the do-notation will return a monad. This provides an interface for composing complex structures.

The realization that a probability distribution forms a monad is not new [15][20][27]. Several implementation of probability monads exist both exact[10] and sample based[41].

5.3.1 Forward Rejection Sampling

Recall from Chapter 4 on Bayesian Networks that samples can be drawn by sampling from each node (representing a primitive distribution) in topological order. Each primitive distribution needs access to a random generator to generate a sample. Since the generator is immutable the generator returned by the previous primitive distribution needs to be passed. Recognize that the generator can be viewed as a state and can therefore be modeled as a state monad (see Appendix A.4).

```
import Control.Monad.State (State, state, runState)
import System.Random (RandomGen, Random, randomR, mkStdGen)

uniform :: (RandomGen g, Random a, Num a) => a -> a -> State g a
uniform a b = state (randomR (a, b))

-- Example usage
let dist = do
    a <- uniform 0 10
    b <- uniform a 20
    return (a, b)

>>> runState dist (mkStdGen 2)
((6,15),508393462 1655838864)
```

Here a uniform sampler `randomR` is lifted into a monad which enables the use of do-notation that passes the generator state. This specific monad represents a network of node `a` and `b` where the lower bound of the `b` distribution is given by the stochastic result of `a`. Note how this allows one to specify the dependency between random variables without dealing with how the final distribution is used (e.g. by sampling).

The state monad in this specific implementation has another benefit in that it returns a function. This allows multiple samples to be drawn similarly to the approach used in [42].

```
sample = flip runState -- flips the argument order
>>> sample (mkStdGen 2) $ sequence $ replicate 10 $ dist
([(6,15),(2,17),(0,10),(7,12),(8,18)],792448003 1336516156)
```

This does not consider conditioning on the value of the random variables however. Rejection sampling is a simple way to handle this as was shown in Chapter 4 on Bayesian Network. Rejection sampling is possible by introducing the `Maybe` monad in the monad stack as proposed in [43] (see Appendix A.5 for imports).

```
uniform :: (RandomGen s, MonadState s m) => Double -> Double -> MaybeT m Double
uniform a b = lift . state $ randomR (a, b)

bernoulli :: (RandomGen s, MonadState s m) => Double -> MaybeT m Bool
bernoulli p = (< p) <$> uniform 0 1

condition :: (RandomGen s) => Bool -> MaybeT (State s) ()
condition = MaybeT . return . toMaybe
    where toMaybe True  = Just ()
          toMaybe False = Nothing

dist = do
     b <- uniform 0 2
     condition $ b < 0.5
     return b

sample = flip evalState
>>> catMaybes $ sample (mkStdGen 1) $ replicateM 10 $ runMaybeT dist
[0.15558075016668682,0.4176847224318532,0.10884877410599492,0.486820808729149]
```

This approach uses the `MaybeT` monad transformer to short-circuit the computation with `Nothing` if the

condition is `False` similarly to the example in Section 5.1.8.6 [4]. `catMaybes` is a built-in function that simply filters out `Nothing` values which is equivalent to rejecting those samples for which `condition` is parsed `False`.

5.3.2 Free Monad for Multiple Interpretations

Practical Probabilistic Programming with Monads suggest modelling a probabilistic program with a free monad [42] to allow for different kinds of interpretation. A simple implementation using Haskell has been prototyped and is included in Appendix A.6. This uses the approach covered in Section 5.1.8.7 but using the built-in `Free` monad. The recursiveness in the interpreter can be encapsulated using built-in helper function like `iterM` to avoid boilerplate. This simplification has been left out since it does not translate to the JavaScript implementation that is based on this prototype.

The Haskell implementation shows how to interpret the free monad by effectively transforming it to the sampling monad introduced in Section 5.3.1. In JavaScript the same interpretation pattern is then used to instead run a MCMC algorithm on the model. A model can be specified using generators to mimic do-notation:

```
Do(function* () {
    const a = yield bernoulli('a', 0.7);
    if (a) {
        return normal('n', 0, 1);
    } else {
        return normal('n', 5, 10);
    }
})
```

To use MCMC to infer parameters from data it has to be possible to specify observed data as described in Chapter 4 Bayesian Network. The syntax for specifying an observation is inspired by *Angelican*[51] and is done using the function `observe` that takes a distribution and an observation as parameters. A model specification using it is provided below:

```
Do(function* () {
    const lambda1 = yield exponential('lambda1', 1);
    const lambda2 = yield exponential('lambda2', 1);
    const tau = yield discreteUniform('tau', 0, data.length);
    for (const [idx, value] of data) {
        if (idx < tau) {
        yield observe(poisson('obs', lambda1), value);
        } else {
        yield observe(poisson('obs', lambda2), value);
        }
    }
    return Return(null);
});
```

Details on how the DSL has been extended to allow for this can be found in the implementation[5]. The DSL is interpreted with the Metropolis-Hastings algorithm proposed by Wingate, which will be covered next.

[4] `condition` can be replaced by the more general `guard` in Haskell which means rejection sampling is achieved by just lifting the monad to that of type `(RandomGen s, MonadState s m) => MaybeT m Double`.

[5] `https://github.com/tmpethick/mcmc/tree/thesis`

Chapter 6

Implementation

The domain specific language implemented with the free monad allows for multiple interpretation. The JavaScript implementation uses the lightweight Metropolis-Hastings algorithm proposed by Wingate et al.[50]. It is convenient as a proof of concept due to its simplicity. This chapter will formally define how it works.

6.1 Markov Chain Monte Carlo with Random Database

Wingate et al. proposed a method of extending an arbitrary programming language to a probabilistic programming language with Metropolis-Hastings MCMC inference coined by [51] as *Random Database*. This section will provide an overview of the terminology used and a description of the method. For proof of correctness the reader is referred to the original paper.

6.1.1 Overview

To understand it, we first consider a probabilistic program without conditioning which we define as a parameterless function f containing elementary random primitives (ERP) which in this context are functions with stochastic return values. During execution of f the ERPs will be evaluated and, depending on the return value, different paths through f will get executed. These paths are called *execution traces*.

The kth ERP in a given execution path is called $f_{k|x_1,\ldots,x_{k-1}}$ where x_k is the return value for the kth ERP. Note that across different execution paths the kth ERP might be of different type and have different parameters. Additionally, even the execution path might be of a different length. Even though these conditions are critical, we omit them for brevity and use f_k to represent $f_{k|x_1,\ldots,x_{k-1}}$.

The papers key insight is to transform the function f into another function f' that uses a database to look up the return values for f_k. This is done by first associating a name with each f_k for every execution trace. The function f' is then created by replacing every f_k with f'_k. When f'_k runs in a given transformed execution path, it will look up the value of x_k in the database using its name.

Provided it is not there it will sample from the associated ERP and store the return value in the database. Additionally it accumulates the likelihood score, since this is required for the acceptance step in Metropolis-Hastings.

This enables an MCMC algorithm to propose a new execution trace by modifying a sample x_k in the database. Executing f' with the modified database would then change the execution trace and return a likelihood that could be compared with the old trace.

6.1.2 Random Database

With this intuition in place the database and the procedure will be described in more detail. The database $\mathcal{D}$ is a mapping $\mathcal{N} \Rightarrow \mathcal{T} \times \mathcal{X} \times \mathcal{L} \times \Theta$ where an element $n \in \mathcal{N}$ is a name for a f_k mapping to $(t, x, l, \theta) \in \mathcal{T} \times \mathcal{X} \times \mathcal{L} \times \Theta$ where t is the ERP type, x is the sample from f_k, l is the likelihood of x for the ERP specified by t and θ, and θ is the ERP parameters. Missing entries are allowed. The database should support getting and setting values. Additionally it should define a method for picking a name uniformly at random from $N \subset \mathcal{N}$ where N is all names for which an entry exists.

6.1.3 Update Trace

f' is then a function taking $\mathcal{D}$ as parameter. For every f'_k it will retrieve its associated name $n \in \mathcal{N}$, parameters $\theta_c \in \theta$, and $t_c \in \mathcal{T}$. If $(t_{db}, x, l, \theta_{db}) = \mathcal{D}(n)$ exists and $t_c = t_{db}$ then x is used as return value for f'_k in the continued execution of f'. Otherwise we sample x' from the ERP, compute the likelihood l' and update the database accordingly, *store* $\mathcal{D}(n) = (t_c, x', l', \theta_c)$. There are a few subtleties that are covered in Algorithm 2 extracted from the paper. `trace_update` will from here on be used interchangeably with f'.

Algorithm 1 MCMC trace sampler

1: select a random name n from $\mathcal{D}$
2: $[ll, \mathcal{D}] =$ `trace_update`$(\emptyset)$
3: **loop**
4: $\quad get \quad (t, x, l, \theta_{db}) = \mathcal{D}(n)$
5: $\quad$ Propose a new $x' \sim \mathcal{K}_t(.|x, \theta_{db})$
6: $\quad F = log\mathcal{K}_t(x'|x, \theta_{db})$
7: $\quad R = log\mathcal{K}_t(x|x', \theta_{db})$
8: $\quad l' = logp_t(x'|\theta_{db})$
9: $\quad$ Let $\mathcal{D}' = \mathcal{D}$
10: $\quad$ store $\mathcal{D}'(n) = (t, x', l', \theta_{db})$
11: $\quad [ll', \mathcal{D}'] =$ `trace_update`$(\mathcal{D}')$
12: $\quad$ **if** $log(rand) < ll' - ll + R - F$ **then**
13: $\quad\quad$ // Accept
14: $\quad\quad [ll, \mathcal{D}] = [ll', \mathcal{D}']$

6.1.4 Markov Chain Monte Carlo Algorithm

As has already been specified it should be possible to calculate likelihood and sample from every ERP. Given a sample x there should also exist a mapping from a ERP with type t and parameters θ to a proposal kernel $\mathcal{K}_t(x'|x, \theta)$ that also is an ERP. This will be used in the proposal step of the MCMC algorithm.

Algorithm 2 `trace_update`($\mathcal{D}$)

$ll = 0$
$\mathcal{D}' = \emptyset$
for all For all random choices f_k in sequential order **do**
    Determine (n, t_c, θ_c) for f_k
    get $(t_{db}, x, l, \theta_{db}) = \mathcal{D}(n)$
    if if element exists in $\mathcal{D}$ and $t_c = t_{db}$ **then**
        if $\theta_c = \theta_{db}$ **then**
            $ll = ll + l$
        else
            $l = logp_{t_c}(x'|\theta_c)$
            $ll = ll + l$
    else
        sample $x \sim p_{t_c}(.|\theta_{db})$
        $l = logp_{t_c}(x'|\theta_c)$
        $ll = ll + l$
    store $\mathcal{D}'(n) = (t_c, x, l, \theta_c)$
    Set the return value of f_k to x to potentially alter the execution trace
return $[ll, \mathcal{D}']$

An MCMC algorithm can be defined based on the way f' updates the execution trace. Given a current trace defined by a state of the database and a calculated likelihood, pick a random name for which an entry $(t_{db}, x, l, \theta_{db})$ exists in the database. Propose a new x' by sampling from the proposal kernel $\mathcal{K}_t(x'|x, \theta)$ and store the modification in the database. We calculate the likelihood of the new trace by executing `trace_update` on the modified database. The acceptance ratio can then be calculated based on the likelihood of the two traces and the proposal kernels similarly to the description in the Metropolis-Hastings section.

6.1.5 Modifications

We deviate from the original implementation slightly. The original implementation removes unused entries from $\mathcal{D}$ if the proposed trace is accepted. This ensures that we sample from the prior if we reexplore an execution trace later. It also ensures the next transition step does not pick a random name that has no possibility of updating the execution trace. It does not specify how the cleanup is done, however. We propose a slight modification that instead accumulates a new database in `trace_update`. Since only one execution path is visited in f' it follows that stale entries are not included in the accumulated database. This may have performance implications, but it allows for a simpler implementation.

The paper avoids mentioning how conditioning is handled even though this is what motivates MCMC. A solution to this is proposed in [51] where the likelihood for each observed ERP is accumulated with the observed value as its sample.

Both of these modification are included in the pseudo code provided in Algorithm 1 and Algorithm 2. Notice that `trace_update` accumulates the log likelihood instead of the likelihood since this is closer to the actual implementation. We will use the fact that it can be viewed as a sum monoid and similarly the database is a monoid defined by the triplet $(\mathcal{N} \times \mathcal{T} \times \mathcal{X} \times \mathcal{L} \times \Theta, \emptyset, store)$ if seen as a set of input and output pairs.

So far a naming scheme has been assumed. Wingate moves on to define how this naming scheme can be automated in the transformation from f to f' with a source-to-source compilation. We leave this out for simplicity by assuming every f_k in every execution trace has a unique name defined manually

in the probabilistic program. Uniqueness is not a strict requirement for convergence as shown in [50] but it increases the acceptance rate.

6.1.6 Relationship with the Free Monad

With a free monad it is possible to turn a structural representation of the program into an executable one through a natural transformation of the monad. We have already seen this for a very simple probablistic program using rejection sampling. Note how this transformation is similar to the transformation done from f to f'. This means that f can be specified with do-notation and later interpreted into f' making it possible to run the trace-based MCMC algorithm. The implementation can be found on `github.com/tmpethick/mcmc`.

Chapter 7

Evaluation

Having implemented the Metropolis-Hastings algorithm, it can now be used for probabilistic programming.

7.1 Revisiting Student Heights

Recall the BN describing the mean height of the students at DTU from Section 4.2, seen again on Figure 7.1.

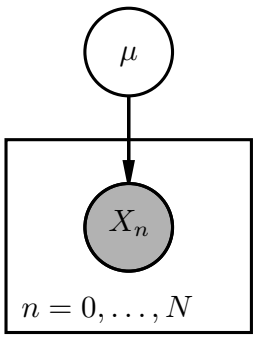


Figure 7.1: Student Heights Example Using BN

The distributions of the random variables were determined as follows

$$\mu \sim \mathcal{N}(170, 10)$$
$$X_n|\mu \sim \mathcal{N}(\mu, 12.4272^2)$$

This can then be modelled in the implemented probabilistic programming language as follows

```
Do(function* () {
    const mu = yield normal('mu', 170, 10);
```

```
    for (const o of obs) {
        yield observe(normal('o', mu, 12.4272), o);
    }
    return Return(null);
}),
```

Where obs is an array containing the data points.

Here the simplicity that a Probabilistic Programming Language offers is evident. With little theoretical foundation, it is possible to model dependent distributions. All that is needed to write the above script, is the knowledge as to which distributions are suitable for the problem. This allows researchers that have otherwise been wary of going into probabilistic modelling, to try their hand at estimating distributions, rather than single parameters.

Running this with $10,000$ samples, the convergence of the samples for μ can be seen on Figure 7.2, where only the first $1,000$ samples are shown

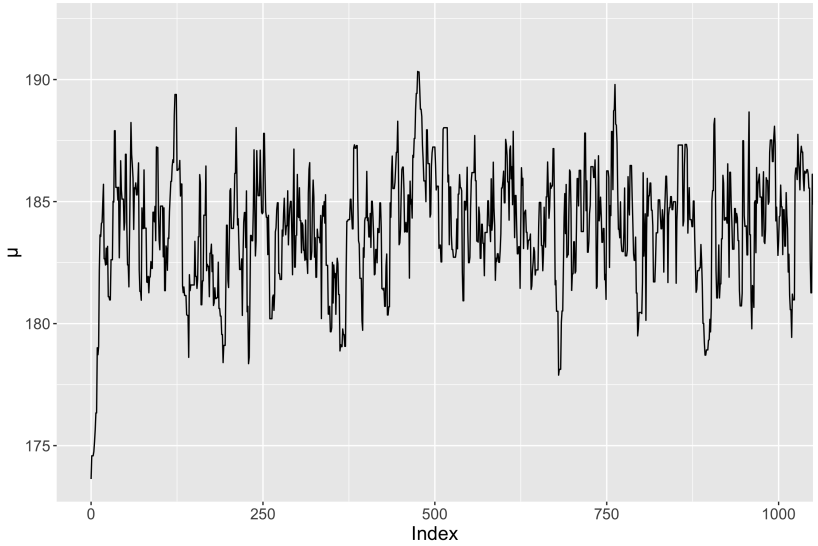


Figure 7.2: Convergence plot for samples describing μ

It quickly turns into a stationary distribution, which is why a burn-in of 500 is chosen. This yields the estimated distribution for μ first seen in Section 3.2.1.2, shown again on Figure 7.3 for completeness

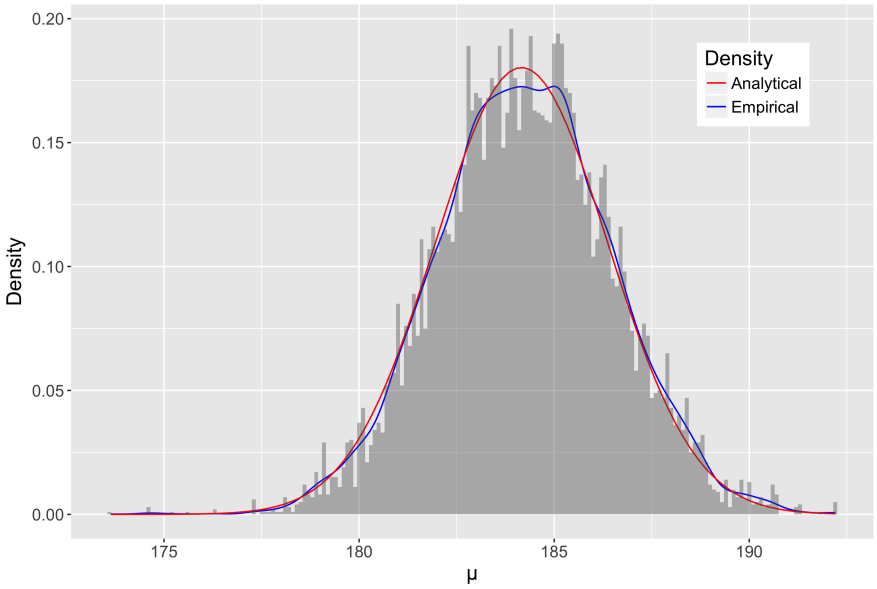


Figure 7.3: Sample and analytical density describing μ, using $9,500$ samples

7.2 Revisiting Linear Regression

Now recall the BN describing probabilistic linear regression from Section 4.3.1, seen again on Figure 7.4.

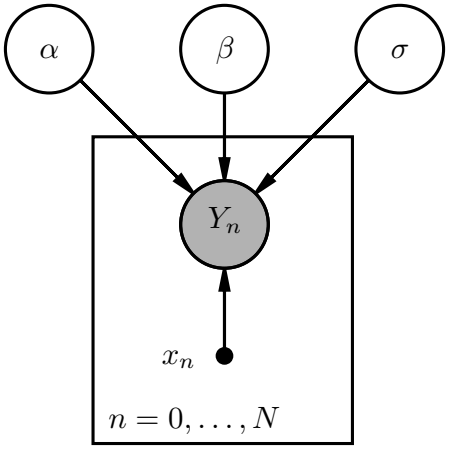


Figure 7.4: Linear Regression Using BN

The parameters of this model can now be estimated using the implemented Metropolis-Hastings algorithm.

To do this, it is necessary to first simulate some data. This is done by defining the target line, and adding noise, ϵ. For this example, the following is chosen

$$Y = 2x + 6 + \epsilon, \quad \epsilon \sim \mathcal{N}(0, 1).$$

This is the model used for simulating the data seen on Figure 7.5.

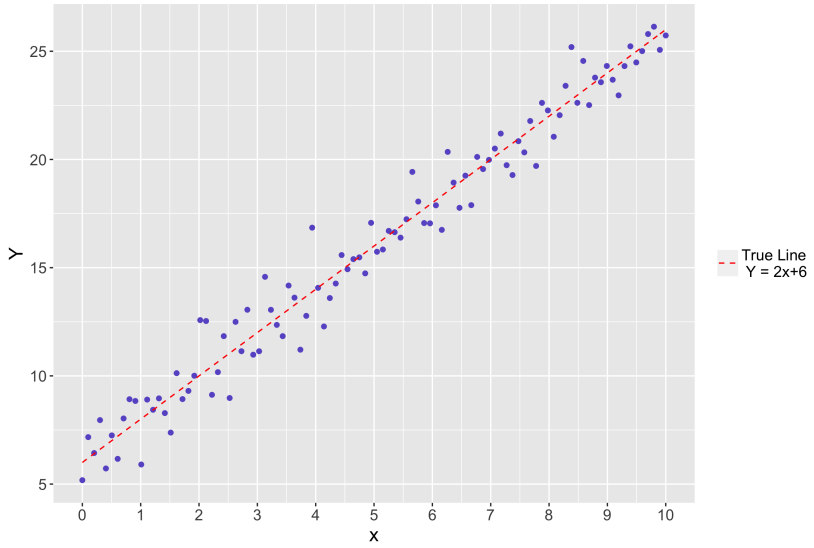


Figure 7.5: Simulated data for Linear Regression

Now let the true parameters be unknown, it is then interesting to model the parent nodes from the BN seen on Figure 7.4.

As mentioned in Section 4.3.1, σ is the standard deviation of a normal distribution, and as such is restricted to $\mathbb{R}_+$. To ensure this, it is assumed that $\sigma \sim \text{Exp}(1)$. As there is no prior knowledge regarding the slope and intercept and $\alpha, \beta \in \mathbb{R}$, they are both given the same prior distribution $\alpha, \beta \sim \mathcal{N}(0, 10)$. Further it is assumed that $Y_n \sim \mathcal{N}(\alpha x_n + \beta, \sigma^2)$.

Now the distributions for the random variables in the BN on Figure 7.4 can be written

$$\begin{aligned}
\alpha &\sim \mathcal{N}(0, 10) \\
\beta &\sim \mathcal{N}(0, 10) \\
\sigma &\sim \text{Exp}(1) \\
Y_n|\alpha, \beta, \sigma &\sim \mathcal{N}(\alpha x_n + \beta, \sigma^2).
\end{aligned}$$

Having defined the distributions for each of the random variables, the factorization can now be determined. From Chapter 4 it is known that this is the stationary distribution $\boldsymbol{\pi}$ that is used by Metropolis-Hastings. While this is not necessary to know in order to model the program, it gives an understanding of the inner workings of the program.

By utilizing that α and β follow the same normal distribution to simplify the expression, the factorization can be written

$$\begin{aligned}
&f(\alpha)f(\beta)f(\sigma)f(Y|\alpha, \beta, \sigma, \mathbf{x}) \\
&\quad = \frac{1}{2\pi \cdot 10^2} \exp\left\{-\frac{\alpha^2 + \beta^2}{2 \cdot 10^2}\right\} \cdot e^{-\sigma} \cdot \prod_{n=0}^{N} \left(\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(y_n - (\alpha x_n + \beta))^2}{2\sigma^2}\right\}\right).
\end{aligned} \tag{7.1}$$

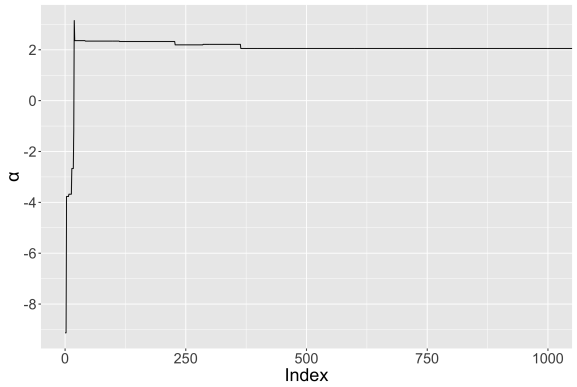
The problem can now be modelled in the implemented probabilistic programming language as follows

```
Do(function* () {
        const sigma = yield exponential('sigma', 0.5);
        const beta = yield normal('beta', 0, 10);
        const alpha = yield normal('alpha', 0, 10);
```

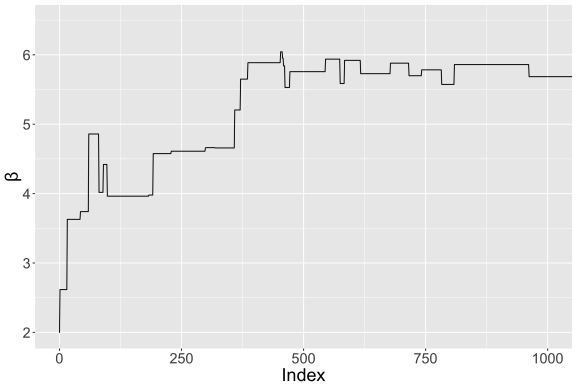
```
    for (const [x, y] of data) {
      yield observe(normal('o', beta + alpha * x, sigma), y);
    }
    return Return(null);
  }),
```

Again it is seen how a Probabilistic Programming Language simplifies the process of probabilistic modelling. There is no need to work with the likelihoods or priors, as these are all handled by the program.

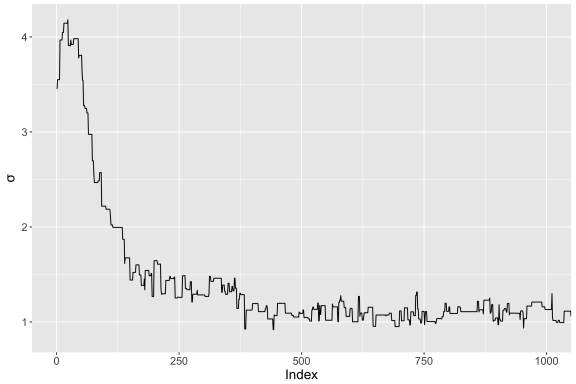
Running this with $10,000$ samples yields the following convergence plots for the three parameters, where only the first $1,000$ samples are shown



(a) Convergence plot for α



(b) Convergence plot for β

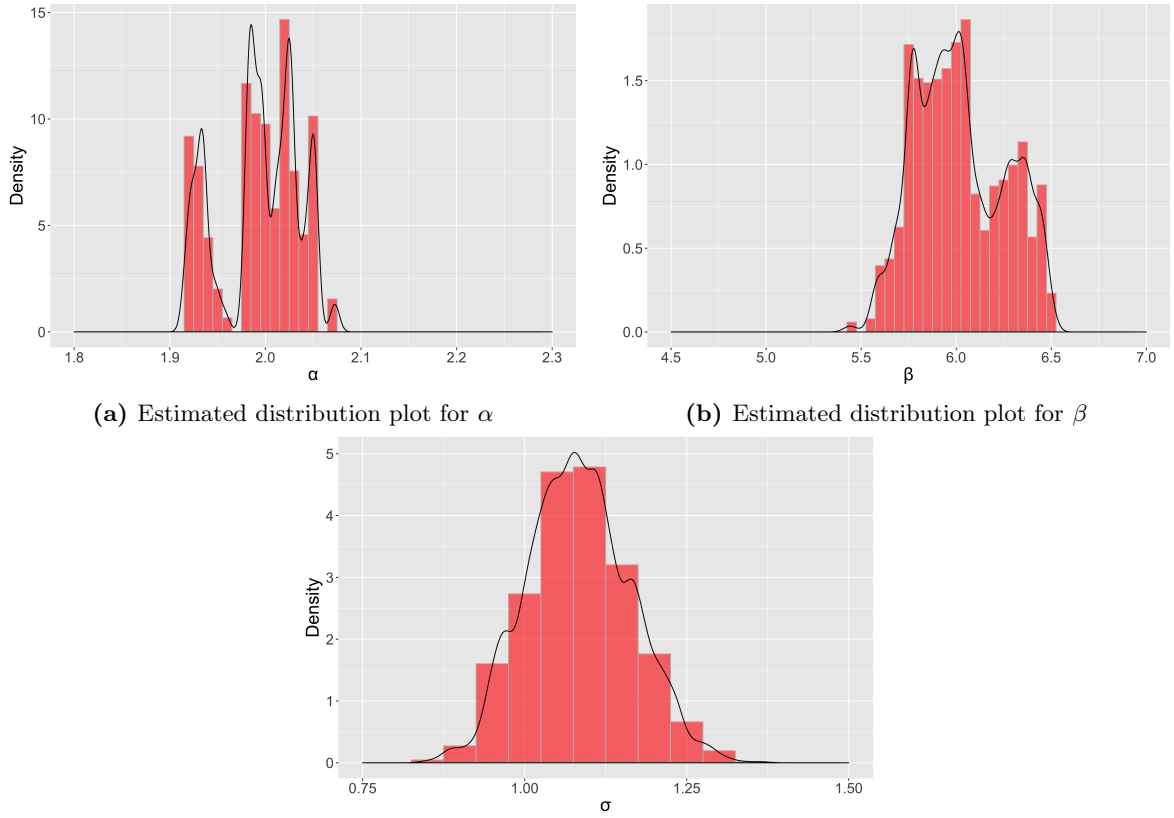


(c) Convergence plot for σ

Figure 7.6: Convergence plots for the parameters in the Linear Regression model

From Figure 7.6 it is seen that all of the parameters have converged around step 500, therefore this is chosen as the burn-in. It is seen that α has a low acceptance rate. This might be because the standard deviation of the proposal distribution is too large, once α has found the true value, $\alpha_{true} = 2$. This is also evident from the jaggedness of the estimated distribution for α seen on Figure 7.7a.

Having determined convergence, the distributions for the parameters can be seen on Figure 7.7



(a) Estimated distribution plot for α

(b) Estimated distribution plot for β

(c) Estimated distribution plot for σ

Figure 7.7: Estimated distribution plots for the parameters in the Linear Regression model

From Figure 7.7 it is seen that all of the distributions capture the true parameters $\alpha_{true} = 2$, $\beta_{true} = 6$ and $\sigma_{true} = 1$. Further as distributions are available for α and β, this gives a range of linear regressions, instead of a single line, whereas the estimation of σ gives an indication of the size of the observations noise.

7.3 Revisiting Poisson Changepoint

Now recall the BN describing a change of rate for a Poisson process from Section 4.3.2, seen again on Figure 7.8.

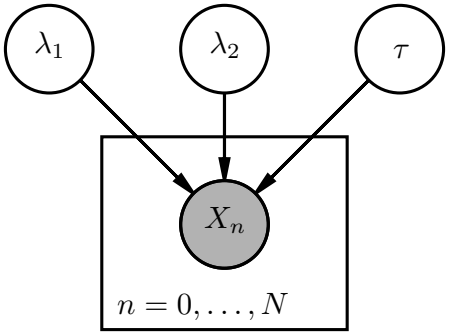


Figure 7.8: Model of Mining Disasters Using BN

From [4, p. 395] a data set describing the number of British coal-mining disasters is available. This contains the number of times that 10 or more people died each year in a coal mining explosion from 1851 to 1962. As such the data has a discrete index, corresponding to the given year, and the observed data can be assumed to follow a Poisson process, making it suitable for a changepoint analysis.

The observed data can be seen on Figure 7.9

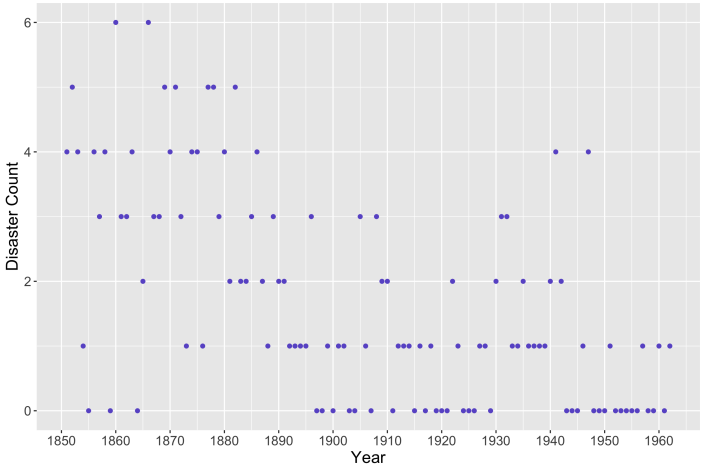


Figure 7.9: Amount of British coal-mining disasters from 1851 to 1962

As the rates λ_1 and λ_2 are defined on $\mathbb{R}_+$, a noninformative prior which maps to the proper range has to be chosen. One possible choice is $\lambda_1, \lambda_2 \sim \text{Exp}(1)$. Further, as the index is discrete and there is no prior knowledge with regards to the changepoint, the prior for τ can be chosen as the discrete uniform distribution over the entire time-period, $\tau \sim \mathcal{U}(1851, 1962)$. Lastly, the observations X_n are assumed to follow a Poisson process, with the rate being determined by whether it is before or after the changepoint. Summarized that yields the following distributions

$$\begin{aligned} \lambda_1 &\sim \text{Exp}(1) \\ \lambda_2 &\sim \text{Exp}(1) \\ \tau &\sim \mathcal{U}\{0, 111\} \\ X_n | \lambda_1, \lambda_2, \tau &\sim \begin{cases} \text{Pois}(\lambda_1) & n < \tau \\ \text{Pois}(\lambda_2) & n \geq \tau \end{cases}. \end{aligned}$$

For completeness, the factorized joint distribution, and thus target stationary distribution $\boldsymbol{\pi}$, is written

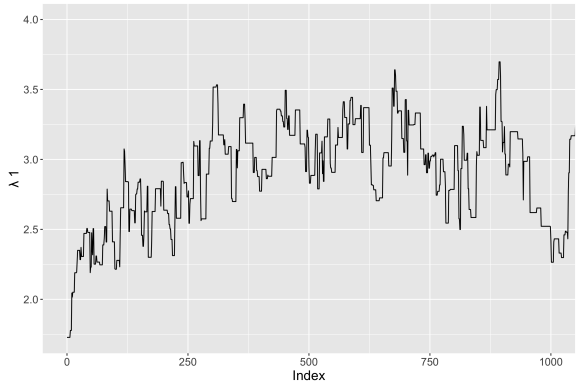
$$f(\lambda_1)f(\lambda_2)f(\tau)f(X|\lambda_1,\lambda_2,\tau) = e^{-\lambda_1}e^{-\lambda_2}\frac{1}{112}\prod_{n=0}^{\tau-1}\frac{\lambda_1^{x_n}}{(x_n)!}e^{-\lambda_1}\prod_{n=\tau}^{N}\frac{\lambda_2^{x_n}}{(x_n)!}e^{-\lambda_2}. \tag{7.2}$$

However, like with the linear regression model, the value of this is calculated behind the scenes as the program samples, and as such is not a part of the modelling. To model the problem, it is only necessary to write up the assumed distributions, as well as how they are dependent on one another. That is done like follows

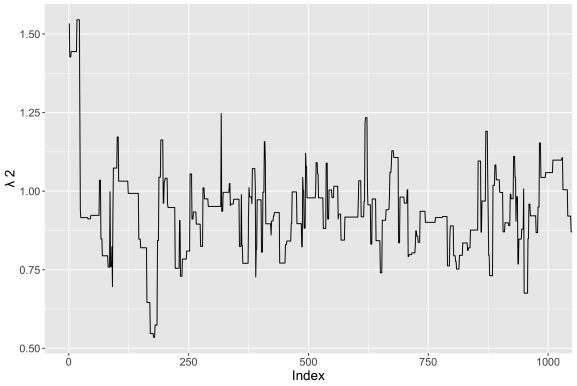
```
Do(function* () {
    const lambda1 = yield exponential('lambda1', alpha);
    const lambda2 = yield exponential('lambda2', alpha);
    const tau = yield discreteUniform('tau', 0, data.length);
    for (const [idx, value] of data) {
        if (idx < tau) {
            yield observe(poisson('obs', lambda1), value);
        } else {
            yield observe(poisson('obs', lambda2), value);
        }
    }
    return Return(null);
});
```

Listing 1: PPL for infering changepoint

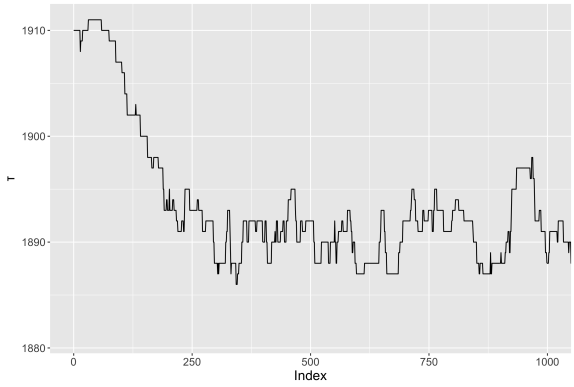
Running this model with $10,000$ samples yields the following convergence plots for the three parameters, where only the first $1,000$ samples are shown



(a) Convergence plot for λ_1



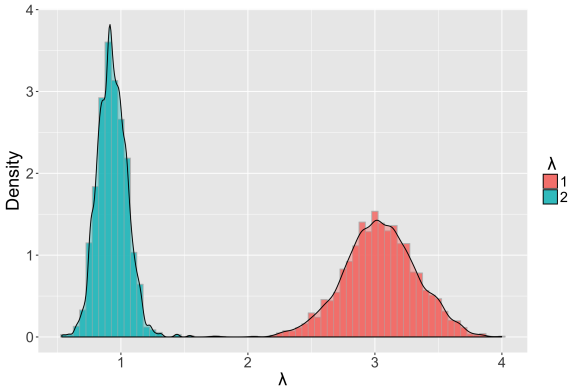
(b) Convergence plot for λ_2



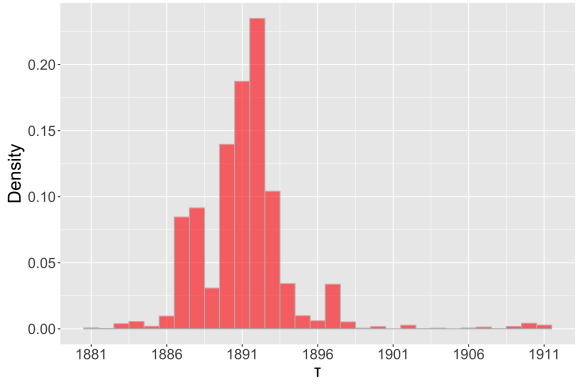
(c) Convergence plot for τ

Figure 7.10: Convergence plots for the parameters in the Coal Mining Disaster changepoint model

From Figure 7.10 it is seen that the parameters have all converged at step 500. Therefore this is chosen to be the burn-in. Using the remaining $9,500$ samples yields the following distributions for λ_1, λ_2 and τ



(a) Histogram of the values λ_1 (red) and λ_2 (teal) sampled



(b) Histogram of sampled τ values showing potential switch points

Figure 7.11: Estimated distributions for λ_1, λ_2 and τ

From Figure 7.11 it is seen that the distributions for λ_1 and λ_2 are disjoint, which indicates that there are in fact two different rates. As for the changepoint, the distribution for τ shows a couple of

possibilities, however the majority of these lie within a 10 year span, with especially years 1892 and 1893 being likely candidates for the change. This coincides nicely with the Mining Regulation Act of 1887 which imposed additional safety features such as prohibiting single shaft mines [17]. This is also in line with the results found in [4], indicating a correct convergence of the program.

Chapter 8

Discussion

The work has focused on building an expressive DSL for probabilistic programming in JavaScript as a proof of concept. It has been shown to infer correctly on smaller problems. It is orders of magnitudes slower than state of the art probabilistic programs like Church or PyMC though. This chapter will discuss four areas: 1) possible immediate performance improvements, 2) how the program can be extended, 3) the expressivity of the implemented DSL, and 4) a type safe environment in JavaScript.

It will finish by providing an overview over similar existing work.

8.1 Performance

Our interest has been in Bayesian inference, which provided a prior, and observations calculates a posterior. The evaluation cases has reflected this by all sampling from a conditioned distribution. The majority of ERPs composing the joint distribution are in fact observed. This is problematic performance-wise since every observed distribution is yielded (see e.g. Listing 1). Every one of these `yield`s in the generator syntax is turned into a `bind` for which a recursive call to the interpreter is made. This could be avoided by allowing vector arguments for both ERP distributions and the `observe` expression. This is a method seen in PyMC but the performance gain has not been explored.

As has been seen the `yield` structure and monads are evaluated sequentially. This naturally leads us to the second performance improvement: independence. This provides a simple evaluation scheme but it does not allow the algorithm to exploit independence assumption in the distribution since the DSL is simply not expressive enough. Currently every ERP requires all previous ERPs in the monad chains to be evaluated. However, it is possible to expand the current implementation using something called *Free Applicative Functors* [3] to allow for these assumptions to be specified. This is a general method that allows for the DSL to express context-free effects [30]. An implementation that does not use Applicatives but allows for expressing independent distributions is the $C^{\#}$ [19] port of *Practical Probabilistic Programming with Monads* (using a different Metropolis-Hastings algorithm, however).

The naive implementation of the free monads used has quadratic time complexity in regard to the number of `bind` [47, p. 12]. Several suggestions exist that reduce this to linear [47] [35] for which implementations exists in Haskell but not JavaScript, to the best of our knowledge.

There is another layer to the complexity which is the use of generators in JavaScript. Because of the non-native immutability mentioned in Section 5.2.1 the generator has a time complexity of $O(n^2)$ where n is the number of yields [6]. It is possible to achieve linear performance by transpiling the generator syntax directly to the `bind` structure or extend JavaScript with do notation using e.g. sweet.js [44]. However, there already exists mature projects like WebPPL that extends the JavaScript language this way. One of the benefits of our implementation is that it avoids source-to-source transpilation.

8.2 Features

Each ERP has been equipped with a proposal kernel as described in the Metropolis-Hastings algorithm used [50]. The convergence rate of MCMC depends heavily on the proposal [...] so it would be beneficial to support user defined proposal kernels. A JavaScript implementation, WebPPL, support this by allowing the user to specify the mapping from an ERP to a proposal kernel [8]. Currently the built-in proposals for all continuous distributions are Gaussian based on the standard deviation of the ERP and centered around the previous sample. A more sophisticated mapping could be considered.

Another important factor for convergence is the choice of initial state in the Markov Chain which has an effect on the burn-in period. Currently they are drawn from the prior using `trace_update` (see Algorithm 2) which is not problematic for the small problems used in Chapter 7. Production-ready libraries like PyMC allow for custom initial state so that optimization algorithms like *maximum a posteriori* (MAP) can be used [36].

8.3 Expressiveness

The Random Database based Metropolis-Hastings algorithm implemented allows for an unbounded number of random choices as has been shown with the recursive model for the geometric distribution [50, p. 775]. The geometric distribution can be expressed recursively with the free monad as well:

```
geo p = do
    a <- bernoulli p
    if a then return 1
    else do
        g <- geo p
        return $ g + 1
```

The language can thus express models that those compiling to finite factor graphs like BUGS and Infer.NET cannot. Some of the performance implications of this expressiveness have already been covered in section 8.1.

The automated naming scheme from [50] was not implemented which makes the interface similar to PyMC in that a name is required for each constructed ERP. Since the DSL and the interpreter is separated using the free monad the name in not entangled in the ERP implementation. This is one of the benefits of the separation.

8.4 Types

Type checking is a powerful tool to ensure correct use of the program by the end user but also in developing it. A few attempts were tried to make the Monad implementation type safe but none was successful. The exploration is included here as a short review of the current landscape. There are two options for static type checking in JavaScript, Flow [12] and TypeScript[46]. None of them have support for higher-kinded polymorphism needed for Monads but there exists a way to mimic this [52]. Some of the common algebraic types in Haskell has been ported to Flow using this method [1] as well as TypeScript [2]. At the time of writing these are quite new initiatives which lack some of the required types but they seem promising.

8.5 Related Work

The language that most closely resembles ours is the mentioned WebPPL which implements its own subset of JavaScript augmented to allow for representation and manipulation of probability distribution. Our implementation differs in that no source-to-source transpilation is needed by exploiting the similarity between do-notation in Haskell and generators in JavaScript. The construction of WebPPL is well documented and implements many more inference algorithm including enumeration, Hamiltonian Monte Carlo and variational inference in *The Design and Implementation of Probabilistic Programming Languages* [16]. It does not cover the theory behind MCMC as thoroughly but we suggest it as a primary source on learning how to implement a probablistic programming language.

Another related language is BUGS which does inference on a finite graphical model using Gibbs sampling[1]. It promotes modelling the problem as a directed graphical model similarly to how we have approach our problems [31]. The comparison stops at that, however, since BUGS is a much faster and production ready tool.

Even though the problems has been modeled as finite graphical models our language is more expressive as was shown with the recursive definition of the geometric distribution. This is closer to *universal probablistic languages* like Church, Venture and Anglican in terms of expressiveness.

It is only possible to cover a small subset of the field in such a review. We refer to [42, p. 173] for a more complete overview of the landscape.

[1] Gibbs sampling is a MCMC method.

Chapter 9

Conclusion

The goal of this paper was to provide the reader with a firm theoretical foundation in probabilistic modelling, with a special focus on the Markov Chain Monte Carlo method and Metropolis-Hastings algorithm. The theory was then followed by an in-depth look at how to implement a probabilistic programming language using functional programming and monads. This was done in an effort to bridge the gap between the theoretical understanding of Probabilistic Programming, and the application to real problems through ones own implementation.

Primarily as a proof of concept, we demonstrated the possibility to do Probabilistic Programming using JavaScript. This is also evident in the speed of the program, as it was not built to be efficient, but to showcase possible overlaps between Haskell, monads and JavaScript. As evident from Chapter 7, this was successful, as solutions to the problems modelled in Chapter 4 were all successfully estimated.

This was achieved by a description of the theory behind Markov Chains, Monte Carlo Sampling, Bayesian Networks, Functional Programming and Monads. Further the paper was made to be replicable with all of the source material available on GitHub[1]. As such all of the figures and results can be replicated, by running the examples and R scripts found on the repository.

All in all this paper provides the reader with an understanding of the core elements of probabilistic programming, giving insight into what mathematics is being done behind the scenes, as well as how the program functions from a programming perspective.

[1] `https://github.com/tmpethick/mcmc/tree/thesis`

Bibliography

[1] Giulio Canti. *flow-static-land: Implementation of common algebraic types in JavaScript + Flow*. original-date: 2016-08-14T12:42:44Z. May 27, 2017. URL: https://github.com/gcanti/flow-static-land.

[2] Giulio Canti. *fp-ts: Implementation of common algebraic types in TypeScript*. original-date: 2017-01-25T17:44:09Z. June 4, 2017. URL: https://github.com/gcanti/fp-ts.

[3] Paolo Capriotti and Ambrus Kaposi. "Free Applicative Functors". In: *Electronic Proceedings in Theoretical Computer Science* 153 (June 5, 2014), pp. 2–30. ISSN: 2075-2180. DOI: 10.4204/EPTCS.153.2. arXiv: 1403.0749. URL: http://arxiv.org/abs/1403.0749.

[4] Bradley P. Carlin, Alan E. Gelfand, and Adrian FM Smith. "Hierarchical Bayesian analysis of changepoint problems". In: *Applied statistics* (1992), pp. 389–405. URL: http://www.jstor.org/stable/2347570 (visited on 06/04/2017).

[5] *Control.Monad.Free.Church*. URL: https://hackage.haskell.org/package/free-4.12.4/docs/Control-Monad-Free-Church.html (visited on 05/29/2017).

[6] Tom Crockett. *immutagen: A library for simulating immutable generators in JavaScript*. original-date: 2016-05-21T19:34:45Z. Apr. 12, 2017. URL: https://github.com/pelotom/immutagen.

[7] *Design Patterns: Elements of Reusable Object-Oriented Software: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch: 8601419047741: Amazon.com: Books*. URL: https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612 (visited on 05/29/2017).

[8] *Drift Kernels — webppl 0.9.8 documentation*. URL: https://webppl.readthedocs.io/en/master/driftkernels.html?highlight=drift%20kernel (visited on 06/04/2017).

[9] Roger Eckhardt. "Stan Ulam, John Von Neumann, and the Monte Carlo Method". In: (). URL: http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-88-9068 (visited on 05/26/2017).

[10] Martin Erwig and Steve Kollmansberger. "Functional pearls: Probabilistic functional programming in Haskell". In: *Journal of Functional Programming* 16.1 (2006), pp. 21–34. URL: http://journals.cambridge.org/abstract_S0956796805005721 (visited on 05/29/2017).

[11] *fantasy-land: Specification for interoperability of common algebraic structures in JavaScript*. original-date: 2013-04-12T03:10:22Z. June 5, 2017. URL: https://github.com/fantasyland/fantasy-land.

[12] *Flow: A Static Type Checker for JavaScript*. Flow. URL: https://flow.org/en/ (visited on 06/04/2017).

[13] *function**. Mozilla Developer Network. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function* (visited on 06/03/2017).

[14] *Functions*. Mozilla Developer Network. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions (visited on 06/03/2017).

[15] Michele Giry. "A categorical approach to probability theory". In: *Categorical aspects of topology and analysis*. Springer, 1982, pp. 68–85. URL: http://link.springer.com/content/pdf/10.1007/BFb0092872.pdf (visited on 05/29/2017).

[16] Noah D Goodman and Andreas Stuhlmüller. *The Design and Implementation of Probabilistic Programming Languages*. 2014. URL: http://dippl.org.

[17] Henry Best Hans Hamilton. *Digest of the Statutory Law Relating to the Management and Rating of Collieries for the Use of Colliery Owners, Viewers and Inspectors*. Walter Scott Publishing Company, 1902.

[18] Graham Hutton. *Programming in Haskell*. OCLC: 173844370. Cambridge, UK; New York: Cambridge University Press, 2007. ISBN: 978-0-511-29615-4 978-0-511-29218-7 978-0-521-69269-4 978-0-521-87172-3 978-0-511-29538-6 978-0-511-81367-2. URL: http://www.books24x7.com/marc.asp?bookid=23158 (visited on 06/02/2017).

[19] joashc. *csharp-probability-monad: A probabilistic programming framework for C#*. original-date: 2016-04-13T18:33:00Z. June 2, 2017. URL: https://github.com/joashc/csharp-probability-monad.

[20] C. Jones and G. D. Plotkin. *A Probabilistic Powerdomain of Evaluations*. URL: http://homepages.inf.ed.ac.uk/gdp/publications/Prob_Powerdomain.pdf (visited on 05/29/2017).

[21] Mark P. Jones. "A system of constructor classes: overloading and implicit higher-order polymorphism". In: *Journal of functional programming* 5.1 (1995), pp. 1–35. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/a-system-of-constructor-classes-overloading-and-implicit-higher-order-polymorphism/D2D05B479E365B913F7F0CD3543CB7EE (visited on 06/03/2017).

[22] Mark P. Jones and Luc Duponcheel. *Composing monads*. Technical Report YALEU/DCS/RR-1004, Department of Computer Science. Yale University, 1993. URL: https://pdfs.semanticscholar.org/0603/2750d7d0fa672f76d294e18c992bee4f1d55.pdf (visited on 06/03/2017).

[23] *kan-extensions: Kan extensions, Kan lifts, various forms of the Yoneda lemma, and (co)density (co)monads*. URL: https://hackage.haskell.org/package/kan-extensions (visited on 05/29/2017).

[24] Eric Kidd. "Build your own probability monads". In: *Draft paper for Hac* 7 (2007). URL: https://pdfs.semanticscholar.org/95e7/55313d37e49f4cae9ca86f8a11dc92ae276d.pdf (visited on 06/02/2017).

[25] Oleg Kiselyov and Hiromi Ishii. "Freer Monads, More Extensible Effects". In: *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. Haskell '15. New York, NY, USA: ACM, 2015, pp. 94–105. ISBN: 978-1-4503-3808-0. DOI: 10.1145/2804302.2804319. URL: http://doi.acm.org/10.1145/2804302.2804319.

[26] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. Adaptive computation and machine learning. Cambridge, MA: MIT Press, 2009. ISBN: 978-0-262-01319-2.

[27] F. W. Lawvere. *The Category of Probabilistic Mappings*. URL: http://www.fuw.edu.pl/~kostecki/scans/lawvere1962.pdf (visited on 05/29/2017).

[28] Henrik Madsen. *Time Series Analysis*. 1st ed. Chapman Hall/CRC Texts in Statistical Science. Chapman and Hall/CRC, 2007. ISBN: 142005967X,9781420059670.

[29] Simon Marlow and Bjarne Stroustrup. *Haskell 2010 Language Report*. URL: https://www.haskell.org/definition/haskell2010.pdf (visited on 06/02/2017).

[30] Conor McBride and Ross Paterson. "Applicative programming with effects". In: *Journal of functional programming* 18.1 (2008), pp. 1–13. URL: http://journals.cambridge.org/article_S0956796807006326 (visited on 06/04/2017).

[31] *ModelSpecification*. URL: http://www.openbugs.net/Manuals/ModelSpecification.html (visited on 06/05/2017).

[32] E. Moggi. "Computational Lambda-calculus and Monads". In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. Piscataway, NJ, USA: IEEE Press, 1989, pp. 14–23. ISBN: 978-0-8186-1954-0. URL: http://dl.acm.org/citation.cfm?id=77350.77353.

[33] *monet.js: Monadic types library for JavaScript*. original-date: 2013-10-09T20:21:53Z. June 2, 2017. URL: https://github.com/monet/monet.js.

[34] Bernt Oksendal. *Stochastic Differential Equations: An Introduction with Applications*. Google-Books-ID: XbDxCAAAQBAJ. Springer Science & Business Media, Apr. 17, 2013. 199 pp. ISBN: 978-3-662-02574-1.

[35] Atze van der Ploeg and Oleg Kiselyov. "Reflection without remorse: revealing a hidden sequence to speed up monadic reflection". In: *ACM SIGPLAN Notices*. Vol. 49. ACM, 2014, pp. 133–144. URL: http://dl.acm.org/citation.cfm?id=2633360 (visited on 06/04/2017).

[36] *pymc3: Probabilistic Programming in Python: Bayesian Modeling and Probabilistic Machine Learning with Theano*. original-date: 2009-05-05T09:43:50Z. June 4, 2017. URL: https://github.com/pymc-devs/pymc3.

[37] *ramda-fantasy: :ram::sparkles: Fantasy-Land compatible types for easy integration with Ramda.js*. original-date: 2014-12-02T19:08:53Z. June 5, 2017. URL: https://github.com/ramda/ramda-fantasy.

[38] Sheldon M Ross. *Stochastic processes. 1996*. 1996.

[39] *sanctuary: :see_no_evil: Refuge from unsafe JavaScript*. original-date: 2015-01-19T05:52:10Z. June 3, 2017. URL: https://github.com/sanctuary-js/sanctuary.

[40] Nathanael Schilling. "Monads in Haskell". In: (2014). URL: https://www21.in.tum.de/teaching/perlen/WS1415/unterlagen/Monads_in_Haskell.pdf (visited on 06/02/2017).

[41] Adam Scibior and Zoubin Ghahramani. "Modular construction of Bayesian inference algorithms". In: (). URL: https://pdfs.semanticscholar.org/e727/de3ebae5e4d66d96afdbf956047aab45c6c6.pdf (visited on 05/28/2017).

[42] Adam Ścibior, Zoubin Ghahramani, and Andrew D. Gordon. "Practical probabilistic programming with monads". In: ACM Press, 2015, pp. 165–176. ISBN: 978-1-4503-3808-0. DOI: 10.1145/2804302.2804317. URL: http://dl.acm.org/citation.cfm?doid=2804302.2804317 (visited on 06/03/2017).

[43] Adam Scibior et al. "Building inference algorithms from monad transformers". In: (). URL: https://pdfs.semanticscholar.org/76ad/0090bf4a076391fe2cc6d6029f79ebc66308.pdf (visited on 05/28/2017).

[44] *Sweet.js - Hygienic Macros for JavaScript*. URL: https://www.sweetjs.org/ (visited on 06/04/2017).

[45] Wouter Swierstra. "Data types à la carte". In: *Journal of functional programming* 18.4 (2008), pp. 423–436. URL: http://journals.cambridge.org/article_S0956796808006758 (visited on 05/29/2017).

[46] *TypeScript - JavaScript that scales.* URL: http://www.typescriptlang.org/ (visited on 06/04/2017).

[47] Janis Voigtländer. "Asymptotic improvement of computations over free monads". In: *International Conference on Mathematics of Program Construction*. Springer, 2008, pp. 388–403. URL: http://link.springer.com/chapter/10.1007/978-3-540-70594-9_20 (visited on 05/29/2017).

[48] Philip Wadler. "The essence of functional programming". In: *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1992, pp. 1–14. URL: http://dl.acm.org/citation.cfm?id=143169 (visited on 06/03/2017).

[49] Akhil Wali. *Mastering Clojure*. Google-Books-ID: oJrjCwAAQBAJ. Packt Publishing Ltd, Mar. 28, 2016. 267 pp. ISBN: 978-1-78588-205-0.

[50] David Wingate, Andreas Stuhlmueller, and Noah Goodman. "Lightweight implementations of probabilistic programming languages via transformational compilation". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 2011, pp. 770–778. URL: http://www.jmlr.org/proceedings/papers/v15/wingate11a/wingate11a.pdf (visited on 05/28/2017).

[51] Frank Wood, Jan Willem Meent, and Vikash Mansinghka. "A new approach to probabilistic programming inference". In: *Artificial Intelligence and Statistics*. 2014, pp. 1024–1032. URL: http://www.jmlr.org/proceedings/papers/v33/wood14.pdf (visited on 05/28/2017).

[52] Jeremy Yallop and Leo White. "Lightweight higher-kinded polymorphism". In: *International Symposium on Functional and Logic Programming*. Springer, 2014, pp. 119–135. URL: http://link.springer.com/chapter/10.1007/978-3-319-07151-0_8 (visited on 05/28/2017).

Appendix A

Appendix

A.1 Monad Transformer

```haskell
module Thesis.Transformer where
import Control.Monad.Trans.Maybe
import Control.Monad.State

pop :: MaybeT (State [a]) a
pop = do
     s <- lift get
     case s of
       []     -> MaybeT $ return Nothing
       (r:sx) -> do
          lift $ put sx
          return r

push :: a -> MaybeT (State [a]) ()
push x = do
    s <- lift get
    lift . put $ x:s
    return ()

program1 = pop >> pop >> pop
program2 = pop >> push 3 >> pop

result1 = runState (runMaybeT program2) [1, 2] == (Nothing, [])
result2 = runState (runMaybeT program2) [1, 2] == (Just 3, [2])
```

A.2 Free State

```haskell
module Thesis.FreeState where

-- Based on work from http://okmij.org/ftp/Computation/FreeState.hs

-- State with only functor definition
newtype State s a = State{runState :: s -> (a,s)}

instance Functor (State s) where
  fmap f (State g) = State (\st -> let (x, st') = g st
                                   in (f x, st'))

get :: State s s
get = State $ \s -> (s,s)

put :: s -> State s ()
put s = State $ \_ -> ((),s)

-- Free monad
data Free f r = Impure (f (Free f r)) | Pure r

instance Functor f => Functor (Free f) where
  fmap f (Pure x)   = Pure $ f x
  fmap f (Impure m) = Impure $ fmap (fmap f) m

-- Applicatives has not been covered but every monad is also an Applicative.
instance Functor f => Applicative (Free f) where
  pure = Pure
  Pure f <*> m   = fmap f m
  Impure f <*> m = Impure $ fmap (<*> m) f

instance Functor f => Monad (Free f) where
  return = Pure
  Pure a   >>= k = k a
  Impure m >>= k = Impure (fmap (>>= k) m)

liftF :: Functor f => f a -> Free f a
liftF = Impure . fmap Pure

-- Free State monad
type FState s = Free (State s)

getF :: FState s s
getF = liftF get

putF :: s -> FState s ()
putF = liftF . put

-- Interpreter
runFState :: FState s a -> s -> (a,s)
runFState (Pure x) s   = (x,s)
runFState (Impure m) s = let (m',s') = runState m s in runFState m' s'
```

```
test :: FState Int Int
test = do
  x <- getF
  putF 12
  return x

result = (10, 12) == runFState test 10
```

A.3 Tele

```
module Thesis.Tele where
import Control.Monad.Free
import Control.Monad.State
import Control.Monad.Writer

data TeletypeF a =
               PutLine String a
             | GetLine (String -> a)

instance Functor TeletypeF where
    fmap f (PutLine str x)  = PutLine str (f x)
    fmap f (GetLine      k) = GetLine (f . k)

getLine' :: Free TeletypeF String
getLine' = liftF $ GetLine id

putLine' :: String -> Free TeletypeF ()
putLine' l = liftF $ PutLine l ()

pop :: State [String] String
pop = state $ \(x:xs) -> (x, xs)

interpret :: Free TeletypeF a -> WriterT [String] (State [String]) a
interpret = iterM $ \x -> case x of
          PutLine l k -> tell [l] >> k
          GetLine k   -> (lift pop) >>= k

test = do
    l <- getLine'
    putLine' "output1"
    putLine' l

program = interpret test

expected = ((((),["output1","input1"]),["input2"]))
result = expected == runState (runWriterT program) ["input1", "input2"]
```

A.4 Random

```
module Thesis.Random where
import System.Random
import Control.Monad.State

uniform :: (RandomGen s, MonadState s m) => Double -> Double -> m Double
uniform a b = state $ randomR (a, b)

bernoulli :: (RandomGen s, MonadState s m) => Double -> m Bool
bernoulli p = (< p) <$> uniform 0 1
```

A.5 Rejection Sampling

```
module Thesis.Rejection where
import System.Random
import Control.Monad.State
import Control.Monad.Trans.Maybe
import Data.Maybe
import Thesis.Random (uniform, bernoulli)

condition :: (RandomGen s) => Bool -> MaybeT (State s) ()
condition = MaybeT . return . toMaybe
  where toMaybe True  = Just ()
        toMaybe False = Nothing

sample = flip evalState
sampleMany m = catMaybes $ sample (mkStdGen 1) $ replicateM 10 $ runMaybeT m

dist1 = do
     b <- uniform 0 2
     condition $ b < 0.5
     return b

result1 = sampleMany dist1

-- built in does this
dist2 = do
     b <- uniform 0 2
     guard $ b < 0.5
     uniform 1 2

result2 = sampleMany dist2
```

A.6 Free Model

```
module Thesis.Model where
import Control.Monad
import Control.Monad.Free
```

```haskell
import Control.Monad.State
import System.Random (mkStdGen, randomR)
import qualified Thesis.Random as R

data ModelF r =
    BernoulliF Double (Bool -> r)
  | UniformF Double Double (Double -> r)

instance Functor ModelF where
  fmap f (BernoulliF a k) = BernoulliF a (f . k)
  fmap f (UniformF a b k) = UniformF a b (f . k)

type Model = Free ModelF

uniform :: Double -> Double -> Model Double
uniform a b = liftF (UniformF a b id)

bernoulli :: Double -> Model Bool
bernoulli p = liftF (BernoulliF p id)

sample f g = runState f g

interpret (Pure x) s = (x, s)
interpret (Free m) s = case m of
  (BernoulliF p f) -> let (m', s') = sample (R.bernoulli p) s
                      in interpret (f m') s'
  (UniformF a b f) -> let (m', s') = sample (R.uniform a b) s
                      in interpret (f m') s'

-- Example
dist = do
  a <- uniform 0 2
  a' <- uniform 0 2
  b <- bernoulli 0.2
  if b then return a else return a'


geo p = do
    a <- bernoulli p
    if a then return 1
    else do
      g <- geo p
      return (g + 1)

distResult = interpret dist (mkStdGen 1)
geoResult  = interpret (geo 0.1) (mkStdGen 1)
```