Scalable Bayesian Time Series Modelling for Streaming Data

Jonathan Law

Thesis submitted for the degree of
Doctor of Philosophy

School of Mathematics, Statistics & Physics
Newcastle University
Newcastle upon Tyne
United Kingdom

October 2019

Acknowledgements

Firstly, I would like to thank my supervisor Darren Wilkinson for his help and support and insightful discussions when researching and writing this thesis. His passion and deep understanding of statistical modelling and programming is inspiring.

I would like to thank the academic members of staff in the Cloud Computing for Big Data CDT, the School of Mathematics, Statistics and Physics and the School of Computing for many informative and inspiring talks. Specific thanks go to Paul Watson, Matt Forshaw, Oonagh McGee, Jen Wood and Sarah Heaps. Finally, thanks to my fellow PhD students in the Cloud Computing for Big Data CDT for their friendship and support.

Abstract

Ubiquitous cheap processing power and reduced storage costs have led to increased deployment of connected devices used to collect and store information about their surroundings. Examples include environmental sensors used to measure pollution levels and temperature, or vibration sensors deployed on machinery to detect faults. This data is often streamed in real time to cloud services and used to make decisions such as when to perform maintenance on critical machinery, and monitor systems, such as how interventions to reduce pollution are performing.

The data recorded at these sensors is unbounded, heterogeneous and often inaccurate, recorded with different sampling frequencies, and often on irregular time grids. Connection problems or hardware faults can cause information to be missing for days at a time. Additionally, multiple co-located sensors can report different readings for the same process. A flexible class of dynamic models can be used to ameliorate these issues and used to smooth and interpolate the data.

Irregularly observed time series can be conveniently modelled using state space models with a continuous time latent-state represented by diffusion processes. In order to model the wide array of different environmental sensors the observation distributions of these dynamic models are flexible, in all cases particle filtering methods can be used for inference and in some cases the exact Kalman filter can be used. The models along with a binary composition operator form a semigroup, making model composition and reuse straightforward. Heteroskedastic time series are accounted for by using a factor structure to model a full-rank time-dependent system noise matrix for the dynamic models which can account for changes in variance and the correlation structure between each time series in a multivariate model. Finally, to model multiple nearby sensors a dynamic model is used to model a time-dependent common mean and a time-invariant Gaussian process can account for the spatial variation between the sensors.

Functional programming in Scala is used to implement these time series models. Functional programming provides a unified principled API (application programming interface) for interacting with different collection types using higher order functions. This, combined with the type-class pattern, makes it possible to write inference algorithms once and deploy them locally using serial collections and later on unbounded time series data using libraries such as Akka streams using techniques from functional reactive programming.

Contents

List of Figures

List of Tables

Chapter 1

Time Series and Bayesian Inference

The main contribution of this thesis is to unite principled Bayesian statistical modelling for time series data with state of the art functional programming, illustrated here using the Scala programming language [Odersky et al., 2004]. This unification enables flexible model building and scalable statistical inference algorithms. Three Scala libraries have been developed which implement the ideas discussed in this thesis, the first is for performing Bayesian inference in Dynamic State Space Models[1], discussed in chapters 4, 5 and 7. The second[2] is a companion package to a publication [Law and Wilkinson, 2017], which considers a class of composable state space models with diffusion processes representing the continuous time latent-state. The third is a library for spatial modelling using Gaussian Processes[3].

The time series data considered comes from the Urban Observatory [James et al., 2014a], a large sensor network in the North East of England. The data is open and freely available for use by both industry and researchers[4]. However, since the data is often missing or inaccurate, space-time models can be applied to smooth readings, provide forecasts and interpolate missing data. The models developed in this thesis aim to model the range of sensor data in the Urban Observatory. The models can be easily composed, re-used and built upon. Additionally the models and inference schemes can be deployed on unbounded streaming sensor data.

This chapter provides a concise introduction to Bayesian inference and time series analysis with a focus on computational inference, using simple examples. This helps to establish notation used in the remainder of this thesis and gives a grounding for some of

[1]https://git.io/dlm
[2]https://git.io/statespace
[3]https://git.io/gaussianprocesses
[4]http://uoweb1.ncl.ac.uk/

the more advanced tools and techniques presented in later chapters. For a comprehensive overview of Bayesian statistics see Gelman et al. [2013]. For an overview of modern Bayesian inference in time series see Prado and West [2010]. Gamerman and Lopes [2006] provides an in-depth guide to Markov chain Monte Carlo methods for Bayesian statistics.

Chapter 2 highlights important aspects of programming languages, with an emphasis on functional programming which can be used to build expressive APIs (application programming interfaces) for statistical model building and inference. This chapter introduces a reactive streams implementation in Scala, Akka Streams. Reactive programming is natural for analysis of unbounded time series data. Chapter 3 introduces probabilistic programming languages as embedded domain specific languages (DSL) in functional programming languages using the tools introduced in Chapter 2.

Chapter 4 is the first chapter with a focus on time series modelling and introduces Gaussian dynamic linear models (DLMs) including specification, composition and inference for univariate and multivariate models, culminating in a multivariate DLM example for a pollution sensor in the Urban Observatory. In addition, a continuous time extension to the DLM using a diffusion process to represent the latent-state is considered in section 4.7.

Chapter 5 considers non-Gaussian state space models including filtering and parameter inference. The chapter concludes with three case studies, Student's-t distributed models for Nitrogen Monoxide (NO) and temperature showing improved model diagnostics compared to the Gaussian DLM example in chapter 4 and a count model for regularly observed traffic data.

Chapter 6 is adapted from a published paper [Law and Wilkinson, 2017] and considers irregularly observed time series data with a composable latent-state represented by multiple diffusion processes. This chapter emphasizes the streaming nature of modern time series data and how inference can be performed online for live unbounded data using streaming data frameworks such as Akka streams.

Chapter 7 introduces heteroskedastic time series and factor models. Using the same model building API as chapters 4 and 5, multivariate DLMs with non-diagonal, time dependent system and observation noise matrices can be built in order to model multivariate time series with dependent latent-state and observation noise variances.

Finally, Chapter 8 is concerned with Gaussian processes (GP) for spatial modelling of multiple nearby sensors. The GP is used to interpolate temperature data in an example from the Urban Observatory.

Chapter 9 considers the contribution of this thesis and explores the possibility of further work in the domain of Bayesian time series modelling and functional programming.

1.1 Introduction to Bayesian Inference

There are several paradigms of statistical inference commonly used in modern statistics and machine learning. They include classical (or frequentist), Bayesian and likelihood-based. This thesis is solely concerned with Bayesian statistics.

Bayesian statistics is concerned with expressing uncertainty using probability. Consider reasoning about an unknown outcome such as the flip of a coin, result of a football match or a scientific experiment. Before the unknown event has occurred an individual may have an opinion on the outcome, for instance the probability of heads a coin flip is thought to be $\frac{1}{2}$. Each individual can have different beliefs, for instance a passionate football fan may have extra confidence in his or her own team and believe they can win even if they are not the bookmakers favourites. These beliefs can be updated in the presence of new information, for instance the star striker in the football example can pull up injured leading up to the game and the passionate fan starts to lose hope. Bayesian inference formalises this belief updating by combining prior beliefs with data. Prior beliefs are encoded using probability distributions.

In order to understand Bayesian inference, it is important to first consider some rules of probability. A probability is a number between 0 and 1 which represents whether an event will occur, an event that can not occur will have probability 0 whereas an event which is certain to occur is said to have probability 1.

Consider an example of flipping a coin. If the coin is fair then $Pr(\text{H}) = 0.5$ and $Pr(\text{T}) = 1 - P(\text{H}) = 0.5$ since $Pr(\neg A) = 1 - Pr(A)$ where A is any event and $\neg A$ is the complement of A. The outcome of a coin flip is called an event and $\{H, T\}$ is the sample space. The events are mutually exclusive (formally $Pr(H \cap T) = 0$) which means $Pr(H \cup T) = P(H) + P(T)$.

Consider two independent events, for instance flipping two fair coins, then the probability of landing heads on both coins is given by the product of landing heads on each coin. That is $Pr(H_1 \cap H_2) = Pr(H_1)Pr(H_2) = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$. However, if the coins are flipped at different times and it is known that the first coin has landed heads then the probability of landing two heads is changed to $Pr(H_2|H_1) = Pr(H_1 \cap H_2)/Pr(H_2) = \frac{1}{4}/\frac{1}{2} = \frac{1}{2}$ by the law of conditional probability

Definition. The law of *conditional probability* states $Pr(A|B) = \frac{Pr(A \cap B)}{Pr(B)}$.

Theorem 1.1. Bayes' theorem *states that* $Pr(A|B) = \frac{Pr(A)Pr(B|A)}{Pr(B)}$.

Proof. Bayes' theorem can be derived from the definition of conditional probability 1.1.

Given two events A, B in the set of events, then the probability of both events occurring is commutative $Pr(A \cap B) = Pr(B \cap A)$ and

$$\begin{aligned} Pr(A|B)Pr(B) = Pr(A \cap B) &= Pr(B \cap A) = Pr(B|A)Pr(A) \\ Pr(A|B) &= \frac{Pr(A)Pr(B|A)}{P(A)}. \end{aligned}$$

□

In the context of Bayesian inference; the quantity $Pr(B|A)$ is known as the likelihood and $Pr(A)$ is known as the prior. This equation is often written $Pr(A|B) \propto Pr(A)Pr(B|A)$, read as the posterior is proportional to the prior times the likelihood. Informally, this allows us to turn conditional probabilities around.

A parametric model is a statistical model consisting of a probability distribution controlled by finitely many parameters ψ, the model is believed to explain data collected from an experiment. A parametric statistical model can be thought of as a family of statistical models which express different relationships based on the value of the parameters. In the Bayesian paradigm, parameters are considered uncertain and follow a probability distribution. Before the data are observed, a parameter prior distribution $p(\psi)$ is specified and after the data, Y, are observed a parameter posterior distribution can be identified as $p(\psi|Y)$.

As an example, consider a (possibly biased) coin flip experiment. The parameter of interest is the probability of heads p_h. A Beta distribution is chosen for the prior of p_h, $p(p_h) = \mathcal{B}(\alpha, \beta)$. The Beta distribution has support between 0 and 1, which is appropriate for a probability. The likelihood of a coin flip is Bernoulli, however the coin should be flipped several times in order to learn the parameter p_h. The distribution for n independent Bernoulli trials is the Binomial distribution, hence the likelihood can be written as $\text{Bin}(Y; n, p_h)$. The coin is flipped $n = 10$ times and the results are displayed below:

$$\{H, H, T, H, H, H, T, T, H, T\}$$

Then $Y = 6$, and it remains to determine the posterior distribution of the parameter p_h representing the probability of obtaining heads. Applying Bayes' theorem

$$\begin{aligned} p(p_h|Y) &= \frac{p(p_h)\text{Bin}(Y; n, p_h)}{\int_0^1 p(Y|p_h)dp_h} \\ &\propto p_h^{\alpha-1}(1-p_h)^{\beta-1}\binom{n}{Y}p_h^Y(1-p_h)^{n-Y} \\ &= p_h^{Y+\alpha-1}(1-p_h)^{n-Y+\beta-1}. \end{aligned}$$

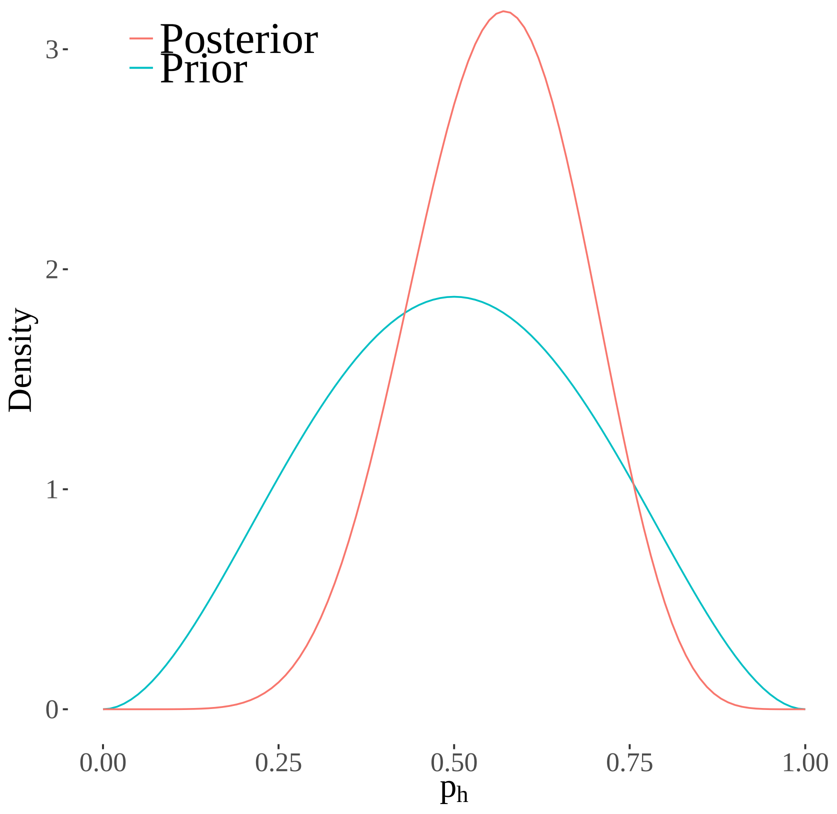


Figure 1.1: The prior and posterior distribution for the coin flip example, the posterior is more concentrated than the prior and seems to hint that the coin is biased towards heads.

The posterior distribution is a Beta distribution which we denote by $\mathcal{B}(Y+\alpha, n-Y+\beta)$.

In order to complete this example, suitable values should be chosen for the hyper parameters of the prior distribution, eg. $\alpha = 3$ and $\beta = 3$. Then fill in the values $Y = 6$ and $n = 10$. Figure 1.1 shows the prior and posterior distributions for the parameter p_h. The posterior distribution is more concentrated than the prior and seems to show that the coin is biased towards heads. If more trials were conducted, the parameter posterior distribution tends to be even more concentrated.

If it was decided that more trials were needed to determine the fairness of the coin, then the results of this preliminary experiment do not need to be discarded. The prior distribution of p_h can be set to be the posterior distribution found in the initial 10 coin flips since the likelihood factorises over flips.

Parametric statistical modelling in the Bayesian paradigm follows the same steps outlined above in the coin flip experiment. First a model is specified for the data, then a prior distribution is chosen for the parameters, these can be based on expert judgement or from the outcome of previous similar experiments. The likelihood of the parameters given the observed data is written (or calculated in the case of an intractable likelihood) and Bayes' theorem is applied to determine the posterior distribution of the parameters given the observed data.

The coin flip experiment is particularly simple, the Beta prior distribution is conjugate to the Binomial likelihood meaning the posterior distribution can be calculated in closed form and is the same distribution as the prior. However, this is the general framework for reasoning with uncertainty using Bayes theorem. In practice, posterior distributions are often intractable and computational techniques are used to evaluate summary statistics of interest from the posterior distribution, such as the mean and posterior probability intervals. Section 1.3 introduces commonly used computational techniques for more general intractable problems.

1.2 Time Series Modelling

A time series is a sequence of observations indexed by time. In statistical analysis of time series, a single realisation of a time series at time t is a random variable and can be written as $Y_t \sim p(Y_t|Y_{1:t-1} = y_{1:t-1})$ where p is a probability distribution of random variables and $Y_{1:t-1}$ represents the set containing the past $t-1$ observations of the time series $Y_{1:t-1} = \{Y_1, Y_2 \dots, Y_{t-1}\}$. Some examples of time series include stock prices, weather systems and sensor data such as those found in the Urban Observatory [James et al., 2014a].

Time series modelling is concerned with forecasting, interpolation and inference for model parameters. If the joint probability model for an observed time series $y_{1:N} = \{y_1, y_2, \dots, y_N\}$ can be written as $p(Y_{1:n})$, for any $n \geq 1$ then a forecast from this model can be determined using the law of conditional probability 1.1

$$p(Y_{N+1}|Y_{1:N} = y_{1:N}) = \frac{p(Y_{1:N+1})}{p(Y_{1:N})} \tag{1.1}$$

Specifying such a model is challenging in general. Forecasts and interpolation can be facilitated by the use of a parametric statistical model. In a parametric model, data collected until time $t = N$ is used to determine the posterior distribution of the parameters of a model $p(\psi|Y_{1:N})$, then the parameter posterior distribution can be used to determine the behaviour and relationships present in the data and perform forecasting or interpolation.

Bayesian time series modelling correctly quantifies forecast uncertainty by taking into account parameter uncertainty. The Bayesian paradigm allows prior knowledge to be incorporated formally at the model fitting stage and is natural for sequential learning as the posterior at time t becomes the prior at time $t+1$. The distributions of interest in statistical time series modelling include the posterior distribution of the parameters $p(\psi|Y_{1:N})$, one-step and k-step forecast distributions and smoothing distributions used for interpolation.

To derive the forecast distributions consider a time series model which has a joint distribution of the observations and parameters $p(Y_{1:t}, \psi)$, then a one-step forecast is the marginal distribution:

$$p(Y_{t+1}|Y_{1:t} = y_{1:t}) = \int_{\psi} p(Y_{t+1}|Y_{1:t} = y_{1:t}, \psi)p(\psi|y_{1:t})\mathrm{d}\psi. \tag{1.2}$$

This marginal distribution is the integral over the parameter posterior distribution determined using the currently available observed data $y_{1:t}$. Similarly a k-step forecast can be produced:

$$p(Y_{t+k}|Y_{1:t} = y_{1:t}) = \int_{Y}\int_{\psi} \prod_{i=1}^{k} p(Y_{t+i}|Y_{1:t} = y_{1:t}, \psi)p(\psi|y_{1:t})\mathrm{d}\psi\mathrm{d}Y_{t+1:t+k}$$

An example of a time series model is the autoregressive process of order $p = 1$, often written AR(1)

$$Y_t = \mu + \phi(Y_{t-1} - \mu) + \varepsilon_t, \quad \varepsilon_t \sim \mathcal{N}(0, \sigma^2). \tag{1.3}$$

The innovations, ε_t are independent and Normally distributed. The autoregressive model is so called because it regresses upon earlier observations. The AR(p) process is dependent on only the previous p observations of the series. It is said the AR(p) is a Markov chain of order p.

Definition. The series $Y_{1:t}$ has the first order *Markov* property if $p(Y_t|Y_{t-1} = y_{t-1}, \dots, Y_0 = y_0) = p(Y_t|Y_{t-1} = y_{t-1})$

In other words the next observation is conditionally independent of the past values of the series given the current value.

Definition. A time series is said to be Markov chain of order p if $p(Y_t|Y_{t-1} = y_{t-1}, \dots, Y_0 = y_0) = p(Y_t|Y_{t-1} = y_{t-1}, \dots, Y_{t-p} = y_{t-p})$

The Markov transition kernel of the AR(1) process is:

$$p(Y_t|Y_{1:t-1}) = p(Y_t|Y_{t-1}) = \mathcal{N}\left(\mu + \phi(Y_{t-1} - \mu), \sigma^2\right). \tag{1.4}$$

The parameters of the AR(1) process are $\psi = \{\mu, \phi, \sigma\}$, the AR(1) process is stationary if the absolute value of the autoregressive parameter is less than one, ie $|\phi| < 1$. In order to fit the AR model to real-world data it remains to determine the posterior distribution of the parameters conditional upon the observed data, $p(\psi|y_{1:N})$. The posterior distribution for this model (and most real world problems) is not analytically tractable, and so computational inference algorithms are used as introduced in section 1.3. The problem of inferring the posterior distribution of an AR(1) model is considered in Section 1.3.3. Once the parameter posterior distribution of the parameters of the AR(1) model is known then forecast distributions can be determined.

1.2.1 State Space Models

The previous example for the AR(1) process considers a parametric statistical model in which realisations of the time series depend only on static parameters, ψ. In a *state space model* the parameters depend on a latent stochastic process which is not observed. Each observation is then considered conditionally independent given the corresponding realisation of the latent-state. State-space models are a flexible class of models and can be used to model univariate and multivariate times series with both non-stationary and non-linear relationships. In addition, complex models can be built from simple components incorporating seasonality and trends. Models can be considered in continuous or discrete time, thus simplifying the calculations for certain problem types. For an overview of state space modelling and filtering, see Doucet et al. [2001].

The evolution of the latent-state is often considered to be a first order Markov process governed by a transition kernel:

$$p(\theta_t|\theta_{1:t-1}) = p(\theta_t|\theta_{t-1}). \tag{1.5}$$

An observation is then related to a corresponding latent-state using an observation distribution:

$$p(Y_t|\theta_t, y_{1:t-1}) = p(Y_t|\theta_t). \tag{1.6}$$

The observation distribution can be chosen to be appropriate for the time series of interest and represents the independent measurement error of the observed process $Y_{1:N}$. The observations are conditionally independent given the latent state. A problem of interest is to determine the filtering distribution, which is the distribution of the latent-state given the observations up to time t:

$$p(\theta_t|y_{1:t}) = \frac{p(\theta_t|y_{1:t-1})p(y_t|\theta_t, y_{1:t-1})}{p(y_t|y_{1:t-1})}$$

This is an application of Bayes' theorem: the prior distribution is the advanced state at time t given the data up to time $t-1$, $p(\theta_t|y_{1:t-1})$, also known as the one step ahead predictive density of the state. The likelihood is the conditional likelihood of an observation, $p(y_t|\theta_t) = p(y_t|\theta_t, y_{1:t-1})$ since y_t is conditionally independent of $y_{1:t-1}$ given θ_t. The one step ahead predictive density of the state can be calculated as:

$$p(\theta_t|y_{1:t-1}) = \int p(\theta_t|\theta_{t-1})p(\theta_{t-1}|y_{1:t-1})\mathrm{d}\theta_{t-1}$$

This allows the one-step ahead predictive density, $p(y_t|y_{t-1})$ to be computed:

$$p(y_t|y_{t-1}) = \int p(y_t|\theta_t)p(\theta_t|y_{1:t-1})\mathrm{d}\theta_t$$

This forms the basis for filtering and forecasting in a general state space model. For the case when the model is specified by Gaussian transition and observation distributions, filtering is available in closed form via the Kalman filter which is introduced in section 4.2. In more general state space models where the observation and transition distributions are non-Gaussian, the particle filter introduced in section 1.3.5 can be used to determine the filtering distribution.

In addition to learning the posterior distribution of the latent-state, the posterior distribution of the static parameters ψ associated with the observation and transition distributions can be determined using appropriate MCMC algorithms. The main MCMC algorithms used in this thesis are introduced in section 1.3.

1.3 Markov Chain Monte Carlo

It is often the case that the posterior distribution of a specified model is not available in closed from. In this case sampling based methods can be used to approximate the posterior distribution. This section introduces the building blocks of the inference algorithms used throughout this thesis with simple illustrative examples. The inference methods include Markov chain Monte Carlo (MCMC) and sequential Monte Carlo (SMC) methods. For a comprehensive review of MCMC see Brooks et al. [2011] and SMC see Doucet et al. [2001].

Real world models often have many parameters and hence an algorithm which scales well to high dimensions is required. Markov chain Monte Carlo (MCMC) methods form such a class of scalable algorithms.

MCMC methods work by repeatedly sampling from a Markov transition kernel until the Markov chain has converged to a unique stationary distribution coinciding with the target posterior distribution. In order to verify that the Markov chain converges to the target posterior distribution it must be proven that the stationary distribution of the Markov chain exists and is unique.

Sometimes an MCMC algorithm can be initialised far away from an area of high posterior density. This is no problem in theory and eventually the Markov chain will sample from its stationary distribution as the number of iterations tends to infinity. The initial iterations are sometimes referred to as burn in (or warm-up) and can be discarded before calculating summaries of the posterior distribution.

The nature of MCMC samplers means that the samples are autocorrelated, meaning samples from the distribution are not independent. Thinning can be applied by discarding values such that only every n$^{\text{th}}$ value of the chain is retained. Thinning helps to reduce auto-correlation but can mean the MCMC algorithm is required to run for longer in order to calculate summary statistics of interest from the posterior distribution.

1.3.1 Pseudo Random Number Generators

All statistical computer simulation algorithms rely on pseudo random number generators (PRNGs). PRNGs are not actually random number generators, but rely on a deterministic transformation of a random number state in order to produce numbers which appear random. This state is mutated each time another random number is requested. It is important to know how the pseudo random numbers are distributed in order to be able to simulate from other arbitrary distributions. For the algorithms which follow, it is assumed that a good quality PRNG is available which produces uniform random numbers between 0 and 1. For more information on PRNGs see Ripley [1987], note that special considerations must be made for simulating random numbers in parallel [Wilkinson, 2005].

1.3.2 Metropolis-Hastings

The Metropolis-Hastings algorithm is an MCMC algorithm used to simulate values from a target distribution which can not be sampled from directly [Metropolis et al., 1953, Hastings, 1970]. The Metropolis-Hastings (MH) algorithm converges to a unique stationary distribution which is the target distribution.

The MH algorithm is straightforward to program and implement. Suppose the most recently accepted parameter value is ψ, the likelihood is specified as $\pi(Y|\psi)$, the prior is $p(\psi)$ and the proposal distribution for the parameters is $q(\psi^\star|\psi)$. First a new set of parameters are simulated from the proposal distribution then the un-normalised posterior $\pi(Y|\psi^\star)p(\psi^\star)$ is evaluated at the value of the newly sampled parameters and the proposed parameter is accepted with probability according to the Metropolis choice

$$A(\psi^\star|\psi) = \min\left(1, \frac{p(\psi^\star)\pi(Y|\psi^\star)q(\psi|\psi^\star)}{p(\psi)\pi(Y|\psi)q(\psi^\star|\psi)}\right). \tag{1.7}$$

This process forms a Markov chain with stationary distribution equal to the posterior distribution, $p(\psi|Y)$. A common choice for the proposal distribution is a multivariate Normal distribution centred at the previously accepted value $q(\psi^\star|\psi) = \text{MVN}(\psi^\star|\psi, \varLambda)$. $\varLambda$ is a covariance matrix which is a tuning parameter. The value of $\varLambda$ is chosen such that the proportion of accepted moves in the MH sampler is approximately $\frac{1}{3}$ [Sherlock et al., 2009]. If the proportion of accepted moves is too high, then the sampler is inefficient, produces highly auto-correlated samples and is not exploring the full posterior distribution, however the algorithm is guaranteed to converge to the correct posterior distributions if more draws from the Markov chain are simulated. If the proportion of accepted moves is very low then the sampler does not have many unique samples from the posterior distribution.

The MH algorithm can be simplified by choosing a proposal distribution symmetric about the previously accepted value in the Markov chain, the acceptance ratio simplifies to:

$$A(\psi^\star|\psi) = \min\left(1, \frac{p(\psi^\star)\pi(Y|\psi^\star)}{p(\psi)\pi(Y|\psi)}\right) \tag{1.8}$$

An algorithm for simulating M iterations from the Metropolis-Hastings Kernel is summarised in algorithm 1.

Algorithm 1: Metropolis-Hastings algorithm

```
   Result: Return $\psi_i, i = 1, \dots, M$
 1 Given observed data $Y$, a proposal distribution $q$ Initialise $\psi_1 \sim p(\psi)$;
 2 for $i$ in 2:M do
 3     Propose $\psi^\star \sim q(\psi^\star|\psi_{i-1})$;
 4     Calculate $A(\psi^\star|\psi) = \frac{p(\psi^\star)\pi(Y|\psi^\star)q(\psi_{i-1}|\psi^\star)}{p(\psi_{i-1})\pi(Y|\psi_{i-1})q(\psi^\star|\psi_{i-1})}$;
 5     Sample $u \sim U[0,1]$;
 6     if $u < A(\psi^\star|\psi)$ then
 7         Set $\psi_i = \psi^\star$;
 8     else
 9         Set $\psi_i = \psi_{i-1}$;
10 end
```

Example: Coin Flip

The Metropolis-Hastings algorithm can be applied to the coin flip example, the proposal distribution should be chosen carefully so that values of p_h are not proposed outside of the acceptable range of $(0, 1)$. The proposal distribution is chosen to be a beta distribution

$$q(p_h^\star|p_h) = \mathcal{B}(p_h^\star|\lambda p_h + \tau, \lambda(1 - p_h) + \tau).$$

λ and τ are tuning parameters, λ controls the variance of the distribution, τ is very small and prevents the sampler getting stuck around zero or one. The mean of the $\mathcal{B}(\alpha, \beta)$ distribution is $\beta/(\alpha + \beta)$, hence the mean of the distribution is $\frac{\lambda p_h + \tau}{\lambda + 2\tau} \approx p_h$. The variance is $\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)} \approx \frac{p_h}{\lambda}$. Alternatively, the parameter p_h can be transformed using, for example, a logistic function and the parameter can be proposed using a random walk.

Figure 1.2 shows the results of two chains each with 10,000 draws, the chain on the left was run with $\tau = 0.01$ and $\lambda = 1000$ and the proportion of accepted values is too high

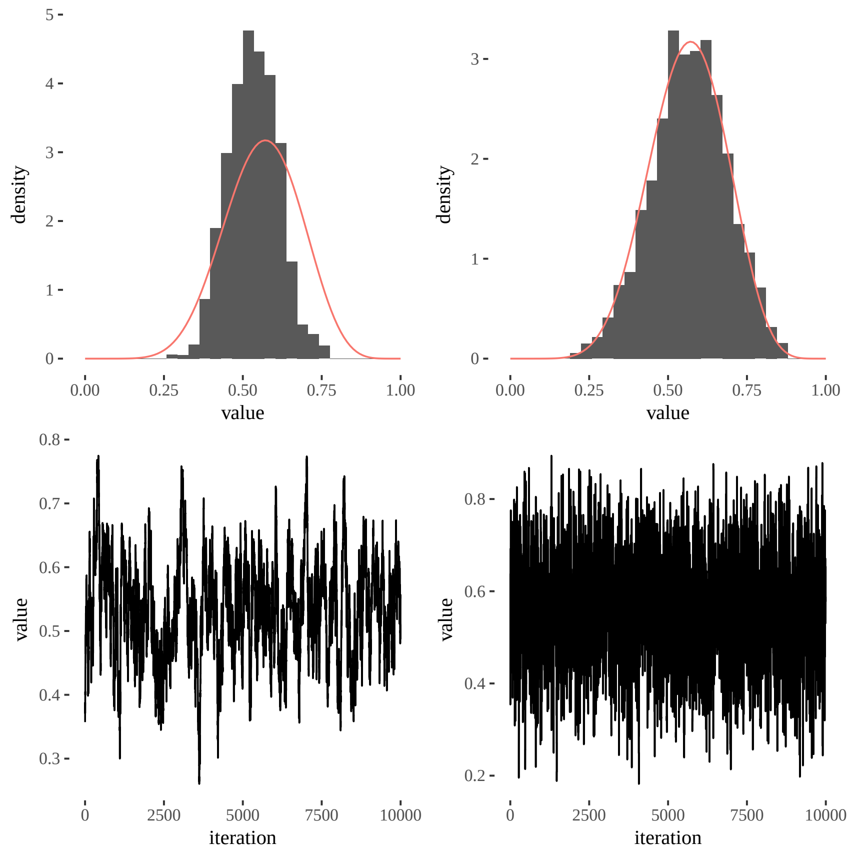


Figure 1.2: Traceplots and histograms for the Metropolis-Hastings algorithm applied to the coin flip example. A sampler where the proportion of accepted values is too high (0.92), (left). A sampler where the proportion of accepted is close to optimal (0.33), (right).

(0.92) (this is because the variance of the proposal distribution is too small) and does not approximate the posterior accurately. The chain depicted on the right with $\tau = 0.01$ and $\lambda = 2$ has 0.33 proportion of accepted moves and the density is very close to the analytic posterior distribution overlaid in red.

Tuning Metropolis algorithms in high dimensions can be challenging and is typically a manual process. However, there are adaptive algorithms which adjust the proposal distribution as the chain is running to target an optimal proportion of accepted values. If not done carefully, this can affect the stationary distribution of the Markov chain resulting in convergence to the incorrect posterior distribution [Andrieu and Thoms, 2008].

1.3.3 Gibbs Sampling

Gibbs Sampling is an MCMC algorithm which utilises full conditional distributions which are semi-conjugate [Geman and Geman, 1987]. The Gibbs sampler is a special case of the Metropolis-Hastings algorithm where the acceptance ratio always evaluates to one and hence the "proposed" value is always accepted. The Metropolis-Hastings algorithm is more general and allows for more flexible prior specifications, however it is often difficult to tune for high dimensional problems. When the MH algorithm is running efficiently, it rejects most proposed moves. Gibbs sampling does not have to reject moves as samples are drawn from the exact conditional distributions.

Gibbs sampling proceeds by constructing the full-conditional distributions. If the parameter set is m-dimensional, $\boldsymbol{\psi} = \{\psi_1, \dots, \psi_m\}$, then it is often possible to utilise conjugate structure in the conditional distributions of the parameters. Full conditional distributions of the form $p(\psi_i|\psi_{-i}, Y)$, $i = 1, \dots, m$ are calculated, where ψ_{-i} represents the parameter set without the i^{th} element. The full conditional distributions are designed to be conjugate to the prior distribution and hence straightforward to sample from, though this does limit the choice of prior distribution. Algorithm 2 shows a general forward-sweep Gibbs sampling algorithm.

Algorithm 2: Forward-sweep Gibbs Sampling algorithm

```
  Result: Return $\psi^{(i)}, i = 1, \dots, M$
1 Given observed data $Y$;
2 $\psi$ is a set of $m$ parameters;
3 Sample $\psi^{(1)}$ from the prior distribution $p(\psi)$;
4 for $i$ in 2:M do
5     for $j$ in 1:m do
6         Sample from the full conditional distribution for the $j^{\text{th}}$ parameter
          $\psi_j^{(i)} \sim p(\psi_j^{(i)}|\psi_1^{(i)}, \dots, \psi_{j-1}^{(i)}, \psi_{j+1}^{(i-1)}, \dots, \psi_m^{(i-1)}, Y)$;
7     end
8 end
```

Example: AR Process

Consider an autoregressive process of order one

$$\alpha_t = \mu + \phi(\alpha_{t-1} - \mu) + \eta_t, \quad \eta_t \sim \mathcal{N}(0, \sigma^2). \tag{1.9}$$

The static parameters of the AR(1) process are $\psi = \{\phi, \mu, \sigma\}$. The likelihood of N observations of this time series process can be written as

$$p(\alpha_{0:N}|\psi) = \mathcal{N}\left(\alpha_0|\mu, \frac{\sigma^2}{1 - \phi^2}\right) \prod_{t=1}^{N} \mathcal{N}(\alpha_t|\mu + \phi(\alpha_{t-1} - \mu), \sigma^2). \tag{1.10}$$

If the prior distribution of the autoregressive parameter ϕ is chosen to be Normal with mean μ_ϕ, and variance σ_ϕ^2 then the full conditional can be derived as follows:

$$
\begin{aligned}
p(\phi|\alpha_{0:N}, \sigma, \mu) &= p(\phi)p(\alpha_0)\prod_{t=1}^{N} p(\alpha_t|\alpha_{1-t}, \psi) \\
&= \mathcal{N}(\phi; \mu_\phi, \sigma_\phi^2)p(\alpha_0)\prod_{t=1}^{N} \mathcal{N}(\mu + \phi(\alpha_{t-1} - \mu), \sigma) \\
&= \mathcal{N}\left(\frac{\frac{\mu_\phi}{\sigma_\phi^2} + \frac{1}{\sigma^2}\sum_{i=1}^{N}(\alpha_i - \mu)(\alpha_{i-1} - \mu)}{\frac{1}{\sigma_\phi^2} + \frac{1}{\sigma^2}\sum_{i=1}^{N}(\alpha_i - \mu)^2}, \frac{1}{\frac{1}{\sigma_\phi^2} + \frac{1}{\sigma^2}\sum_{i=1}^{N}(\alpha_i - \mu)^2}\right)
\end{aligned} \tag{1.11}
$$

ignoring the contribution from the initial distribution, $p(\alpha_0)$. Conditionally conjugate prior distributions are specified for the remaining static parameters in the model

$$
\begin{aligned}
\mu &\sim \mathcal{N}(\mu_\mu, \sigma_\mu^2), \\
\sigma &\sim \text{Inv-Gamma}(\alpha_\sigma, \beta_\sigma).
\end{aligned} \tag{1.12}
$$

Then the remaining full conditional distributions can be written as

$$
\begin{aligned}
p(\mu|\alpha_{0:N}, \phi, \sigma) &= \mathcal{N}\left(\frac{\frac{\mu_\mu}{\sigma_\mu^2} + \frac{1-\phi}{\sigma^2}\sum_{i=2}^{N}(\alpha_i - \phi\alpha_{i-1})}{\frac{1}{\sigma_\mu^2} + \frac{(n-1)(1-\phi)^2}{\sigma^2}}, \frac{1}{\frac{1}{\sigma_\mu^2} + \frac{(n-1)(1-\phi)^2}{\sigma^2}}\right), \\
p(\sigma^2|\alpha_{0:N}, \phi, \mu) &= \text{Inv-Gamma}\left(\alpha_\sigma + \frac{N}{2}, \beta_\sigma + \frac{1}{2}\sum_{i=1}^{N}((\alpha_i - \mu) - \phi(\alpha_{i-1} - \mu))^2\right).
\end{aligned} \tag{1.13}
$$

The full conditional distributions are simulated from conditional on the most recently sampled parameter values to complete one iteration of the Gibbs sampler for the AR(1) model. In this example simulated observations from an AR(1) process with known parameter values is used. In order to simulate realisations from an AR(1) process, firstly the value of α_0 is simulated from the stationary distribution $p(\alpha_0) = \mathcal{N}(\mu, \sigma^2/(1 - \phi^2))$, then 1,000 observations are sampled using the Markov transition kernel for the AR(1) process in equation (1.4). Figure 1.3 shows the diagnostic plots for the posterior distributions of the AR(1) model parameters after running the Gibbs sampling scheme for this synthetic data set.

1.3.4 Hamiltonian Monte Carlo

Hamiltonian Monte Carlo (HMC) methods use first order derivatives of the log-posterior in order to target areas of high posterior density; thus, avoiding the random walk behaviour of the Metropolis-Hastings algorithm [Duane et al., 1987] while retaining the flexible choice

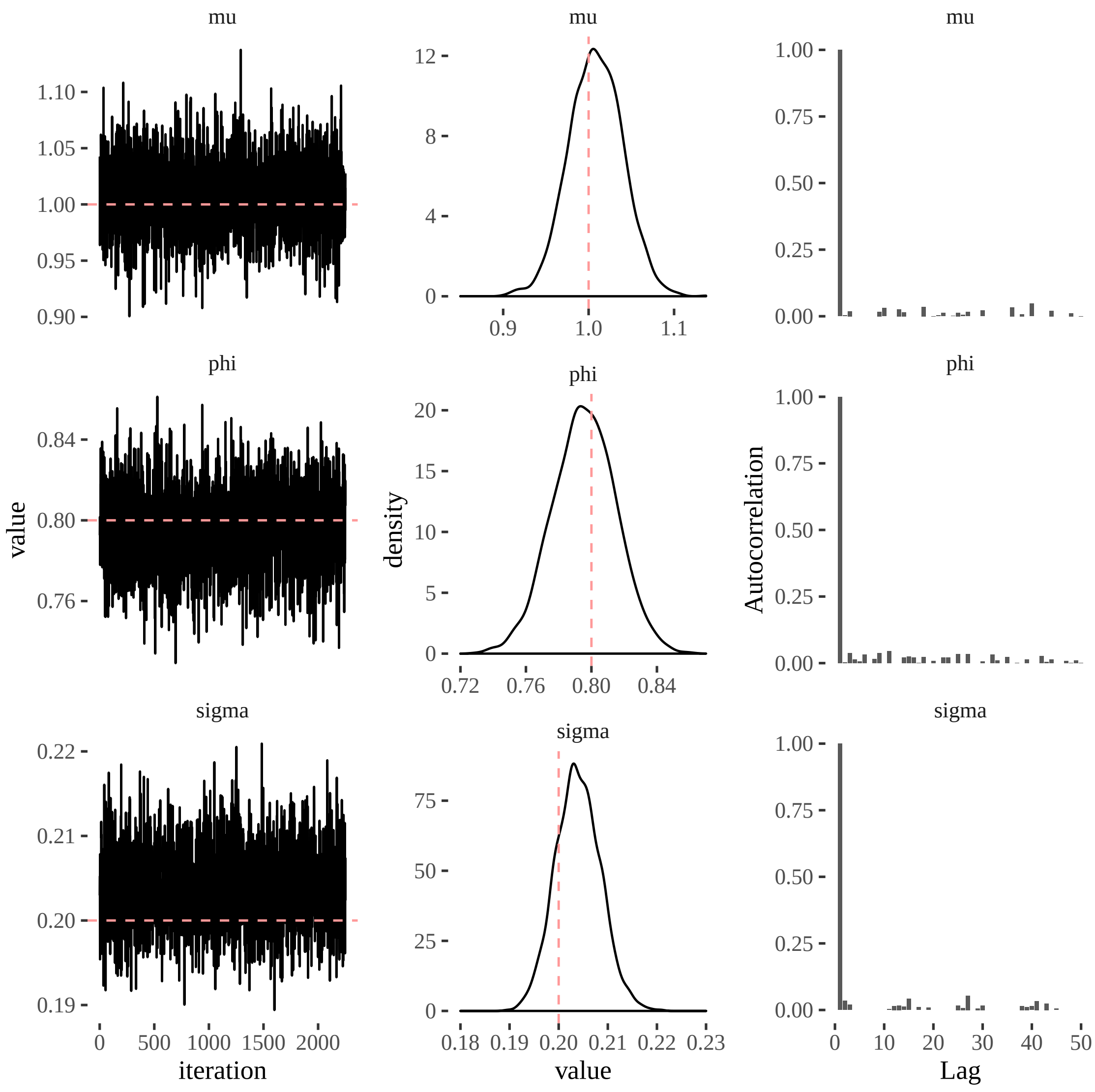


Figure 1.3: Diagnostic plots for the static parameters of the AR(1) model using the Gibbs sampler. There are 10,000 iterations with 1,000 iterations discarded as burn-in and every 4th iteration is retained to reduce auto-correlation. Trace plots of the marginal parameter posterior distributions, (left). Empirical posterior densities of the static parameters, (right).

of prior distributions. The gradient information used in HMC can lead to faster mixing and hence a reduced number of iterations are required at the expense of computing derivatives of the log-posterior. HMC is especially useful for drawing samples from high dimensional posterior distributions. HMC can not be used to sample discrete parameters, but it is sometimes possible to integrate discrete parameters out of a model.

The intuition for HMC is developed from Hamiltonian dynamics. Hamilton's equations are written as

$$
\begin{aligned}
\frac{\mathrm{d}\boldsymbol{p}}{\mathrm{d}t} &= -\frac{\partial \mathcal{H}}{\partial \boldsymbol{q}}, \\
\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t} &= +\frac{\partial \mathcal{H}}{\partial \boldsymbol{p}}.
\end{aligned}
$$

Where $\boldsymbol{q}$ represents a particle position and $\boldsymbol{p}$ is the momentum of the particle. The Hamiltonian of a physical system can be written as the sum of the kinetic and potential energy

$$
\mathcal{H}(\boldsymbol{q}, \boldsymbol{p}) = T(\boldsymbol{q}) + V(\boldsymbol{p}). \tag{1.14}
$$

The HMC algorithm uses a combination of Gibbs sampling, Hamiltonian dynamics and a Metropolis-Hastings step; hence, it is sometimes called Hybrid Monte Carlo. First, the posterior distribution is augmented with an additional momentum parameter ϕ, this is an auxiliary parameter which is not of direct interest when calculating the parameter posterior distribution. The static parameters, ψ correspond to the position in Hamilton's equations, the kinetic energy is $T(\phi) = \frac{1}{2}\phi^T\phi$ since $\phi = m\dot{q}$ ($\dot{q}$ is the first derivative of position with respect to time, i.e. the particle velocity) where $m = 1$ is the particles mass and $T = \frac{1}{2}m\dot{q}^2$. The joint density of the position and momentum can be written as

$$
p(\psi, \phi) \propto \exp\left\{\log p(\psi|y) - \frac{1}{2}\phi^T\phi\right\}, \tag{1.15}
$$

where $\log p(\psi|y)$ is the log of the target posterior distribution written up to a normalising constant and represents the negative potential energy. The kinetic energy is the kernel of a standard multivariate Normal distribution, the identity covariance matrix can replaced by a tuning parameter, $\varSigma$ termed the mass matrix, from analytical mechanics [Betancourt, 2017]. Hamilton's equations are discretised in order to update the values of the static parameters, ψ (the position in Hamilton's equations) and the momentum ϕ. A special discretisation is used which exactly conserves volume in the Hamiltonian called a leapfrog step:

$$
\begin{aligned}
\phi_{t+\varepsilon/2} &= \phi_t + \frac{\varepsilon}{2}\nabla_\psi \log p(\psi_t|y), \\
\psi_{t+\varepsilon} &= \psi_t + \varepsilon\phi_{t+\varepsilon/2}, \\
\phi_{t+\varepsilon} &= \phi_{t+\varepsilon/2} + \frac{\varepsilon}{2}\nabla_\psi \log p(\psi_{t+\varepsilon}|y),
\end{aligned} \tag{1.16}
$$

where $\nabla_\psi \log(p(\psi|y))$ is the gradient of the un-normalised log-posterior distribution. ε is a tuning parameter in the HMC algorithm and represents the step-size of a leapfrog step. The full algorithm is written in Algorithm 3.

Algorithm 3: Hamiltonian Monte Carlo

```
   Result: Return $\psi_i, i = 1, \dots, M$
1  Given $p(\psi)$, $\log p(\psi|y)$ and tuning parameters $\varepsilon$, $L$;
2  Sample $\psi_0 \sim p(\psi)$;
3  for $i$ in 1:M do
4      Sample $\phi \sim \text{MVN}(0, \Sigma)$.;
5      Set $\psi^\star := \psi_{i-1}$, $\phi^\star := \phi$;
6      for $j$ in 1:L do
7          $\psi^\star, \phi^\star = \text{Leapfrog}(\psi^\star, \phi^\star, \varepsilon)$
8      end
9      Calculate $\alpha = \log p(\psi^\star|y) - \frac{1}{2}\phi^{\star T}\Sigma^{-1}\phi^\star - \log p(\psi_{i-1}|y) + \frac{1}{2}\phi^T M^{-1}\phi$;
10     Sample $u \sim U(0, 1)$;
11     if $\log(u) < \alpha$ then
12         Set $\psi_i := \psi^\star$
13     else
14         Set $\psi_i := \psi_{i-1}$
15 end
16 function Leapfrog($\psi$, $\phi$, $\varepsilon$):
17     $\tilde{\phi} = \phi + \frac{\varepsilon}{2}\nabla_\psi \log p(\psi|y)$;
18     $\tilde{\psi} = \psi + \varepsilon M^{-1}\tilde{\phi}$;
19     $\tilde{\tilde{\phi}} = \tilde{\phi} + \frac{\varepsilon}{2}\nabla_\psi \log p(\tilde{\psi}|y)$;
20     return $\tilde{\psi}, \tilde{\tilde{\phi}}$;
```

HMC can be more efficient in terms of mixing then random walk Metropolis-Hastings schemes for models with a high dimensional parameter space [Neal et al., 2011]. Selecting the tuning parameters corresponding to the leapfrog step size ε and the number of leapfrog steps per iteration, L is typically done using short pilot runs of the chain and targeting an acceptance rate of 0.65. Efforts have been made to automate selection of tuning parameters in the HMC algorithm and have led to the No-U-turn sampler which performs the optimum number of leapfrog steps [Hoffman and Gelman, 2014] and empirical HMC [Wu et al., 2018]. In both cases the optimum number of leapfrog steps are chosen such that the discretised steps in the posterior distribution do not make a U-turn, by first heading away from the previously accepted parameter value, then turning back in the direction of the previous parameter when the value of the gradient changes.

One drawback of HMC is deriving and programming gradients of the log posterior distribution as this can be tedious and error prone. In order to simplify this requirement, automatic differentiation (AD) programs can be used, which evaluate the function and the exact derivative of a function at a point: see chapter 3 for an in-depth discussion of

AD. Stan is an example of a program which implements AD for HMC and variational inference [Carpenter et al., 2016]. TensorFlow [Abadi et al., 2015] is a software library which implements deep learning architectures and trains them using gradient descent and calculates gradients automatically using back-propagation. TensorFlow Probability is a recent add on which aims to bring Bayesian inference to TensorFlow [TensorFlow, 2018].

To illustrate the basic concepts, the HMC algorithm is applied to the problem of identifying the static parameters of the AR(1) model first introduced in Section 1.3.3. The log-measure is simply the log-likelihood of the AR(1) process plus the log-priors on the parameters. An advantage of using HMC over Gibbs sampling in this case is the freedom of choosing more appropriate prior distributions for the parameters. The prior distributions chosen for the static parameters are given in Equation 1.17.

$$\begin{aligned}\phi &\sim \mathcal{B}(3,4)\\ \mu &\sim \mathcal{N}(0,1)\\ \sigma &\sim \frac{1}{2}\text{Cauchy}(6,6)\end{aligned} \tag{1.17}$$

A half Cauchy prior distribution is used for the standard deviation, σ, of the AR(1) process [Polson et al., 2012]. The probability density function of the half-Cauchy distribution with location ℓ and scale γ is:

$$p(\sigma) = \frac{2I(\sigma > 0)}{\pi\gamma\left[1 + \left(\frac{\sigma-\ell}{\gamma}\right)^2\right]}, \tag{1.18}$$

where $I(\sigma > 0)$ is an indicator function which takes the value 1 when $\sigma > 0$ and 0 otherwise. The likelihood of the AR(1) process is given in equation 1.10. The log-likelihood is:

$$\begin{aligned}\log p(\alpha_{1:N}|y) = \frac{1}{2}\log(1-\phi^2) - \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(\alpha_1-\mu)^2(1-\phi^2)\\ - \frac{N}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{t=2}^{N}(\alpha_t - \mu - \phi(\alpha_{t-1}-\mu))^2.\end{aligned} \tag{1.19}$$

The autoregressive parameter ϕ and the variance σ^2 are bounded parameters. For the AR(1) model to be stationary $-1 \leq \phi \leq 1$ and the standard deviation is strictly positive $\sigma > 0$. There exist modifications to the HMC algorithm for sampling constrained parameters, the simplest of which is to compute a change of variables for the bounded parameters and perform HMC in the unconstrained space.

The un-normalised log-posterior and the gradient, $\frac{\partial \log p(\psi|\alpha_{1:N})}{\partial \psi_i}$ must be evaluated

Constraint	$g(x)$	$g^{-1}(x)$	$\frac{\mathrm{d}g^{-1}(x)}{\mathrm{d}x}$	$\frac{\mathrm{d}}{\mathrm{d}x}\log\left(\frac{\mathrm{d}g^{-1}(x)}{\mathrm{d}x}\right)$
$x > a$	$\log(x-a)$	$\exp(x)+a$	$\exp(x)$	1
$a < x < b$	$\log\left(\frac{y-a}{b-y}\right)$	$\frac{b-a}{(1+\exp(-x))}+a$	$\frac{(b-a)\exp(-x)}{(1+\exp(-x))^2}$	$-1+\frac{2\exp(-x)}{1+\exp(-x)}$
$x < b$	$\log(b-x)$	$b-\exp(-x)$	$\exp(-x)$	-1

Table 1.1: Transformation functions and derivatives for use in Hamiltonian Monte Carlo

with respect to unconstrained parameters $\psi' = g(\psi)$. In order to evaluate the probability density function of ψ' given the probability density function of ψ is $f_\psi(\psi)$ then:

$$f_{\psi'}(\psi') = \left|\frac{\mathrm{d}}{\mathrm{d}\psi}g^{-1}(\psi)\right| f_\psi(g^{-1}(\psi')) \tag{1.20}$$

The evaluation of the log-posterior using the unconstrained parameters requires the log-Jacobian (a Jacobian is a matrix of first-order partial derivatives) and the evaluation of the gradient requires the derivative of the log-Jacobian. Each constraint requires a unique transformation, the required transformations, inverses and derivatives are presented in Table 1.1.

The transformations required to transform the parameters to the unconstrained parameters are $\psi' = \{\phi', \mu', \sigma'\} = \{\log(\phi/(1-\phi)), \mu, \log(\sigma)\}$. The inverse transformation, required when computing the un-normalised log-posterior is $\psi = \{1/(1-\exp(-\phi')), \mu', \exp(\sigma')\}$. The density of the prior distributions evaluated at the unconstrained parameters using the log-Jacobian is:

$$\begin{aligned}
\log p(\phi') &= (\alpha_\phi - 1)\log(\phi') + (\beta_\phi - 1)\log(1-\phi') - \log(\mathrm{B}(\alpha_\phi, \beta_\phi) \\
&\quad - \phi' - 2\log(1+\exp(-\phi')) \\
\log p(\mu') &= -\frac{1}{2}\log(2\pi\sigma_\mu^2) - \frac{1}{2}(\mu' - \mu_\mu^2)^2 \\
\log p(\sigma') &= \log(2) - \log(\pi\gamma) - \log\left(\left[1 + \left(\frac{\sigma' - \ell}{\gamma}\right)\right]\right) + \sigma'.
\end{aligned}$$

The gradient of the log-posterior consists of the partial derivatives with respect to each of the static parameters. Equation 1.21 shows the partial derivatives required for the HMC algorithm.

$$\begin{aligned}
\frac{\partial \log p(\alpha_{1:N}|\psi)}{\partial \phi} &= \frac{(\alpha-1)}{\phi} + \frac{1-\beta}{1-\phi} \\
&- \frac{\phi}{1-\phi^2} - \frac{\phi(\alpha_1-\mu)^2}{\sigma^2} \\
&+ \frac{1}{\sigma^2}\sum_{t=2}^{N}(\mu-\alpha_{t-1})(\alpha_t-\mu-\phi(\alpha_{t-1}-\mu)) \\
\frac{\partial \log p(\alpha_{1:N}|\psi)}{\partial \mu} &= -\frac{1}{\sigma_\mu^2}(\mu-\mu_\mu) - \frac{1}{\sigma^2}(\alpha_1-\mu)(1-\phi) \\
&+ \frac{(1-\phi)}{\sigma^2}\sum_{t=2}^{N}(\alpha_t-(\mu+\phi(\alpha_{t-1}-\mu))) \qquad (1.21)\\
\frac{\partial \log p(\alpha_{1:N}|\psi)}{\partial \sigma} &= -\frac{\gamma}{\gamma+\sigma-\ell} + 1 - \frac{(\alpha_1-\mu)^2(1-\phi^2)}{\sigma^3} - \frac{1}{\sigma} \\
&-\frac{N}{\sigma} + \frac{1}{\sigma^3}\sum_{t=2}^{N}(\alpha_t-(\mu+\phi(\alpha_{t-1}-\mu)))^2
\end{aligned}$$

Figure 1.4 shows diagnostic plots from 10,000 iterations of the HMC algorithm, a step size of $\varepsilon = 0.1$ with $L = 10$ leapfrog steps were chosen which resulted in 0.51 proportion of accepted moves in the HMC sampler.

1.3.5 Sequential Monte Carlo

MCMC algorithms typically require all of the data at once to make inferences; however time series are sequential and new observations can arrive potentially changing the results of the inference. Consider that the posterior distribution of the parameters of a statistical model for a time series observed at times $t = 0, \dots, N$ are determined and used to make a prediction for a time-step k steps ahead. After k further time-steps, the analysis is to be repeated and there are now an additional k new observations. The full MCMC must then be re-run taking into account the new observations.

Sequential Monte Carlo (SMC) algorithms can be used in order to avoid this re-computation. SMC methods allow online inference, by using the posterior distribution at timestep t to be the prior distribution at time $t + 1$. Consider a state space model for a time series of the form:

$$\begin{aligned}
Y_t|\theta_t &\sim \pi(Y_t|\theta_t), \\
\theta_t|\theta_{t-1} &\sim p(\theta_t|\theta_{t-1}), \\
\theta_0 &\sim p(\theta_0).
\end{aligned}$$

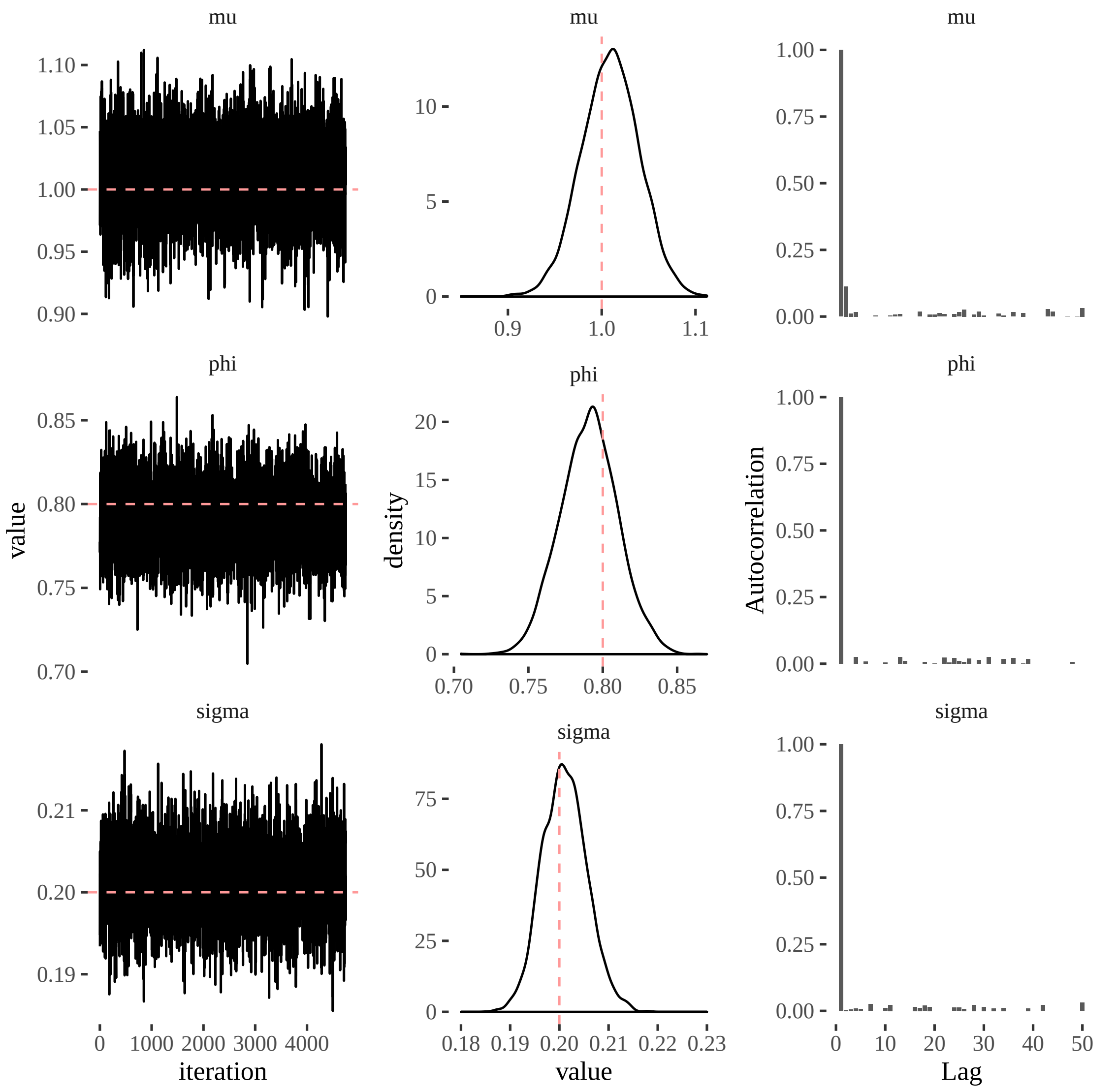


Figure 1.4: Diagnostic plots for the static parameters of the AR(1) model inferred using Hamiltonian Monte Carlo with the parameter value used to simulate the data denoted using a dotted line. Traceplots of the marginal posterior distributions, (left). Marginal posterior density, (middle). Autocorrelation function, (right).

The time series is observed discretely and has measurement noise modelled using a general observation distribution $\pi(Y_t|\theta_t)$ at times $t = 1, \dots, N$. The observations are conditionally independent given the corresponding realisation of the latent-state $\pi(Y_t|\theta_t, y_{1:t-1}) = \pi(Y_t|\theta_t)$. The latent-state is Markovian and has a general transition density which can be forward simulated from $p(\theta_t|\theta_{t-1})$.

The problem of interest is to determine the filtering distribution, which can be inferred using MCMC. However, SMC can be used to develop an online recursive algorithm to determine the latent-state at time $t+1$ given an estimate of the filtering state up to time t, $p(\theta_t|Y_{1:t})$

The most straightforward SMC algorithm is the bootstrap particle filter [Gordon et al., 1993]. Suppose that the filtering state at time t is approximated by a set of M weighted samples so that $p(\theta_t|y_{1:t}) \approx \{(\theta_t^{(i)}, w_t^{(i)}), i = 1, \dots, M\}$. Then a new measurement is observed at time $t+1$, Y_{t+1}. The filtering state at time $t+1$ can be calculated by first advancing the particles approximating the filtering density $\theta_{t+1}^{(i)} \sim p(\theta_{t+1}|\theta_t^{(i)})$ for $i = 1, \dots, M$. Now the likelihood weights corresponding to each particle are calculated using the observation likelihood $w_{t+1}^{(i)} = \pi(y_{t+1}|\theta_{t+1}^{(i)})$. The particles are then re-sampled by sampling the particles with replacement with probability proportional to each particle's weight.

Algorithm 4: Bootstrap Particle Filter

```
  Result: Return $\{\theta_t^{(i)} : i = 1, \dots, M\}, t = 1 : N$
1 Given observations $y_{1:N}$;
2 Simulate from the prior distribution $\theta_0^{(i)} \sim p(\theta_0)$, for $i = 1, \dots, M$;
3 for $t$ in 1 to $N$ do
4     for $i$ in 1 to $M$ do
5         Advance the $i^{\text{th}}$ state particle to the time of the next observation
            $\theta_t^{(i)} \sim p(\theta_t|\theta_{t-1}^{(i)})$;
6         Calculate the conditional likelihood the particle $w_t^{(i)} \sim \pi(y_t|\theta_t^{(i)})$;
7     end
8     Resample the particles with replacement such that the probability of including
        particle $i$ is proportional to $w_t^{(i)}$;
9 end
```

The bootstrap particle filter algorithm is applied to a dynamic Poisson process:

$$\begin{aligned}
Y_t|\lambda_t &\sim \text{Poisson}(\lambda_t), \\
\lambda_t &= \exp(\alpha_t), \\
\alpha_t &= \mu + \phi(\alpha_{t-1} - \mu) + \eta_t, \quad \eta_t \sim \mathcal{N}(0, \sigma^2)
\end{aligned}$$

The rate of the Poisson process evolves in time according to an autoregressive process of order one. The rate of the Poisson distribution is strictly positive, so the latent-state $\alpha_{0:N}$ is exponentiated. The resulting model is non-linear and non-Gaussian. 300 simulations are made from the Poisson process then the filtering state mean and 95% credible intervals are estimated using $M = 1,000$ particles in the bootstrap particle filter. Figure 1.5 shows the simulated observations, Y_t, the rate λ_t and the filtered state.

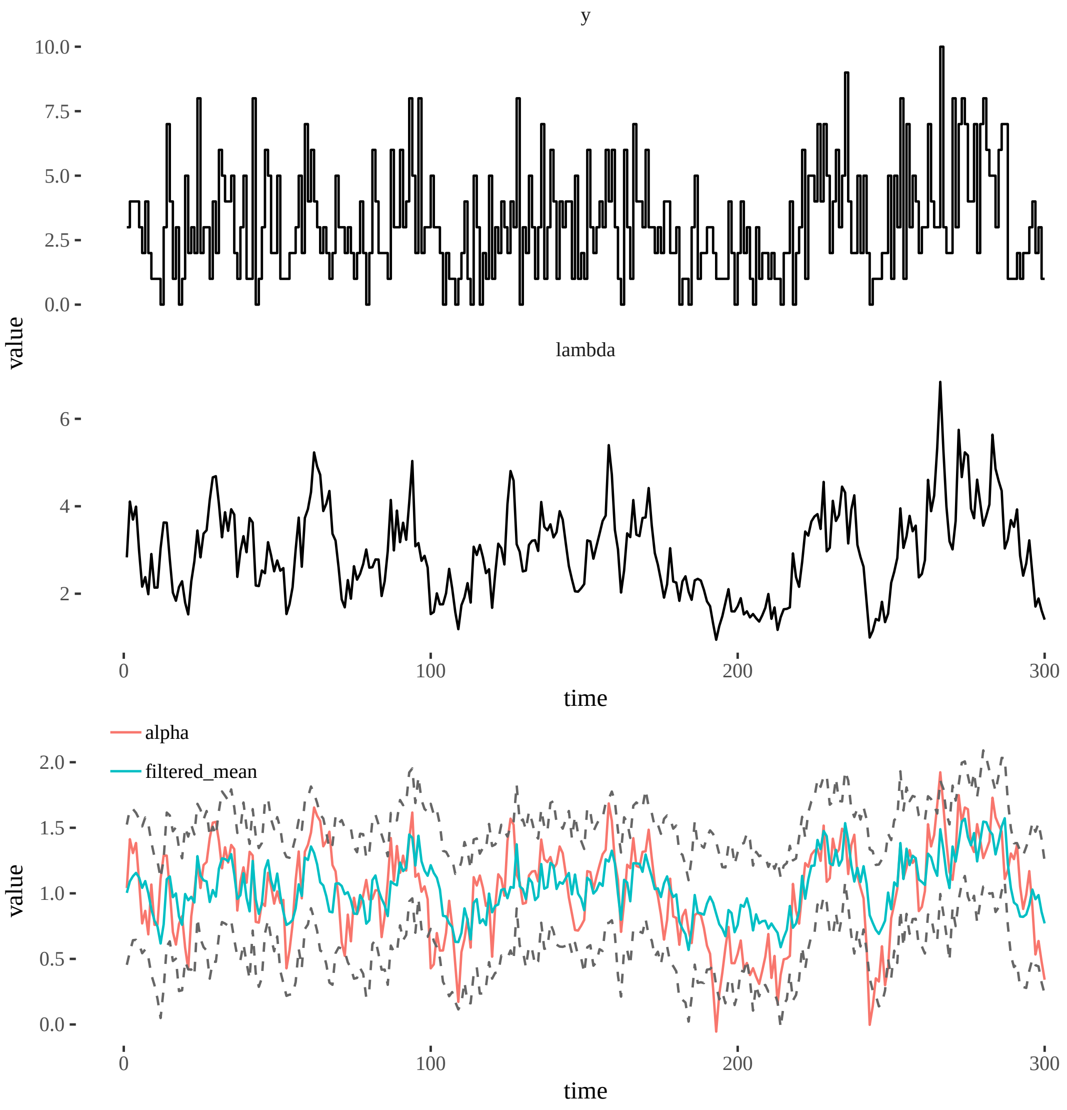


Figure 1.5: Summary of the filtering distribution of an inhomogeneous Poisson Process with an AR(1) latent-state using a particle filter. Simulated observations, (top). Simulated latent-rate, (middle). Particle filtered state with 95% credible intervals, (bottom).

The credible intervals of the filtered state are calculated by ordering the unweighted state particles and selecting the appropriate particles for the chosen interval. In this case, for a 95% interval with $M = 1,000$ particles the 975th and 25th particles are chosen for the upper and lower bound for each time point.

1.4 Summary

This chapter has provided a brief introduction to Bayesian inference for time series analysis including algorithms for determining intractable posterior distributions. The purpose of this chapter is to establish notation and give a background on the inference methods used in this thesis.

Chapter 2

Functional Programming for Statistical Computing

Computing is an important part of modern statistics, as exemplified by the computationally intense inference methods presented in Chapter 1. Therefore, choice of computing environments and programming languages are an important consideration.

Parallel programming is becoming increasingly important. Previously, processor clock speed was increasing year on year, due to the number of transistor chips per silicon chip doubling every year in a process known as Moore's Law [Moore, 1998]. Recently, Moore's law is not being obeyed, and instead investments have been made in multi-core architecture. Parallel programming is challenging, due to modifying shared memory which can cause race-conditions where objects are modified by two threads simultaneously. Scala provides first class support for concurrent and parallel programming in the form of Futures and parallel collections respectively. In addition, the Big Data library Apache Spark [Zaharia et al., 2010] is written in Scala and can be used to perform operations efficiently on data which can't fit on a single machine.

Scala is a functional programming language which makes heavy use of function composition, higher-order functions (HOFs, see section 2.1.4) and type classes (enabling ad-hoc polymorphism) to build large applications from simple functions. These features, along with referential transparency (section 2.1.1) allow programmers to build complex programs whilst maximising code-reuse and minimising lines-of-code. This style of programming is a departure from object oriented or imperative programming and as such can be difficult to learn for those coming from languages which emphasize this approach. The functional approach is often described as declarative which means the programmer expresses the intent of a transformation without telling the computer how to perform the operations. Imperative languages use for-loops or while loops to modify collections of elements which feature additional mutable variables, such as counters which are not of interest to the final calculation. Using loops can result in off-by-one errors whereby all elements in a collection are not modified as intended. These run-time errors are hard to debug and are virtually eliminated using HOFs to modify elements of a collection.

The category theory (functors, applicatives, monads etc.) presented in this chapter unifies the collection types (and other type constructors) in a mathematical framework. This means that when a type constructor (such as a `List`, `Vector` which are collections with have different performance characteristics or a `Future` which can be used to asynchronously retrieve a result without blocking the main thread) forms a monad then the interface for the type is predictable and hence a functional programmer with knowledge of monads knows how to write programs using this data type.

All of the code used to perform statistical inference in this thesis (except the graphics which were created using R [R Core Team, 2015] and ggplot2 [Wickham, 2016]) was developed using Scala, a statically typed functional programming language which runs on the Java virtual machine. This chapter introduces the functional features in Scala which are useful for statistical computing. The features illustrated here are present in other statically typed functional programming languages such as Haskell. In addition, some features are present in dynamic languages such as R, which has support for higher order functions as exemplified in the `apply` family of functions. Scala was chosen since it runs on the Java virtual machine which means deployment of programs on different operating systems and in the cloud is straightforward. Scala has access to all available Java libraries which means building on existing established libraries is straightforward. Scala has an interactive console which can be used to quickly explore and prototype new ideas without starting a new project. Finally, Scala has a performance advantage over dynamic languages such as R and Python, typically slow running elements of an R program are re-written using C/C++ or FORTRAN which results in context switching, slower development and code which is harder to maintain.

This chapter introduces the fundamental concepts of functional programming and applies them to building a novel functional particle filter in section 2.1.15.

2.1 Principles of Functional Programming

In functional programming, a program is comprised of many small functions each of which are easy to test and reason about in isolation. These small functions are composed together to form a larger program, which in practice is more difficult to reason about and test. The programmer gains confidence in the correctness of the program by ensuring each individual function in the composition is correct.

2.1.1 Referential Transparency and Pure Functions

Referential transparency is a fundamental principle of functional programming. A function is referentially transparent if it produces the same output value for the same function parameters upon repeated evaluation and produces no side effects. This enables the programmer to directly replace any referentially transparent function with the value of that function without changing the output of the program. This is called reasoning by substitution. A function which is referentially transparent is often called deterministic.

A *pure function* is a function which is referentially transparent, inculpable and total. A function is total if it returns a value (not an exception) for every possible input. Pure functions are more straightforward to test, reason about and reuse by composing with other pure functions. A function is called *inculpable* if it is free from side effects.

Functions which have *side effects* are not referentially transparent. An example of a side effect is printing to the screen, reading from a file or database, accepting input from the user or mutating a global variable. Some or all of these actions are needed for a program to be useful. Functional programming makes side effects explicit and good functional design ensures that side effects are only performed when needed, typically at the edges of a program.

An example of a common side effect in statistical programming is generating a random number. In Scala (which is not a *pure* functional programming language) a random double between 0 and 1 can be generated using `scala.util.Random.nextDouble()`. When calling this function it is impossible to know which random number will appear next without knowing the current state of the random number generator. Additionally, when this function is called multiple times in a program it will return different values. This is due to the fact that the function has the side effect of mutating the global random number state. The state monad can be used to implement a purely functional random number generator and this is expanded upon in section 2.1.11.

2.1.2 Static Types

Functions can accept a range of parameters and each of these parameters has a type. For instance the function $f(x) = x + 1$ is a mathematical function which adds one to its parameter, x. In this case, the parameter type is implicitly a numeric type as the method $+$ is associated with numbers. The return type of the function f is also a numeric type. Types can be either inferred (as in the function f) or they can be made explicit. The example Scala code below shows a version of the function f written with an explicit type declaration of `Int => Int`.

```
val f: Int => Int = x => x + 1
```

If this function was applied to a `String` data-type, the program would not compile. In a language with dynamic types, the program would run and possibly try to convert the `String` into an `Int` this can lead to runtime errors which are typically difficult to debug as they can potentially go unnoticed.

Static types are not exclusive to functional programming, however programs using static types are typically easier to reason about; Scala and Haskell are examples of functional programming languages with static types.

2.1.3 Immutable Data

Immutable data is an important aspect of functional programming. An Immutable data structure can not be modified in place. This is a strict requirement which ultimately makes code easier to reason about. However it means that typical imperative programming constructs, such as for-loops and while-loops can not be used.

2.1.4 Higher Order Functions

In programming languages which have functions as first class citizens, functions can be passed as an argument to other functions. Functions which accept other functions as arguments or have a function as a return type are called *higher order functions*.

In order to motivate the introduction of higher order functions, consider a datatype which can be used to represent a sequence of data (for instance a time series): a singly linked list of integers. This can be defined recursively by first defining a `case class` (which is a Scala object with methods defined to create the class and update the abstract members `head` and `tail`) `Cons` which constructs a list by binding an element of type `Int` and a list of elements of type `Int` together.

```scala
sealed trait List
case class Cons(head: Int, tail: List) extends List
case object Nil extends List
```

This is an *algebraic data type* (ADT) known as a sum type, a `List` can be either a `Cons` with a single element `head` and the rest of the list, `tail` or an empty list `Nil`. `Cons` is short for construct and can be used to construct a list, the `tail` of the final `Cons` is the empty list

```scala
val list = Cons(1, Cons(2, Cons(3, Nil)))
```

This defines a singly-linked list containing the integers 1, 2 and 3. A common operation performed on lists is to apply a function, `f`, to each element in the list; in imperative programming this would be done using a loop. The loop is used to extract each element from the list in turn and mutate the value in place by applying `f`. However, data structures are immutable in functional programming and mutating values is not possible; a copy of the list with the function applied to each element should be produced. In addition, iterating through a loop by incrementing a counter variable is also not possible, a recursive function must be used. The function is of the following form, taking a list of integers and the desired function from `Int => Int` and returning a `List[Int]`

```
def map(fa: List)(f: Int => Int): List = fa match {
  case Nil => Nil
  case Cons(x, xs) => Cons(f(x), map(xs)(f))
}
```

`map` utilises pattern matching to access the subtypes of the `List` data type: if the list is empty it returns the empty list, and if the list is a `Cons` then the function `f` is applied to the first element of the `Cons` and prepended to a recursive call to the `map` function.

Another operation commonly performed using collections is reduction, for instance summing the elements of the list. The required function is again recursive

```
def sum(l: List): Int = {
  def loop(rem: List, acc: Int): Int = rem match {
    case Nil => acc
    case Cons(x, xs) => loop(xs, acc + x)
  }
  loop(l, 0)
}
```

The `sum` function has a tail recursive helper function `loop`, tail recursive functions are optimised by the compiler in a process called tail call elimination which removes the recursive calls from the stack so the sum of very long lists can be calculated without overflowing the stack [Friedman and Wise, 1974]. `map` can also be re-written using a tail call. The accumulator `acc` is initialised to zero (the identity operator for addition) and the binary function applied each iteration of the loop is plus. This can be further generalised to accept any binary operation, `f: (Int, Int) => Int` and any zero, `b`:

```
def foldLeft(l: List)(b: Int)(f: (Int, Int) => Int): Int = l match {
  case Nil => b
  case Cons(x, xs) => foldLeft(xs)(f(b, x))(f)
}
```

`foldLeft` can be used sum the elements of the list

```
val sum = foldLeft(list)(0)((a: Int, b: Int) => a + b)
```

Listing 2.1: Summing integers in a singly linked list using a higher order function

A complementary function, `foldRight` can also be defined which reduces the values of a list from the right.

`map`, `foldLeft` and `foldRight` are commonly used higher order functions implemented in the Scala standard library for many collection types, each with different performance properties. Higher order functions are common across collections, hence learning the operations for one collection will generalise to other collections. Section 2.1.6 shows how collection types sharing the same higher-order functions can be abstracted over.

2.1.5 Polymorphism

The definition of the singly linked `List` in section 2.1.4 is for lists of integers, however lists can be used to hold any data type. Polymorphism is a way of abstracting over a type, allowing the programmer to write a function once for many different data types. This is straightforward in untyped programming languages such as `R` and Python, however the application of a function expecting different types can cause runtime errors which are difficult to debug. Polymorphism allows the programmer to retain type safety whilst writing functions which can be applied to multiple similar data types. The `List` can be redefined in the following way

```scala
sealed trait List[+A]
case class Cons[A](head: A, tail: List[A]) extends List[A]
case object Nil extends List[Nothing]
```

The type `A` is a placeholder for any type, or a type variable. This means any data type can go into a list, `Double`, `String` or another `List[Double]`, however this list must contain the same types (lists which contain a mixture of different data types are called Heterogeneous-lists or H-lists). The notation `+A` means the type `A` is covariant which means that if a `A` is a subtype of `B` then `List[A]` is a subtype of `List[B]`. The higher order functions which operate on lists can be redefined in order to be type agnostic

```scala
def map[A, B](l: List[A])(f: A => B): List[B]
def foldLeft[A, B](l: List[A])(b: B)(f: (B, A) => B): B
```

Consider writing a function to sum the values in a list, it does not matter which type of number the list holds, `Double`, `Int` or `Float`, `+` is defined for all of these number types. Hence a function implementing a reduction with `+` on a list containing any of these number types will look equivalent:

```scala
def sumInts(l: List[Int]): Int =
  l.foldLeft(0)(_ + _)
def sumFloats(l: List[Float]): Float =
  l.foldLeft(0)(_ + _)
def sumDoubles(l: List[Double]): Double =
    l.foldLeft(0)(_ + _)
```

Listing 2.2: Calculate the sum of different data types contained in a list

The code block in listing 2.2 shows repeated code and this is liable to code bugs if one definition is updated and the others neglected. These functions can be written such that they are agnostic to the type of numeric value, thus reducing the amount of code to maintain. The Scala library Spire[1] is used to write polymorphic functions for general numeric values. Spire also provides additional number types such as `Rational`, a perfect precision fraction, `Complex[A]`, an implementation of complex numbers and type classes from abstract algebra such as `Group` and `Ring`. Spire also has an implementation of a purely functional random number generator as motivated in section 2.1.10.

```
def sum[A: Numeric](xs: List[A]): A =
  xs.foldLeft(0)(_ + _)
```

Listing 2.3: A polymorphic function which calculates the sum of a list containing Numeric types

The function in listing 2.3 will sum a list containing any data type which has a `Numeric` instance. Numeric is a type class [Hall et al., 1994] which implements operations specific to numbers, `Int` and `Double` both have `Numeric` instances defined in the Scala standard library and hence this function will apply to both number types. Using polymorphism and type bounds, type safety is retained and the function can be written once for all types which have a Numeric instance. This makes programs easier to debug as the compiler will warm about types which do not "line up". The type class pattern is used to write generic functions. New implementations of the `Numeric` type class can be defined by the user in order to extend the usefulness of generic functions, as explored later in section 3.5 on automatic differentiation.

2.1.6 Higher Kinded Types

`List` is sometimes referred to as a type constructor and is similar to other collection types such as `Option` or `Vector`. An `Option` is a collection which can be either empty (`None`) or contain exactly one element (`Some`) and can used for handling failure. `Vector` is a collection of none or more elements, similar to `List`. The implementations of `List` and `Vector` are different and both have different performance characteristics. A singly linked list provides $O(1)$ access to the head of the list but $O(n)$ access time to a specific element. `Vector` is implemented as a trie [Fredkin, 1960] with a branching factor of 32 and this means that random access to elements in a `Vector` is faster than in a `List`. Different

[1]https://github.com/non/spire

collections should be chosen for different purposes, but it does not mean that all functions have to specialise for a given collection.

All these collections (and other type constructors) share common abstractions which can be used to unify them and then write code which is polymorphic not just in the type, but in the type constructor. This allows for greater code re-use.

Higher kinded types are not common in non-functional languages and are far from ubiquitous even in functional languages. However they provide powerful abstractions leading to cleaner, more elegant code. The abstractions come from a branch of mathematics known as category theory and are hence well defined. This section provides a brief introduction to category theory; see Barr and Wells [1990] for a comprehensive review. For an introduction to category theory aimed at programmers, see Milewski [2018].

Category theory is concerned with describing functions and function composition between mathematical objects in a very general way.

Definition. A *category* $\mathcal{C}$ is a collection of objects and morphisms which go between them such that for the objects X, Y and Z in $\mathcal{C}$ and morphisms $g : X \to Y$ and $h : Y \to Z$

1. There must exist a morphism $f : X \to Z$ which is the composition of g and h, $f = h \circ g$
2. Each object in a category must have an identity morphism written as $\text{id}_X : X \to X$
3. Composition of morphisms must be associative, for all $f : X \to Y$, $g : Y \to Z$ and $h : Z \to W$ then $h \circ (f \circ g) = (h \circ f) \circ g = h \circ f \circ g$
4. For every morphism $g : X \to Y$ then $\text{id}_Y \circ g = g = g \circ \text{id}_X$

The collection of objects in a category $\mathcal{C}$ is often denoted $\text{obj}(\mathcal{C})$ and morphisms $\text{hom}(\mathcal{C})$. The set of morphisms from X to Y is written as $\text{hom}_{\mathcal{C}}(X, Y)$. Figure 2.1 shows a commutative diagram for a category. Following the arrows of the commutative diagram from any given start and end point is equivalent.

An example of a category is the category of sets, where each set is an object and the morphisms are functions between the sets, the identity morphism is the identity function. In functional programming, the category under consideration is similar to the category of sets, each object is a collection of types and the morphisms are the functions between the types.

2.1.7 Functors

A functor is a structure preserving mapping from one category to another.

$$\begin{tikzcd}
X \arrow[loop above, "\mathrm{id}_X"] \arrow[r, "g"] \arrow[d, "f"'] & Y \arrow[loop above, "\mathrm{id}_Y"] \arrow[dl, "h"] \\
Z \arrow[loop below, "\mathrm{id}_Z"] &
\end{tikzcd}$$

Figure 2.1: Representation of a simple category, $\mathcal{C}$, as a directed graph.

Definition. If $\mathcal{C}$ and $\mathcal{D}$ are categories, then a *functor* $F : \mathcal{C} \to \mathcal{D}$ takes each object X in $\mathcal{C}$ to an object $F(X)$ in the category $\mathcal{D}$ and each morphism, $f : X \to Y$ in $\mathcal{C}$ to a morphism in $\mathcal{D}$, $F(f) : F(X) \to F(Y)$ such that

1. $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$ for each X in $\mathcal{C}$
2. $F(g \circ f) = F(g) \circ F(f)$ for all morphisms, f and g in $\mathcal{C}$

Since functional programming is mainly concerned only about the category Set, all functors are actually endofunctors, $F : \text{Set} \to \text{Set}$. An endofunctor is defined as a functor from a category to itself, $F : \mathcal{C} \to \mathcal{C}$.

A functor represents a type constructor which can be `mapped` over. The signature of the `map` function for each collection `List`, `Option` or `Vector` can be written as follows:

```scala
def map[A, B](fa: List[A])(f: A => B): List[B]
def map[A, B](fa: Option[A])(f: A => B): Option[B]
def map[A, B](fa: Vector[A])(f: A => B): Vector[B]
```

The function `f: A => B` is lifted into the context of `List`, `Option` and `Vector` respectively. The higher order function `map` allows `f` to operate on an element in a context. The `map` functions defined above are identical apart from the type constructor, the type constructor can thus be abstracted over using the functor type class:

```scala
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

The type constructor `F[_]` has been abstracted over using a *higher kinded type*, the `Functor`. Now functions can be defined in terms of type constructors which form a functor without reference to the specific type constructor. The Scala library Cats[2] (a shortening of Category) provides type class instances for built in Scala types. Cats implements type classes in a more sophisticated way, and when combined with Spires type classes, allows polymorphic functions to be written in idiomatic Scala style without sacrificing readability:

```scala
def addOne[F[_]: Functor, A: Numeric](fa: F[A]): F[A] =
  fa map (_ + 1)
```

This function can be applied to any functor containing Numeric values.

2.1.8 Applicatives

The data constructors considered so far are also applicatives [McBride and Paterson, 2008]. An applicative is a special functor which has a `pure` method which lifts a value into the context of the applicative and a `ap` which can be used to apply a function to a value in the applicative context (the `F[_]`)

```scala
trait Applicative[F[_]] extends Functor[F] {
  def pure[A](a: A): F[A]
  def ap[A, B](fa: F[A])(f: F[A => B]): F[B]
}
```

To define `pure`, use the type constructors for the collections:

```scala
def pure[A](x: A): List[A] = x :: Nil
def pure[A](x: A): Option[A] = Some(x)
def pure[A](x: A): Vector[A] = Vector(x)
```

Note that `::` is used as an infix `Cons` operator. In Cartesian closed categories, like Set, `ap` can be defined in terms of `zip` and `map`. In category theory an applicative defined in terms of `ap` is a lax closed functor and an applicative defined in terms of `zip` is a lax monoidal functor. In Cartesian closed categories, these are equivalent. As such, the applicative type class can be defined in terms of the functions `zip`, `map` and `pure`. The type signature of `zip` is given by

```scala
def zip[A, B](fa: F[A], fb: F[B]): F[(A, B)]
```

Then `ap` can be derived

```scala
def ap[A, B](fa: F[A])(f: F[A => B]): F[B] =
  map(zip(f, fa)){ case (fi, a) => fi(a) }
```

[2]https://typelevel.org/cats/

The definition of `zip` for the list collection can be defined recursively as follows,

```
def zip[A, B](fa: List[A], fb: List[B]): List[(A, B)] = (fa, fb) match {
  case (Nil, lb) => Nil
  case (la, Nil) => Nil
  case (a :: ta, b :: tb) => (a, b) :: zip(ta, tb)
}
```

If the lists are different lengths, `zip` will return a list of tuples the same length as the shortest list.

For a useful example of an applicative functor, consider the (built in) type used to encapsulate asynchronous execution, `Future`. Asynchronous execution is where execution is started on a given function but does not have to complete before the program resumes, this allows multiple asynchronous functions to operate in parallel. The following three function signatures are used to query a datastore,

```
def getName(id: Int): Future[String]
def getAge(id: Int): Future[String]
def getAddress(id: Int): Future[String]
```

Each of these functions perform similar operations but connect to different data sources to retrieve the information. Each function is independent of the others, as the only argument is the `id`. Presumably, these functions can be started simultaneously and can run in parallel. The Scala standard library provides a specialised `sequence` method for performing computations which return a `Future` in parallel, `Future.sequence`:

```
Future.sequence(List(getName(id), getAge(id), getAddress(id)))
// Future[List[String]]
```

The function `sequence` can be generalised to any applicative and any collection which has a traverse instance, this is expanded upon in section 2.1.14.

2.1.9 Natural Transformation

A functor is a morphism between two categories (which itself has morphisms between its objects). A *natural transformation* is a morphism between functors which preserves the structure of the categories.

Definition. If $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{C} \to \mathcal{D}$ are both functors between the categories $\mathcal{C}$ and $\mathcal{D}$ then a *natural transformation* $\alpha : F \Rightarrow G$ is a family of morphisms such that

1. For all $X \in \mathcal{C}$ then $\alpha_X : F(X) \to G(X)$ is a morphism in $\mathcal{D}$ called the component of α at X
2. For each morphism $f \in \mathcal{C}$, $f : X \to Y$ then $\alpha_Y \circ F(f) = G(f) \circ \alpha_X$

$$\begin{tikzcd}
F(X) \arrow[r, "F(f)"] \arrow[d, "\alpha_X"'] & F(Y) \arrow[d, "\alpha_Y"'] \\
G(X) \arrow[r, "G(f)"] & G(Y)
\end{tikzcd}$$

Figure 2.2: Commutative diagram of the natural transformation.

Figure 2.2 shows a commutative diagram of the second natural transformation law.

A natural transformation between rank-1 types, a constructor such as `List[A]` or `Vector[A]`, for any type `A` can be defined as follows

```
trait ~>[F[_], G[_]] {
  def apply[A](a: F[A]): G[A]
}
val toList  = new (Option ~> List) {
  def apply[A](a: Option[A]): List[A] = a.toList
}
```

This natural transformation may appear trivial, but it is the cleanest way to write this function in Scala. If the function is written naïvely, as `def toList[A](l: Option[A]): List[A]` then the method will compile but the type is `Option[Nothing] => List[Nothing]` (as revealed by defining the function `val toListFn = toList _` in the Scala console) and can not be run on, for instance, an `Option[Int]` since `Int` is not `Nothing`. This is caused by type erasure at runtime.

Natural transformations are useful for understanding Monads which are ubiquitous in functional programming, introduced in Section 2.1.10. In addition, natural transformations are used when interpreting embedded programming languages written using the Free Monad [Swierstra, 2008]. Embedded probabilistic programming languages are considered in chapter 3.

2.1.10 Monads

The collections considered so far are monads. A monad represents a value in a context which can be manipulated in a consistent way [Wadler, 1995]. Monads are integral to safe purely functional programming and are used to encapsulate unsafe program behaviour such

$$\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"'] \\
T^2 \arrow[r, "\mu"] & T
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
T \arrow[r, "\eta T"] \arrow[d, "T\eta"'] \arrow[dr, equal] & T^2 \arrow[d, "\mu"'] \\
T^2 \arrow[r, "\mu"] & T
\end{tikzcd}$$

(b)

Figure 2.3: (a) Commutative diagram expressing the first condition of the monad laws. (b) Commutative diagram expressing the second condition of the monad laws.

as IO (input output), async programs, random number generators and partial functions. Suitable monads for these unsafe behaviours include cats effect `IO`[3], `Future`, Breeze `Rand` and `Option` respectively. These monads have unsafe methods which can be used to access the contents of the monad and hence perform the side effect. The unsafe access method is only performed at the end of the program when the side effect is desired, until then combinations of `map` and `flatMap` are used to manipulate the values inside of the monadic context.

Definition. A monad on the category $\mathcal{C}$ is an endofunctor, $T : \mathcal{C} \to \mathcal{C}$ with two natural transformations, $\eta : \mathrm{Id}_c \to T$ and $\mu : T \circ T \to T$ such that

1. $\mu \circ T\mu = \mu \circ \mu T$
2. $\mu \circ T\eta = \mu \circ \eta T = \mathrm{Id}_T$

$T\mu : T(T \circ T) \to T$ is a natural transformation defined by taking the natural transformation μ and the endofunctor, $T : \mathcal{C} \to \mathcal{C}$ then $(T\mu)_C = T\mu_C$. The natural transformation μT is a similar transformation with T composed from the right. Figure 2.3 shows commutative diagrams of the monad laws.

To define a monad in Scala, simply define the two natural transformations required. The natural transformation corresponding to $\eta : \mathrm{Id}_C \to T$ is `pure`, inherited from the `Applicative` type class. The second natural transformation, $\mu : T \circ T \to T$, is referred to as `join`

[3]https://typelevel.org/cats-effect/

```
trait Monad[F[_]] extends Functor[F[_]] {
  def join[A](fa: F[F[A]]): F[A]
  def pure[A](x: A): F[A]
}
```

Typically, in functional programming monads are defined in terms of `flatMap`

```
def flatMap(fa: F[A])(f: A => F[B]) =
  join(map(fa)(f))
```

which is a map followed by a `join` or flatten. Typically, a monad is defined in terms of `flatMap` and `pure`

```
trait Monad[F[_]] {
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
  def pure[A](x: A): F[A]
}
```

Note that `map` can be written in terms of `flatMap` and `pure` and `ap` can be written in terms of `flatMap` and `map`. This means when a monad instance is defined for a given type, only `flatMap` and `pure` must be defined and the methods from functor and applicative are available. Hence all monads are applicative functors, but the converse is not true.

`Option` is a monad which can be used to turn a partial function into a total-function by capturing error states in the type information. Consider two functions which could fail:

```
def sqrt(x: Double): Option[Double] =
  if (x < 0) None else Some(math.sqrt(x))
def log(x: Double): Option[Double] =
  if (x <= 0) None else Some(math.log(x))
```

In order to compose these two functions, a `flatMap` can be used:

```
def composed(x: Double): Option[Double] =
  sqrt(x).flatMap(y => log(y))
```

The same principal can be extended to more than two applications:

```
def composeMore(x: Double): Option[Double] =
  sqrt(x).flatMap(y => log(y)).flatMap(z => log(z))
```

The readability of the sequential application of these partial functions can be improved by using a `for`-comprehension which is a syntactic-sugar for applications of `flatMap` and `map`:

```
def composeFor(x: Double): Option[Double] = for {
  y <- sqrt(x)
```

```
    z <- log(y)
    res <- log(z)
  } yield res
```

The functions returning an `Option` (or any monad) are placed on the right side of the arrow `<-` extracting the value from the monadic context. The values on the left represent the value inside of the monadic context and can be referenced in the same for-comprehension. The function `composeFor` is equivalent to `composeMore`. The for-comprehension is "desugared" into a chain of `flatMap`s and `map`s. This is equivalent to do-notation in Haskell.

In general, different monads do not compose [Jones and Duponcheel, 1993], hence functions with different monadic contexts can not be mixed inside of a for-comprehension. However different contexts are often required, and for certain combinations monad transformers can be used which combine monadic contexts [Liang et al., 1995]. Alternatively, extensible effects and the freer monad can be used for combining effects [Kiselyov et al., 2013]. The freer monad allows the user to combine monadic effects such as error handling and asynchronous computation. These effects are both required when connecting to a remote data source such as a web API or database.

2.1.11 Purely Functional Pseudo Random Number Generator

The `State` monad can be used to keep track of internal state which is cumbersome to pass around as an argument. The state monad is a constructor with a function from the current state to a tuple containing the new value and a new value of the state:

```
case class State[S, A](run: S => (S, A))
```

Since it is a monad, there are implementations of `flatMap` and `pure`:

```
def flatMap[B](f: A => State[S, B]): State[S, B] = State { x =>
  val (st, a) = run(x)
  f(a).run(st)
}
def pure[B](a: B) = State(st => (st, a))
```

The `State` monad can be used to implement a purely-functional pseudo-random number generator (PRNG) by threading a PRNG state through multiple function calls.

Sequences of apparently random numbers are generated by mutating an internal PRNG state each time a function returning a random number is evaluated. Each time a pseudo-random number is asked for a different value is returned and the internal PRNG state is mutated

```
def randNaive = scala.util.Random.nextDouble()
randNaive
```

```
// 0.26167644649515154
randNaive
// 0.4732297985296451
```

This function violates referential transparency (note that a `def` is needed here rather than a `val` since `val` is eagerly evaluated) since each time `randNaive` returns different values at each invocation. In order to consider a purely functional random number generator the internal state can be passed around explicitly. One of the simplest implementations of a pseudo random number generator is a linear congruential generator. This generator is defined by a recursive relation

$$X_{n+1} = (aX_n + c) \bmod m \tag{2.1}$$

where $m > 0$ is the modulus, $0 \leq c < m$ is the increment and $0 < a < m$ is the multiplier and the initial seed is $0 \leq X_0 < m$. It is challenging to choose the parameters a, c and m to produce a long apparently random chain, a problem which is not considered here. The generator can be implemented in Scala as follows:

```
def linearGenerator(a: Long, c: Long, state: Long) =
  a * state + c
def toDouble(x: Long) =
  (x >>> 11) * math.pow(2, -53)
def myGenerator(state: Long): (Long, Double) = {
  val newState = linearGenerator(6364136223846793005L,
 1442695040888963407L, state)
  (newState, toDouble(newState))
}
```

The linear generator is initialised with no modulus and this is equivalent to the modulus being `Long.MaxValue`. The function `toDouble` performs a logical right bit-shift by 11 places, then divides by 2^{53} (53 = 64 - 11). This is because a Scala `Double` is stored using 64 bits. The right shift of the `Long` value by 11 bits and the subsequent division ensures there is no numeric overflow. This has the effect of transforming the seed to be a `Double` between zero and one. In order to generate a new number, `myGenerator` must be given an initial value, which is the initial state of the of the random number generator

```
myGenerator(13515670)
// (3730281199248434349,0.2022189490103492)
```

The function returns the next state on the left of the tuple and a "random" number between zero and one. The function can be initialised from a "random" or chosen seed. In practice, pseudo-random number generators are more complex than this. A well used PRNG with a long non-repeating chain is the Mersenne Twister algorithm [Matsumoto and Nishimura, 1998].

A function to draw a pseudo-random number can be now be written as:

```
type Rand[A] = State[Long, A]
def randDouble: Rand[Double] =
  State(myGenerator)
```

Then in order to sample several numbers, `flatMap` can be used

```
val res = for {
  firstNumber <- randDouble
  nextNumber <- randDouble
} yield (firstNumber, nextNumber)
println(res.run(13515670))
// (2060103227662993080,(0.2022189490103492,0.11167841974883075))
```

The state monad passes along the state to the next invocation of `randDouble`. In order to extract the pseudo-random numbers from the state monad, the function `run` is called with a starting seed. The result is on the right of the tuple with the next seed on the left. This construct forms a referentially transparent PRNG. Note also, that the same initial seed is used as in the previous example and hence the same pseudo-random number is produced. The next number however is different, showing that the PRNG state is mutated in the background.

Since `State` is a functor, `map` can be used to transform the output of the function `randDouble` in order to build other random generators:

```
def randInt(from: Int, to: Int): Rand[Int] =
  randDouble.map(x => (x * (to - from) + from).toInt)
def randBool: Rand[Boolean] =
  randDouble.map(_ > 0.5)
```

The State monad can be used to define a `Rand` monad using other algorithms which generate uniformly distributed random variables, and these can be transformed appropriately to generate values distributed according to other useful distributions. For example the Box-Muller transform is an efficient method of generating standard Normal random numbers [Box and Muller, 1958].

2.1.12 Kleisli Composition

Function composition is essential to functional programming. Consider a data analysis pipeline which has several functions used to clean and process data:

```
val f[A, B]: A => B
val g[B, C]: B => C
```

```scala
val h[C, D]: C => D

def pipeline[A, D]: A => D = f andThen g andThen h
```

The functions are composed using the `andThen` operator. The `pipeline` function is written in what is known as pointfree style, by composing functions without reference to the arguments.

Now consider that each function in the process may fail, failure can be denoted by returning an `Option[A]`

```scala
val f[A, B]: A => Option[B]
val g[B, C]: B => Option[C]
val h[C, D]: C => Option[D]
def pipeline[A, D]: A => Option[D] = (a: A) =>
  f(a).flatMap(b => g(b)).flatMap(c => h(c))
```

It is not easy to see that the `pipeline` function is just the composition of `f`, `g` and `h` since the functions are chained with calls to `flatMap`. This can be written using Scala's monadic for comprehension:

```scala
def pipeline[A, D]: A => Option[D] = (a: A) =>
  for {
    b <- f(a)
    c <- g(b)
    d <- h(c)
  } yield d
```

This notation is easier, but still takes longer to parse than function composition using `andThen`. There is another data type which can be used to compose monadic functions in a natural way, Kleisli

```scala
case class Kleisli[F[_], -A, B](run: A => F[B])(implicit m: Monad[F]) {
  def andThen[C](f: Kleisli[F, B, C]): Kleisli[F, A, C] =
    Kleisli((a: A) => m.flatMap(run(a))(f.run))
}
```

This type is also a monad if the type returned by the function is a monad. A Kleisli represents a function `A => M[B]` and provides the methods `andThen` and `compose`. Now the monadic pipeline can be written as

```scala
def pipeline[A, B]: Kleisli[Option, A, D] =
  Kleisli(f) andThen Kleisli(g) andThen Kleisli(h)
```

This emphasises the compositional nature of the `pipeline` function and it's easy to create new pipelines by combining monadic functions. This can be useful for developing MCMC kernels, where functions are defined from parameters to a monadic distribution over the parameters [Ścibior et al., 2015], ie. `P => Rand[P]`. Kleisli composition can then be used to combine and re-use functions for sampling parameters in hierarchical models.

2.1.13 Foldable

Foldable represents a type class which can be collapsed into a single value from the left or right with a binary reduction function using the functions `foldLeft` and `foldRight` respectively. The collection data types considered so far (List, Vector, Option) are all Foldables. The function signatures of `foldLeft` and `foldRight` are

```scala
trait Foldable[F[_]] {
  def foldLeft[A, B](fa: F[A])(z: B)(f: (B, A) => B): B
  def foldRight[A, B](fa: F[A])(z: B)(f: (A, B) => B): B
}
```

`foldLeft` and `foldRight` generalise many recursive functions for summarising a collection. Calculating the an estimate of the log-likelihood using a particle filter can be implemented using a foldable collection as outlined in section 2.1.15.

2.1.14 Traversable

Traversable is a type class which can be used to apply monadic functions (`f: A => M[B]`) to a collection and return the result in the monadic context. For a concrete example, consider a stochastic function which advances the state of a Markov Chain:

```scala
val stepState: Double => Rand[Double] =
  x0 => Gaussian(x0, 0.5)
```

Listing 2.4: A random walk step utilising a distribution Monad

This function is the Markov transition kernel for a random walk, the function signature is `Double => Rand[Double]`, where `Rand` is a monad which represents a probability distribution [Giry, 1982]. This distribution is a Monad which has the special property that a random sample can be drawn from it. Consider a particle filter (introduced in section 1.3.5), where a collection of particles must be advanced to the next time point of interest. Since this is just applying a function to each element of a collection, a `map` function might be used:

```scala
val particles: Vector[Double]
val advanced: Vector[Rand[Double]] = particles map stepState
```

This results in a `Vector[Rand[Double]]`, interpreted as a collection of distributions for each particle. If instead the intended type was `Rand[Vector[Double]]`, a distribution over the particles, then the `traverse` function can be used

```
def traverse[G[_]: Applicative, A, B](
  xs: Vector[A])(f: A => G[B]): G[Vector[B]]
```

This definition is similar to the `map` function, however allows proper application of a function which returns an applicative `G`. The example of running several `Future`s in parallel from section 2.1.8 shows the use of `sequence` which is an application of `traverse` with the identity function. The traverse type class can be written as

```
trait Traverse[F[_]] extends Functor[F[_]] with Foldable[F[_]] {
  def traverse[G[_]: Applicative, A, B](
    xs: F[A])(f: A => G[B]): G[F[B]]
}
```

The defining function of the `Traverse` type class `traverse` can be implemented in terms of a fold [Meijer et al., 1991]. For a specific collection, `List`, traverse can be implemented as

```
def traverse[G[_]: Applicative, A, B](fa: List[A])(f: A => G[B]): G[List[B]] =
  fa.foldLeft(G.pure(List[B]()))(
    (b, a) => G.map2(f(a), b)(_ :: _) )
```

2.1.15 Functional Particle Filter

Consider implementing a particle filter using the `Foldable` type class and `traverse`. The particle filter computes the approximate posterior distribution up to the current time of the latent-state using a cloud of particles. When performing inference the entire series of posterior distributions (the filtering distributions) are of interest. If the bootstrap particle filter is implemented using a `foldLeft` without accumulating the intermediate states, then they will be lost.

In section 2.1.13 the foldable type class was introduced which reduces a collection of values into one value. The key functions in the foldable type class are `foldLeft` and `foldRight`, two similar related functions are `scanLeft` and `scanRight` which keep the intermediate values from the binary reduction.

Consider the partially observed Poisson model from section 1.3.5, where the latent state is represented by the random walk (`stepState`) from listing 2.4. To simulate data from the model, define a transition kernel which advances the state $x_t \sim p(x_t|x_{t-1})$, then sample an observation from the Poisson distribution with rate $\lambda_t = \exp(x_t)$:

```
def step(x0: Double): Rand[(Double, Double)] = for {
  x1 <- stepState(x0)
  y <- Poisson(math.exp(x1))
} yield (x1, y)

val init = (Gaussian(0.0, 1.0).draw, 0)
val data = MarkovChain(init){ case (x, y) => step(x) }
```

The particle filter is decomposed into smaller functions which can be reused and combined, the conditional log-likelihood of an observation and the Markov transition kernel to advance the state are model-specific, however any suitable resampling scheme can be used.

```
def conditionalLl(x: Double, y: Int): Double = {
  val lambda = math.exp(x)
  Poisson(lambda).logProbabilityOf(y)
}
def mean(xs: Double): Double =
  xs.sum / xs.size
def resample(
  weights: Vector[Double],
  particles: Vector[Double]): Rand[Vector[Double]]
case class PfState(
  ll: Double,
  x0: Vector[Double])
def stepPf
  (state: PfState, y: Int): Rand[PfState] = {
  for {
    advanced <- x0.traverse(stepState)
    w = advanced map (x => conditionalLl(x, y))
    ll = state.ll + mean(w)
    resampled <- resample(w, advanced)
    } yield PfState(ll, resampled)
}
```

Note that the implementation of `resample` is not provided, only the signature. Multinomial, stratified and systematic resampling are all common, acceptable choices. `foldM` (from the Cats library) is used in order to apply the monadic function `stepPf` to a time series of observations. The signature for `foldM` is:

```
def foldM[G[_]: Monad, A, B](fa: F[A], z: B)(f: (B, A) => G[B])
```

The difference between `foldM` and `foldLeft` is that the binary reduction function is monadic and hence the function `stepPf` can be used in order to calculate the posterior distribution of the final latent-state represented by the unweighted particle cloud and the pseudo-marginal log-likelihood:

```
def pFilter(x0: Vector[Double], ys: Vector[Int]): Rand[PfState] = {
  val init = State(0.0, x0)
  ys.foldM(x0)(stepPf)
}
```

The intermediate states of the particle filter can be accumulated in `stepPf` by updating the state of the filter `PfState` to contain a list of particle clouds and appending the most recent particle cloud to the head of the list

```
case class PfState(
  ll: Double,
  x: List[Vector[Double]])
def stepPfAccum
  (state: PfState, y: Int): Rand[PfState] = {
  for {
    advanced <- state.x.head.traverse(stepState)
    w = advanced map (x => conditionalLl(x, y))
    ll = state.ll + mean(w)
    resampled <- resample(w, advanced)
    } yield PfState(ll, resampled :: state.x)
}
```

This can then be applied using `foldM` and the particle clouds representing the latent state will be accumulated.

Although all these type classes have properties and laws associated with them from category theory, the laws are not enforced in the types. In order to verify that the collections introduced in this chapter are monads, for instance, property based testing with the Cats Laws library [Typelevel, 2018] can be used.

Property based testing differs from typical unit testing. In property based testing a property is specified and random data is generated in order to attempt to falsify the property. To determine if a type class forms a monad, then the property tests are the monad laws from section 2.1.10. New properties can be defined for bespoke functions, or laws from other algebraic structures, such as groups or rings can be used. This can be useful when combining collections of parameters defined using a custom case class. In order to summarise the parameters by calculating the mean and variance, a monoid (a monoid is a set S with a closed, associative binary function, $f : S \times S \to S$ with an identity element e such that for every $x \in S$, $f(x, e) = x = f(e, x)$) can be defined for the class of parameters. The associative binary function is defined to be the addition of two parameters and the identity is an empty parameter. The monoid laws can be checked in order to verify the addition of two parameters has been implemented correctly.

2.2 Parallel and Distributed Programming

Modern hardware is parallel, with most consumer computers (and even mobile phones) containing CPUs with multiple cores. The sequential collection types, `List` and `Vector` do not take advantage of multithreading. In Scala, there are parallel collections which share the same monadic interface which enables functions to be written in a generic way and easily parallelised to take advantage of environments with multiple CPU cores.

Any collection can be transformed into a parallel collection, by calling `.par`. Higher order functions defined on the parallel collections will run in parallel. `map` and `filter` are naturally parallel and evaluated by splitting the collection up and evaluating the `map` or `filter` on each partition using different threads. Reduction operations need more careful consideration. `foldLeft` and `foldRight` are inherently sequential, however `reduce` can be implemented using an associative binary function and then the reduction can be performed in parallel using a parallel tree reduction.

Consider evaluating the likelihood of a linear model, see section 3.6.1 for the model specification. The likelihood can be written as:

$$p(y|\psi) = (2\pi\sigma^2)^{-\frac{N}{2}} \exp\left\{-\frac{1}{2\sigma^2}\sum_{i=1}^{N}(y_i - (\beta_0 + \beta_1 x))^2\right\}$$

The log-likelihood can then be written as:

$$\log p(y|\psi) = -\frac{N}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{N}(y_i - (\beta_0 + \beta_1 x))^2$$

This log-likelihood can be evaluated in Scala using a `map` followed by a `reduce`

```
ys.
  map { case (x, y) => Gaussian(beta0 + beta1 * x, sigma).logPdf(y) }.
  reduce(_ + _)
```

Where `ys: F[(Double, Double)]` is a collection of tuples containing the dependent and independent variables. To apply the function to a parallel collection, apply `.par` to the collection:

```
ys.par.
  map { case (x, y) => Gaussian(beta0 + beta1 * x, sigma).logPdf(y) }.
  reduce(_ + _)
```

The parallel reduction will only speed up processing if the collections are very large, or the individual likelihood contributions are expensive to evaluate.

If the data contained in the collection is too large to fit into the memory of a single computer, then higher order functions such as map and reduce can be used for distributed computing [Dean and Ghemawat, 2008]. This pattern was named MapReduce after the familiar higher order functions. MapReduce is commonly used when a large amount of input data can not fit in the memory of a single node. The large dataset is partitioned onto several worker nodes, orchestrated by a single master node. During the map stage, each worker node performs a function independently on each element of the data. Next, results are grouped by key and sent to appropriate worker nodes such that all data belonging to each key is stored on an individual worker machine. This if referred to as the shuffle stage. In the reduce stage a binary operator is used to combine the elements. If the data for a given key can not fit on a single worker node, then the binary reduction operator must be associative and can be performed in multiple steps across multiple worker nodes. The map and reduce steps can be performed by multiple nodes in parallel, meaning computationally intensive tasks can be sped up by increasing the number of worker nodes. In practice the speed of computation does not increase linearly, because of the cost of messaging between nodes during the setup and shuffle stages.

MapReduce is a general pattern which can be used to implement many algorithms, including the page rank algorithm used by Google to rank search results [Brin and Page, 1998]. MapReduce aims to make distributed programming easier by removing the possibility of race-conditions. Listing 2.5 shows a word count implementation written in a MapReduce style. The `map` stage assigns the number one to each word in the collection. The shuffle stage is represented by the function `groupBy` which groups by word, each group is a tuple where the first element is a single word and the second element is a collection of tuples containing the same word and the number one. The reduce stage is implemented using `map` and `reduce`, the `map` takes each tuple containing each word and collection and performs the binary reduction function `+` to add the numbers associated with each word. The `reduce` function is a `fold` without an initial value, as such it can only be performed on a non-empty collection.

```scala
val wordCount: Map[String, Int] = words.
  map(w => (w -> 1)).
  toList.
  groupBy(_._1).
  map { case (w, ws) => (w, ws.map(_._2).reduce(_ + _)) }
```

Listing 2.5: Illustrative MapReduce program written using Scala

Apache Spark [Apache, 2018b] is a platform for distributed computing for big data. Spark utilises a high level functional interface written in Scala allowing parallel processing for large datasets and streaming data. Spark primarily stores data in main memory of compute nodes, whereas Hadoop [Apache, 2018a] (a map reduce implementation) stores the data on the HDD or SSD of the worker nodes, and this means Spark is typically orders of magnitude faster for the same tasks. Apache Spark has additional capabilities, including batch processing of streaming data and a library for machine learning called MLlib. The main collection type in Apache Spark is the resilient distributed dataset (RDD) which is an immutable distributed monadic collection partitioned onto different nodes in the Spark cluster. The RDD can be operated on using the higher order functions familiar to users of the built in Scala collections.

2.3 Streams

Scala is eagerly evaluated, but can be lazy if required. A list is not explicitly lazy and hence can not represent an infinite data structure. In order to represent a lazy (possibly infinite) collection a `Stream` is defined. A stream is a cons list where the tail of the list is represented as a function of no arguments (sometimes called a thunk).

Infinite data structures are common in statistical programming, for instance a Markov chain representing a random walk can be defined as an infinite sequence, using the step function from listing 2.4:

```scala
Stream.iterate(0.0)(x => step(x).draw)
// Stream(0.0, ?)
```

`iterate` is a function `def iterate[A](x0: A)(f: A => A): Stream[A]` defined for all collection types in Scala, however with a stream it is not required to specify the resulting length of the collection.

2.3.1 Akka Streams

Sometimes datasets are too large to fit in to the main memory of a single machine. In some cases distributed programming can be used as introduced in section 2.2. However time series data arrives continuously and batch processing of time series data can be inefficient if online algorithms are available. Akka Streams[4] is a Scala library which defines its own stream data-type in order to facilitate large scale processing of streaming data from disparate sources. Akka Streams is built upon the actor model [Hewitt et al., 1973], a model for concurrent programming based upon message passing between actors.

Actors are straightforward to reason about, they can send and receive messages, create child actors and modify local state. Actors have a queue of messages received from other actors which they can respond to by doing work, updating internal state and sending messages. Since messages are "fire-and-forget" an actor can never be locked or waiting on work to complete, if there is a message in the mailbox to respond to the actor can work on it and send the message immediately. This is the key to the actor model of concurrency,

[4]http://akka.io

as exemplified by the Erlang programming language. In addition, there can be no race conditions as the actor only has local state and no shared state.

Actors can create child actors when performing new tasks.This allows for redundancy and if one child actor fails (for instance due to a hardware error) then a message is returned to the parent actor and the parent can spawn a new child to complete the task. Failures are sent through the network of actors in exactly the same way as any other message.

Akka streams is an abstraction built on top of Akka actors. Akka streams is an implementation of the reactive streams standard[5] providing tools for dealing with large, potentially unbounded streams of data from various sources. Akka streams can query multiple heterogeneous data sources without blocking and perform processing online as each observation arrives. If data is arriving too frequently for the processing stages, then backpressure can be used to reduce the rate of the producer. The backpressuring method is user specified. Some examples include thinning the stream or buffering elements if it is expected that the time between observations will eventually increase.

Akka Streams have three main components. `Source` represents the data flow and can be read from a database, web connection, file etc. A `Flow` is a processing stage and a `Sink` represents the termination of a data flow (writing to a file, printing to the screen etc.) Within this framework it is possibly to model large scale streaming data.

A `Source` is a stream, a lazy list which can be initialised in a number of ways, for example a pure `Source` consisting of a sequence, or an impure source such as reading from a file or database. `Source` has two type arguments, one which describes the actual type of the data and the other which describes the context of the data. A pure source consisting of a sequence of numbers will have type `Source[Int, NotUsed]`, since no context is required, Akka defines a special type `NotUsed`. If the same data is read from a database or file, then an asynchronous context is required to read from the data source and the type is `Source[Int, Future[IOResult]]`. Future is a monad which encapsulates a concurrent task, starting the connection to the database does not result in the main thread of the program blocking. The `Source` now has an associated context.

A `Flow[Int, A, NotUsed]` can be used to transform the stream of integers defined previously. The `Flow` can consist of any combination of one or more of the higher order functions defined for Akka streams, these include `scan`, `fold`, `map` recognisable from the Scala collections library. `Flows` are reusable transformations and can be applied to a `Source` with or without a context. A `Source` connected to a `Flow` defines another `Source`, as expected.

A `Sink` is the final connection needed to be able to run a computation within the Akka streams framework. No data flows through the Akka stream until it is fully connected to a `Sink` and run using a materialiser.

[5]http://www.reactive-streams.org/

Akka streams can be used to implement reactive programs which respond to new data continually. Consider an application which always runs, modelling an environmental process recorded on a sensor and transferred in real time to a database. Akka streams are an abstraction over the actor model, as such they require an actor system to run. The actor system and a materialiser are initialised:

```
implicit val system = ActorSystem("EnvironmentSensor")
implicit val materializer = ActorMaterializer()
```

These are initialised as `implicit` which means they do not have to passed explicitly to each method in the program. This reduces the amount of code and improves the readability of the main program. Next an Akka source for reading from a database is defined:

```
val connectDatabase: Future[Data]
val src: Source[Data, Future[IOResult] =
  Source.fromFuture(connectDatabase)
```

`connectDatabase` is a connection which returns the `Future` monad. A `Flow` is used to select the last 300 data points (updated live as new data arrives) and perform an inference algorithm:

```
def inference: Seq[Data] => Parameters
def myFlow = Flow[Data].
  sliding(300).
  map(inference)
```

Next, a `Sink` is defined which writes the result back to the database:

```
def mySink: Sink[Parameters, Future[IOResult]]
```

Finally the program can be run by connecting the source to the flow and sink then running it.

```
src.via(myFlow).runWith(mySink)
```

Constructing data flows like this is a natural way of processing streaming data and time series. Additionally, the Akka stream is a pure function, the side effects of reading and processing the data do not occur until the pipeline is run with a materializer, using `runWith`. The Akka stream can be thought of as a collection like `List` and hence instances of `Functor` and `Foldable` can be defined.

2.4 Summary

This chapter has outlined the properties of functional programming, with illustrative examples in Scala, that make it suitable and powerful for scientific and statistical applications. Static typing reduces difficult to find runtime errors and ensures that refactoring code is straightforward with the help of the compiler. This means code is not over-engineered from the start of project as changes can be made to any size code base as and when the specification grows. Static typing does not increase the size of the code base since well defined abstractions from category theory allow for polymorphic code with type safety. Type inference, allows return types which are obvious to be omitted from the source code and inferred by the compiler which is especially helpful when defining anonymous functions.

Higher order functions such as `map` and `reduce` are preferred over explicit "for" loops, these are easier to reason about and straightforward to parallelise on one machine (using parallel collections) or in a distributed environment (using Apache Spark). This makes speeding up slow running code on parallel architectures trivial to implement and test. When the aforementioned abstractions are used to write functions which are polymorphic over different collection types, then the same functions can be used to run serially or on a Spark cluster. Thus reducing runtime errors. This saves programmer time and compute resources.

Use of higher-order functions and type classes were illustrated with a novel functional implementation of a particle filter. The implementation safely encapsulates the side-effect of randomly advancing the particle cloud using a monad for random number generation, `Rand`. A reduction function `foldM` can be used to apply the function to any collection with a `Foldable` instance. This is an elegant way to write a statistical algorithm which can be applied to multiple collection types.

Finally, reactive Streams are a natural way to model unbounded time series data from a variety of services. Akka streams use the higher order functions familiar to functional programmers. `Flows` consisting of HOFs can be freely reused and combined to perform inference algorithms on live streaming data. The time series algorithms developed in chapters 4 to 7 are implemented using Akka streams.

Chapter 3

Probabilistic Programming

Bayesian inference using probabilistic programming is becoming of interest to large industry-backed artificial intelligence (AI) labs such as Google, Uber and Stripe [TensorFlow, 2018, Uber, 2018, Stripe, 2018]. Deep learning continues to solve inference problems with large amounts of labeled data allowing self-driving cars and image recognition along with many traditional applications of statistical models such as retention and campaign response modelling. However deep neural networks are typically used as a “black box” and state of the art networks can involve many parameters [Huang et al., 2017]. Some dense convolutional neural networks can have millions of parameters, the values of these parameters are impossible to reason about independently and require a large amount of data to learn.

In a Bayesian model, the likelihood and prior distribution must be specified using the judgment of the statistician and/or additional subject matter experts. This results in an interpretable model which can be used to explain relationships present in the data and perform accurate predictions with uncertainty. Bayesian methods are especially useful when an experiment has only a small number of observations, for instance in an A/B test using a small number of highly valuable customers or in a clinical trial where patient data is scarce. Bayesian methods provide a natural encoding for uncertainty; if only a small number of observations are available to inform the likelihood then the posterior distribution will reflect the prior. Specifying uncertainty honestly along with predictions can allow practitioners to make informed decisions in the presence of uncertainty. Well specified Bayesian models ensure industries such as insurance and finance, which price products and shares based on mathematical and statistical models, can be transparent in the decisions that they make.

Traditionally, parameter inference in Bayesian statistics has involved constructing bespoke MCMC schemes for each newly proposed statistical model. This results in new models being explored more slowly than practitioners would like, sometimes resulting in plausible models not being considered since efficient inference schemes are challenging to develop or implement. Probabilistic programming aims to reduce the barrier to performing Bayesian inference by developing a domain specific language (DSL) for model specification which does not rely on a detailed understanding of the parameter inference algorithms. In this way new models can be explored without relying on detailed understanding of the underlying inference methods.

There exist many DSLs for performing Bayesian inference. The most well known in the statistical community are BUGS [Lunn et al., 2000b] and Jags [Plummer et al., 2003] in which a simple syntax can be used to specify complex hierarchical models. BUGS and Jags both use Gibbs sampling to perform inference. Stan is another popular DSL which uses similar syntax to BUGS and Jags. Stan uses Hamiltonian Monte Carlo (HMC) with automatic differentiation for efficient sampling [Carpenter et al., 2016]. Stan also implements approximate variational inference schemes.

TensorFlow is a program created at Google for deep learning and neural networks. The core of TensorFlow has been used to build TensorFlow Probability [TensorFlow, 2018], a Python library for Bayesian inference utilising the TensorFlow architecture. TensorFlow probability uses automatic differentiation and GPU acceleration from TensorFlow for variational inference and HMC. Ridesharing company Uber have developed a Python library called Pyro [Uber, 2018] for probabilistic programming, marrying deep learning and Bayesian inference using PyTorch [Paszke et al., 2017] for automatic differentiation and GPU acceleration. Another probabilistic programming language written in Python of note is PyMC3 [Salvatier et al., 2016]. These probabilistic programming languages can be easily deployed in existing Python projects. However the syntax for implementing a model is more verbose than that of BUGS, JAGS and Stan. Hakaru [Narayanan et al., 2016] is an example of a probabilistic programming language with both an embedded DSL written in Haskell and a more accessible DSL written in the style of Python. This design choice was to minimise the learning curve of those coming from languages similar to Python.

An effort has been made to formalise the semantics of Bayesian inference in category theory by defining a probability monad grounded in measure theory [Giry, 1982, Lawvere, 1962]. A more pragmatic approach using the sampling based probability monad, built upon the State monad introduced in section 6.1.5, with applications to robot localisation and mapping is presented in Park et al. [2005]. A probability monad defined using measure terms for efficiently computing values such as the expectation of a distribution is introduced in Ramsey and Pfeffer [2002].

BUGS, Jags and Stan are written in DSLs which are in turn interpreted, or compiled to lower-level languages. For instance in a Stan script, arbitrary C++ can be written to define a model or inference scheme, but in a C++ program the Stan DSL for defining a model cannot be easily used. This is in contrast to an embedded DSL, where the DSL is written directly in the target language and can be deployed inside a larger program.

Rainier [Stripe, 2018] is an embedded probabilistic programming language written in Scala developed by the online payment processing company, Stripe. Rainier provides a monadic syntax for specification of models. The model has an associated compute graph which can be used to calculate the log of the un-normalised posterior and its gradient using automatic differentiation. The log-posterior and its gradient is then used for HMC [Duane et al., 1987]. The monadic interface is familiar to functional programmers and means that the well researched and developed features of existing functional programming languages can be used when developing probabilistic programs. This is demonstrated in the examples presented in this chapter: a random effects model is presented in section 3.6.3. The linear models required in the random effects model are reused from example 3.6.1. This code reuse is straightforward with a well developed programming language such as Scala and this is the main advantage of embedded probabilistic programming languages.

Embedded DSLs are straightforward to write in functional languages using, for example, free monads and generalised algebraic data types [Swierstra, 2008]. Ścibior et al. implements an algebra in Haskell for operating on distributions using a Generalised Algebraic Datatype and the Free Monad [Ścibior et al., 2015]. This allows the implementation details of complex programs (in this case statistical inference algorithms) to be decoupled from the model specification. Since the DSL is implemented in the same programming language, it is straightforward to use within existing programs. An alternative representation for an embedded probabilistic programming language using finally tagless encoding is considered in [Kiselyov and Shan, 2009].The importance of building DSLs for scientific computing is expanded upon in an article advocating the use of Lisp in bioinformatics [Khomtchouk et al., 2016]. Lisps are a collection of dialects of dynamically typed functional programming languages, which share many similarities with functional programming in Scala.

This chapter provides a high-level description of the implementation of several monadic probabilistic programming languages. The first defines a sampling-based probability monad which can be used to perform exact rejection sampling in section 3.2. It is acknowledged that exact rejection sampling is not scalable to real-world problems and more sophisticated parameter inference methods must be developed. In section 3.3 the Rainier probabilistic programming language is introduced and used to develop a Metropolis-Hastings algorithm. This section demonstrates how the modelling API is decoupled from the inference algorithm. The modelling API constructs a function representing the log of the un-normalised posterior (and its derivative) which can be used to implement Metropolis-Hastings. Section 3.4 describes methods for tuning Hamiltonian Monte Carlo algorithms, including dual-averaging, the No-U-turn sampler and eHMC [Wu et al., 2018]. The eHMC algorithm was implemented in Scala and contributed to the Rainier probabilistic programming language. This allows users to perform inference using HMC without specifying the step-size and number of leapfrog steps. sections 3.5.1 and 3.5.2 presents an overview of automatic differentiation, the key to implementing efficient gradient-based inference algorithms in probabilistic programs. Section 3.6 shows several examples of monadic probabilistic programs written using Rainier, the language is concise and embedded within the Scala programming language allowing straightforward code-reuse via function calls as exemplified in the random effects example in section 3.6.3. The probabilistic programming language is embedded in the host language which means all the features implemented in the host language are available to the user. This is much more straightforward than developing a programming language from the ground up. Finally, section 3.7 presents a novel probability monad using sampling importance resampling and presents an for inference of the filtering distribution of a partially observed Markov process model with a Poisson observation distribution.

3.1 Probability Monad

In order to define a probabilistic program using the abstractions introduced so far, it remains to show that probability measures form a monad. Monads are well understood and ubiquitous in functional programming and programming language theory, hence have well supported language features such as the for-comprehension in Scala and do-notation in Haskell. Their uniformity enables programmers to manipulate monadic values in a consistent way using `map` and `flatMap` independent of the monadic context. This makes defining probabilistic programs straightforward using powerful, well-developed functional programming languages with support for monads.

The foundations of the probability monad were first developed by Lawvere [1962] and it was extended in Giry [1982]. A review of the Giry monad is presented in Ramsey and Pfeffer [2002]. The Giry monad is defined on the category of measurable spaces, which is not a cartesian-closed category and hence is not suitable for applications of higher order functions, additionally distributions over functions are not supported. A probability monad based on Quasi-Borel spaces has been proposed to ameliorate these problems [Heunen et al., 2017].

This theoretical underpinning paves the way for developing monadic probabilistic programming DSLs in functional programming languages with minimal efford beyond specifying the two natural transformations required for the distribution to form a monad (see section 2.1.10).

3.2 A Sampling Based Probability Monad

The aim is to implement a domain specific language for probabilistic modelling and inference. This section considers a simple, sampling based probability monad with exact rejection sampling for inference. Later sections introduce more complex inference algorithms for performing inference using real-world models.

The first challenge is to specify a probability distribution in the programming language. The probability distribution must assign positive probability to each outcome and the probability of each outcome should sum to one. For discrete distributions with finite support this is straightforward to verify. As an example consider the Bernoulli distribution. The support of the Bernoulli distribution is $\{1, 0\}$ which can be interpreted as true or false. Then the probability of a 1 is assigned the value $p > 0$, and hence $p + (1 - p) = 1$. A trait for a general distribution which can be sampled from can be written as

```
trait Dist[A] {
  def probabilityOf(a: A): Double
}
```

Then the Bernoulli distribution can be written as

```
case class Bernoulli(p: Double) extends Dist[Boolean] {
  def probabilityOf(a: Boolean): Double =
    if (a) p else (1 - p)
}
```

The function `probabilityOf` is the probability mass function of the Bernoulli distribution. A function which verifies the distribution satisfies the probability axioms can be written as

```
def sumToOne[A](d: Dist[A], support: List[A]): Boolean =
  support.map(d.probabilityOf).sum == 1.0
def positive[A](d: Dist[A], support: List[A]): Boolean =
  support.map(d.probabilityOf).forall(_ >= 0)
```

The function `sumToOne` verifies that the probabilities corresponding to each possible outcome in the support of the distribution sum to one. `positive` verifies that the probability of each outcome is non-negative. However, when considering a distribution with infinite support, such as any continuous distribution, then all possible outcomes can not be enumerated. Consider a uniform distribution on a bounded interval $[0, 1]$, any value between 0 and 1 is equally likely and there are infinite values in the interval. The probability of a random variable distributed according to a continuous distribution taking a single value is zero. The probability of the random variable taking a value in a given interval is $Pr(a < X \leq b)$ which can be found using the cumulative distribution function of a continuous distribution. Individual traits can be defined for continuous and discrete distributions, continuous distributions can evaluate the (natural log of the) probability density function, `logPdf`, and discrete distributions can evaluate the (natural log of the) probability mass function, `logPmf`:

```
trait Continuous[A] {
  def logPdf(a: A): Double
  def gen: Rand[A]
}
trait Discrete[A] {
  def logPmf(a: A): Double
  def gen: Rand[A]
}
```

Listing 3.1: Abstract interfaces for discrete and continuous distributions

These traits replace the earlier `Dist` trait. Each distribution trait has a method `gen`, which can transform the distribution to a sampling based distribution, `Rand`. `Rand` is defined with the following interface:

```
trait Rand[A] { self =>
  def draw(): A
  def sample(n: Int): List[A] = List.fill(n)(draw)
  def map[B](f: A => B): Rand[B] = new Rand[B] {
    override def draw = f(self.draw)
  }
  def flatMap[B](f: A => Rand[B]): Rand[B] = new Rand[B] {
    override def draw = f(self.draw).draw
  }
  def pure[A](a: A): Rand[A] = new Rand[A] {
    override def draw = a
  }
}
```

Then the only function which remains abstract and must be defined for a new `Rand` is `draw`, hence this monad has no information about the `logPdf` or `logPmf` functions. `pure` is represented by the dirac distribution. `map` and `flatMap` provide ways of transforming and combining distributions to build complex statistical models from individual distributions.

The function `map` can be used to apply a function to elements inside of the distribution, for example a uniform distribution on the interval $[0, 1]$ can be transformed to a new support using `map`:

```
def uniform(a: Double, b: Double): Rand[Double] =
  Uniform.map(x => x * (b - a) + a)
```

To show an application of `flatmap` consider building a Bayesian statistical model where each parameter in the model has a prior distribution. The joint prior distribution of the parameters is combined with the likelihood in order to calculate the posterior distribution of the parameters given the observed data. Consider a coin flip experiment with a biased coin, the data consists of n coin flips and the unobserved parameter of interest is the probability of heads, p_h. A beta prior distribution is chosen for the parameter and the likelihood is the Binomial distribution with n trials and probability of success p_h. `flatMap` can be used to combine the prior and the likelihood to determine the posterior distribution of p_h:

```scala
val a = 1
val b = 1
val n = 10
val program: Rand[(Double, Int)] = Beta(a, b).flatMap(p =>
  Binomial(n, p).map(y => (p, y)))
```

If `map` was used then the program would be of type `Rand[Rand[(Double, Int)]]`. This program can then be used to forward simulate data from the model:

```scala
program.draw
// (0.7444018968952699,6)
```

This tuple consists of a draw from the prior distribution for $p = 0.744...$ and the number of heads observed, 6. Consider the problem of estimating the posterior distribution of the parameter p_h using this sampling monad given that we know 6 heads were observed from the 10 coin flips. The program can be sampled from many times, conditional on the number of observed heads. This inference algorithm is exact rejection sampling, and works by computing many realisations from the forward model and matching the resulting samples to the data. To implement exact rejection sampling, define `condition` which filters the draws which match a predicate `c`

```scala
def condition(c: A => Boolean): Dist[A] = new Dist[A] {
  override def draw = {
    val a = self.draw
    if (c(a)) a else this.draw
  }
}
```

The for comprehension (see section 2.1.10) can be used to express the `program` as:

```scala
val program: Dist[(Double, Int)] = for {
  p <- Beta(a, b)
  heads <- Binomial(n, p)
} yield (p, heads)
```

Listing 3.2: Probabilistic program expressing a coin flip example with a beta prior distribution for the probability of heads

For a more complete sampling based probability monad in Scala, see [1]. The posterior distribution of p_h can be determined using the `condition` function:

```
program
  .condition { case (ph, heads) => heads == 6 }
  .sample(1000)
// (0.5728470229411422,6)
// (0.7147104470387118,6)
// (0.6365933523815765,6)
// (0.7618795585149065,6)
// ...
```

The program then returns 1,000 draws, each draw has six heads out of ten total coin flips. The first argument of the tuple represents draws from the posterior distribution of the probability of heads - summary statistics such as the mean can then be calculated. This is not a very efficient method of inference and as the number of trials increases then selecting draws which match the predicate `c` becomes more difficult and results in many wasted computations. However, the probabilistic program expression (`program`) is concise and clear in its intent, this monadic syntax can be used with more efficient inference algorithms as expanded upon in the following sections.

This method of inference does not use the `logPdf` or `logPmf` as defined for the `Continuous` and `Discrete` distribution traits. Inference algorithms such as Metropolis-Hastings and HMC use the log of the un-normalised posterior to perform inference efficiently. These distributions can be used as a monad with a different specification for `flatMap` which combines the distributions in order to generate the log of the un-normalised posterior distribution. This is expanded upon in section 3.3.

3.3 Metropolis-Hastings

The sampling based probability monad introduced in section 3.2 is inefficient for inferential purposes. In the case of the coin flip example (and many other statistical models) the likelihood is available and is the probability mass function of the Bernoulli distribution. Metropolis-Hastings (see section 1.3.2) is a popular method for performing inference for models with intractable posterior distributions, where the likelihood can be specified and evaluated up to a normalising constant. In a Bayesian model, the prior distribution of

[1]https://github.com/jliszka/probability-monad/

the parameters is also required, the log-density of the prior can be summed with the log-likelihood, forming a function from the parameters to the log of the un-normalised posterior.

The Metropolis-Hastings algorithm requires a function for the log of the un-normalised posterior `val posterior: P => Double`. In addition a proposal distribution is required of the form `val prop: P => Dist[P]`. A common choice for the proposal distribution in MH is a symmetric multivariate Normal distribution with mean equal to value of the previously accepted parameter value in the Markov chain and the variance left as a tuning parameter. These two functions depend on the model specification, but the rest of the algorithm is identical for any model:

```scala
def step[P](
  pos: P => Double,
  prop: P => Rand[P]): Rand[P] = { p: P =>
  for {
    ps <- prop(p)
    a = pos(ps) - pos(p)
    u <- Uniform(0, 1)
    next = if (log(u) < a) ps else p
  } yield next
}
```

Listing 3.3: Kernel of the Metropolis algorithm using a symmetric proposal distribution

The function `step` implements a single step of the Metropolis algorithm, assuming the proposal distribution is symmetric. The Markov kernel can then be used in an `iterate` function to form a lazily evaluated stream (see section 2.3). The stationary distribution of the Markov chain is equal to the posterior distribution, and the samples from the Markov chain can be used to construct any summary of the posterior distribution which is required.

In Ścibior et al. [2015], the probability monad is implemented as a free monad, and several inference algorithms are developed and interpreted using a sampling-based interpreter. The model specification language is independent of the chosen inference algorithms, and additionally the inference algorithms can be composed. An example in the paper shows how straightforward it is to compose a particle filter with a Metropolis-Hastings algorithm to implement a pseudo-marginal Metropolis-Hastings algorithm (see section 5.3). Among the inference algorithms is a Metropolis-Hastings algorithm which utilises the prior distribution as the proposal distribution for the parameters. The reason for this choice is that there was no way to collect the information from the model specification to implement a generic proposal distribution function as mentioned in section 8 of the paper.

The Scala library Rainier[Stripe, 2018] can be used to address this problem. Rainier is a probabilistic programming language which uses HMC for sampling, however the sampler is independent of the model building API and hence new samplers can be implemented which utilise the monadic model building API.

Rainier builds a compute graph over the parameters of a model. The compute graph is essentially a variadic function (a function which an undetermined number of arguments) from many input parameters to an output. This variadic function is a representation of the un-normalised log posterior function and can return both the evaluation of the function and the gradient (using automatic differentiation, see section 3.5). The compute graph can be compiled to a function with signature `Array[Double] => Double` representing the evaluation of the log of the un-normalised posterior or `Array[Double] => Array[Double]` representing the gradient. The function built using the compute graph is used to implement a Metropolis algorithm.

The distributions defined in listing 3.1 can be used to build the function to evaluate the log of the un-normalised posterior. A new function `param` is defined which allows the distribution to be used as a prior distribution. When defining the proposal distribution for the parameters, the support must be respected. A straightforward way to implement this is to appropriately transform the parameter such that an unconstrained random walk proposal can be used. The distribution trait can be augmented with the additional methods:

```scala
def transform(x: A): A
def logJacobian(x: A): Double
```

When the parameters are added to the compute graph, using `param` they are transformed to the support of the distribution when evaluating the log-density. The log-Jacobian of the transformation is added to the log-density and this is needed since the parameters are proposed in the unconstrained space. The function `param` can then be defined for a continuous distribution as

```scala
def param = {
  val x = new Variable
  val constrained = transform(x)
  val density = logPdf(constrained) + logJacobian(x)
  RandomVariable(constrained, density)
}
```

This is similar for a discrete distribution, but the `logPdf` is replaced with the `logPmf`. Table 1.1 summarises possible transformations for bounded parameters. Note that the derivatives of the transformations are not needed for the Metropolis algorithm.

Additionally, each distribution has a `fit` method which allows the `logDensity` of that distribution to be used as the likelihood. The coin flip model can then be specified as a Rainier model with some small changes

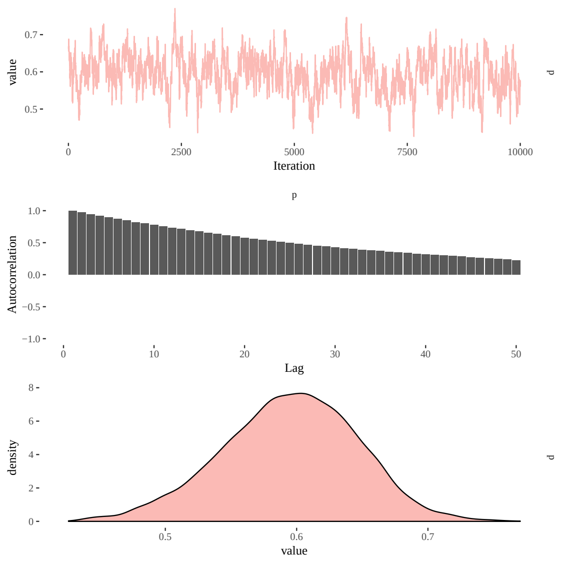


Figure 3.1: Diagnostics for the posterior distribution of the probability of heads, p in a coin flip experiment. The traceplots of the MCMC draws, (top). The autocorrelation function for the posterior distribution of p showing a slow decay, indicating correlated samples from the posterior, (middle). The empirical marginal posterior density of p, (bottom).

```
val model = for {
  p <- Beta(3, 3).param
  _ <- Binomial(p, 10).fit(6)
} yield p
```

Listing 3.4: Rainier probabilistic program expressing a coin flip example with a beta prior distribution for the probability of heads

The model can be sampled from using the Metropolis algorithm with a random walk proposal $q(\psi^\star|\psi) = \mathcal{N}(\psi^\star|\psi, 0.05I_p)$ where I_p is the $p \times p$ identity matrix:

```
model.sample(Metropolis(0.05), 1000, 10000, 5)
```

This performs Metropolis samples with 1,000 burn-in followed by 10,000 sampling iterations with a thin of 5 resulting in 2,000 samples from the posterior distribution. The argument to the `Metropolis` function is the diagonal element of the Normal proposal distribution. This implementation of the Metropolis algorithm could be changed in order to accept as an argument a covariance matrix for the symmetric normal proposal distribution. Alternatively, the function could accept an arbitrary proposal distribution and density and hence be used as a Metropolis-Hastings sampler.

Figure 3.1 shows the samples from the posterior distribution of the parameter p_h from the coin-flip model.

This implementation of the Metropolis algorithm means the specification of the model is decoupled from the inference algorithm. The log of the un-normalised posterior is generated from the compute graph of the `model` in listing 3.4 which can be specified using the monadic DSL. While this inference method is straightforward to implement, widely applicable and guaranteed to converge, it is not necessarily efficient. A more efficient method of generating samples from the posterior distribution is required for fitting real world models.

3.4 Automatic Tuning of HMC

One of the aims of probabilistic programming is to separate the details of the inference algorithm from the implementation of the statistical model. This allows for rapid exploration of different models without having to devise new inference algorithms. Metropolis-Hastings has convergence guarantees and if run for long enough will give samples from the posterior distribution. However this can often be an unacceptably long time. The proposal distribution requires tuning in order to achieve the optimal acceptance rate (which is subject to conditions), of around 0.234 [Roberts et al., 1997]. Adaptive methods can be used to tune the proposal distribution automatically [Atchadé et al., 2005]. Even optimal tuning of the proposal distribution does not take into account the geometry of the posterior distribution, which is contained in the gradient of the posterior. More efficient gradient-based algorithms can be used for drawing samples from the posterior distribution.

Hamiltonian Monte Carlo was first introduced in section 1.3.4 and can be implemented using functional programming principles. The components of a statistical model which are required to use HMC for parameter inference are the log of the un-normalised posterior distribution (as in the Metropolis-Hastings algorithm implemented in section 3.3) and the gradient of the log of the un-normalised posterior.

The Hamiltonian Monte Carlo algorithm provides an efficient proposal distribution for parameters by augmenting the parameter space with a momentum parameter. However, HMC requires the user to specify the step size and number of leapfrog steps. If the step size is too small and number of leapfrog steps too low this can reduce the efficiency of the sampler to that of a random walk, however if the number of leapfrog steps is too large the algorithm can traverse the parameter space first in one direction and then return to an area it has already traversed, performing a U-turn and wasting computation. Dual averaging can be used during the warm-up phase of the Markov chain to determine a suitable leapfrog step size [Nesterov, 2009]. The modified dual averaging scheme presented in Hoffman and Gelman [2014] is summarised below.

The purpose of setting an appropriate step size in the HMC algorithm is to avoid wasting computation. The acceptance rate is the governing quantity, hence define the statistic $H_k = \delta - \alpha_k$ where δ is the target acceptance rate, α_k is the acceptance probability at iteration k and ε is the step-size of the HMC algorithm. Then define the expectation of H_k:

$$h(\varepsilon) = \mathbb{E}(H_k|\varepsilon) \approx \frac{1}{k}\sum_{i=1}^{k} \mathbb{E}(H_i|\varepsilon) \tag{3.1}$$

The aim of the dual averaging algorithm is to reduce this expectation to zero and hence the actual acceptance rate converges in mean to the target acceptance rate. Values of the tuning parameter are updated using,

$$\log \varepsilon_k = \mu - \frac{\sqrt{k}}{\gamma(k + k_0)} \sum_{i=1}^{k} H_i \tag{3.2}$$

where γ and μ are free parameters, μ is the value which $\log \varepsilon_k$ is shrunk towards and γ controls the rate at which the step-size is shrunk. The parameter k_0 is a free-parameter chosen to stabilise the algorithm during early iterations by avoiding unrealistically large values of the step size. The second term in equation (3.2) will converge to zero as the number of iterations increases. A sequence of weighted averages is used to calculate the final step-size

$$\log \bar{\varepsilon}_k = \eta_k \log \varepsilon_{k-1} + (1 - \eta_k) \log \bar{\varepsilon}_{k-1} \tag{3.3}$$

which converges for values of $\eta_k = k^{-\beta}$ with $\beta \in [0.5, 1]$. Which means as $k \to \infty$ the value of $\log \bar{\varepsilon}_{k-1} - \log \bar{\varepsilon}_k \to 0$ as required. The dual averaging algorithm is run for a pre-determined number of warmup iterations at the start of the HMC algorithm using the value of ε_k at each leapfrog step. At the end of the warmup run, the value $\bar{\varepsilon}_k$ is used in the final monitoring run of the HMC sampler. It is known that the optimal leapfrog step size at the stationary distribution can be different to that of the warmup phase, hence sometimes long warmup runs are required. The recommended value for the target acceptance rate is $\delta = 0.65$. Algorithm 5 presents an implementation of the function required to update the step size during the warmup phase of an HMC chain at each iteration of the HMC algorithm given in algorithm 3. Updating the step size is performed after line 9 in the HMC algorithm, just after the log-acceptance probability, $\log \alpha$ is calculated. For initialisation at $k = 0$ set $H_0 = 0$.

Algorithm 5 is suitable for selecting the step-size during the warm-up phase of HMC, Hoffman and Gelman discuss ways of defining H_t and hence selecting the optimal step-size in the No-U-Turn sampler, since there is no single accept/reject step.

After the tuning of the step-size, an appropriate number of leapfrog steps is calculated in order to avoid performing a U-turn and wasting computation. To test for the presence of a U-turn the following calculation is performed:

Algorithm 5: Update Step Size function for performing Dual Averaging in an HMC algorithm to automatically select an appropriate step size

```
1 function UpdateStepSize($k$, $\log \alpha$, $H_{k-1}$, $\delta = 0.65$, $\mu = \log(10\varepsilon_0)$, $\beta = 0.75$,
    $\gamma = 0.05$, $k_0 = 10.0$):
2     Calculate $R = 1/(k + k_0)$;
3     Calculate $H_k = (1 - R)H_{k-1} + R(\delta - \alpha)$;
4     Set $\log \varepsilon_k = \mu - \frac{\sqrt{k}H_k}{\gamma}$;
5     Set $\log \bar{\varepsilon}_k = k^{-\beta} \log \varepsilon_k + (1.0 - k^{-\beta}) \log \bar{\varepsilon}_{k-1}$;
6     return $H_k$, $\log \varepsilon_k$, $\log \bar{\varepsilon}_k$;
```

$$\frac{d}{dt}\left[\frac{1}{2}(\psi^\star - \psi) \cdot (\psi^\star - \psi)\right] = (\psi^\star - \psi) \cdot \phi^\star \tag{3.4}$$

where $\psi^\star$ represents the proposed parameter after $\ell \leq L$ leapfrog steps, ψ represents the parameter at the start of the iteration before any leapfrog steps and $\phi^\star$ is the momentum after ℓ leapfrog steps. This is the time derivative of the distance between the initial position and final position. If the calculated derivative is less than zero, the sampler has performed a U-turn and further leapfrog steps are proposing parameters in an area which has already been explored.

Empirical Hamiltonian Monte Carlo (eHMC) can be used to learn a distribution of the longest steps until a U-turn [Wu et al., 2018]. eHMC begins by assuming an appropriate step-size has been selected and an initial value for the number of leapfrog steps, ℓ_0 has been specified. Then the empirical distribution of longest steps until U-turn can be determined. To construct the empirical distribution, the number of leapfrog steps in a single direction are calculated until the U-turn condition in equation (3.4) is less than zero and this value is recorded as ℓ_u. If $\ell_u > \ell_0$ then the value of ϕ and ψ at ℓ_0 steps are used as the new proposed values. Alternatively, if $\ell_u < \ell_0$ then a further $\ell_0 - \ell_u$ leapfrog steps are performed. This ensures the position and moment are advanced exactly ℓ_0 leapfrog steps and determines the number of steps until a U-turn without wasting computation. The proposed values of ψ and ϕ are then used in an unaltered HMC algorithm. The values of ℓ_u at each step are recorded during the warm-up phase to form the empirical distribution of longest steps until U-turn.

During the main sampling run of eHMC, the empirical distribution of longest step lengths is sampled from to determine the number of leapfrog steps to perform in the HMC kernel. The key to the effectiveness of this algorithm is that the number of leapfrog steps is independent of the current position of Markov chain. eHMC is implemented in Rainier by the author as a tuning free method for performing inference using HMC.

The NUTS algorithm is more complex and a calculation for the number of steps is performed at each iteration of the main sampling run. The condition in equation (3.4) is used in order to determine the number of leapfrog steps to perform. However it is noted that selecting the number of steps directly using this condition at each iteration means that time reversibility is not guaranteed. The leapfrog steps are run both forward and backward in time to preserve reversibility.

Firstly the momentum is sampled from a multivariate Normal distribution $\phi^\star \sim \mathcal{N}(0, I_p)$, then a slice variable is sampled from $u \sim U(0, \exp(\log p(\psi|y) - \frac{1}{2}\phi^{\star T}\phi^\star))$. Then a binary tree is constructed by sampling a direction, $v \in \{-1, 1\}$ and taking a number of leapfrog steps with step-size $v\varepsilon$. At each level, j of the tree, 2^j leapfrog steps are taken, for each step if $\log(u) \leq \log p(\psi^\star|y) - \frac{1}{2}\phi^{\star T}\phi^\star$ then the proposed pair is added to the candidate set. The binary tree stops growing when either direction has made a U-turn or the position corresponds to area of extremely low likelihood. If a branch of the tree makes a U-turn, the level of the tree corresponding to that branch is not added to the candidate set. Finally, the position (and momentum) is sampled uniformly from the candidate set. A more recent version of NUTS uses multinomial resampling of the trajectories instead of the slice-sampling step described above [Betancourt, 2017].

The combination of dual-averaging and eHMC or NUTS allows for "tuning-free" inference algorithms, which empowers practitioners to explore new models without implementing new inference algorithms. There are limitations to these algorithms, however, in that they require an log of the un-normalised posterior function which can be both evaluated and differentiated. Hence, if there does not exist a re-parameterisation, discrete parameters can not be used as this introduces discontinuities in the log of the un-normalised posterior and results in the function not being differentiable. Recently gradient based proposal methods for PMMH (see section 5.3) have been explored by approximating both the gradient and the likelihood using the particle filter [Nemeth et al., 2016].

3.5 Automatic Differentiation

The need to calculate gradients in many inference and optimisation algorithms such as Hamiltonian Monte Carlo (HMC), variational inference [Kucukelbir et al., 2017] and the Metropolis-adjusted Langevin algorithm [Roberts and Rosenthal, 1998], among others has led to an interest in automatic differentiation (AD). AD is a way of calculating derivatives of functions while at the same time evaluating them. This is not numerical differentiation or symbolic differentiation but rather exact differentiation which returns the value of a derivative at a point [Wang et al., 2018]. It is straightforward to implement forward mode automatic differentiation in a class based language. However, the number of computations performed using forward mode AD depends on the dimension of the input space, and hence does not scale well to parameter inference in models involving a large number of parameters. On the other hand the number of computations required for reverse mode automatic differentiation scales with the dimension of the output. In gradient based inference methods used for Bayesian inference the gradient of the log-posterior with respect to the free-parameters is required, this is a function from $\mathbb{R}^n \rightarrow \mathbb{R}$ and hence reverse mode AD is typically more efficient. Automatic differentiation can be used to automate the calculation of exact derivatives needed when performing the leapfrog steps in HMC.

3.5.1 Forward Mode Automatic Differentiation

Dual numbers can be used in order to calculate derivatives automatically and exactly. Each real number has a corresponding dual number, which is the number, $y \in \mathbb{R}$ plus a small innovation, ε such that $\varepsilon^2 = 0$.

Derivatives can be calculated by evaluating functions using the dual number, for instance the function $f : \mathbb{R} \rightarrow \mathbb{R}$ defined by $f(x) = x^2+2x+5$, the derivative is $f'(x) = 2x+2$. In order to calculate the derivative automatically using dual numbers, the function is evaluated using the dual number equivalent to a chosen value of x for instance $x = 5$, has the dual number $5 + \varepsilon$ then

$$\begin{aligned}
f(5+\varepsilon) &= (5+\varepsilon)^2 + 2(5+\varepsilon) + 5, \\
&= 25 + 10\varepsilon + \varepsilon^2 + 10 + 2\varepsilon + 5, \\
&= 40 + 12\varepsilon.
\end{aligned}$$

In this way, the evaluation of f using the dual number has resulted in the evaluation of $f(x)$, and $f'(x)$ simultaneously with $f'(x)$ being given by the coefficient of ε. This is the essence of forward mode automatic differentiation.

Dual numbers can be used to implement forward mode automatic differentiation in Scala by overloading the numeric operators needed to implement the desired function. Firstly a class representing the dual number is defined as an algebraic data type:

```scala
case class Dual(real: Double, eps: Double)
```

Then primitive numeric operators are implemented, with their associated derivative in the `eps` argument:

```scala
def plus(x: Dual, y: Dual) =
  Dual(x.real + y.real, x.eps + y.eps)
def minus(x: Dual, y: Dual): Dual =
  Dual(y.real - x.real, y.eps - x.eps)
def times(x: Dual, y: Dual) =
  Dual(x.real * y.real, x.eps * y.real + y.eps * x.real)
def div(x: Dual, y: Dual) =
    Dual(x.real / y.real,
    (x.eps * y.real - x.real * y.eps) / (y.real * y.real))
```

Listing 3.5: Primitive functions implemented using dual numbers

The operators can be used to simultaneously evaluate a function and its derivative. Note that the function `times` uses the product rule and `div` uses the quotient rule to calculate the second argument. Consider the function $g(x) = 5x$ with derivative $g'(x) = 5$. A function with type `Dual => Dual` can be implemented using the functions defined in listing 3.5

```
def f(x: Dual): Dual = times(x, Dual(5, 0))
```

The function can then be evaluated at $x = 3$, substitution is used to reason about the evaluation of the function

```
f(Dual(3, 1)) = times(Dual(3, 1), Dual(5, 0))
 = Dual(3 * 5, 1 * 5 + 0 * 3)
 = Dual(15, 5)
```

Constants are represented as a `Dual` number with zero in the `eps` argument. Special functions such as log, exp and trigonometric functions can be can be implemented in terms of dual numbers. The `Numeric` type class can be used to implement the operators, then the function can be written exactly the same as a function of built in numeric types. This implementation allows *univariate* functions to be automatically differentiated.

Example: Normal Likelihood

Consider n-independent observations assumed to follow a Normal distribution:

$$y_i \sim \mathcal{N}(\mu, 2^2), \quad i = 1, \dots, n, \tag{3.5}$$

with known standard deviation $\sigma = 2$. The prior distribution is chosen to be $\mu \sim \mathcal{N}(0, 1)$ and the log-posterior is

$$\begin{aligned} p(\mu|y) &\propto p(\mu)\prod_{i=1}^{N} p(y_i|\mu) \\ &= (2\pi)^{-\frac{1}{2}} \exp\left(-\frac{\mu^2}{2}\right)(8\pi)^{-\frac{N}{2}} \exp\left(-\frac{1}{8}\sum_{i=1}^{N}(y_i-\mu)^2\right) \end{aligned}$$

Then taking the log gives

$$\log p(\mu|y) = -\frac{1}{2}\log(2\pi) - \frac{\mu^2}{2} - \frac{N}{2}\log(8\pi) - \frac{1}{8}\sum_{i=1}^{N}(y_i-\mu)^2$$

The kernel of the log-posterior is a function of `mu`, defined as a `Numeric` value:

```
def logPos[A: Numeric](ys: Vector[Double], mu: A) =
  - 0.5 * (mu * mu) - 0.125 * ys.
    map(_ - mu).
    map(x => x * x).
    reduce(_ + _)
```

The derivative of the log-posterior with respect to μ can be calculated by-hand

$$\frac{\partial \log p(\mu|y)}{\partial \mu} = -\mu + \frac{1}{4}\sum_{i=1}^{N}(y_i - \mu)$$

When the function `logPos` is evaluated it should return the log-posterior and the derivative in a `Dual` object. The derivative calculated using dual numbers can then be compared to the analytic derivative.

```
logPos(ys, Dual.pure(2.0))
```

Automatic differentiation using dual numbers is equivalent to applications of the chain rule:

$$(f \circ g)' = (f' \circ g)g' \tag{3.6}$$

Combinations of primitive functions can be differentiated using repeated applications of the chain rule. `flatMap` can be defined for the `Dual` class which encapsulates the chain rule. This presentation of forward mode automatic differentiation was first outlined by [Welsh, 2018].

```
case class Dual[A: Numeric](real: A, eps: Double) {
  def flatMap[B: Numeric](f: A => Dual[B]): Dual[B] =
    val x = f(real)
    val nextEps = eps * x.eps
    Dual(x.real, nextEps)
}
```

The `Numeric` context bound is required since the primitive functions, such as multiplication, are defined explicitly in terms of their real result and derivative; hence an `eps: Double` has to be multiplied by a `real: A` when applying the product rule for differentiation:

```
def times(x: Dual[A], y: Dual[A])(implicit ev: ConvertableFrom[A]) =
  Dual[A](x.real * y.real,
    x.eps * ev.toType[Double](y.real) +
      y.eps * ev.toType[Double](x.real))
```

The derivatives of special functions can be written as:

```
def sin[A: Numeric](a: Dual[A])(implicit ev: A =:= Double) =
  Dual(math.sin(real), math.cos(real) * eps)
def log[A: Numeric](a: Dual[A])(implicit ev: A =:= Double) =
  Dual(math.log(a.real), a.eps / a.real)
```

The derivatives of the special functions can also be written using `flatMap`

```
def sin(a: Dual[A])(implicit ev: A =:= Double) =
  a.flatMap(x => Dual(math.sin(x), math.cos(x)))
def log(a: Dual[A])(implicit ev: A =:= Double) =
  a.flatMap(x => Dual(math.log(real), 1 / real))
```

Implementing forward mode AD using a monad and the `Numeric` type class means the function $g(x) = 5x$ can be written as:

```
def g[A: Numeric](x: A): A = x * 5
```

Then `flatMap` can be used to apply another function with signature `Double => Dual[Double]`. If the function applied is $h(x) = 2x$ then the expected result evaluated at $x = 3$ is $h(g(3)) = 30$ and $(h \circ g)'(3) = 10$. The function can be applied as follows

```
def h(x: Double): Dual[Double] = pure(x).times(const(2.0))
g(Dual.pure(3.0)).flatMap(h)
// Dual(30.0, 10.0)
```

The implementation becomes more complex as functions of multiple arguments are considered, since a separate ε is required for the derivative with respect to each argument [Manzyuk, 2012]. Figure 3.2 is a representation of a function with multiple arguments $f(x_1, x_2) = x_1x_2 + x_1^2$. The arrows represent function application, intermediate nodes represent the result of each primitive function and are labelled x_3 and x_4.

Differentiation of the function with respect to all variables in the graph using forward mode automatic differentiation requires two traversals of the graph. The derivative of a node, x_k with parent nodes $Pa(x_k)$ with respect to the input variable x_1 is derived from the chain rule

$$\frac{\partial x_k}{\partial x_1} = \sum_{j \in Pa(x_k)} \frac{\partial x_k}{\partial x_j}\frac{\partial x_j}{\partial x_1}. \tag{3.7}$$

Figure 3.3 shows the forward pass required to calculate the gradient $\left.\frac{\partial f}{\partial x_1}\right|_{x_1=1, x_2=2}$. This single forward pass evaluates the function and the derivative with respect to x_1 given the known derivatives of primitive functions such as product and power. A second pass has to be computed in order to determine the derivative with respect to x_2. The gradient

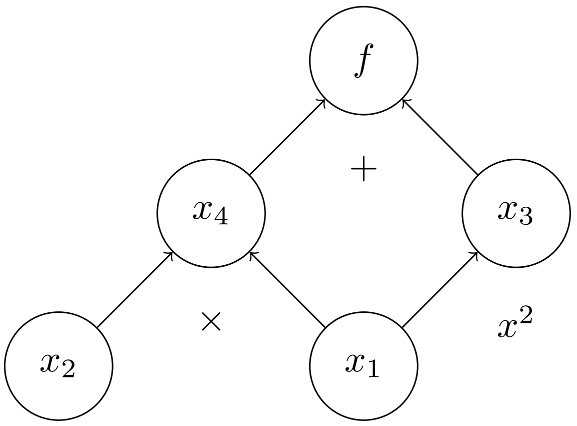


Figure 3.2: A computational graph showing the decomposition of $f(x_1, x_2) = x_1x_2 + x_1^2$ into primitive functions.

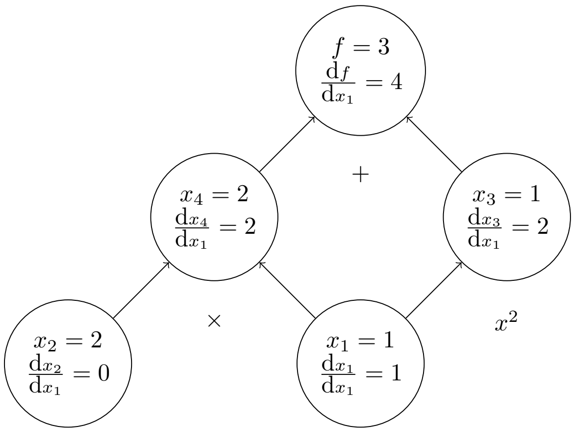

$f = 3$
$\frac{\mathrm{d}f}{\mathrm{d}x_1} = 4$
$+$
$x_4 = 2$
$\frac{\mathrm{d}x_4}{\mathrm{d}x_1} = 2$
$x_3 = 1$
$\frac{\mathrm{d}x_3}{\mathrm{d}x_1} = 2$
$\times$
x^2
$x_2 = 2$
$\frac{\mathrm{d}x_2}{\mathrm{d}x_1} = 0$
$x_1 = 1$
$\frac{\mathrm{d}x_1}{\mathrm{d}x_1} = 1$

Figure 3.3: A computational graph showing the applications of the chain rule required to calculate the partial derivative $\frac{\partial f}{\partial x}$.

calculated by hand is, $\frac{\partial f}{\partial x_1} = x_2 + 2x_1$

A method used to implement automatic differentiation for functions of multiple arguments $f : \mathbb{R}^n \to \mathbb{R}$ is to track a vector of derivatives. The definition of the `Dual` class can be altered (now called `DualV` for vector) to include multiple derivatives:

```
case class DualV(real: Double, dual: Vector[Double])
```

Then functions are defined to create dual numbers from constants and variables. However the dimension of the resulting function is required a-priori, the dimension corresponds to the number of parameters in the log-posterior since the length of the vector `dual` must be equal to the number of parameters:

```
def constant(x: Double, dim: Int) =
  DualV(x, Vector.fill(dim)(0.0))
def variable(x: Double, dim: Int, index: Int) =
    DualV(x, Vector.fill(dim)(0.0).updated(index, 1.0))
```

The operators must be defined again for the class `DualV`, however they are identical to the univariate calculations and are performed element-wise. The location of the indices determines if each parameter should be treated as a constant or a variable when calculating the `dual` argument.

Then to define the log-posterior of a model involving multiple parameters each constant must be lifted into dual number using `constant`, with the variables lifted using `variable`. Consider the same Normally distributed example from equation (3.5), but this time the standard deviation is also unknown. The prior for the mean is Normal and the prior for the variance is the inverse Gamma distribution with shape and rate, $\alpha = 3, \beta = 3$. The log-density of the prior distributions can be written as:

```
  def priorMu(mu: DualV): DualV = {
    val muMu = const(0.0, 2)
    val sigmaMu = const(1.0, 2)

    -0.5 * log((0.5 * const(math.Pi, 2)) * sigmaMu) -
      0.5 * ((mu - muMu) * (mu - muMu))
  }
  def priorSigma(sigma: DualV): DualV = {
    val alpha = const(3.0, 2)
    val beta = const(3.0, 2)

    -(alpha + 1) * log(sigma) - (beta / sigma)
  }
```

Then the log-likelihood for the model can be written as:

```
  def ll(ys: Vector[Double])(mu: DualV, sigma: DualV): DualV = {
    -const(ys.length, 2) * log((2 * const(math.Pi, 2)) * (sigma * sigma)) -
     (const(0.5, 2) / (sigma * sigma)) * ys.
       map(y => const(y, 2) - mu).
       map(x => x * x).
       reduce(_ + _)
  }
```

Where `ys` is the observed data. Notice that each constant has to be lifted into the context of the dual number using `const` with the correct dimension. Now the log-posterior is the sum of the log-priors and the log-likelihood. When the log posterior is evaluated using the variables `mu` and `sigma` the log-posterior and the gradient will be returned simultaneously in the `DualV` class.

The log-likelihood and gradient information can be used to implement HMC. Au-

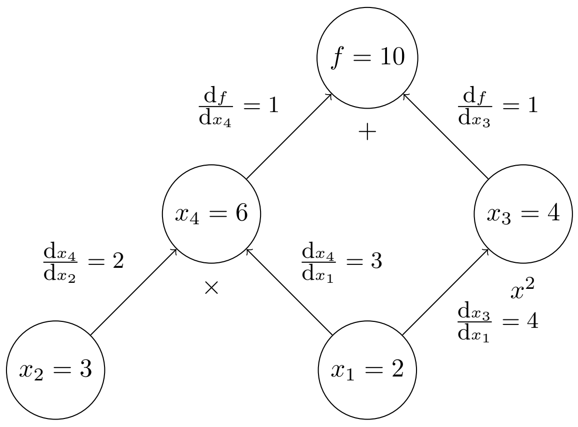

$f = 10$
$\frac{\mathrm{d}f}{\mathrm{d}x_4} = 1$
$+$
$\frac{\mathrm{d}f}{\mathrm{d}x_3} = 1$
$x_4 = 6$
$x_3 = 4$
$\frac{\mathrm{d}x_4}{\mathrm{d}x_2} = 2$
$\times$
$\frac{\mathrm{d}x_4}{\mathrm{d}x_1} = 3$
x^2
$\frac{\mathrm{d}x_3}{\mathrm{d}x_1} = 4$
$x_2 = 3$
$x_1 = 2$

Figure 3.4: A computational graph with derivatives of each input variable calculated on the nodes of the adjoint graph.

tomatic differentiation can be a part of a monadic probabilistic program which allows distributions to be composed into hierarchical models. Each distribution implements the density in terms of dual numbers and hence the mechanics of calculating and deriving the log of the un-normalised posterior for a model are abstracted away from the end user.

3.5.2 Reverse Mode Automatic Differentiation

Reverse mode AD is typically faster than forward mode AD when functions have a larger input space than output space ie. $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ where $n > m$. The log of the un-normalised posterior is a function from the parameters of dimension n to a single real number.

In order to differentiate a function using reverse mode automatic differentiation the steps are similar to forward mode differentiation, the derivatives of simple primitive functions are defined (see the implementation of numeric operations for dual numbers in listing 3.5). The function is then decomposed into its constituent primitive functions as shown in Figure 3.2. The derivative of $f(x_1, x_2) = x_1 x_2 + x_1^2$ with respect to each node of the adjoint graph is calculated at $x_1 = 2$, $x_2 = 3$ in figure 3.4.

Then proceeding in reverse through the graph to calculate the derivative of the function with respect to the input variables:

$$\frac{\partial f}{\partial x_k} = \sum_{j \in Ch(x_k)} \frac{\partial x_j}{\partial x_k} \frac{\partial f}{\partial x_j}$$

where $Ch(x_k)$ represents the set of children of node x_k. Figure 3.5 shows the reverse sweep through the computation graph required to calculate the derivative with respect to all of the arguments of the function.

Reverse mode AD is a graph traversal and can be implemented in an FP language

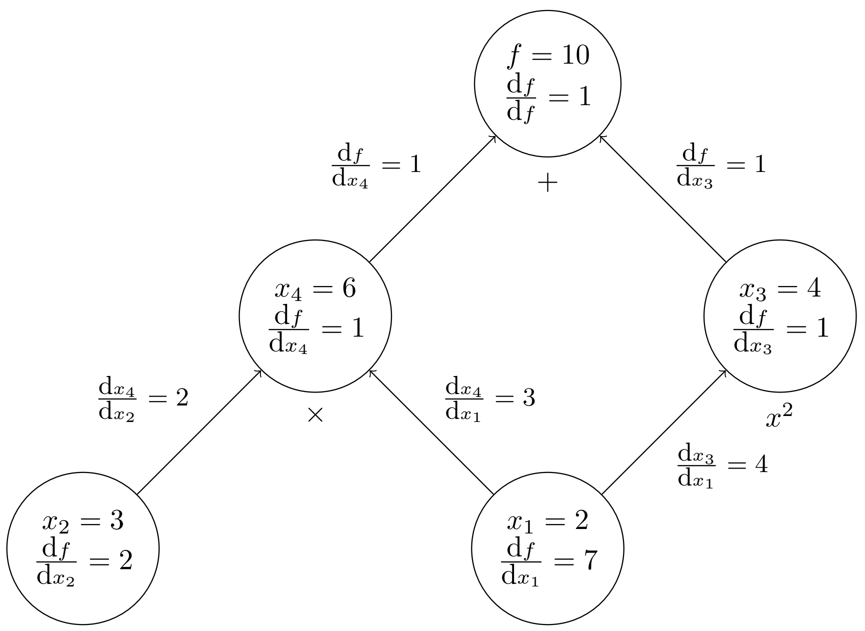


Figure 3.5: Reverse pass through the graph to calculate the deriviative of $f(x_1, x_2) = x_1x_2 + x_1^2$ with respect to all the input variables.

using pattern matching on the operators. This is the approach used by Rainier.

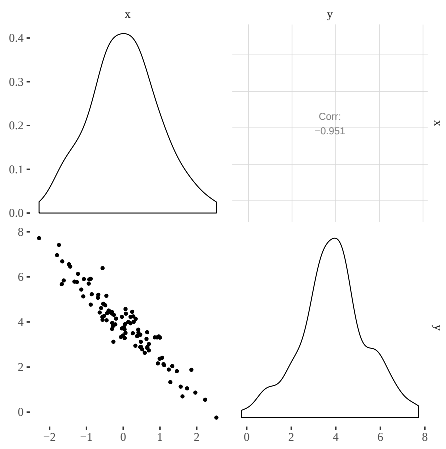
3.6 Example Probabilistic Programs

This section contains some example programs written using the Rainier DSL. The models are intentionally simple, as they are intended to represent the starting point of a probabilistic program. Areas in which assumptions of these simple models can be relaxed in order to improve model fit are highlighted. This emphasizes the flexibility of probabilistic programming.

For each of the examples the HMC algorithm with dual averaging and $L = 5$ leapfrog steps was used with 10,000 iterations for the warm-up run during which the initial leapfrog step size is determined followed by 50,000 sampling iterations with a thinning factor of 5.

3.6.1 Linear Model

Linear models are statistical models specifying a relationship between covariates, $X \in \mathbb{R}^{n\times p}$ and a univariate outcome $Y \in \mathbb{R}^n$ for each example via the coefficient β, a $(p+1)$-vector. Each row of the outcome matrix Y is denoted as y_i and is related to each row of the covariate matrix, x_i by the coefficient matrix β. The covariate matrix has a column vector of ones prepended as the first column which represents the intercept. The observations have independent normally distributed noise with equal variance:


(a)

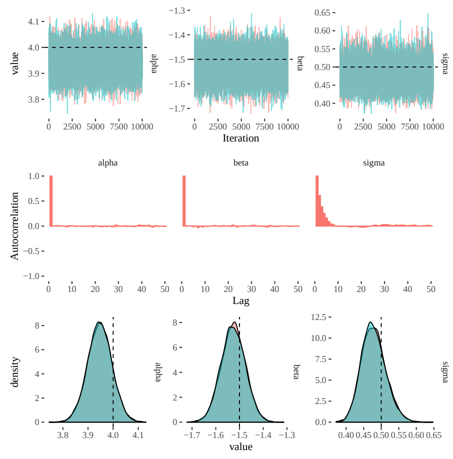

(b)

Figure 3.6: (a) Simulated data from a linear model with $\beta_0 = 4.0, \beta_1 = -1.5, \sigma = 0.5$ and covariates simulated independently from a standard Normal distribution. (b) Parameter posterior diagnostic plots for the linear model using the HMC kernel and simulated data, the actual values of the parameters are plotted using the dashed lines. The traceplots of the MCMC draws, (top). The autocorrelation function for each parameter plots, (middle). The empirical marginal parameter posterior densities, (bottom).

$$\begin{aligned}
y_i &= \beta^T x_i + \varepsilon_i, \qquad \varepsilon_i \sim \mathcal{N}(0, \sigma^2), \\
\beta &\sim \text{MVN}(\mu_\beta, \Sigma_\beta), \\
\sigma &\sim \text{Exponential}(\lambda_\sigma).
\end{aligned} \tag{3.8}$$

The model assumptions can be relaxed, but then model interpretation can become more challenging. 1,000 observations from this model are simulated with $\beta_0 = 4.0, \beta_1 = -1.5, \sigma = 0.5$ and the covariates simulated from the standard Normal distribution. The bivariate relationship of the simulated data is plotted in Figure 3.6 (a).

Listing 14 shows a probabilistic program for a linear model with covariates `x` and outcome `y`. Each parameter is added to the model using the `.param` notation and the, p covariates ($p = 1$) are related to each scalar outcome using `Predictor.from`. The linear model is expressed as a function which can be re-used in more complex hierarchical models, such as the random effects model presented in section 3.6.3.

```
def linearModel(pa: Real, pas: Real, pb: Real, pbs: Real,
                sigma: Real, data: Vector[(Double, Double)]) = for {
  alpha <- Normal(pa, pas).param
  beta <- Normal(pb, pbs).param
  _ <- Predictor[Double].from { x =>
5      Normal(alpha + beta * x, sigma)
6     }
7     .fit(data)
8 } yield Map("alpha" -> alpha, "beta" -> beta, "sigma" -> sigma)
9
10 val model = for {
11   sigma <- Exponential(3.0).param
12   params <- linearModel(0.0, 0.01, 0.0, 0.01, sigma, data)
13 } yield params
```

Listing 3.6: Simple linear model with one covariate and an intercept with a Normal observation model and fully specified prior distributions

Figure 3.6 (b) shows posterior inferences using simulated data with $\beta_0 = 4.0, \beta_1 = -1.5, \sigma = 0.5$ and the covariates simulated from a standard Normal distribution.

It is straightforward to change the observation distribution of this model, for instance to a generalised linear model with a Poisson observation distribution by changing line 5 to

```
Poisson((alpha + beta * x).exp)
```

In addition the standard deviation parameter `sigma` is no longer required when using a Poisson observation distribution. Any suitable prior distributions can be specified for the parameters; they do not have to be conditionally conjugate, as would be required for Gibbs sampling.

3.6.2 Mixture Model

In a mixture model the observed data is assumed to arise from a finite mixture of independent distributions. The mixture model considered here is a mixture of $m = 3$ Normal distributions with different mean values, $\mu_k, k = 1, \dots, m$ and a common variance.

$$
\begin{aligned}
y_i &\sim \mathcal{N}(\mu_k, \sigma^2), \\
k &\sim \mathcal{F}(\theta), \\
\theta &\sim \text{Dir}(\boldsymbol{\alpha}_\theta), \\
\mu_k &\sim \mathcal{N}(\mu_{\mu_k}, \sigma^2_{\mu_k}), \\
\sigma &\sim \text{Exponential}(\lambda_\sigma).
\end{aligned}
$$

where each index k is drawn from a discrete distribution $\mathcal{F}$ with probabilities θ. The likelihood of the mixture model can be written as:

$$p(y|\theta, \mu, \sigma) = \sum_{k=1}^{m} \theta_k \mathcal{N}(y; \mu_k, \sigma). \tag{3.9}$$

This mixture distribution has a smooth log-density and hence can be differentiated and inference can be performed using HMC. In addition the mixture distribution is already implemented in Rainier. The mixture model can be defined as

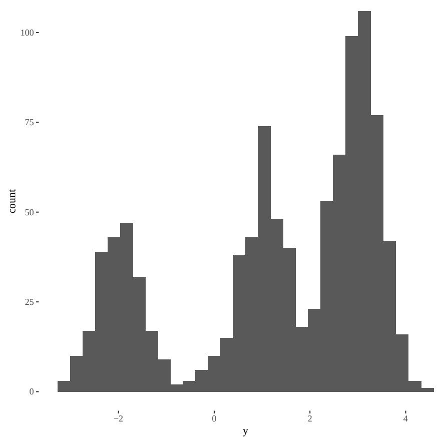
```
def normalise(alphas: Seq[Real]) = {
  val total = alphas.reduce(_ + _)
  alphas.map(a => a / total)
}

val model = for {
  unnormThetas <-
    RandomVariable.traverse(Vector.fill(3)(Gamma(3.0, 1.0).param))
  thetas = normalise(unnormThetas)
  mus <- RandomVariable.traverse(Vector.fill(3)(Normal(0, 1).param))
  sigma <- Exponential(3.0).param

  dists = mus.
    map(mu => Normal(mu, sigma)
  components: Map[Continuous, Real] = dists.zip(thetas).toMap
  _ <- Mixture(components).fit(ys)
} yield thetas ++ mus ++ sigma
```

Listing 3.7: Rainier model for a Gaussian mixture model with $m = 3$ components.

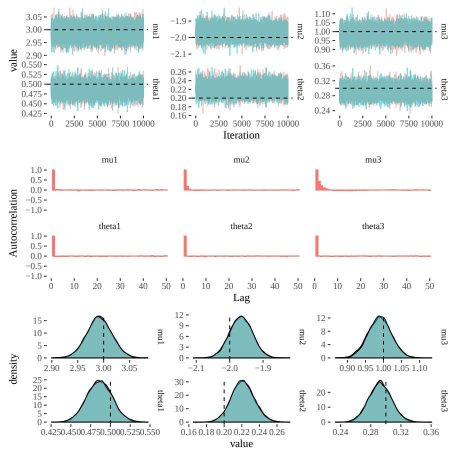
The probability of each mixing component is θ_i, $i = 1, 2, 3$. The mixing components must be greater than zero and sum-to-one, they are drawn from a Dirichlet distribution by drawing each component from a Gamma distribution with scale, $\theta = 1$ then normalising the values. Figure 3.7 (a) shows a simulation from the mixture model and (b) shows the posterior diagnostics for the means and mixing components.

3.6.3 Random Effects Model

Hierarchical models give an opportunity to emphasise the manipulation of probabilistic programs as values in the embedded DSL. This model is a random effects model with p classes each with n observations. Each class is modelled using an independent linear regression.



(a)



(b)

Figure 3.7: (a) 10,000 simulations from a three component Normal mixture model with mean values $\mu = \{-2.0, 1.0, 3.0\}$, common variance $\sigma = 0.5$ and proportions $\theta = \{0.2, 0.3, 0.5\}$ (b) Parameter posterior marginal density plots for the mixture model parameters, the parameter values used to simulate the data are vertical dashed lines. Note that the inferred components are permuted due to the truth, due to the permutation symmetry in the posterior.

$$\begin{aligned}
Y_{ij} &= \alpha_i + \beta_i x_j + \varepsilon_{ij}, & \varepsilon_{ij} &\sim \mathcal{N}(0, \sigma^2),\\
\alpha_i &\sim \mathcal{N}(\alpha_c, \sigma_a^2), & &\\
\beta_i &\sim \mathcal{N}(\beta_c, \sigma_b^2). & &
\end{aligned}$$

Each of the $i = 1, \dots, p$ classes have the same covariates $x_j, j = 1, \dots, n$ and have a linear relationship with independent coefficients, α_i and β_i. The standard deviation of the observation, σ is assumed to be the same for each class and observation. The coefficients of the covariates for all models are assumed to be Normally distributed with the same mean and variance, this induces correlation between the p linear models. $\alpha_c, \sigma_a, \beta_c, \sigma_b$ and σ are given the following prior distributions:

$$\begin{aligned}
\alpha_c &\sim \mathcal{N}(0, 10^6),\\
\sigma_a &\sim \text{Gamma}(10^{-3}, 10^3),\\
\beta_c &\sim \mathcal{N}(0, 10^6),\\
\sigma_b &\sim \text{Gamma}(10^{-3}, 10^3),\\
\sigma &\sim \text{Exponential}(3).
\end{aligned}$$

The Gamma distribution is parameterised here using shape, k and scale, θ such that the mean is $k\theta$. Each observation is represented by a class containing the index of the class, the covariate, x, and the observation, y:

```
case class Observation(
  id: Int,
  x: Double,
  y: Double
)
```

Then the prior distributions are translated to Rainier. The Normal distribution is parameterised by its mean, μ and standard deviation σ.

```
val prior = for {
  alphaC <- Normal(0.0, 1000).param
  alphaSigma <- Gamma(0.001, 1000).param
  betaC <- Normal(0.0, 1000).param
  betaSigma <- Gamma(0.001, 1000).param
  sigma <- Exponential(3).param
} yield (alphaC, alphaSigma, betaC, betaSigma, sigma)
```

A function to fit a single linear regression for the i^{th} class can be re-used from the function `linearModel` presented in section 3.6.1.

`linearModel` returns a `RandomVariable` monad which is its own probabilistic program, but this program only specifies the model for a single class. The collection of observations, `x` and `y` for a single class can be grouped by the `id` of the class using `groupBy` and a linear regression can be fit for each class using `map`. When the function `groupBy` is called on the collection of observations, it returns a `Map[Int, Iterable[Observation]]`, the key of the map is an integer representing the `id` of each class and the `Iterable` in the value of the `Map` corresponds to the observations for each class. The application of `linearModel` to each element of the `Map` returns an `Iterable[RandomVariable[(Double, Double)]` which is converted to a `Vector` before applying the function `RandomVariable.traverse`. `traverse` "reverses" the order of the effects to return a `RandomVariable[Vector[(Double, Double)]]` which is a model and can be sampled from.

```
val model: RandomVariable[Map[String, Real]] = for {
  (ac, sa, bc, sb, sigma) <- prior
  _ <- RandomVariable.traverse(
    observations.
      groupBy(_.id).
      map { case (id, obs) =>
        linearModel(ac, sa, bc, sb, sigma, obs.map(ys => (ys.x, ys.y)))
```

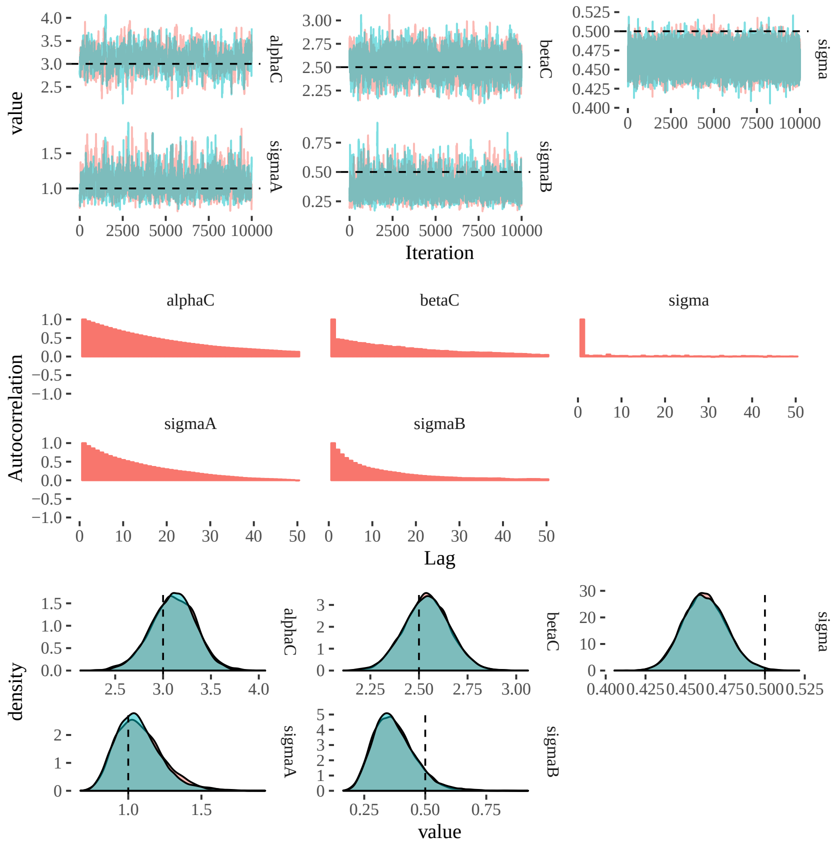


Figure 3.8: Diagnostic plots for the hierarchical random effects model with the values used to simulate the overlaid using dashed lines. Traceplots, (top). Autocorrelation function, (middle). Empirical marginal posterior densities, (bottom).

```
        }.toVector)
  } yield (ac, sa, bc, sb, sigma)
```

This implementation emphasises the compositionality of probabilistic programs embedded in a host language. The higher order functions, `groupBy` and `map` are familiar to any functional programmer and can be used to combine simple models into complex hierarchical models.

The diagnostics of the draws from the posterior distribution obtained using HMC on the random effects hierarchical example are presented in figure 3.8. The actual values used to simulate the data are plotted using dashed lines.

3.7 Sequential Monte Carlo

Sequential Monte Carlo (SMC) methods were introduced in section 1.3.5. Several functional implementations of SMC algorithms such as the SMC^2, pseudo marginal Metropolis-

Hastings (introduced in section 5.3) and resample move SMC have been proposed in Ścibior et al. [2018]. The approach by Ścibior et al. uses a sampling based probability monad and a conditioning monad. SMC and MH inference methods are developed which are called inference transformers, these are similar to monad transformers which allow (some) monads to compose. The SMC and MH kernels can then be combined to produce the aforementioned algorithms. This section presents a simple implementation of an SMC monad which reuses elements of the sampling based probability monad in section 3.2 in order to build general transition kernels by composing distributions with calls to `flatMap`.

In SMC a distribution is represented by a particle cloud with associated importance weights, $\{(x_i, w_i) : i = 1, \dots, N\}$. Hence a distribution monad with an internal particle cloud can be used to implement SMC:

```scala
trait Smc[A] {
  val cloud: Vector[(A, Double)]
  implicit val n: Int
  def flatMap[B](f: A => Smc[B]): Smc[B] = {
    val sms = cloud.flatMap {
      case (x, lw) => {
        val next = f(x)
        next.cloud.map { case (x, lw1) => (x, lw1 + lw) }
      }
    }
    SmcDist(sms).resample
  }
  def resample: Smc[A] = {
    val logWeights = cloud.map(_._2)
    val ns = logWeights.size
    val maximum = logWeights.max
    val expMax = logWeights map (x => exp(x - maximum))
    val meanWeight = expMax.sum / ns
    val likelihood = maximum + log(meanWeight)
    val indices = Multinomial(DenseVector(expMax.toArray)).
      sample(n).toVector
    val newParticles = indices map (i => cloud(i))
    SmcDist(newParticles map { case (x, lw) => (x, likelihood) })
  }
}
case class SmcDist[A](cloud: Vector[(A, Double)])
  (implicit val n: Int) extends Smc[A]
```

This defines the `Smc` trait which contains a particle cloud consisting of a vector of tuples, each tuple contains the value of the particle and the associated log-weight. Additionally, the trait contains the `flatMap` and `resample` methods. `SmcDist` behaves as a constructor for an `Smc` trait and defines the particle cloud and has as an implicit argument, the value for the size of the particle cloud `n`. `flatMap` applies a monadic function transforming the values in the particle cloud and updating the weights using by calculating the product (addition on the log-scale). Next, the particle cloud is put back into the context of the `Smc` monad using the constructor and resampled. The `resample` method uses the log-sum-exp trick as expanded upon in section 5.2.1. The `flatMap` method is used to compose empirical distributions represented by particle clouds, since the `Smc` monad has a particle cloud of dimension n used to represent the distribution, composing with another distribution with dimension m results in a particle cloud of dimension $n \times m$. This is especially troubling for filtering in time series where each observation has an associated unknown latent variable; in this case the particle cloud grows as the length of the time series increases. To avoid the program exhausting memory and allow filtering to scale to large time series the particle cloud must be thinned at each application of `flatMap`. Thinning is performed by sampling a fixed number of particles when performing the `resample` step at the end of the `flatMap`, the multinomial distribution with $n \times m$ weights is used to sample n particles.

The constructor in listing 3.8 evaluates the parameter `a` by name, this is a lazy evaluation of the argument. This means that a distribution of weighted values can be created using a suitable sampling function.

```scala
def apply[A](numParticles: Int, a: => A): Smc[A] = {
  implicit val n = numParticles
  SmcDist(Vector.fill(n)((a, 0.0)))
}
```

Listing 3.8: Constructor for the SMC class which evaluates the argument by name in order to represent a distribution of values using n points

The sampling function then creates a weighted particle cloud. Any distribution which has been implemented using the `Rand` monad introduced in section 3.2 can be reused for an `Smc` by implementing a new method in the `Rand` class:

```scala
def smc(implicit n: Int): Smc[A] =
  Smc(n, draw)
```

Now any distribution which is already defined in `Rand` can be used in the transition kernel for the latent state in the particle filter, this utilises the `apply` function defined in listing 3.8. `Rand` however does not provide an implementation for the likelihood, in order to use a distribution as a likelihood and evaluate the pdf or pmf then a new trait must be defined:

```scala
trait Likelihood[A] extends Smc[A] {
  def ll(y: A): Double
```

```
    def ll(ys: A*): Double = ys.map(ll).sum
    def fit(ys: A*): Smc[A] =
      SmcDist(cloud map { case (x, lw) => (x, lw + ll(ys: _*)) })
  }
```

This trait leaves the likelihood for a single observation of generic type unimplemented and provides implementations for sequences of observations. The `fit` method is used to calculate the likelihood of a new observation in the `Smc` monadic program.

The Normal distribution can be implemented by re-using the random generator already defined using the `Rand` monad to define the particle `cloud` and by defining the `ll` as the probability density function:

```
  def normal(mu: Double, sigma: Double)(implicit numParticles: Int):
   Likelihood[Double] = new Likelihood[Double] {
    val cloud: Vector[(Double, Double)] =
      Rand.normal(mu, sigma).smc(numParticles).cloud
    implicit val n: Int = numParticles
    def ll(y: Double) = Gaussian(mu, sigma).logPdf(y)
  }
```

In order to determine the posterior distribution of a latent-variable at time t given the particle cloud at time $t-1$ then `flatMap` is applied to the distribution at time $t-1$ using a state transition function represented by a distribution or composition of distributions. As an example consider an AR(1) state space model introduced in section 1.3.5

```
  def kernel(xs: Smc[Double], y: Double): Smc[Double] =
    for {
      a <- xs
      x1 <- normal(p.mu + p.phi * (a.head - p.mu), p.sigma)
      _ <- normal(x1, 0.5).fit(y)
    } yield (x1 :: xs)
```

The function `kernel` is a single step of the bootstrap particle filter with resampling performed at every step. This single step of the bootstrap particle filter can be applied to a time series using the function `foldLeft`, returning an `Smc` monad containing a list of posterior distributions representing the latent-state:

```
  def pFilter(
    p: Parameters,
    ys: Vector[Double]): Smc[List[Double]] = {
    val sd = p.sigma / sqrt((1 - p.phi * p.phi))
    val init: Smc[List[Double]] = for {
      x0 <- Smc.normal(p.mu, sd)
    } yield List(x0)
```

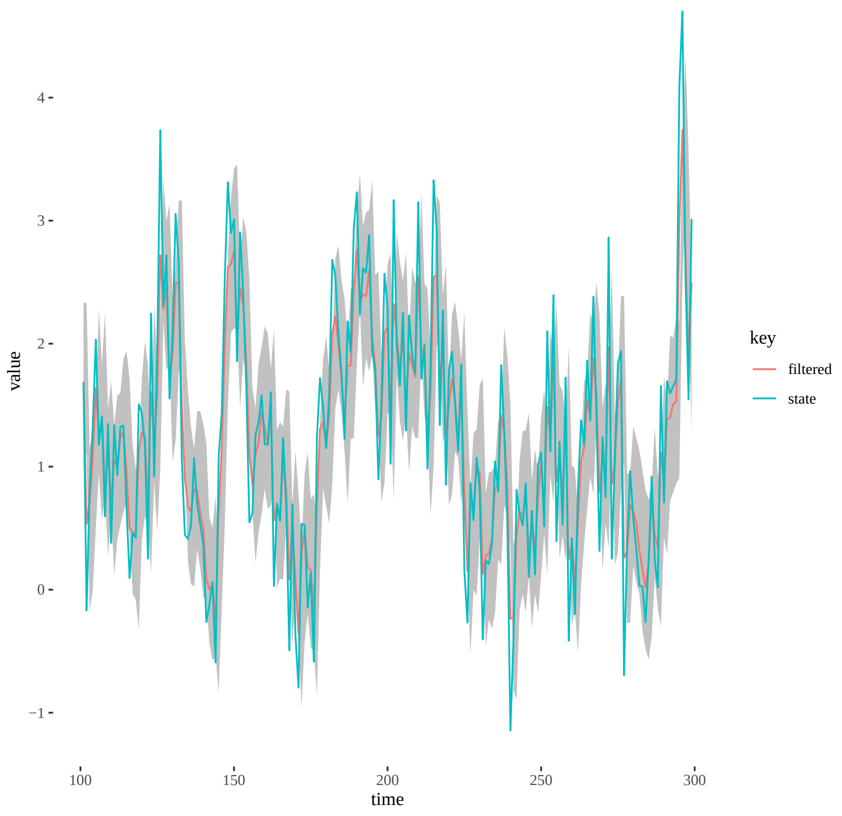


Figure 3.9: Simulated latent-state of the AR(1) model with $\mu = 1, \phi = 0.8, \sigma = 0.3$ and the mean of the filtered state with 90% credible intervals.

```
    ys.foldLeft(init)(kernel)
  }
```

Figure 3.9 shows the output from applying the particle filtering algorithm to a simulated AR(1) dataset. The mean of the filtering distribution is calculated along with a 90% credible interval for the state.

The SMC monad implemented in this chapter can be used to develop arbitrary transition kernels by transforming the distributions from section 3.2 into weighted particle clouds. The particle clouds can then be combined using `flatMap` to make arbitrarily complex latent-state transition distributions. Conditioning is performed by calculating the weights using the `fit` method defined in the `Likelihood` trait, representing conditional likelihood of an observation in the state space model . To prevent the particle cloud from growing large when applying a `flatMap`, implicit thinning is used by sampling a fixed number of particles during the multinomial resampling step. This implementation uses forward simulation from the transition kernel to build a particle filter, this means the implementation is very general and can support any transition kernel which can be forward simulated.

3.8 Summary

Probabilistic programming aims to unify Bayesian inference with general programming languages in order to simplify model exploration and inference. The allows practitioners to develop novel statistical models with minimal consideration for the underlying inference algorithms.

This chapter presents a high level overview of the Rainier probabilistic programming language embedded in Scala. This understanding lead to the contribution of eHMC to the inference methods provided to practitioners using the Rainier probabilistic programming language.

In order to maximise the utility of these models, the programming languages should be embedded in existing general purpose programming languages. Functional programming languages have powerful, theoretically sound abstractions from category theory, such as monads, which can be used to develop embedded DSLs for probabilistic programming.

The monadic DSLs introduced in this chapter provide a powerful modelling language embedded in the Scala language. The for-comprehension (or similar do-notation in Haskell) is familiar to Scala programmers and can be used to manipulate distribution objects. The monadic syntax provided by the for-comprehension is reminiscent of the syntax in BUGS, Jags and Stan, allows experts in these languages to easily transition to using the embedded DSL in Scala. In addition, the embedded nature of the language allows practitioners to utilise the features which come with Scala such as functions which can be used to develop hierarchical models in a modular, composable fashion, as exemplified in section 3.6.3.

Section 3.7 introduced a new probability monad using sampling importance resampling to perform parameter inference. The new language was demonstrated using a partially observed Markov process with a Poisson observation distribution.

Chapter 4

Dynamic Linear Models

Dynamic Linear Models (DLM) are a flexible and well studied class of time series models suitable for modelling non-stationary time series with both univariate and multivariate observations. Dynamic linear models are otherwise known as linear dynamical systems or state space models, the latter name highlighting the latent unobserved state but alluding to a larger class of models with (possibly) non-linear dynamics and non-Gaussian latent-state and observation distribution. DLMs are linear, Gaussian systems which means the latent-state is continuous and Gaussian and the observations consist of a linear transformation of the latent-state plus Normal noise representing measurement error. Special properties of the Normal distribution allow the Kalman filter [Kalman, 1960] to be used to determine the distribution of the latent-state at each time conditional on all the information up to that time, called the filtering distribution.

Dynamic linear models are flexible, have online forecasting available in a closed form using the Kalman Filter and are therefore well suited to the domain of interest, dynamic sensor networks such as the Urban Observatory. Specification of different models for various environmental phenomenon is straightforward, by specifying observation and system matrices as in section 4.1. Additionally, simple models can be composed to model complex univariate and multivariate systems as described in section 4.1.1 for univariate models and section 4.1.2 for multivariate models. Inference is scalable for large streams of data since filtering can be performed online using the Kalman Filter described in section 4.2, the filter can be applied to a potentially unbounded stream of incoming data. Online filtering allows forecasts to be updated given the latest observations without re-running offline inference algorithms (such as MCMC). Simultaneous online state and parameter estimation is also possible using particle filtering methods and marginalisation in the case of the Rao-Blackwell particle filter introduced in section 4.6.

This chapter will introduce the DLM and the Kalman filter for online state-inference in section 4.2 and offline parameter inference using forward filtering backward sampling and

MCMC in section 4.4. Section 4.5 describes practical, numerically stable, implementations of the Kalman filter using Joseph form [Bucy and Joseph, 2005] updating and singular value decomposition of the latent-state covariance matrix. Section 4.6 introduces online simultaneous state and parameter estimation using particle filtering. An extension of the DLM such that the latent state is represented in continuous time using a Gaussian diffusion process is presented in section 4.7. The continuous time latent-state enables the DLM to be used to model irregularly observed data in a natural way. See West and Harrison [1997], Petris et al. [2009] for detailed treatments of Dynamic Linear models.

4.1 Introduction to the DLM

Dynamic linear models consist of a latent state, $\theta_t \in \mathbb{R}^n$, $t = 0, \dots, N$, the state evolution kernel is a linear Gaussian Markov process, $\theta_t \sim \mathcal{N}(\theta_t; G_t\theta_{t-1}, W_t)$. Observations, $Y_t \in \mathbb{R}^p$ are observed at discrete time points, $t = 1, \dots, N$. The observations are conditionally independent given the latent-state at time t and are corrupted with Gaussian noise. A general DLM is written as

$$\begin{aligned} Y_t &= F_t^T\theta_t + v_t, & v_t &\sim \text{MVN}_p(0, V_t), \\ \theta_t &= G_t\theta_{t-1} + w_t, & w_t &\sim \text{MVN}_n(0, W_t), \\ \theta_0 &\sim \text{MVN}_n(m_0, C_0). \end{aligned} \tag{4.1}$$

Here, F_t is an $n \times p$ known observation matrix, G_t is an $n \times n$ known system evolution matrix. V_t is the $p \times p$ observation covariance matrix representing the measurement noise and W_t is the $n \times n$ system covariance matrix. The observation matrix and the system matrix (F_t and G_t) are typically chosen by the statistician or scientist analysing the data whereas the system and observation covariance matrices are learned from data using appropriate inference schemes. This framework can be used to analyse both stationary and non-stationary time series. Figure 4.1 shows a directed a-cyclical graph (DAG) of the DLM which shows the dependence structure of the model.

The first order polynomial DLM is a simple example of a useful dynamic linear model. The latent state is a random walk with Gaussian innovations. This is a Markov process with transition kernel $p(x_t|x_{1:t-1}, \sigma) = \mathcal{N}(x_t|x_{t-1}, \sigma^2)$.

The first order polynomial DLM can be written as follows with static observation and system matrices $F_t = 1$ and $G_t = 1$

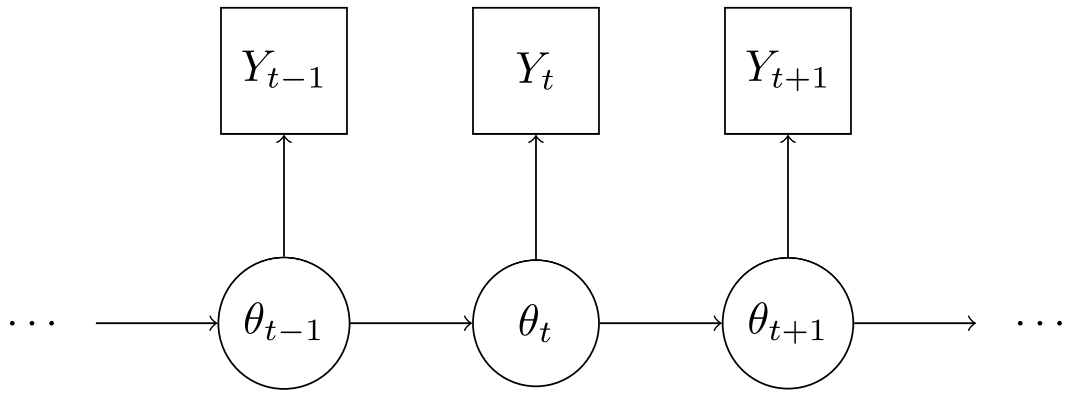


Figure 4.1: Directed a-cyclical graph of a Dynamic linear model. The bottom row shows the evolution of the latent-state, the horizontal arrows represent state transitions. The top row shows the observations, which are conditionally independent given the latent-state at each time point.

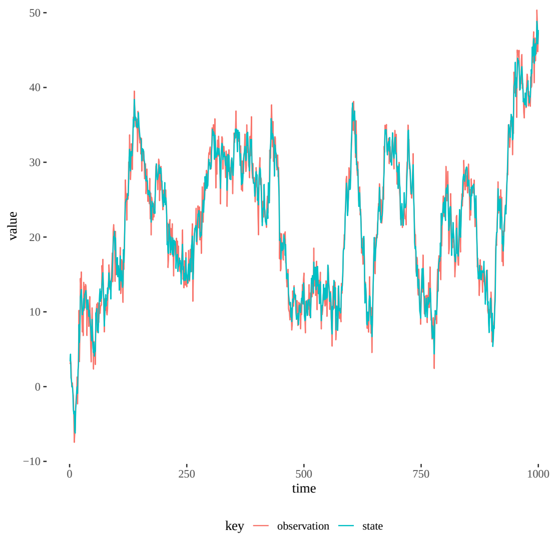


Figure 4.2: Simulated observations and latent-state from a first order DLM with $m_0 = 0, C_0 = 1, W = 3, V = 2$.

$$\begin{aligned} Y_t &= \theta_t + v_t, & v_t &\sim \mathcal{N}(0, V), \\ \theta_t &= \theta_{t-1} + w_t, & w_t &\sim \mathcal{N}(0, W), \\ \theta_0 &\sim \mathcal{N}(m_0, C_0). \end{aligned} \tag{4.2}$$

Figure 4.2 shows 300 simulated values from the first order DLM given in Equation 4.2 with initial state $\theta_0 \sim \mathcal{N}(0, 1)$. The first order DLM has time homogeneous V and W.

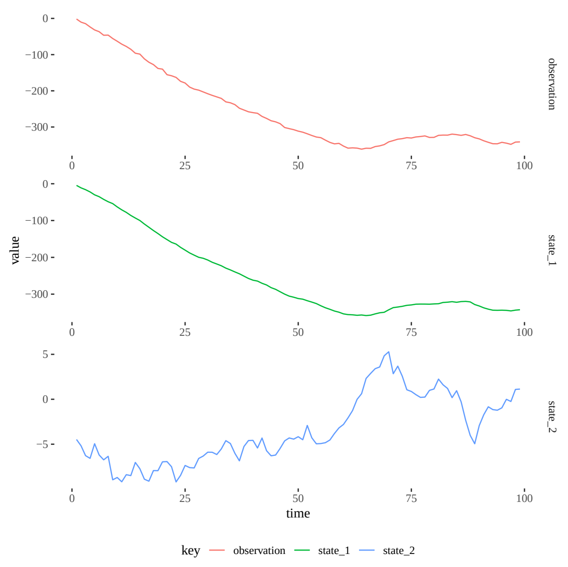


Figure 4.3: Simulated values from a second order DLM showing a strongly negative trend with parameter values $V = 3, W = \text{diag}(2.0, 1.0), \mathbf{m}_0 = (0.0 \quad 0.0)^T, C_0 = \text{diag}(100.0, 100.0)$.

The first order model does not display much structure and is useful as a building block in larger models. The second order DLM can be used to model linear trends. In the second order model the latent-state is two dimensional; as such, the system noise matrix W is a 2×2 matrix. The observation matrix and system evolution matrices are given by

$$F = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad G = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}. \tag{4.3}$$

Figure 4.3 shows a simulation from the second order DLM with a strongly negative trend dominated by state 1.

Dynamic linear models are closely related to linear models. Linear models express a static relationship between independent observations and associated covariates via a linear Gaussian model. The regression DLM can be considered as an extension of a linear model to time varying coefficients. If it is thought that the time series of covariates $X_{1:N}$ is related to the time series $Y_{1:N}$ in a time inhomogeneous way, then this relationship can be expressed as a regression DLM

$$\begin{aligned}
Y_t &= \beta_t^T x_t + v_t, & v_t &\sim \text{MVN}_p(0, V_t), \\
\beta_t &= G_t \beta_{t-1} + w_t, & w_t &\sim \text{MVN}_n(0, W_t), \\
\beta_0 &\sim \text{MVN}_n(m_0, C_0).
\end{aligned} \tag{4.4}$$

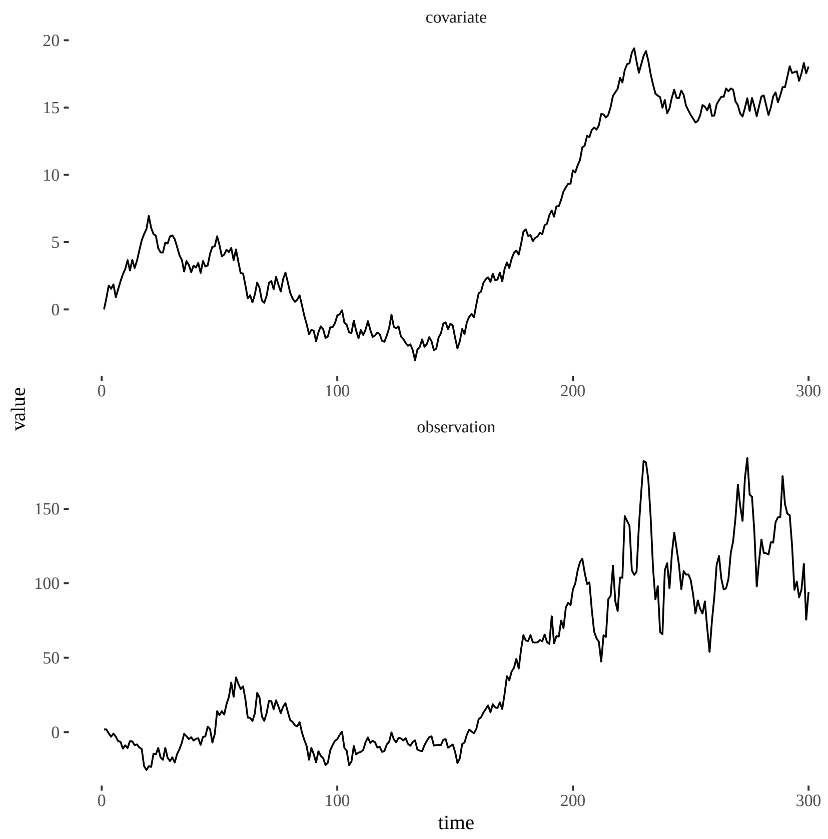


Figure 4.4: Simulated values from a Regression time series, where the independent time series $X_{1:N}$ is a random walk with transition kernel $x_t|x_{t-1} = x_{t-1}+u_t, u_t \sim \mathcal{U}(-1,1)$. Covariates following a random walk with uniform innovations, (top). Simulated observations, (bottom).

In the regression DLM, the coefficients, $\beta_{0:N}$ of the dependent time series $X_{1:N}$ evolve in time according to the system evolution matrix, G_t and the identified system covariance matrix, W_t. Figure 4.4 shows a simulation from a regression time series.

Seasonality is an important pattern in time series modelling, such as in environmental monitoring of temperature, humidity or pollutants. It is possible to model seasonality using a time dependent observation matrix. The observation matrix consists of trigonometric functions which create a Fourier series. Equation 4.5 shows the observation matrix, P represents the period of the seasonality, t is the current time of the observation and h is the harmonic. In practice multiple harmonics are used to capture seasonality at varying time scales. In this case, the system evolution matrix is a $2h \times 2h$ identity matrix. We have that

$$F_t = \begin{pmatrix} \sin(\frac{2\pi t}{P}) \\ \cos(\frac{2\pi t}{P}) \\ \vdots \\ \sin(\frac{2\pi h t}{P}) \\ \cos(\frac{2\pi h t}{P}) \end{pmatrix} \tag{4.5}$$

An alternative, and more common approach to modelling seasonality is to encode the transformation into the system matrix. This has the added benefit that the model matrices, F_t and G_t, are independent of time for regular observations. The G matrix is a block diagonal matrix consisting of rotation matrices

$$G = \begin{pmatrix} R(\omega,1) & 0 & \dots & 0 \\ \vdots & R(\omega,2) & \dots & 0 \\ \vdots & 0 & \ddots & 0 \\ 0 & \dots & 0 & R(\omega,h) \end{pmatrix} \tag{4.6}$$

The rotation matrices are parameterised by the frequency $\omega = \frac{2\pi}{P}$ and the harmonic h where P is the period of the seasonality. The rotation matrices are:

$$R(\omega,h) = \begin{pmatrix} \cos(h\omega) & -\sin(h\omega) \\ \sin(h\omega) & \cos(h\omega) \end{pmatrix}. \tag{4.7}$$

The corresponding observation matrix is $2h \times 1$ with alternating zeros and ones:

$$F = \begin{pmatrix} 1 & 0 & 1 & \dots & 1 & 0 \end{pmatrix}^T. \tag{4.8}$$

Many environmental time series display seasonal patterns, such as temperature which is higher at noon and during summertime and PM10 (particulate matter measuring less than 10μm in diameter) which is typically higher during rush hour traffic and lower at weekends.

4.1.1 Model Composition

In order to model complex univariate time series, the simple models discussed in the previous section can be composed by combining the system and observation matrices, G_t and F_t respectively and the system and observation covariance matrices, W_t and V_t. The observation matrices are to be horizontally concatenated to produce a new observation matrix, whereas the system evolution and system noise covariance matrices are block concatenated. The composition of two models with observation and system matrices F_1,

G_1 and F_2, G_2 is:

$$F = \begin{bmatrix} F_1 \\ F_2 \end{bmatrix}, \quad G = \begin{bmatrix} G_1 & \mathbf{0} \\ \mathbf{0} & G_2 \end{bmatrix}. \tag{4.9}$$

The system covariance matrix is also block concatenated, whereas the observation covariance matrices are not combined and the leftmost observation matrix in the composition is selected:

$$V = \begin{bmatrix} V_1 \end{bmatrix}, \quad W = \begin{bmatrix} W_1 & \mathbf{0} \\ \mathbf{0} & W_2 \end{bmatrix}. \tag{4.10}$$

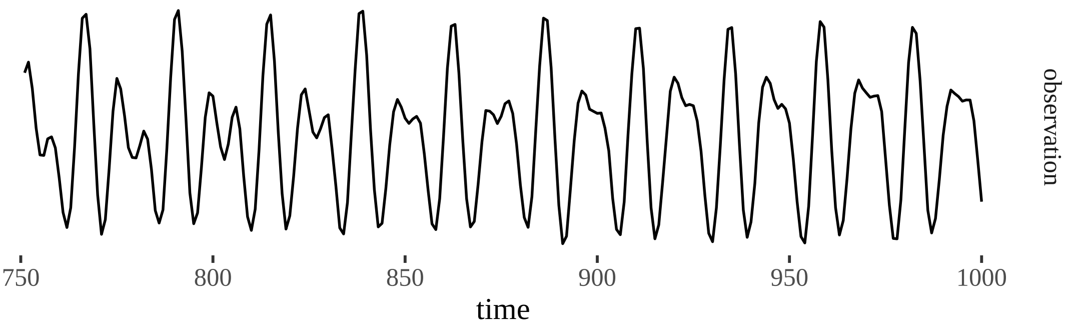
As an example, consider a seasonal model with a linear trend, the observation and system matrices for the first order linear trend are $F = 1$ and $G = 1$. The observation and system matrices for the seasonal model are as in equations 4.8 and 4.6 respectively. The composed model matrices are:

$$F = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ \dots \\ 1 \\ 0 \end{pmatrix}, \quad G = \begin{pmatrix} 1 & 0 & & \dots & 0 \\ 0 & R(\omega, 1) & 0 & \dots & 0 \\ \vdots & 0 & R(\omega, 2) & \dots & 0 \\ \vdots & 0 & & \ddots & 0 \\ 0 & \dots & & 0 & R(\omega, h) \end{pmatrix}. \tag{4.11}$$

The observation and system noise covariance matrices are 1-dimensional and $2h + 1$-dimensional square matrices respectively. A simulation from a seasonal model with a linear trend with $h = 3$ harmonics is plotted in figure 4.5. DLM composition for a univariate model forms a semigroup with an associative binary operation defined as above. The binary operation to combine DLMs is closed which means that the composition of two DLMs is also a DLM and can be composed to form arbitrarily complex models.

4.1.2 Modelling Multivariate Time Series

Some time series are very closely related, such as exchange rates from several different European countries or environmental processes such as temperature in neighbouring counties. These examples of closely related time series can be modelled using a joint DLM. DLMs can be combined to model multivariate time series where each observation Y_t is a p-dimensional vector.



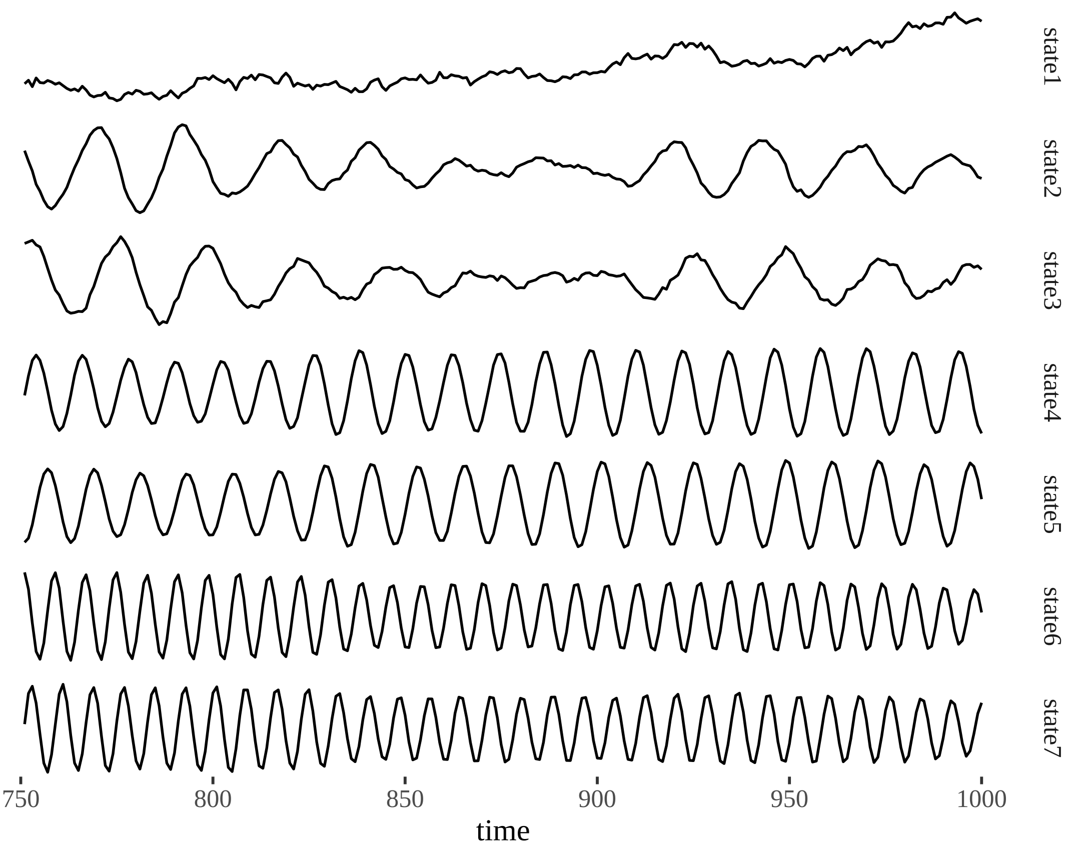


Figure 4.5: Simulated observations, (top) and states, (bottom) from a seasonal DLM composed with linear trend DLM. The parameter values were chosen to be $V = 1.0$, $W = \text{diag}(0.01, 0.2, 0.4, 0.5, 0.2, 0.1, 0.4)$.

Any two DLMs can be combined into a multivariate DLM, by block concatenating the observation matrices, system matrices and error matrices. If the model matrices are F_1, F_2 and G_1, G_2 then the resulting model matrices of the multivariate DLM are:

$$F = \begin{bmatrix} F_1 & \mathbf{0} \\ \mathbf{0} & F_2 \end{bmatrix}, \quad G = \begin{bmatrix} G_1 & \mathbf{0} \\ \mathbf{0} & G_2 \end{bmatrix}. \tag{4.12}$$

In order to couple these DLMs, the latent-state can be coupled by introducing parameters in the off diagonal of the system evolution matrix, G. Additionally, the system noise covariance matrix can be considered to be a full rank non-diagonal matrix. This can prove problematic for models with a high dimensional latent-state due to the large number of parameters to estimate. A factor model or a parameterised covariance function can be used to reduce the number of free parameters in a large covariance matrix. Factor models are considered in chapter 7.

4.2 Kalman Filter

One area of interest when modelling time series data using DLMs is to determine the filtering distribution $p(\theta_{1:N}|Y_{1:N}, \psi)$ where $\psi = \{V, W, m_0, C_0\}$ represents the static parameters. The Kalman filter is a recursive, online algorithm which can be used to estimate the latent-state of a DLM.

The Kalman filter consists of repeated applications of Bayes theorem where the posterior of the latent-state from the previous step is used as the prior in the subsequent step. An inductive argument, and special properties of the Normal distribution can be used to derive the Kalman filter recursions. Suppose the posterior latent-state at time t is given by $p(\theta_t|y_{1:t}) = \mathcal{N}(m_t, C_t)$ then the mean of the prior state at time $t + 1$ is given by

$$\begin{aligned} a_{t+1} &= \mathbb{E}(\theta_{t+1}|y_{1:t}) \\ &= \mathbb{E}(\mathbb{E}(\theta_{t+1}|\theta_t, y_{1:t})|y_{1:t}), \\ &= \mathbb{E}(G_{t+1}\theta_t|y_{1:t}) = G_{t+1}m_t \end{aligned}$$

This is an application of the law of total expectation. The variance can be calculated similarly using the law of total variance:

$$\begin{aligned}
R_{t+1} &= \mathrm{Var}(\theta_{t+1}|y_{1:t}) \\
&= \mathbb{E}(\mathrm{Var}(\theta_{t+1}|y_{1:t}, \theta_t)|y_{1:t}) + \mathrm{Var}(\mathbb{E}(\theta_{t+1}|y_{1:t}, \theta_t)|y_{1:t}), \\
&= \mathbb{E}(W_{t+1}|y_{1:t}) + \mathrm{Var}(G_{t+1}\theta_t|y_{1:t}), \\
&= W_{t+1} + G_{t+1}C_tG_{t+1}^T
\end{aligned}$$

The mean of the one-step forecast for the observation at time $t+1$ is given by:

$$\begin{aligned}
f_{t+1} &= \mathbb{E}(Y_{t+1}|y_{1:t}), \\
&= \mathbb{E}(\mathbb{E}(Y_{t+1}|\theta_{t+1}, y_{1:t})|y_{1:t}), \\
&= \mathbb{E}(F_{t+1}\theta_{t+1}|y_{1:t}), \\
&= F_{t+1}a_{t+1}.
\end{aligned}$$

The variance of the one-step forecast is given by:

$$\begin{aligned}
Q_{t+1} &= \mathrm{Var}(Y_{t+1}|y_{1:t}), \\
&= \mathbb{E}(\mathrm{Var}(Y_{t+1}|\theta_{t+1}, y_{1:t})|y_{1:t}) + \mathrm{Var}(\mathbb{E}(Y_{t+1}|\theta_{t+1}, y_{1:t})|y_{1:t}), \\
&= \mathbb{E}(V_{t+1}|y_{1:t}) + \mathrm{Var}(F_{t+1}\theta_{t+1}|y_{1:T}), \\
&= V_{t+1} + F_{t+1}R_{t+1}F_{t+1}^T.
\end{aligned}$$

Now suppose y_{t+1} is observed and the latent-state is updated to the posterior distribution. $p(\theta_{t+1}|y_{1:t}) = \mathcal{N}(a_{t+1}, R_{t+1})$ is the prior distribution of the latent-state at time $t+1$, the conditional likelihood of the observation is $p(y_{t+1}|\theta_{t+1}) = \mathcal{N}(F_{t+1}\theta_{t+1}, V_{t+1})$ then the posterior $p(\theta_{t+1}|y_{1:t+1})$ can be found using Bayes theorem:

$$\begin{aligned}
p(\theta_{t+1}|y_{1:t+1}) &\propto p(y_{t+1}|\theta_{t+1})p(\theta_{t+1}|y_{1:t}) \\
&= \mathcal{N}(y_{t+1}; F_{t+1}\theta_{t+1}, V_{t+1})\mathcal{N}(\theta_{t+1}; a_{t+1}, R_{t+1}) \\
&\propto \exp\left\{\frac{1}{2}(y_{t+1} - F_{t+1}\theta_{t+1})^T V_{t+1}^{-1}(y_{t+1} - F_{t+1}\theta_{t+1})\right. \\
&\left. + \frac{1}{2}(\theta_{t+1} - a_{t+1})^T R_{t+1}^{-1}(\theta_{t+1} - a_{t+1})\right\} \\
&\propto \exp\left\{\frac{1}{2}\theta_{t+1}^T(R_{t+1}^{-1} + F_{t+1}^T V_{t+1}^{-1}F_{t+1})\theta_{t+1} - 2\theta_{t+1}^T(F_{t+1}^T V_{t+1}^{-1}y_{t+1} + a_{t+1}R_{t+1}^{-1})\right\} \\
&= \mathcal{N}(m_{t+1}, C_{t+1}).
\end{aligned}$$

Then the mean and covariance can be written as

$$m_{t+1} = C_{t+1}(F_{t+1}^T V_{t+1}^{-1} y_{t+1} + a_{t+1}R_{t+1}^{-1}), \tag{4.13}$$

$$C_{t+1} = (R_{t+1}^{-1} + F_{t+1}^T V_{t+1}^{-1}F_{t+1})^{-1}. \tag{4.14}$$

This can be expressed in a sequential form using the Woodbury matrix identity [Woodbury, 1950], which states:

$$(A + UCV)^{-1} = A^{-1} - A^{-1}U\left(C^{-1} + VA^{-1}U\right)^{-1}VA^{-1} \tag{4.15}$$

Using this identity the covariance can be re-written as

$$\begin{aligned}
C_{t+1} &= R_{t+1} - R_{t+1}F_{t+1}^T(V_{t+1} + F_{t+1}R_{t+1}F_{t+1}^T)^{-1}F_{t+1}R_{t+1}, \\
&= R_{t+1} - R_{t+1}F_{t+1}^T Q_{t+1}^{-1}F_{t+1}R_{t+1}.
\end{aligned}$$

Then the mean can be re-written as

$$m_{t+1} = m_t + R_{t+1}F_{t+1}^T Q_{t+1}^{-1}(y_{t+1} - F_{t+1}a_{t+1}),$$

using applications of the same identity. The Kalman gain is defined as $K_t = R_t F_t^T Q_t^{-1}$. The Kalman gain dictates the update to the state posterior mean and covariance. The system and observation covariance matrices can be thought of as the ratio of signal to noise. If the signal is strong (V is small relative to W) then the observation is considered to be reliable, however if the opposite is true then the latent-state is considered to be more reliable. If the observation noise covariance matrix is small relative to the system noise covariance matrix, then the Kalman gain is larger and hence a larger correction is applied to the state mean.

The Kalman filter for a general DLM is summarised in Algorithm 6. Since the latent-state is Gaussian, it is entirely determined by its mean and variance, hence the filter involves updating the mean and variance at each time point.

Algorithm 6: The Kalman filter to determine the filtering distribution of a DLM

```
   Result: Return $p(\theta_{0:N}|y_{1:N}, \psi) = \mathcal{N}(m_{0:N}, C_{0:N})$
1  Given observations $Y_{1:N}$ and static parameters $\psi = \{V, W, m_0, C_0\}$;
2  for $t$ in 0 to $N-1$ do
3      Calculate prior state $\tilde{\theta}_{t+1} \sim \mathcal{N}(a_{t+1}, R_{t+1})$;
4      $a_{t+1} = G_{t+1} m_t$;
5      $R_{t+1} = G_{t+1} C_t G_{t+1}^T + W_{t+1}$;
6      Calculate one step prediction: $\hat{y}_{t+1} \sim \mathcal{N}(f_{t+1}, Q_{t+1})$;
7      $f_{t+1} = F_{t+1} a_{t+1}$;
8      $Q_{t+1} = F_{t+1} R_{t+1} F_{t+1}^T + V_{t+1}$;
9      Observe $y_{t+1}$ then update the state $\theta_{t+1} \sim \mathcal{N}(m_{t+1}, C_{t+1})$;
10     $m_{t+1} = a_{t+1} + K_{t+1} e_{t+1}$;
11     $C_{t+1} = R_{t+1} - K_{t+1} F_{t+1} R_{t+1}$;
12     where $K_{t+1} = R_{t+1} F_{t+1}^T Q_{t+1}^{-1}$, $e_{t+1} = y_{t+1} - f_{t+1}$.
13 end
```

In the case of a missing observation, the state update is not performed and the state is simply advanced to the time of the next non-missing observation. This results in the uncertainty in the state estimate growing until the time of the next observation.

The filter can be modified in the case of partially missing data. Consider a p-dimensional observation Y_t where $m < p$ observations are missing and the indices of the missing variables are known and given by $k_i, i = 1, \dots, m$. Then the observation matrix, F_t and observation Y_t are modified by omitting rows corresponding to the indices k_i and the observation noise matrix V_t is modified to remove the rows and columns corresponding to the indices k_i. Then the dimension of the fitted mean $F_t x_t$ is equal to the dimension of $Y_t^{(-k_i)}$. Then the state-update step can be performed in the presence of partially missing data.

The Kalman filter is straightforward to implement in Scala using the functional programming principles introduced in chapter 2. The implementation follows the same pattern to that of the particle filter presented in section 2.1.15. First a single step of the algorithm is expressed as a pure function from the state at time t to the updated state `(Gaussian, A) => Gaussian` where `A` represents an observation and `Gaussian` represents the latent state defined by a Normal distribution. Next a reduction function such as `foldLeft` (or `scanLeft` if the intermediate filtering distributions are of interest) is used to run the filter over the entire collection.

4.3 Backward Smoothing

Backward smoothing produces posterior distributions of the latent-state with less uncertainty than the Kalman filter, which is useful for interpolating missing measurements. Unlike the Kalman filter, the backward smoother cannot be performed online since it requires knowledge of the entire time series.

Suppose a time series is observed at times $t = 1, \dots, N$ then the Kalman filter is performed to learn the filtering distribution $p(\theta_{1:N}|y_{1:N}, \psi)$ which is entirely determined by the mean and covariance of the latent-state at times $t = 0, \dots, N$. In order to perform backward smoothing the algorithm starts with the posterior of the latent-state at time N, $\mathcal{N}(m_N, C_N)$. The algorithm then proceeds in reverse by considering the backwards transition probability

$$\begin{aligned} p(\theta_t|y_{1:N}, \theta_{t+1}) &= p(\theta_t|y_{1:t}, \theta_{t+1}), \\ &= \frac{p(\theta_t|y_{1:t})p(\theta_{t+1}|\theta_t, y_{1:t})}{p(\theta_{t+1}|y_{1:t})}, \\ &= \frac{p(\theta_t|y_{1:t})p(\theta_{t+1}|\theta_t)}{p(\theta_{t+1}|y_{1:t})}. \end{aligned}$$

This follows since $Y_{t+1:N}$ is conditionally independent of θ_t given θ_{t+1}, and θ_{t+1} is conditionally independent of $Y_{1:N}$ given the previous value of the state θ_t. Then:

$$\begin{aligned} p(\theta_t|y_{1:N}) &= \int p(\theta_t, \theta_{t+1}|y_{1:N})\mathrm{d}\theta_{t+1}, \\ &= \int p(\theta_t|\theta_{t+1}, y_{1:N})p(\theta_{t+1}|y_{1:N})\mathrm{d}\theta_{t+1}, \\ &= \int \frac{p(\theta_t|y_{1:t})p(\theta_{t+1}|\theta_t)}{p(\theta_{t+1}|y_{1:t})} p(\theta_{t+1}|y_{1:N})\mathrm{d}\theta_{t+1}, \\ &= p(\theta_t|y_{1:t}) \int \frac{p(\theta_{t+1}|\theta_t)p(\theta_{t+1}|y_{1:N})}{p(\theta_{t+1}|y_{1:t})}\mathrm{d}\theta_{t+1}. \end{aligned}$$

The backward smoother for a DLM as specified in equation (4.1) starts with the final filtered state $p(\theta_N|y_{1:N}) = \mathcal{N}(m_N, C_N)$ and proceeds recursively for $t = N-1, \dots, 1$:

$$\begin{aligned} s_t &= m_t + C_t G_{t+1}^T R_{t+1}^{-1}(s_{t+1} - a_{t+1}), \\ S_t &= C_t - C_t G_{t+1}^T R_{t+1}^{-1}(R_{t+1} - S_{t+1})R_{t+1}^{-1} G_{t+1} C_t. \end{aligned}$$

For a DLM the distributions of the backward smoother are Gaussian and available in closed form. Starting with the mean:

$$\begin{aligned} s_t &= \mathbb{E}(\theta_t|y_{1:N}), \\ &= \mathbb{E}(\mathbb{E}(\theta_t|y_{1:N}, \theta_{t+1})|y_{1:N}), \end{aligned}$$

θ_t is conditionally independent of $y_{t+1:N}$ given θ_{t+1} hence

$$\begin{aligned} p(\theta_t|y_{1:t}, \theta_{t+1}) &\propto p(\theta_t|y_{1:t})p(\theta_{t+1}|\theta_t, y_{1:t}) \\ &= p(\theta_t|y_{1:t})p(\theta_{t+1}|\theta_t) \\ &= \mathcal{N}(\theta_t; m_t, C_t)\mathcal{N}(\theta_{t+1}; G_{t+1}\theta_t, W_{t+1}) \\ &= (2\pi C_t)^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(\theta_t - m_t)^T C_t^{-1}(\theta_t - m_t)\right) \\ &\times (2\pi W_{t+1})^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(\theta_{t+1} - G_{t+1}\theta_t)^T W_{t+1}^{-1}(\theta_{t+1} - G_{t+1}\theta_t)\right) \\ &\propto \exp\left(-\frac{1}{2}\left[\theta_t^T \left(C_t^{-1} + G_{t+1}^T W_{t+1}^{-1} G_{t+1}\right)\theta_t - 2\theta_t\left(m_t C_t^{-1} + G_{t+1} W_{t+1}^{-1}\theta_{t+1}\right)\right]\right) \\ &= \mathcal{N}\left((C_t^{-1} + G_{t+1}^T W_{t+1}^{-1} G_{t+1})^{-1}(m_t C_t^{-1} + G_{t+1} W_{t+1}^{-1}\theta_{t+1}), (C_t^{-1} + G_{t+1}^T W_{t+1}^{-1} G_{t+1})^{-1}\right). \end{aligned}$$

Then from the Woodbury Matrix Identity [Woodbury, 1950], which states:

$$(A + UCV)^{-1} = A^{-1} - A^{-1}U\left(C^{-1} + VA^{-1}U\right)^{-1} VA^{-1}.$$

Then the mean and covariance can be expressed as

$$\begin{aligned} \mathbb{E}(\theta_t|\theta_{t+1}, y_{1:t}) &= m_t + C_t G_{t+1}^T (G_{t+1} C_t G_{t+1}^T + W_{t+1})^{-1}(\theta_{t+1} - G_{t+1} m_t) \\ &= m_t + C_t G_{t+1}^T R_{t+1}^{-1}(\theta_{t+1} - a_t), \\ \text{Var}(\theta_t|\theta_{t+1}, y_{1:t}) &= C_t - C_t G_{t+1}^T R_{t+1}^{-1} G_{t+1} C_t. \end{aligned}$$

Then finally, the mean is given by

$$\begin{aligned} s_t &= \mathbb{E}(\mathbb{E}(\theta_t|y_{1:N}, \theta_{t+1})|y_{1:N}) \\ &= m_t + C_t G_{t+1}^T R_{t+1}^{-1}(s_{t+1} - a_{t+1}), \end{aligned}$$

since $\mathbb{E}(\theta_{t+1}|y_{1:N}) = s_{t+1}$. Now consider the variance, from the law of total variance:

$$\begin{aligned}
S_t &= \mathrm{Var}(\theta_t|y_{1:t},\theta_{t+1}) \\
&= \mathrm{Var}(\mathbb{E}(\theta_t|y_{1:t},\theta_{t+1})|y_{1:N}) + \mathbb{E}(\mathrm{Var}(\theta_t|y_{1:t},\theta_{t+1})|y_{1:N}) \\
&= \mathrm{Var}(m_t + C_tG_{t+1}^TR_{t+1}^{-1}(\theta_{t+1}-a_{t+1})|y_{1:N}) + C_t - C_tG_{t+1}^TR_{t+1}^{-1}G_{t+1}C_t \\
&= C_tG_{t+1}^TR_{t+1}^{-1}\mathrm{Var}(\theta_{t+1}-a_{t+1}|y_{1:N})R_{t+1}^{-1}G_{t+1}C_t + C_t - C_tG_{t+1}^TR_{t+1}^{-1}G_{t+1}C_t, \\
&= C_tG_{t+1}^TR_{t+1}^{-1}S_{t+1}R_{t+1}^{-1}G_{t+1}C_t + C_t - C_tG_{t+1}^TR_{t+1}^{-1}G_{t+1}C_t, \\
&= C_t - C_tG_{t+1}^TR_{t+1}^{-1}(R_{t+1}-S_{t+1})R_{t+1}^{-1}G_{t+1}C_t.
\end{aligned}$$

as required.

Algorithm 7: The backward smoothing algorithm for a general DLM

```
  Result: Return smoothed state $p(\theta_{0:N}|y_{1:N},\psi)$
1 Given observations of the time series, $y_{1:N}$ and a value of the static parameters $\psi$;
2 Perform the Kalman filter using algorithm 6, then using the filtered output $m_{1:N}$,
    $C_{1:N}$, $a_{1:N}$, $R_{1:N}$;
3 Sample $\theta_N \sim \mathcal{N}(m_N, C_N)$;
4 for $t$ in $N:1$ do
5     Calculate $s_t = m_t + C_tG_{t+1}^TR_{t+1}^{-1}(\theta_{t+1}-a_{t+1})$,;
6     Calculate $S_t = C_t - C_tG_{t+1}^TR_{t+1}^{-1}G_{t+1}C_t$,;
7 end
8 The smoothing state is $\mathcal{N}(s_t, S_t)$
```

Backward smoothing is summarised in algorithm 7. Figure 4.6 shows a the Kalman filter and smoothed states for a simulated first order DLM.

4.4 Parameter Inference using Gibbs Sampling

The static parameters in the DLM are $\psi = \{V_t, W_t, m_0, C_0\}$. Consider a model where the initial state is known and it remains to specify constant system and observation noise covariance matrices. Gibbs sampling can be used in order to determine the posterior distributions of V and W. In most cases, it is simpler to target the joint distribution of the latent-state and static parameters $p(\psi, \theta_{0:N}|y_{1:N})$. The joint distribution of all the random variables in the model can be written as

$$p(\theta_{0:N}, Y_{1:N}, \psi) = p(\psi)p(\theta_0)\prod_{t=1}^{N} p(y_t|\theta_t, V)p(\theta_t|\theta_{t-1}, W). \tag{4.16}$$

This factorisation is possible since the observations are conditionally independent of each other given the corresponding state and the state evolution is a first order Markov process.

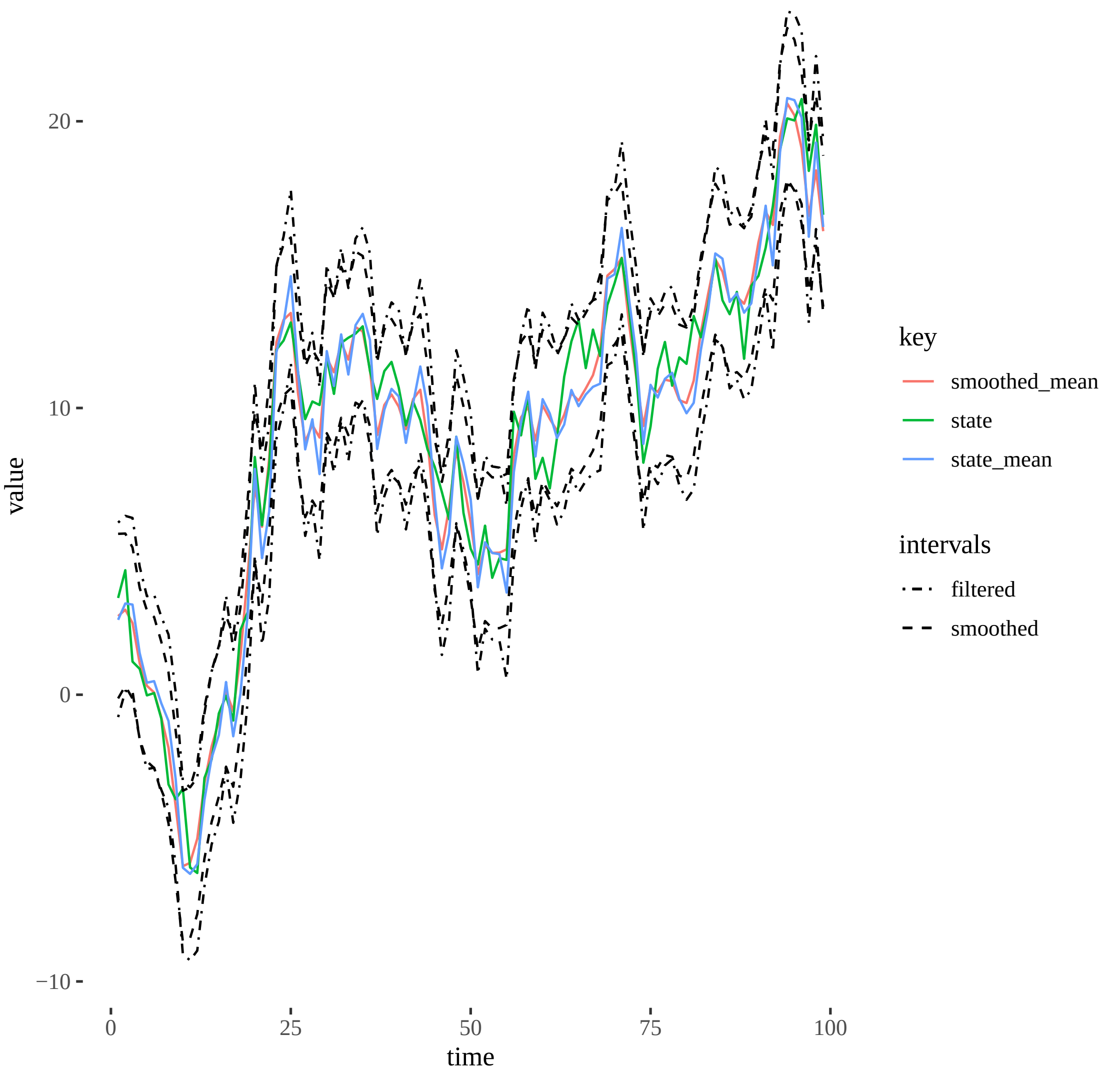


Figure 4.6: Kalman filter and backward smoothing applied to simulated data from the first order DLM first depicted in Figure 4.2 in order to learn the filtering and smoothing distributions. 99% probability intervals are included for the smoothed and filtered state, dashed represents the smoothing state and dot-dash represents the filtered state. The filtered state is more uncertain as expected.

A block Gibbs sampler is used to create a Markov chain whose stationary distribution is equal to the joint posterior distribution $p(\theta_{0:N}, W, V|y_{1:N})$. First consider deriving the full conditional distribution of a diagonal observation variance matrix, V. The prior distribution of the observation variance, V, is chosen to be the inverse Gamma distribution, $p(V) = \text{Inv-Gamma}(\alpha_v, \beta_v)$. The conditional likelihood $p(y_t|\theta_t, V)$ is Normal, with mean $F^T\theta_t$ and variance V. The inverse Gamma distribution is conjugate to the Normal distribution with known mean and unknown variance. The full conditional distribution of the i^{th} diagonal element of the observation noise covariance is derived as follows

$$\begin{aligned}
p(V_{ii}|y_{1:N},\theta_{0:N},W) &= p(\theta_0)p(V_{ii})\prod_{t=1}^{N} p(y_t|V_{ii},\theta_t)p(\theta_t|\theta_{t-1}) \\
&\propto V_{ii}^{-\alpha-1}\exp\left(-\frac{\beta}{V_{ii}}\right)(2\pi V_{ii})^{-N/2}\exp\left\{-V_{ii}^{-1}\sum_{t=1}^{N}(y_t-F_t\theta_t)_i^2\right\} \\
&= V_{ii}^{-(\alpha+N/2)-1}\exp\left\{-\frac{1}{V_{ii}}\left(\beta+\frac{1}{2}\sum_{t=1}^{N}(y_t-F_t\theta_t)_i^2\right)\right\} \\
&= \text{Inv-Gamma}\left(\alpha+\frac{N}{2},\beta+\frac{1}{2}\sum_{t=1}^{N}(y_t-F_t\theta_t)_i^2\right). \qquad (4.17)
\end{aligned}$$

For the system covariance matrix, the Inverse Wishart distribution can be used as the conjugate prior for the multivariate Normal transition kernel for θ_t with unknown covariance, W and known mean $G_t\theta_{t-1}$. The prior for W is written as $p(W) = \mathcal{W}^{-1}(\mathbf{\Psi},\nu)$. The PDF of the inverse Wishart distribution is given by

$$p(W) = \frac{|\mathbf{\Psi}|^{\frac{\nu}{2}}}{2^{\frac{\nu p}{2}}\Gamma_p(\frac{\nu}{2})}|W|^{-\frac{\nu+p+1}{2}}\exp\left\{-\frac{1}{2}\operatorname{tr}(\mathbf{\Psi}W^{-1})\right\}.$$

Where p is the dimension of the matrix W. The full conditional distribution for the parameter W given the values of the state, $\theta_{0:N}$, observation variance, V, and observations, $y_{1:N}$, can be written as

$$\begin{aligned}
p(W|\theta_{0:N},V,y_{1:N}) &= p(\theta_0)p(W)\prod_{i=1}^{N} p(\theta_i|\theta_{i-1},W) \\
&= |W|^{-\frac{\nu+p+1}{2}}\exp\left\{-\frac{1}{2}\operatorname{tr}(\mathbf{\Psi}W^{-1})\right\} \\
&\times |W|^{-\frac{N}{2}}\exp\left\{-\frac{1}{2}\sum_{i=1}^{N}(\theta_i-G\theta_{i-1})^N W^{-1}(\theta_i-G\theta_{i-1})\right\} \\
&= |W|^{\frac{\nu+N+2+1}{2}}\exp\left\{-\frac{1}{2}\operatorname{tr}(\mathbf{\Psi}W^{-1})-\operatorname{tr}\left(\frac{1}{2}\sum_{i=1}^{N}(\theta_i-G\theta_{i-1})(\theta_i-G\theta_{i-1})^T W^{-1}\right)\right\}
\end{aligned}$$

The final line is possible since a matrix trace is invariant under cyclic permutations, $\operatorname{tr}(ABC) = \operatorname{tr}(CAB) = \operatorname{tr}(BCA)$. The full conditional can be further simplified to

$$p(W|\theta_{1:N}, V, y_{1:N}) \propto |W|^{\frac{\nu+N+2+1}{2}} \exp\left\{ -\frac{1}{2}\operatorname{tr}((\mathbf{\Psi} + \sum_{i=1}^{N}(\theta_i - G\theta_{i-1})(\theta_i - G\theta_{i-1})^T)W^{-1}) \right\},$$

since the sum of matrix traces is equal to the trace of the sum of the matrices, $\operatorname{tr}(A+B) = \operatorname{tr}(A) + \operatorname{tr}(B)$ and $\operatorname{tr}(cA) = c\operatorname{tr}(A)$ for a constant c. Then the posterior distribution for the system noise covariance is recognised as the Inverse Wishart distribution

$$p(W|\theta_{1:N}, V, y_{1:N}) = \mathcal{W}^{-1}(\nu + N, \mathbf{\Psi} + \sum_{i=1}^{N}(\theta_i - G\theta_{i-1})(\theta_i - G\theta_{i-1})^T). \tag{4.18}$$

It remains to derive the full conditional distribution for the state vector. This can be done using Forward Filtering Backward Sampling (FFBS) [Frühwirth-Schnatter, 1994, Carter and Kohn, 1994]. The joint full conditional distribution for the latent-state is written as:

$$\begin{aligned} p(\theta_{0:N}|W, V, y_{1:N}) &= p(\theta_0)\prod_{t=1}^{N} p(y_t|\theta_t, V)p(\theta_t|\theta_{t-1}, W) \\ &= \prod_{t=0}^{N} p(\theta_t|\theta_{t+1:N}, y_{1:N}, W, V), \end{aligned}$$

The final factor in the product is, $p(\theta_N|y_{1:N}, W, V)$ which is the filtering distribution of θ_N determined using the Kalman filter (see section 4.2) and given by $\mathcal{N}(m_N, C_N)$. First draw from the final distribution given by the Kalman filter, then proceed recursively by drawing from $p(\theta_t|\theta_{t+1:N}, y_{1:N}, W, V)$ for $t = N-1, N-2, \dots, 0$, the form of this distribution was presented in section 4.3 and is given by

$$\begin{aligned} \gamma_t &= m_t + C_t G_{t+1}^T R_{t+1}^{-1}(\theta_{t+1} - a_{t+1}), \\ \mathit{\Gamma}_t &= C_t - C_t G_{t+1}^T R_{t+1}^{-1} G_{t+1} C_t. \end{aligned} \tag{4.19}$$

Then draw $\theta_t \sim \mathcal{N}(\gamma_t, \mathit{\Gamma}_t)$.

This defines a block Gibbs sampler, since the latent-state is sampled from the joint conditional distribution, $p(\theta_{0:N}|W, V, y_{1:N})$ and not one state at a time from $p(\theta_t|W, V, y_{1:N}, \theta_{-t}), t = 1, \dots, N$. This defines a more efficient Markov chain. The block Gibbs scheme for parameter inference in a DLM with static observation and system covariance matrices is summarised in algorithm 8.

Algorithm 8: Block Gibbs sampler for the DLM

```
  Result: Return $\psi^{(i)}, \theta_{0:N}^{(i)}, i = 1, \dots, M$
1 Given $N$ $p$-dimensional time series observations $y_{1:N}$;
2 Initialise values of the static parameters by drawing from prior distributions,
    $\psi^{(0)} = \{W^{(0)}, V^{(0)}\}$, $W^{(0)} \sim \mathcal{W}^{-1}(\mathbf{psi}, \nu)$, $V_{ii}^{(0)} \sim$ Inv-Gamma$(\alpha_v, \beta_v)$;
3 for $i$ in $1 : M$ do
4     Perform FFBS to sample a value of the latent state conditional on the other
        random variables, $\psi^{(i-1)}, y_{1:N}$ using algorithm 7;
5     Sample $W^{(i)}$ from the full-conditional in equation (4.18);
6     Sample $V^{(i)}$ from the full-conditional in equation (4.17);
7 end
```

4.5 Computational Considerations

The Kalman filter implemented naively as in algorithm 6 is prone to numerical issues. The naive implementation updates the covariance matrix at each time step using the following equation

$$C_{t+1} = R_{t+1} - R_{t+1} F_{t+1}^T Q_{t+1}^{-1} F_{t+1} R_{t+1} \tag{4.20}$$

If the value of V_t is small then

$$\begin{aligned} C_{t+1} &= R_{t+1} - R_{t+1} F_{t+1}^T Q_{t+1}^{-1} F_{t+1} R_{t+1} \\ &= R_{t+1} - R_{t+1} F_{t+1}^T (R_{t+1} + V_{t+1})^{-1} F_{t+1} R_{t+1} \\ &\approx R_{t+1} - R_{t+1}, \end{aligned}$$

this can result in some of the diagonal entries of the covariance matrix, C_{t+1} to be zero or negative due to rounding errors. This means C_{t+1} is no longer a valid covariance matrix, since the diagonal entries represent the marginal variance and hence must be positive. This section considers methods for improving the stability of filtering and backward sampling algorithms for use in real-world time series problems.

4.5.1 Joseph Form

Covariance matrices are symmetric, positive semi-definite matrices. A matrix M is positive semi-definite if for any non-zero vector a, then $aMa^T \geq 0$. One way of ensuring the covariance matrix C_t is positive semi-definite, thus ameliorating the numerical issues which arise in the naive implementation of the Kalman filter, is to use the Joseph Form update [Bucy and Joseph, 2005] presented below

$$C_t = (I - K_tF_t)R_t(I - K_tF_t)^T + K_tV_tK_t^T \tag{4.21}$$

This formula is the sum of two positive semi-definite matrices. This is true because for any matrix A with real entries A^TA is positive semi-definite and symmetric. The product of symmetric positive semi-definite matrices is positive semi-definite if and only if the product is symmetric [Meenakshi and Rajian, 1999]. The sum of two positive semi-definite matrices must also be positive semi-definite. If M and U are positive semi-definite matrices then, $aMa^T \geq 0$ and $bUb^T \geq 0$ and so obviously $a(M+U)a^T = aMa^T + aUa^T \geq 0$.

Lemma 4.1. *The Joseph form update for the posterior covariance at time t in the Kalman Filter is equivalent to the naive update:*

$$C_t = R_t - R_tF_tQ_t^{-1}F_tR_t = (I - K_tF_t)R_t(I - K_tF_t)^T + K_tV_tK_t^T$$

Proof.

$$\begin{aligned}
C_t &= (I - K_tF_t)R_t(I - K_tF_t)^T + K_tV_tK_t^T,\\
&= (R_t - K_tF_tR_t)(I - K_TF_t)^T + K_tV_tK_t^T,\\
&= R_t - K_tF_tR_t - R_t(K_tF_t)^T + K_tF_tR_tF_t^TK_t^T + K_tV_tK_t^T,\\
&= R_t + K_t(F_tR_tF_t^T + V_t)K_t^T - K_tF_tR_t - R_tF_t^TK_t^T,\\
&= R_t + K_tQ_tK_t^T - K_tF_tR_t - R_tF_t^TK_t^T,\\
&= R_t + R_tF_tQ_t^{-1}Q_t(Q_t^{-1})^TF_t^TR_t^T - R_tF_tQ_t^{-1}F_tR_t - R_tF_t^T(R_tF_tQ_t^{-1})^T,\\
&= R_t - R_tF_tQ_t^{-1}F_tR_t.
\end{aligned}$$

□

Line 6 of the proof follows since for any invertible matrix, A, $(A^T)^{-1} = (A^{-1})^T$ and for any symmetric matrix B, $B^T = B$. Hence $Q_t(Q_t^{-1})^T = Q_t(Q_t^T)^{-1} = Q_tQ_t^{-1} = I$. The Joseph form update can also be used to sample the state in the backward sampler from algorithm 7.

Lemma 4.2. *The Joseph form update for the covariance at time t in the forward filtering backward sampling algorithm is equivalent to the naive update:*

$$\Gamma_t = (I - H_tG_{t+1})C_t(I - H_tG_{t+1})^T + H_tW_tH_t^T = C_t - C_tG_{t+1}R_{t+1}^{-1}G_{t+1}C_t,$$

where $H_t = C_tG_{t+1}R_{t+1}^{-1}$

Proof.

$$
\begin{aligned}
\Gamma_t &= (I - H_t G_{t+1}) C_t (I - H_t G_{t+1})^T + H_t W_t H_t^T, \\
&= (C_t - H_t G_{t+1} C_t)(I - H_T G_{t+1})^T + H_t W_t H_t^T, \\
&= C_t - H_t G_{t+1} C_t - C_t (H_t G_{t+1})^T + H_t G_{t+1} C_t G_{t+1}^T H_t^T + H_t W_t H_t^T, \\
&= C_t + H_t (G_{t+1} C_t G_{t+1}^T + W_t) H_t^T - H_t G_{t+1} C_t - C_t G_{t+1}^T H_t^T, \\
&= C_t + H_t R_{t+1} H_t^T - H_t G_{t+1} C_t - C_t G_{t+1}^T H_t^T, \\
&= C_t + C_t G_{t+1} R_{t+1}^{-1} R_{t+1} (R_{t+1}^{-1})^T G_{t+1}^T C_t^T - C_t G_{t+1} R_{t+1}^{-1} G_{t+1} C_t - C_t G_{t+1}^T (C_t G_{t+1} R_{t+1}^{-1})^T, \\
&= C_t - C_t G_{t+1} R_{t+1}^{-1} G_{t+1} C_t.
\end{aligned}
$$

□

The Joseph form update requires more calculations, but results in a more stable filter.

4.5.2 Square Root Filtering and Sampling

The Kalman filter is concerned with updating the mean and covariance of the latent state recursively. The backward sampler uses the filtered values to sample a single trajectory of the state for use in a Gibbs sampling algorithm. The states are sampled from multivariate Normal distributions. One method of sampling from a multivariate Normal distribution begins by first calculating the Cholesky decomposition of the covariance matrix. The Cholesky decomposition of a symmetric, positive definite matrix, A with real entries is $A = LL^T$, where L is a lower triangular matrix. In order to simulate values from a p dimensional multivariate Normal distribution with mean μ and covariance Σ, first calculate the Cholesky decomposition $\Sigma = LL^T$, then simulate p standard Normal random variables z then $\mu + Lz$ is a draw from a multivariate Normal distribution with mean μ and covariance Σ:

$$
\begin{aligned}
\mathbb{E}(\mu + Lz) &= \mu + L\mathbb{E}(z) = \mu \\
\operatorname{Var}(\mu + Lz) &= L \operatorname{Var}(z) L^T, \\
&= LL^T = \Sigma.
\end{aligned}
$$

The Kalman filter can be implemented in terms of the Cholesky decomposition, this means the lower triangular matrix L is updated and hence the product LL^T is guaranteed to be symmetric and positive semi-definite. Since the decomposition is already available, it is straightforward to sample a state trajectory given the Cholesky factor, L. Other square root filters have been proposed, using the $\Sigma = UDU^T$ decomposition [Thornton and Bierman, 1980, Bierman, 2006]. In general square root filters will increase numerical precision of the calculated covariance by a factor two which is useful when numerical precision is low.

The Kalman filter can also be implemented using the singular value decomposition [Wang et al., 1992, Kulikova and Tsyganova, 2017]. The singular value decomposition is a factorisation of a rectangular $m \times n$ real or complex matrix, A such that $A = USB^\star$ where $B^\star$ denotes the conjugate transpose of B. U is an $m \times m$ unitary matrix, S is a unique diagonal matrix containing the singular values of the matrix A and $B^\star$ is a unitary matrix. A unitary matrix, U multiplied by its conjugate transpose is equal to the identity, $UU^\star = I$. Hence the transpose of a unitary matrix it equal to its inverse.

Implementing the Kalman filter using the SVD has another computational advantage when performing the backward sampler. The SVD can be updated and sampling from the latent state is straightforward given the SVD of the covariance. In order to simulate from a multivariate Normal distribution with a $p \times p$ covariance matrix $\Sigma = USU^T$ and mean μ then sample p standard Normal random variables, z and transform them such that $\mu + US^{\frac{1}{2}}z$. Then $\text{Var}(\mu + US^{\frac{1}{2}}z) = USU^T = \Sigma$ and $\mathbb{E}(\mu + USz) = \mu$.

The SVD filter starts by decomposing the covariance matrix at time $t = 0$, $C_0 = U_{C_0}S^2_{C_0}U^T_{C_0}$ and updating the unitary matrix U_{C_t} and matrix of singular values S_{C_0} at each time point. To initialise the filter, the SVD of the system and observation noise covariance matrices is calculated and the following transformations performed

$$\begin{aligned} W^{\frac{1}{2}} &= S_W^{\frac{1}{2}}U_W^T, \\ V^{-\frac{1}{2}} &= S_V^{-\frac{1}{2}}U_V^T. \end{aligned}$$

In the first step of the main filtering recursions the state is advanced to the next time-point

$$\begin{aligned} a_t &= G_t m_{t-1}, \\ R_t &= G_t U_{C_{t-1}} S^2_{C_{t-1}} U^T_{C_{t-1}} G_t^T + W \end{aligned} \tag{4.22}$$

Then the SVD of R_t is calculated. However it equivalent to first calculate the SVD of the vertically concatenated $2n \times n$ matrix

$$\begin{pmatrix} D_{C_{t-1}} U^T_{C_{t-1}} G_t^T \\ W^{\frac{1}{2}} \end{pmatrix} = U_{R_t} S_{R_t} B^T_{R_t}. \tag{4.23}$$

Then calculate R_t by pre-multiply each side by the transpose

$$\begin{aligned} G_t U_{C_{t-1}} S_{C_{t-1}}^T S_{C_{t-1}} U_{C_{t-1}}^T G_t^T + (W^{\frac{1}{2}})^T W^{\frac{1}{2}} &= B_{R_t} S_{R_t}^T U_{R_t}^T U_{R_t} S_{R_t} B_{R_t}^T, \\ G_t U_{C_{t-1}} S_{C_{t-1}}^2 U_{C_{t-1}}^T G_t^T + W &= B_{R_t} S_{R_t}^2 B_{R_t}^T \end{aligned}$$

this follows since U_{R_t} is a unitary matrix. Then R_t is given by $B_{R_t} S_{R_t}^2 B_{R_t}^T$ and determined by the SVD calculated in equation (4.23).

Next an observation is made at time t and the state update is calculated. The update to the state covariance in the naive Kalman filter from equation (4.13) and substitute the SVD of C_t and R_t

$$\begin{aligned} C_t &= (R_t^{-1} + F_t^T V^{-1} F_t)^{-1}, \\ (U_{C_t} S_{C_t}^2 U_{C_t}^T)^{-1} &= (U_{R_t} S_{R_t}^2 U_{R_t}^T)^{-1} + F_t^T V^{-1} F_t, \\ &= (U_{R_t}^T)^{-1} S_{R_t}^{-2} U_{R_t}^{-1} + (U_{R_t}^{-1})^T U_{R_t}^T F_t^T V^{-1} F_t U_{R_t} U_{R_t}^{-1}, \\ &= (U_{R_t}^T)^{-1} (S_{R_t}^{-2} + U_{R_t}^T F_t^T V^{-1} F_t U_{R_t}) U_{R_t}^{-1}, \end{aligned} \tag{4.24}$$

Then consider the SVD of the value in brackets using the same method as used previously, by first considering the SVD of the vertically concatenated matrix

$$\begin{pmatrix} V^{-\frac{1}{2}} F_t U_{R_t} \\ S_{R_t}^{-1} \end{pmatrix} = U^\star \Lambda B^{\star T}.$$

Then pre-multiply by the transpose to get

$$\begin{aligned} U_{R_t}^T F_t^T (V^{-\frac{1}{2}})^T V^{-\frac{1}{2}} F_t U_{R_t} + (S_{R_t}^{-1})^T S_{R_t}^{-1} &= B^\star \Lambda^T U^{\star T} U^\star \Lambda B^{\star T}. \\ U_{R_t}^T F_t^T V^{-1} F_t U_{R_t} + S_{R_t}^{-2} &= B^\star \Lambda^2 B^{\star T} \end{aligned}$$

Then substituting the value into equation (4.24)

$$\begin{aligned} (U_{C_t} D_{C_t}^2 U_{C_t}^T)^{-1} &= (U_{R_t}^T)^{-1} (B^\star \Lambda^2 B^{\star T}) U_{R_t}^{-1}, \\ U_{C_t}^T D_{C_t}^{-2} U_{C_t} &= U_{R_t}^T B^\star \Lambda^2 B^{\star T} U_{R_t}^{-1}. \end{aligned}$$

It follows that $U_{C_t} = U_{R_t} B^\star$ and $D_{C_t} = \Lambda^{-1}$. The SVD Kalman filter is summarised in Algorithm 9.

The singular value decomposition filter performs more operations per step than the naive Kalman filter and hence is expected to be slower. In order to test this hypothesis multiple runs of the SVD filter and Kalman Filter are compared. The Java Microbench-

Algorithm 9: Kalman filter updating the singular value decomposition of the covariance matrix.

```
   Result: Return mean and variance of filtering distribution $p(\theta_{0:N}|y_{1:N}, \psi)$
1  Given observations $y_{1:N}$ and parameters $\psi = \{m_0, C_0, W, V\}$;
2  Calculate the SVD of the latent-state covariance matrix, $C_0 = U_{C_0} D_{C_0} U_{C_0}^T$;
3  Calculate the SVD of $W = U_W D_W U_W^T$ and $V = U_V D_V U_V^T$;
4  Calculate $W^{\frac{1}{2}} = D_W^{\frac{1}{2}} U_W^T$ and $V^{-\frac{1}{2}} = D_V^{-\frac{1}{2}} U_V^T$;
5  for $t$ in 1:N do
6      Advance the state $a_t = G_t m_{t-1}$;
7      Calculate the SVD of $\begin{pmatrix} D_{C_{t-1}} U_{C_{t-1}}^T G_t^T \\ W^{\frac{1}{2}} \end{pmatrix}$, which is $R_t = U_{R_t} D_{R_t} U_{R_t}^T$;
8      Calculate the one-step forecast $f_t = F_t a_t$;
9      Calculate the SVD of $\begin{pmatrix} V^{-\frac{1}{2}} F_t U_{R_t} \\ D_{R_t}^{-1} \end{pmatrix}$, which is $U^\star \Lambda B^{\star T}$;
10     Calculate $U_{C_t} = U_{R_t} B^\star$ and $D_{C_t} = \Lambda^{-1}$;
11     Update the state covariance $C_t = U_{C_t} D_{C_t} U_{C_t}^T$;
12     Calculate the Kalman gain as $K_t = C_t F_t V^{-1}$;
13     Update the state mean $m_t = a_t + K_t(y_t - f_t)$;
14 end
```

Method	Mean (op/s)	Lower 0.05%	Upper 99.5%
Kalman filter	3389.50	3356.84	3422.15
SVD filter	2442.37	2387.70	2497.05

Table 4.1: Performance in operations per second for the Kalman filter and the SVD filter with mean, and 99% probability interval assuming the independent runs of the algorithm are Normally distributed

mark Harness[1] (JMH) was used in order to determine the performance differences between the two methods. 20 warmup iterations are performed for each JVM fork (a fork is a distinct JVM instance), 20 measurement iterations are performed for each fork, 10 forks were used in total for each method.

The SVD filter and Kalman filter (using the Joseph form update) are both applied to a time series of length 10 (contained in a `Vector`) simulated from a first order polynomial DLM. The results are presented in number of operations per second, indicating how many times the method could run in a single second, hence a higher number is better. The results are presented in table 4.1.

[1]`http://openjdk.java.net/projects/code-tools/jmh/`

4.6 Online Parameter Estimation

There are several algorithms for online simultaneous state and parameter estimation for dynamic linear models. The first considered algorithm is an analytic solution for calculating the posterior distribution of the latent-state and observation variance. The other solutions presented in this section are simulation based filters which produce approximations of the posterior distribution and can generalise to non-Gaussian, non-linear state space models.

The Kalman filter is an online algorithm, since it only depends on the previous state and the current observation, in addition any filtering algorithm which can be expressed as a pure function in this way is an online algorithm. The general signature for a `stepFunction` is:

```
def stepFunction[A, S](state: S, obs: A): S
```

Where the state and observation at time t is represented by the types `S` and `A` respectively. The application of the filter is performed using a `scanLeft` or a `foldLeft` depending on if all the states are of interest or only the final state. To apply the filter to a collection of observations, `ys: F[A]` with an initial state `init: S` using a scan:

```
ys.scanLeft(init)(stepFunction)
```

This will produce a collection of states representing the filtering distribution. In some cases, particle approximations are used in the filtering step. This means without proper care, the step function is non-deterministic and hence not a pure function. The function can be written using a random sampling probability monad, as seen in section 2.1.11. `foldM` can be used to perform a fold using a binary function returning a monad as explained in section 2.1.15.

4.6.1 Conjugate Filter for Unknown Observation Variance

There is an extension to the Kalman filter which permits an unknown observation variance which can be learned online. This extension is exact and closed form. For simplicity, the examples in this section consider a first order univariate Gaussian dynamic linear model defined in equation (4.2).

Consider that the value of the system variance, W, is known. The value of the observation variance, V, assumed scalar, can be found using a conjugate update.

The inverse Gamma distribution is conjugate to the Normal distribution with an unknown variance. The prior distribution for the precision $\tau = 1/V$ is $\text{Gamma}(n_t/2, d_t/2)$, where $n_t = n_{t-1} + 1$ and $d_t = d_{t-1} + V_{t-1}e_t^2/Q_t$. Then the inverse of the mean of the Gamma distribution gives the mean of the variance, $V_t = d_t/n_t$, at time t. The mean of the variance at time t is given by the recursive relationship:

$$V_t = V_{t-1} + \frac{v_{t-1}}{n_t}\left(\frac{e_t^2}{Q_t} - 1\right). \tag{4.25}$$

Where $e_t = y_t - F_t^T a_t$ is the residual, where $a_t = G_t m_{t-1}$ and $Q_t = F_t^T R_t F_t + V_t$ is the forecast variance where $R_t = G_t^T C_{t-1} G_t + W_t$. Figure 4.7 shows the expectation of the observation variance and 95% posterior intervals, along with the value used to simulate the data plotted as a dashed line.

This method can be extended to multivariate DLMs using the inverse Wishart distribution which is conjugate to the multivariate Normal distribution with unknown covariance. Additionally, diagonal observation noise covariance matrices can be considered using n univariate inverse Gamma distributions. Unfortunately, there is no analytic solution for learning the value of the system covariance, W. However there are several simulation based filters which aim to tackle this problem.

4.6.2 Liu and West

The Liu and West filter [Liu and West, 2001] uses kernel smoothing to approximate the distribution of parameters using a mixture of Normals:

$$p(\psi|Y_{1:t-1}) \approx \sum_{m=1}^{N} w_{t-1}^{(m)} \mathcal{N}(\psi|m_{t-1}^{(m)}, (1-a^2)\Lambda_{t-1}).$$

Here a is a smoothing parameter and Λ_{t-1} is the variance of the parameters given the information at time $t-1$, $\text{Var}(\psi|Y_{1:t-1})$. $m_{t-1}^{(i)} = a\psi_{t-1}^{(i)} + (1-a)\bar{\psi}_{t-1}$.

The Liu and West filter uses a method from the auxiliary particle filter [Pitt and Shephard, 1999b] for advancing the particle cloud. Assume that the distribution of the latent state at time t is approximated by the weighted particle cloud, $\{\theta_t^{(i)}, w_t^{(i)}|i = 1, \ldots, M\}$. The transition density function $p(\theta_t|\theta_{t-1}, \psi)$ can be used to advance the state particles, $\theta_t^{(i)} = p(\theta_t|\theta_{t-1}^{(i)}, \psi), i = 1, \ldots, M$. However, this naive transition density function can advance the particles to areas of low probability density when considering the observation at time t. This means the weight, $w_t^{(i)} = p(y_t|\theta_t^{(i)}, \psi), i = 1, \ldots, M$, associated with the low probability particles are low and hence these particles are unlikely to be selected during the resampling stage. If the observation at time t is available, then the transition density function can be improved using auxiliary variables.

Firstly, the mean (or mode) of the transition density is calculated, $\mathbb{E}(p(\theta_t|\theta_{t-1} = \theta_{t-1}^{(k)}, \psi))$ and given the observation at time t auxiliary variables are sampled from

$$Pr(k^{(i)} = k) \propto w_{t-1}^{(k)} p(y_t|\mu_t^{(k)}, \psi),$$

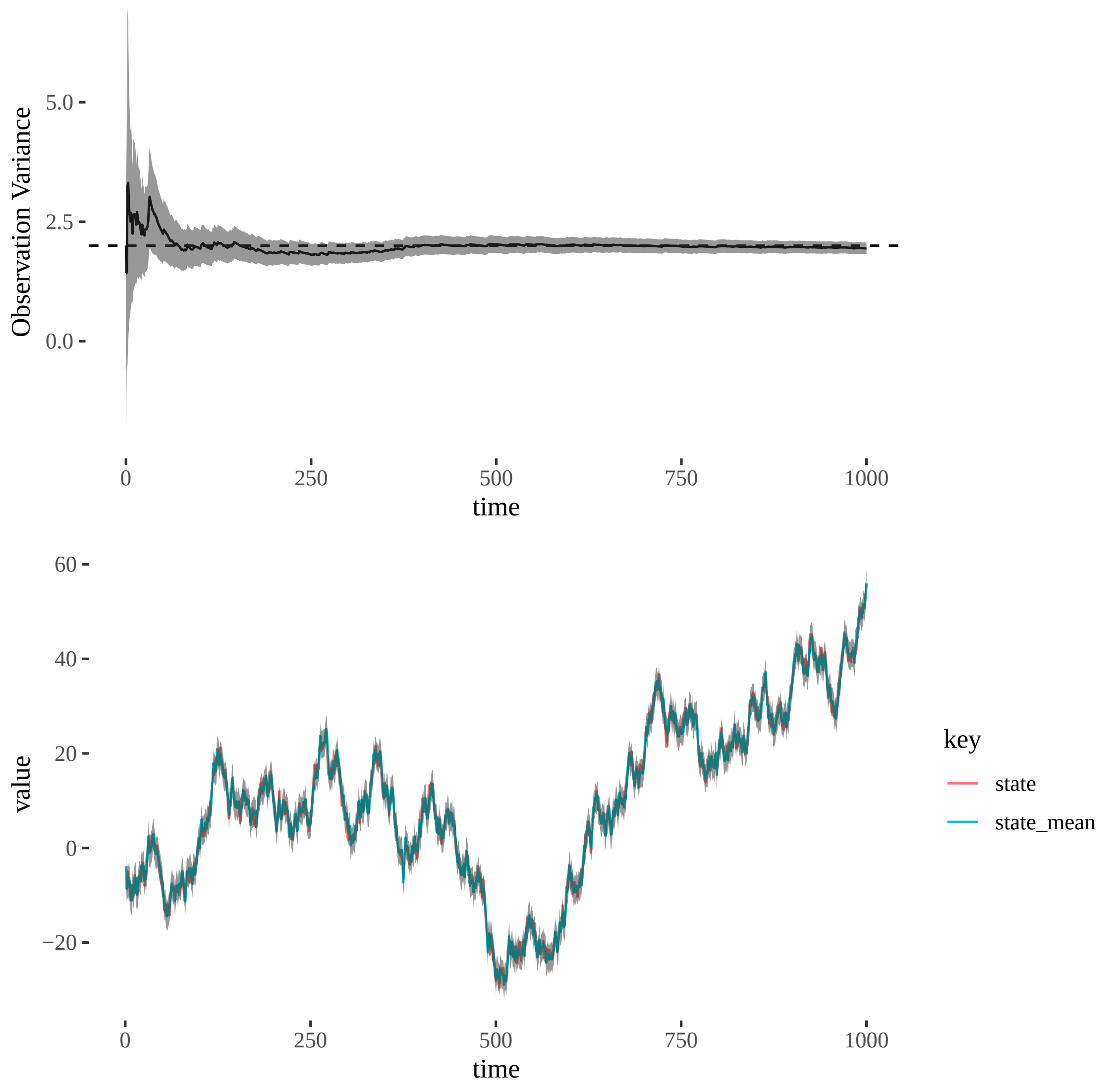


Figure 4.7: An application of the conjugate filter to learn the posterior distribution of the observation variance for simulated observations from the first order DLM. The mean of the observation variance parameter at each time point with ±2 standard deviations from the mean. The actual value used to simulate the data is represented as a horizontal dashed line, (top). The filtered state and 95% credible intervals, (bottom).

for $i = 1, \dots, M$. Then for each auxiliary variable sample the next particle from the transition density

$$\theta_t^{(i)} \sim p(\theta_t | \theta_{t-1}^{k(i)}, \psi).$$

This has the effect of pre-selecting the particles which are likely to have a high probability density (and weight) at time t. This method of advancing the particle cloud was first introduced for the auxiliary particle filter [Pitt and Shephard, 1999b]. The Liu and West algorithm to determine the full-joint posterior of a general state space model is outlined in algorithm 10.

Algorithm 10: Liu and West filter

```
   Result: Return a weighted sample representing the joint posterior distribution
           $\{\theta_{0:N}^{(i)}, \psi_{0:N}^{(i)}, w_{0:N}^{(i)} | i = 1, \ldots, M\}$
1  Initialise the state particles by sampling from the state prior distribution
     $\theta_0^{(i)} \sim p(\theta_0)$ for $i = 1, \ldots, M$;
2  Sample the initial parameter particles from the parameter prior distribution
     $\psi_0^{(i)} \sim p(\psi)$ for $i = 1, \ldots, M$;
3  Initialise the particle weights $w_0^{(i)} = 1/M$ for $i = 1, \ldots, M$;
4  for $t$ in 1:N do
5      for $i$ in 1:M do
6          Calculate prior point estimates of the state and parameters
7              $\mu_t^{(i)} = \mathbb{E}(\theta_t | \theta_{t-1}^{(i)}, \psi_{t-1}^{(i)}),$
               $m_{t-1}^{(i)} = a\psi_{t-1}^{(i)} - (1-a)\bar{\psi}_{t-1}.$
8          Sample an auxiliary variable, $k^{(i)}$, with probabilities proportional to
9          $Pr(k^{(i)} = k) \propto w_{t-1}^{(k)} p(y_t | \mu_t^{(k)}, m_{t-1}^{(k)})$;
10         Sample a new parameter vector from the mixture of Normals
               $\psi_t^{(i)} \sim \mathcal{N}(m_{t-1}^{k^{(i)}}, (1-a^2)\Lambda_{t-1}),$
               where $\Lambda_{t-1} = \sum_{i=1}^{M} w_{t-1}^{(i)} (\psi_{t-1}^{(i)} - \bar{\psi}_{t-1})(\psi_{t-1}^{(i)} - \bar{\psi}_{t-1})^T.$
11         Sample a value of the state vector, given the new set of parameter particles
             and the auxiliary state
12         $\theta_t^{(i)} \sim p(\theta_t | \theta_{t-1}^{(k^{(i)})}, \psi_t^{(i)})$;
13         Compute importance weights: $w_t^{(i)} \propto \frac{p(y_t | \theta_t^{(i)}, \psi_t^{(i)})}{p(y_t | \mu_t^{(k^{(i)})}, m_{t-1}^{(k^{(i)})})}$;
14     end
15     Resample $\{\psi_{0:t}^{(i)}, \theta_{0:t}^{(i)} | i = 1, \ldots, M\}$ by sampling $M$ times with replacement
         with probability proportional to the importance weights;
16 end
```

The algorithm is applied to the first order DLM introduced in section 4.1. The number of particles is chosen to be $M = 500$ and the smoothing parameter $a = \frac{3\delta - 1}{2\delta}$, with $\delta = 0.95$ for the mixture of Gaussians. The parameter and state prior distributions are chosen to be:

$$\begin{aligned}
p(V) &= \text{Inv-Gamma}(3,3), \\
p(W) &= \text{Inv-Gamma}(3,3), \\
p(\theta_0) &= \mathcal{N}(0,10).
\end{aligned} \tag{4.26}$$

Figure 4.8 shows the parameter estimates for simulated data from the first order Gaussian DLM. The Liu and West algorithm is able to approximately identify the system and observation variances of the first order model.

The Liu and West filter is very flexible and general, as it does not assume the form of the observation distribution or the state transition kernel. This particle filter can be used for non-linear, non-Gaussian state space models which are considered in chapters 5 and 6.

In the case of DLMs, the state is Gaussian, the observation is Gaussian and linear. Hence certain simplifications can be made which avoids the computational complexity of the Liu and West filter. The Storvik filter in section 4.6.4 relies on a tractable transition kernel and observation distribution in order to perform a conditionally conjugate update for the noise covariance matrices. The filtering distribution of a DLM can be found analytically using the Kalman filter hence it is attractive to combine particle filtering for the intractable parameters and the Kalman filter to marginalise the state exactly. This is considered in section 4.6.3.

4.6.3 Rao-Blackwellised Particle Filter

The Rao-Blackwellised particle filter [Doucet et al., 2000] can be used to marginalise the latent-state exactly using the Kalman filter, while sampling the parameters using a particle filter. The latent-state and unknown parameters are represented by M particles, the Kalman filter is applied to each particle quadruple $\{V_t^{(i)}, W_t^{(i)}, m_t^{(i)}, C_t^{(i)}\}$ representing the latent-state and the value of the unknown parameters given the observed data so far. This reduces the variance of the particle cloud and hence a smaller number of particles can be used, reducing the computational cost.

The Rao-Blackwell filter is a perfect example where functional programming and function composition can be used to simplify the implementation of algorithms. The Rao-Blackwell filter consists of the application of the Kalman filter to the quadruple, $\{V_t^{(i)}, W_t^{(i)}, m_t^{(i)}, C_t^{(i)}\}$, hence the function implementing the Kalman filter can be re-used.

Once the new parameters have been proposed a single step of the Kalman filter, `kalmanStep` is applied using a `map` function:

```
(ps zip state).map { case (p, x) => kalmanStep(p)(y, x) }
```

`ps` contains a collection of newly proposed system and observation variances and `state`

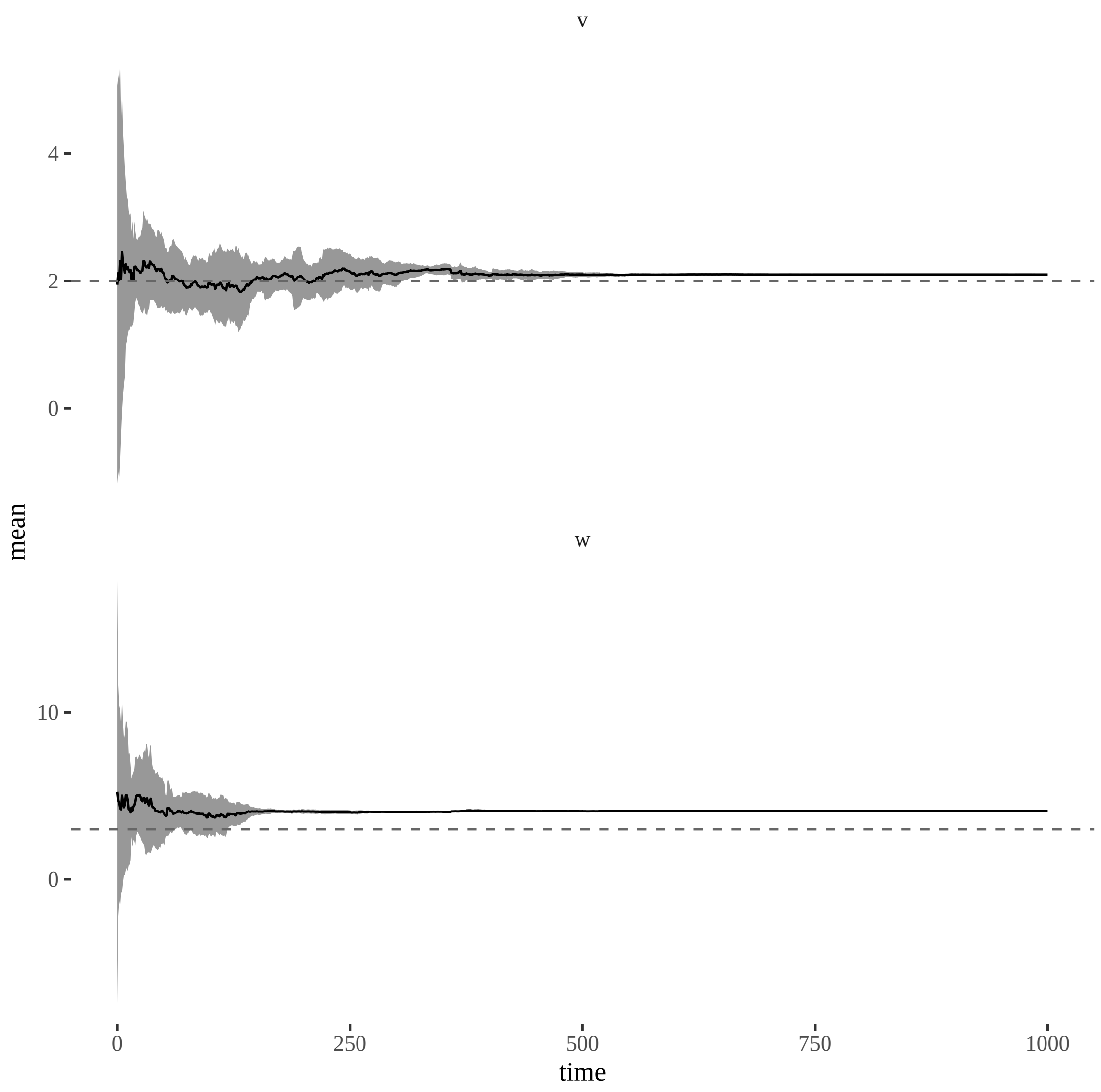


Figure 4.8: Running mean and ±2 standard deviations of the online estimates of the system and observation variances for simulated data from the first order univariate DLM using the Liu and West filter. The true parameter values are plotted using dashed lines.

contains the particles representing the mean and variance of the state. Since two lists must be combined and mapped over, `parMapN` can be is used which is a parallel map of multiple collections

```
(ps, state).parMapN { case (p, x) => kalmanStep(p)(y, x) }
```

This performs the map in parallel on a CPU. This code-reuse is straightforward, since the function `kalmanStep` does not have any side effects and depends only on its arguments and returns a deterministic state update, it is simple to reason about and incorporate in

Algorithm 11: The Rao-Blackwellised particle filter for a DLM

```
   Result: Return a weighted sample representing the joint posterior distribution
           ${m^{(i)}_{0:N}, C^{(i)}_{0:N}, \psi^{(i)}_{0:N}, w^{(i)}_{0:N} | i = 1, \dots, M}$
 1 Given a set of observations $y_{1:N}$ Sample the initial parameter particles from the
     parameter prior distribution $V^{(i)}_0 \sim p(V)$, $W^{(i)}_0 \sim p(W)$ for $i = 1, \dots, M$;
 2 Initialise the particle weights $w^{(i)}_0 = 1/M$ for $i = 1, \dots, M$;
 3 for $t$ in 1:N do
 4     for $i$ in 1:M do
 5         Sample new parameters from a proposal distribution:
           $V^{(i)}_t \sim q(V_t | V^{(i)}_{0:t-1}, y_{1:t-1})$, $W^{(i)}_t \sim q(W_t | W^{(i)}_{0:t-1}, y_{1:t-1})$;
 6         Perform the a single step of the Kalman filter (algorithm 6):
 7         Advance the state:
               $a^{(i)}_t = G_t m^{(i)}_t,$
               $R^{(i)}_t = G_t C^{(i)}_t G^T_t + W^{(i)}_t$
 8         Calculate the one-step forecast:
               $f^{(i)}_t = F^T_t a^{(i)}_t$
               $Q^{(i)}_t = F^T_t R^{(i)}_t F_t + V^{(i)}_t$
 9         Update to the state posterior:
               $m^{(i)}_t = m^{(i)}_t + R^{(i)}_t F_t (Q^{-1}_t)^{(i)} (y_t - f^{(i)}_t),$
               $C^{(i)}_t = R^{(i)}_t - R^{(i)}_t F_t (Q^{-1}_t)^{(i)} F_t R^{(i)}_t.$
            Calculate importance weights: $w^{(i)}_t = \mathcal{N}(y_t; f^{(i)}_t, Q^{(i)}_t)$;
10     end
11     Resample ${V^{(i)}_{1:t}, W^{(i)}_{0:t}, m^{(i)}_{0:t}, C^{(i)}_{0:t} | i = 1, \dots, M}$ by sampling $M$ times with
         replacement with probability proportional to the importance weights;
12 end
```

the implementation of the Rao-Blackwell filter.

Figure 4.9 shows an application of the Rao-Blackwell filter to the simulated first order DLM introduced in section 4.1. The prior distributions for the parameters are chosen to be the same as in the Liu and West Filter specified in equation (4.26).

In order to initialise the particle cloud, the $M = 300$ parameters are sampled from the prior distributions and the weights are initialised to $1/M$. In order to propose the parameters, the mixture of Gaussians approximation from the Liu and West filter in section 4.6.2 is used with $\delta = 0.95$.

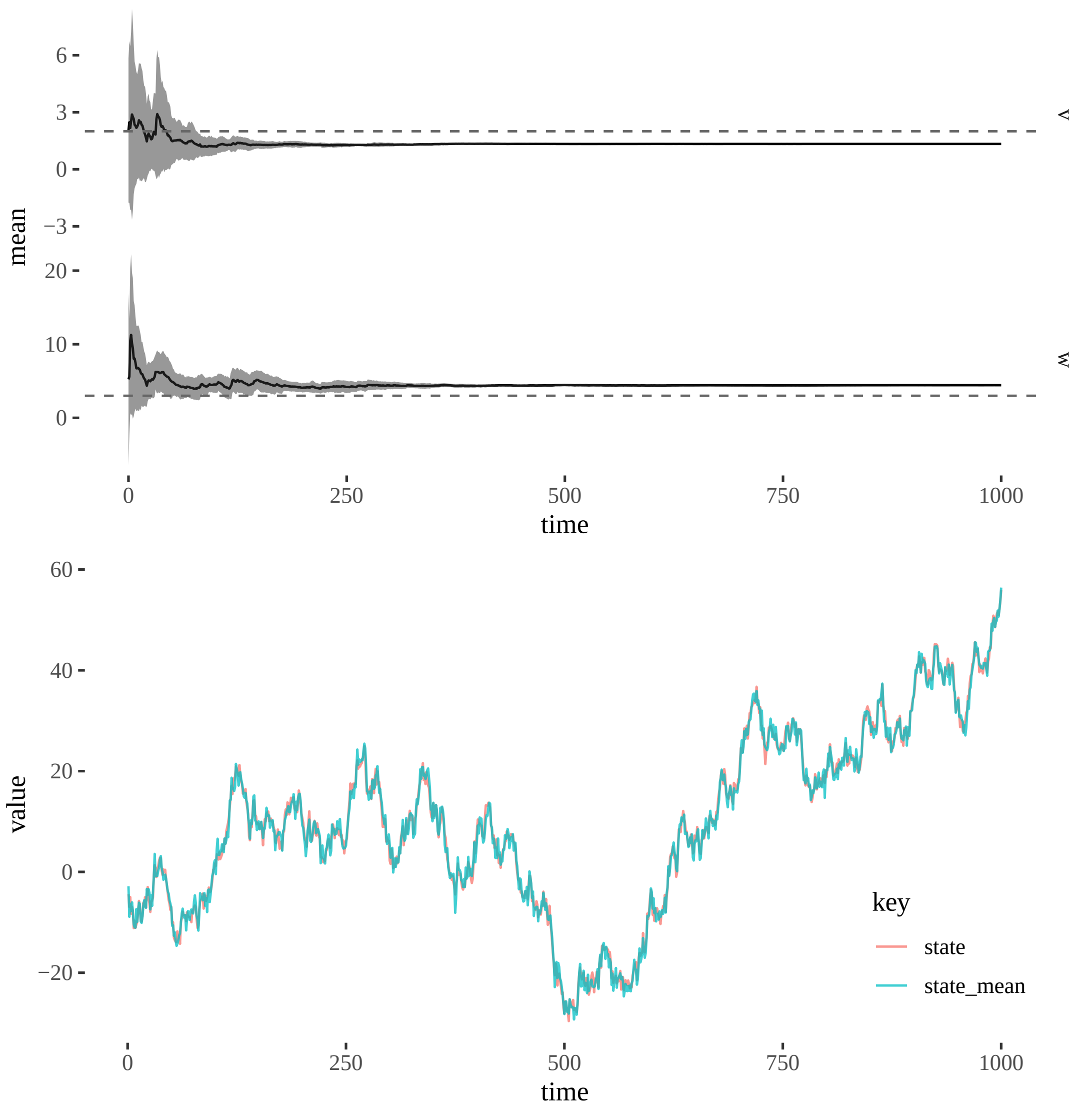


Figure 4.9: Rao-Blackwellised particle filter applied to data simulated from the first order DLM first cosidered in figure 4.2. Parameter estimates ±2 standard deviations by time for the variance of the observation and system noise

4.6.4 Storvik Filter

The Storvik filter [Storvik, 2002] is another online algorithm which can be used to recursively estimate the joint state and parameter posterior distribution for DLMs. The Storvik filter requires a tractable transition kernel for the latent state, but is flexible in the observation distribution. Unlike the Liu and West particle filter, the parameter posterior distributions are not approximated with a weighted particle cloud, instead the parameter state is summarised by a set of sufficient statistics.

Consider a DLM with unknown diagonal observation and system noise covariance

Algorithm 12: The Storvik filter.

```
   Result: Return a weighted sample representing the joint posterior distribution
           $\{\theta_{0:N}^{(i)}, \psi_{0:N}^{(i)}, w_{0:N}^{(i)} | i = 1, \dots, M\}$
 1 Initialise the state particles by sampling from the state prior distribution
     $\theta_0^{(i)} \sim p(\theta_0)$ for $i = 1, \dots, M$;
 2 Initialise the particle weights $w_0^{(i)} = 1/M$ for $i = 1, \dots, M$;
 3 for $t$ in 1:N do
 4     for $i$ in 1:M do
 5         Sample new parameter values using the sufficient statistics
             $\psi_t^{(i)} \sim p(\psi_t | S_{t-1}^{(i)})$;
 6         Advance the state particles according to the state transition function
             $\theta_t^{(i)} \sim p(\theta_t | \theta_{t-1}^{(i)}, \psi_t^{(i)})$;
 7         Calculate the importance weights $w_t^{(i)} \propto p(y_t | \theta_t^{(i)}, \psi_t^{(i)})$;
 8     end
 9     Resample, $\{\theta_t^{(i)}, \psi_t^{(i)}, S_{t-1}^{(i)}, i = 1, \dots, M\}$, by sampling $M$ times with
         replacement, ie. sample index $j$ from a Multinomial distribution with
         probabilities $w_t^{(i)}$, $i = 1, \dots, M$, and selecting the $j^{th}$ particles;
10     Update the sufficient statistics, $S_t^{(i)} = T(S_{t-1}^{(i)}, x_t^{(i)}, y_t)$ for $i = 1, \dots, M$;
11 end
```

matrices. The inverse Gamma distribution is conjugate to a Gaussian distribution with unknown variance. M state particles are initialised by sampling from the prior distribution of the initial state, $\theta_0^{(i)} \sim \mathcal{N}(m_0, C_0), i = 1, \dots, M$. Next specify the prior distributions for the diagonal elements of the observation and system noise covariance matrices.

As each observation arrives, the sufficient statistics are updated recursively starting from the values of the shape and scale specified a-priori. The sufficient statistics for the k^{th} diagonal element of the observation covariance, $V_{kk}, k = 1, \dots, p$ at time t are the shape and scale of the inverse Gamma distribution:

$$\begin{aligned}
\alpha_t^{(k)} &= \alpha_{t-1}^{(k)} + \frac{1}{2}, \\
\beta_t^{(k)} &= \beta_{t-1}^{(k)} + \frac{1}{2}(y_t - F_t\theta_t)_k^2.
\end{aligned}$$

This represents the shape and scale of the posterior distribution $p(V_{kk}|\theta_t, y_t) = \text{Inv-Gamma}(\alpha_t^{(k)}, \beta_t^{(k)})$ at time t. This distribution is sampled from M times at each time step. The sufficient statistics for the diagonal elements of the system covariance are:

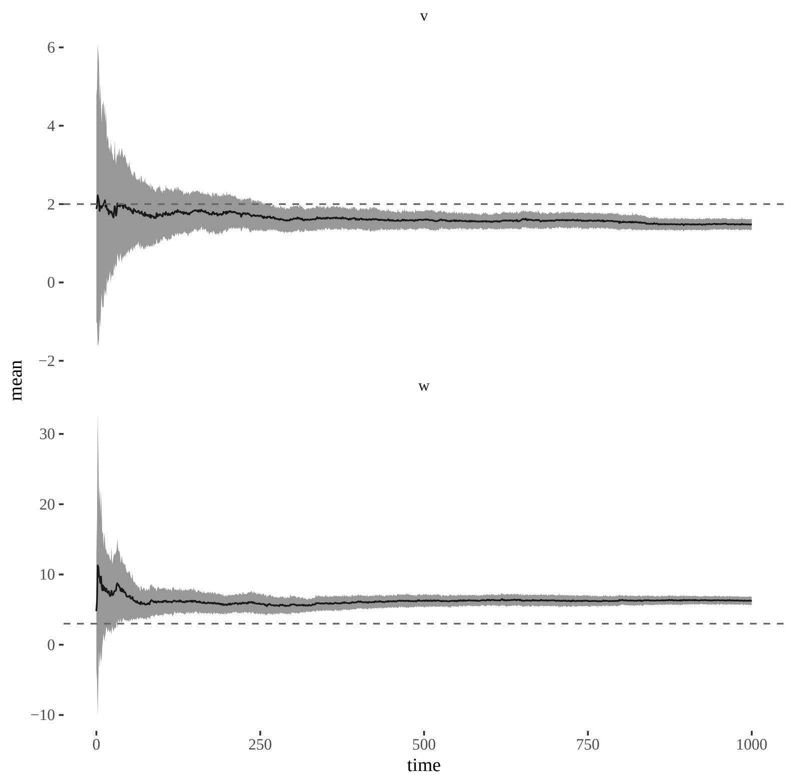


Figure 4.10: Observation and system variance of a first order DLM estimated using the Storvik filter, the actual parameter values used to simulate the data is plotted using horizontal dashed lines.

$$\begin{aligned}\alpha_t^{(j)} &= \alpha_{t-1}^{(j)} + \frac{1}{2},\\ \beta_t^{(j)} &= \beta_{t-1}^{(j)} + \frac{1}{2}(\theta_t - G_t\theta_{t-1})^2_{(j)}.\end{aligned}$$

The precision of the estimate will increase as each subsequent measurement, y_t is observed. Next, the state particles are advanced according to their corresponding state covariance noise matrix particle, $p(\theta_t^{(i)}|\theta_{t-1}^{(i)}, W^{(i)})$ for $i = 1, \dots, M$. The importance weights are calculated using the conditional likelihood of the observation given each state particle and observation variance, $w_t^{(i)} = p(y_t|\theta_{t-1}^{(i)}, V^{(i)})$ for $i = 1, \dots, M$. Finally the particles representing the state, parameter and sufficient statistics are resampled with probability proportional to the importance weights. Algorithm 12 summarises the Storvik filter. Figure 4.10 shows the estimated observation and system noise variance for the first order DLM model using the Storvik filter with 500 particles. The prior distributions are the same as in the Liu and West filter specified in equation (4.26).

There are other simultaneous online state and parameter learning algorithms such as Particle Learning [Carvalho et al., 2010b]. [Vieira and Wilkinson, 2016] surveys and evaluates competing simultaneous online state and parameter estimation algorithms.

4.7 Continuous Time DLMs

When modelling real time streaming sensor data there are often transmission delays caused by network outages or localised power outages. In this case, observations can arrive irregularly. It is useful then to be able to transition to modelling in continuous time. In order to apply the DLM to an irregularly observed time series the latent-state transition function can be modelled using a stochastic differential equation (SDE) (see Oksendal [2013] or Iacus [2009] for detailed treatments of stochastic differential equations) representing a random walk in continuous time

$$\mathrm{d}\theta(t) = \sqrt{W}\mathrm{d}B(t), \tag{4.27}$$

To emphasise the continuous time nature of this process (and others in this thesis) a random variable at time t is written as $X(t)$. Equation (4.27) is shorthand for writing

$$\theta(t + \delta t) - \theta(t) = \int_{s=t}^{s=t+\delta t} W^{\frac{1}{2}} dB(s) \tag{4.28}$$

The integral is known as an Itô integral and produces a stochastic result (i.e a distribution), $B(t)$ is known as Brownian motion (sometimes referred to as Wiener process). The properties of the Brownian motion are

1. $B(0) = 0$
2. $B(t) - B(s) = \mathcal{N}(0, t - s)$ for all $t > s$
3. The increments of non-consecutive intervals are independent, ie. $B(t) - B(s)$ is independent of $B(u) - B(v)$ for all $t > s \geq u > v$

From the first and second properties, $B(t) \sim \mathcal{N}(0, t)$. Then the integral in equation (4.28) can be solved

$$\begin{aligned}
\int_{s=t}^{s=t+\delta t} W^{\frac{1}{2}} dB(s) &= W^{\frac{1}{2}} \int_{s=t}^{s=t+\delta t} dB(s) \\
&= W^{\frac{1}{2}} \left(B_{t+\delta_t} - B_t\right) \\
&= \mathcal{N}(0, W(t + \delta_t - t)).
\end{aligned}$$

It follows that exact transition kernel for this process is Gaussian $p(\theta(t_i)|\theta(t_{i-1}), \sigma) = \mathcal{N}(\theta(t_{i-1}), W\delta t_i)$ where $\delta t_i = t_i - t_{i-1}$. This transition kernel can be used instead of the discretised random walk in a standard DLM.

Special care must be taken for a seasonal model as the transition matrix now depends on the time difference between observations. The angle needed in the rotation matrices which make up the system transition matrix is $\omega = \frac{2\pi h(\delta t \% P)}{P}$, where % represents the modulus operation, h is the harmonic and P is the period of the seasonality.

4.7.1 Ornstein-Uhlenbeck Process

The Ornstein-Uhlenbeck (OU) process [Doob, 1942] is the continuous time analogue to the AR(1) process. The OU process can be written as

$$\mathrm{d}\alpha(t) = \phi(\mu - \alpha(t))\mathrm{d}t + \sigma \mathrm{d}B(t), \tag{4.29}$$

where $\phi > 0$ and $\sigma > 0$. Where $B(t)$ is Brownian motion as introduced in section 4.7. The solution to this equation can be calculated using an integrating factor, $e^{\phi t}$

$$\begin{aligned}
\mathrm{d}\alpha(t) + \phi\alpha(t)\mathrm{d}t &= \phi\mu\mathrm{d}t + \sigma\mathrm{d}B(t), \\
e^{\phi t}\mathrm{d}\alpha(t) + e^{\phi t}\phi\alpha(t)\mathrm{d}t &= \phi\mu e^{\phi t}\mathrm{d}t + \sigma e^{\phi t}\mathrm{d}B(t).
\end{aligned}$$

Then applying Itô's product rule which is given by

$$\mathrm{d}X(t)Y(t) = X(t)\mathrm{d}Y(t) + Y(t)\mathrm{d}X(t) + \mathrm{d}X(t)\mathrm{d}Y(t).$$

The equation becomes

$$\begin{aligned}
\mathrm{d}e^{\phi t}\alpha(t) &= \phi\mu e^{\phi t}\mathrm{d}t + e^{\phi t}\sigma\mathrm{d}B(t), \\
\int_{s=0}^{t} \mathrm{d}e^{\phi s}\alpha(s) &= \int_{s=0}^{t} \phi\mu e^{\phi s}\mathrm{d}s + \int_{s=0}^{t} e^{\phi s}\sigma\mathrm{d}B(s), \\
\left[e^{\phi s}\alpha(t)\right]_{s=0}^{t} &= \frac{\phi\mu}{\phi}\left[e^{\phi s}\right]_{s=0}^{t} + \sigma\int_{s=0}^{t} e^{\phi s}\mathrm{d}B(s), \\
e^{\phi t}\alpha(t) - \alpha(0) &= \mu e^{\phi t} - \mu + \sigma\int_{s=0}^{t} e^{\phi s}\mathrm{d}B(s), \\
\alpha(t) &= e^{-\phi t}\alpha(0) + \mu(1 - e^{-\phi t}) + \sigma e^{-\phi t}\int_{s=0}^{t} e^{\phi s}\mathrm{d}B(s)
\end{aligned}$$

since $\mathrm{d}t\phi e^{\phi t}\mathrm{d}\alpha(t) = 0$. The integral of the deterministic function with respect to Brownian motion is a Gaussian distribution and hence entirely specified by its mean and variance.

The mean and variance can be calculated as

$$
\begin{aligned}
\mathbb{E}(\alpha(t)) &= e^{-\phi t}\alpha(0) + \mu(1 - e^{-\phi t}),\\
\mathrm{Var}(\alpha(t)) &= \mathbb{E}\left[(\alpha(t) - \mathbb{E}(\alpha(t)))^2\right]\\
&= \mathbb{E}\left[\left(\sigma e^{-\phi t}\int_{s=0}^{t} e^{\phi s}\mathrm{d}B(s)\right)^2\right]\\
&= \mathbb{E}\left(\sigma^2 e^{-2\phi t}\int_{s=0}^{t} e^{2\phi s}\mathrm{d}s\right)\\
&= \frac{\sigma^2}{2\phi}\left[e^{2\phi(s-t)}\right]_{s=0}^{t}\\
&= \frac{\sigma^2}{2\phi}(1 - e^{-2\phi t})
\end{aligned}
$$

The equation for the variance follows from Itô isometry. The transition kernel of the OU process is

$$
p(\alpha(t_i)|\alpha(t_{i-1}), \phi, \mu, \sigma) = \mathcal{N}\left(\alpha(t_i)|\mu + e^{-\phi\delta t_i}(\alpha(t_{i-1}) - \mu), \frac{\sigma^2}{2\phi}(1 - e^{-2\phi\delta t_i})\right) \tag{4.30}
$$

where $\alpha(t_i)$ is the realisation of α at time t_i and $\delta t_i = t_i - t_{i-1}$ represent the time difference between realisations. Given the equation of the transition kernel, a DLM with an OU latent-state can be written as:

$$
\begin{aligned}
Y(t_i) &= F(t_i)\alpha(t_i) + v(t_i), & v(t_i) &\sim \mathcal{N}(0, V),\\
\alpha(t_i) &= \mu + e^{-\phi\delta t_i}(\alpha(t_{i-1}) - \mu) + \sigma(t_i), & \sigma(t_i) &\sim \mathcal{N}\left(0, \frac{\sigma^2}{2\phi}(1 - e^{-2\phi\delta t_i})\right)
\end{aligned}
$$

Since the solution of the OU process is Gaussian, a modified Kalman filter can be used to calculate the mean and variance of the latent-state. The necessary changes are in the a-priori state estimate at time t_i given the mean and variance of the state at time t_{i-1}, $m(t_{i-1})$ and $C(t_{i-1})$):

$$
\begin{aligned}
a(t_i) &= \mu + e^{-\phi(t_i - t_{i-1})}(\alpha(t_{i-1}) - \mu)\\
R(t_i) &= e^{-2\phi(t_i - t_{i-1})}C(t_{i-1}) + \frac{\sigma^2}{2\phi}(1 - e^{-2\phi(t_i - t_{i-1})})
\end{aligned}
$$

Time	Observation	State
0.00	0.00	1.11
0.48	0.92	1.15
1.19	0.51	0.97
3.04	0.25	0.55
3.09	0.45	0.49

Table 4.2: The first five simulated values of the observation and state from the OU DLM showing that there are irregular time differences between observations.

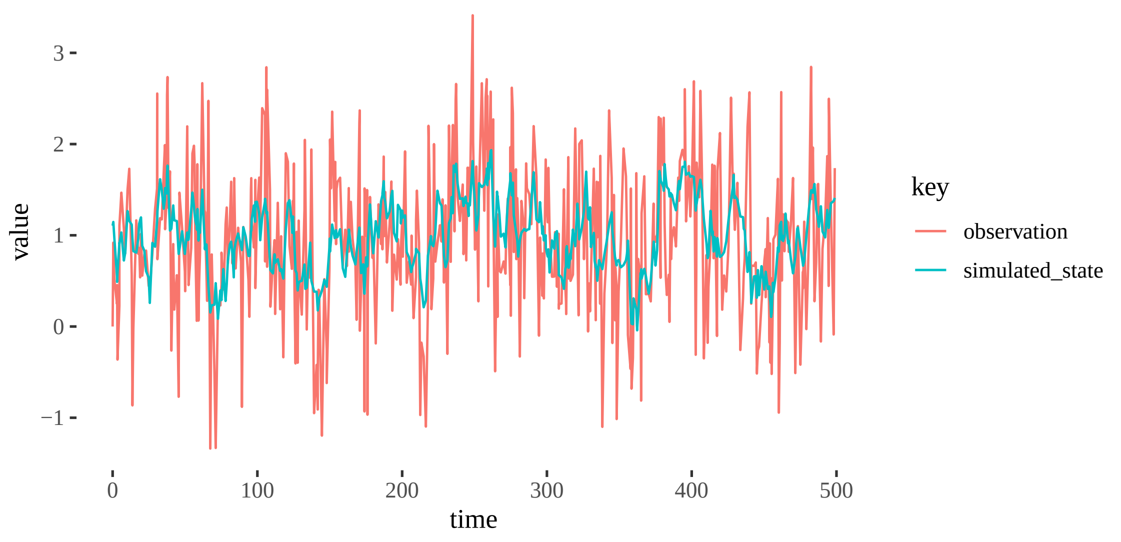
Figure 4.11 shows a simulation from the first order DLM with OU latent-state and the result of filtering the latent-state using the Kalman filter. The values are observed at irregular times, the first five observations are in table 4.2.

A Metropolis-within-Gibbs (see sections 1.3.2 and 1.3.3) scheme is developed to learn the parameters in DLM with the latent OU process. An unrestricted symmetric random walk proposal is used for the mean parameter μ, the proposal distribution is, $q(\mu^\star|\mu) = \mathcal{N}(\mu^\star|\mu, \delta)$, a Normal distribution centred at the previous value of the parameter. For σ and ϕ, the strictly positive parameters are proposed on the log-scale using a symmetric normal distribution. The final parameter of the OU DLM is the observation variance which is sampled using a Gibbs-step. This requires using the FFBS algorithm for the OU process. The backward sampling algorithm requires a change for the irregularly observed OU process. Given the filtered state and initial state $m(t_{0:N}), C(t_{0:N})$, the backward sampler first samples a value from the final state $\alpha(t_N) \sim \mathcal{N}(m(t_N), C(t_N))$ then proceeds backward through the filtered state:

$$\begin{aligned}
s(t_i) &= m(t_i) + \frac{C(t_i)\exp(-\phi \mathrm{d}t_i)}{R(t_{i+1})}(\alpha(t_{i+1}) - a(t_{i+1})), \\
S(t_i) &= C(t_i) - \frac{C(t_i)^2\exp(-2\phi \mathrm{d}t_i)}{R(t_{i+1})}.
\end{aligned}$$

Then $\alpha(t_i)$ is sampled from $\mathcal{N}(s(t_i), S(t_i))$. The latent-state likelihood for the irregularly observed OU DLM observed at times $t = t_0, \dots, t_N$ can be written as

$$\begin{aligned}
p(\alpha(t_{0:N})|\psi) &= p(\alpha(t_0))\prod_{i=1}^{N} p(\alpha(t_i)|\alpha(t_{i-1}), \psi) \\
&= \mathcal{N}\left(\mu, \frac{\sigma^2}{2\phi}\right)\prod_{i=1}^{N}\mathcal{N}\left(\mu + e^{-\phi(t_i - t_{i-1})}(\alpha(t_{i-1}) - \mu), \frac{\sigma^2}{2\phi}\left(1 - e^{-2\phi(t_i - t_{i-1})}\right)\right) \qquad (4.31)
\end{aligned}$$



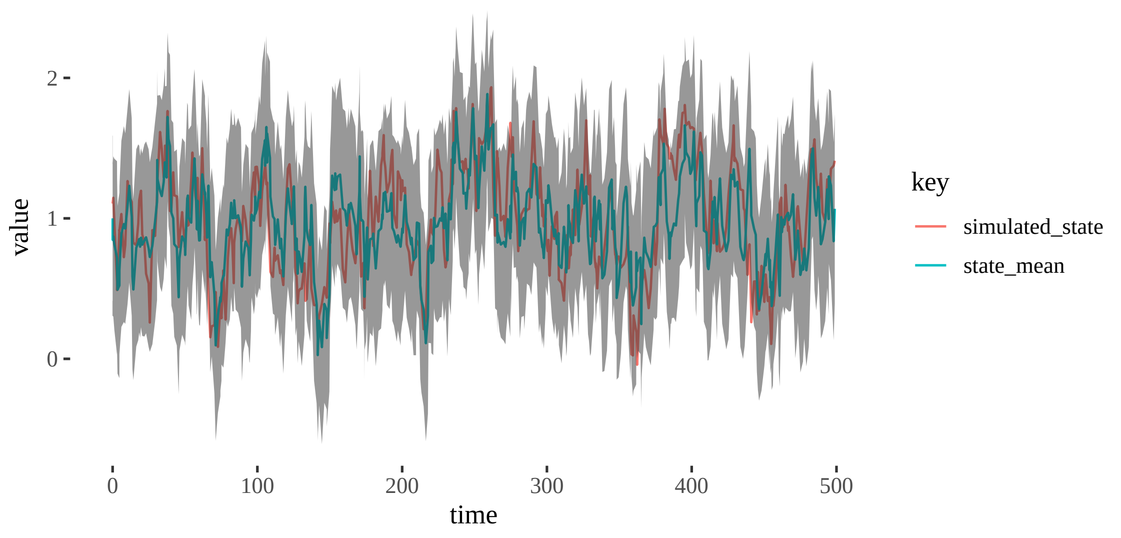


Figure 4.11: Simulated values from a first order DLM with an Ornstein-Uhlenbeck latent state with parameter values $\phi = 0.1, \mu = 1.0, \sigma = 0.2, V = 0.5$, (top). Mean and 95% probability intervals of the filtered latent-state computed using the particle filter, (bottom).

The sampled state can be used to sample a new value of the observation variance using a Gibbs step as outlined in section 4.4. The prior distributions for the static parameters are chosen to be

$$
\begin{aligned}
p(\phi) &= \mathcal{B}(2,5), \\
p(\mu) &= \mathcal{N}(1,1), \\
p(\sigma) &= \text{Inv-Gamma}(5,1), \\
p(V) &= \text{Inv-Gamma}(2,2).
\end{aligned}
$$

The proposal distribution for the OU parameters is a multivariate Normal distribution with mean, $\bar{\psi} = (\log \phi, \mu, \log \sigma)$ with diagonal covariance matrix $\Delta = (0.05, 0.05, 0.05)$. Two chains were run for 100,000 iterations with the first 10,000 iterations discarded then the chain was thinned by a factor of 20. The Metropolis-within-Gibbs algorithm is summarised in algorithm 13.

Algorithm 13: A Metropolis-within-Gibbs algorithm for the OU DLM

```
Result: Return $M$ correlated samples from the posterior distribution
        $p(\psi, V|y(t_{0:N}))$, where $\psi = \{\phi, \mu, \sigma\}$
1  Initialise the values of the static parameters by sampling from the prior
     distribution;
2  for $i$ in 1:M do
3      Perform FFBS to sample the latent state conditional on $\psi^{(i-1)}, V^{(i-1)}$;
4      Sample $V^{(i)}$ using a Gibbs step from equation (4.17);
5      for $\tilde{\psi}$ in $\psi^{(i-1)}$ do
6          Calculate the likelihood, $p(\alpha(t_{0:N})|\psi^{(i-1)})$ using equation (4.31);
7          Propose a new value of the parameter $\tilde{\psi}^\star \sim q(\tilde{\psi}^\star|\tilde{\psi})$;
8          Denote $\psi^\star$ as the parameter vector, $\psi^{(i-1)}$, updated to include the
             proposed value $\tilde{\psi}^\star$;
9          Calculate the likelihood, $p(\alpha(t_{0:N})|\psi^\star)$ using equation (4.31);
10         Calculate $A = \frac{p(\tilde{\psi}^\star)p(\alpha(t_{0:N})|\psi^\star)q(\tilde{\psi}|\tilde{\psi}^\star)}{p(\tilde{\psi})p(\alpha(t_{0:N})|\psi^{(i-1)})q(\tilde{\psi}^\star|\tilde{\psi})}$;
11         Sample $u \sim U[0,1]$;
12         if $u < A(\psi^\star|\psi)$ then
13             Set $\psi^{(i)} = \psi^\star$;
14         else
15             Set $\psi^{(i)} = \psi^{(i-1)}$;
16     end
17 end
```

Figure 4.12 shows the diagnostic plots for the static parameters.

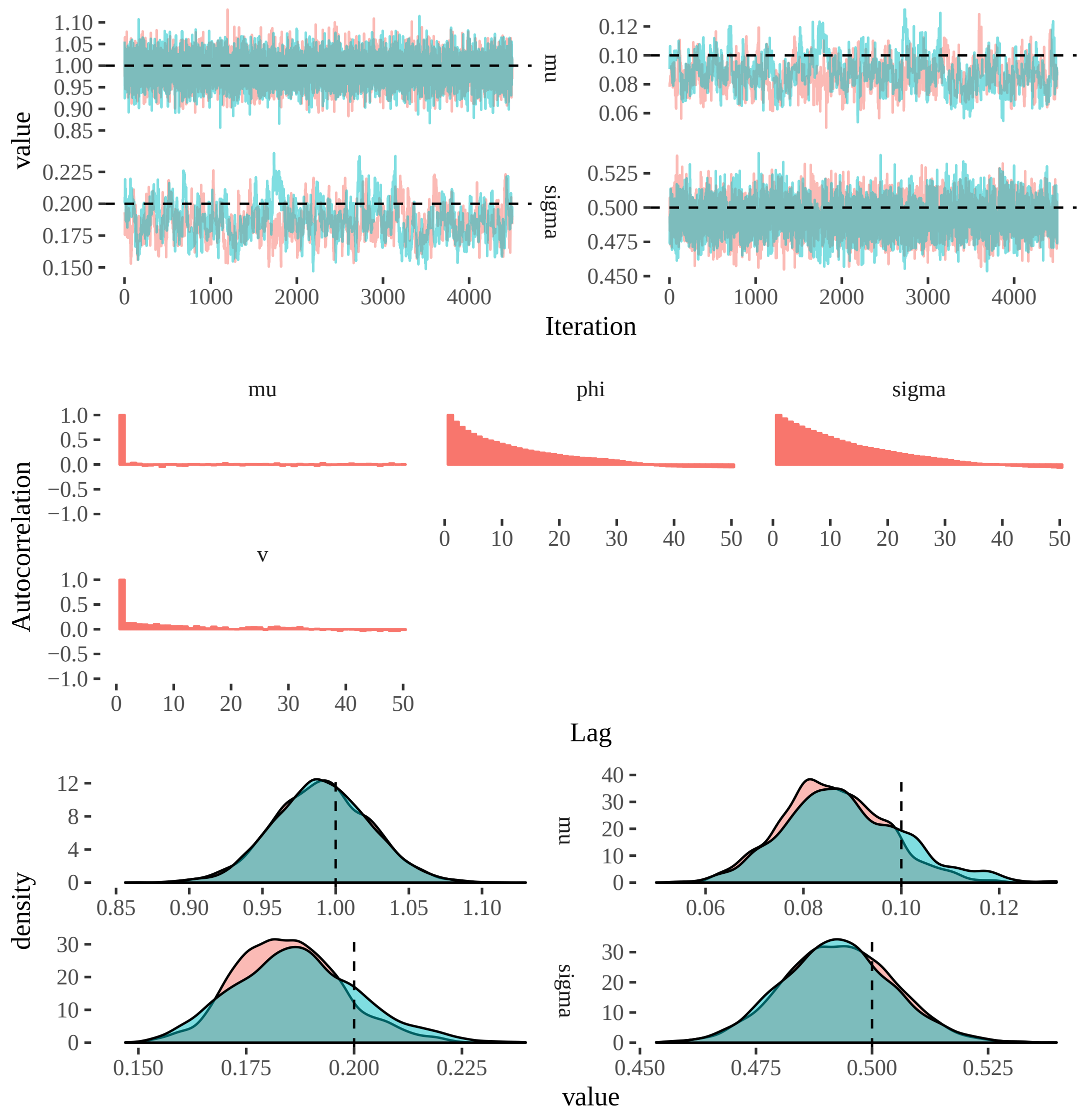


Figure 4.12: Diagnostic plots of the draws from the parameter posterior distribution of the OU DLM using MCMC. The traceplots of the MCMC draws, (top). The Autocorrelation for chain 1 (middle) The empirical marginal posterior densities, (bottom)

4.8 Modelling Language

The Gaussian state-space models and associated inference algorithms presented in this chapter have been implemented in an open source Scala library which can be found on the Sonatype central repository and developed on Github[2]. The API for model building is straightforward and many common models are already implemented. The DLM is implemented as a product type with time varying observation and system matrices, F_t

[2]`https://git.io/dlm`

and G_t respectively, represented as functions from time to a matrix:

```
case class Dlm(
  f: Time => DenseMatrix[Double],
  g: Time => DenseMatrix[Double]
)
```

The companion object `Dlm` contains functions to build several common models, such as polynomial and seasonal models introduced in section 4.1:

```
val linear = Dlm.polynomial(1)
val seasonal = Dlm.seasonal(period = 24, harmonics = 3)
```

The seasonal model has a period of 24, representing the number of hours in a day. This is suitable if the time series has a daily periodicity and measured hourly. These models can be composed using a binary operator, for instance a seasonal model with a linear trend:

```
val seasonalTrend = linear |+| seasonal
```

Multivariate models (introduced in section 4.1.2) can be specified by composing multiple univariate DLMs:

```
val multivariateModel = List.fill(6)(seasonalTrend).reduce(_ |*| _)
```

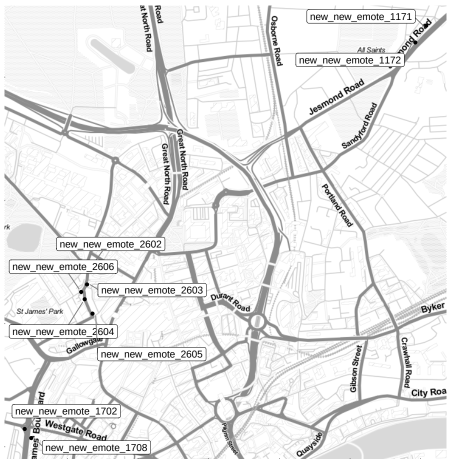
This multivariate model is suitable for modelling 6 time series with the same seasonal structure.

The library has support for missing and partially missing data as well as irregularly observed data. All the algorithms described in this chapter are implemented in the library and can be used for any model specification.

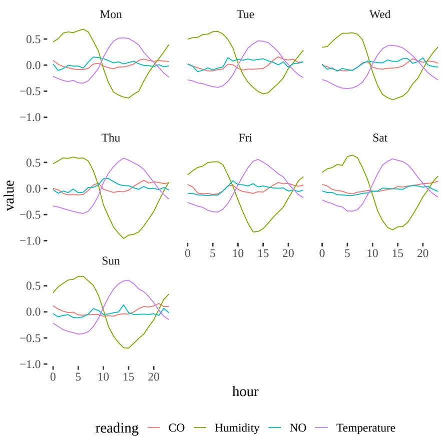
4.9 Modelling Multivariate Environment Sensors

Air pollution in urban areas is a major problem, it has been shown that urban air pollution contributes to childhood respiratory problems [Calderon-Garciduenas et al., 2003], is related to an increase in asthma and other allergic conditions [Cacciola et al., 2002] and affects cariopulmonary health [Pope III and Dockery, 2006]. Prevalence of air quality related diseases will increase if nothing is done to effectively monitor and decrease airborne pollution. More people are choosing to live in heavily populated urban areas, (in 2016 54% of the worlds population lived in urban areas [Bank, 2014]), and hence are at a greater risk of developing the diseases associated with air pollution. Anthropogenic sources of pollution include industrial processes, manufacturing and traffic related chemicals such as nitrogen dioxide NO_2, carbon monoxide CO and patriculate matter PM.

In order to tackle the growing problem of air-pollution, pollution levels and its effects



(a)



(b)

Figure 4.13: (a) The location of ten Urban Observatory sensors measuring environmental variables. (b) Normalised average value of the environmental processes across all sensors by hour and weekday showing the seasonal patterns of each process.

must be monitored closely and reliably. A growing trend in smart cities is to record and monitor air pollution at a local level with the use of remote and possibly wireless sensors. One such smart city is Newcastle upon Tyne and its large open sensor network the Urban Observatory [James et al., 2014a]. A class of environment sensors are positioned in and around the city measuring NO_x, CO and temperature among other weather and environmental processes. These sensors are placed in convenient locations, for example on lampposts and not necessarily in positions where researchers would like to record the value of the environmental processes.

Sensors deployed in the urban environment are unreliable. Network outages can cause missed observations and sensor placement can affect the quality and reliability of the collected data. Without reliable data, it is difficult to perform experiments analysing this data. Statistical modelling can ameliorate these problems, by interpolating, smoothing and forecasting measured data and providing probability intervals for forecasts and interpolated data.

Ten environment sensors are chosen, the locations of the sensors are plotted in figure 4.13 (a).

Figure 4.13 (b) shows the average of the normalised values of each of the environmental processes by hour and week day. This reveals the seasonal structure of each of the processes. The model is chosen to be a multivariate DLM with drift and seasonal components. The number of harmonics is chosen to be three for each seasonal component with the period $T = 24$, $\omega = \frac{2\pi}{T}$. The system and observation matrices are

$$F = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ \vdots \\ 1 \\ 0 \end{pmatrix}, \quad G = \begin{pmatrix} 1 & 0 & \dots & \dots & 0 \\ 0 & R(\omega, 1) & 0 & \dots & 0 \\ \vdots & 0 & R(\omega, 2) & \dots & 0 \\ \vdots & \dots & \dots & \ddots & 0 \\ 0 & \dots & \dots & 0 & R(\omega, 3) \end{pmatrix}$$

Each model is then combined into a multivariate model using block-diagonal concatenation as explained in section 4.1.2. The ten sensors are fit independently and model fit is assessed using the residuals of the one-step forecasts.

Parameter inference is performed using Gibbs sampling and the observation and system noise covariance matrices are assumed to be diagonal. Hence, the prior distributions for the diagonal elements of the covariance matrices are Inverse Gamma:

$$\begin{aligned} V_{ii} &\sim \text{Inv-Gamma}(2.0, 3.0), i = 1, \dots, 4, \\ W_{jj} &\sim \text{Inv-Gamma}(2.0, 3.0), j = 1, \dots, 7. \end{aligned}$$

The raw sensor data consists of irregularly spaced measurements approximately every minute. In order to simplify inference and reduce the computational cost the data is thinned; the majority of the measurements are discarded and the measurement closest to each hour is retained.

10,000 iterations of the Gibbs sampler are used with 1,000 discarded as burn-in and every other iteration kept to get approximately independent samples from the parameter posterior distributions. Figure 4.14 shows the parameter diagnostics for the observation variances corresponding to CO, humidity, NO and temperature respectively.

The observation variance of NO (V3) is quite large, NO is measured in parts per million and sometimes reaches values in excess of 600. However, the majority of the time, the observations are in the 0-200 range. The Student's-t distribution could be a better fit for this type of data with a large amount of outliers, this is explored in section 5.7.

Figure 4.15 shows the diagnostics for the diagonal elements of the system matrix relating to the NO process.

One step forecasts for each of the ten sensors are calculated by taking the mean of the parameter posterior distribution from the MCMC output and performing the Kalman filter. The results of the one-step forecasting for sensor 2603 are plotted in Figure 4.16 (a).

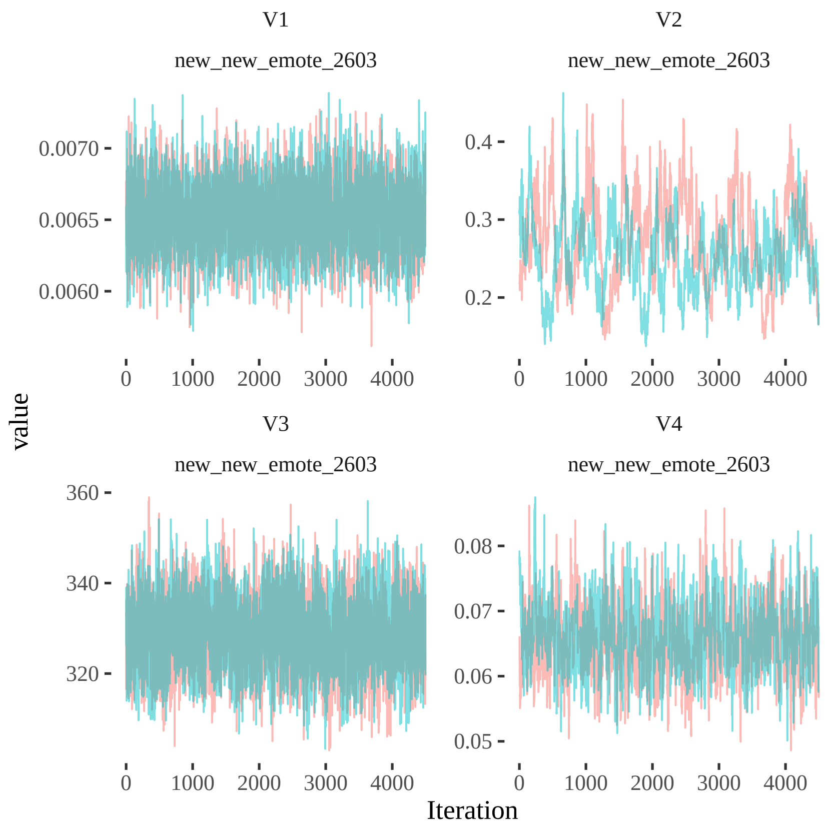


Figure 4.14: The traceplots for the diagonal elements of the observation matrix for sensor 2603.

The one-step forecasts are fairly accurate for CO, Temperature and Humidity, however the large observation variance for NO has resulted in large uncertainty. Normal qq-plots of the residuals for the one-step forecasts at sensor 2603 are plotted in Figure 4.16 (b).

It is thought that a log transformation of the NO variable could improve the fit and reduce the uncertainty of the one-step forecast intervals. Figure 4.17 (a) shows the forecast using a log-transform of the NO variable, showing reduced uncertainty for the one-step forecasts. Figure 4.17 (b) shows qq-plots for the residuals using the log-transformed data, the NO residuals appear to be more normal and exhibit a good fit.

4.10 Summary

The time series models and inference methods presented in this chapter apply to Normally distributed time series data. Many time series can be transformed or summarised (due to the central limit theorem) in order to appear Normal. However, parameter posterior distributions can be difficult to interpret for transformed time series. Sometimes data

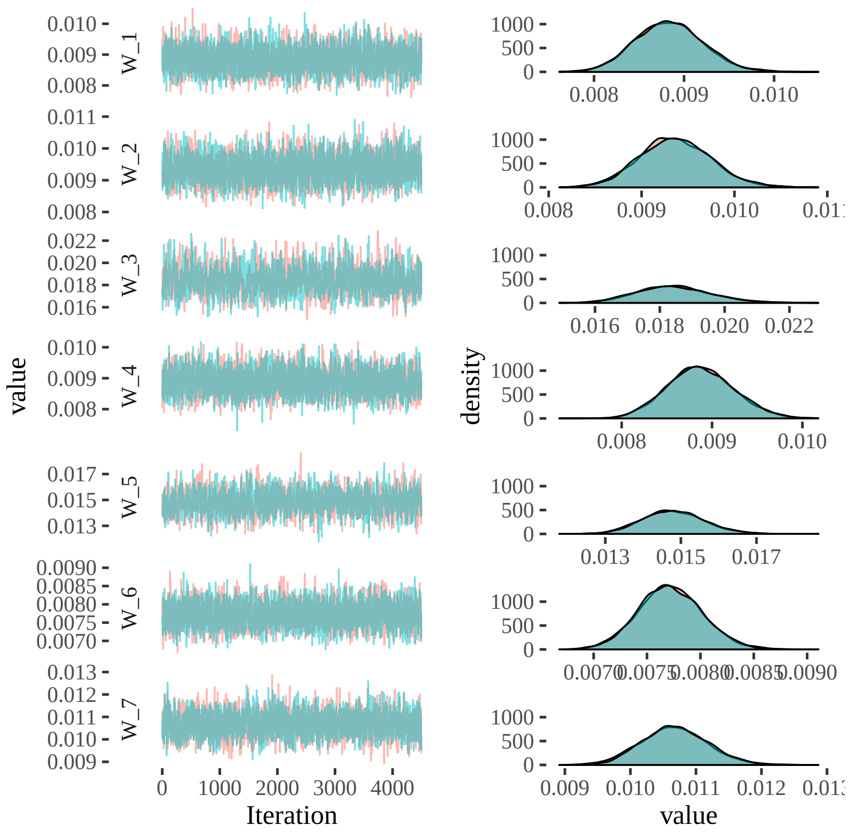


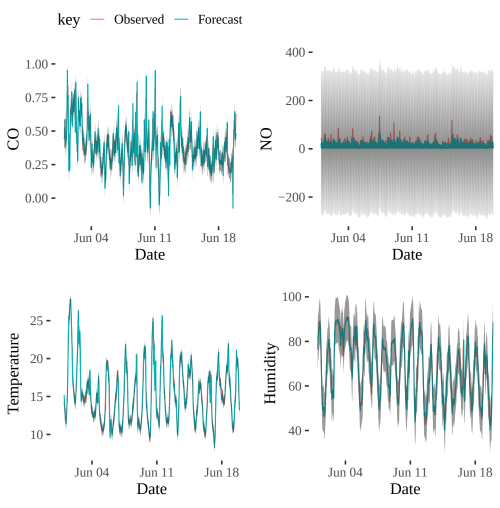
Figure 4.15: Posterior MCMC diagnostics for the diagonal elements of the system matrix corresponding to the CO observations for sensor 2603. The traceplots of the MCMC draws, (left). The empirical marginal posterior densities, (right).

can not be transformed to appear Normal, count valued time series are one such example where it is challenging to transform the data appropriately. Chapter 5 considers a class of non-Gaussian time series for modelling count data and data with outliers using the robust Students-t distribution. This leads to improved model fit for the NO and temperature data seen in example 4.9.

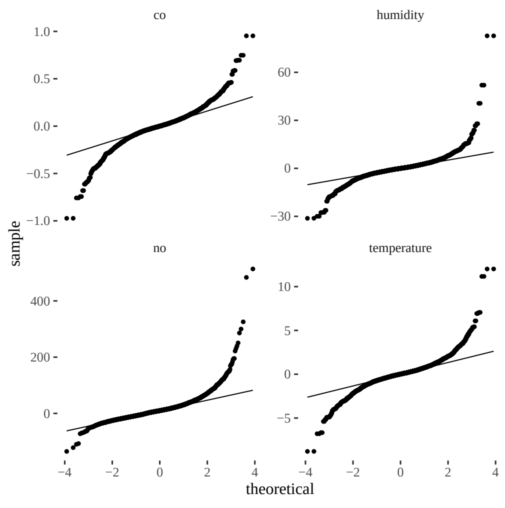
Section 4.6 considered several algorithms for online parameter estimation in dynamic linear models. These online algorithms can be used to simultaneously perform inference for the latent-state and unknown static parameters. Section 4.7 showed a straightforward way to model DLMs in continuous time for irregularly observed data. This is useful for sensor data which can go offline for periods due to sensor problems or network connectivity issues.

Section 4.8 introduced the modelling language for building DLMs using the Scala library Bayesian DLMs[3]. The models can be used with any of the inference algorithms

[3]https://git.io/dlm

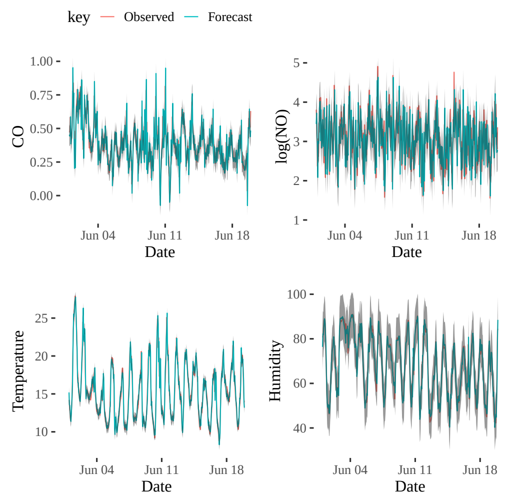


(a)

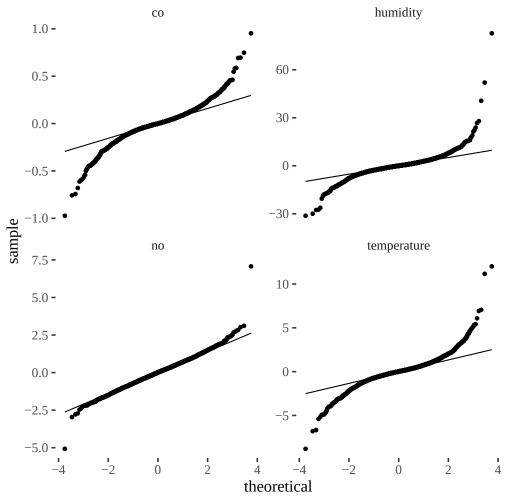


(b)

Figure 4.16: (a) The one-step forecast mean and 50% probability intervals for sensor 2603 during the first three weeks of June 2018. The NO forecast uncertainty is unacceptably large. (b) The residuals of the one-step forecast mean for sensor 2603 for the month of June 2018.



(a)



(b)

Figure 4.17: (a) One-step forecasts for sensor new new emote 2603 during June 2018, using the logarithm of NO. (b) Normal Q-Q plot for the residuals of the multivariate Urban Observatory DLM using a log-transform on the NO measurements.

introduced in this chapter for any user defined DLM and data. The filtering algorithms are written using binary reduction functions and applied to collections of observations using `fold` and `scan`. This implementation allows the same function to be re-used in multiple contexts - such as an Akka stream for filtering live streaming data (see section 2.3.1) or a built in collection such as `Vector` when using FFBS in offline inference algorithms.

Chapter 5

Non Gaussian State Space Models

Time series observations arising in environmental science often are not distributed according to a Gaussian distribution. If the data cannot be suitably transformed then it can not be adequately modelled with the DLMs introduced in chapter 4. Non-Gaussian time series include those which have a high proportion of outliers, measurements of proportions or count time series. More suitable observation distributions for these time series could include the Student's t-distribution, the Beta distribution and the Poisson distribution respectively. The DLM considered can be extended to have a non-Gaussian observation distribution. This results in a more general class of models which can be used to model non-transformed data. The downside to this generalisation is that filtering is not available in closed form using the Kalman filter.

The class of non-Gaussian models with observations distributions which are from the exponential family are referred to as dynamic generalised linear models (DGLMs) [West et al., 1985]. West et al. proposed using conjugate prior distributions for the exponential family observation distributions leading to approximate closed form predictive distributions. The class of DGLMs is a dynamic extension of generalised linear models [Nelder and Wedderburn, 1972] in which independent observations are considered to follow a exponential family model and are related to covariates which do not evolve in time. Additionally, there are useful observation distributions which are not included in the exponential family for which filters can be developed such as for the Student's t-distribution in section 5.4 and the zero-inflated Poisson distribution 5.1.1. This chapter is focused upon simulation based inference using particle filtering and pseudo-marginal methods [Andrieu et al., 2010] for state space models with Gaussian latent-state and non-Gaussian observation distributions. Forecasting is performed using simulation as described in section 5.5. Non-Gaussian state space models have been used in a variety of fields, such as finance and economic time series [Triantafyllopoulos, 2008].

Non-Gaussian state space models are useful for modelling heterogeneous data from distributed sensor-networks with a high data velocity. Just as the Kalman filter can be performed online to determine the filtering distribution in a DLM, the particle filter can be performed online to learn the posterior distribution of the latent-state in a non-Gaussian state space model. The only requirement for the observation distribution is that the probability density function can be evaluated. Unfortunately, particle filtering is more computationally intensive than the Kalman filter and this motivates a new conditional Kalman filter for Student's t-distributed observations in section 5.4. The computational considerations needed to implement an efficient particle filter are discussed in section 5.2.1. This section presents a novel parallel implementation of the bootstrap particle filter using functional programming idioms. This can be used to speed up advancing the state and resampling steps when using the particle filter with large particle clouds.

In addition to the flexibility of choosing a non-Gaussian observation distribution, these models can be easily composed like DLMs to create complex univariate and multivariate models as described in sections 4.1.1 and 4.1.2. The models are implemented in a Scala library published by the author [1].

A non-Gaussian state space model can be written hierarchically as follows:

$$\begin{aligned}
Y_t|\eta_t &\sim p(y_t|\eta_t, \psi), \\
\eta_t|\theta_t &= g(F_t^T\theta_t), \\
\theta_t|\theta_{t-1} &= G_t\theta_t + w_t, \qquad w_t \sim \text{MVN}_n(0, W_t), \\
\theta_0 &\sim \mathcal{N}(m_0, C_0).
\end{aligned} \tag{5.1}$$

The observations Y_t are univariate random variables observed at discrete time points. As in the DLM, the latent-state, $\theta_t \in \mathbb{R}^n$, $t = 0, \dots, N$, is Gaussian and the state evolution is a first order Markov process parameterised by the system evolution matrix $G_t \in \mathbb{R}^{n\times n}$ and the covariance matrix $W_t \in \mathbb{R}^{n\times n}$. The latent-state is transformed using the observation matrix $F_t \in \mathbb{R}^n$ and the link function $g(\cdot)$. η_t is the mean or location of the observation distribution and ψ are additional static parameters required for the chosen observation distribution. Figure 5.1 shows the structure of the non-Gaussian state-space model. The joint posterior distribution of the state, $\theta_{0:N}$ and parameters W and ψ can be learned from the data using particle filtering and psuedo-marginal MCMC schemes [Andrieu et al., 2010]. This chapter introduces useful inference algorithms for non-linear, non-Gaussian state space models illustrated with examples using both simulated and real-world data.

[1] `https://git.io/dlm`

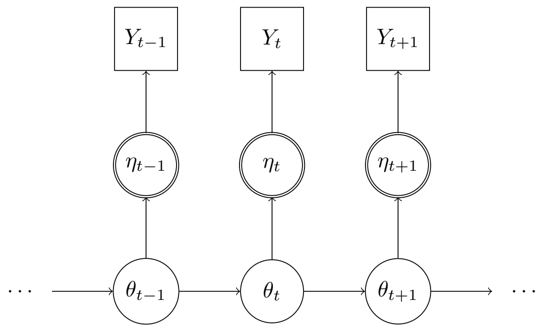


Figure 5.1: Directed a-cyclic graph showing the structure of the non-Gaussian state space model. The latent-state is Markovian with the horizontal arrows representing state transitions. The vertical arrows to η_t represent the (deterministic) transformation of the latent-state. The vertical arrows to Y_t represent a draw from the observation distribution, the observations are conditionally independent given the corresponding value of the latent-state.

5.1 Example Non-Gaussian Models

5.1.1 Count Data

Count data with low-counts or high prevalence of zero readings presents difficulties for models with Gaussian observation distributions. The simplest distribution used to model count data is the Poisson distribution. The Poisson distribution has one parameter, the rate $\lambda > 0$, which is equal to the mean and variance the Poisson distribution has support on the non-negative integers, $Y_t \in \{0, 1, \dots\}$. The link function is $g(x) = \exp(x)$. The latent-state is as in equation 5.1, with the observation equations

$$\begin{aligned}
Y_t|\eta_t &\sim \text{Poisson}(\eta_t), \\
\eta_t|\theta_t &= \exp(F_t^T\theta_t).
\end{aligned}$$

Sometimes in count data there are many zeros, for instance a traffic sensor on a road used for commuting to a business park may have no cars passing on weekends or overnight. The Poisson distribution alone is not a good fit for this type of data. However a mixture distribution can be used:

$$p(Y_t|\eta_t, p) = \begin{cases} p + (1-p)e^{-\eta_t} & \text{if } Y_t = 0, \\ (1-p)\frac{\eta_t^{Y_t}}{Y_t!}e^{-\eta_t} & \text{if } Y_t > 0. \end{cases}$$

This states that the probability of observing a zero at time t is p plus the probability of observing a zero from the Poisson distribution. This mixture distribution is commonly known as a zero-inflated Poisson distribution. To show that this is a probability mass function, it must sum to one for all possible outcomes, $Y \in \{0, 1, 2, \dots\}$:

$$\begin{aligned}
\sum_{Y \in \mathbb{Z}^+} Pr(Y|\eta, p) &= p + (1-p)e^{-\eta} + (1-p)\sum_{Y=1}^{\infty} \frac{\eta^Y}{Y!}e^{-\eta} \\
&= p + (1-p)\sum_{Y=0}^{\infty} \frac{\eta^Y}{Y!}e^{-\eta} \\
&= p + (1-p)\cdot 1 \\
&= 1.
\end{aligned}$$

This observation distribution results in a non-Gaussian state space model which is not in the exponential family - nevertheless the joint posterior distribution of the latent-state and static parameters can be inferred using the pseudo-marginal Metropolis-Hastings inference algorithm introduced in section 5.3.

Count data is often over-dispersed with respect to a Poisson distribution, meaning that as the sample mean of the count increases so does the variability. An alternative parameterisation of the Negative Binomial distribution is often used to model over-dispersed count data. The Negative Binomial arises as a Gamma mixture of Poisson distributions where the rate of the Poisson distribution is distributed according to a Gamma distribution with shape $\alpha = r$ and rate $\beta = \frac{r}{m}$, then

$$\begin{aligned}
p(y|r, m) &= \int_0^\infty \text{Gamma}\left(\lambda | r, \frac{r}{m}\right) \text{Poisson}(\lambda)\text{d}\lambda, \\
&= \frac{\left(\frac{r}{m}\right)^r}{\Gamma(r)y!}\int_0^\infty \lambda^{r-1}\exp\left(-\frac{r}{m}\lambda\right)\frac{\lambda^y \exp(-\lambda)}{y!}\text{d}\lambda, \\
&= \frac{\left(\frac{r}{m}\right)^r}{\Gamma(r)y!}\int_0^\infty \lambda^{r+y-1}\exp\left(-\lambda\left(\frac{m+r}{m}\right)\right)\text{d}\lambda, \\
&= \frac{\Gamma(r+y)}{\Gamma(r)y!}\left(\frac{m}{m+r}\right)^y \left(\frac{r}{r+m}\right)^r
\end{aligned} \tag{5.2}$$

The support of this distribution is $Y_t \in \{0, 1, 2, \dots\}$, the parameters representing the scale and mean are both non-negative quantities $r, m \geq 0$. The variance of this distribution is $\text{Var}(Y_t) = m + \frac{m^2}{r}$ which increases as the value of the mean increases and can be controlled by the parameter r. In order to model a dynamic mean let $m = \eta_t$. To complete the specification of the model the link function is $g(x) = \exp(x)$ and r is a static parameter. Simulating from this distribution is straightforward, by first sampling the rate, λ from a Gamma distribution then sampling from the Poisson distribution with rate λ.

A simulation from these three models for count time series is shown in Figure 5.2. Each of the simulations share the same latent state, θ_t, $t = 1, \dots, 500$ which is then

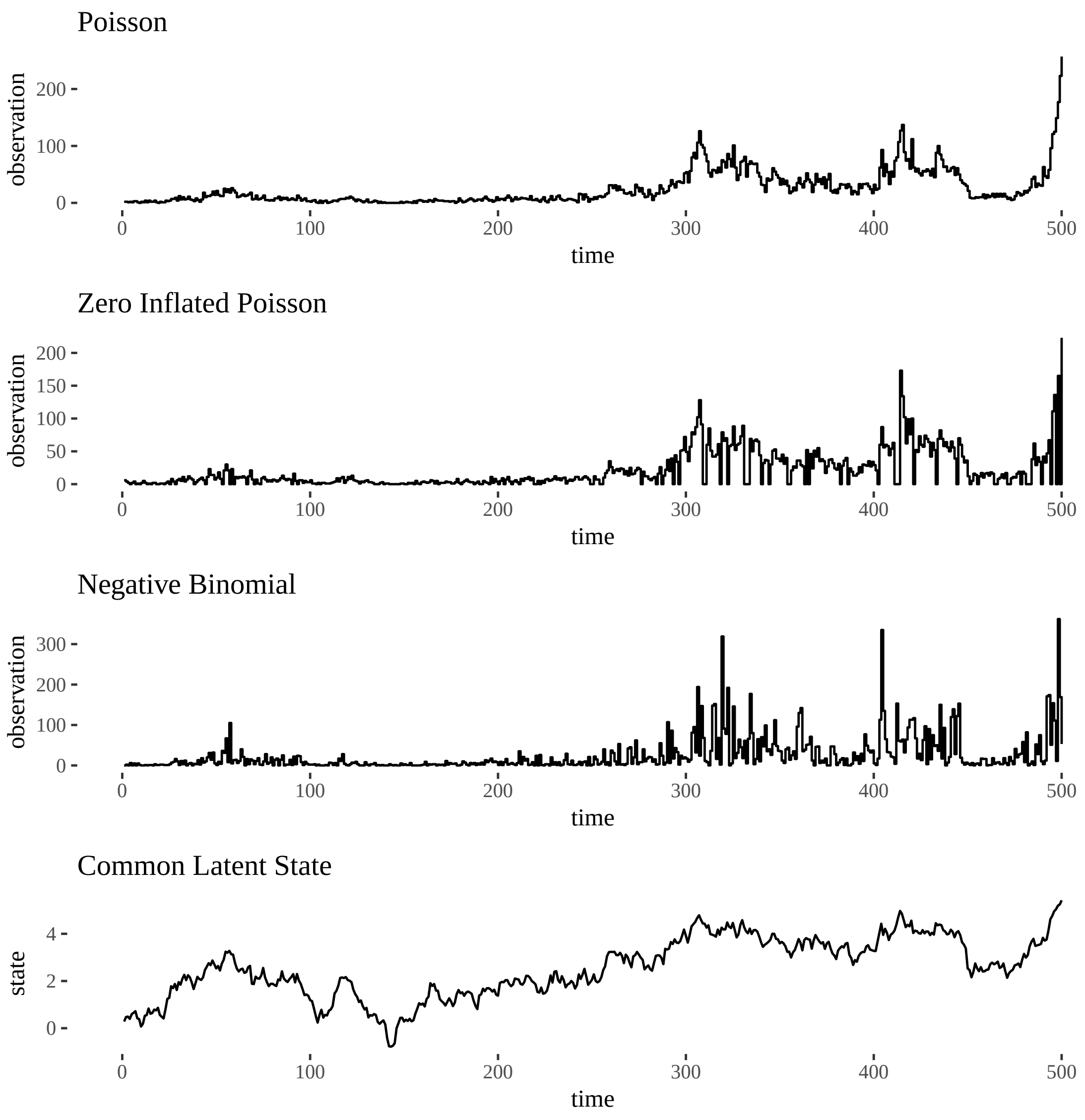


Figure 5.2: Simulated values from first order count time series models with $W = 0.05$. Each model shares the same latent-state with a different observation distribution, the zero inflated Poisson has probability of zero $p = 0.2$ and the negative Binomial distribution has $r = 1.0$.

exponentiated to form the rate of the Poisson and zero-inflated Poisson (ZIP) and the mean of the Negative Binomial. The ZIP clearly has a higher prevalence of zeros than the Poisson observation distribution and both the Negative Binomial and ZIP produce higher variance in the observations.

5.1.2 Robust Time Series Modelling

The Student's t-distribution is a location-scale family distribution which is useful for robust time series modelling. The Student's t-distribution has thicker tails than the normal distribution for low values of the degrees of freedom, hence it is more forgiving of outliers in the measurements. The scaled, shifted Student's t-distribution is parameterised by the location, ℓ, scale s and degrees of freedom ν, the probability density function is:

$$p(x|\ell, s, \nu) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\pi\nu s}\Gamma(\frac{\nu}{2})}\left(\frac{(x-\ell)^2}{\nu s^2}+1\right)^{-\frac{\nu+1}{2}} \tag{5.3}$$

Note that the Student's t-distribution is not a member of the exponential family. The state-space model, with a first order random walk latent-state can be written as:

$$\begin{aligned} Y_t &\sim t_\nu(\theta_t, s^2), \\ \theta_t &= \theta_{t-1} + w_t, \qquad w_t \sim \mathcal{N}(0, W), \\ \theta_0 &\sim \mathcal{N}(m_0, C_0). \end{aligned} \tag{5.4}$$

Figure 5.3 shows 1,000 simulations from the model with $\nu = 3, s^2 = 3, W = 0.1, m_0 = 0, C_0 = 1$. This can be compared with the simulation from the first order polynomial DLM in figure 4.2, the observations in the Student's t state space model display more outliers from the main trend.

5.2 Particle Filtering for State Estimation

The bootstrap particle filter [Gordon et al., 1993] was introduced in section 1.3.5 for a general state space model with a Markov transition kernel which can be forward simulated from and any observation distribution from which the conditional likelihood can be computed. The non-Gaussian state space models considered in this chapter have a Markovian (and Gaussian) transition kernel for the latent state which can be forward simulated from and hence the bootstrap particle filter can be used to find the filtering distribution and an estimate of the log-likelihood. The log-likelihood estimate can be used in pseudo-marginal MCMC schemes to jointly estimate the posterior distribution of the latent-state and static parameters of the models.

Section 5.2.1 considers the numerical stability of the bootstrap particle filter and introduces implementation details which can be used to build a robust particle filter. One strength of the particle filter is that it can be parallelised, section 5.2.2 illustrates a novel implementation of a parallel particle filter with polymorphic functions, using higher kinded types.

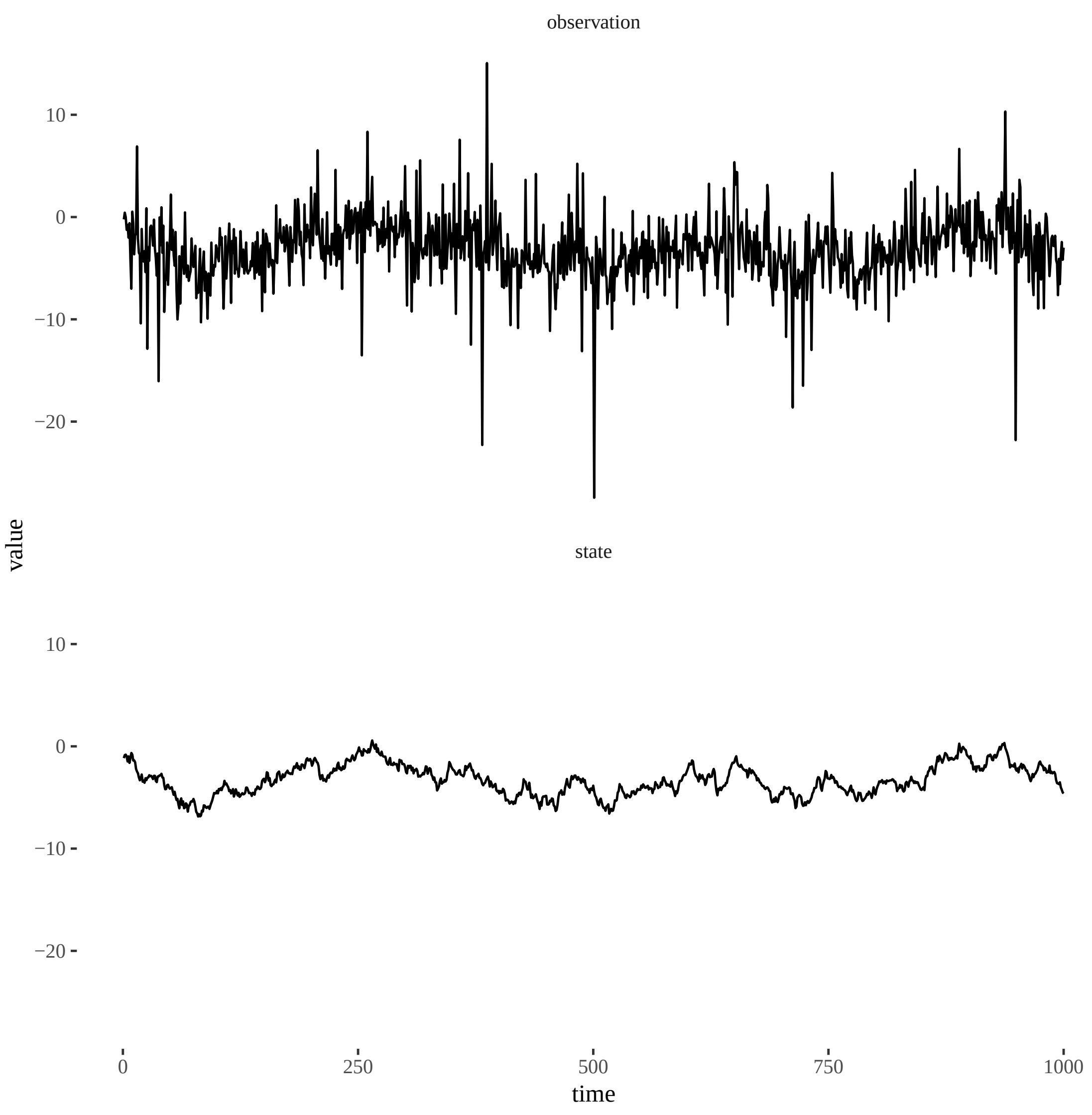


Figure 5.3: Simulated observations, (top), and latent-state, (bottom), from the state space model with Student's-t distributed measurement error with parameter values $\nu = 3, s^2 = 3, W = 0.1, m_0 = 0, C_0 = 1$.

5.2.1 Numerical Stability

The bootstrap particle filter has a problem with sample degeneracy. Intuitively this means that the distribution of weights associated with the particles is unbalanced with a few very-large weights and the others very small. This is caused by many particles advancing to areas of low probability. To combat this problem, a resampling step is employed whereby particles are selected with probability proportional to their weight. This results in a sample with more high-weighted particles and fewer low-weighted particles.

Resampling can lead to another problem, particle impoverishment, whereby particles collapse onto a small number of trajectories. A way of avoiding particle impoverishment is to resample only when the effective sample size of the particle cloud falls below a predefined value. The effective sample size can be calculated by

$$ESS = \frac{1}{\sum_{i=1}^{M} (\tilde{w}_t^{(i)})^2}, \tag{5.5}$$

where $\tilde{w}_t^{(i)}$ represents the normalised weight of particle i. Then resampling occurs when the ESS falls below a threshold value, $M/2$ for instance. This has the dual purpose of speeding up the computation, as resampling is typically an expensive step which often requires cross-processor communication in the case of a parallel particle filter.

Another consideration when implementing the particle filter is the arithmetic precision of the computer. When calculating the weights, it is important to work with the log-weights, u_t to avoid arithmetic underflow. The exponentiated normalised weights, w_t are then needed in order to perform resampling. If values of the log-likelihood are small, then their exponentiated values are approximately zero. This means the weights of each particle will be identical. Since the weights are going to be normalised, take the maximum value of the collection of the weights at time t away from each weight before exponentiating

$$\begin{aligned} a_t &= \max_i u_t^{(i)}, \\ w_t^{(i)} &= \exp(u_t^{(i)} - a_t) \text{ for } i = 1, \dots, M. \end{aligned}$$

where $u_t^{(i)}$ represents the log-weight of particle i at time step t. An estimate of the likelihood can be found by calculating the average of the sum of the un-normalised weights:

$$\begin{aligned} \hat{p}(y_{1:t}) &= \hat{p}(y_1) \prod_{k=2}^{t} \hat{p}(y_k | y_{1:k-1}) \\ &= \prod_{k=1}^{t} \frac{1}{M} \sum_{i=1}^{M} \exp(u_k^{(i)}) \end{aligned} \tag{5.6}$$

The marginal log-likelihood of the observation at time t given the previous observations can then be calculated as

$$\log \hat{p}(y_t | y_{1:t-1}) = a_t + \log \frac{1}{M} \sum_{i=1}^{M} \exp(u_t^{(i)} - a_t) \tag{5.7}$$

using the following identity

$$\log \sum_{i=1}^{M} \exp(u_t^{(i)}) = a_t + \log \frac{1}{M} \sum_{i=1}^{M} \exp(u_t^{(i)} - a_t) \tag{5.8}$$

An estimate of the marginal log-likelihood can be calculated iteratively, using equation (5.8) while performing the particle filter

$$\begin{aligned}\log \hat{p}(y_{1:t}) &= a_t + \log \frac{1}{M} \sum_{i=1}^{M} \exp(u_t^{(i)} - a_t), \\ &= \log \hat{p}(y_{1:t-1}) + a_t + \log \frac{1}{M} \sum_{i=1}^{M} \exp(u_t^{(i)} - a_t).\end{aligned}$$

5.2.2 Parallel Functional Particle Filter

A functional implementation of a simple bootstrap particle filter is considered in section 2.1.14. This implementation is polymorphic in the collection type containing the observations, using the `Traverse` type class. However, the `Vector` collection type is used to hold the particle cloud representing the latent-state. `Vector` is a sequential collection type which supports approximately constant time random access to elements, this is a useful property for resampling, however due to it's sequential nature cannot be used for a parallel particle filter.

Typically the slowest step in a particle filter is sampling from the latent-state transition kernel, $\theta_t^{(i)} \sim p(\theta_t|\theta_{t-1}^{(i)})$ for $i = 1, \dots, M$. This step is straightforward to parallelise, since each state is advanced independently. The higher order function `map` is used to apply a function to each element of a collection independently and a `Functor` (see section 2.1.7) represents a type constructor that can be mapped, therefore a generic function to advance the state is:

```scala
def transitionKernel: State => State
def advanceState[F[_]: Functor](xs: F[State]): F[State] =
  xs map transitionKernel
```

If the transition kernel for the latent-state is expensive to compute, or the number of particles required in the particle filter is extremely large then parallelisation of this step is often beneficial. In order to use this function for a parallel collection then a functor type class instance must be defined for a parallel collection such as `ParVector`.

To use the function `advanceState` in parallel, first define the initial state to be a parallel collection, then apply the function:

```scala
val x0: ParVector[State] = Vector.fill(100)(0.0).par
advanceState(x0)
```

This creates a `Vector` containing 100 copies of 0.0 then converts it to a parallel collection using the method `.par`. The function `advanceState` can be used with any type constructor which has an instance of `Functor` in scope.

The next step of the bootstrap filter is to calculate the weights for each particle $w_t^{(i)} \sim p(y_t|\theta_t^{(i)})$. This step is also independent and can be performed in parallel using `map`:

```
def conditionalLikelihood: Observation => State => LogLikelihood
def calculateWeights[F[_]: Functor](
  xs: F[State],
  y: Observation): F[LogLikelihood] =
  xs map conditionalLikelihood(y)
```

Next is the resampling step, this is more difficult to perform in parallel and is dependent on the resampling scheme chosen, see Murray et al. [2016] for a detailed discussion of parallel resampling in the particle filter. Metropolis resampling is one of the algorithms considered by Murray et al. which produces biased samples, potentially causing a problem for parameter inference using PMMH introduced in section 5.3. However, Metropolis resampling can be performed in parallel as only the ratio between pairs of weights is considered. Metropolis resampling is reproduced in algorithm 14. Systematic resampling is an alternative resampling scheme which produces unbiased samples and is straightforward to parallelise. Resampling schemes which produce unbiased samples such as Multinomial resampling and systematic resampling are used in the particle filter to estimate the marginal likelihood in the PMMH algorithm. For forecasting and interpolation conditional on the static parameters, the parallel Metropolis-Hastings resampling algorithm can be used.

Algorithm 14: The metropolis resampling algorithm.

```
   Result: Return the set of indices representing the ancestors of $M$ particles $\mathbf{a}$
1  Given $\mathbf{w} = \{w^{(1)}, \dots, w^{(M)}\}$ and a preset number of iterations $B$;
2  for $i$ in $1{:}M$ do
3      Set $k = i$;
4      for $n$ in $1{:}B$ do
5          Sample $u \sim \mathcal{U}(0,1)$;
6          Sample $j \sim \mathcal{U}(1,M)$;
7          if $(u \leq \frac{w^{(j)}}{w^{(k)}})$ Set $k = j$;
8      end
9      $a^{(i)} = k$
10 end
```

First write the inner loop using a tail-recursive function, this depends on the value of k, the vector of weights `w` and the number of iterations `b`.

```
def loop(b: Int, k: Int, n: Int, w: Vector[Double]): Int = {
    if (b == 0) {
      k
    } else {
      val u = Uniform(0, 1).draw
      val j = discreteUniform(0, n-1).draw
      if (u <= w(j)/w(k)) {
        loop(b-1, j)
      } else {
        loop(b-1, k)
      }
    }
  }
```

The main function can then utilise this helper function, the outer for loop is implemented as a map function on a parallel collection of the weights, `ParVector[Double]`.

```
def metropolisResampling(b: Int, weights: ParVector[Double]) = {
  val n = weights.size
  weights.indicies.map {i => loop(b, i, n, weights) }
}
```

Note that the inner for loop is performed in parallel, using parallel collections, it does not matter in which order the weights are compared. A single step of the parallel functional particle filter can now be written as:

```
def stepPfParallel(x0: ParVector[Double], y: Int): ParVector[Double] = {
  val advanced = x0 map stepState
  val w = advanced map (x => conditionalLl(x, y))
  metropolisResample(w, advanced).draw
}
```

`stepPfParallel` can now be applied to a collection of observations using `scanLeft` to provide an approximation for the filtering distribution. An application of the bootstrap particle filter with parallel Metropolis resampling was performed using the simulated Poisson data from section 5.1.1, $B = 10$ iterations were used with $M = 500$ particles. Figure 5.4 shows the mean of the filtering distribution with ± 2 standard deviations. A particle filter using Multinomial resampling (which is unbiased) was also used as a comparison for filtering.

5.3 Pseudo-Marginal Metropolis-Hastings

In order to perform inference for the joint posterior distribution $p(\theta_{0:N}, \psi | y_{1:N})$ the particle-marginal Metropolis-Hastings algorithm is used. This is a MH algorithm (see section 1.3.2)

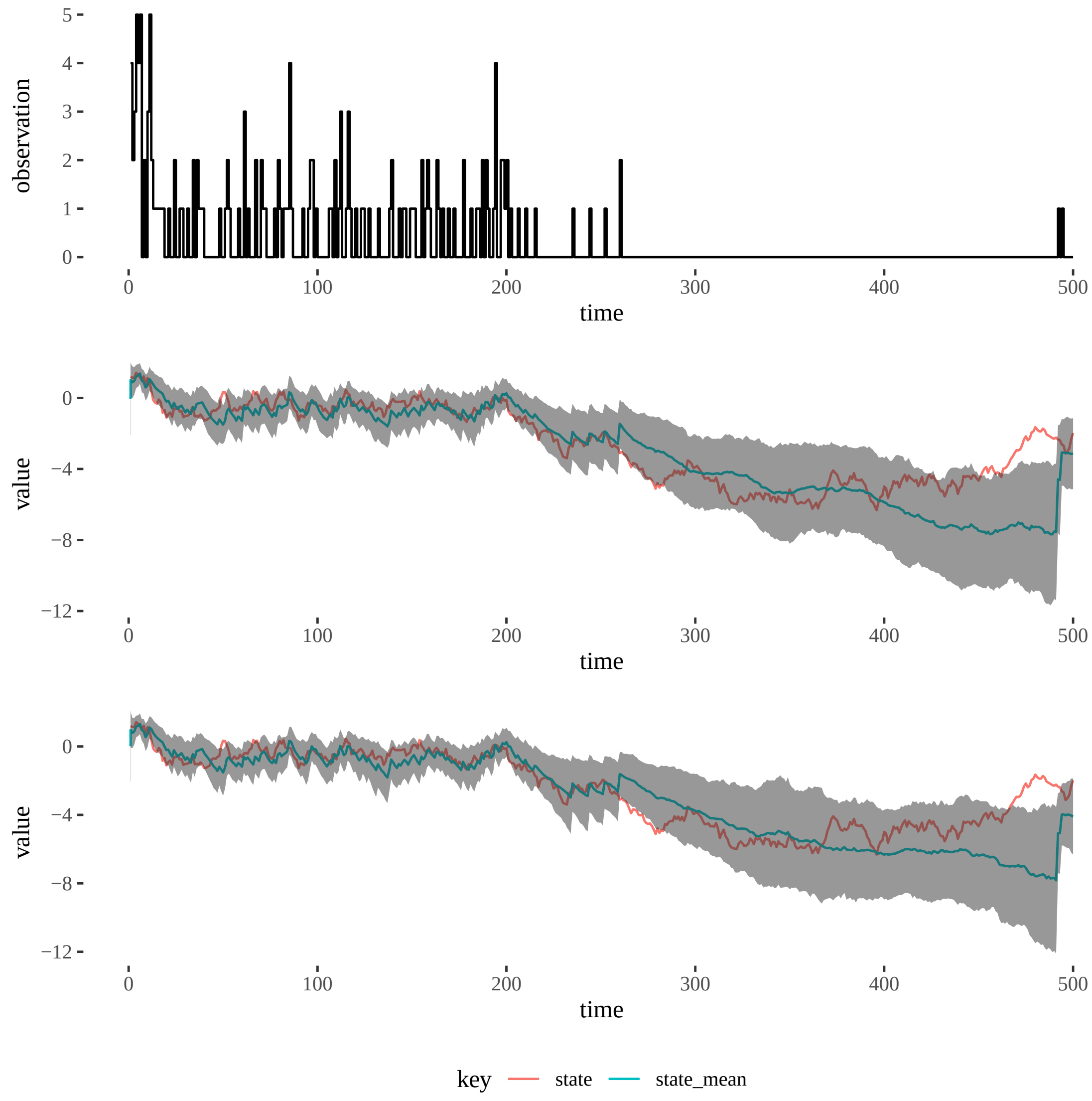


Figure 5.4: (Top) Simulated count observations from the Poisson model (Middle) Filtered state mean, ± 2 standard deviations and simulated state using the parallel bootstrap particle filter with Metropolis resampling (Bottom) Filtered state mean, ± 2 standard deviations using a serial particle filter with Multinomial resampling

where the marginal-likelihood is approximated using the bootstrap particle filter. The marginal-likelihood, $p(Y_{1:N}|\psi) = \int_\theta p(Y_{1:N}|\theta_{0:N})\mathrm{d}\theta$ is required to perform the Metropolis-Hastings algorithm. A particle filter can be used to find an unbiased, (see Del Moral [2004]) estimate of the likelihood as in equation 5.6. The PMMH algorithm is summarised in algorithm 15.

Examples of the Particle Marginal Metropolis-Hastings algorithm are presented in sections 5.4 and 5.9.

Algorithm 15: The particle marginal Metropolis-Hastings algorithm.

```
   Result: Return $(\psi^{(i)}, \theta_{0:N}^{(i)})$ for $i = 1, \dots, M$
 1 Given observations $y_{1:N}$;
 2 Initialise the parameters by sampling from the prior distribution $\psi \sim p(\psi)$;
 3 for $i$ in 1:M do
 4     Propose a new value of the parameters $\psi^\star \sim q(\psi^\star|\psi)$;
 5     Run the particle filter to calculate the pseudo-marginal likelihood $\hat{p}_{\psi^\star}(y_{1:N})$
        and propose a new trajectory for the state $\theta_{0:N}^\star$;
 6     Calculate $A = \frac{p(\psi^\star)\hat{p}_{\psi^\star}(y_{1:N})q(\psi|\psi^\star)}{p(\psi^\star)\hat{p}_{\psi}(Y_{1:N})q(\psi^\star|\psi)}$;
 7     Sample $u \sim U[0,1]$;
 8     if $u < A$ then
 9         Set $\psi^{(i)} = \psi^\star, \theta_{0:N}^{(i)} = \theta_{0:N}^\star$;
10     else
11         Set $\psi^{(i)} = \psi^{(i-1)}, \theta_{0:N}^{(i)} = \theta_{0:N}^{(i-1)}$;
12 end
```

5.4 Student's-t Filter

Although the particle filter (see section 1.3.5) can be used to determine the latent-state of the non-Gaussian state space model with Student's t-distributed observations, there is a way to use the Kalman Filter to determine the state of a Student's-t state space model. The Kalman filter is more efficient to compute and does not suffer from problems such as particle impoverishment or degeneracy.

The Student's t-distribution arises as an Inverse-Gamma mixture of Normals. Consider that $X|V \sim \mathcal{N}(\mu, V)$, $V \sim \text{Inv-Gamma}(\alpha, \beta)$ then the marginal distribution of X is:

$$
\begin{aligned}
p(X) &= \int_0^\infty p(X|V)p(V)\mathrm{d}V, \\
&= \int_0^\infty \frac{\beta^\alpha}{\sqrt{2\pi}\Gamma(\alpha)} V^{-\frac{1}{2}} \exp\left(\frac{-(x-\mu)^2}{2V}\right) V^{-\alpha-1} \exp\left(-\frac{\beta}{V}\right) \mathrm{d}V, \\
&= \frac{\beta^\alpha}{\sqrt{2\pi}\Gamma(\alpha)} \int_0^\infty V^{-(\alpha+\frac{1}{2})-1} \exp\left\{-\frac{1}{V}\left(\frac{(x-\mu)^2}{2}+\beta\right)\right\} \mathrm{d}V, \\
&= \frac{\beta^\alpha \Gamma\left(\alpha+\frac{1}{2}\right)}{\sqrt{2\pi}\Gamma(\alpha)\left(\frac{(x-\mu)^2}{2}+\beta\right)^{\alpha+\frac{1}{2}}} \\
&= \frac{\beta^\alpha \Gamma\left(\alpha+\frac{1}{2}\right)}{\sqrt{2\pi}\Gamma(\alpha)\beta^{\alpha+\frac{1}{2}}\left(\frac{(x-\mu)^2}{2\beta}+1\right)^{\alpha+\frac{1}{2}}} \\
&= \frac{\Gamma\left(\frac{2\alpha+1}{2}\right)}{\sqrt{2\pi\beta}\Gamma(\alpha)}\left(\frac{(x-\mu)^2}{2\beta}+1\right)^{-\frac{2\alpha+1}{2}}. \qquad (5.9)
\end{aligned}
$$

This is the PDF of a scaled, shifted Student's t-distribution with $\nu = 2\alpha$, $\ell = \mu$ and $s = \sqrt{\beta/\alpha}$.

Now a simulation from the scaled, shifted Student's t-distribution can be obtained by first simulating V from the Inverse Gamma distribution with parameters $\alpha = \nu/2$ and $\beta = \nu s^2/2$, then simulating from a Normal distribution with mean ℓ and variance V. Further, this means that the Student's t-distributed model is conditionally Gaussian and a Kalman filter (see section 4.2) can be used to determine the filtering distribution.

In the state space model given in equation 5.4 the location of the Student's t-distribution is dynamic and given by $\eta_t = F_t^T\theta_t$, the scale and degrees-of-freedom are static parameters. Consider the measurement update step of the Kalman filter with the new observation distribution:

$$
p(Y_t|\theta_t) = \int_0^\infty \mathcal{N}\left(Y_t; F_t\theta_t, V_t\right) \text{InverseGamma}\left(V_t; \frac{\nu}{2}, \frac{\nu s^2}{2}\right) \mathrm{d}V_t
$$

using the result from equation 5.9. The joint distribution of all the random variables in the model can be written as:

$$
\begin{aligned}
p(\theta_{0:N}, y_{1:N}, s^2, \nu, W) &= p(\nu)p(s^2)p(W)p(\theta_0)\prod_{t=1}^N p(V_t)p(Y_t|\theta_t, V_t)p(\theta_t|\theta_{t-1}, W) \\
&= p(s^2)p(\nu)\text{Inv-Gamma}(W_j; \alpha, \beta)\mathcal{N}(\theta_0; m_0, C_0)\times \\
&\prod_{t=1}^N \text{Inv-Gamma}\left(V_t; \frac{\nu}{2}, \frac{\nu s^2}{2}\right)\mathcal{N}(Y_t; F_t\theta_t, V_t)\mathcal{N}(\theta_t; G_t\theta_{t-1}, W),
\end{aligned}
$$

using the Markov property of the latent-state and the conditional independence in the model.

In order to learn the static parameters of the model, construct a Gibbs sampler, the state can be sampled conditional on the observed data $y_{1:N}$, the static parameters of the model and the auxiliary variance parameters $V_{1:N}$ using FFBS (see section 4.4).

For the system noise matrix, assuming the matrix is diagonal then the conditionally conjugate prior distribution for each diagonal element is the Inverse Gamma distribution

$$\begin{aligned} p(W|y_{1:N}, \theta_{0:N}, \psi^{-W}) &= p(W) \prod_{t=1}^{N} p(\theta_t|\theta_{t-1}, W) \\ &= \text{Inv-Gamma}(\alpha, \beta) \prod_{t=1}^{N} \mathcal{N}(G_t\theta_{t-1}, W) \\ &= \text{Inv-Gamma}\left(\alpha + \frac{T}{2}, \beta + \frac{1}{2}\sum_{t=1}^{N}(\theta_t - G_t\theta_{t-1})^2\right). \end{aligned} \tag{5.10}$$

Alternatively the system noise covariance matrix can be considered full-rank and be drawn from an Inverse Wishart distribution as in equation (4.18). Then for each observation of the time series $y_{1:N}$, a value of the variance is sampled

$$\begin{aligned} p(V_t|y_{1:N}, \theta_{0:N}, \psi) &= p(V_t)p(Y_t|\theta_t, V_t), \\ &= \text{Inv-Gamma}\left(\frac{\nu}{2}, \frac{\nu s^2}{2}\right)\mathcal{N}(F_t\theta_t, V_t), \\ &= \text{Inv-Gamma}\left(\frac{\nu+1}{2}, \frac{\nu s^2 + (y_t - F_t\theta_t)^2}{2}\right). \end{aligned} \tag{5.11}$$

The posterior distribution of the degrees of freedom, ν, and the scale, s^2, can also be learned from the data using a Metropolis-Hastings step. The Kalman filter must be modified to sample the variance at each time-step. The one step update in the Kalman filter conditional upon the sampled variances is

$$\begin{aligned} \text{Prediction:} \qquad & \text{draw } V_t \sim \text{Inv-Gamma}\left(\frac{\nu+1}{2}, \frac{\nu s^2 + (y_t - F_t\theta_t)^2}{2}\right). \\ & f_t = F_t a_t \\ & Q_t = F_t R_t F_t' + V_t \end{aligned}$$

This is sufficient to learn the joint posterior distribution of the parameters and latent-state of a state space model with Student's t-Distributed observation noise. The Gibbs sampler for the Student's t-distributed state space model is summarised in algorithm 16.

Algorithm 16: MCMC algorithm for the Student's t-distributed state space model.

```
Result: Return M samples of the static parameters $\psi = \{s^2, W, \nu\}$ and latent
        state $\theta_{0:N}$
1  Given $N$ observations $y_{1:N}$;
2  Initialise the values of the parameters by sampling from the prior distribution
   $\psi_0 \sim p(\psi)$;
3  for $i$ in 1:M do
4      Sample the state using FFBS, conditional on the auxiliary variances
       $p(\theta_{0:N}|\psi_i, y_{1:N}, V_{1:N})$;
5      Perform a Metropolis-Hastings step to sample $s^2, \nu$;
6      Sample the state noise covariance from the full conditional:
           $$p(W|y_{1:N}, \theta_{0:N}, \psi_i^{-W}) = \text{Inv-Gamma}\left(\alpha + \frac{T}{2}, \beta + \frac{1}{2}\sum_{t=1}^{N}(\theta_t - G_t\theta_{t-1})^2\right).$$
       ;
7      for $t$ in 1:N do
8          Sample the auxiliary variances from
               $$p(V_t|y_t, \theta_t, \psi_i) = \text{Inv-Gamma}\left(\frac{\nu+1}{2}, \frac{\nu s^2 + (y_t - F_t\theta_t)^2}{2}\right).$$
           ;
9      end
10 end
```

The Gibbs algorithm is now compared to the PMMH algorithm for a simulated first order Student-t Model as plotted in Figure 5.3. Both algorithms produce exact inferences about the parameter posterior distributions however the Gibbs algorithm should be much faster. The prior distributions for both inference algorithms were chosen to be:

$$\begin{aligned}
p(W) &= \text{Inv-Gamma}(21.0, 2.0), \\
p(s^2) &= \text{Inv-Gamma}(4.0, 9.0), \\
p(\nu) &= \text{Poisson}(3) \setminus \{0\}.
\end{aligned}$$

The mean of the Inverse Gamma distributions for W and s^2 are the same as the parameters used to simulate the data. A truncated Poisson distribution is used for the prior of the degrees of freedom, excluding values equal to zero. Figure 5.5 shows diagnostic plots for

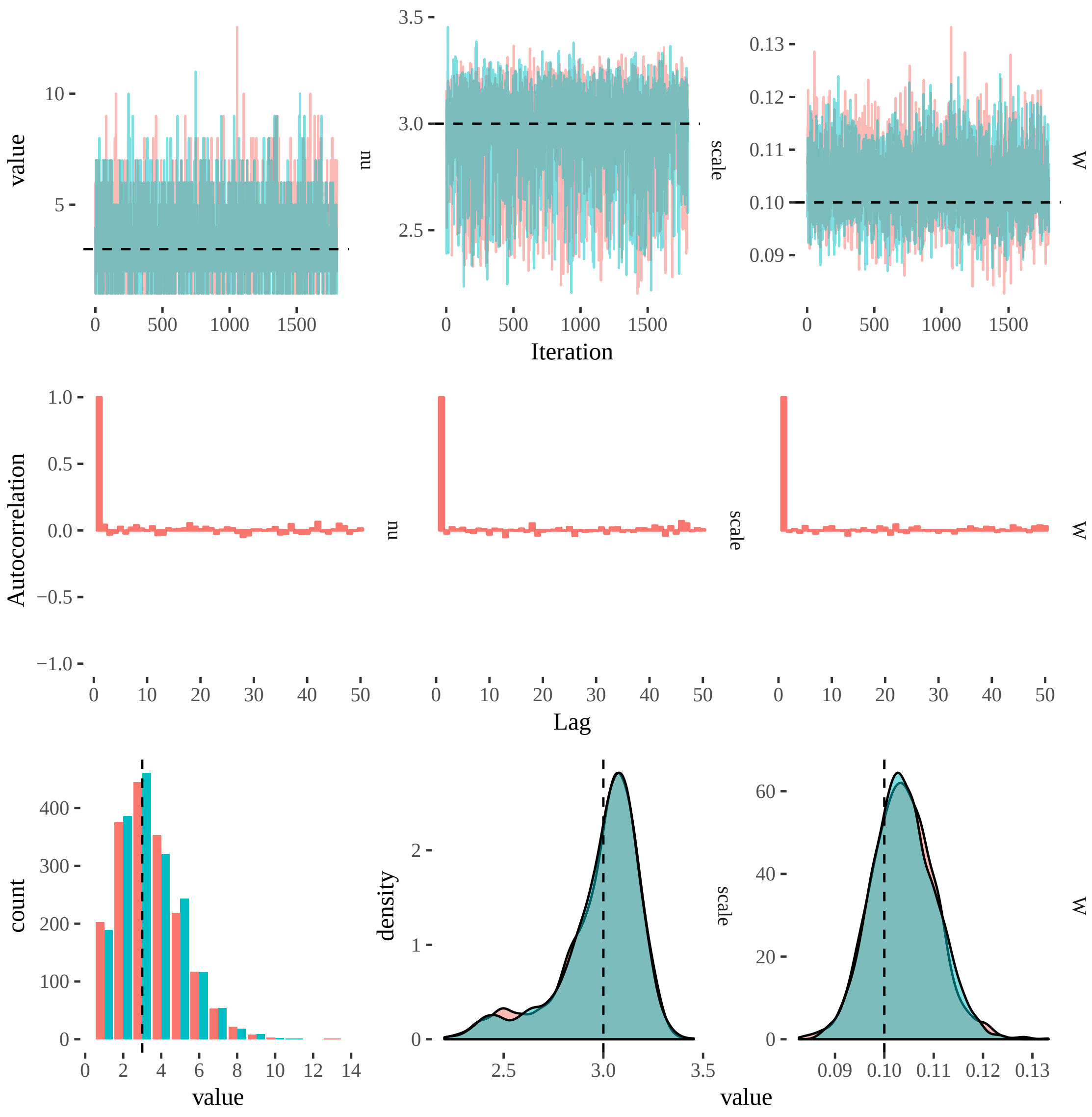


Figure 5.5: Diagnostic plots for the MCMC draws using Gibbs sampling for the simulated Student's t-distributed model, the parameter values used to simulate the data are plotted using the dashed line. The traceplots of the MCMC draws, (top). The autocorrelation function for each parameter showing fast decay indicating low autocorrelation, (middle). The empirical marginal posterior densities, (bottom).

the Gibbs algorithm. Two MCMC chains were run for 10,000 iterations with the first 1,000 iterations discarded as burn-in, thinned by a factor of 5 to reduce autocorrelation in the chain. The algorithm has recovered the values of the parameters used to simulate the data as expected.

For the PMMH algorithm, a symmetric Normal proposal distribution is used for log of the static continuous parameters, s^2 and W with a variance 0.01^2. A Negative Binomial

Particles	$\text{Var}(\log(\hat{p}_\psi(y_{1:N})))$
50	5.28
100	2.47
200	1.15

Table 5.1: The variance of the pseudo-marginal log-likelihood using 1,000 repetitions of the bootstrap particle filter for the Student-t distributed state space model. The parameter values used to simulate the data were when computing the marginal log-likelihood.

distribution is used for the proposal distribution of the degrees of freedom of the observation distribution. A pilot run was used to determine the number of particles to use in the bootstrap filter, a common rule of thumb is to use the minimum number of particles which results in the variance of the log-likelihood estimate to be close to one at a central parameter value (such as the posterior mean) [Doucet et al., 2015]. This results in an efficient PMMH algorithm which accurately estimates the log-likelihood. The results from the pilot run presented in table 5.1. 1,000 runs of the particle filter were performed with 50, 100 and 200 particles each.

A Bootstrap particle filter with 200 particles and systematic resampling at every iteration is used to calculate the pseudo-marginal likelihood. The likelihood from the previous iteration is stored such that the algorithm converges to the correct posterior. Two MCMC chains were run in parallel with 100,000 iterations.

Figure 5.6 shows diagnostic plots for the PMMH algorithm. The PMMH algorithm was ran for 10 times the length of the Gibbs sampler to achieve comparable results, this could be because the parameters are proposed in one block since re-evaluating the pseudo marginal likelihood multiple times at each time step is costly. Whereas using Gibbs sampling and FFBS, the likelihood conditional on the sampled latent-state is cheap to compute and hence the Metropolis-Hastings steps for the scale and degrees of freedom can be performed independently conditional on the other parameters.

In order to compare the efficiency of each of the inference algorithms, the effective sample size (ESS) per second for each of the parameters in the chain is calculated [Ripley, 1987]. The effective sample size of a chain of length n is calculated by:

$$ESS = \frac{n}{1 + 2\sum_{i=1}^{\infty}\rho(i)}, \tag{5.12}$$

where $\rho(i)$ is the auto-correlation of the chain at lag i. Calculating iterations per second is not a good measurement of the efficiency of an inference algorithm using Markov chain Monte Carlo since the iterations from the Markov chain are correlated and not independent draws from the posterior distribution. The ESS/s takes into account the auto-correlation

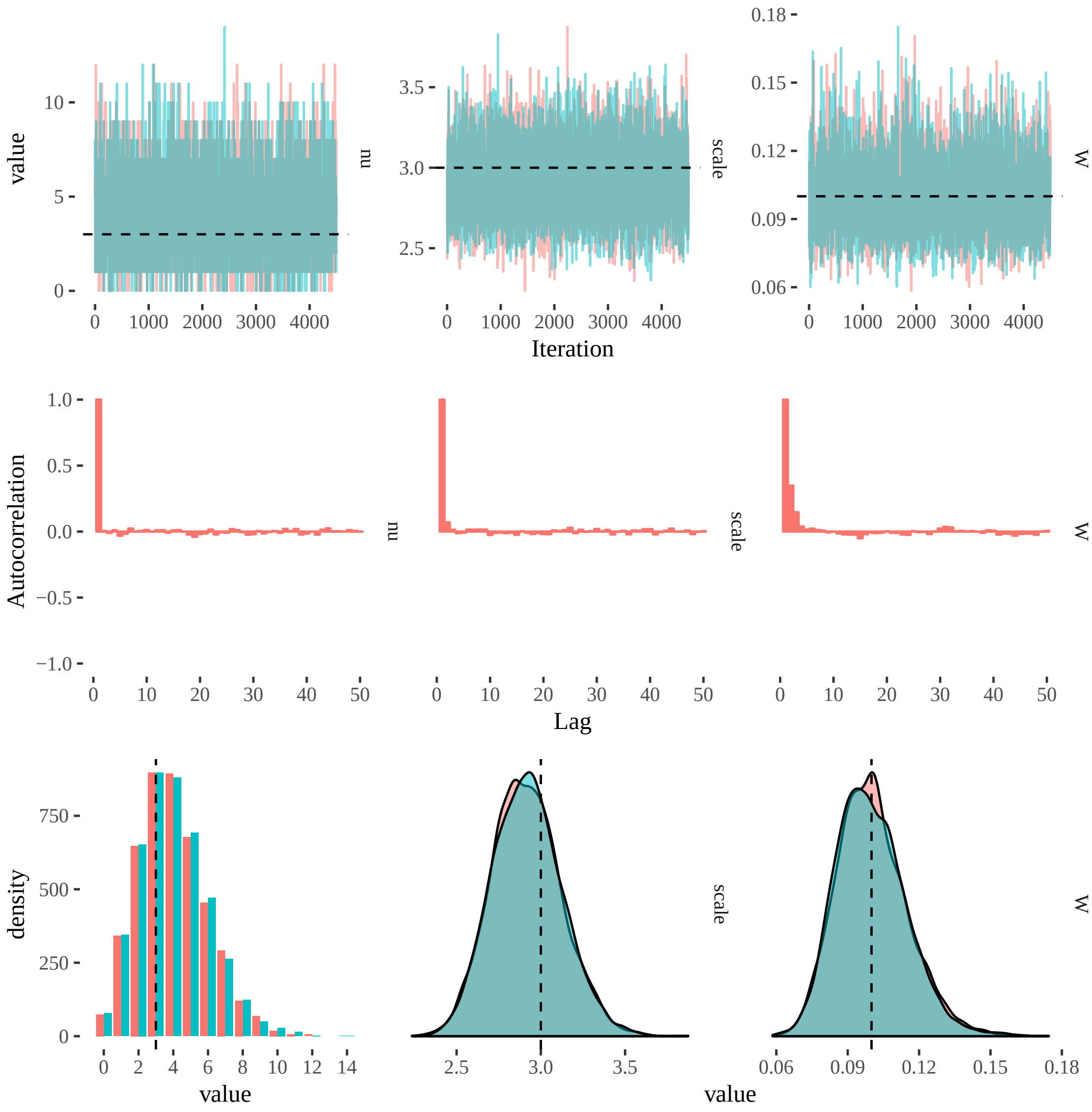


Figure 5.6: Diagnostic plots for parameter inference using the PMMH algorithm for the simulated Student's t-distributed model. Traceplots (top), Autocorrelation plots (middle), empirical marginal density plots (bottom).

of the parameters and hence rewards algorithms which are slower per-iteration, but more effective at drawing independent samples. In a well mixing chain the auto-correlation between successive iterations decays rapidly. An efficient MCMC has a higher ESS/s which takes into account both auto-correlation and the speed of each iteration. Table 5.2 shows the ESS/s for the Gibbs algorithm and the PMMH algorithm for 10,000 iterations of each inference algorithm.

The Gibbs sampling algorithm is clearly more efficient than the PMMH algorithm. However, the PMMH algorithm is more general and can be applied to a wide array of

Parameter	ESS/s Gibbs	ESS/s PMMH
s^2	1.99	0.53
ν	1.67	0.35
W	15.24	3.39

Table 5.2: The efficiency measured in effective sample size per second for the PMMH algorithm and the Gibbs algorithm for the Student-t distributed state space model.

state space models, not just the Student-t model.

5.5 Forecasting Non-linear State Space Models

Once the parameter posterior distributions have been estimated using MCMC given observations from the time series $y_{1:N}$, it is of interest to forecast observations of the process taking into account the uncertainty in the joint posterior distribution for the latent-state and parameters. Suppose M independent samples are available from the parameter posterior distribution, $\psi^{(i)}, i = 1, \dots, M$ and associated samples of the posterior distribution of the latent-state at time N, $\theta_N^{(i)}, i = 1, \dots, M$ for a model of interest. In order to produce accurate forecast distributions with uncertainty, then first advance the state forward k steps using each of these tuples $(\psi^{(i)}, \theta_N^{(i)})$:

$$p(\theta_{N+k}|\psi, \theta_{0:N}) \approx \prod_{t=N}^{N+k} (G_t\theta_t^{(i)} + w_t), \quad w_t \sim \mathcal{N}(0, W^{(i)}), \quad \text{for } i = 1, \dots, M.$$

Call these samples at time $N + k$, $\hat{\theta}_{N+k}^{(i)}, i = 1, \dots, M$, then transform the sample using the observation matrix F_{N+k} and the link function g:

$$\eta_{N+k}|\psi, \theta_{N+k} \approx g(F_{N+k}\hat{\theta}_{N+k}^{(i)}), \quad \text{for } i = 1, \dots, M.$$

Finally, M samples are taken from the observation distribution

$$y_{N+k} \approx \pi(\hat{\eta}_{N+K}^{(i)}, \psi^{(i)}), \quad \text{for } i = 1, \dots, M.$$

Summary statistics such as the mean and credible intervals can be calculated from this sample.

5.6 Non-Gaussian Modelling Language

The Non-Gaussian state-space models introduced in this chapter form part of the Bayesian DLMs[2] Scala library introduced in section 4.8.

A non-Gaussian state space model is referred to as a `Dglm` even though not all models are exponential family models. The class `Dglm` has five associated functions:

```scala
case class Dglm(
  observation: (State, Scale) => Rand[Observation],
  f: Time => DenseMatrix[Double],
  g: Time => DenseMatrix[Double],
  link: State => State,
  conditionalLikelihood:
    (Scale, State, Observation) => LogLikelihood
)
```

The `observation` function is a function from the latent state, `State`, and a scale parameter for the observation distribution to a distribution over the possible observations. `f` and `g` are the observation and system matrices from the DLM class introduced previously. `link` is a link function required to transform the unbound latent-state into the parameter-space of the observation distribution. Finally, `conditionalLikelihood` is a function from the scale of the observation distribution, the latent state and observation to a log-likelihood value.

Many built in models are provided, including the models introduced in this chapter. To define a non-Gaussian state space model, first define a DLM using the API from section 4.8, then choose an observation distribution:

```scala
val dlm = Dlm.polynomial(2) |+| Dlm.seasonal(24, 5)
val nonGaussian = Dglm.beta(dlm)
```

This defines a non-Gaussian state space model with a Beta observation distribution suitable for modelling proportions; the link function is logistic-function.

5.7 Modelling NO

Nitric oxide (NO) is an environmental pollutant which contributes to acid rain, smog and is known to cause damage to the ozone layer. It is also known to cause symptoms of respiratory diseases like emphysema. NO is commonly produced by car engines and so is particularly prevalent in large cities with a high volume of motor traffic. NO is measured in $\mu g m^{-3}$.

[2] `https://git.io/dlm`

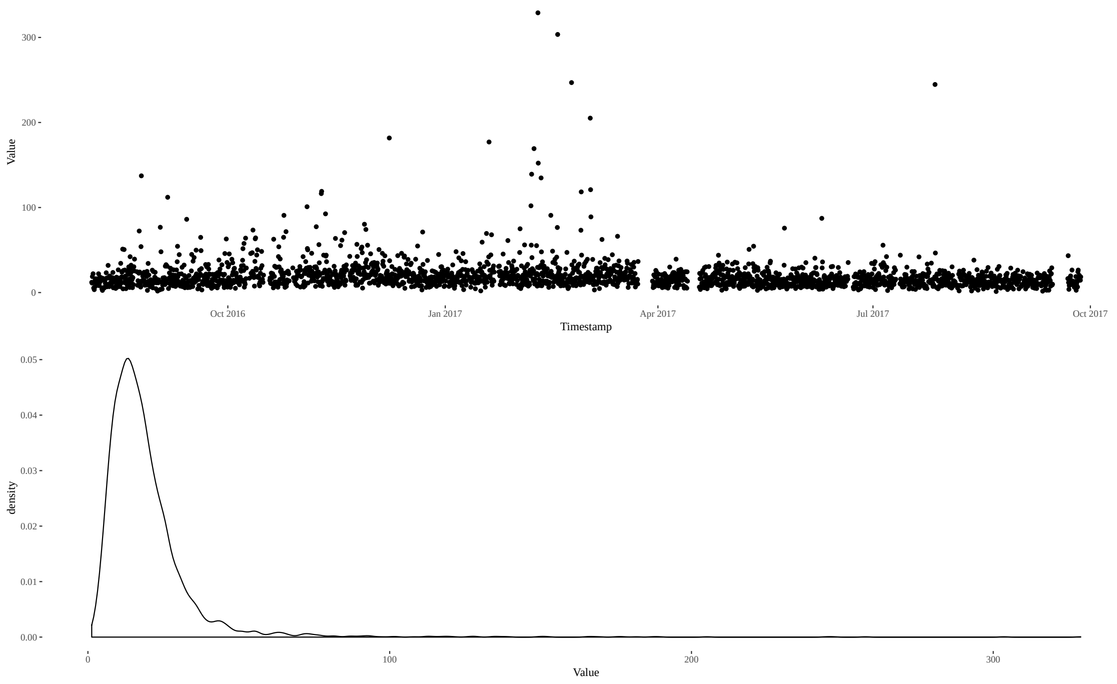


Figure 5.7: NO readings by time showing many outlying values, (top). NO readings as a density showing a long tail, (bottom).

The data is measured by a sensor known as *"new new emote 1108"* located at -1.625 E, 54.974 N (See Figure 5.8), this is just off Westgate Road in Newcastle upon Tyne. The sensor measures humidity, NO, NO_2, Sound, CO and Temperature. The data used to learn the posterior distribution of the static parameters is recorded between 2016-08-04 and 2017-08-15 and data from the period 2017-08-16 to 2017-08-31 is used for evaluation of the model. Figure 5.7 shows the NO readings by time and the density of the NO readings.

Since NO in the city centre is typically caused by emissions from motor vehicles it is expected that there is seasonality in the data. Figure 5.9 shows the mean NO by hour of weekday, indicating that 8am typically has the highest emissions, this corresponds to rush hour and indicates that a seasonal model with a 24 hour period would be suitable for this data. It is also noted that emissions are higher during waking hours and higher Monday to Friday than on weekends.

The model is a composition of a first order polynomial and two seasonal models. The model matrices are chosen to be:

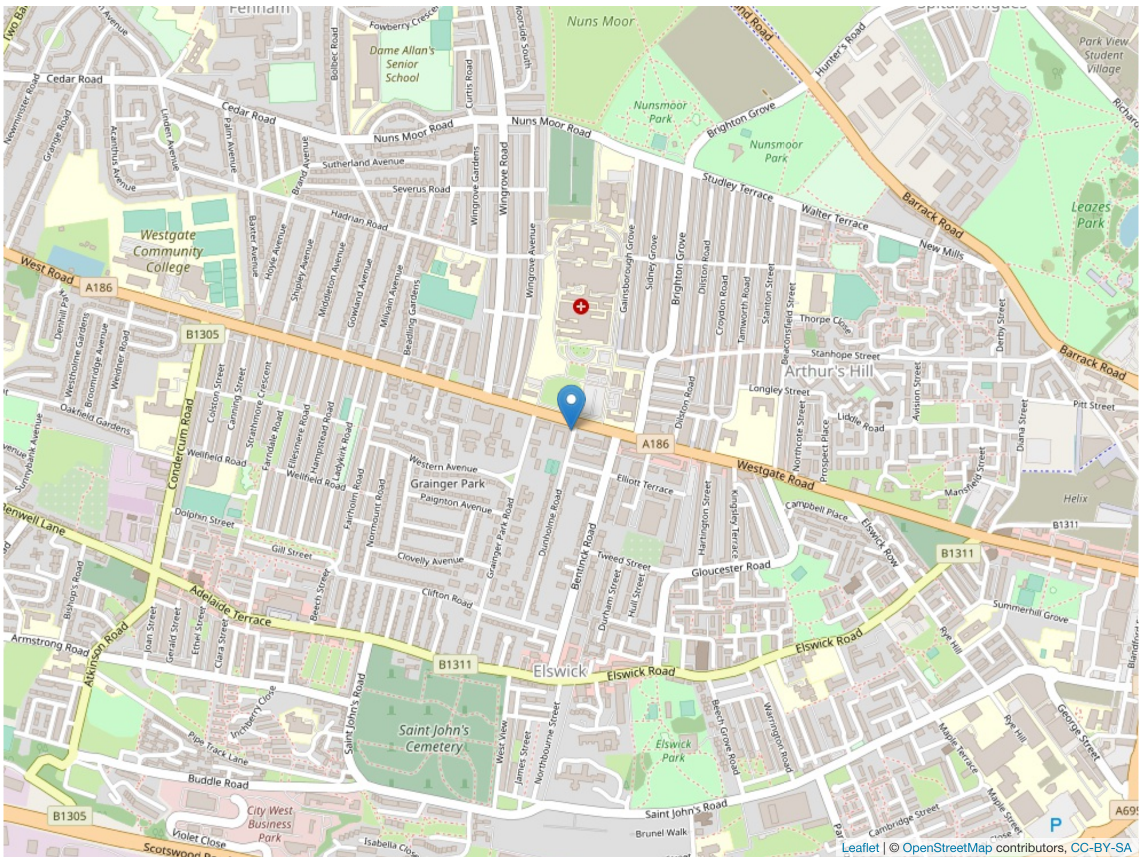


Figure 5.8: A map showing the location of sensor *"new new emote 1108"*.

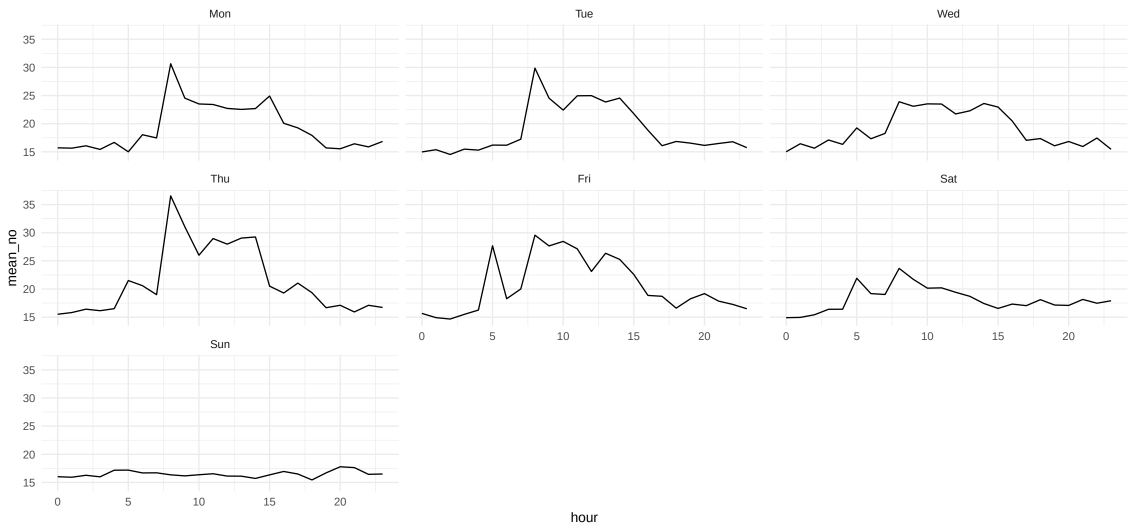


Figure 5.9: The average NO by hour of weekday recorded at *"new new emote 1108"*.

$$F = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ \vdots \\ 1 \\ 0 \end{pmatrix} \quad G_t = \begin{pmatrix} 1 & 0 & \dots & \dots & 0 \\ 0 & R(\omega_1, 1) & 0 & \dots & 0 \\ \vdots & 0 & R(\omega_1, 2) & \dots & 0 \\ \vdots & 0 & \dots & \ddots & 0 \\ 0 & \dots & \dots & 0 & R(\omega_1, h) \end{pmatrix} \tag{5.13}$$

$R(\omega, h)$ is a rotation matrix and $\omega = \frac{2\pi h(\delta t \% P)}{P}$ (see section 4.7 for a more detailed explanation) and the period is $P = 24$.

Due to the large number of outliers present in the data, the Student's t-distribution is used as the observation distribution. The static parameters in the model are the system noise covariance matrix W, the scale of the Student's t-distribution s and the degrees of freedom ν. The posterior distribution of the static parameters will be inferred using the Gibbs sampling algorithm developed in section 5.4. If the mean of the posterior distribution of the degrees of freedom is large this indicates that a Gaussian distribution would be a suitable fit for the data. The prior distributions of the static parameters and initial state were chosen to be:

$$\begin{aligned} p(s^2) &= \text{Inv-Gamma}(10, 3), \\ p(\nu) &= \text{Poisson}(3) \setminus \{0\}, \\ p(W_{ii}) &= \text{Inv-Gamma}(3, 3), \\ p(\theta_0) &= \mathcal{N}(0, 10). \end{aligned}$$

The proposal distribution for the degrees of freedom was chosen to be the Negative Binomial distribution. The MCMC algorithm was run for 100,000 burn-in iterations with two parallel chains, a further 100,000 iterations were monitored and then thinned by a factor of 10 to reduce auto-correlation.

Figure 5.10 shows the diagnostic plots for the scale and the degrees of freedom of the Student t-distributed model for NO. The posterior distribution of the degrees of freedom indicates that the Student's t-distribution is a suitable fit.

Figure 5.11 shows the posterior intervals and median for the diagonal components of the system matrix.

In order to assess the model fit, one-step forecasts are calculated from the held-out test data, starting from August 16th 2017. The particle filter is used to perform one-step

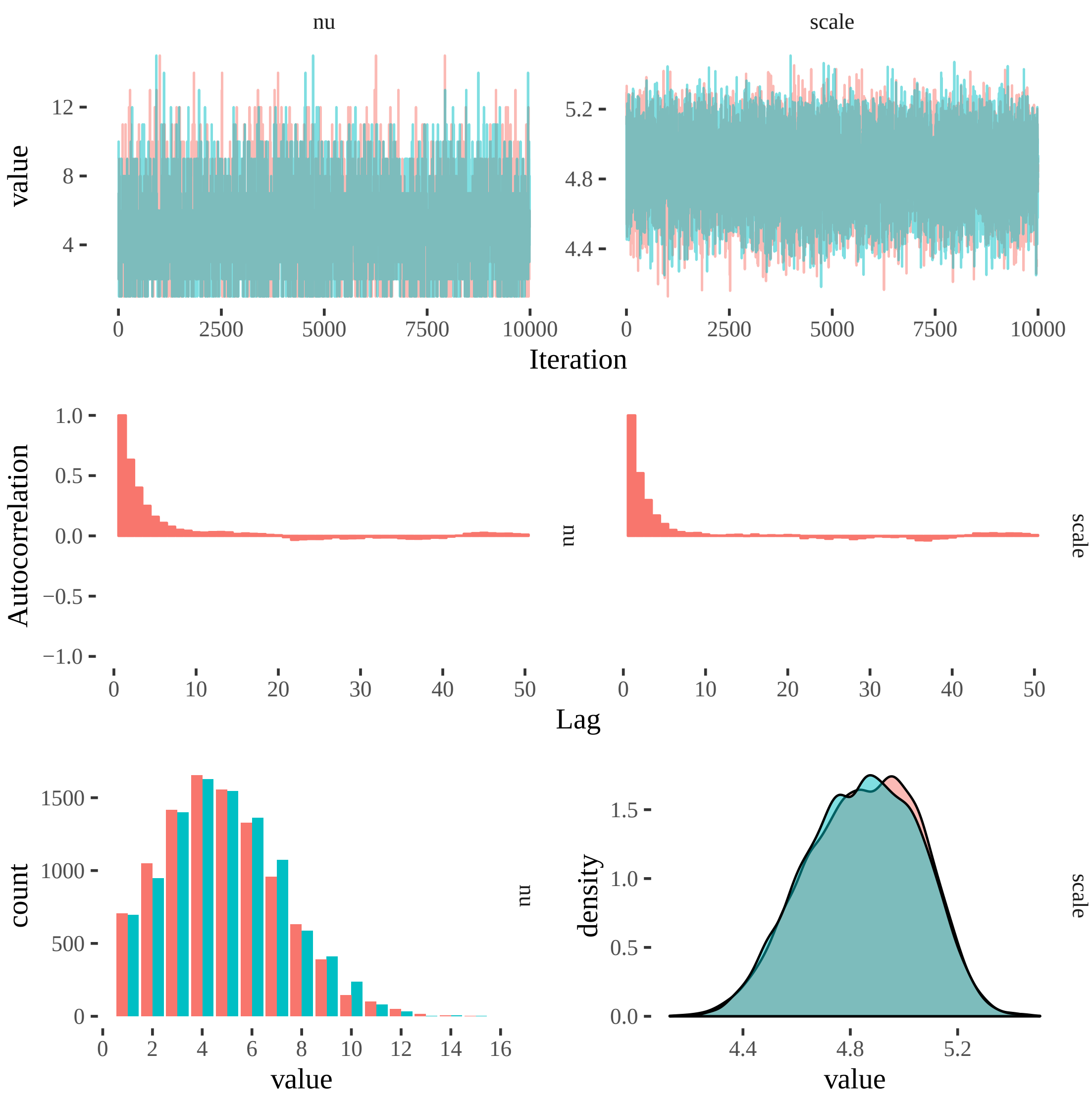


Figure 5.10: Diagnostic plots for the scale and degrees of freedom of the Student t-distributed model for NO pollution. The traceplots of the MCMC draws, (top). Autocorrelation functions showing fast decay indication low autocorrelation, (middle). Empirical marginal posterior densities, (bottom).

forecasts using 100 samples from the parameter posterior distribution. Figure 5.12 (a) shows the forecast mean and 50% credible intervals calculated using $M = 1,000$ particles in the each of the particle filters.

The model appears to be a better fit for the non-transformed NO data than the Gaussian model considered in section 4.9. The scale of the Student's-t distribution is estimated to be smaller than the corresponding variance of the Gaussian distribution resulting in less uncertainty about the forecast values. Figure 5.12 (b) shows the residual Student Q-Q

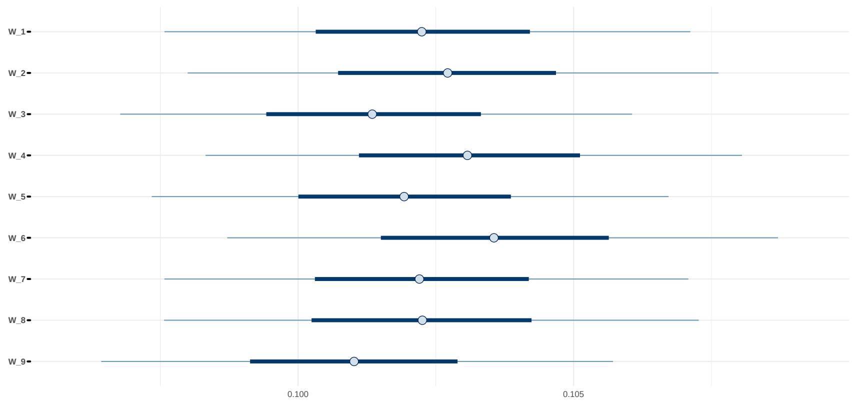
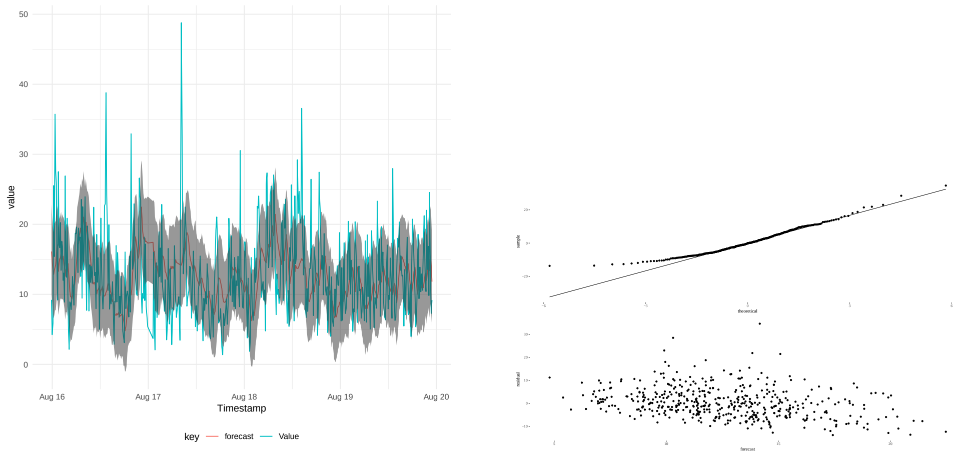


Figure 5.11: Posterior intervals for the diagonal elements of the system matrix for the NO model, central intervals are 50%, outer intervals are 90% and the point estimate is the median.



(a) (b)

Figure 5.12: (a) One step forecast mean and 50% probability intervals for the NO test data for the Student-t DGLM (b) Student Q-Q Plot for the residuals of the one-step forecast values, (top). The residuals and fitted values, showing no heteroskedasticity or non-linearities, (bottom).

plot and residuals against fitted values indicating the Student's-t distributed state space model is a good fit for this data.

5.8 Temperature Student-t Model

A multivariate DLM featuring temperature, NO, CO and humidity in chapter 4 failed to accurately capture the variation in the tails as shown by the normal Q-Q plot in figure 5.12 (b). A Student-t observation distribution may be more appropriate for this data. This example explores the use of the Student-t observation distribution for the Temperature

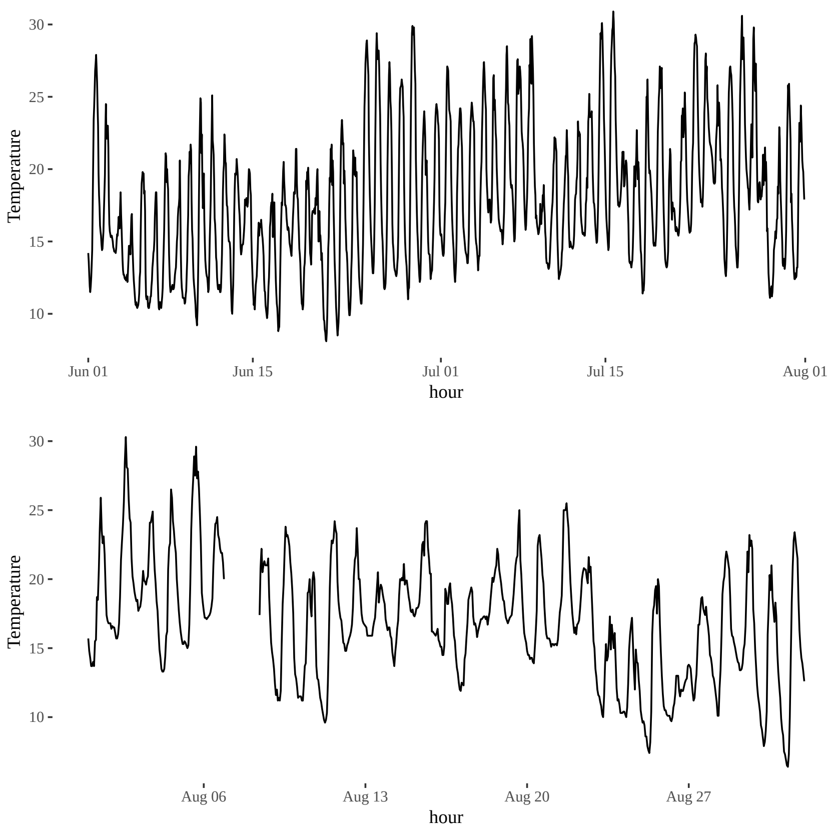


Figure 5.13: The final two months of the training set of temperature readings from *"new new emote 2603"*, (top). The temperature readings in the held out test set, (bottom).

data. Hourly temperature readings were used with eight months of training data from 1st January 2018 up to the 1st August 2018, the month of August 2018 was used as the test set. Figure 5.13 shows the final two months of training data and the month used for the test set. Some data is missing in the test set, however this does not pose a problem for the Kalman filtering algorithm proposed in section 5.4, the same technique is used as described in section 4.2.

The Gibbs sampling algorithm developed in section 5.4 is used to determine the posterior distribution of the Student-t Temperature model. The prior distributions for the static parameters are chosen to be:

$$\begin{aligned}
p(s^2) &= \text{Inv-Gamma}(3.0, 3.0),\\
p(\nu) &= \text{Poisson}(5) \setminus \{0\}\\
p(W_{ii}) &= \text{Inv-Gamma}(3.0, 3.0), i = 1, \dots, 7.
\end{aligned}$$

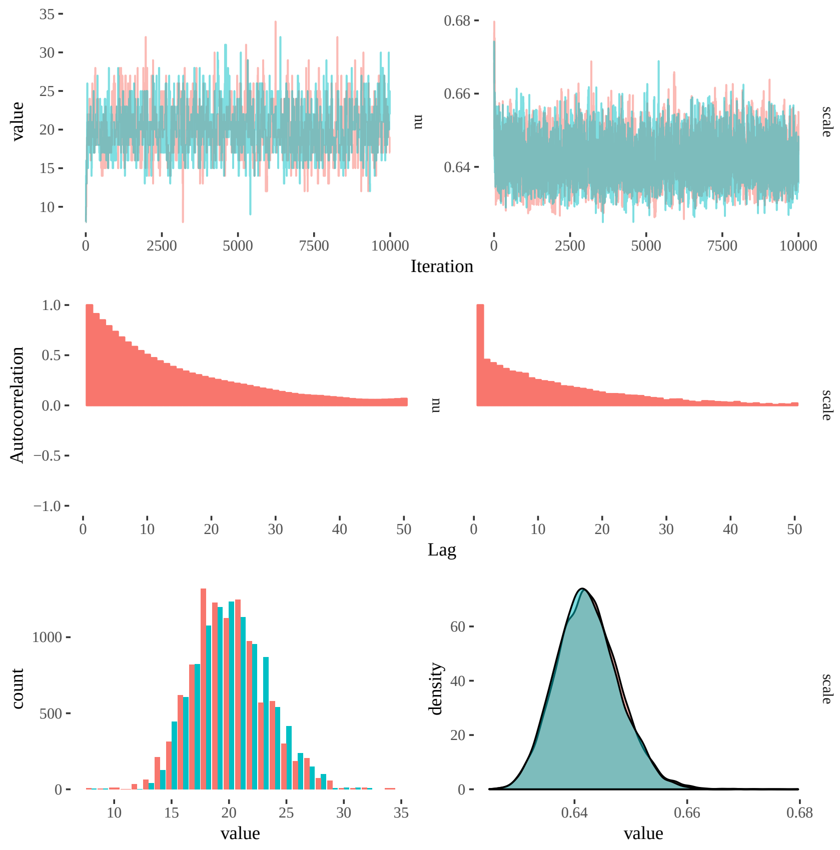
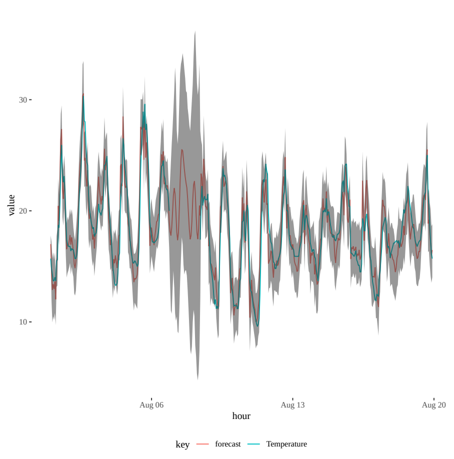


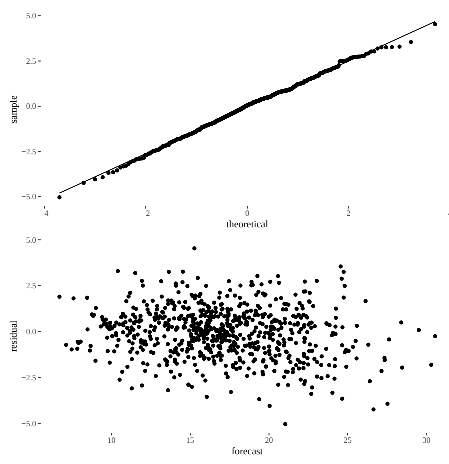
Figure 5.14: Diagnostic plots for the degrees of freedom and scale of the Student's-t observation distribution for the temperature time series recorded at sensor 2603. The traceplots of the MCMC draws, (top). The autocorrelation functions, (middle). The empirical marginal posterior densities, (bottom).

Figure 5.14 shows diagnostic plots from two independent MCMC runs each with 100,000 burn-in iterations followed by 100,000 monitoring iterations and thinned by a factor of 10. The figure shows samples from the posterior distribution of the degrees of freedom and scale of the student-t observation distribution. The degrees are freedom are rather larger than the NO example, indicating the tails of the distribution are not as heavy in this example and that perhaps the Gaussian distribution could be a good fit for this time series. One step forecasts of the traffic data using the test data are presented in figure 5.15 with the residuals for the one-step forecast, the top right plot is a student QQ-plot comparing the quantiles of the residuals to theoretical quantiles of a Student t distribution with degrees of freedom $\nu = 20$. The empirical quantiles are closer to the theoretical quantiles in this QQ-plot than the QQ-plot in chapter 4, figure 4.16 indicating that although the degrees of freedom are large, a Student-t distribution appears to be a better fit.

Figure 5.16 shows the interpolated measurements of the temperature, the interpolation



(a)



(b)

Figure 5.15: (Left) One-step forecast for the test temperature data with 50% credible intervals found by running a particle filter forward with 1,000 particles using the mean of the parameter posterior distribution. (Top Right) Student QQ-plot for the residuals of the Student's-t temperature model, since all points lie close to the line this indicates the model is a good fit for the data (Bottom Right) Residuals against fitted values displaying no noticible patterns.

is performed by determining the joint posterior distribution of the latent-state at the final observation of the training data and the parameters. 1,000 draws from this posterior distribution are used to initialise the backward smoothing algorithm which uses the filtering distribution determined using the conditional Kalman filter determined in section 5.4. The interpolated data is more smooth and less uncertain than the online forecast in figure 5.15.

5.9 Modelling Traffic Data

Traffic patterns can be used to predict emissions of anthropogenic pollutants such as NO_2 and CO. In this example a non-Gaussian state-space model is used to model the total number of vehicles passing through a road segment in a given hour. This can be used to perform forecasting and inform pollution forecasts in the vicinity of the road.

The Urban Observatory [James et al., 2014b] has many sensors for monitoring traffic, sourced from data provided by the North East Combined Authority [NECA, 2016]. Traffic cameras provide information on traffic flow, average speed and congestion. The example considers traffic flow, which is provided in passenger car units at approximately 5 minute intervals. The data is assumed to arrive every five minutes and as a pre-processing step the observations are rounded to the closest five minute interval. Passenger car units represents the size of a vehicle, with larger values representing larger vehicles. In the data provided by the NECA, the values are all integers. Sensor N05171T is the sensor considered in this

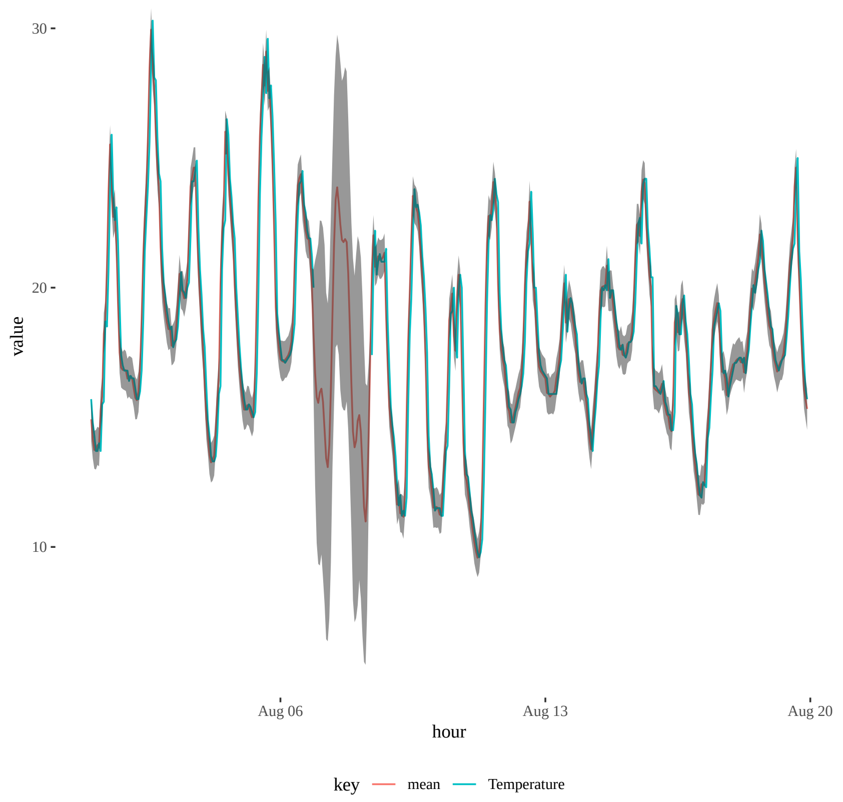
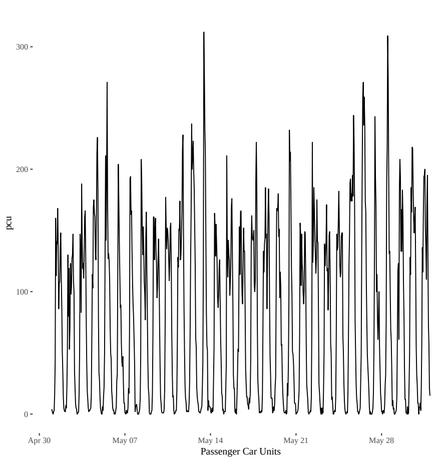


Figure 5.16: The recorded temperature data displays a large period of missing data, this period is interpolated, the mean of the smoothing distribution is plotted in red with the observations plotted in blue. 50% probability intervals are plotted using the shaded grey.

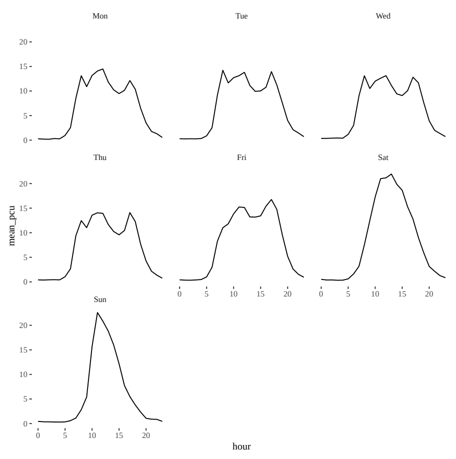
analysis. It is located on Hollinside Road near the Metro Centre in Gateshead. Figure 5.17 (left) shows the sum of PCU by hour during May 2018, showing a strong seasonal trend.

It is thought that the traffic data could have a large amount of zero-readings. Hence the zero inflated Poisson model could be useful for modelling this data. However, the total number of zero-readings at this sensor is 88 (4%) during the month of May 2018. On the other hand, it does appear as if the data is over-dispersed. This indicates that the Negative Binomial distribution could be a good fit for the data

$$\begin{aligned}
Y_t|\eta_t &\sim \text{NegBin}(\eta_t, s^2), \\
\eta_t|\theta_t &= \exp(F^T\theta_t) \\
\theta_t|\theta_{t-1} &= G\theta_{t-1} + w_t, && w_t \sim \text{MVN}(0, W), \\
\theta_0 &\sim \text{MVN}(m_0, C_0).
\end{aligned}$$



(a)



(b)

Figure 5.17: The sum of passenger car units by hour for the month of May 2018, (left). Average passenger car units by hour for each day of the week showing strong daily seasonality, (right).

The system and observation matrices are given by

$$F = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ \vdots \\ 1 \\ 0 \end{pmatrix} \qquad G = \begin{pmatrix} 1 & 0 & \dots & \dots & 0 \\ 0 & R(\omega, 1) & 0 & \dots & 0 \\ \vdots & 0 & R(\omega, 2) & \dots & 0 \\ \vdots & \dots & \dots & \ddots & 0 \\ 0 & \dots & \dots & 0 & R(\omega, h) \end{pmatrix} \tag{5.14}$$

where $R(\omega, h)$ is a rotation matrix through angle $\omega = 2\pi h/24$, the total number of harmonics chosen is $h = 4$. This is a composition of first order polynomial model and a daily seasonal model. Figure 5.17 (Right) shows the average PCU by hour and day of week, showing strong daily seasonality. Saturday and Sunday do appear to have slightly different patterns, this could be accounted for by adding an additional model to the composition with weekly periodicity.

The PMMH algorithm is used to determine the value of the parameters with $M = 500$ particles and a multivariate normal proposal distribution on the log of the diagonal entries of the W matrix and the log of the scale of the Negative Binomial distribution, s^2. The prior distribution for each diagonal entry of W and the scale of the observation distribution is chosen to be:

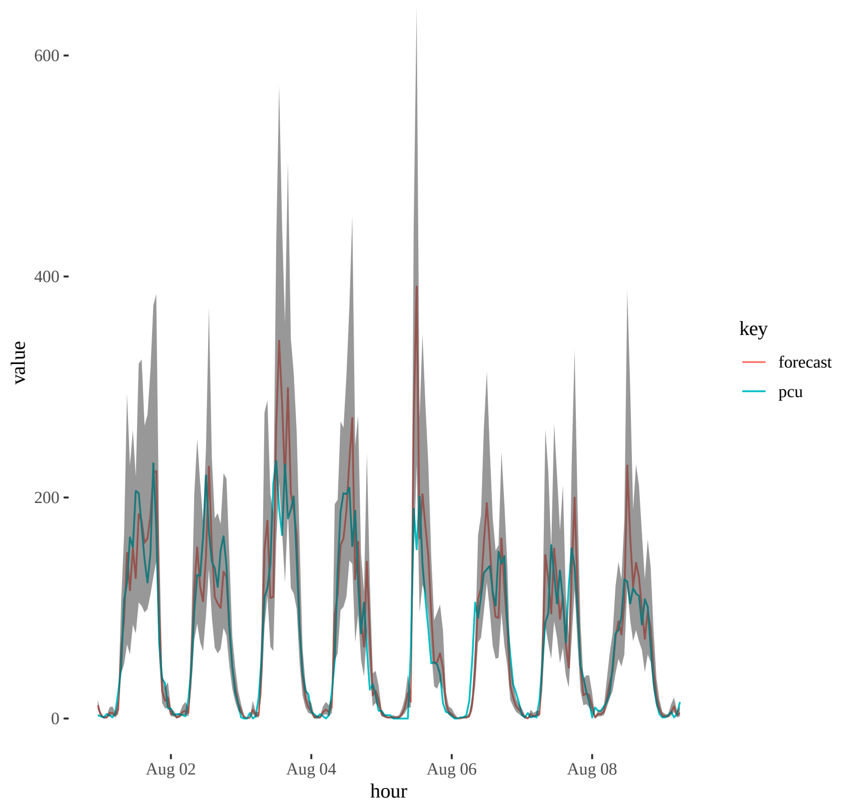


Figure 5.18: One step forecast median and 50% probability intervals for the withheld traffic test data using the Negative Binomial DGLM plotted with the actual measurements in passenger car units per hour.

$$\begin{aligned} s^2 &\sim \text{Inv-Gamma}(3.0, 3.0), \\ W_{ii} &\sim \text{Inv-Gamma}(10.0, 2.0), i = 1, \dots, 9. \end{aligned}$$

Figure 5.18 shows a one-step forecast using the Negative-Binomial DGLM. The solid red line is the median of the forecast distribution with 50% probabilty intervals.

Figure 5.19 shows the residuals against time and residuals against fitted values. The residuals against fitted values plot shows heteroskedasticity, the variance increases as the values of the fitted values increases. The Negative binomial observation distribution is intended to account for this over-dispersion. Heteroskedastic time series are considered in chapter 7, however only for Gaussian observation models.

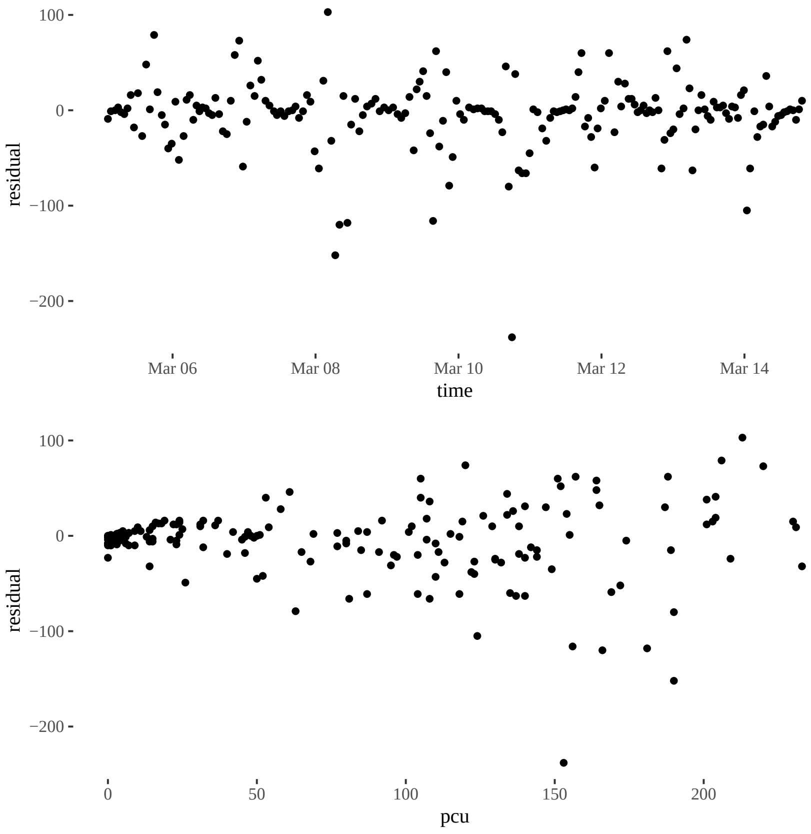


Figure 5.19: Model diagnostics for the Negative binomial state space model. Residuals against time showing some un-modelled seasonality which could be captured using higher-order harmonics, (top). Residuals against fitted values showing heteroskedasticity, the variance appears to increase over time, (bottom).

5.10 Summary

This chapter has introduced non-Gaussian state space models which can be used to model count data with various observation distributions and data with outliers using the Student-t distribution. Inference for the latent-state can be performed using the bootstrap particle filter, pseudo-marginal MCMC can be used to determine the static parameters using an unbiased estimate of the likelihood given by the particle filter.

All the inference methods which are applicable to non-Gaussian state space models (such as particle filters, pseudo-marginal MCMC etc) are also applicable to DLMs, although more efficient exact methods are available.

Numerical stability in the particle filter can be a problem, several strategies for ameliorating this are considered in section 5.2.1. A functional parallel implementation of the particle filter is considered in section 5.2.2 using Metropolis resampling.

These non-Gaussian models are more computationally intensive to fit compared to the Gaussian state space models in chapter 4, especially when using pseudo-marginal schemes to estimate the parameters. A current research avenue for non-Gaussian state space models are fast approximate inference schemes such as integrated nested Laplace approximation (INLA) [Ruiz-Cárdenas et al., 2012]. This inference scheme can quickly determine approximate posterior distributions for static parameters for time series data which is fully available.

Section 5.4 presents a conditionally Gaussian filter for the Student-t distribution using the fact that a Student-t distribution is an Inverse-Gamma mixture of Normals. This means the Kalman filter can be used to perform inference for the latent-state in a state space model with a Student-t observation distribution. This is utilised to model NO and Temperature in sections 5.8 and 5.7.

Chapter 6

Composable Markov Process Models

Observations across sensor networks are highly heterogeneous, such as weather and temperature, counts of cars and bikes or tweets from Twitter. Dynamic Linear Models (DLMs) (as introduced in chapter 4) are often applied to modelling time series data. They are discrete time, latent variable models which can be applied to a wide variety of problems, provided the data is Normally distributed and the model is linear. The Kalman filter is an analytical solution to find the distribution of the latent variables of a DLM, which can be used to perform forecasting [Kalman, 1960]. The closed form solution can be found in the case of a linear-Gaussian model because of special properties of the Gaussian distribution; the sum of two Gaussian distributions is a Gaussian distribution and a linear translation of a Gaussian distribution is still Gaussian with predictable mean and variance. However, data arising from sensor networks is highly heterogeneous and in order to model this data, non-linear models with a wide variety of observation models are needed.

In order to perform inference for non-linear systems, the extended (EKF) and later unscented Kalman filter (UKF) have been developed [Julier and Uhlmann, 1997]. The extended Kalman filter linearises at the current time step, by calculating the Jacobian of the transition and observation functions at the given time and using these in the Kalman filter equations. The extended Kalman filter becomes unstable when applied to highly non-linear problems, so the unscented transform was introduced.

Particle filters (first introduced in section 1.3.5 and expanded upon in sections 4.6 and 5.2) can explore the state space of a non-linear, non-Gaussian latent variable model, with minimal modification between models. The unscented Kalman Filter is more computationally efficient than the simulation based particle filter and can be more accurate when models are near-linear [Bellotto and Hu, 2007]. However, particle filters allow more flexibility of model choice and as such are considered when performing inference for the models presented in this chapter. As the number of particles in the particle filter is increased, the estimation error tends to zero. The same is not true of the UKF, since a limited number of sigma points are used [Simon, 2006]. So with access to more computing power, the particle filter is preferred for accurate inference.

Gordon et al. [1993] developed the theory of the bootstrap particle filter and compared the performance of the new filter to the Extended Kalman Filter (EKF). They found the accuracy of the bootstrap filter to be greatly superior to the EKF using a highly non-linear model. A target tracking model was also considered, where the bootstrap filter again outperforms the EKF.

Partially observed Markov processes (POMP) [Ionides et al., 2006] are a type of state space model. For a review of state space modelling and filtering see Doucet et al. [2001]. In a POMP model, the evolution of the latent state is governed by a continuous-time Markov process which allows modelling of irregularly spaced observations in a natural way. The observations are conditionally independent given the current value of the state.

POMP models are flexible in the choice of state-transition and observation distributions, hence they can be used to model a wide variety of processes, such as counts, proportions or strictly positive time varying data. For a full Bayesian analysis of a POMP model, the full-joint posterior distribution of the parameters and the latent state can be determined using the Particle Marginal Metropolis Hastings algorithm (PMMH) [Andrieu et al., 2010]. PMMH is an off-line algorithm which requires a batch of observations to determine the parameters. On-line filtering and forecasting can be carried out using a particle filter with pre-determined static parameters, and occasional re-training of the model can be carried out as required. On-line learning of model parameters is an active research area. On-line parameter learning [Carvalho et al., 2010a, Vieira and Wilkinson, 2016] can be used for the composable models considered in this chapter, but this is not the main focus. Here we illustrate our approach in the context of on-line state estimation.

It is useful to reuse and combine existing models when exploring new data and building more complex models. The POMP models considered in this chapter can be composed together to create new models. A software package has been written in Scala which allows users to build and compose models and perform inference using the composable particle filter and the PMMH algorithm[1]. On-line filtering and forecasting of streaming data is facilitated using Akka Streams[2] (first introduced in section 2.3.1) for Scala. Akka streams can be used to process unbounded streams of data with bounded resources, hence on-line filtering can be applied to large, live streams of data from application endpoints, databases and files.

There are other software packages for performing inference on POMP models: LibBi [Mur-

[1]https://git.io/statespace
[2]https://akka.io

ray, 2015] implements inference for general state-space models using particle Markov chain Monte Carlo methods and sequential Monte Carlo (SMC). It is optimised for parallel hardware and utilises CUDA (a parallel programming platform by NVIDIA) for GPU (Graphics Processing Unit) programming. There is also an R [R Core Team, 2015] package, POMP [King et al., 2015] which implements Frequentist and Bayesian methods of parameter inference for POMP models. Biips [Todeschini et al., 2017] is a probabilistic programming language written in C++ for SMC and particle MCMC using a similar input language to BUGS [Lunn et al., 2000a]. However, none of these packages support on-line filtering of live streaming data, or model composition.

A Hidden Markov Model (HMM) is a type of state space model with discrete time and discrete state. Factorial HMMs [Ghahramani and Jordan, 1997] have a distributed state, which can reduce the complexity of approximate parameter inference when the discrete state space is large. As mentioned by Ghahramani and Jordan, if the model structure is known to consist of loosely coupled states, then the models can be combined from constituent parts. This is similar in spirit to the composable POMP models considered in this chapter, however here the state is continuous and evolves in continuous time.

The composable POMP models presented in this chapter have been developed to analyse data in the Urban Observatory [James et al., 2014b], a grid of sensors deployed around the city of Newcastle Upon Tyne, UK. In order to demonstrate the utility of this class of composable POMP models, there is a real world example presented in Section 6.4. The example consists of irregularly observed (aggregated) Traffic Flow Data collected from the North East Combined Authority Transport Open Data [NECA, 2016].

Section 6.1 introduces POMP models as a means to model a wide variety of time series data. Section 6.2 describes the composition operator for POMP models, to enable complex models to be developed by combining simple models. Section 6.3.1 presents the bootstrap particle filter and how it can be applied to estimate the latent state of a composed model. Futher, the composed particle filter can be used to calculate an estimate of the marginal likelihood of the observations given a set of parameters. Section 6.3.2 explains how to use PMMH to determine the full-joint posterior distribution of the static parameters and latent state. The PMMH algorithm was first introduced in section 5.3.

6.1 Partially Observed Markov Processes

Streaming data arrives frequently and discretely as an observation, y, and a timestamp, t. In order to describe this time series data, a parametric probabilistic model is proposed:

$$
\begin{aligned}
Y(t_i)|\eta(t_i) &\sim \pi(y(t_i)|\eta(t_i), \psi), \\
\eta(t_i)|\theta(t_i) &= g(F_{t_i}^T \theta(t_i)), \\
\theta(t_i)|\theta(t_{i-1}) &\sim p(\theta(t_i)|\theta(t_{i-1}), \psi), \\
\theta(t_0) &\sim p(\theta(t_0)|\psi).
\end{aligned} \tag{6.1}
$$

$Y(t_i) \in \mathbb{R}$ is a scalar observation from the observation distribution, $\pi(y(t_i)|\eta(t_i), \psi)$. $\theta(t_i)$ is the (possibly multidimensional) unobserved state of the process. The state is Markovian and is governed by the transition kernel $p(\theta(t_i)|\theta(t_{i-1}), \psi)$ that we assume realisations can be generated from, but in general may not represent a tractable distribution, in the sense that we do not assume that we can evaluate the transition kernel pointwise. The prior distribution on the state is given by $p(\theta(t_0)|\psi)$. F_t is a time dependent vector, the application results in the linear predictor $\gamma(t) = F_t^T \theta(t)$, where $\gamma(t) \in \mathbb{R}$. The function g is a link-function allowing a non-linear transformation as required by the observation distribution. For example, the link-function for a Poisson distribution is $g(x) = \exp\{x\}$, since the rate of a Poisson distribution is required to be positive. ψ represents the static parameters of the model, some observation distributions have scale parameters, Markov transition kernels require parameters and the initial state, $\theta(t_0)$, requires parameters.

In order to specify the model, the observation distribution $\pi(\cdot)$, the state evolution kernel $p(\cdot)$, the linking function $g(\cdot)$, and the linear transformation vector F_t^T are assumed known and aren't learned from the data. The parameters of the observation distribution and state evolution kernel are learned from the data. Models with different specifications for the same data can be assesed using model selection techniques [Wasserman, 2000].

The model is expressed as a directed graph in Figure 6.1. Although the latent state evolves continuously, it is indicated only when there is a corresponding observation of the process.

6.1.1 Observation Model

There is a need for flexible observation distributions to model the variety of data recorded by large heterogeneous sensor networks. The scaled Exponential family models can be used for the observation model, which includes, Poisson, Bernoulli and the Gaussian distributions among others.

Following the notation of West and Harrison [1997], the exponential family of dynamic models are parameterised by $\eta(t_i)$, V and three known functions $b(Y(t_i), V)$, $S(Y(t_i))$, $a(\eta(t_i))$, the density, $p(Y(t_i)|\eta(t_i), V)$ is given by

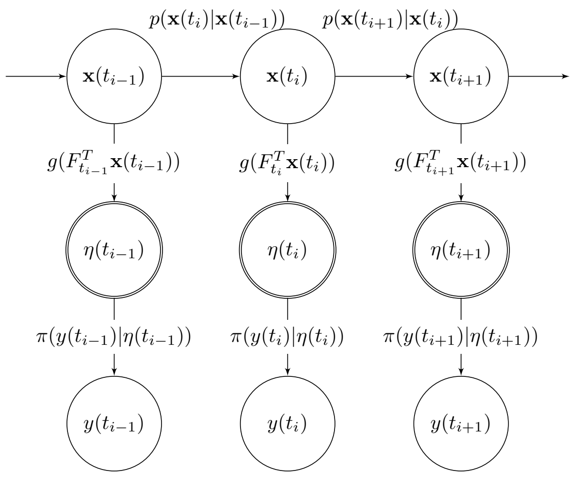

$p(\mathbf{x}(t_i)|\mathbf{x}(t_{i-1}))$
$p(\mathbf{x}(t_{i+1})|\mathbf{x}(t_i))$
$\mathbf{x}(t_{i-1})$
$\mathbf{x}(t_i)$
$\mathbf{x}(t_{i+1})$
$g(F^T_{t_{i-1}}\mathbf{x}(t_{i-1}))$
$g(F^T_{t_i}\mathbf{x}(t_i))$
$g(F^T_{t_{i+1}}\mathbf{x}(t_{i+1}))$
$\eta(t_{i-1})$
$\eta(t_i)$
$\eta(t_{i+1})$
$\pi(y(t_{i-1})|\eta(t_{i-1}))$
$\pi(y(t_i)|\eta(t_i))$
$\pi(y(t_{i+1})|\eta(t_{i+1}))$
$y(t_{i-1})$
$y(t_i)$
$y(t_{i+1})$

Figure 6.1: Representation of a POMP model as a Directed Acyclic Graph (DAG)

$$b(Y(t_i), V)\exp\left\{\frac{S(Y(t_i))\eta(t_i) - a(\eta(t_i))}{V}\right\}, \tag{6.2}$$

$\eta(t_i)$ is called the natural parameter of the distribution, $V > 0$ is the scale parameter, $S(Y(t_i))$ is the sufficient statistic, $b(Y(t_i), V)$ is the base measure and $a(\eta(t_i))$ is known as the log-partition. The model class is then a generalisation of dynamic generalised linear models (DGLM) [West and Harrison, 1997].

An extension to the POMP model class is also considered, the Log-Gaussian Cox-Process, for modelling time-to-event data. It should be noted that additional extensions to non-exponential family observation models are possible with the methods for filtering and inference presented in Section 6.3, but are not considered here.

6.1.2 Modelling Seasonal Data

Many natural phenomena feature predictable periodic changes in the rate of their process. For instance, when monitoring urban traffic, traffic flow is higher around 9am and 5pm as commuters make their way to and from the city. In order to model a periodic process, a time-dependent linear transformation vector is used:

$$F_{t_i} = \begin{pmatrix} \cos(\omega t_i) \\ \sin(\omega t_i) \\ \vdots \\ \cos(\omega h t_i) \\ \sin(\omega h t_i) \end{pmatrix},$$

where $\omega = \frac{2\pi}{P}$ is the frequency, P represents the period of the seasonality and h represents the number of harmonics required. The phase and amplitude of the waves are controlled by the value of the latent state, if the h^{th} harmonic at time t_i is given by

$$S_h = \begin{pmatrix} \cos(\omega h t_i) \\ \sin(\omega h t_i) \end{pmatrix} \cdot \begin{pmatrix} x_1(t_i) \\ x_2(t_i) \end{pmatrix},$$

the phase of the wave is $\varphi = \arctan(-x_2(t_i)/x_1(t_i))$ and the amplitude is, $A = \sqrt{x_2(t_i)^2 + x_1(t_i)^2}$. The first multiplicand of S_h is a component representing the h^{th} harmonic of F_t. The observation model can be any appropriate distribution. The full specification of the seasonal POMP model is given by (6.1) with an identity link-function, $g(x) = x$. The latent state of the seasonal model is $2h$-dimensional.

In DLMs seasonality is commonly represented using block diagonal matrices to transform the state, the blocks contain rotation matrices which are time homogeneous [West and Harrison, 1997]. This is not convenient with continuous-time models where observations can arrive irregularly.

Figure 6.2 shows a simulation from a Normal seasonal model, with a latent state evolution governed by Brownian Motion as introduced in Section 6.1.5.

6.1.3 Time to Event Data: The Log-Gaussian Cox-Process

The Log-Gaussian Cox-Process (LGCP) is an inhomogeneous Poisson process whose rate is driven by a log-Gaussian process. In an inhomogeneous Poisson process the hazard, $\lambda(t)$, varies in time according to a log-Gaussian process. For an inhomogeneous Poisson process, the total number of events in the interval $(t_0, t_n]$ is distributed as:

$$N(t_n) \sim \text{Poisson}\left(\int_{t_0}^{t_n} \lambda(s) ds\right),$$

where the the cumulative hazard can be written as $\Lambda(t_n) = \int_{t_0}^{t_n} \lambda(s) ds$. The cumulative distribution function of the Log-Gaussian Cox-Process is $1 - \exp\{-\Lambda(t_n)\}$. The conditional density is then,

$$\pi(t_n | \lambda(t_n), \Lambda(t_n)) = \lambda(t_n) \exp\{-\Lambda(t_n)\}.$$

The general form of the LGCP POMP model is given by,

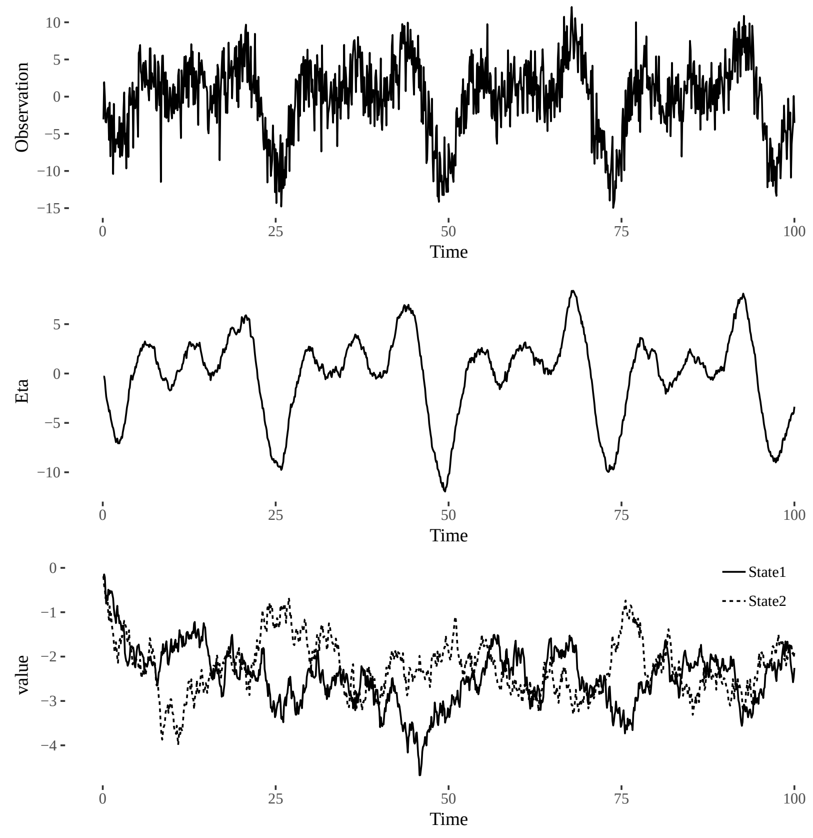


Figure 6.2: (Top) Simulated values from a seasonal model with Normal observations, with observation variance $V = 1.0$, observed discretely at integer time points. (Middle) Transformed State, mean of the Normal observations. (Bottom) The Ornstein-Uhlenbeck latent state with parameter values $\phi = 0.1$, $\mu = 0.1$ and $\sigma = 0.3$ and the initial state drawn from $\mathcal{N}(0.0, 1.0)$.

$$\begin{aligned}
t_i|\lambda(t_i) &\sim \pi(t|\lambda(t_i), \Lambda(t_i)), \\
\begin{pmatrix} \lambda(t_i) \\ \mathrm{d}\Lambda(t_i) \end{pmatrix} &= \begin{pmatrix} \exp\{x(t_i)\} \\ \lambda(t_i)\mathrm{d}t \end{pmatrix}, \\
X(t_i)|x(t_{i-1}) &\sim p(x(t_i)|x(t_{i-1}), \psi), \\
x(t_0) &\sim p(x(t_0)|\psi),
\end{aligned}$$

where $p(x(t_i)|x(t_{i-1}), \psi)$ is the transition kernel of a continuous time Gaussian Markov process. Figure 6.3 shows a simulation from the Log-Gaussian Cox-Process, using an approximate simulation algorithm presented in 17.

Algorithm 17: Algorithm to simulate the Log-Gaussian Cox-Process.

```
   **Result:** Return the set of events $e_k = \{t_s, \dots, t_k\}$ in the interval $[t_0, t_N]$
1  Given the rate $\lambda(t) = \exp(\psi(t))$ has been simulated on a fine grid on the interval $[t_0, t_N]$;
2  Set $U_\lambda = \max_{t \in [t_0, t_n]} \lambda(t)$;
3  while $t_k < t_N$ do
4      Set $k = 1$;
5      Sample $t' \sim \text{Exponential}(U_\lambda)$;
6      Set $t_k = t_{k-1} + t'$;
7      Sample $u \sim U[0, 1]$;
8      if $u \leq \lambda(t_k)/U_\lambda$ then
9          Add the event time to the set of events $e_k = \{e_{k-1}, t_k\}$;
10     end
11     Set $k = k + 1$;
12 end
```

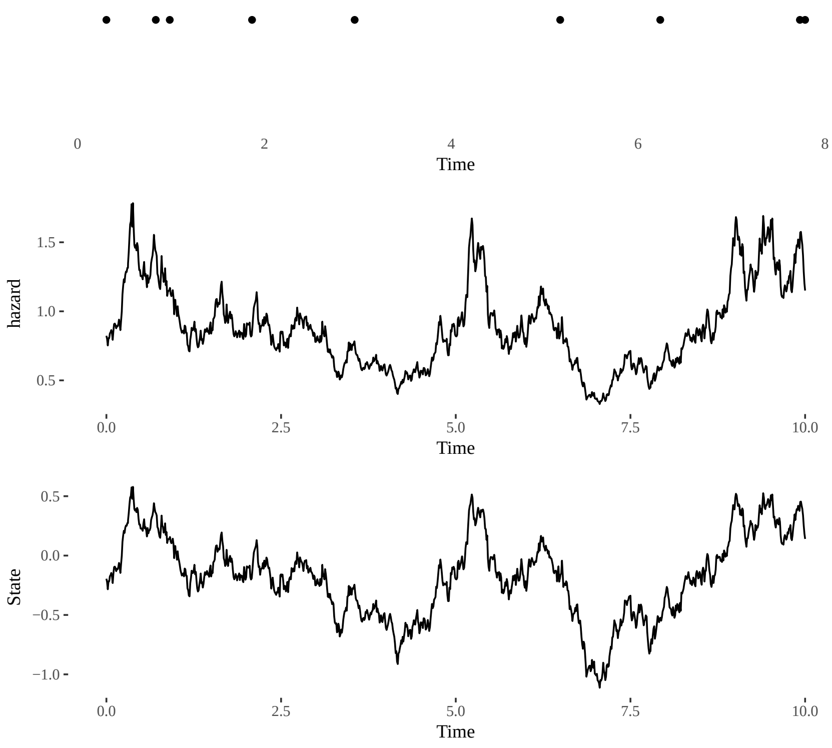


Figure 6.3: (Top) Simulated event times from the Log-Gaussian Cox-Process. (Middle) The unobserved hazard. (Bottom) The unobserved state evolving according to the Ornstein-Uhlenbeck Process with mean $\mu = 0.1$, autoregressive parameter $\phi = 0.4$ and volatility $\sigma = 0.5$, the initial distribution is $\mathcal{N}(\mu, 1.0)$

6.1.4 Simulating synthetic data from a POMP model

In order to perform forecasting and interpolation, a method must be developed to forward simulate from POMP models. After determining the parameters for the model, as in section 6.3.2, forward simulation of the model beyond observed values can be used to predict future observations along with the uncertainty of the predictions. In the case of the traffic data in section 6.4, forecasting can help with route planning and diversions. In addition, forward simulation can also be useful to generate synthetic data to test inferential algorithms. Algorithm 18 illustrates how to simulate from a POMP model.

Algorithm 18: Algorithm to simulate from a POMP model.

```
  Result: Return the pairs of observations and latent-state
          $\{(\theta(t_1), y(t_1)), \dots, (\theta(t_N), y(t_N))\}$
1 Given a list of times to observe the process $t_0, t_1, \dots, t_N$;
2 Sample $\theta(t_0) \sim p(\theta(t_0)|\psi)$;
3 for $i$ in $1 : N$ do
4     Calculate $\delta t = t_i - t_{i-1}$;
5     Sample $\theta(t_i) \sim p(\theta(t_i)|\theta(t_{i-1}), \psi)$;
6     Calculate $\gamma(t_i) = F_{t_i}^T \theta(t_i)$;
7     Calculate $\eta(t_i) = g(\gamma(t_i))$;
8     Sample $y(t_i) \sim \pi(y(t_i)|\eta(t_i), \psi)$;
9 end
```

6.1.5 The Latent State: Diffusion Processes

The latent state evolves according to a continuous time Markov process. Any Markov process can be used to represent the state; however in this chapter, the focus is on Itô diffusion processes (introduced in section 4.7). Diffusion processes are represented by solutions to stochastic differential equations (SDE), (see Oksendal [2013] for a detailed treatment). The continuous-time evolution of the latent space is governed by a time homogeneous SDE of the form.

$$\mathrm{d}\theta(t) = \mu(\theta(t))\mathrm{d}t + \sigma(\theta(t))\mathrm{d}B(t). \tag{6.3}$$

Generally, $\theta \in \mathbb{R}^n$ and $\mu(\cdot) : \mathbb{R}^n \to \mathbb{R}^n$ is referred to as the drift coefficient, $\sigma(\cdot) : \mathbb{R}^n \to \mathbb{R}^{n \times m}$ is the diffusion coefficient and $B(t)$ is an $m \times 1$ Brownian motion.

If an SDE doesn't have an analytic solution, it can be simulated approximately using the Euler-Maruyama approximation or higher-order schemes such as the Milstein method [Kloeden and Platen, 1992]. The Euler-Maruyama approximation gives an approximate solution to an SDE, as a Markov chain. Start with a general SDE for a diffusion process as in Equation 6.3.

The interval of interest, $[t_0, t_N]$ is partitioned into N even sized sub-intervals, of size $\Delta t = \frac{t_N - t_0}{N}$. A value is defined for θ_0, the initial value of the Markov chain, then for $i = 1, \dots, N$

$$\theta_i = \theta_{i-1} + \mu(\theta_{i-1})\Delta t + \sigma(\theta_{i-1})\Delta \mathbf{W}_{i-1}, \tag{6.4}$$

where $\Delta \mathbf{W}_n \sim \text{MVN}(\mathbf{0}, I_m \Delta t)$, are independent and identically distributed Multivariate Normal random variables with mean zero and variance $I_m \Delta t$, where I_m is the m-dimensional identity matrix. The approximate transition density for any diffusion process, for sufficiently small Δt, can then be written as:

$$\theta_i | \theta_{i-1} \sim \text{MVN}\left(\theta_{i-1} + \mu(\theta_{i-1})\Delta t, \sigma(\theta_{i-1})\sigma(\theta_{i-1})^T \Delta t\right).$$

Equation 6.4 is the Euler-Maruyama discretisation. The transition to the next state only depends on the current value, hence the Euler-Maruyama approximation scheme for stochastic differential equations is a Markov Process. Analytic solutions of diffusion processes are also Markovian, necessarily. Figure 6.4 shows eight independent simulations from the Ornstein-Uhlenbeck process using the exact transition kernel and the same fixed parameter values.

6.2 Composing Models

In order to model real-world data, it is convenient to compose simple model components to form a more complex model, representative of the real-world measurements. Consider the traffic data example presented in Section 6.4. The data consists of readings of passenger car units (PCU) in a given time interval, PCUs are positive integer values representing the size of a vehicle, the observation model is chosen to be the Negative Binomial distribution. The traffic data displays daily and weekly seasonal cycles, so the transformed latent state, $\eta(t)$ must vary periodically with time. In order to account for the two periods of seasonality with a Poisson observation model, a seasonal-Poisson model with drift is formed by the composition of a Poisson model with a one-dimensional latent state and two seasonal models. One seasonal model has a weekly period and the other has a daily period.

The composable POMP model can be thought of as a partially observed (uncoupled) multivariate SDE, however model specification has been simplified using model composition.

Consider a POMP model, with its associated functions, distributions and latent state indexed by j, where j represents the model number in the composition indexing from the left. Where $\psi^{(j)}$ represents the parameters of the j^{th} model; we have that

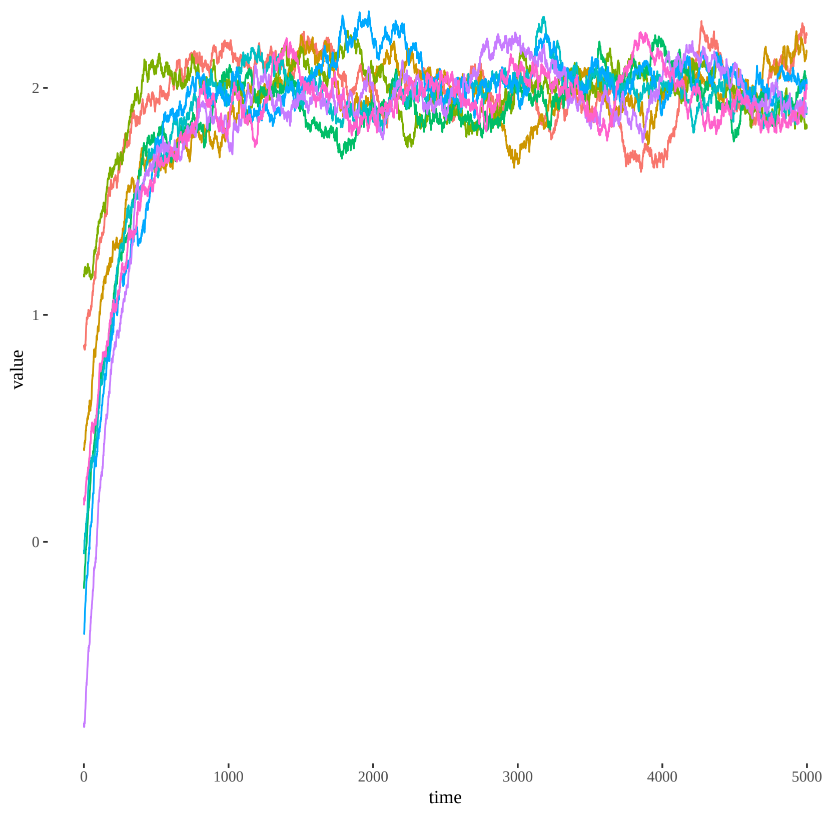


Figure 6.4: Eight simulations from the Ornstein-Uhlenbeck process with parameters $\phi = 0.4, \mu = 2.0, \sigma = 0.1$ with initial state $\mathcal{N}(0, 1)$

$$\begin{aligned}
Y(t_i)|\eta^{(j)}(t_i) &\sim \pi_j(y(t_i)|\eta^{(j)}(t_i), \psi^{(j)}), \\
\eta^{(j)}(t_i)|\theta^{(j)}(t_i) &= g_j(F_{t_i}^{(j)T}\theta^{(j)}(t_i)), \\
\theta^{(j)}(t_i)|\theta^{(j)}(t_{i-1}) &\sim p_j(\theta^{(j)}(t_i)|\theta^{(j)}(t_{i-1}), \psi^{(j)}), \\
\theta^{(j)}(t_0) &\sim p_j(\theta^{(j)}(t_0)|\psi^{(j)}).
\end{aligned} \tag{6.5}$$

Now define the composition of model $\mathcal{M}_1$ and $\mathcal{M}_2$ as $\mathcal{M}_3 = \mathcal{M}_1 \star \mathcal{M}_2$. By convention, the observation model will be that of the left-hand model, $\mathcal{M}_1$ and the observation model of the right-hand model will be discarded. As such, the non-linear linking-function must be that of the left-hand model, $g_1 : \mathbb{R} \to \mathbb{R}$. The linking-function ensures the state is correctly transformed into the parameter space of the observation distribution.

In order to compose the latent state, the initial state vectors are concatenated:

$$\theta^{(3)}(t_o) \sim \begin{pmatrix} p(\theta^{(1)}(t_0)|\psi^{(1)}) \\ p(\theta^{(2)}(t_0)|\psi^{(2)}) \end{pmatrix}.$$

The composed model's transition density for the state is given by

$$\begin{aligned} p_3(\theta(t_i)|\theta(t_{i-1}), \psi) &= p_3(\theta^{(1)}(t_i), \theta^{(2)}(t_i)|\theta^{(1)}(t_{i-1}), \theta^{(2)}(t_{i-1}), \psi^{(1)}, \psi^{(2)}) \\ &= p_1(\theta^{(1)}(t_i)|\theta^{(1)}(t_{i-1}), \psi^{(1)})p_2(\theta^{(2)}(t_i)|\theta^{(2)}(t_{i-1}), \psi^{(2)}), \end{aligned}$$

the joint transition density is a product of the marginal densities since the latent state $\theta(t_i)^{(1)}$ is independent of $\theta(t_i)^{(2)}$ for all $i = 1, \dots, N$. The linear deterministic transformation-vectors, $F^{(j)}(t)$, $j = \{1, 2\}$, are vertically concatenated

$$F^{(3)}(t) = \begin{bmatrix} F^{(1)}(t) \\ F^{(2)}(t) \end{bmatrix}.$$

The vector dot-product with the composed latent state is then computed, such that $F_t^{(3)T}\theta(t) \equiv F_t^{(1)T}\theta^{(1)}(t) + F_t^{(2)T}\theta^{(2)}(t)$. The dot product, $\gamma(t) = F_t^T\theta(t)$ results in a one-dimensional state, $\gamma(t) \in \mathbb{R}$. The full composed model, $\mathcal{M}_3$ can then be expressed as follows

$$\begin{aligned} Y(t_i)|\eta(t_i) &\sim \pi_1(y(t_i)|\eta(t_i)), \\ \eta(t_i)|\theta(t_i) &= g_1(F_{t_i}^{(3)T}\theta(t_i)) \\ \theta(t_i)|\theta(t_{i-1}) &\sim p(\theta^{(1)}(t_i)|\theta^{(1)}(t_{i-1}), \psi^{(1)})p(\theta^{(2)}(t_i)|\theta^{(2)}(t_{i-1}), \psi^{(2)}), \\ \theta(t_0) &\sim \begin{pmatrix} p(\theta^{(1)}(t_0)|\psi^{(1)}) \\ p(\theta^{(2)}(t_0)|\psi^{(2)}) \end{pmatrix}. \end{aligned}$$

The POMP models, along with the closed, associative binary composition operator, form a semigroup. In order to compose three or more models the binary operator is applied pairwise, remembering that it is not commutative:

$$\begin{aligned} \mathcal{M}_5 &= (\mathcal{M}_1 \star \mathcal{M}_2) \star \mathcal{M}_4, \\ &= \mathcal{M}_3 \star \mathcal{M}_4, \\ &= \mathcal{M}_1 \star (\mathcal{M}_2 \star \mathcal{M}_4). \end{aligned} \tag{6.6}$$

In the first line of Equation 6.6, we compose $\mathcal{M}_1$ and $\mathcal{M}_2$ to form $\mathcal{M}_3$ as in the example

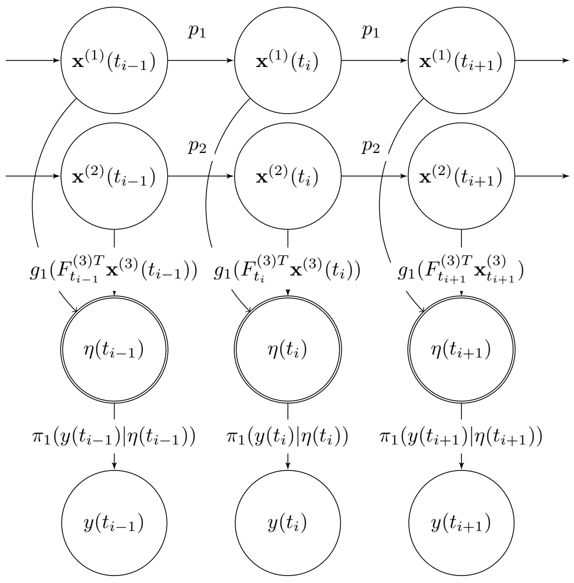

$\mathbf{x}^{(1)}(t_{i-1})$
p_1
$\mathbf{x}^{(1)}(t_i)$
p_1
$\mathbf{x}^{(1)}(t_{i+1})$
$\mathbf{x}^{(2)}(t_{i-1})$
p_2
$\mathbf{x}^{(2)}(t_i)$
p_2
$\mathbf{x}^{(2)}(t_{i+1})$
$g_1(F^{(3)T}_{t_{i-1}}\mathbf{x}^{(3)}(t_{i-1}))$
$g_1(F^{(3)T}_{t_i}\mathbf{x}^{(3)}(t_i))$
$g_1(F^{(3)T}_{t_{i+1}}\mathbf{x}^{(3)}_{t_{i+1}})$
$\eta(t_{i-1})$
$\eta(t_i)$
$\eta(t_{i+1})$
$\pi_1(y(t_{i-1})|\eta(t_{i-1}))$
$\pi_1(y(t_i)|\eta(t_i))$
$\pi_1(y(t_{i+1})|\eta(t_{i+1}))$
$y(t_{i-1})$
$y(t_i)$
$y(t_{i+1})$

Figure 6.5: A directed acyclic graph representing the composition of two models, the latent state is represented as separate, and advancing according to each model components transition kernel: p_i

above, then we proceed to compose $\mathcal{M}_3$ and $\mathcal{M}_4$ to form $\mathcal{M}_5$.

The semigroup structure exhibited by the composable model class is the most basic of algebraic properties, but it is nevertheless very powerful. In particular, it allows the creation of very large, complex models from simpler components in a recursive, hierarchical fashion. Many functional programming languages provide support for datatypes with semigroup structure, and provide convenient syntax for composition. Figure 6.5 shows a directed acyclic graph of a composed model.

6.2.1 Computing With Composed Models

A model is defined in the Scala language using:

1. A stochastic observation function:
   ```scala
   val observation : Eta => Rand[Observation]
   ```
2. A deterministic, non-linear linking function:
   ```scala
   val link: Gamma => Eta
   ```
3. A deterministic linear transformation function:
   ```scala
   val f: (State, Time) => Gamma
   ```
4. A stochastic differential equation representing the latent state:
   ```scala
   val sde: SDE
   ```

5. A likelihood function:
   ```
   val dataLikelihood: (Eta, Observation) =>
   LogLikelihood
   ```

`observation` is a function from the transformed latent state $\eta(t)$ to a distribution over the observations, $\pi(y(t)|\eta(t))$. The distribution, `Rand[Observation]` can be sampled from by calling the method `draw`.

The linking function ensures the value of $\eta(t)$ is appropriate for the observation distribution. The function `f` is used to perform a linear transformation of the latent state. The latent state is represented by an `SDE` object, `sde` contains `stepFunction`, a function from `State` and `TimeIncrement` to the next `Rand[State]` and `initialState` which returns a distribution over states, which can be sampled from.

6.2.2 A Binary Operation for Composing Models

In order to compose models, define an associative binary operation called `combine` which combines two models at a time. `combine` accepts two un-parameterised models, (parameterisation is described in Section 6.2.3) `mod1` and `mod2` and returns a third model representing the composition of the two models. Along with the `combine` operator, the models form a semigroup, since the `combine` operation is associative and closed.

Further, an identity model, e, can be defined such that for any model $\mathcal{M}_a$, $e \star \mathcal{M}_a = \mathcal{M}_a = \mathcal{M}_a \star e$. Un-parameterised models now form a monoid.

In order to define the `combine` operation, firstly consider combining the stochastic differential equations representing the evolution of the latent-state of two models: A binary tree is used in order to represent the latent state of a model, this is depicted in Figure 6.6. For a single sde `Sde`, the latent-state is a `Leaf`. In order to represent the state of two models, the state of each model is combined into a `Branch` where the left and right branches correspond to the latent state of each model in the composition. A binary tree is defined as a generalised algebraic data type in Scala as:

```
sealed trait Tree[A]
case class Leaf[A](value: A) extends Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]
case object EmptyLeaf extends Tree[Nothing]
```

`Leaf` and `Branch` both extend the `Tree` trait, when a function accepts a parameter of type `Tree` it can be either a `Leaf` or a `Branch`. Pattern matching is used in order to decompose the tree and perform functions on the values contained in the leaf nodes. The value `A` is a placeholder value, stating that the values at the `Leaf` of the `Tree` can be anything. In the case of the state tree, it is represented as a `DenseVector[Double]`. The state tree is defined as a type alias `type State = Tree[DenseVector[Double]]`.

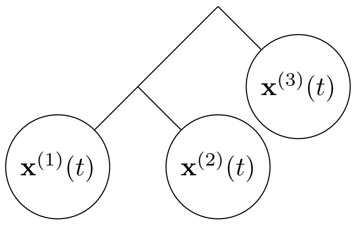


Figure 6.6: Illustration of the latent state as a binary tree. This composed model state consists of the combined states of three models.

Consider advancing the state space of a composed SDE consisting of `sde1` and `sde2`, by drawing from the transition kernel conditional on the current state using the `stepFunction`. Each SDE in the combined SDE has a corresponding `Leaf` state and `stepFunction`, each model's `stepFunction` must act on the corresponding state. To advance the state of a composed model, the left-hand (correspondingly right-hand) model's `stepFunction` is applied to the left-hand (right-hand) model's state space:

```
def stepFunction(dt: TimeIncrement, s: State): Rand[State] = s match {
  case Branch(l, r) => for {
    ls <- sde1.stepFunction(dt)(l)
    rs <- sde2.stepFunction(dt)(r)
} yield Tree.branch(ls, rs)
```

In order to simulate from an SDE, there must be an initial distribution for the state, this is given by the `initialState` function. Adding two models will result in a `Branch` for the initial state:

```
val initialState = for {
  l <- sde1.initialState
  r <- sde2.initialState
} yield Tree.branch(l, r)
```

Now that the SDE representing the latent-state can be composed, the model class which contains a definition of an SDE must be composed. The application of the vector F_t is performed using the function `def f: (State, Time) => Gamma`. For two models, this function expects a `Branch`, which can be decomposed using pattern matching:

```
def f(s: State, t: Time) = s match {
  case Branch(ls, rs) =>
  mod1.f(ls, t) + mod2.f(rs, t)
}
```

The observation distribution is taken to be that of the left-hand model in the composition, `val observation = mod1.observation`, hence the linking function must also be that of the left-hand model `val link = mod1.link`.

6.2.3 Parameterising the Model

The parameters of a model given in Equation 6.1 are that of the Markov transition for the state space, $p(\theta(t_i)|\theta(t_{i-1}), \psi)$, the parameters of the initial system state $\theta(t_0) \sim p(\theta(t_0)|\psi)$ and any additional parameters for the observation distribution. An un-parameterised model is a function from `Parameters => Model`, which can then be supplied with appropriate parameters to form a `Model`.

When composing two models, a set of parameters for each model must be combined into a new set of parameters for the new model. As with the state-space, it is natural to represent the parameters of a composed model as a binary tree. A single model is parameterised by a `Leaf` containing a `ParamNode`. A `ParamNode` contains the `sdeParameters` and optional scale parameter for the observation distribution, `scale: Option[Double]`. When composing parameters, the two `Leafs` will be combined into a single `Branch` just like when composing models.

Parameters must be combined in the same order as models are combined, ie. combining parameters is not commutative.

In order to combine parameter trees, an instance of the Monoid type class is defined. A monoid is a set with an associative, closed `combine` operation and an `empty` which is an identity element with respect to the `combine` operator. Listing 6.2.3 explains how to define a new monoid for the binary `Tree`.

```scala
val treeMonoid = new Monoid[Tree] {
  def combine[A](x: Tree[A], y: Tree[A]): Tree[A] = Tree.branch(x, y)
  def empty: Tree[Nothing] = Tree.empty
}
```

Upon defining a type class with the `combine` operator, cats provides an infix operator, `|+|`, which is an alias for `combine`.

```scala
val (p, p1, p2) = (Leaf(.), Leaf(.), Leaf(.))
val combParams = p |+| p1 |+| p2
```

This is equivalent to constructing the parameters as nested `Branchs`:

```scala
val combParams = Branch(Branch(p, p1), p2)
```

Similar methods are used to parameterise the observation distribution, the likelihood function and the initial state distribution. Once a monoid is defined for the unparameterised models, the infix notation, `|+|`, is available to use.

Once the un-parameterised model and the parameters have been composed, the un-parameterised model can be applied as a function to the parameters as illustrated in the example in section 6.2.4.

6.2.4 Example: A Seasonal-Poisson Model

To illustrate model composition, consider a Poisson model with a seasonally-varying time-dependent rate parameter, $\lambda(t)$. This model is the composition of two models, a single Poisson model, $\mathcal{M}_1$, and a single seasonal model, $\mathcal{M}_2$, to make $\mathcal{M}_3 = \mathcal{M}_1 \star \mathcal{M}_2$. This model could represent the flow of traffic through a city, as in the example in Section 6.4.

The Poisson model, $\mathcal{M}_1$, has a 1-dimensional latent state, which evolves according to generalised Brownian motion. The linking function is the exponential function, $g(x) = \exp(x)$, and the linear transformation vector is one, $F_{t_i}^{(1)} = 1$. The Poisson model is given by:

$$\begin{aligned}
N(t_i) &\sim \text{Poisson}(\lambda(t_i)), \\
\lambda(t_i)|x(t_i) &= \exp\{x(t_i)\}, \\
\mathrm{d}X(t_i) &= \mu^{(1)}\mathrm{d}t + \sigma \mathrm{d}W^{(1)}(t_i).
\end{aligned}$$

The latent state of the seasonal model, $\mathcal{M}_2$, has a dimension of $2h$, where h represents the number of harmonics. Generalised Brownian motion is used to represent the time evolution of the $2h$-dimensional latent state, hence the drift coefficient $\boldsymbol{\mu}^{(2)}$ is $2h \times 1$ vector and the diffusion coefficient, Σ, is a $2h \times 2h$ diagonal matrix. The standard seasonal model has the Normal distribution as the observation distribution, and the linking function is the identity function:

$$\begin{aligned}
N(t_i)|\eta(t_i) &\sim \mathcal{N}(\eta(t_i), \sigma^2), \\
\eta(t_i)|\theta(t_i) &= \mathbf{F}_{t_i}^{(2)T}\theta(t_i), \\
\mathrm{d}\theta(t_i) &= \boldsymbol{\mu}^{(2)}\mathrm{d}t + \Sigma \mathrm{d}\mathbf{B}^{(2)}(t_i)
\end{aligned}$$

The vector $\mathbf{F}_{t_i}^{(2)}$ is a $2h \times 1$ vector of fourier components:

$$F_{t_i}^{(2)} = \begin{pmatrix} \cos \omega t \\ \sin \omega t \\ \cos 2\omega t \\ \sin 2\omega t \\ \vdots \\ \cos h\omega t \\ \sin h\omega t \end{pmatrix}, \tag{6.7}$$

$\omega = 2\pi/P$ is the frequency and P is the period of the seasonality. In order to model a daily seasonality, $P = 24$, if the observation timestamps are in hour units.

In the composed model, the observation model of $\mathcal{M}_1$ is chosen, in this case the Poisson distribution. This necessitates the left-hand linking function ($g(x) = \exp(x)$) since the rate parameter of the Poisson distribution is greater than zero, $\lambda(t) > 0$. The full composed model can be expressed as

$$\begin{aligned} N(t_i)|\lambda(t_i) &\sim \text{Poisson}(\lambda(t_i)), \\ \lambda(t_i)|\theta(t) &= \exp\{F_{t_i}^{(3)T}\theta(t_i)\}, \\ \mathrm{d}\theta(t_i) &= \boldsymbol{\mu}\mathrm{d}t + \begin{pmatrix} \sigma & 0 \\ 0 & \Sigma \end{pmatrix} \mathrm{d}\mathbf{W}(t_i). \end{aligned}$$

The system state is $n = 2h + 1$ dimensional, hence the drift coefficient $\boldsymbol{\mu}$ is an $n \times 1$ dimensional vector and the diffusion coefficient is an $n \times n$ diagonal matrix. Components of $B^{(1)}$ and $\mathbf{B}^{(2)}$ are independent. The linear transformation vector $\mathbf{F}_{t_i}^{(3)}$ is a time-dependent, $n \times 1$ vector, which is the concatenation of the two $\mathbf{F}_{t_i}^{(j)}, j = 1, 2$, vectors:

$$\mathbf{F}_{t_i}^{(3)} = \begin{pmatrix} 1 \\ \cos \omega t \\ \sin \omega t \\ \vdots \\ \cos h\omega t \\ \sin h\omega t \end{pmatrix}. \tag{6.8}$$

Composed Model in Scala

In order to model complex phenomena, the models can be composed. This section proposes a seasonal model with a Negative binomial observation distribution. Define a model with an `sde`, an object which contains a stochastic differential equation including a `stepFunction: (TimeIncrement, State) => Rand[State]` and the definition of the initial state `initialState: Rand[State]`.

```scala
val sde = Sde.brownianMotion(1)
val negbinMod = Model.negativeBinomial(sde)
val sdeParameters = SdeParameter.
  brownianParameter(0.0)(1.0)(0.01)
val negbinParam = Parameters(Some(2.0, sdeParameters))
```

Each model is parameterised by the latent-state, allowing flexibility in the choice of SDE controlling the evolution of the latent state. The `Sde` accepts one parameter - the dimension. If a dimension is specified which is different to that of the parameters then the parameter values will be cycled for each dimension required. In the following listing an 8-dimensional OU latent-state is specified each with eight different values for the mean, μ. The other parameters, m_0, C_0, σ and ϕ are replicated until the dimension is satisfied. `Parameter` defines a `Leaf` containing a `ParamNode` and contains the scale parameter for the Negative Binomial distribution and the parameters for the diffusion process representing the latent state. Now define a single seasonal model:

```scala
val sde2 = Sde.ouProcess(8)
val seasonalMod = Model.seasonal(period = 24, harmonics = 3, sde2)
val sde2Parameters = SdeParameter.
  ouParameter(0.0)(1.0)(0.3)(1.5, 1.5, 1.0, 1.0, 1.5, 1.5, 0.1, 0.1)(0.1)
val seasonalParam = Parameters(None, sde2Parameters)
```

The seasonal model has a Gaussian observation distribution. In order to use the seasonal model by itself the observation noise (standard deviation) for the Gaussian observation model must be specified in the scale parameter (currently set as None). The infix notation `|+|` is available on the unparameterised model and the parameters because both define a semigroup and hence have a binary `combine` function (described in detail in section 6.2.1). Note that parameters are stored unconstrained, for strictly positive parameters the logarithm is stored and the logistic function is used to transform bounded parameters. Now the parameters and model can be composed

```scala
val combinedParams = negbinParam |+| seasonalParam
val combinedModel = negbinMod |+| seasonalMod
val parameterisedModel = combinedModel(combinedParams)
```

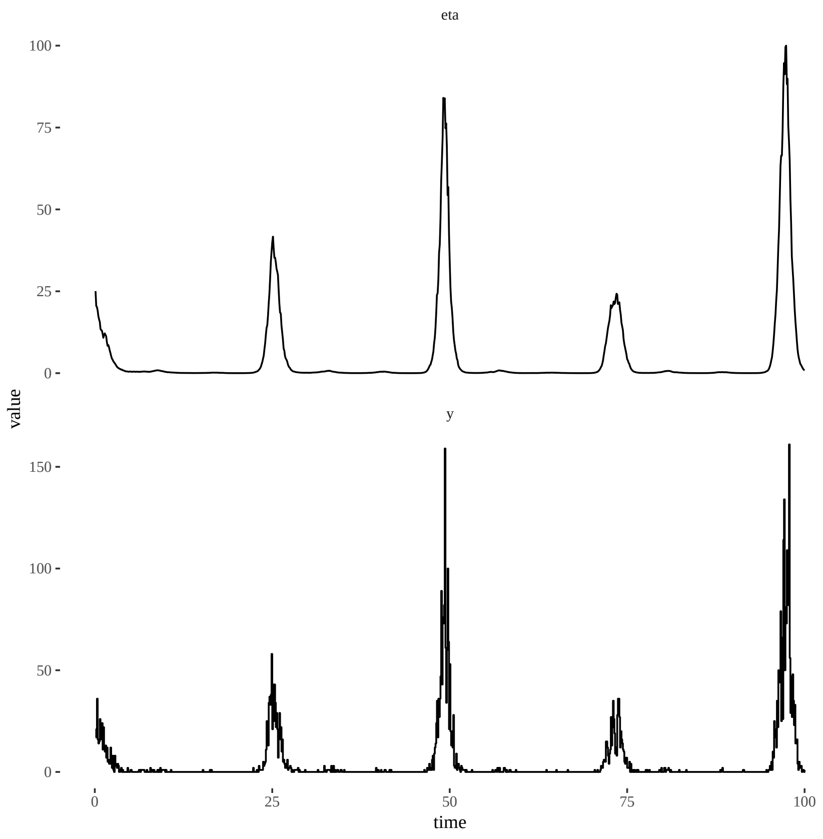


Figure 6.7: Simulation from a composed negative Binomial model with scale parameter $r = 2.0$ and composed state space with a Brownian motion state space with $\sigma = 0.01$ and Ornstein-Uhlenbeck process with parameters $\phi = 0.3$, $\mu = (1.5, 1.5, 1, 1, 1.5, 1.5, 0.1, 0.1)$ and $\sigma = 0.1$. Transformed latent state $\eta(t)$ representing the mean of the Negative Binomial Observation distribution, (top). Observations from the seasonal Negative Binomial POMP model, (bottom).

The `parameterisedModel` can be simulated from, it can be fit to observed data using the PMMH algorithm, and a particle filter and can used for on-line filtering as described in Section 6.3. Figure 6.7 shows a simulation from the composed Negative Binomial model developed in this section, the observation distribution is identical to that presented in section 5.1.1.

6.3 Statistical Inference

Consider a POMP model of the form in Equation 6.1, observed at discrete times, $\{t_1, t_2, \dots, t_N\}$. The joint distribution of all the random variables in the model can be written as:

$$\begin{aligned}
&p(\theta(t_{0:N}), y(t_{1:N}), \psi) \\
&= p(\theta(t_{0:N}), \psi)\pi(y(t_{1:N})|g(F^T_{t_{0:N}}\theta(t_{0:N})), \psi) \\
&= p(\psi)p(\theta(t_0)|\psi)\left[\prod_{i=1}^{N} p(\theta(t_i)|\theta(t_{i-1}), \psi)\pi(y(t_i)|\eta(t_i), \psi)\right].
\end{aligned}$$

Since the latent state is a Markov process and the observations $y(t_{1:N})$ are conditionally independent given the latent state, the joint distribution can be written as the product of the distribution over the parameters, $p(\psi)$, the initial distribution of the latent state $p(\theta(t_0)|\psi)$, the transition density, $p(\theta(t_i)|\theta(t_{i-1}), \psi)$ and the conditional density for the observations $\pi(y(t_i)|\eta(t_i), \psi)$.

In general the full-joint posterior distribution of the parameters and state is analytically intractable. The particle marginal Metropolis-Hastings algorithm, presented in Section 5.3, can be used to determine the joint-posterior of the parameters and state, $p(\theta(t_{0:N}), \psi|y(t_{1:N}))$. The PMMH utilises the pseudo-marginal likelihood estimated using the composable particle filter presented in Section 6.3.1.

6.3.1 On-line State Estimation using the Composable Particle Filter

The bootstrap particle filter [Gordon et al., 1993] can be used to estimate the latent state of a POMP model. The latent state is a Markov process with an associated transition kernel $\theta(t_i)|\theta(t_{i-1}) \sim p(\theta(t_i)|\theta(t_{i-1}))$, from which realisations can be sampled. The transition kernel can be any continuous time Markov process, but in this chapter only diffusion processes represented by SDEs are considered. The transition kernel can either be an exact solution of an SDE or simulated on a fine grid using the Euler-Maruyama method, presented in Section 6.1.5. The process is observed through an observation model $Y(t_i)|\eta(t_i) \sim \pi(y(t_i)|\eta(t_i))$.

A particle filter is used to estimate the unobserved system state by approximating the filtering distribution, $p(\theta(t_i)|y(t_{1:i}))$. The algorithm to determine an empirical approximation of the latent state of a composable POMP model at each observation time is summarised in algorithm 19

The average of the un-normalised weights at each time point gives an estimate of the marginal likelihood of the current data point given the data observed so far:

Algorithm 19: Bootstrap particle filter for composable POMP models.

```
Result: Return $\{\theta^{(j)}(t_i) : j = 1, \dots, M\}, i = 1 : N$ and the pseudo log-likelihood
1  Given observations $y(t_{1:N})$;
2  Simulate from the prior distribution $\theta^{(j)}(t_0) \sim p(\theta(t_0))$, for $j = 1, \dots, M$;
3  Initialise the log-likelihood $\ell(t_0) = 0$;
4  for i in 1:N do
5      for j in 1:M do
6          Advance the state particles to the time of the next observation
             $\theta^{(j)}(t_i) \sim p(\theta(t_i)|\theta^{(j)}(t_{i-1}))$;
7          Transform each particle appropriately as required by the observation
             model, $\eta(t_i)^{(j)} = g(F_{t_i}^T \theta(t_i)^{(j)})$;
8          Calculate the conditional likelihood for each particle
             $w^{*(j)}(t_i) \sim p(y(t_i)|\eta^{(j)}(t_i))$;
9      end
10     Increment the log-likelihood $\ell(t_i) = \ell(t_{i-1}) + \frac{1}{M}\sum_{k=1}^{M} w^{*(j)}(t_i)$;
11     Normalise the weights, $w(t_i)^{(k)} = \frac{w^*(t_i)^{(k)}}{\sum_{j=1}^{N} w^*(t_i)^{(j)}}$;
12     Resample the particles with replacement such that the probability of including
         particle $j$ is proportional to $w^{(j)}(t_i)$;
13 end
```

$$\hat{p}_\psi(y(t_i)|y(t_{1:i-1})) = \frac{1}{N}\sum_{j=1}^{N} w^*(t_i)^j.$$

The estimate of the likelihood of the full path is given by:

$$\hat{p}_\psi(y(t_{1:M})) = \hat{p}(y(t_1))\prod_{i=2}^{M}\hat{p}(y(t_i)|y(t_{1:i-1})).$$

The estimated marginal likelihood is consistent, meaning that as the number of particles are increased, $N \to \infty$, then the estimate converges in probability to the true value. The estimate is also unbiased, meaning that the expectation of the estimate is equal to the true value, $\mathbb{E}(\hat{p}_{\psi^*}(y(t_{1:M}))) = p_{\psi^*}(y(t_{1:M}))$ where the expectation is taken over all the random variables generated in the particle filter; see Del Moral [2004] for a proof. This marginal likelihood is used in the PMMH algorithm discussed in Section 5.3.

In practice, the bootstrap particle filter can be easily parallelised, advancing the state and calculating the likelihood are naturally parallel. Multinomial resampling (as described above) requires communication between multiple threads of execution, in order to sum the value of the weights. Other resampling schemes, such as stratified and systematic resampling are more amenable to parallelisation, see section 5.2.2 for a parallel implementation of the particle filter. Figure 6.8 shows the latent-state, $\eta(t)$, of a simulation from the com-

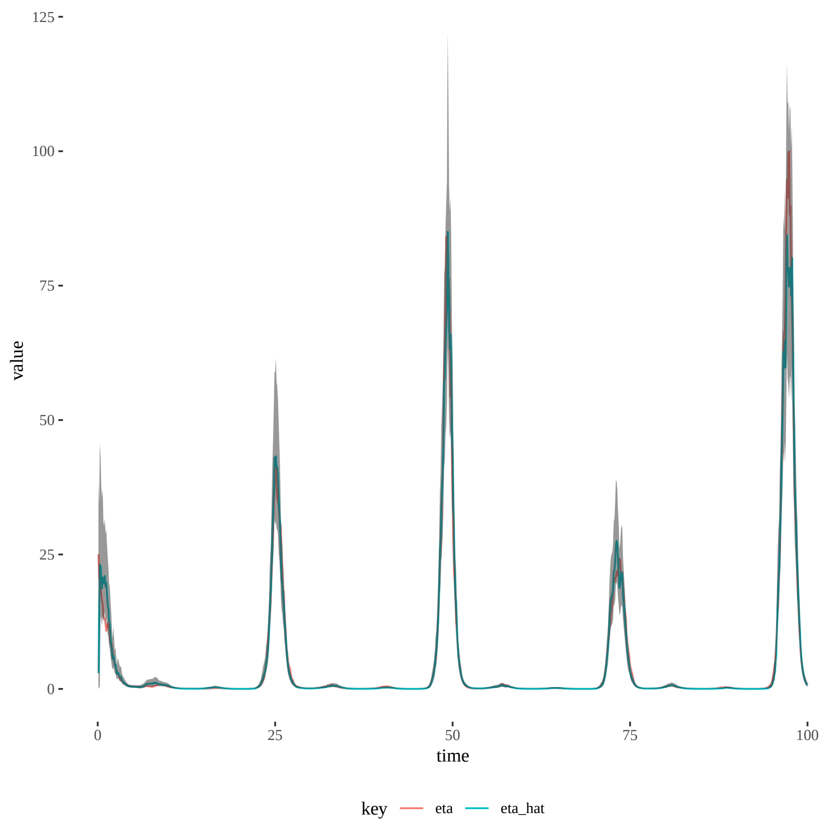


Figure 6.8: Simulated state and filtering distribution with 50% probability intervals for the simulated seasonal Negative Binomial POMP model. The actual parameter values used to simulate the data were used for the particle filter with 1,000 particles and systematic resampling

posed Negative Binomial model. The filtering distribution of the latent-state is determined using the particle filter.

Implementation (Fold and Scan)

The composable models and inference algorithms described in this chapter have been implemented using functional reactive programming in Scala. Observations arrive as a stream (introduced in Section 2.3). A stream is a pair, where the first element is a computed value and the second element is a thunk (an un-evaluated computation). In practice, when considering live streaming data, the function in the second element of the pair might be one which polls a web service for the next observation. The approach taken in this chapter is similar to Beckman's series of papers on Kalman folding, implementing the Kalman filter on streams in the Wolfram language [Beckman, 2016].

Since a stream is a `Foldable` collection, the previous functional implementations of the particle filter in section 2.1.14 and section 5.2.2 can be used when performing inference for the filtering state - however there are a few details omitted which are relevant when fitting composable POMP models. The POMP models feature irregularly observed data, hence the time difference between observations is required and the log-likelihood must be accumulated in order to perform inference for the static parameters.

In the application of the composable particle filter there is an internal state which propagates as the stream of data arrives. The internal state includes the particle cloud, the time of the most recent observation and the log-likelihood. Implementations of the particle filter have been considered in sections 2.1.15 and 5.2.2. However these implementations used discrete time state transition functions. The POMP models considered in this chapter have a continuous time latent-state and hence the transition function depends on the time difference between each pair of observations

```
def transition: (TimeIncrement) => (State) => State
```

Then, a single step of the particle filter which calculates the running pseudo-marginal log-likelihood and determines the filtering distribution using a particle approximation can be written using the transition function:

```
def filterStep(s: FilterState, y: Data) = {
  val dt = y.t - s.t
  val x1 = s.particles map (transition(dt))
  val logWeights = x1 map (dataLikelihood(y.observation))
  val ll = s.ll + log(mean(logWeights map exp))
  FilterState(y.t, resample(s.particles, logWeights), ll)
}
```

`FilterState` and `Data` are `case class`es defined as

```
case class FilterState(
  t: Time,
  particles: Vector[State],
  ll: LogLikelihood)
case class Data(
  t: Time,
  observation: Observation)
```

Firstly the time difference between subsequent observations `val dt = y.t - s.t` is calculated. The time difference is used in the `transition` function to advance the state. The new weights are calculated, given a new observation, using the `dataLikelihood` function. The states are resampled to get an approximate unweighted random sample from the filtering distribution $p(\theta(t_i)|y(t_{1:i}))$. As mentioned in section 5.2.1, in practice the log-likelihood is computed to avoid arithmetic underflow from multiplying many small values. The value of the log-likelihood, $\log \hat{p}_\psi(y(t_{1:M}))$, is updated by adding the log of the mean of the unnormalised-weights to the previous value. Note that the log-sum-exp trick [Murphy, 2012] should be used when updating the value of the log likelihood in line 4

```scala
val max = logWeights.max
val w = logWeights map { a => exp(a - max) }
val ll = s.ll + max + log(mean(w))
```

This rescaling of the weights ensures that large numbers aren't exponentiated, hence preventing overflow. It also ensures that not all weights will underflow. The largest weight is rescaled to $\exp(0) = 1$, but since

$$\log \sum_{i=1}^{N} \exp(w_i) = a + \log \sum_{i=1}^{N} \exp(w_i - a),$$

the value of the log-likelhood should remain the same.

In order to calculate the pseudo-marginal log-likelihood of a composable POMP model `mod` of a path, given observations, `data`, the function `foldLeft` is used:

```scala
val initState = FilterState(
  mod.sde.initialState.sample(1000), 0.0, 0.0)
foldLeft(data, initState)(filterStep).ll
```

Listing 6.1: Scala code to calculate the log-likelihood of a set of observations

The value `initState` is implemented by sampling 1,000 times (equivalent to 1,000 particles) from the initial state distribution of the model. The initial time `t0` is 0.0 and the initial value of the log-likelihood is set to zero. The log-likelihood is extracted by appending `.ll` on the call to `foldLeft`. To retrieve the approximate filtering distribution represented by the unweighted particles at each time step, then `scanLeft` can be used to apply `filterStep` and retain all intermediate values in the binary reduction as in section 2.1.14.

Filtering for the Log-Gaussian Cox-Process

When considering observations from the LGCP, the filtering needs to be performed slightly differently. The likelihood for the LGCP is given by

$$\pi(t_n|\lambda(t_n), \Lambda(t_n)) = \lambda(t_n)\exp(-\Lambda(t_n)),$$

which depends on the instantaneous hazard, $\lambda(t_n)$ and the cumulative hazard, $\Lambda(t_n) = \int_{t_0}^{t_n} \lambda(s)ds$. The log-likelihood is calculated to avoid arithmetic underflow and to simplify calculations, the log-likelihood is given by; $\ell = \log(\lambda(t)) - \Lambda(t)$. The state must be augmented to include the cumulative hazard, $\Lambda(t)$ in addition to $\lambda(t)$. In practice the value of $\Lambda(t)$ is calculated approximately, using numerical integration.

6.3.2 Parameter Estimation in POMP models

In order to perform on-line filtering to learn the posterior distribution of the dynamic latent-state of the composable models, the static parameters of the model must be determined. PMMH provides an offline method of parameter inference in situations where an unbiased estimate of the likelihood can be determined using a particle filter. Online parameter inference can be performed for the composable models using some of the algorithms introduced in section 4.6.

The particle marginal Metropolis-Hastings algorithm (PMMH) [Andrieu et al., 2010] is an offline Markov chain Monte Carlo (MCMC) algorithm which targets the full joint posterior $p(\theta(t_{0:M}), \psi | y(t_{1:M}))$ of a POMP model. The PMMH algorithm was introduced in section 5.3. The parameters of a POMP model as in Equation 6.1 include the measurement noise in the observation distribution, the parameters of the Markov transition kernel for the system state and the parameters of the initial state distribution.

Implementation (MCMC as a stream)

The particle marginal Metropolis-Hastings algorithm must be applied to a batch of data. Window functions, such as `grouped`, can be applied to a stream of data to aggregate observations into a batch. `grouped` accepts an integer, n, and groups each observation into another (finite) stream of size n.

The PMMH algorithm can then be applied to the aggregated group using `map`. Iterations from the PMMH algorithm are naturally implemented as a stream. In the Scala standard library there is a method for producing infinite streams from an initial seed:

```scala
def iterate[A](start: A)(f: A => A): Stream[A]
```

`iterate`, applies the function `f` to the starting value, then passes on the result to the next evaluation of `f`. The generates a Markov chain, where each iteration is dependent on the previous simulated value. `iterate` is an anamorphism, it is the categorical dual to the catamorphism which are generalisations of `folds` [Meijer et al., 1991].

Iterations of an MCMC algorithm can be generated using `iterate`, by starting with an initial value of the state (usually the likelihood evaluated at the last accepted value of the parameters (initialised at a large negative number) and the initial value of parameters) and applying the Metropolis-Hastings update at each iteration. Inside of each application of `f`, a new value of the parameters is proposed, the marginal likelihood is calculated using the proposed parameters (using the bootstrap particle filter) and the Metropolis-Hastings update is applied.

An example of a single step in the PMMH algorithm can be seen in Listing 6.2. The functions important

```
val prior: Parameters => LogLikelihood
val proposal: Parameters => Rand[Parameters]
val logLikelihood: Parameters => LogLikelihood
```

The function `logLikelihood` is a particle filter, with the observed data and number of particles fixed, which outputs an estimate of the log-likelihood for a given value of the parameters. `prior` represents the prior distribution over the static parameters and initial distribution of the latent state. This implementation is similar to the implementation in listing 3.3 in section 3.3, but in this case, `MetropState` stores the previous value of the log-likelihood. This is especially important when implementing pseudo-marginal MCMC methods in order to avoid running the particle filter twice using the same set of parameters to get an estimate of the log-likelihood.

```
val stepMetrop: MetropState => Rand[MetropState] = s => {
  for {
    propParams <- proposal(s.params)
    propll = logLikelihood(propParams)
    a = propll + prior(propParams) - s.ll - prior(s.params)
    u <- Uniform(0, 1)
    next = if (log(u) < a)
      MetropState(propll, propParams)
    else s
  } yield next
}
```

Listing 6.2: Metropolis-Hastings implementation without re-evaluating the pseudo-marginal log-likelihood for the previously accepted parameter value in the Markov chain

The Breeze library provides a class called, `MarkovChain` which can be used to generate a sequence dependent draws from a Markov chain given a transition kernel of type `A => Rand[A]`:

```
val initState = MetropState(-1e99, initParams)
val iters = MarkovChain(initState)(stepMetrop)
```

`initParams` are drawn from the prior distribution and the initial value of the log-likelihood is chosen to be very small so the first iteration of the PMMH is accepted.

Keeping the state of the PMMH algorithm inside of the `Rand` monad until the program is run and results are required, preserves referential transparency until the main method of the program, which is expected to have side effects, such as printing to the screen, writing to a file or database or generating pseudo-random numbers. The particle filter can also

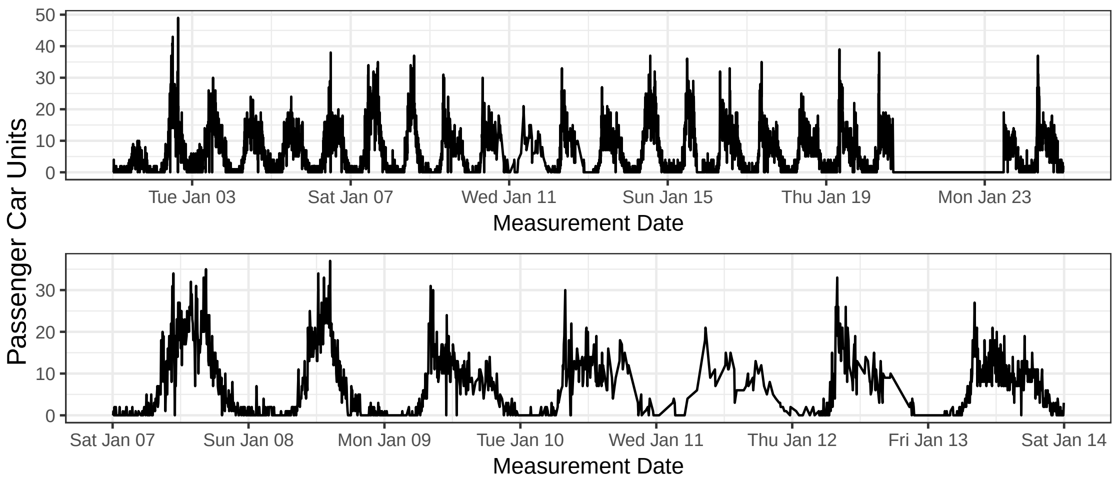


Figure 6.9: (Top) Measurements of passenger car units from sensor N05171T for the first three weeks of January 2017, (Bottom) Passenger car units for the second week of January 2017.

be written using monadic style to preserve referential transparency and is considered in section 2.1.14.

Built in stream operations can be used to discard burn-in iterations and thin the iterations to reduce auto-correlation between samples. The stream can be written to a file or database at each iteration, so the PMMH algorithm implemented as a stream uses constant memory as the chain size increases.

6.4 Example: Traffic Monitoring

This example is concerned with the traffic data introduced in section 5.9. Figure 6.9 (top) shows readings from the first three weeks of January. There are two strong trends appearing, a daily trend where most traffic is observed to be during the day and a weekly trend where most traffic is on the weekend with an exception for the bank holiday on Monday 2nd January.

The time difference between observations is typically five minutes or multiples of five minutes, however on the 9th January there is a time gap of 356 seconds. This means all of the following observations are out of sync with observations made before the 9th January as shown in table 6.1. This is not problematic for the class of composable models, as these models have a continuous time latent state, but this could cause difficulties for a discrete time analysis.

In Figure 6.9 (bottom), starting Wed 11th January the measurements are received at sporadic intervals, again this poses no problems for the class of composable POMP models, and is typical of sensor data.

Timestamp	Value	Δt (secs)
2017-01-09 07:59:17	21	300
2017-01-09 08:04:17	21	300
2017-01-09 08:09:17	31	356
2017-01-09 08:15:13	10	300
2017-01-09 08:20:13	19	300

Table 6.1: Five readings from sensor N05171T, with varying time difference (Δt)

The timestamp is translated to an hourly scale, ie. five minutes is represented as $5/60 \approx 0.083$. A sensible choice of observation model is the Poisson distribution. However it was found that the Poisson model is not a good fit for this data as the data is overdispersed, meaning the sample variance of the count data is greater than the sample mean. The Negative Binomial distribution is commonly used for overdispersed count data, and is chosen as the observation model. The data shows a clear daily seasonal trend, and a slightly less pronounced weekly trend. A sensible model is therefore a composition of a Negative Binomial model and two seasonal models, one with daily seasonality, the other with weekly seasonality.

$$\begin{aligned}
Y(t_i)|\eta(t_i) &\sim \text{NegBin}(\eta(t_i), r), \\
\eta(t_i)|\theta(t_i) &= \exp\{F_{t_i}^T \theta(t_i)\}, \\
\theta(t)|(\theta(t_{i-1}) = \theta(t_{i-1})) &\sim p(\theta(t_i)|\theta(t_{i-1}), \psi), \\
\theta(t_0)|\psi &\sim \text{MVN}(\mathbf{m}_0|\mathbf{C}_0).
\end{aligned}$$

The Negative Binomial model is the leftmost model in the composition, $\mathcal{M}_1$ and has a scalar latent state which evolves according to Brownian motion. This will account for drift in the observed data not modelled by the seasonal components. $\eta(t)$ is the mean of the Negative Binomial which is positive, therefore the link function is the log-link. The parameter r is the size parameter which controls the overdispersion of the data. The probability mass function for the Negative Binomial distribution with mean $\eta(t_i)$ and size ϕ is:

$$\text{NegBin}(y(t_i)|\eta(t_i), r) \sim \\ \binom{y(t_i) + r - 1}{y(t_i)} \left(\frac{\eta(t_i)}{\eta(t_i) + r}\right)^{y(t_i)} \left(\frac{r}{\eta(t_i) + r}\right)^{r}.$$

The mean is $\eta(t_i)$ and the variance is $\eta(t_i) + \eta(t_i)^2/r$, so $\eta(t_i)^2/r$ represents the additional variance above that of the Poisson distribution.

The second model in the composition is a daily seasonal model, $\mathcal{M}_2$. The daily seasonal model is chosen to have four harmonics, and period, $P = 24$, which means the frequency is $\omega = \frac{2\pi}{24}$, the vector F_t is:

$$F_t = \begin{pmatrix} \sin(\omega t) \\ \cos(\omega t) \\ \vdots \\ \sin(4\omega t) \\ \cos(4\omega t) \end{pmatrix}.$$

The daily seasonal model is chosen to have an Ornstein-Uhlenbeck latent state with a non-zero mean, μ:

$$\mathrm{d}\theta_t = \phi \vec{1}_8(\mu - \theta_t)\mathrm{d}t + \sigma I_8, \mathrm{d}\mathbf{W}_t$$

The rate, ϕ, variance σ were chosen to be the same for each state component. $\vec{1}_n$ is a vector of ones of dimension n and I_n is the identity matrix of dimension n.

The weekly seasonal model, $\mathcal{M}_3$ is the same as the daily seasonal model, but with the period of the seasonality, $P = 24 \times 7$ and the number of harmonics, $h = 2$, hence the latent state is 4-dimensional. The fully composed model is given by $\mathcal{M} = \mathcal{M}_1 \star \mathcal{M}_2 \star \mathcal{M}_3$.

6.4.1 Tuning the PMMH

The PMMH (as described in Section 5.3) is used to determine the joint posterior distribution of the parameters and latent state, $p(x, \psi|y(t))$, given the training data. The likelihood in the particle Metropolis Hastings algorithm is estimated using the bootstrap particle filter. The accuracy of the estimated marginal likelihood can be improved by increasing the number of particles in the filter, however this results in more computational time required for each MCMC iteration.

The PMMH algorithm requires the particle filter to run once per iteration (re-using the likelihood estimate from the previous iteration) and hence deciding on the optimum number of particles is a non-trivial task. Several pilot runs with different numbers of particles are performed to ensure the variance of the log of the likelihood estimate is close to one [Doucet et al., 2015]. This is difficult to achieve as the variance of the estimate can vary at different parameter values, not only with the number of particles. From several pilot runs at different values of the parameters, it was decided that 500 particles provided the best compromise between accuracy and computational time.

6.4.2 Parameter Estimates

In order to estimate the value of the parameters, 500 particles were used in the particle filter estimating the value of the marginal likelihood $\hat{p}_\psi(y(t_{1:M}))$, 100,000 iterations of the PMMH algorithm were taken with 10,000 iterations discarded as burn-in. The prior distributions on the parameters were chosen to be weakly informative. The initial state of the local-level model follows a Gaussian distribution, the prior on the each seasonal model initial state is a Multivariate Normal distribution with diagonal covariance matrices, $C_0^{(i)} I_{2h_i}$ where i is the model number and h_i is the number of harmonics in model i. The prior distributions for the parameters of the initial state distributions were chosen to be the same for each model component: $m_0^{(i)}$, $C_0^{(i)}$, $i = 1, 2, 3$, where i represents the model number:

$$\begin{aligned} m_0^{(i)} &\sim \mathcal{N}(0.5, 3^2), \\ C_0^{(i)} &\sim \text{InverseGamma}(2, 0.4), \end{aligned}$$

the mean of the initial state was chosen to have a Gaussian prior, and the standard deviation a Gamma prior. The Gamma distribution is parameterised by shape, α and rate β so that the expectation is α/β and the variance is α/β^2. Next, the diffusion parameter of the Brownian motion controlling the evolution of the latent state in the Negative Binomial model component:

$$\sigma \sim \text{Inv-Gamma}(2, 7).$$

The scale parameter, r, of the Negative Binomial observation distribution which controls the overdispersion is chosen to have an Inverse Gamma prior:

$$r \sim \text{Inv-Gamma}(10, 2).$$

The Ornstein-Uhlenbeck process controls the evolution of the latent state of the daily and weekly model components, the parameters of the OU process in model components 2 and 3 were given the following prior distributions:

$$\begin{aligned}
\mu_j^{(i)} &\sim \mathcal{N}(0.5, 3^2), \\
\phi^{(i)} &\sim \text{Gamma}(0.1, 0.3), \\
\sigma^{(i)} &\sim \text{Inv-Gamma}(2, 0.4).
\end{aligned}$$

The parameters are indexed by their model number, i, for $i = \{2, 3\}$, additionally the mean $\mu_j^{(i)}$ is indexed by j as there is a different value for each harmonic in the model. The mean of the Ornstein-Uhlenbeck process, $\mu^{(i)}$ is assigned a Gaussian prior, $\phi^{(i)}$ is given a bounded Beta prior distribution and $\sigma^{(i)}$ is given an Inverse-Gamma prior.

The proposal distribution was chosen to be a random walk centered at the previous values of the parameters, the bounded parameters, such as the strictly positive variance parameters were proposed on the log-scale.

A sample from the parameter prior distribution is used to initialise the PMMH algorithm. The mean of the parameters and associated 95% posterior intervals are presented in table 6.2. The mean value of the size parameter, r in the Negative Binomial observation distribution was 3.23.

6.4.3 On-Line Filtering

The joint posterior of the state and parameters, $p(x, \psi|y)$ can be used to perform filtering and forecasting on-line. The data used for on-line filtering is the last week of January, these observations have not been considered when estimating the posterior distribution. In order to take into account uncertainty in the posterior distribution, a pair (x^*, ψ^*) is sampled from the posterior distribution and used to initialise 100 particle filters each with 1,000 particles. In order to get a precise representation of the filtering distribution, many more particles can be used than in the PMMH algorithm. Once an approximation to the filtering distribution is determined at time t, it is then advanced to the time of the next expected measurement (t + 5 minutes in this case) and the associated observation is simulated for each particle, giving a predictive distribution for the observations which can be summarised.

Figure 6.10 shows the actual measurements on 25th and 26th January and the associated one-step forecast mean with 90% prediction intervals. It is clear that the variance of the one-step-predictive distribution is larger when the expected value of passenger car units is larger.

Parameter	2.5%	mean	97.5%
r	1.50	3.23	4.74
$m_0^{(1)}$	-2.57	-0.73	1.07
$C_0^{(1)}$	0.06	0.35	1.07
$\sigma^{(1)}$	0.01	0.01	0.02
$\phi^{(2)}$	0.18	0.45	0.94
$\sigma^{(2)}$	0.51	0.84	1.41
$\mu_1^{(2)}$	-5.03	-3.92	-1.82
$\mu_2^{(2)}$	-0.19	-0.05	0.18
$\mu_3^{(2)}$	-0.19	-0.01	0.19
$\mu_4^{(2)}$	-0.17	0.02	0.19
$\mu_5^{(2)}$	-2.03	-1.42	-0.82
$\mu_6^{(2)}$	-2.5	-1.05	1.18
$\mu_7^{(2)}$	-0.19	-0.01	0.19
$\mu_8^{(2)}$	-0.40	0.02	0.33
$m_0^{(2)}$	-0.44	-0.01	0.40
$C_0^{(2)}$	0.83	1.02	1.21
$\phi^{(3)}$	0.19	1.60	11.54
$\sigma^{(3)}$	0.49	2.00	10.04
$\mu_1^{(3)}$	-4.71	-2.60	-0.75
$\mu_2^{(3)}$	-0.99	-0.09	0.59
$\mu_3^{(2)}$	-0.5	-0.10	0.3
$\mu_4^{(2)}$	-0.65	-0.20	0.26
$m_0^{(3)}$	-1.02	-0.16	1.11
$C_0^{(3)}$	0.11	0.93	3.40

Table 6.2: Posterior mean and 95% posterior intervals for the parameters in the Negative Binomial Traffic Model

6.5 Conclusion

Composable Markov process models are a flexible class of models which can be used to model a variety of real world data. The class of composable models together with the composition operator form a semigroup, making it easy to build complex models by combining simple models components. The use of the particle filter for simulation based inference allows for a flexible choice of observation model and SDE governing the latent state. Further, by using a continuous time Markov process to represent the latent state, the requirement to simulate the state on a regular grid is relaxed. If there is an exact solution to the Markov process, then the latent state can be simulated forward to the next

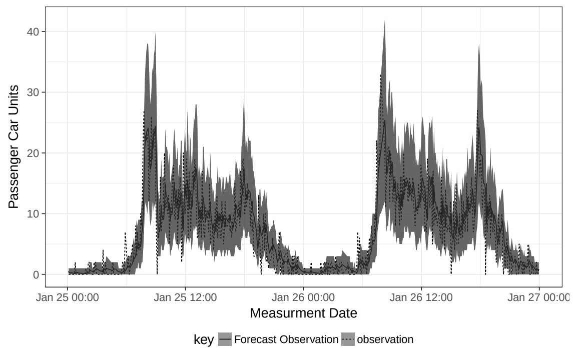


Figure 6.10: On-Line filtering with mean of the one-step forecast and 90% prediction intervals

time of interest using the time increment and the previous realisation of the state. This allows for faster computation when using real world data and is important since many sensors sample at an adaptive rate, or consumption must be slowed down due to limited computing resources for inference and forecasting.

Incoming observations are represented as *streams*, which can represent live observations from a webservice or other source. This allows for flexible and scalable analysis. The composable model framework can be easily tested on simulated data and the same code deployed for real world applications. The particle filter is implemented as a functional `fold`, a higher-order function allowing recursive data structures (such as the cons-list or cons-stream) to be reduced pairwise without side-effecting. The PMMH algorithm is expressed as a stream using, `MarkovChain`, defined in the Breeze numerical library for Scala. These functions are often called unfold operations and start with a seed value and first apply a function to the seed, the each subsequent value to create an infinite stream of values which can be manipulated using built-in stream processing functions. In the accompanying software, the streams are implemented using Akka Streams, this allows for easy implementation of parallel chains, online monitoring and constant-time memory usage, when used with asynchronous-IO.

The Scala implementation is typically faster than dynamic langauges such as R and Python, but slightly slower than C or C++. Algorithms such as the particle filter can be written in a functional way, using higher order functions (`map`, `fold` etc.) which allows one implementation of the algorithm to be parametric over different collection types. This simplifies development and testing, since the algorithm can be tested and debugged using sequential collections and deployed in production using parallel collections, or distributed collections, for example RDDs (Resilient Distributed Datasets) in Apache Spark [Zaharia et al., 2010].

Chapter 7

Heteroscedastic Time Series

Streaming data from large sensor fields arrives continuously and is often non-stationary. The time series models and offline inference methods considered so far have assumed the sensor data is stationary and the variance does not change with time. However, this is typically not a valid assumption and models for which both the variance and the mean evolves in time are considered in this chapter.

Similar to the specification of state-space models with a dynamic mean term, the log-variance is assumed to evolve in time according to a stochastic process. A naive construction of the model can lead to the size of the latent-state increasing according to the length of the time series under consideration and the size of the covariance matrix which is considered to be time inhomogeneous. This large latent-space is a challenge for inference algorithms. An approach to reduce the size of the latent-state using a factor structure for the covariance matrix is considered. Another approach commonly used to model large covariance matrices is using a covariance (or kernel) function; this is common in spatial modelling and is considered in chapter 8. In this chapter a factor structure is used to model multiple non-stationary, correlated time series.

7.1 Univariate Stochastic Volatility

In order to model a time series with non-constant variance, the variance can be considered a latent-state modelled using a stochastic process. Stochastic volatility (SV) models are a well studied class of models with non-constant variance. SV models are used extensively in finance applications, representing the uncertainty in the return on an investment or stock. High volatility refers to a stock that has high variance and is unpredictable.

The model in equation (7.1) is a discrete time stochastic volatility model with an AR(1) latent state.

$$\begin{aligned} Y_t &= \varepsilon_t \exp(\alpha_t/2), & \varepsilon_t &\sim \mathcal{N}(0,1), \\ \alpha_t &= \mu + \phi(\alpha_{t-1} - \mu) + \eta_t, & \eta_t &\sim \mathcal{N}(0, \sigma_\eta^2). \end{aligned} \tag{7.1}$$

Y_t, $t = 1, \ldots, N$ are discrete observations of a mean-zero time series process. The latent state represents the log-volatility and α_t follows an AR(1) process with mean μ.

Figure 7.1 shows a simulation from this model with fixed values of the parameters, $\phi = 0.8, \mu = 1, \sigma_\eta = 0.1$, the initial state of the log-volatility is the stationary solution of the AR(1) process: $\alpha_0 \sim \mathcal{N}\left(\mu, \frac{\sigma_\eta^2}{1-\phi^2}\right)$.

7.1.1 Parameter Inference

The posterior distributions of the parameters in the model, $\psi = \{\phi, \mu, \sigma_\eta\}$ and the latent state $\alpha_{0:N}$ can be learned by constructing a Markov Chain whose stationary distribution is equivalent to the joint posterior distribution $p(\psi, \alpha_{0:N}|y_{1:N})$ as in Shephard and Pitt [1997]. The steps are summarised below. First consider the log of the square of the observations to get the model:

$$\begin{aligned} \log(y_t^2) &= \alpha_t + \log(\varepsilon_t^2), & \varepsilon_t &\sim \mathcal{N}(0,1), \\ \alpha_t &= \mu + \phi(\alpha_{t-1} - \mu) + \eta_t, & \eta_t &\sim \mathcal{N}(0, \sigma_\eta^2). \end{aligned} \tag{7.2}$$

This is a linear system, but the quantity $\log(\varepsilon_t^2)$ is not Gaussian. Now calculate the mean and variance of this quantity and use a Gaussian approximation to sample the state using FFBS (see section 4.4). To calculate the mean and variance, note that the mean and variance of ε is 0 and 1 respectively then transform the probability density of ε:

$$\begin{aligned} p(\log(\varepsilon^2)) &= f(g^{-1}(\varepsilon)) \left| \frac{d}{d\varepsilon} g^{-1}(\varepsilon) \right| \\ &= f\left(\exp\left\{\frac{\varepsilon}{2}\right\}\right) \frac{1}{2} \exp\left\{\frac{\varepsilon}{2}\right\} \\ &= (2\pi)^{-1/2} \exp\left\{-\frac{1}{2} \exp\left\{\frac{\varepsilon}{2}\right\}^2\right\} \frac{1}{2} \exp\left\{\frac{\varepsilon}{2}\right\} \\ &= \frac{1}{2\sqrt{2\pi}} \exp\left\{\frac{\varepsilon}{2} - \frac{1}{2}\exp\{\varepsilon\}\right\}. \end{aligned}$$

$g(\varepsilon) = \log(\varepsilon^2)$, then $g^{-1}(\varepsilon) = \exp(\varepsilon/2)$. Since the function g is not monotonic and has two solutions in x for $g(x) = y$ this needs to be doubled. This is the log-chi-squared distribution with $\nu = 1$, the probability density function of the log-chi-squared distribution is

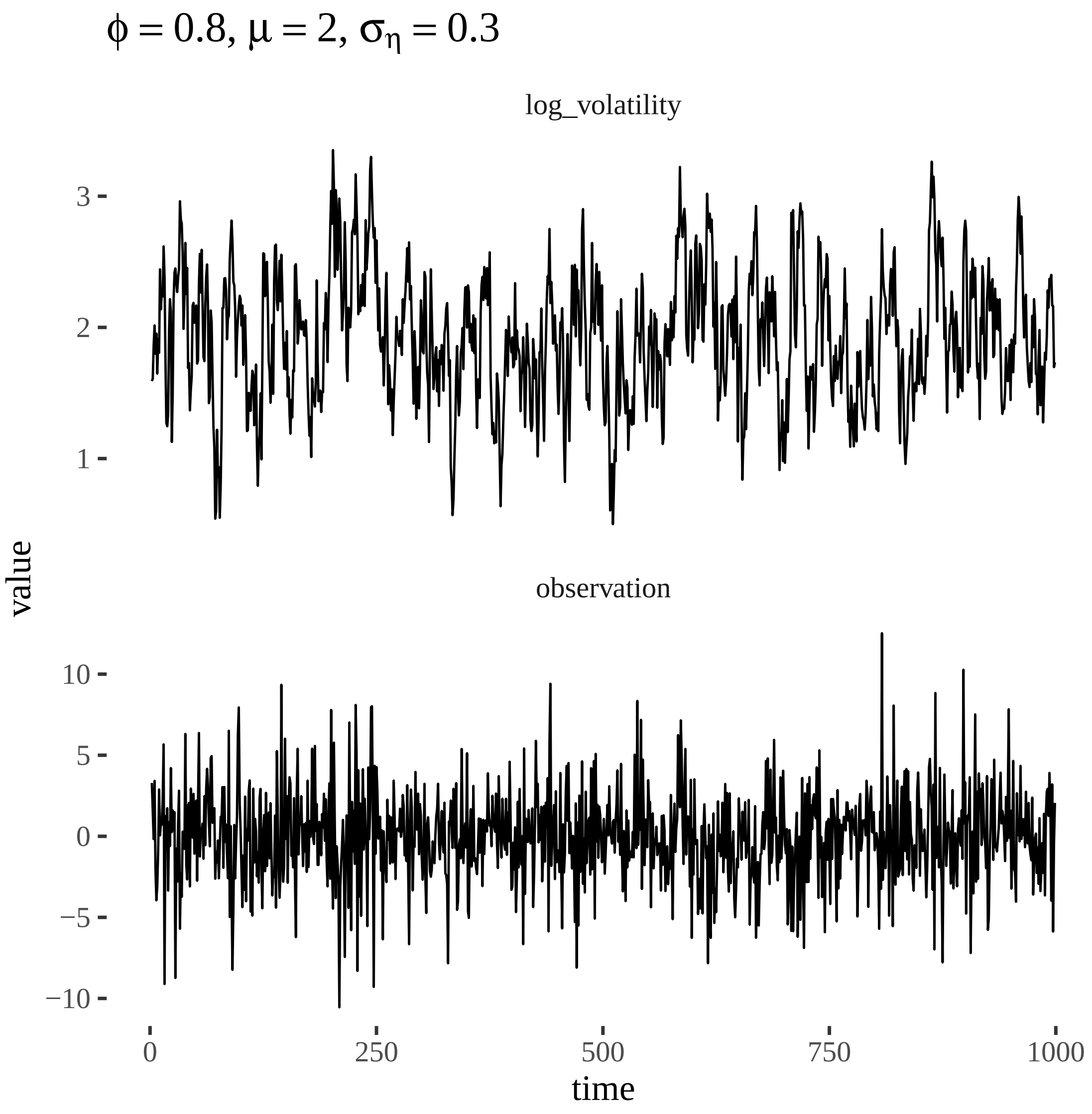


Figure 7.1: Simulated latent state representing the log-volatility of the series, (top). Simulated observations from the univariate stochastic volatility model, (bottom).

$$p(X) = \frac{1}{2^{\nu/2}\Gamma\left(\frac{\nu}{2}\right)} \exp\left\{\frac{\nu X}{2} - \frac{1}{2}\exp(X)\right\}, \tag{7.3}$$

since $2^{1/2}\Gamma\left(\frac{1}{2}\right) = \sqrt{2\pi}$. Now calculate the moment generating function for this distribution:

$$\begin{aligned}\mathbb{E}(e^{tX}) &= \int_{-\infty}^{\infty} (2\pi)^{-1/2} \exp\left\{\frac{x}{2} - \frac{1}{2}\exp(x) + tx\right\} \mathrm{d}x \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \exp\left\{\frac{x(1+2t)}{2} - \frac{1}{2}\exp(x)\right\} \mathrm{d}x \\ M(t) &= \frac{2^{1/2+t}\Gamma\left(\frac{1}{2}+t\right)}{\sqrt{2\pi}}\end{aligned}$$

The n^{th} derivative of the moment generating function with respect to t evaluated at $t = 0$ corresponds to the n^{th} moment. For the first moment (the mean) calculate the first derivative:

$$M'(t) = \psi(t + 1/2) + \log(2),$$

where ψ is the digamma function. The mean of the log-Chi-Squared distribution with $\nu = 1$ is found by setting $t = 0$ in the moment generating function: $\psi(1/2) + \log(2) = -1.27$. The second moment, the variance, is $M''(t = 0) = \psi'(1/2) = \pi^2/2$ where ψ' is the trigamma function. FFBS is used to sample the latent log-volatility with the transformed observations, $\tilde{y}_t = \log(y_t^2)$, the system evolution density is $p(\alpha_t|\alpha_{t-1}, \mu, \phi, \sigma_\eta) = \mathcal{N}(\alpha_t|\mu + \phi(\alpha_{t-1} - \mu), \sigma_\eta^2)$. Suppose the mean and variance of the posterior of the filtering distribution at time $t - 1$, are m_{t-1}, C_{t-1}, then the update to the posterior distribution at time t are given by the Kalman filtering equations for the AR(1) process:

$$\begin{aligned}&\text{Prior:} && a_t = \mu + \phi(m_{t-1} - \mu) \\ & && R_t = \phi^2 C_{t-1} + \sigma_\eta^2 \\ &\text{Update:} && m_t = a_t + A_t e_t \\ & && C_t = A_t V, \\ &\text{where} && A_t = R_t/(R_t + V), \quad e_t = \tilde{y}_t - a_t,\end{aligned}$$

and the observation variance is, $V = \frac{\pi^2}{2}$.

To perform backward sampling start by sampling from the posterior of the filtering distribution at time $t = N$, $\alpha_N \sim \mathcal{N}(m_N, C_N)$ then proceed backwards through the filtering distribution:

$$\begin{aligned}\gamma_t &= m_t + \frac{C_t \phi}{R_{t+1}}(\alpha_{t+1} - a_{t+1}),\\ \mathit{\Gamma}_t &= C_t - \frac{C_t^2 \phi^2}{R_{t+1}},\end{aligned}$$

then sample α_t from the distribution $\mathcal{N}(\gamma_t, \mathit{\Gamma}_t)$. This is sufficient to sample the approximate latent-state where the innovations are considered approximately Gaussian. In order to correct the approximate samples to target the correct posterior a Metropolis correction must be applied to the samples of the state. The state is divided into blocks, each block starts and ends with a knot. Each block is accepted according to a Metropolis-step with $A = \ell(\alpha_i^\star)\ell_G(\alpha_i)/\ell_G(\alpha_i^\star)(\ell(\alpha_i)$ where ℓ_G is the likelihood of the approximate Gaussian latent-state and ℓ is the likelihood of the exact latent-state. The size of the blocks to sample is a tuning parameter for the MCMC. Shephard and Pitt suggests sampling the size of the blocks from a discrete uniform distribution with upper and lower bounds, $k \sim \mathrm{U}(a, b)$, note that the final block is allowed to be smaller than a. If the block size is chosen to be small, then it is more likely to be accepted, however mixing will be slow. Additionally, the positions of the knots which are conditioned upon change with each iteration of the MCMC, hence sampling can be performed in parallel as each block is conditionally independent from the others given the knots.

The log-likelihood for a block of size n starting at $t = k$ for the Gaussian approximation is written as

$$\begin{aligned}\ell_G(\alpha_i) &= \log\left(\prod_{t=k}^{k+n}(2\pi^3/2)^{-\frac{1}{2}}\exp\left\{-\frac{2}{2\pi^2}(\log(y_t^2) + 1.27 - \alpha_t)^2\right\}\right),\\ &= -\frac{3n}{2}\log(\pi) - \frac{1}{\pi^2}\sum_{t=k}^{k+n}(\log(y_t^2) + 1.27 - \alpha_t)^2.\end{aligned}$$

The conditional likelihood (given a value of the volatility α_t) of an observation under the true model is

$$\begin{aligned}p(y_t|\alpha_t) &= \mathcal{N}(0, e^{\alpha_t})\\ &= (2\pi e^{\alpha_t})^{-1/2}\exp\left\{-\frac{y_t^2}{2e^{\alpha_t}}\right\}\\ &= (2\pi)^{-1/2}\exp\left\{-\frac{y_t^2}{2e^{\alpha_t}} - \frac{\alpha_t}{2}\right\}.\end{aligned}$$

The log-likelihood for a block of size n starting at $t = k$ can be written as:

$$\ell(\alpha_i) = -\frac{n}{2}\log(2\pi) - \frac{1}{2}\sum_{t=k}^{k+n}(\alpha_t + y_t^2 \exp(-\alpha_t)) \tag{7.4}$$

Then a Metropolis correction can be performed in order to sample the latent log-volatility from the correct posterior distribution. Each proposed block $\alpha_i^\star$ is accepted with probability $\min(1, A)$ where $\log(A) = (\ell(\alpha_i^\star) - \ell_G(\alpha_i^\star)) - (\ell(\alpha_i) - \ell_G(\alpha_i))$.

7.1.2 Mixture Model Approximation

An alternative inference scheme for the stochastic volatility model has been proposed where the the quantity $\varepsilon_t \sim \log\chi_1^2$ is approximated using a mixture model [Kim et al., 1998]. The distribution of ε_t can be approximated by

$$p(\varepsilon_t) \approx \sum_{i=1}^{7} \pi_i \mathcal{N}(\mu_i, v_i^2),$$

where

$$\begin{aligned}
\pi &= \begin{pmatrix} 0.0073 & 0.1056 & 0.00002 & 0.044 & 0.34 & 0.2457 & 0.2575 \end{pmatrix}, \\
\mu &= \begin{pmatrix} -11.4 & -5.24 & -9.84 & 1.51 & -0.65 & 0.53 & -2.36 \end{pmatrix}, \\
v^2 &= \begin{pmatrix} 5.8 & 2.61 & 5.18 & 0.17 & 0.64 & 0.34 & 1.26 \end{pmatrix}.
\end{aligned}$$

These parameters and the number of components ($k = 7$) were determined in Kim et al. [1998] by matching the first four moments of the approximating distribution to the actual error distribution using non-linear least squares.

Then, writing $z_t = \log(y_t^2) - \mu_{s_t}$ the state space model is conditionally Gaussian:

$$\begin{aligned}
z_t|\alpha_t, s_t &= \alpha_t + v_t, & v_t &\sim \mathcal{N}(0, v_{s_t}^2), \\
\alpha_t &= \mu + \phi(\alpha_{t-1} - \mu) + \eta_t, & \eta_t &\sim \mathcal{N}(0, \sigma_\eta^2).
\end{aligned}$$

Here, s_t is an indicator variable of length N sampled with probability $P(s_t = i) \propto \pi_i \mathcal{N}(z_t|\alpha_t, v_i^2)$. A Gibbs sampling scheme to sample from the posterior distribution $p(\psi, \alpha_{0:T}, s_t|z_t)$ can then be constructed as described in algorithm 20. The auxiliary variables, s_t can be marginalised out leaving the correct joint-posterior distribution.

Algorithm 20: Gibbs Sampler for the mixture model approximation of the stochastic volatility model

```
   Result: Return $\psi_i, \alpha_{0:N}^{(i)}$ for $i = 1, \ldots, M$
 1 Given observation $Y_t, t = 1, \ldots, N$;
 2 Sample an initial trajectory of the state using the Gaussian approximation in
     equation (7.2);
 3 Initialise the indicator variables $s_t, t = 1, \ldots, N$ conditional on the approximate
     state;
 4 Sample initial values of the parameters from the prior, $\psi_0 \sim p(\psi)$;
 5 for $i$ in $1 : M$ do
 6     Sample from the full conditional distribution of the latent-state using FFBS;
 7     Sample each indicator $s_i$, $i = 1, \ldots, t$ from a multinomial distribution with
         probabilities proportional to $\pi_j \mathcal{N}(z_t|\alpha_t, v_j^2)$, $j = 1, \ldots, 7$;
 8     Sample the parameters from the full conditional distributions according to
         eqs. (1.11) and (1.13);
 9 end
```

Sampling the Autoregression Parameter

An AR(1) process is stationary if and only if $|\phi| < 1$. in order to sample ϕ values which admit a stationary AR(1) process a Beta prior distribution, $p(\phi) \sim \mathcal{B}(\alpha_\phi, \beta_\phi)$ is chosen. The support of the Beta distribution is $(0, 1)$, any of these values are suitable for the value of the autoregressive parameter ϕ and all admit a stationary AR(1) process. A Metropolis-Hastings step is used to update the value of ϕ using a beta proposal distribution centred at the previous value of ϕ. The proposal distribution is:

$$q(\phi^\star|\phi) = \mathcal{B}(\lambda\phi + \tau, \lambda(1 - \phi) + \tau), \tag{7.5}$$

where λ and τ are tuning parameters, τ is very small and prevents the sampler becoming stuck near zero. The mean is $\mathbb{E}(\phi^\star) = \frac{\alpha}{\alpha+\beta} = \frac{\lambda\phi+\tau}{\lambda+2\tau} \approx \phi$. The likelihood of the AR(1) process can be written as:

$$\begin{aligned} p_\psi(\alpha_{0:N}) &= p(\alpha_0) \prod_{t=1}^{N} p(\alpha_t|\mu, \sigma_\eta, \phi, \alpha_{t-1}), \\ &= \mathcal{N}\left(\mu, \frac{\sigma_\eta^2}{1 - \phi^2}\right) \prod_{t=1}^{N} \mathcal{N}(a_t; \mu + \phi(\alpha_{t-1} - \mu), \sigma_\eta^2), \\ &\approx (2\pi\sigma_\eta^2)^{-\frac{N}{2}} \exp\left\{-\frac{1}{2\sigma_\eta^2} \sum_{t=1}^{N} (\alpha_t - \mu - \phi(\alpha_{t-1} - \mu))^2\right\}. \end{aligned}$$

Ignoring the small contribution from α_0. The Metropolis-Hastings acceptance ratio is then:

$$A = \frac{p_{\psi^\star}(\alpha_{1:N})p(\phi^\star)q(\phi|\phi^\star)}{p_{\psi}(\alpha_{1:N})p(\phi)q(\phi^\star|\phi)} \tag{7.6}$$

Where $\psi^\star$ represents the parameters of the AR(1) process with a newly proposed value of ϕ.

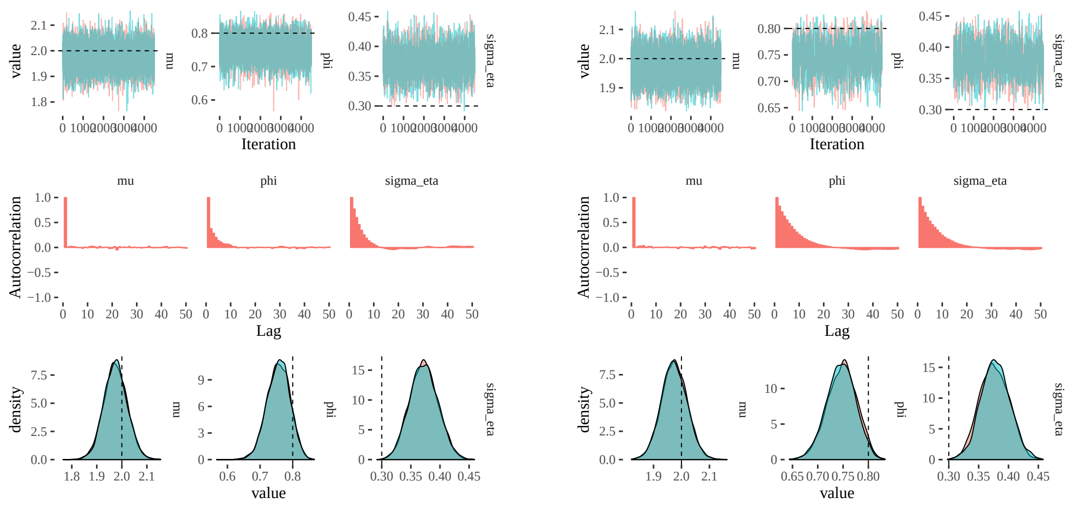
Example: Stochastic Volatility with AR(1) latent state

This section compares the two inference algorithms for the latent-state and static parameters of the stochastic volatility model using the simulated data in Figure 7.1. Firstly, the mixture model approximation is used with the semi-conjugate prior distributions

$$\begin{aligned}
p(\phi) &= \mathcal{N}(0.8, 0.1), \\
p(\mu) &= \mathcal{N}(2.0, 1.0), \\
p(\sigma_\eta) &= \text{Inv-Gamma}(2, 2).
\end{aligned}$$

The static parameters are initialised by drawing from the independent prior distributions, the state is initialised using the FFBS algorithm on the Gaussian approximation without a correction. The indicators s_t can then be initialised conditional on the initial approximate state. In the second inference algorithm the Beta prior for the autoregressive parameter ϕ is used. The tuning parameters of the proposal distribution for ϕ are selected to be $\lambda = 100$ and $\tau = 0.05$. The values of the tuning parameters were determined by monitoring runs of the MCMC and aiming for one third of the proposals for the Metropolis-Hastings step for ϕ to be accepted. The prior distribution for ϕ is $\mathcal{B}(5, 2)$. 100,000 iterations from each MCMC algorithm is used, the first 10,000 iterations are discarded as burn-in and every 20th iteration is kept, reducing the autocorrelation of the Markov chains. Figure 7.2 shows the diagnostic plots for the mixture approximation using the Normal prior (a) and Beta prior (b). In addition, Figure 7.3 shows the prior and posterior distributions for the mixture model approximation using the Normal prior distribution for the autoregressive parameter.

Figure 7.4 shows the mean and 99% credible intervals of the log-volatility from 1,000 MCMC samples using the mixture model approximation described in this section and the block updating described in section 7.1.1. The mean latent-state for each algorithm is very similar.



(a) (b)

Figure 7.2: (a) Normal prior for ϕ (b) Beta prior for ϕ (a and b) Diagnostic plots for parameters of the SV model using the mixture model approximation (Top) Traceplot for latent-state parameters (Middle) Autocorrelation plots of the MCMC draws (Bottom) Empirical marginal posterior density for latent-state parameters

7.1.3 Continuous Time Stochastic Volatility Model

In order to model irregularly observed time series, $Y(t_i)$, $i = 1, \ldots, N$ with a time varying variance, a stochastic volatility model with continuous time latent log-volatility can be used. The Ornstein-Uhlenbeck process (see section 4.7.1) is a continuous time analogue to the autoregressive process:

$$
\begin{aligned}
Y(t_i) &= \varepsilon(t_i)\exp(\alpha(t_i)/2), & \varepsilon(t_i) &\sim \mathcal{N}(0,1), \\
\mathrm{d}\alpha_t &= \phi(\alpha_t - \mu)\mathrm{d}t + \sigma \mathrm{d}W_t. & & \qquad (7.7)
\end{aligned}
$$

Using the analytical solution [Doob, 1942], the state space model can be written as:

$$
\begin{aligned}
Y(t_i) &= \varepsilon(t_i)\exp(\alpha(t_i)/2), & \varepsilon(t_i) &\sim \mathcal{N}(0,1), \\
\alpha(t_i) &= \mu + e^{-\phi(t_i - t_{i-1})}(\alpha(t_{i-1}) - \mu) + \sigma(t_i), & \sigma(t_i) &\sim \mathcal{N}\left(0, \frac{\sigma^2}{2\phi}(1 - e^{-2\phi(t_i - t_{i-1})})\right)
\end{aligned}
$$

A simulation from the process is presented in Figure 7.5.

Inference can be performed using either the mixture model approximation or the block update method. The posterior distributions for the static parameters, ϕ, μ, σ can be de-

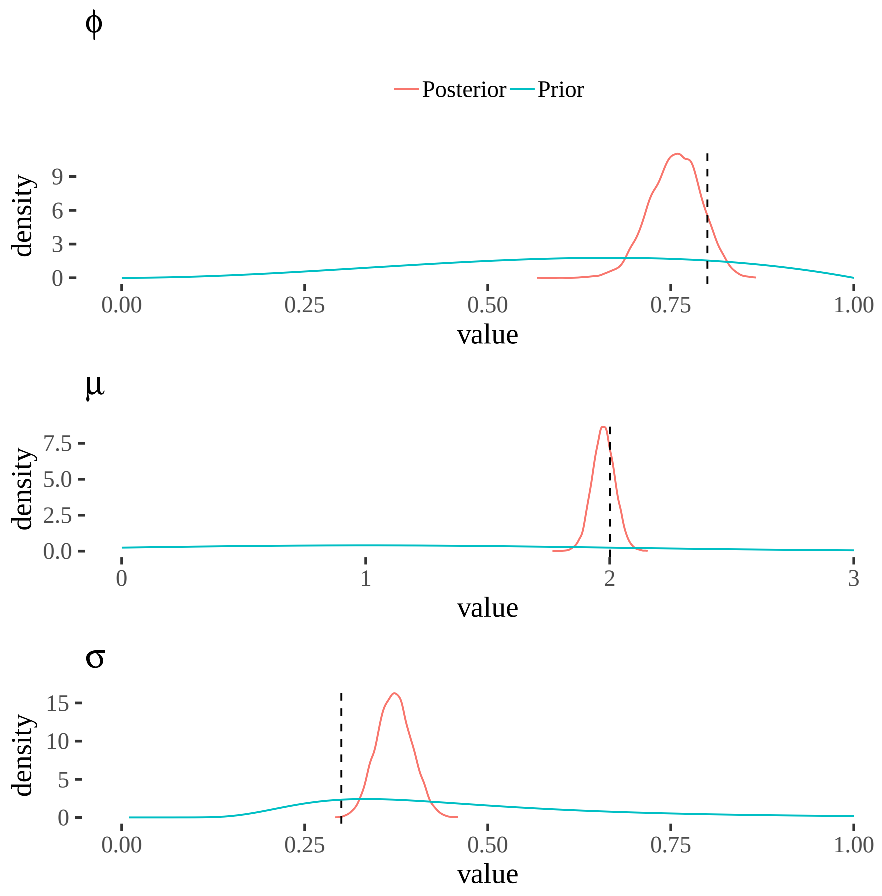


Figure 7.3: Prior and posterior distributions for the parameters of the stochastic volatility model using the mixture model approximation with Normal prior for the autoregressive parameter ϕ.

termined using Metropolis-Hastings as detailed in section 4.7.1 conditional on the sampled state, $\alpha(t_{0:N})$.

7.2 Factor Stochastic Volatility

Factor analysis is used to express dependence among random variables

$$Y - \mu = \beta f + v. \tag{7.8}$$

The observed values form a matrix Y with dimension $p \times n$, each column is a single observation and μ is a $p \times n$ matrix of mean values. β is the $p \times k$ factor loading matrix which is shared among all the observations, f is a $k \times n$ matrix of common factors. v is a $p \times n$ matrix with each column a p-dimensional random vector distributed according to $\text{MVN}(0, \Sigma_y)$ where $\Sigma_y = \text{diag}(\sigma_1, \dots, \sigma_p)$ is a diagonal matrix representing the variance

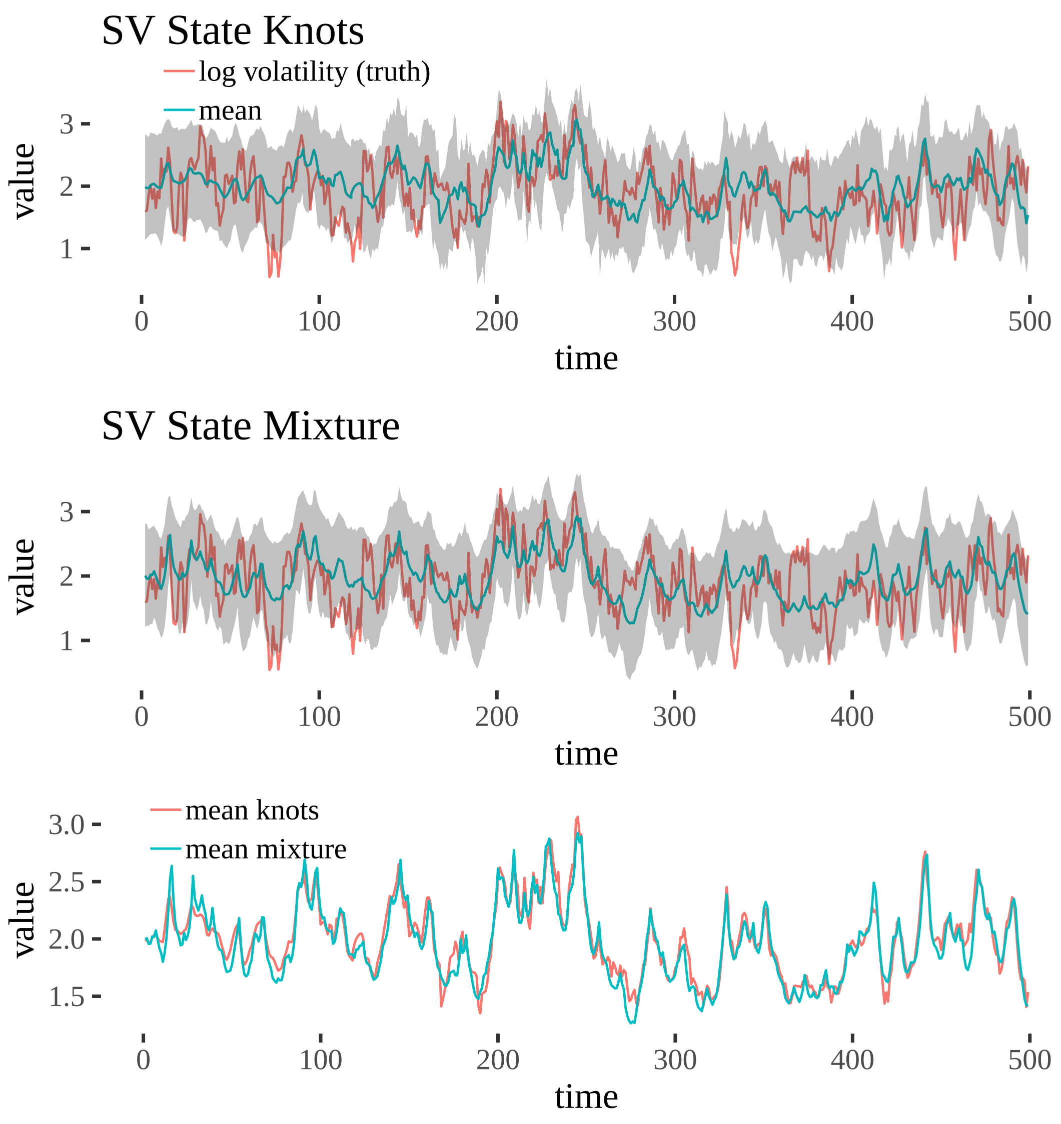


Figure 7.4: Sampled state of the stochastic volatility model using the mixture model approximation and the Gaussian approximation with the simulated state and 99% intervals. Gaussian approximation, (top). Mixture model approximation, (middle). The posterior mean of the latent state using both the mixture model and Gaussian approximation, (bottom).

not explained by the common factors and factor loading matrix. Typically $k < p$ which means the model has fewer free parameters than a model using a full covariance matrix. Factor analysis has been extensively studied, see for example Kaufman and Press [1973] for a Bayesian analysis of factor models.

Factor analysis can be extended to modelling time series with time dependent variance [Pitt and Shephard, 1999a]. The $p \times k$ factor loading matrix β provides a relationship

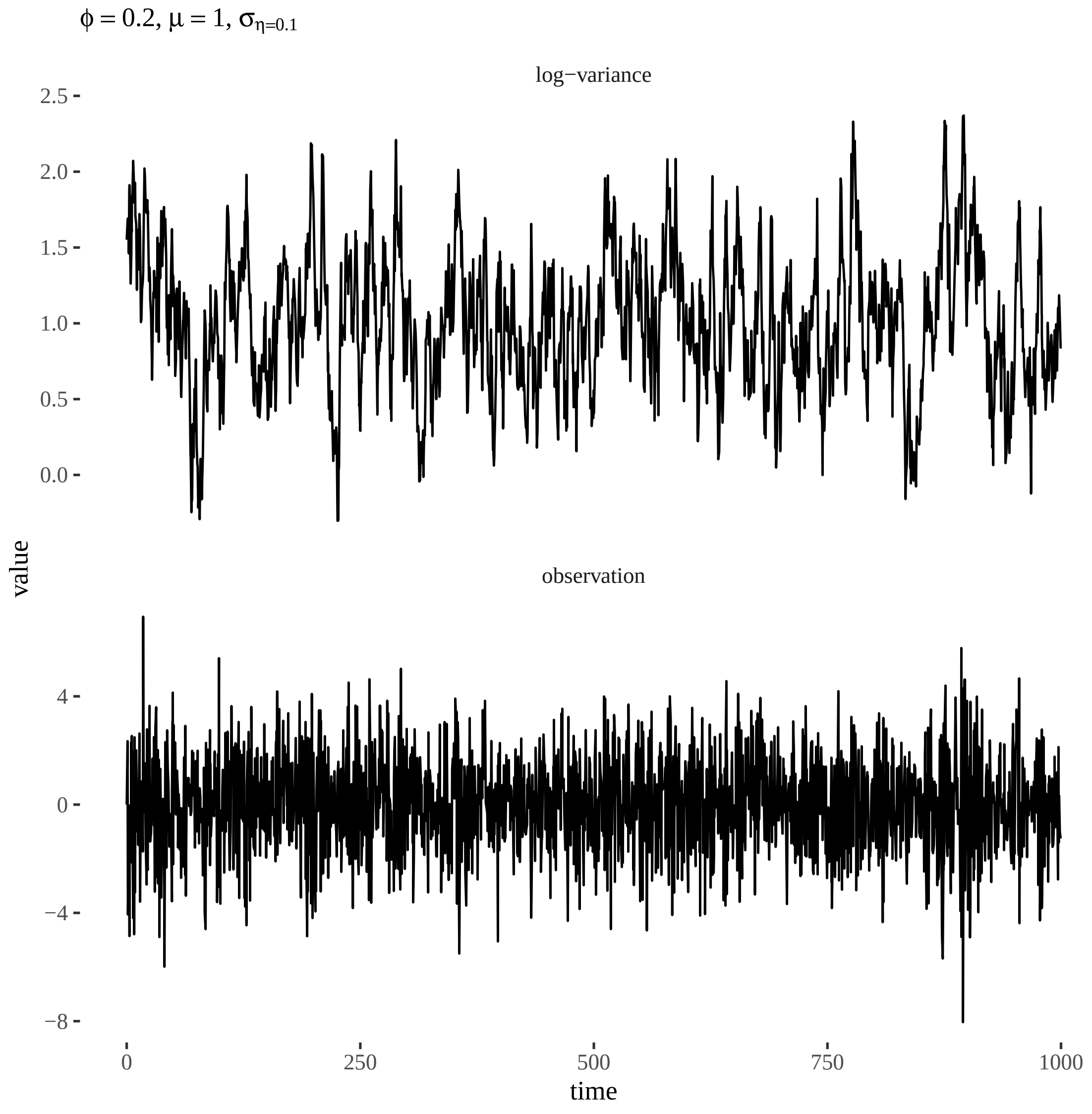


Figure 7.5: Simulated Observations from the univariate stochastic volatility model with Ornstein-Uhlenbeck latent state, (top). Simulated Latent State, representing the log-volatility of the series, (bottom),

between k dimensional common factors, f_t and the p-dimensional observation, Y_t. The factors, f_t evolve in time according to a stochastic volatility model where the log-volatility is modelled using a stochastic process. The measurement error v_t is Multivariate Normal with mean zero and diagonal variance $\Sigma_y = \text{diag}(\sigma_y^{2(1)}, \dots, \sigma_y^{2(p)})$. The model is completely specified in equation (7.9) with an AR(1) model representing the log-volatility of the latent common factors

$$\begin{aligned}
Y_t &= \beta f_t + v_t, & v_t &\sim \mathrm{MVN}_p(0, \Sigma_y) \\
f_t^{(j)} &= \varepsilon_t^{(j)} \sigma \exp(\alpha_t^{(j)}/2), & \varepsilon_t^{(j)} &\sim \mathcal{N}(0,1) \\
\alpha_t^{(j)} &= \mu^{(j)} + \phi^{(j)}(\alpha_{t-1}^{(j)} - \mu^{(j)}) + \eta_t^{(j)}, & \eta_t^{(j)} &\sim \mathcal{N}(0, \sigma_\eta^{(j)}), \\
\alpha_0^{(j)} &\sim \mathcal{N}\left(\mu^{(j)}, \frac{\sigma_\eta^{(j)2}}{1-\phi^{(j)2}}\right), & j &= 1, \ldots, k.
\end{aligned} \tag{7.9}$$

The factors follow an independent stochastic volatility model centred at σ, called the modal volatility. The centred model, with $\sigma = 1$ is written as $f \sim \mathrm{ISV}_N(\phi; \sigma_\eta; \mu)$.

The parameters of the log-volatility ϕ, μ, σ_η can be either scalar values or $k \times 1$ vectors. In the latter case the product $\phi \circ \alpha_{t-1}$ is the Hadamard product. The factor loading matrix β must be a lower-triangular matrix in order to be identifiable ie the upper triangle of β must be zero and the leading diagonal must be 1s [Kaufman and Press, 1973].

A simulation from the factor stochastic volatility model is presented in figure 7.6, with factor loading matrix as in Equation (7.10). The number of factors is chosen to be $k = 2$ and the factors are not correlated, but follow independent AR(1) processes with the same static parameters, $\phi^{(j)} = 0.8$, $\mu^{(j)} = 2.0$ and $\sigma_\eta^{(j)} = 0.1$, $j = 1, 2$.

$$\beta = \begin{pmatrix} 1.0 & 0.0 \\ 0.3 & 1.0 \\ 0.07 & 0.25 \\ 0.23 & 0.23 \\ 0.4 & 0.25 \\ 0.2 & 0.23 \end{pmatrix} \tag{7.10}$$

7.2.1 Parameter Inference in the Factor-SV Model

The unknown parameters consist of $\psi = \{\sigma_y, \beta, \phi^{(j)}, \mu^{(j)}, \sigma_\eta^{(j)}, j = 1, \ldots, k\}$ and the latent-states, $f_{1:N}, \alpha_{0:N}$. The joint distribution of all the random variables in the model is

$$\begin{aligned}
p(y_{1:N}, f_{0:N}, \psi, \alpha_{0:N}) &= p(\psi) \prod_{t=1}^{N} p(y_t | f_t, \Sigma_y, \beta) p(f_t | \alpha_t, \sigma) p(\alpha_t | \alpha_{t-1}, \phi, \mu, \sigma_\eta), \\
&= p(\beta) p(\sigma_y) p(\phi) p(\mu) p(\sigma_\eta) \prod_{t=1}^{N} p(y_t | f_t, \Sigma_y, \beta) p(f_t | \alpha_t) p(\alpha_t | \alpha_{t-1}, \phi, \mu, \sigma_\eta).
\end{aligned}$$

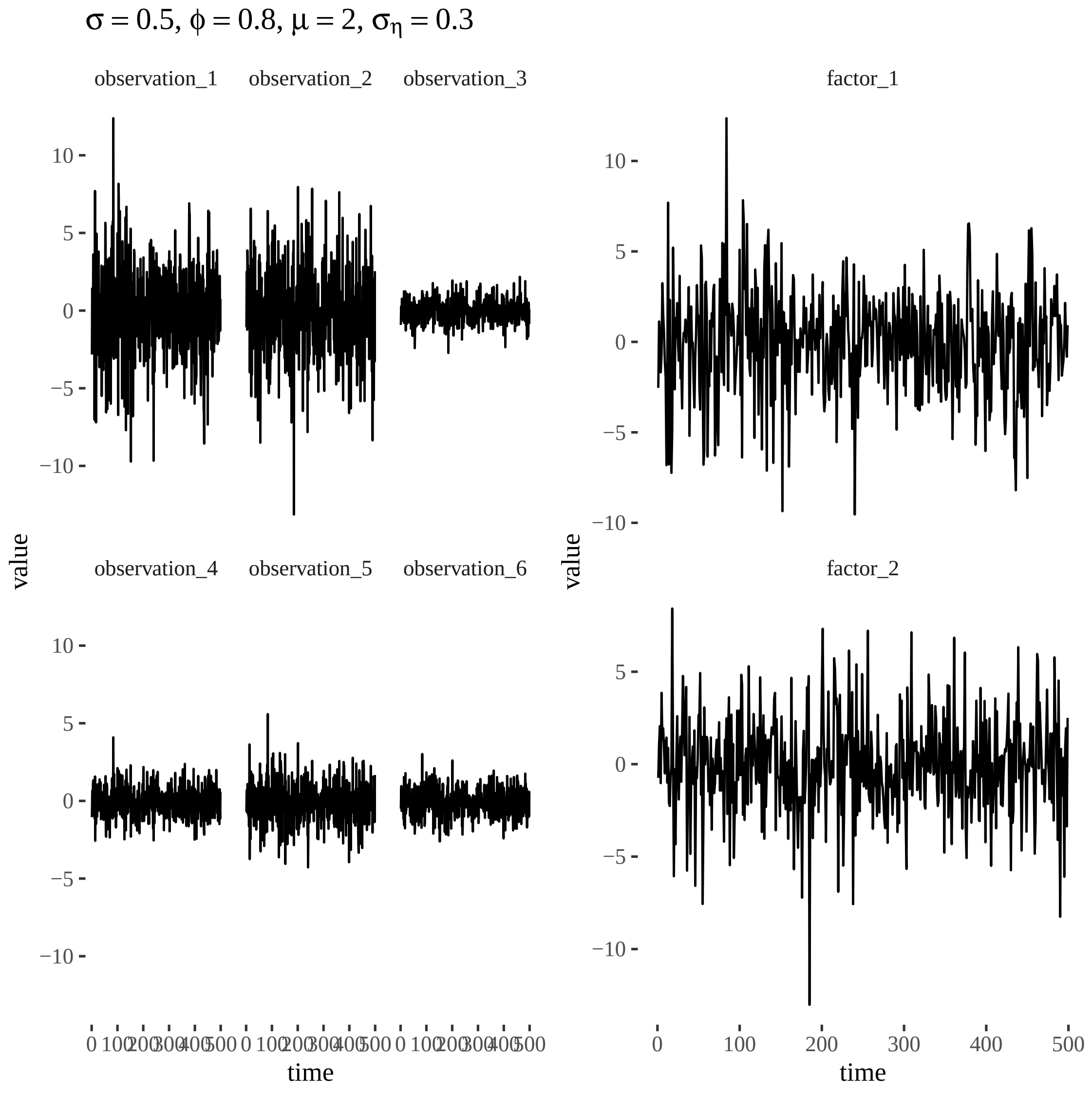


Figure 7.6: (Left) First 500 simulated observations from the factor stochastic volatility model (Right) Simulated factors

This decomposition is possible since the observations are conditionally independent of each other given the corresponding latent factor. The latent log-volatility is Markovian so $p(\alpha_t|\alpha_{1:t-1}) = p(\alpha_t|\alpha_{t-1})$. Finally, the common factors are conditionally independent given the latent log-volatility. To determine the joint posterior distribution $p(\psi, \alpha_{0:N}, f_{1:N}|y_{1:N})$ construct a block Gibbs sampler. First choose prior distributions for the static parameters of the latent-log volatility AR(1) process

$$\begin{aligned}
p(\phi^{(j)}) &= \mathcal{B}(\alpha_\phi, \beta_\phi),\\
p(\sigma_\eta^{(j)}) &= \text{Inv-Gamma}(\alpha_\sigma, \beta_\sigma),\\
p(\mu^{(j)}) &= \mathcal{N}(\mu_\mu, \sigma_\mu^2), j = 1, \dots, k.
\end{aligned}$$

Then choose a prior distribution for the diagonal entries of the observation variance:

$$p(\sigma_y) = \text{Inv-Gamma}(\alpha_\sigma, \beta_\sigma).$$

Independent, element-wise Normal prior distributions are applied to the lower triangle of the factor loading matrix, $\beta_{ij} \sim \mathcal{N}(0, \sigma_\beta^2)$, $j < i$.

The common factors, f_t are sampled conditionally on the factor loading matrix, β and observations $y_{1:N}$. Conditional on the common factors, the posterior distribution for the factor loading matrix and the variance, σ_y is identical to multiple linear regression. Consider sampling from the rows of the factor loading matrix β_i, $i = 1, \dots, p$, for the first $i = 1, \dots, k$ rows, where only $i-1$ elements are non-zero, denote $\Sigma_\beta^{(i)} = \text{diag}(\sigma_\beta^{(1)}, \dots, \sigma_\beta^{(i)})$ then,

$$\begin{aligned}
p(\beta_i|\beta_{-i}, f_t) &= p(\beta_i) \prod_{t=1}^{N} p(y_t|f_t, \sigma_y, \beta),\\
&= \det(2\pi\Sigma_\beta^{(i)})^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}\beta_i^T \Sigma_\beta^{-1(i)} \beta_i\right\} \prod_{t=1}^{N} (2\pi\sigma_y)^{-\frac{1}{2}}\\
&\times \exp\left\{-\frac{1}{2\sigma_y^2}(y_t - \beta f_t)^T (y_t - \beta f_t)\right\},\\
&\propto \exp\left\{-\frac{1}{2}\beta_i^T \Sigma_\beta^{-1(i)} \beta_i - \frac{1}{2\sigma_y^2}\sum_{t=1}^{N}(y_t - \beta f_t)^T (y_t - \beta f_t)\right\},\\
&= \text{MVN}_k(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t).
\end{aligned}$$

β is split into two sets, for the non-zero elements of the rows $i = 1, \dots, k$

$$\begin{aligned}
\boldsymbol{\Sigma}_t^{(i)} &= \left(\frac{1}{\sigma_y^2}\sum_{t=1}^{N} f_{ti} f_{ti}^T + \Sigma_\beta^{-1(i)}\right)^{-1},\\
\boldsymbol{\mu}_t^{(i)} &= \boldsymbol{\Sigma}_t^{(i)} \left(\frac{1}{\sigma_y^2}\sum_{t=1}^{N} f_{ti} y_{ti}\right).
\end{aligned} \tag{7.11}$$

Where y_{ti} is the t^{th} observation of the i^{th} time series, f_{ti} contains the first i elements of the t^{th} latent factor.

Then for the rows $i = k+1, \dots, p$

$$
\begin{aligned}
\boldsymbol{\Sigma}_t^{(i)} &= \left(\frac{1}{\sigma_y^2} \sum_{t=1}^{N} f_t f_t^T + \Sigma_\beta^{-1} \right)^{-1}, \\
\boldsymbol{\mu}_t^{(i)} &= \boldsymbol{\Sigma}_t^{(i)} \left(\frac{1}{\sigma_y^2} \sum_{t=1}^{N} f_t y_{ti} \right).
\end{aligned} \tag{7.12}
$$

The full conditional for the factors can be derived as follows

$$
\begin{aligned}
p(f_t|\beta, \sigma_y, y_t, \alpha_t) &= p(y_t|f_t, \sigma_y, \beta) p(f_t|\alpha_t), \\
&= \text{MVN}_p(y_t; \beta f_t, I_p \sigma_y^2) \text{MVN}_k(f_t; 0, I_k \exp(\alpha_t)) \\
&\propto \exp\left\{ -\frac{1}{2}(y_t - \beta f_t)^T I_p \sigma_y^{-2} (y_t - \beta f_t) - \frac{1}{2} f_t^T I_k e^{-\alpha_t} f_t \right\} \\
&\propto \exp\left\{ -\frac{1}{2} \left(f_t^T \beta^T I_p \sigma_y^{-2} \beta f_t - 2 f_t^T \beta^T I_p \sigma_y^{-2} y_t + f_t^T I_k e^{-\alpha_t} f_t + y_t^T I_p \sigma_y^{-2} y_t \right) \right\} \\
&= \text{MVN}_k((I_k \exp(-\alpha_t) + \beta^T \sigma_y^{-2} \beta)^{-1} \beta^T \sigma_y^{-2} y_t, (I_k \exp(-\alpha_t) + \beta^T \sigma_y^{-2} \beta)^{-1}).
\end{aligned} \tag{7.13}
$$

Then the full conditional for a single variance parameter σ_{yi}^2 is

$$
\begin{aligned}
p(\sigma_{yi}^2 | y_{ti}, f_t, \beta_i) &= p(\sigma_{yi}^2) \prod_{t=1}^{N} p(y_{ti}|f_t, \beta_i), \\
&= \text{Inv-Gamma}\left(\alpha_\sigma + \frac{N}{2}, \beta_\sigma + \frac{1}{2} \sum_{t=1}^{N} (y_{ti} - \beta_i f_t)^2 \right).
\end{aligned} \tag{7.14}
$$

If it is decided that each series shares the same variance, σ_y^2 then the full conditional can be written as:

$$
\begin{aligned}
p(\sigma_y|y_{1:N}, f_{0:N}, \beta) &= p(\sigma_y)\prod_{t=1}^{N} p(y_t|f_t, \beta, \sigma_y) \\
&= \text{Inv-Gamma}(\sigma_y; \alpha_\sigma, \beta_\sigma)\prod_{t=1}^{N}\mathcal{N}(y_t; \beta f_t, \sigma_y^2) \\
&= (\sigma_y^2)^{-\alpha_\sigma-1}\exp\left\{\frac{-\beta_\sigma}{\sigma_y^2}\right\}\det(2\pi I_p\sigma_y)^{-\frac{N}{2}}\exp\left\{\frac{1}{2\sigma_y^2}\sum_{t=1}^{N}(y_t-\beta f_t)^T I_p(y_t-\beta f_t)\right\} \\
&= (\sigma_y^2)^{-\alpha_\sigma-1}\exp\left\{\frac{-\beta_\sigma}{\sigma_y^2}\right\}(2\pi p\sigma_y)^{-\frac{N}{2}}\exp\left\{\frac{1}{2\sigma_y^2}\sum_{t=1}^{N}(y_t-\beta f_t)^T(y_t-\beta f_t)\right\} \\
&\propto (\sigma_y^2)^{-(\alpha_\sigma+\frac{N}{2})-1}\exp\left\{-\frac{1}{\sigma_y^2}\left(\beta_\sigma+\frac{1}{2}\sum_{t=1}^{N}(y_t-\beta f_t)^T(y_t-\beta f_t)\right)\right\} \\
&= \text{Inv-Gamma}\left(\alpha_\sigma+\frac{N}{2}, \beta_\sigma+\frac{1}{2}\sum_{t=1}^{N}(y_t-\beta f_t)^T(y_t-\beta f_t)\right).
\end{aligned}
$$

Algorithm 21 summarised the block Gibbs sampler for a factor stochastic volatility model using the mixture model approximation to sample the latent log-volatility. Equivalently the block sampling method can be used.

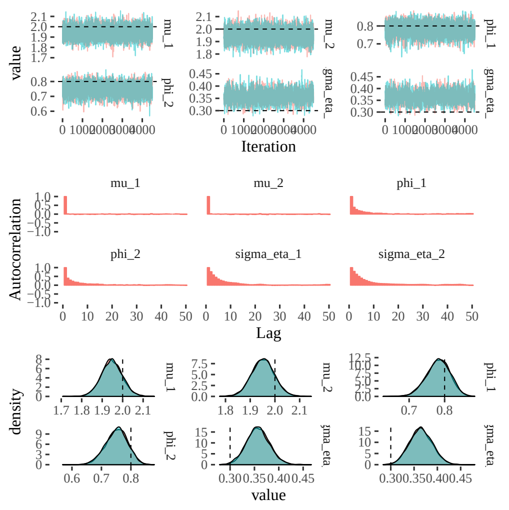
7.2.2 Example: Simulated Factor Stochastic Volatility Model

This example shows parameter inference for the simulated data plotted in figure 7.6. The prior distributions for the static parameters of the factor stochastic volatility model are chosen to be:

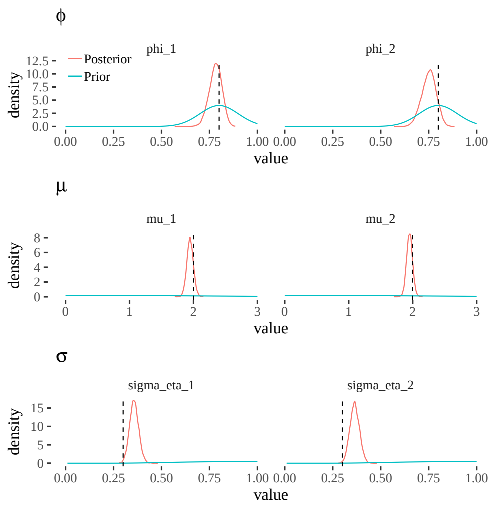
$$
\begin{aligned}
p(\beta_{ij}) &= \mathcal{N}(0.0, 5.0), \\
p(\sigma_y) &= \text{Inv-Gamma}(2.0, 3.0), \\
p(\mu) &= \mathcal{N}(2.0, 1.0), \\
p(\phi) &= \mathcal{N}(0.8, 0.1), \\
p(\sigma_\eta) &= \text{Inv-Gamma}(2.0, 3.0).
\end{aligned}
$$

Table 7.1 shows summary statistics for the AR(1) latent state. Figure 7.7 shows diagnostic plots for the AR(1) latent state parameters.

Figure 7.8 (a) shows the traceplots for the factor loading matrix (b) shows the sample mean and 99% credible intervals of the first factor $f_{0:N}^{(1)}$ calculated from 1,000 draws of the MCMC algorithm. The posterior distribution of the factors very closely matches the simulated factors in the example.

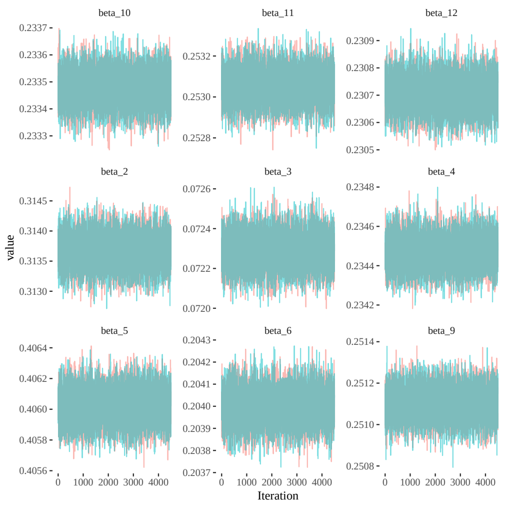


(a)

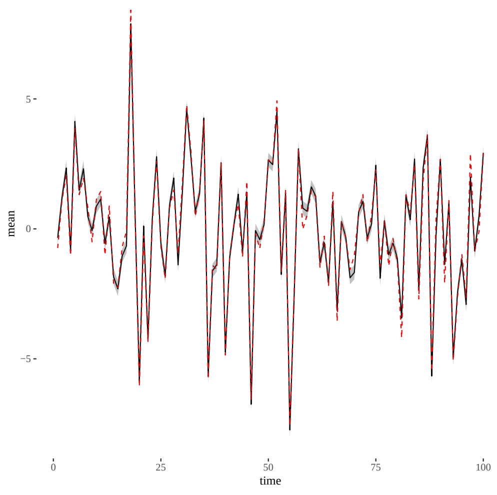


(b)

Figure 7.7: (a) Diagnostic plots for the factor stochastic volatility model for the latent log-volatility parameters (Top) Traceplots (Middle) Autocorrelation (Bottom) Marginal posterior densities (b) Prior and posterior distributions



(a)



(b)

Figure 7.8: (a) Traceplots of the parameters in the factor loading matrix (b) Sampled mean of the first factor and 99% credible intervals calculated from 1,000 MCMC draws, overlayed with the actual simulated state in dashed red

Algorithm 21: Gibbs sampling algorithm for the factor stochastic volatility model

```
Result: Return $\psi_i, i = 1, \dots, M$
1  Given observations $Y_t$, $t = 1, \dots, N$;
2  Initialise the static parameters by drawing from the prior $\psi_0 \sim p(\psi)$;
3  Draw the log-volatilities, $\alpha_{0:N}$ from the prior conditional on $\psi_0$;
4  for $i$ in 1 to $M$ do
5      for $t$ in 1 to $N$ do
6          Sample the $k$-dimensional factors from $p(f_t|\beta, \sigma_y, y_t, \alpha_t)$ using
             equation (7.13);
7      end
8      for $j$ in 1 to $k$ do
9          Sample the $j^{\text{th}}$ log-volatility conditional on the factors and state
             parameters, using FFBS and the Gaussian mixture approximation;
10         Sample the state parameters conditional on the latent log-volatility,
             $\mu^{(j)}, \phi^{(j)}, \sigma^{(j)}$ using eqs. (1.11) and (1.13) ;
11     end
12     for $j$ in 1 to $p$ do
13         Sample $\beta_{i,\cdot}$ using eqs. (7.11) and (7.12);
14     end
15     Sample $\sigma_y^2$ from full conditional in equation (7.14);
16 end
```

Parameter	Mean	Median	Upper (95%)	Lower (5%)	ESS	Actual Value
mu_1	1.95	1.95	2.03	1.86	2761	2.0
mu_2	1.95	1.95	2.03	1.88	3148	2.0
phi_1	0.78	0.78	0.83	0.72	1490	0.8
phi_2	0.75	0.75	0.81	0.69	1280	0.8
sigma_eta_1	0.36	0.36	0.40	0.32	931	0.3
sigma_eta_2	0.36	0.36	0.41	0.33	778	0.3

Table 7.1: Summary statistics of the posterior distribution for the volatility parameters in the simulated factor stochastic volatility model and the value used to simulate the data labelled using "Actual Value"

7.3 Factor Stochastic Volatility with Dynamic Mean

A factor stochastic volatility model with a dynamic mean can be used to model multiple correlated time series with different mean functions [Simpson, 2002]. The dynamic mean can be represented by a dynamic latent state, $\boldsymbol{\theta}_t$, transformed using the observation matrix F_t.

Table 7.2: Summary Statistics of the posterior distribution of the factor loading matrix and the value used to simulate the data labelled "Actual Value"

Parameter	Mean	Median	Upper (95%)	Lower (5%)	ESS	Actual Value
beta_10	0.23	0.23	0.23	0.23	2802	0.23
beta_11	0.25	0.25	0.25	0.25	2913	0.25
beta_12	0.23	0.23	0.23	0.23	2957	0.23
beta_2	0.31	0.31	0.31	0.31	2940	0.30
beta_3	0.07	0.07	0.07	0.07	2884	0.07
beta_4	0.23	0.23	0.23	0.23	3038	0.23
beta_5	0.41	0.41	0.41	0.41	3093	0.40
beta_6	0.20	0.20	0.20	0.20	3005	0.20
beta_9	0.25	0.25	0.25	0.25	2979	0.25

$$\begin{aligned} Y_t &= F_t^T \boldsymbol{\theta}_t + v_t, & v_t &\sim \text{FSV}_N(\beta, \sigma, f_t), \\ \boldsymbol{\theta}_t &= G_t \boldsymbol{\theta}_{t-1} + w_t, & w_t &\sim \text{MVN}_m(0, W), \\ \boldsymbol{\theta}_0 &\sim \text{MVN}_m(\mathbf{m}_0, C_0). \end{aligned} \tag{7.15}$$

$Y_{1:N}$ is a time series of N observations of p-dimensional vectors. v_t is drawn from a factor stochastic volatility model with a $p \times k$ factor loading matrix β and error variance $I_p\sigma$. The model has k factors $f_t^{(i)}$, $i = 1, \ldots, k$ and $v_t = \beta f_t + \varepsilon_t$, $\varepsilon_t \sim \text{MVN}_p(0, I_p\sigma)$ where:

$$f_t = \begin{bmatrix} f_t^{(1)} \\ \vdots \\ f_t^{(k)} \end{bmatrix}.$$

The factors are driven by a stochastic volatility process with an AR(1) latent state representing the log-volatility: $f_t^{(i)} \sim \text{ISV}_N(\phi^{(i)}; \mu^{(i)}; \sigma_\eta^{(i)})$. The observation matrix, F_t is $m \times p$. The latent-state is m-dimensional and w_t is the system error at time t drawn from a multivariate Normal distribution with zero mean and covariance $W = \text{diag}(w_1, \ldots, w_m)$.

The multivariate non-centered model written out in full is

$$
\begin{aligned}
Y_t &= F_t^T \boldsymbol{\theta}_t + \beta f_t + \varepsilon_t, & \varepsilon_t &\sim \text{MVN}_p(0, I_p\sigma_y^2),\\
\boldsymbol{\theta}_t &= G_t \boldsymbol{\theta}_{t-1} + w_t, & w_t &\sim \text{MVN}_m(0, W),\\
f_t^{(i)} &= \sigma_t^{(i)} \exp\left\{\frac{\alpha_t^{(i)}}{2}\right\}, & \sigma_t^{(i)} &\sim \mathcal{N}(0,1),\\
\alpha_t^{(i)} &= \mu^{(i)} + \phi^{(i)}(\alpha_{t-1}^{(i)} - \mu^{(i)}) + \eta^{(i)}, & \eta^{(i)} &\sim \mathcal{N}(0, \sigma_\eta^{2(i)})\\
\alpha_0^{(i)} &\sim \mathcal{N}\left(\mu^{(i)}, \frac{\sigma_\eta^{(i)2}}{1-\phi^{(i)2}}\right), i = 1, \dots, k, & &\\
\boldsymbol{\theta}_0 &\sim \text{MVN}_m(\mathbf{m}_0, C_0). & &
\end{aligned}
$$

Figure 7.9 shows a simulation from a model with a first order DLM representing the mean of the process.

7.3.1 Parameter inference in the non-centered model

In order to learn the posterior distribution of the parameters in the DLM-FSV model first write the joint distribution of all the random variables in the model

$$
\begin{aligned}
p(\psi, &\boldsymbol{\theta}_{0:N}, \alpha_{0:N}, f_{1:N}, y_{1:N}) = p(\psi)p(\boldsymbol{\theta}_0)p(\alpha_0)\prod_{t=1}^{N} p(y_t|f_t, \boldsymbol{\theta}_t, I_p\sigma_y)p(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}, W)p(f_t|\alpha_t)\\
&\times p(\alpha_t|\alpha_{t-1}, \phi, \mu, \sigma_\eta)\\
&= p(\psi)\mathcal{N}(m_0, C_0)\mathcal{N}\left(\mu, \frac{\sigma_\eta^2}{1-\phi^2}\right)\prod_{t=1}^{N}\mathcal{N}(y_t; F_t^T\boldsymbol{\theta}_t + \beta f_t, I_p\sigma_y^2)\mathcal{N}(\theta_{t+1}; G_t\theta_t, W)\\
&\times \mathcal{N}(f_t; 0, \exp(\alpha_t))\mathcal{N}(\alpha_t; \mu + \phi(\alpha_{t-1} - \mu), \sigma_\eta^2)
\end{aligned}
$$

A block Gibbs sampler is used to learn the joint posterior distribution of the latent states and associated static parameters, $p(\psi, \boldsymbol{\theta}_{0:N}, \alpha_{0:N}, f_{1:N}|y_{1:N})$. First consider determining the factors conditional on the observations and the mean latent state, $y_{1:N}$ and $\boldsymbol{\theta}_{0:N}$ respectively. Then the observation equation can be re-written as

$$
y_t - F_t\boldsymbol{\theta}_t = \beta f_t + v_t, \quad v_t \sim \text{MVN}_p(0, I_p\sigma_y^2). \tag{7.16}
$$

This is a factor stochastic volatility model and inference for the factor loading matrix, log-volatility, common factors and unexplained variance $(\beta, \alpha_{0:N}, f_{1:N}, \sigma_y)$ can be performed as detailed in section 7.2.1.

It remains to determine the filtering distribution $p(\boldsymbol{\theta}_{0:N}|y_{1:N}, \psi)$ and the system co-

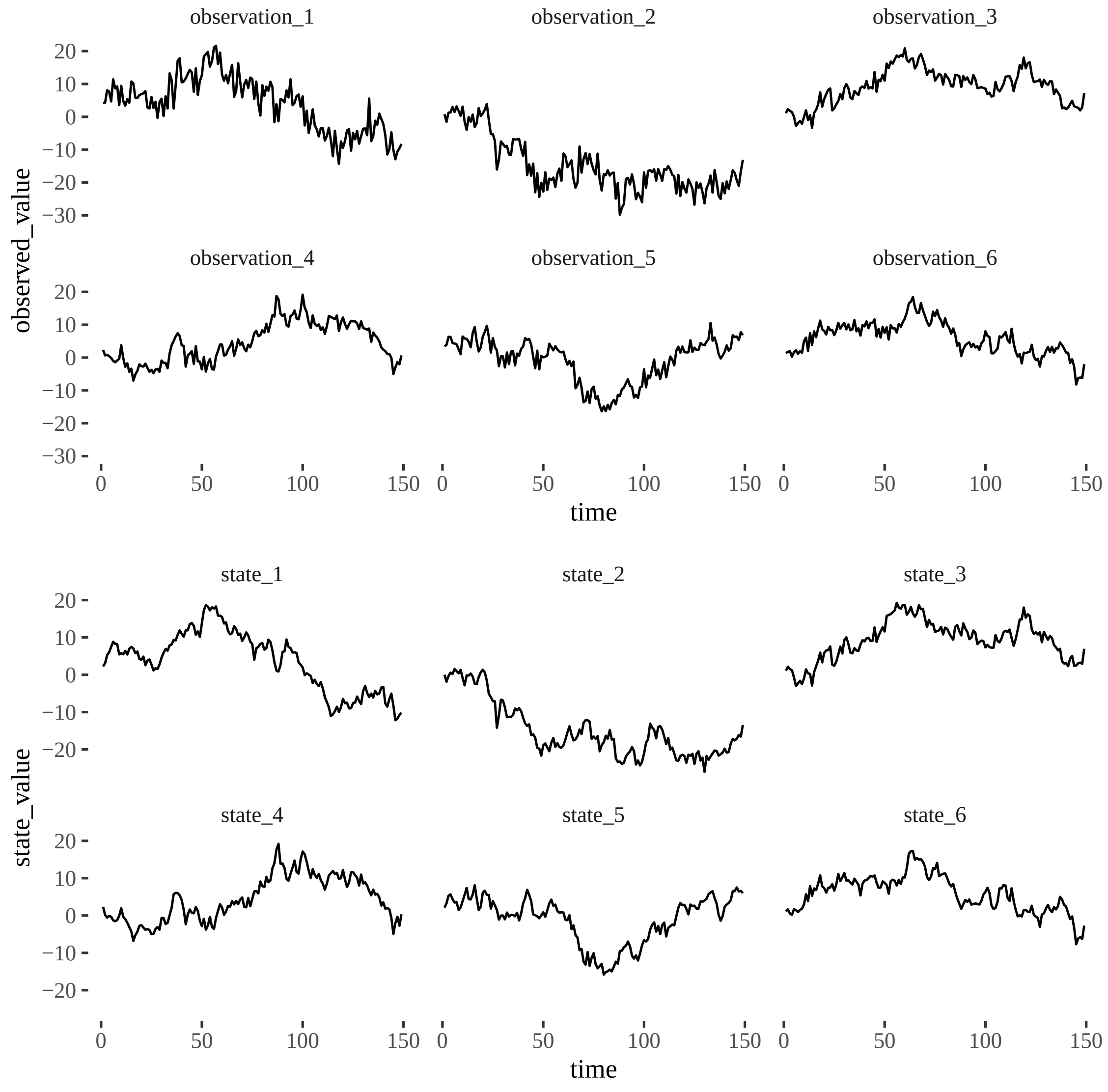


Figure 7.9: Simulated non centered factor stochastic volatility model (DLM FSV) (Top) Simulated observations (Bottom) Mean state

variance matrix. The full conditional distributions for the remaining static parameter, the system noise covariance matrix, W can be derived as detailed in equation (5.10) by specifying an inverse Gamma semi-conjugate prior distribution.

To sample the mean latent-state ($\boldsymbol{\theta}_{0:N}$) conditional on the other parameters re-write the model equations as:

$$
\begin{aligned}
Y_t - \beta f_t &= F_t \boldsymbol{\theta}_t + v_t, & v_t &\sim \text{MVN}_p(0, I_p \sigma_y^2), \\
\boldsymbol{\theta}_t &= G_t \boldsymbol{\theta}_{t-1} + w_t, & w_t &\sim \text{MVN}_m(0, W), \\
\boldsymbol{\theta}_0 &\sim \text{MVN}_m(\mathbf{m}_0, C_0).
\end{aligned}
$$

Then the full conditional of the latent-state can be sampled from using FFBS. The MCMC algorithm for the DLM FSV is summarised in algorithm 22.

Algorithm 22: Gibbs sampling algorithm to determine the marginal posterior distribution of the static parameters of the factor stochastic volatility model with dynamic mean

```
   Result: Return $\psi_i, i = 1, \dots, M$
 1 Initialise the static parameters by sampling from the prior $\psi_0 \sim p(\psi)$;
 2 Initialise the latent state $\boldsymbol{\theta}_{0:N}$ by sampling from the prior;
 3 Initialise the latent log-volatility $\alpha_{0:N}$ by sampling from the prior;
 4 for i in 1 to M do
 5     Calculate the centred observations, $y_t - F_t\theta_t = \beta f_t + v_t$ to recognise an FSV
         model;
 6     Perform parameter inference as in the FSV model, algorithm 21;
 7     Calculate the residuals $Y_t - \beta f_t = F_t\boldsymbol{\theta}_t + v_t$ to recognise a DLM;
 8     Sample the state, $\boldsymbol{\theta}_{0:N}$ using FFBS;
 9     Sample system covariance matrix $W$ from the full conditional in
         equation (5.10);
10 end
```

Alternatively (or additionally) a factor structure can be used when modelling the system matrix, an extension to the model considered in Simpson [2002]. The system matrix, W, can grow very large when modelling complex univariate or multivariate time series. Learning a non-diagonal W matrix using a Gibbs step with an inverse Wishart prior can be challenging. The state-space model with a factor structure proposed for the system matrix can be written as:

$$
\begin{aligned}
Y_t &= F_t^T \boldsymbol{\theta}_t + v_t, & v_t &\sim \text{MVN}_p(0, I_p \sigma_y^2), \\
\boldsymbol{\theta}_t &= G_t \boldsymbol{\theta}_{t-1} + \beta f_t^{(i)} + \tau_t, & \tau_t &\sim \text{MVN}_m(0, I_m \sigma_x^2), \\
\boldsymbol{\theta}_t &\sim \text{MVN}_m(\mathbf{m}_0, C_0), \\
f_t^{(i)} &\sim \text{ISV}_N(\phi^{(i)}; \sigma_\eta^{(i)}; \mu^{(i)}), \quad i = 1, \dots, k.
\end{aligned}
$$

As before, $Y_t, t = 1, \dots, N$ is a time series of N observations, each Y_t is a p-dimensional vector. The observation and system matrices, F_t and G_t respectively are created by composing models as described in section 4.1.1 and 4.1.2. The latent-state is m-dimensional with k dynamic factors, f_t and a $m \times k$ factor loading matrix β, together representing the time dependent variance of the latent-state. Figure 7.10 shows a simulation from a univariate state space model with seasonal structure. The parameters of the latent log-volatility are $\phi^{(i)} = 0.8, \mu^{(i)} = 0.0, \sigma_\eta^{(i)} = 0.2$, the observation variance is $V = 0.5$ and the unexplained variance is $I_{13}\sigma_x$ where $\sigma_x = 0.1$, finally the factor loading matrix is

$$\beta = \begin{pmatrix} 1.00 & 0.00 \\ 4.94 & 1.00 \\ 3.38 & 2.74 \\ 5.00 & 0.95 \\ -4.57 & -7.40 \\ -0.12 & 8.20 \\ 1.35 & -2.80 \\ -10.61 & -2.20 \\ -1.39 & -0.75 \\ 2.15 & 0.31 \\ -1.16 & 1.46 \\ 2.07 & 2.18 \\ 12.52 & -4.00 \end{pmatrix} \tag{7.17}$$

Again, a Gibbs sampling algorithm is used to determine the joint posterior distribution $p(\psi, \boldsymbol{\theta}_{0:N}, \alpha_{0:N}, f_{1:N}|y_{1:N})$ of the non-centered model with dynamic system noise covariance. The joint distribution of the random variables in the model can be written as:

$$\begin{aligned} p(y_{1:N}, &\boldsymbol{\theta}_{0:N}, \alpha_{0:N}, f_{1:N}, \psi) = p(\psi)p(\boldsymbol{\theta}_0)p(\alpha_0)\prod_{t=1}^{N} p(y_t|\boldsymbol{\theta}_t, \sigma_y)p(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}, \beta, \sigma_x, f_t) \\ &\times p(f_t|\alpha_t)p(\alpha_t|\alpha_{t-1}, \phi, \mu, \sigma_\eta) \\ &= p(\psi)\mathcal{N}(m_0, C_0)\mathcal{N}\left(\mu, \frac{\sigma_\eta^2}{1-\phi^2}\right)\prod_{t=1}^{N}\mathcal{N}(y_t; F_t^T\boldsymbol{\theta}_t, I_p\sigma_y^2)\mathcal{N}(\boldsymbol{\theta}_{t+1}; G_t\boldsymbol{\theta}_t + \beta f_t, I_m\sigma_x) \\ &\times \mathcal{N}(f_t; 0, \exp(\alpha_t))\mathcal{N}(\alpha_t; \mu + \phi(\alpha_{t-1} - \mu), \sigma_\eta^2). \end{aligned}$$

Consider deriving the full conditional distributions for the static parameters in the model, $\psi = \{\sigma_y, \phi^{(i)}, \mu^{(i)}, \sigma_\eta^{(i)}, \beta, \sigma_x\}, i = 1, \dots, k$ conditional on the observations, $y_{1:N}$, latent-

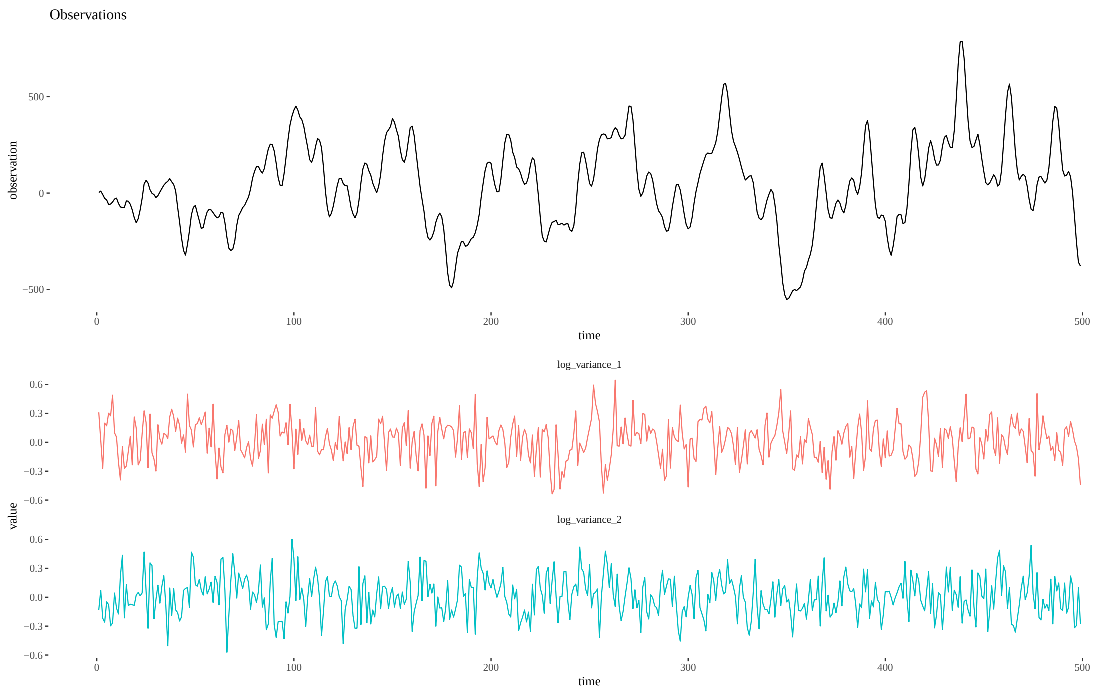


Figure 7.10: Simulated data from a state space model with a dynamic system noise covariance matrix.

states, $\alpha_{0:N}$, $\theta_{0:N}$ and common factors $f_{1:N}$.

The full condition of the observation variance is equivalent to the DLM and derived in equation 4.17. The full conditionals for the parameters of the latent log-volatility can be written as in section 1.3.3 for the AR(1) model. The full conditional distributions for the factor loading matrix, β, common factors $f_{1:N}$ and the unexplained variance σ_x are identical to the factor stochastic volatility model presented in section 7.2.1.

It remains to consider how to sample the latent states, $\alpha_{0:N}$ and $\theta_{0:N}$. The full conditional distribution for the mean latent-state is

$$p(\boldsymbol{\theta}_{0:N}|y_{1:N}, \alpha_{0:N}, f_{1:N}, \psi) = p(\theta_0) \prod_{t=1}^{N} p(y_t|\theta_t, \sigma_y) p(\theta_t|\theta_{t-1}, \beta, f_t, \sigma_x), n$$

which can be written as:

$$p(\boldsymbol{\theta}_{0:N}|y_{1:N}, \alpha_{0:N}, f_{1:N}, \psi) = \prod_{t=0}^{N} p(\theta_t|\theta_{t+1:N}, y_{1:N}, \beta, f_t, \sigma_x).$$

The final factor in the product is, $p(\theta_N|y_{1:N}, \beta, \sigma_x, f_N)$ which is the filtering distribution of θ_N, which is determined using the Kalman filter (see section 4.2). The posterior at time $t-1$ is Gaussian and specified by mean and variance m_{t-1} and C_{t-1} then the Kalman filter equations, conditional on the values of the factor loading matrix, common factors and unexplained variance, required to calculate the posterior at time t for the non-centered FSV model are

$$\begin{aligned}
\text{Prior:} \qquad & a_t = G_t m_{t-1} + \beta f_t \\
& R_t = G_t C_{t-1} G_t^T + \sigma_x^2 \\
\text{Update:} \qquad & m_t = a_t + A_t e_t \\
& C_t = A_t V, \\
\text{where} \qquad & A_t = R_t/(R_t + V), \quad e_t = \tilde{y}_t - a_t.
\end{aligned}$$

The backward sampler can then be performed by first simulating $\theta_N \sim \mathcal{N}(m_N, C_N)$ and then simulating recursively for $t = N-1, N-2, \dots, 0$ according to equation (4.19).

The latent log-volatility is sampled conditional on the value of the common factors, mean latent-state and other static parameters using either the mixture model approximation from section 7.1.2 or the blocking described in section 7.1.1. The centered state is calculated in order to recognise an FSV model

$$e_t = \boldsymbol{\theta}_t - G_t \boldsymbol{\theta}_{t-1} = \beta f_t + I_m \sigma_x$$

The remaining static parameters, $\psi = \{\sigma_x, \beta, \phi^{(j)}, \sigma_\eta^{(j)}, \mu^{(j)}\}$, $j = 1, \dots, k$ can be sampled using the full-conditional distributions derived for the factor stochastic volatility model in section 7.2.1 the full block Gibbs sampler is summarised in algorithm 23.

Algorithm 23: MCMC algorithm for the DLM with factor structure for the system noise covariance matrix.

```
  Result: Return $\psi_i, i = 1, \dots, M$
1 Initialise the static parameters, $\psi_0 \sim p(\psi)$;
2 Initialise the latent state $\boldsymbol{\theta}_{0:N}$ by sampling from the prior;
3 Initialise the latent log-volatility by sampling from the prior;
4 for $i$ in 1 to $M$ do
5     Use FFBS to sample the state $p(\boldsymbol{\theta}_{0:N}|\psi, V)$;
6     Calculate the centered state $e_t = \boldsymbol{\theta}_t - G_t\boldsymbol{\theta}_{t-1}$;
7     Sample the log-volatility and static parameters from the FSV model
        $e_t = \beta f_t + \tau_t, \tau_t \sim \text{MVN}_d(0, I_m\sigma_x)$ using algorithm 21;
8     Sample the observation variance $\sigma_y$ from the full conditional distribution in
        equation (4.17);
9 end
```

7.4 Example: Environment Sensor

This example considers sensor readings at a single sensor, *"new new emote 2604"* using the FSV with dynamic mean model. The sensor is located in Newcastle upon Tyne near the Urban Sciences Building and records NO, CO, humidity and temperature approximately every five minutes. The system matrix is considered to follow a factor stochastic volatility model. The discretely observed hourly sensor data introduced in section 4.9 is used, the training data is from January 2018 until August 2018. The factor loading matrix is now $28 \times k$ matrix, where $k = 2$ is the number of factors.

In order to jointly model the measurements made using sensor 2604 a model for each dynamic mean process is defined. The specification of the model matrices, F and G is the same as the multivariate DLM example considered in section 4.9. The model equation is:

$$\begin{aligned}
Y_t &= F^T x_t + v_t, & v_t &\sim \mathcal{N}(0, V), \\
x_t &= G x_{t-1} + w_t, & w_t &\sim \text{FSV}(\beta, \sigma, f_t), \\
x_0 &\sim \mathcal{N}(m_0, C_0).
\end{aligned}$$

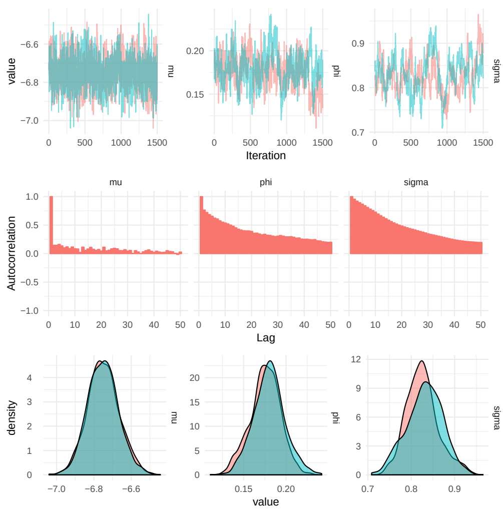
The prior distributions for the static parameters are:

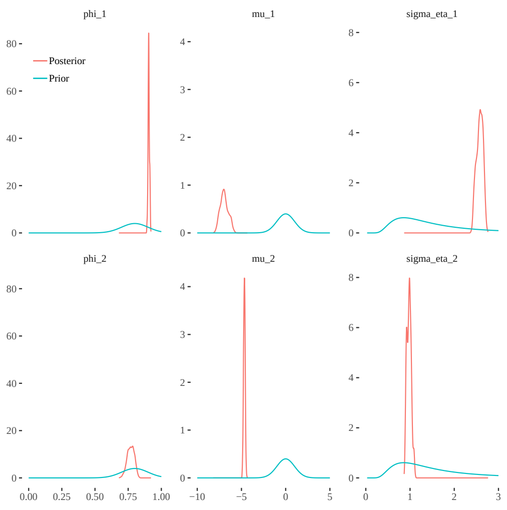
$$\begin{aligned}
p(\beta_{ij}) &= \mathcal{N}(0, 3), \\
p(\phi_i) &= \mathcal{N}(0.8, 0.1), \\
p(\mu_i) &= \mathcal{N}(0, 1), \\
p(\sigma_{\eta i}) &= \text{Inv-Gamma}(2.5, 3), \\
p(\sigma_j) &= \text{Inv-Gamma}(2.5, 3), \\
p(\text{diag}(V)) &= \text{Inv-Gamma}(2.5, 3), \\
j = 1, \dots, m i &= 1, \dots, k.
\end{aligned}$$

Here, $m = 28$ is the dimension of the latent-state and $k = 2$ is the number of factors.

40,000 iterations of the MCMC were run, with 10,000 dropped as burn-in with a thinning of 20 before plotting the diagnostics for posterior distributions of the parameters. Figure 7.11 (a) shows the trace-plots and empirical density for the posterior distribution of the latent-factor parameters and (b) shows the prior and posterior distributions for the static parameters of the log-volatility process. Figure 7.13 shows the diagnostics for the factor loading matrix, β.

Figure 7.12 shows interpolated values and 90% probability intervals for the four measurements made at sensor 2604 during the month of September 2018. The mean of the interpolated values appear to be plausible for all the values the sensor records. The uncer-


(a)



(b)

Figure 7.11: (a) Diagnostic plots for log-volatility parameters (Top) Trace-plots (Middle) Autocorrelation plots (Bottom) Empirical posterior Density (b) Prior and posterior distributions for the parameters of the stochastic volatility latent state for the non-centered factor stochastic volatility model with factor structure system matrix.

tainty for the CO data is large, a log-transform on the data could result in an improved fit as shown in example 4.9.

7.5 Summary

This chapter has considered a class of models with time dependent variance modelled using a latent log-Gaussian process. The class was extended and combined with the dynamic models considered in chapters 4 to 6. The algorithms described in this chapter have been implemented in the Bayesian DLMs scala package[1].

The mean models introduced are Gaussian and can take advantage of exact filtering using the Kalman Filter. However this prevents modelling of heteroscedastic time series with non-linear and non-Gaussian dynamics. It would be straightforward to extend the observation distribution to a Student-t using the Gamma mixture of Normal distributions presented in section 5.4.

The non-centered models could be further extended to non-linear, non-Gaussian models using particle Gibbs sampling, by sampling the latent-state of the mean process $\theta_{0:T}$ using a particle filter and sampling the latent log-volatility conditional on the mean latent-state using one of the methods described in this chapter.

[1]https://git.io/dlms

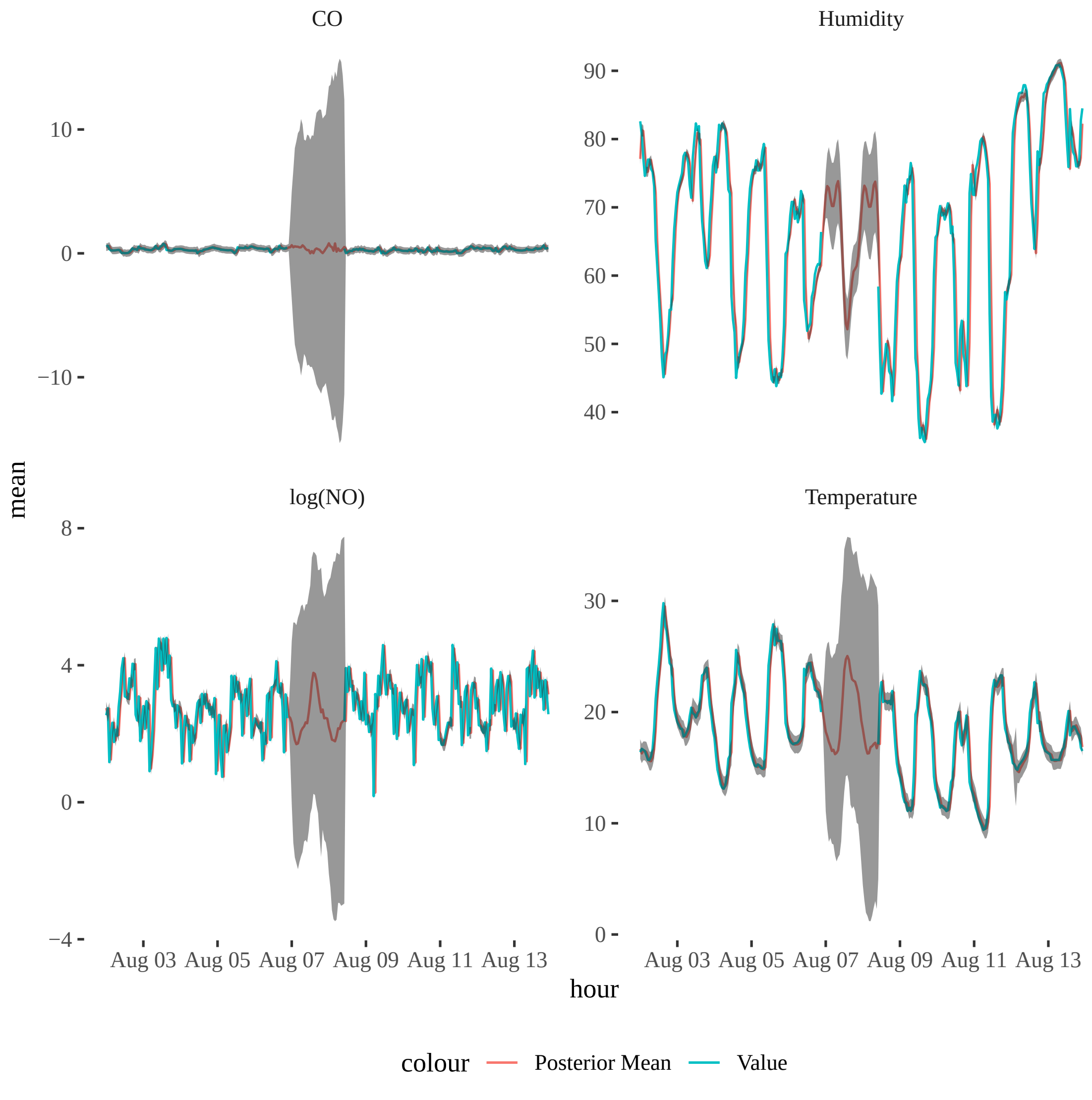


Figure 7.12: Interpolated values from the non centered factor stochastic volatility model for sensor new new emote 2604

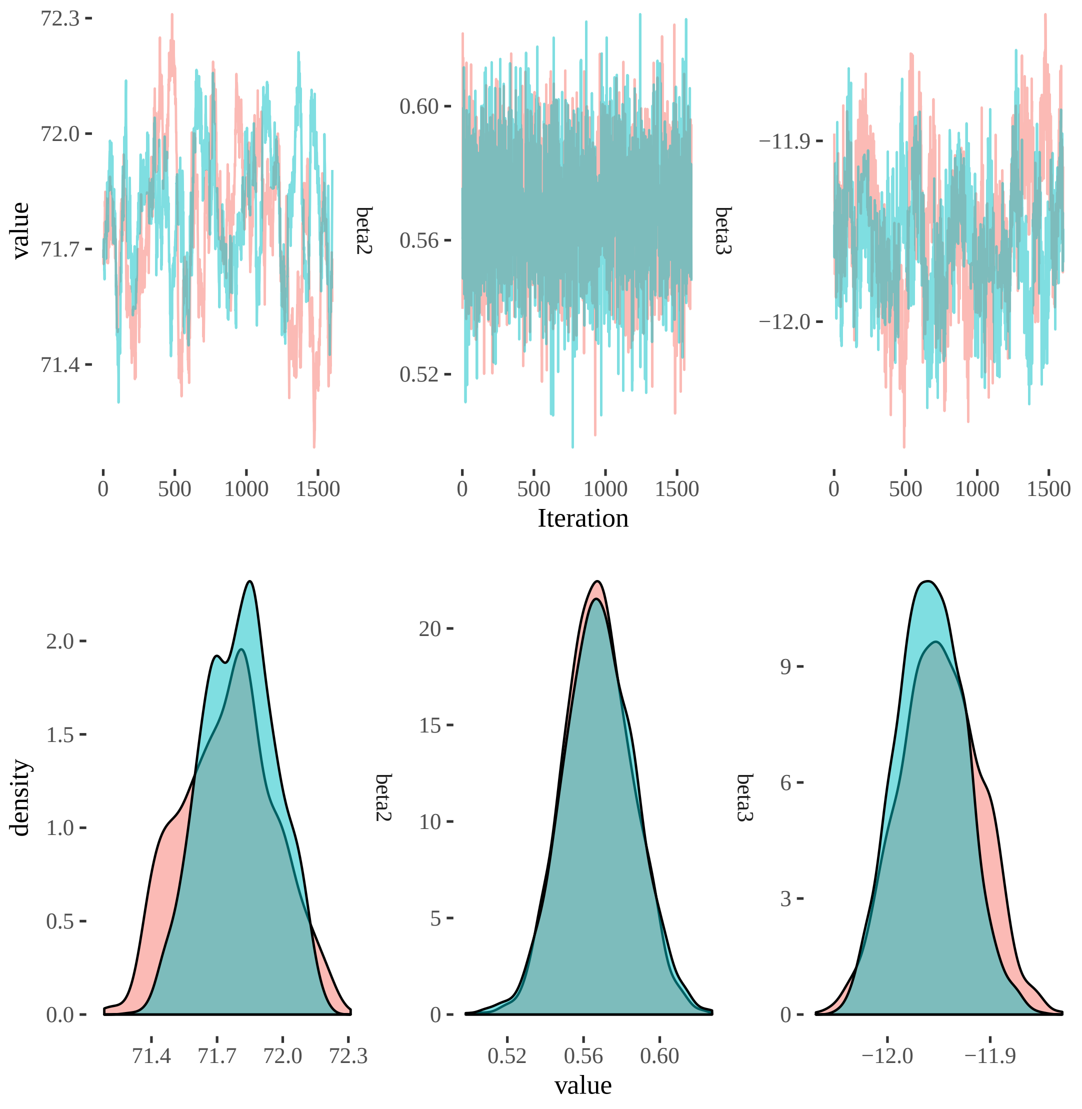


Figure 7.13: Parameter diagnostics, trace-plots and densities for the non-zero elements of the factor loading matrix

Chapter 8

Spatial Modelling with Gaussian Processes

Sensors in the Urban Observatory are deployed in a dense network across the north east of England. So far, time series models have been considered for individual sensors however sometimes an individual sensor can fail to record observations for a prolonged period due to hardware defects or network connectivity issues. In this scenario similar sensors in the vicinity can be used to interpolate the missing measurements. In addition, sensors are often placed at convenient locations rather than locations of interest, hence interpolation is required to understand how the process behaves across the whole space.

One method of modelling spatial dependence is using a Gaussian process. A Gaussian process (GP) is an infinite collection of points, of which finitely many points are distributed according to a multivariate Normal distribution. Gaussian processes can be used to learn non-linear relationships of the form $y = f(x)$, where the Gaussian process can approximate the latent function f and the corresponding uncertainty given the observations. Gaussian processes are a non-parametric method, so minimal model specification is required to learn the latent function.

Gaussian processes have been used successfully in regression and classification problems, including modelling spatial data [Cressie, 1992]. Optimal positioning of sensors is a problem which can be tackled using GPs; by positioning sensors in areas which have high posterior variance or by maximising the mutual information between sensors [Krause et al., 2008]. GPs have been successfully applied in epidemiology to assess the global burden of disease, to spatially interpolate missing covariates [Murray et al., 2012]. GPs have also been used to model climate and weather data [Osborne and Roberts, 2007].

This chapter focuses on how to fit Gaussian process regression models efficiently to a small number of data points and considers how to model spatial differences in a sensor array in the Urban Observatory in section 8.3. For an in depth text on Gaussian Process models for classification and regression consult Williams and Rasmussen [2006].

8.1 Gaussian Process

Consider a set of pairs of independent and dependent variables (X_i, Y_i), $i = 1, \ldots, N$ then X_i is assumed to be related to Y_i by a latent function f and the observations of the dependent variables, Y_i, are corrupted by Gaussian noise

$$Y_i = f(X_i) + \varepsilon_i, \quad \varepsilon_i \sim \mathcal{N}(0, \sigma_y). \tag{8.1}$$

where $Y_i \in \mathbb{R}^p$ and $X_i \in \mathbb{R}^m$. The function f is considered to be drawn from a Gaussian process with a mean function $\boldsymbol{\mu}(X)$ and covariance function, $k(X, X^\star)$. The mean and covariance functions have associated hyper-parameters which control the function f, the posterior distribution of these hyper-parameters can be determined conditional upon observed data using appropriate inference algorithms.

An example of a covariance function is the squared exponential covariance function which is parameterised by a length scale σ, signal variance h and a distance metric $d(x_i, x_j)$:

$$k(x_i, x_j) = h \exp\left(-\frac{d(x_i, x_j)}{2\sigma^2}\right).$$

The length scale σ determines how much the observations change at different values of the independent variables, i.e. how different y_1 is from y_2 given a fixed distance $d(x_1, x_2)$ between the two observations. Intuitively, in a spatial model where each x_i represents a location in space, if σ is large then the observations, y_1 and y_2 change slowly as the distance between x_1 and x_2 increases (as $d(x_1, x_2) \to \infty$) however if σ is small then a small change in distance (in any direction) results in a noticable change in the observation. In spatial analysis strong prior opinions can be formed for σ. The signal variance, h, controls the amount the dependent variables change. The squared exponential covariance function is isotropic, this means the covariance between two points depends only on the distance between them. Weather effects such as the wind are directional and hence the squared exponential covariance function is not appropriate.

A covariance function is used to build a covariance matrix which then represents the measurements $y(x)$, the observation y at each location in the vector x, as a draw from a multivariate Normal distribution

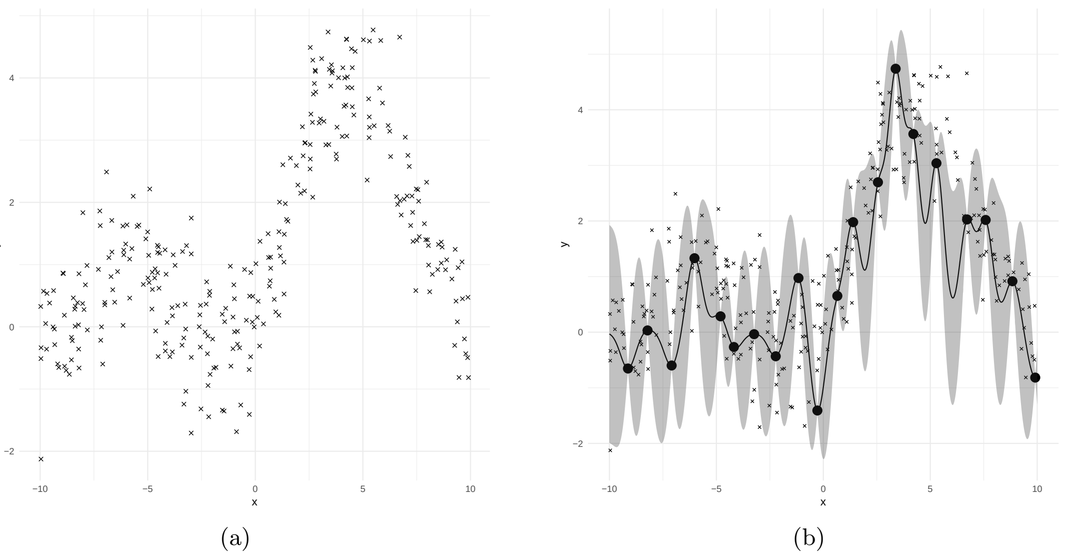


Figure 8.1: (a) 300 simulated values from a Gaussian process with parameter values $\sigma_y = 0.09$, $h = 1.0$ and $\sigma = 2.0$. (b) Posterior mean of the fitted function with 95% probability intervals in shaded grey, the observed values are large round circles with other unobserved draws from the same Gaussian process plotted as small crosses.

$$K(\mathbf{x}, \mathbf{x}) = \begin{pmatrix} k(x_1, x_1) & k(x_1, x_2) & \dots & k(x_1, x_n) \\ k(x_2, x_1) & k(x_2, x_2) & \dots & k(x_2, x_n) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_n, x_1) & k(x_n, x_2) & \dots & k(x_n, x_n) \end{pmatrix}$$

$$p(\mathbf{y}(\mathbf{x})) = \text{MVN}(m(\mathbf{x}), K(\mathbf{x}, \mathbf{x}) + I\sigma_y^2)$$

The distance metric d is problem dependent and should be chosen appropriately. For small scale spatial modelling Euclidean distance is approximately correct, however for large distances on the globe great circle distance is more appropriate.

$m(\mathbf{x})$ is the mean function of the multivariate Normal distribution. A small amount of noise, $I_p\sigma_y^2$ is added to the covariance of the multivariate Normal distribution to represent the measurement error of $y(x)$. Figure 8.1 shows a single draw from a Gaussian process with a squared exponential covariance function, with $h = 1.0$ and length scale $\sigma = 2.0$, the variance of the observations is $\sigma_y = 0.5$, the mean function is zero.

Properties of the multivariate Normal distribution can be used in order to calculate the posterior prediction at an unobserved value of the independent variable, $x^\star$. The joint density of $\mathbf{y}(\mathbf{x})$ and $f(x^\star)$ is

$$p\left(\begin{bmatrix}\mathbf{y}(\mathbf{x})\\ f(x^\star)\end{bmatrix}\right) = \text{MVN}\left(\begin{bmatrix}m(\mathbf{x})\\ m(x^\star)\end{bmatrix}, \begin{bmatrix}K(\mathbf{x},\mathbf{x}) + I_p\sigma_y^2 & K(\mathbf{x}, x^\star)\\ K(x^\star, \mathbf{x}) & k(x^\star, x^\star)\end{bmatrix}\right). \tag{8.2}$$

Then the posterior $p(f(x^\star)|f(\mathbf{x}), \mathbf{y}, x^\star)$ is Gaussian and hence completely specified by its mean and covariance

$$\begin{aligned}\mathbb{E}(f(x^\star)) &= m(x^\star) + K(x_\star, \mathbf{x})(K(\mathbf{x},\mathbf{x}) + I_p\sigma_y^2)^{-1}(\mathbf{y} - m(\mathbf{x})),\\ \text{Var}(f(x^\star)) &= k(x_\star, x_\star) - K(x_\star, \mathbf{x})(K(\mathbf{x},\mathbf{x}) + I_p\sigma_y^2)^{-1}K(x_\star, \mathbf{x}).\end{aligned}$$

which can be extended to learning the posterior distribution at many points. To calculate the predictive distribution of the observed point $y(x^\star)$ then add σ^2 to the covariance. Figure 8.1 (b) summarises the posterior distribution of the function $f(x)$, the mean is plotted with a solid line and 95% probability intervals are represented by the shaded grey area. The observed data consists of every 10^{th} observation from the simulated Gaussian process in figure 8.1 (a). The observed data is highlighted using larger points with the unobserved draws from the Gaussian process plotted as smaller crosses.

8.2 Parameter Inference

Different values of the hyper-parameters affect the smoothness of the latent function $f(x)$. This section considers how to determine the posterior distribution of the hyper-parameters of the covariance and mean functions. This allows posterior predictions to be drawn from the Gaussian process representing the latent function $f(x)$, accounting for hyper-parameter uncertainty.

Consider observations $y(\mathbf{x}) \in \mathbb{R}$ at locations $\mathbf{x} = (x_1, \dots, x_n)$, corrupted with independent Gaussian noise with covariance $I_p\sigma_y^2$ as in equation 8.1. A GP prior is placed on the function f with a squared exponential covariance function and a linear mean function, $m(\mathbf{x}) = X\beta$ where $\beta \in \mathbb{R}^{2\times 1}$ and $X \in \mathbb{R}^{n\times 2}$ is a design matrix

$$X = \begin{pmatrix}1 & x_1\\ \vdots & \vdots\\ 1 & x_n\end{pmatrix}.$$

Then the model can be specified as

$$\begin{aligned} y(\mathbf{x}) &= m(\mathbf{x}) + f(\mathbf{x}) + I_p\sigma_y^2, \\ f(\mathbf{x}) &\sim \mathcal{GP}(0, K(\mathbf{x},\mathbf{x})), \end{aligned}$$

and in matrix form

$$\begin{aligned} \begin{bmatrix} Y_1 \\ \vdots \\ Y_n \end{bmatrix} &= \begin{pmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix} \beta + f(\mathbf{x}) + \varepsilon_y, \quad \varepsilon_y \sim \mathcal{N}(0, I_n\sigma_y^2) \\ f(\mathbf{x}) &\sim \mathcal{GP}(0, K_y). \end{aligned}$$

An MCMC algorithm (see section 1.3.2) can be used to determine the posterior distribution of the hyper-parameters, ψ in the covariance and mean of the Gaussian process representing the latent function, f. The following is an Metropolis-Hastings algorithm to determine the posterior distribution of the hyper-parameters with a linear mean function.

The parameters of the model are $\psi = \{\sigma_y, \sigma, h, \boldsymbol{\beta}\}$, then the joint distribution of the random variables in the model can be written as

$$\begin{aligned} p(y(\mathbf{x}), \psi, f(\mathbf{x})) &= p(\psi)p(f(\mathbf{x})|h,\sigma)p(y(\mathbf{x})|f(\mathbf{x}), \beta, \sigma_y), \\ &= p(\psi)\mathrm{MVN}_p(f(\mathbf{x})|0, K(\mathbf{x},\mathbf{x}))\mathrm{MVN}_p(y(\mathbf{x})|X\beta + f(\mathbf{x}), I_p\sigma_y^2)). \end{aligned}$$

The prior and proposal distributions for the mean coefficient vector, β and covariance hyper-parameters, and the marginal likelihood of the data $p(y(\mathbf{x})|\psi, \mathbf{x})$ must be specified. A symmetric Normal proposal distribution for the log-hyper-parameters and the unconstrained coefficient matrix β is chosen. The hyper-parameters of the squared exponential covariance function are all strictly positive values. The marginal log-likelihood of the process can be written as

$$\log p(y(\mathbf{x})|X, \psi) = -\frac{1}{2}(\mathbf{y} - X\beta)^T K_y^{-1}(\mathbf{y} - X\beta) - \frac{1}{2}\log|K_y| - \frac{n}{2}\log(2\pi) \tag{8.3}$$

where $K_y = K(\mathbf{x},\mathbf{x}) + I_n\sigma_y^2$ is the covariance at the known points plus measurement noise. The un-normalised log-posterior is calculated by taking the log-pdf of the hyper-parameter prior distributions and adding the value of the marginal log-likelihood from equation 8.3.

The Metropolis algorithm used to determine the posterior distribution of the hyper-parameters is summarised in algorithm 24.

Algorithm 24: Metropolis algorithm for the Gaussian process model with a linear mean function.

```
Result: Return $\psi_i = \{\beta^{(i)}, \sigma_y^{(i)}, \sigma^{(i)}, h^{(i)}\}, i = 1, \dots, M$
1 Given pairs of locations and observations $(x_i, y_i), i = 1, \dots, N$;
2 Initialise the parameters by sampling from the prior $\psi_0 \sim p(\psi)$;
3 for $i$ in 1 to $M$ do
4     Propose a new parameter $\psi^\star \sim q(\psi^\star|\psi_{i-1})$;
5     Calculate $\log A = \log p(\psi^\star|Y, X) - \log p(\psi_{i-1}|Y, X)$;
6     Sample $u \sim U[0, 1]$;
7     if $\log u < \log A$ then
8         Set $\psi_i = \psi^\star$;
9     else
10        Set $\psi_i = \psi_{i-1}$;
11 end
```

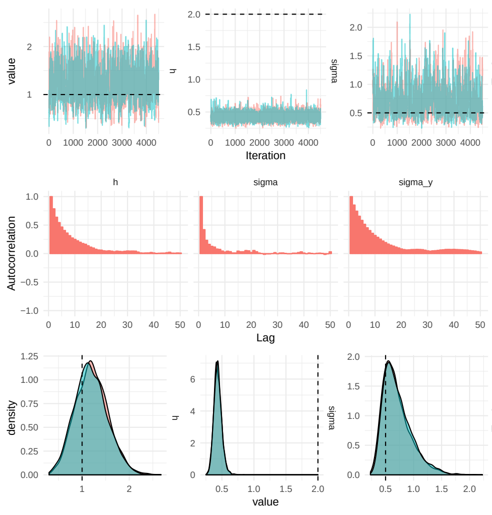
The data simulated to produce Figure 8.1 is used to demonstrate parameter inference using the Metropolis algorithm. The following weakly informative prior distributions were chosen

$$\begin{aligned}
p(h) &= \text{Inv-Gamma}(3, 6), \\
p(\sigma) &= \text{Inv-Gamma}(3, 6), \\
p(\sigma_y) &= \text{Inv-Gamma}(10, 6), \\
p(\beta_i) &= \mathcal{N}(0, 5).
\end{aligned}$$

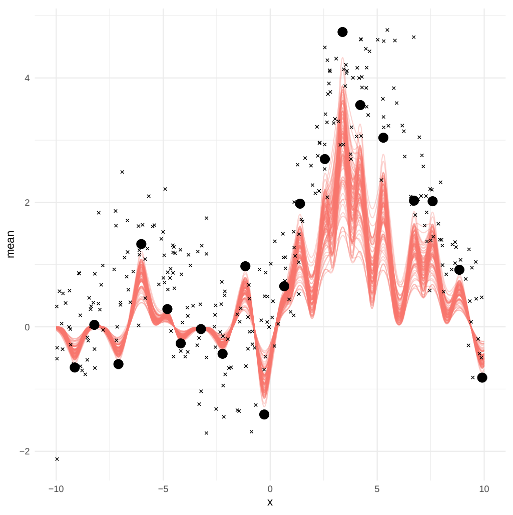
100,000 iterations of the MCMC scheme was performed, 10,000 iterations were discarded as burn-in and the chain was thinned by a factor of 20 to reduce autocorrelation. Figure 8.2 (a) shows the diagnostic plots for the hyper-parameter posterior distributions.

Since the data size is small, the space of functions which fit this data is quite large. Additionally the length scale, σ and marginal variance h interact and are challenging to identify. As such the posterior predictive distribution for previously unseen points can be informative for assessing model fit. With a conjugate Normal prior on the β parameters the posterior predictive distribution can be calculated in closed form [O'Hagan, 1978]. However, the approach taken here is to take draws from the hyper-parameter posterior then draw from the predictive distribution with mean and covariance as follows

$$\begin{aligned}
\mathbb{E}(y(x^\star)) &= x^\star\beta + K(x^\star, \mathbf{x})(K(\mathbf{x}, \mathbf{x}) + I_p\sigma_y^2)^{-1}(y - X\beta), \\
\text{Var}(y(x^\star)) &= K(x_\star, x_\star) - K(x_\star, \mathbf{x})(K(\mathbf{x}, \mathbf{x}) + I_p\sigma_y^2)^{-1}K(x_\star, \mathbf{x}) + I_p\sigma_y^2.
\end{aligned}$$



(a)



(b)

Figure 8.2: (a) Diagnostic plots of the MCMC draws from the posterior distribution of the parameters using the simulated Gaussian Process data, the actual values used to simulate the data are represented as dashed lines. Traceplots, (top). Autocorrelation, (middle). Marginal posterior densities, (bottom). (b) 100 draws from the posterior distribution of the latent function using 100 independent draws from the hyper-parameter posterior, the observed values are plotted as large circles

This accounts for the uncertainty in the mean parameter, β and the hyper-parameters ψ. Figure 8.2 (b) shows 100 draws from the posterior predictive distribution, created by sampling from the hyper-parameter posterior distribution 100 times and determining the conditional distribution of the unobserved data given the observed data. Since the length scale did not have an informative prior distribution and the mean of the posterior distribution of the length scale is approximately one-quarter of the size of the actual value used to simulate the data, the draws from the posterior predictive distribution change more between observations than the actual simulated function.

Hamiltonian Monte Carlo (HMC) (see section 1.3.4) can be used as a more efficient alternative to Metropolis-Hastings to determine the posterior distribution the kernel hyper-parameters. Alternatively, there are fast approximate methods for computing the posterior distribution of parameters in latent-Gaussian models using integrated nested Laplace approximations [Rue et al., 2009].

8.3 Example: Spatial Modelling of Environment Data

This example utilises the sensors under consideration in the multivariate DLM example in section 4.9. Figure 8.3 shows the daily average temperature from January to July

2018 at eight sensor locations. The temperature displays seasonality on a weekly level which is common in urban environments. A similar model is presented by [Lai et al., 2018] where several (more widely separated) temperature and humidity sensors are jointly modelled using a dynamic linear model with seasonal components. The spatial relationship is accounted for using a Gaussian process in the state equation rather than the observation equation as presented in this example.

Eight of the sensors are used as training data with one sensor held-out in order to examine the model fit, "new new emote 2604" is chosen. The locations of the sensors are plotted in figure 8.4 (Left). Figure 8.4 (Right) shows the monthly average temperature at each location, notice that May has the highest spatial variability of temperature with the sensors located in the North East (top right of the figure) reporting the highest temperature readings. The maximum number of readings for each sensor is $n = 244$, however some readings are missing if an entire day had no temperature readings.

Figure 8.5 (left) shows the within-sample residuals of the DLM from section 4.9 plotted against time. This shows some structure for sensors 1171 and 1172 which could be accounted for by spatial structure in the model. Figure 8.5 (right) shows the mean of the residuals and that the two sensors are co-located in the north east, further reinforcing the need for a spatial model.

The temperature time series is observed at times $t = 1, \dots, 244$ and $p = 8$ locations x_j, $j = 1, \dots, p$. It is believed that the time series of temperatures have local spatial dependence, and the dependence between measurements decreases as the distance between the sensors increases. A Dynamic linear model (DLM) is used to model the time dependence of the process. It is assumed the latent-state $\theta_{0:N}$ of the DLM evolves stochastically in discrete time and is Markovian, the measurements $Y_{1:N}$ are conditionally independent given the latent state. Since the DLM is linear and Gaussian, the Kalman filter can be used to perform inference for the latent state.

A Gaussian process is used to model the spatial dependence between locations. The Gaussian process is considered to be have a zero mean function and a covariance function, $k(x, x^\star)$ which depends on the Euclidean distance between the sensors and the kernel hyper-parameters ψ. The observations of the process are p-dimensional, however some observations are missing.

$$\begin{aligned}
Y_t &= \tilde{F}^T\boldsymbol{\theta}_t + \mathbf{f} + v_t, && \mathbf{f} \sim \text{MVN}_p(0, K(\mathbf{x}, \mathbf{x})), v_t \sim \text{MVN}_p(0, V), \\
\boldsymbol{\theta}_t &= G\boldsymbol{\theta}_{t-1} + w_t, && w_t \sim \text{MVN}_m(0, W), \\
\boldsymbol{\theta}_0 &\sim \text{MVN}_m(m_0, C_0).
\end{aligned} \tag{8.4}$$

The DLM is chosen to share the same latent state and parameters, except the value of

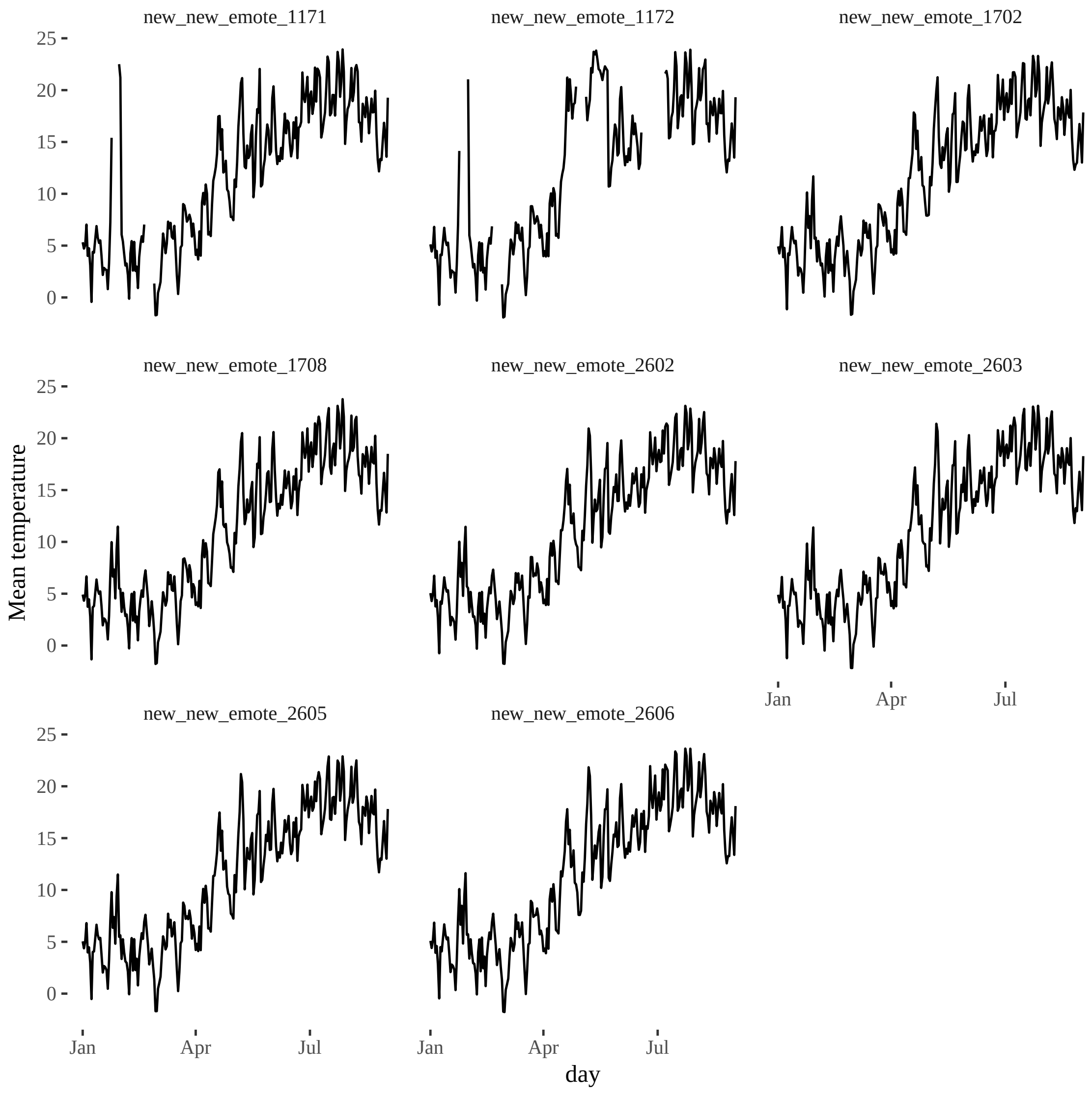


Figure 8.3: Daily average temperature at eight of the sensors used to determine the posterior distribution of the space-time model.

the observation variance is unique at each sensor location. The observation matrix is $p \times m$ where p is the dimension of the observations and m is the dimension of the common latent-state. $\tilde{F}$ is constructed by horizontally concatenating p model matrices:

$$\tilde{F} = \begin{pmatrix} F & F & \dots & F \end{pmatrix} = \mathbb{1} \otimes F$$

By sharing the same latent-state and noise covariance parameters, the DLM is assumed to account for the time variation of the measured temperature and the remaining differences are accounted for by the location of the sensor using a Gaussian process. Equivalently,

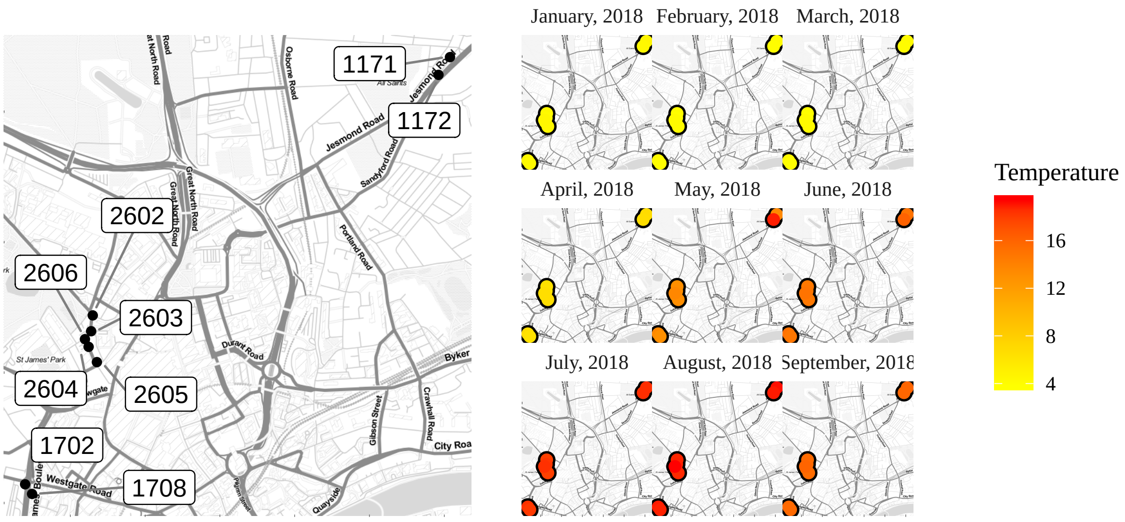
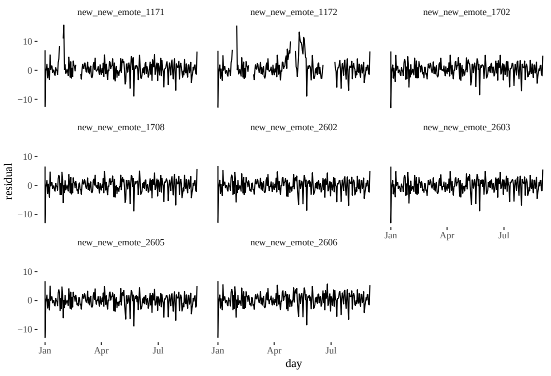


Figure 8.4: Locations of each temperature sensor, (left). Average temperature reading by month for each of the sensor locations, (right).

this model can be thought of as a Gaussian Process with a time dependent mean-function represented by a dynamic linear model:

$$
\begin{aligned}
Y_t &= \mathcal{GP}(\tilde{F}^T \boldsymbol{\theta}_t, K(\mathbf{x}, \mathbf{x}) + V), \\
\boldsymbol{\theta}_t &= G\boldsymbol{\theta}_{t-1} + w_t, \qquad w_t \sim \text{MVN}_m(0, W), \\
\boldsymbol{\theta}_0 &\sim \text{MVN}_m(m_0, C_0).
\end{aligned}
$$

The static parameters of the DLM GP model include the hyper-parameters of the chosen



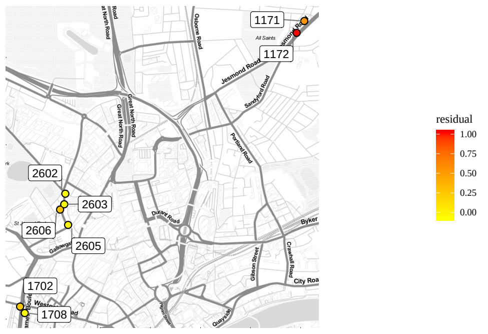


Figure 8.5: Residuals $y_t - F_t\theta_t$ for each sensor, showing that sensors 171 and 172 have an unusual pattern which could be explained by a spatial model, (left). Mean of the residuals at each sensor for all time, showing that sensors 171 and 172 are positioned away from the other sensors and near each other. They also have a higher than average residual for the period under consideration, indicating that it may be warmer in this region, (right).

covariance function, in this case $\xi = \{h, \sigma\}$ and the observation and system noise covariance matrices V and W, hence $\psi = \{\xi, V, W\}$. The joint distribution of all the random variables in the model can be written as

$$\begin{aligned}p(y_{1:N}, \boldsymbol{\theta}_{0:N}, \mathbf{f}, \sigma, h, V, W) &= p(\psi)p(\boldsymbol{\theta}_0)p(f(\mathbf{x})|\xi)\prod_{t=1}^{N} p(y_t(\mathbf{x})|\boldsymbol{\theta}_t, f(\mathbf{x}), V)p(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}, W)\\ &= p(\psi)\mathrm{MVN}_m(\boldsymbol{\theta}_0|m_0, C_0)\mathrm{MVN}_p(\mathbf{f}|0, K(\mathbf{x},\mathbf{x}))\\ &\times \prod_{t=1}^{N}\mathrm{MVN}_p(y_t(\mathbf{x})|\tilde{F}^T\boldsymbol{\theta}_t + \mathbf{f})V)\mathrm{MVN}_m(\boldsymbol{\theta}_t|G\boldsymbol{\theta}_{t-1}, W).\end{aligned}$$

In order to determine the joint posterior distribution a block Metropolis-within-Gibbs sampler is used whose stationary distribution is the posterior distribution

$$\begin{aligned}p(\boldsymbol{\theta}_{0:N}, \psi|y_{1:N}(\mathbf{x})) &= \int p(f|\xi)p(\boldsymbol{\theta}_{0:N}, f, \psi|y_{1:N}(\mathbf{x})))df,\\ &= \int p(\psi)\mathrm{MVN}_p(f|0, K(\mathbf{x},\mathbf{x}))p(\psi)\prod_{t=1}^{N}\mathrm{MVN}_p(y_t|\tilde{F}^T\boldsymbol{\theta}_t + f, V)df,\\ &= p(\psi)\prod_{t=1}^{N}\mathrm{MVN}_m(\boldsymbol{\theta}_t|G\boldsymbol{\theta}_{t-1}, W)\int \mathrm{MVN}_p(f|0, K(\mathbf{x},\mathbf{x}))\mathrm{MVN}_p(y_t|\tilde{F}^T\boldsymbol{\theta}_t + f, V)\\ &= p(\psi)\prod_{t=1}^{N}\mathrm{MVN}_m(\boldsymbol{\theta}_t|G\boldsymbol{\theta}_{t-1}, W)\mathrm{MVN}_p(y_t|\tilde{F}^T\boldsymbol{\theta}_t, K(\mathbf{x},\mathbf{x}) + V)\end{aligned}$$

First the latent-state $\boldsymbol{\theta}_{0:N}$ is sampled conditional on the other parameters using FFBS (see section 4.4), to do this write the observation equation as

$$y_t(\mathbf{x}) = \tilde{F}^T\boldsymbol{\theta}_t + \tilde{v}_t, \quad \tilde{v}_t \sim \text{MVN}_p(0, K(\mathbf{x}, \mathbf{x}) + V). \tag{8.5}$$

Then the model is in the form of a DLM and conditional on the other random variables in the model, FFBS can be used to jointly sample from the full conditional distribution of the latent-state.

The system and observation noise covariance matrices are considered to be diagonal ($V = \text{diag}(v_1, \dots v_p)$, $W = \text{diag}(w_1, \dots w_m)$). The system matrix is sampled conditional on the latent-state using a Gibbs step. The full conditional distribution of the j^{th} diagonal element of the system noise covariance matrix is written as

$$p(W_{jj}|\boldsymbol{\theta}_{0:N}, \psi^{(-W_{jj})}) = \text{Inv-Gamma}\left(\alpha + \frac{N}{2}, \beta + \frac{1}{2}\sum_{i=1}^{N}(\theta_t - G_t\theta_{t-1})^T(\theta_t - G_t\theta_{t-1})_{jj}\right). \tag{8.6}$$

Where the subscript jj on the sum in the scale term represents the j^{th} diagonal element of the matrix created from the product $(\theta_t - G_t\theta_{t-1})^T(\theta_t - G_t\theta_{t-1})$.

To sample the values of the hyper-parameters and the diagonal elements of the observation matrix, $V_{ii}, i = 1, \dots, p$ write the log of the marginal likelihood as

$$\begin{aligned}\log p(y_{1:N}|\boldsymbol{\theta}_{0:N}, \psi) &= \log\left(\prod_{t=1}^{N} \text{MVN}_p(y_t(x)|\tilde{F}^T\boldsymbol{\theta}_t, K(\mathbf{x}, \mathbf{x}) + V)\right) \\ &= -\frac{Np}{2}\log(2\pi) - \frac{N}{2}\log|\Sigma| - \frac{1}{2}\sum_{t=1}^{N}(y_t(\mathbf{x}) - \tilde{F}^T\boldsymbol{\theta}_t)^T\Sigma^{-1}(y_t(\mathbf{x}) - \tilde{F}^T\boldsymbol{\theta}_t)\end{aligned} \tag{8.7}$$

where $\Sigma = K(\mathbf{x}, \mathbf{x}) + V$. The diagonal elements of the observation variance and the hyper-parameters of the Gaussian process covariance function are determined using independent Metropolis-Hastings steps using the marginal log-likelihood in equation (8.7).

The prior distributions for the static parameters and hyper-parameters are

$$\begin{aligned} p(h) &= \text{Inv-Gamma}(3, 3), \\ p(\sigma) &= \text{Inv-Gamma}(3, 3), \\ p(V_{ii}) &= \text{Inv-Gamma}(3.0, 3.0), i = 1, \dots, p, \\ p(W_{jj}) &= \text{Inv-Gamma}(3.0, 3.0), i = 1, \dots, m. \end{aligned}$$

Algorithm **??** shows the steps required for the MCMC scheme used to determine the posterior distribution of the parameters in the temperature DLM GP.

Algorithm 25: Metropolis-within-Gibbs sampling algorithm for the Gaussian process model with a dynamic linear mean function.

```
   Result: Return $\psi_i = \{\sigma^{(i)}, h^{(i)}, V^{(i)}, W^{(i)}\}, i = 1, \dots, M$
1  Given pairs of locations and observations $(x_i, y_i), i = 1, \dots, N$;
2  Initialise the parameters by sampling from the prior $\psi_0 \sim p(\psi)$;
3  Define the covariance hyper-parameters to be the set $\xi$, hence $\psi = \{\xi, V, W\}$;
4  for $i$ in 1 to $M$ do
5      Sample from the full conditional for the latent state $p(\boldsymbol{\theta}_{0:N}|\psi, y_{1:N})$ using
         FFBS and equation (8.5);
6      Sample from the full conditional for the system noise covariance matrix from
         equation (8.6);
7      for $x \in \{\xi, V\}$ do
8          Propose a new log-parameter $x^\star \sim \mathcal{N}(x^\star|x, \delta)$;
9          Update $\psi^\star$ with the parameter $x^\star$;
10         Calculate $p(\psi^\star|Y, \theta)$ using equation (8.7);
11         Calculate $\log A = \log p(\psi^\star|Y, \theta) - \log p(\psi_{i-1}|Y, \theta)$;
12         Sample $u \sim U[0, 1]$;
13         if $\log u < \log A$ then
14             Set $\psi_i = \psi^\star$;
15         else
16             Set $\psi_i = \psi_{i-1}$;
17     end
18 end
```

Figure 8.6 shows the diagnostic plots for the diagonal elements of the observation covariance matrix for the training sensors. The diagonal elements corresponding to the sensors located in the north-east (V_{11} and V_{22}) are noticeably larger, indicating the measurements from these sensors are less reliable. Figure 8.7 shows the diagnostic plots for the diagonal elements of the common system covariance matrices.

To evaluate the fit of this spatial model the one-step posterior predictive distribution at the test location can be calculated. This model has unique observation variances for each sensor, hence a value of the observation variance for the new sensor must be known in order to learn the posterior predictive distribution.

The value of the observation variance at the test-location is unknown, the model assumptions could be updated to assume all sensors behave similarly and hence share the same observation variance, which could be plausible since the sensors are all from the same manufacturer. However, when looking at the posterior distribution of the observation noise covariance matrix it, two sensors have a higher posterior variance, possibly indicating that measurements from these sensors are less reliable The conjugate Kalman Filter

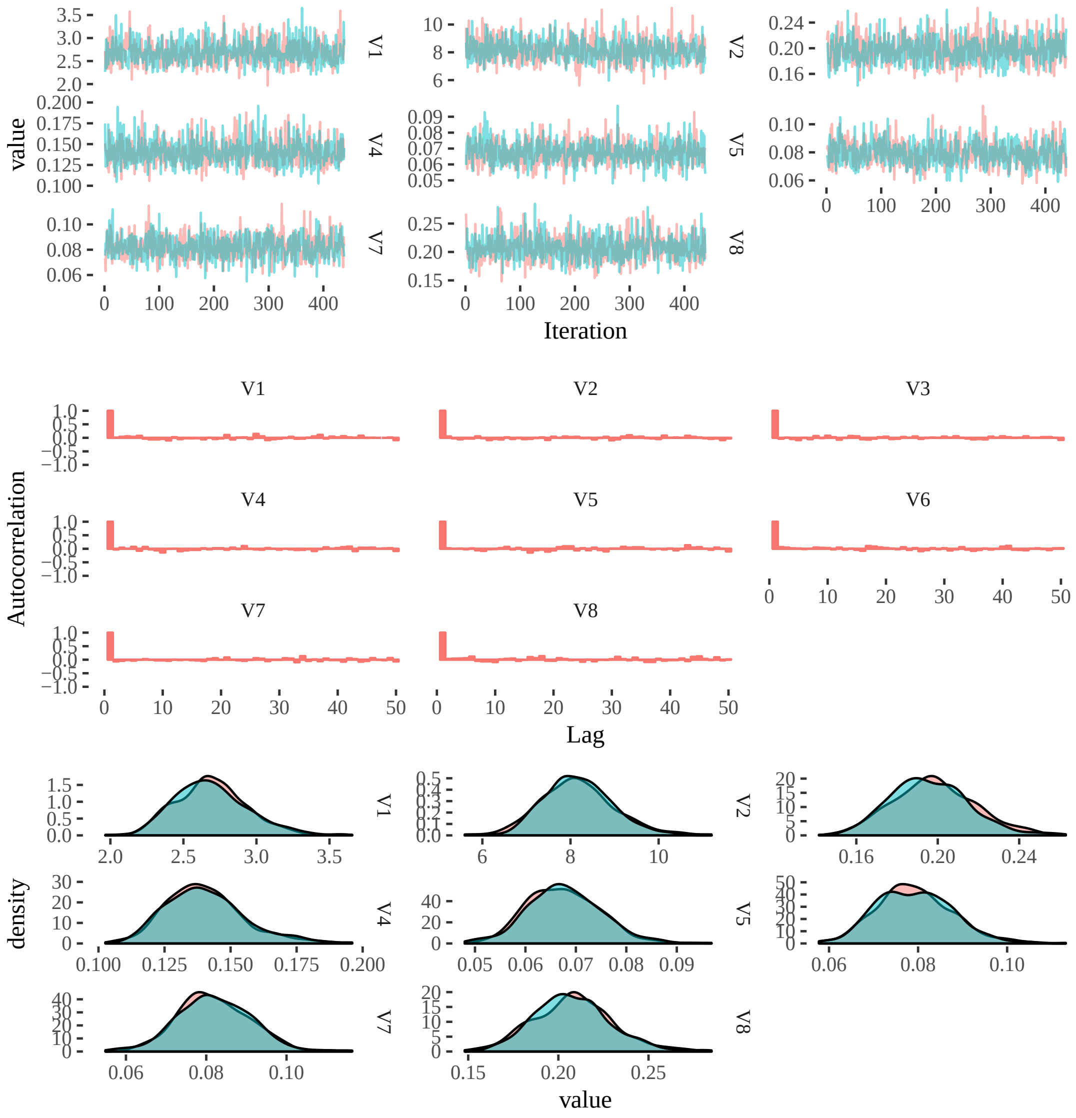


Figure 8.6: Diagnostic Plots for the diagonal components of the observation variance matrix V. Traceplots, (top). Autocorrelation of chain 1, (middle). Marginal density plots, (bottom).

from section 4.6.1 can be used to learn the joint posterior distribution of the latent-state and measurement noise covariance simultaneously. Figure 8.9 shows the posterior predictive distribution at the test-location, along with the one-step prediction residuals. The one-step forecast distribution for the conjugate filter is Student-t distribution [Murphy, 2007]. Alternatively, a random effects model could be considered here. In a random effects model the observation variances could be drawn from a common Inverse-Gamma distribution which is conjugate to the Gaussian observation distribution. The Inverse-Gamma distribution is in turn parameterised by hyper-parameters which allow the sensors to share

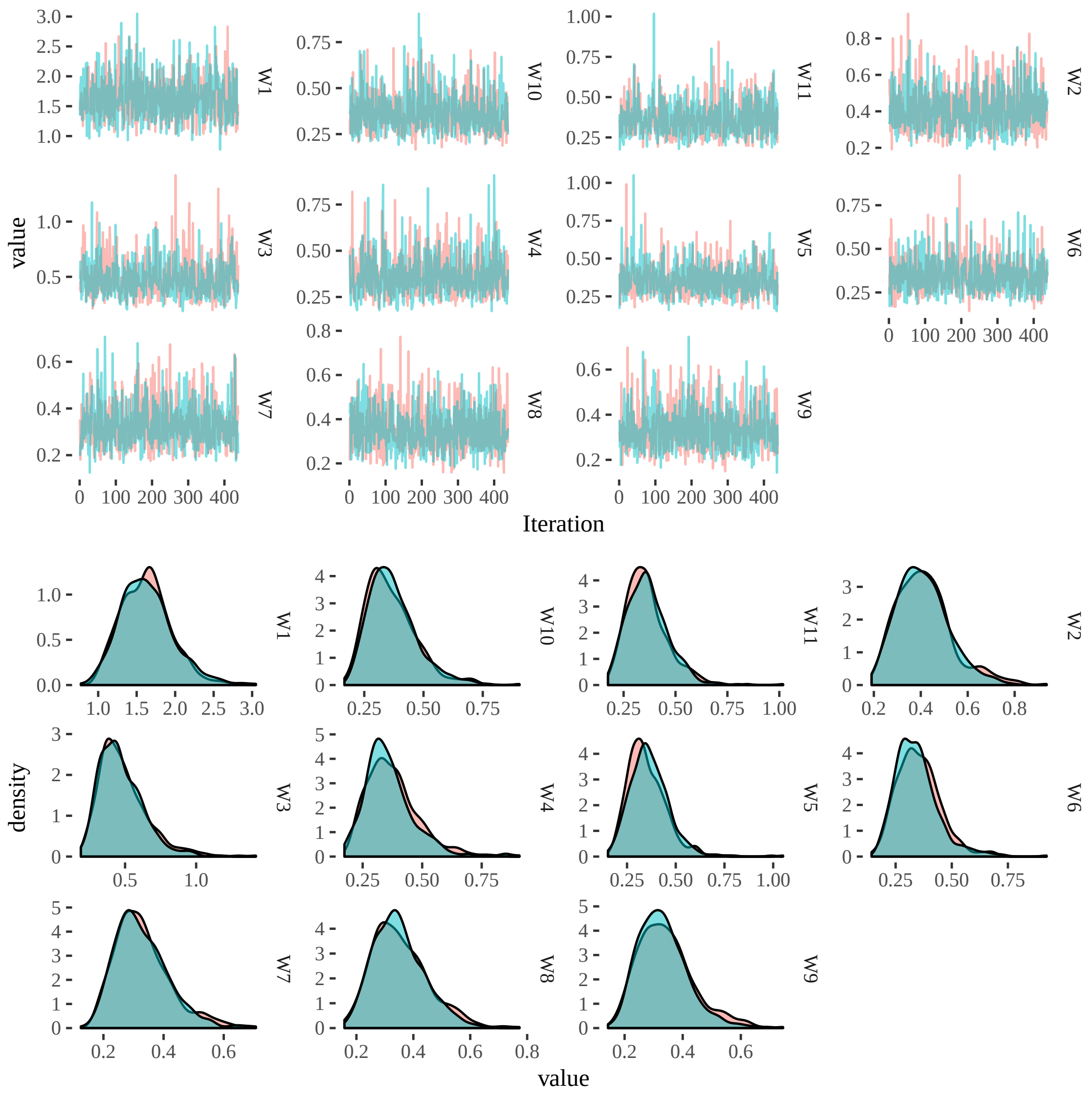


Figure 8.7: Diagnostic Plots for the diagonal components of the system variance matrix W. Traceplots, (top). Autocorrelation of chain 1, (middle). Marginal density plots, (bottom).

information for the variance between them. This hierarchical approach is appropriate since the temperature sensors are produced by the same manufacturer.

Sensors are typically placed in convenient locations, such as mounted to lampposts along roadways in areas which are accessible to those fitting the sensors. However, these are not the only areas of interest but merely convenient locations to place sensors. The dynamic Gaussian process developed in this section can be used to interpolate the temperature at areas which currently do not have any sensors. Since each sensor is assumed to have its own measurement variance, the sensors which are less accurate will tend to contribute less to the shared latent-state of the DLM and hence less to the prediction at

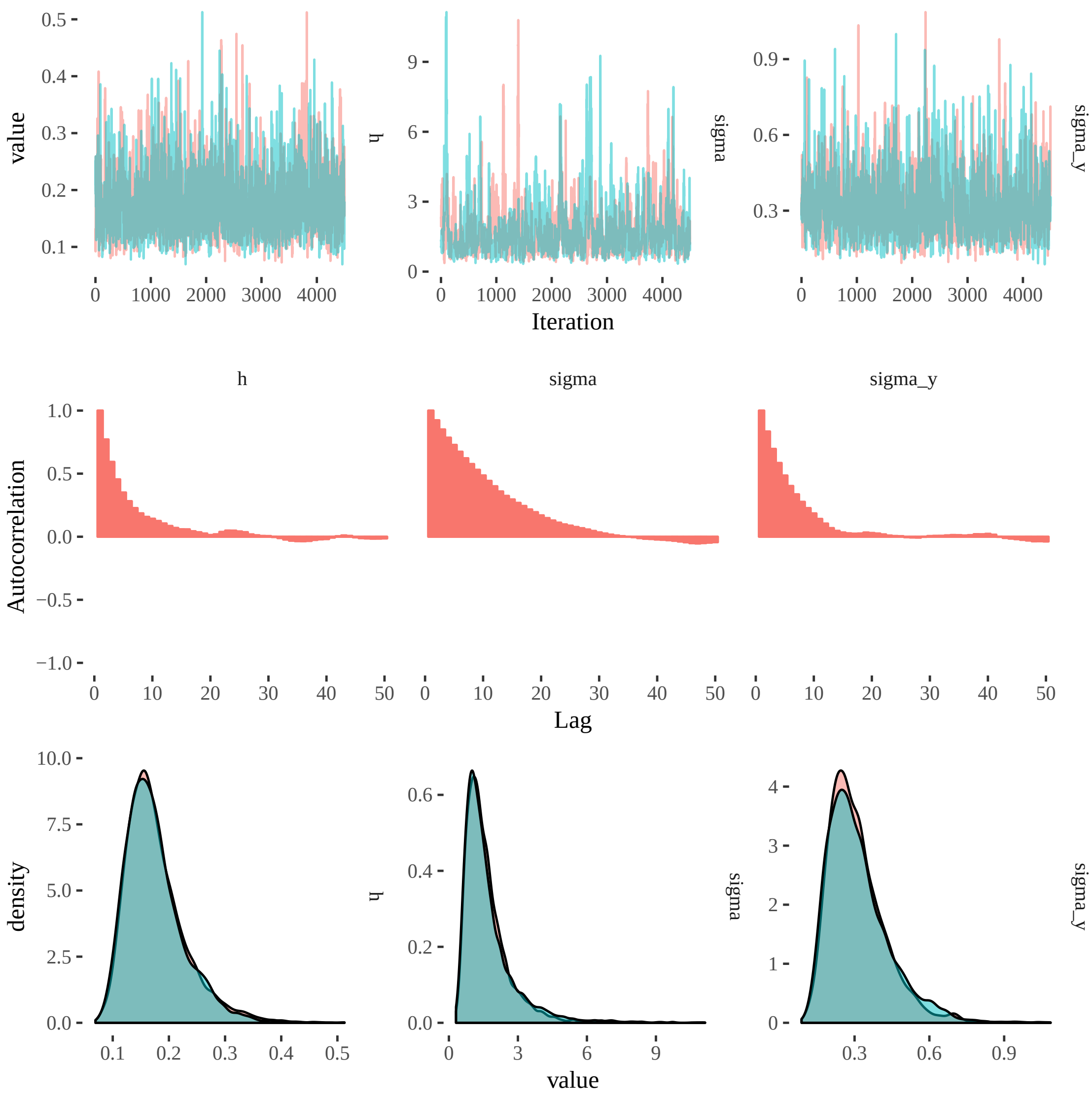


Figure 8.8: Diagnostic plots of the covariance function hyper-parameters σ_y, h, v. Trace-plots, (top). Autocorrelation of chain 1, (middle). Marginal density plots, (bottom).

the test site.

Figure 8.10 (a) shows the mean interpolated temperature for a single day in January 2018, 2018-01-31. On this day, some spurious measurements were made at the two sensors located in the north-east on Jesmond Road. Temperature values recorded at sensor 1171 and 1172 were around 21 degrees, whereas recordings at the other sensors ranged in value from 5.4 to 5.8 degrees. The smoothing provided by the model have provided realistic values for the actual temperature. The posterior variance of the Gaussian process on the right of the figure shows the Gaussian process is most uncertain around the two sensors on Jesmond Road where the spurious readings were recorded.

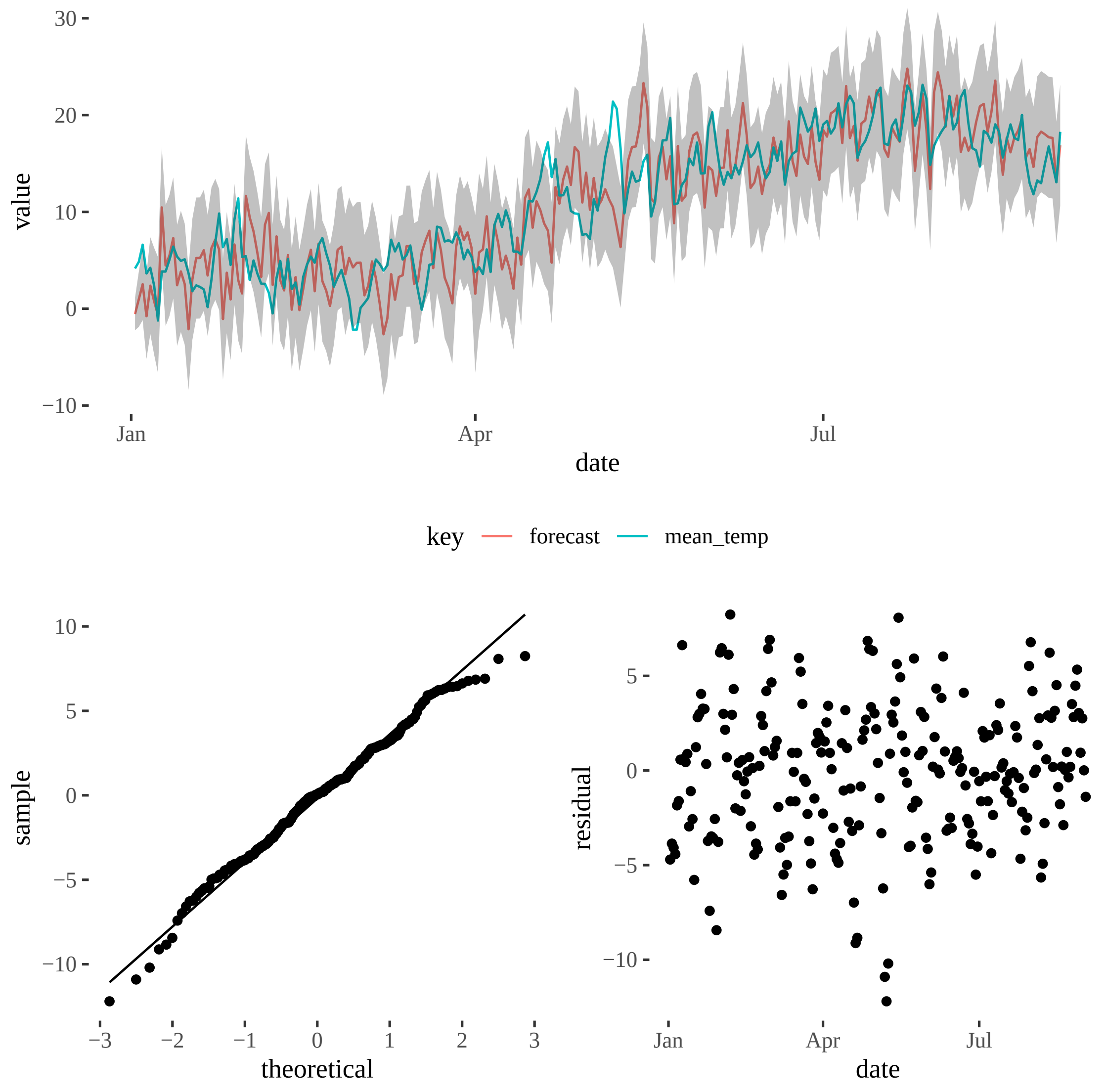
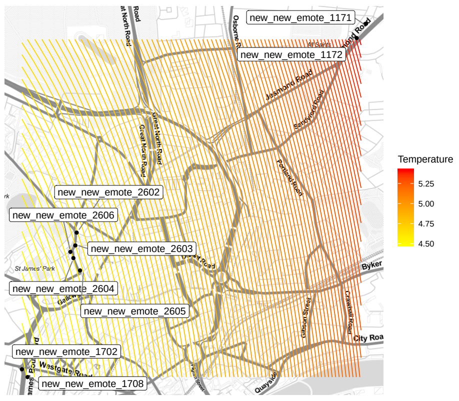


Figure 8.9: Mean prediction and 50% probability intervals at the test location, (top). QQ-Plot of residuals, (bottom left). Residuals against time, (bottom right).

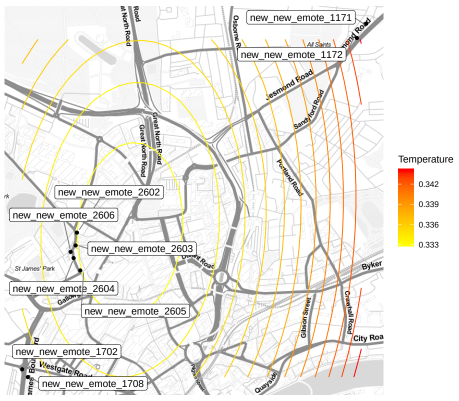
8.4 Summary

Urban sensor arrays such as those in the Urban Observatory often have sensors positioned in convenient locations, rather than locations of interest. Hence it is vital to perform accurate spatial modelling in order to understand the process of interest across the entire environment. In addition, joint spatial modelling allows sensors which are potentially reading incorrect measurements to be corrected by nearby sensors.

Modelling observed pollution data would require a more complex spatial process with an anisotropic covariance function depending on the wind direction and speed, since air-



(a)



(b)

Figure 8.10: (a) Posterior mean of temperature on January 31st 2018. (b) Posterior variance of the Gaussian process.

borne urban pollution is distributed according to the wind. The squared exponential covariance function considered in this chapter is isotropic and smooth which might not be the most appropriate choice of covariance function for sensors in an urban environment.

A novel spatial model for several nearby temperature sensors has been proposed with a dynamic mean specified using the building blocks of the DLM and a spatial component using a Gaussian process has been presented in this chapter. It has been shown that the Gaussian process is able to "correct" the values of sensors on Jesmond road which were reporting extremely high values of temperature which are assumed to be incorrect readings. In addition, the uncertainty in these estimates is accurately reflected indicating a higher posterior variance in the locations where the temperature readings were furthest from the global mean.

For future work it may be interesting to consider a similar model considered here for a larger array of sensors. The Urban Observatory has many more sensors available.

Chapter 9

Conclusion

This thesis has introduced a variety of dynamic models for the variety of environmental sensors found in the Urban Observatory. The inference algorithms and model building has been implemented in the functional programming language, Scala, which has features which allow for scalable implementations of inference algorithms. These features include straightforward parallelisation of statistical algorithms such as the particle filter in section 5.2.2 and MCMC algorithms such as sampling blocks in the stochastic volatility model in section 7.1.1. Chapter 2 introduces functional programming principles with an emphasis on statistical programming for time series data. Several Scala libraries have been developed implementing the models and inference schemes in this thesis, the first is a for dynamic linear models[1] the second[2] is for the composable models introduces in chapter 6. The third is a library for spatial modelling using Gaussian Processes[3] used in chapter 8.

The Akka library, introduced in section 2.3.1 allows data to be ingested from a variety of sources, such as web services, databases and files. Akka is an implementation of reactive streams, which are a natural way to model unbounded time series data. Once this data is ingested and parsed, it can be processed identically, independently of the source. The features implemented by Akka streams include sophisticated logging to identify problems with data or processing stages and monitor the progress of inference algorithms asynchronously. Akka has built-in flexible schemes for controlling the rate of data arriving in the stream - for instance a dynamically sampling sensor can drastically increase the rate of sampling when an important event occurs. An example of this is a river-level sensor used for flood prediction beginning to sample more frequently during heavy rain. The user can choose to buffer these measurements and wait for rate of incoming data reduce and hence the consumer (most likely an inference algorithm) can catch up. The time series algorithms developed in chapters 4 to 7 are implemented using Akka streams.

[1]https://git.io/dlm
[2]https://git.io/statespace
[3]https://git.io/gaussianprocesses

Chapter 3 presents the probability monad, and this allows all of the features of programming in a well developed functional programming language with higher kinded types to be used when specifying probabilistic models. This means it is straightforward to build complex probabilistic programs representing hierarchical models by combining other probabilistic programs using features implemented in the host language. This is illustrated in the example in section 3.6.3 where a hierarchical model is implemented by writing the linear model in section 3.6.1 as a Scala function and utilising it for each group in the hierarchical model. This implementation decouples the inference implementation from the model specification, which can allow quicker model prototyping.

Chapter 4 introduces dynamic linear models which can be used to model discrete time Gaussian time series (DLMs can be generalised to modelling continuous time using diffusion processes to represent the latent state as described in section 4.7). DLMs were used in an example to interpolate data from an environment sensor measuring four different processes, CO, NO, humidity and temperature. A block Gibbs sampler was used to determine the joint posterior distribution of the latent-state and covariance parameters. Various numerical problems with the Kalman filter (and hence FFBS) forced the research and implementation of numerically stable filters such as the Joseph form filter in section 4.5.1 and the SVD filter in section 4.5.2. These filters were implemented using the functional programming principles outlined in chapter 2 and described in the section on the functional particle filter 2.1.15 end of the section on the naive Kalman filter 4.2. Immutable data and higher order functions such as `foldLeft` allow us to determine which algorithms are online, requiring only information from the previous step. The pattern of defining a single binary function `(X, Y) => X` where `X` represents the latent-state at time $t-1$ and `Y` represents the observation at time t is a straightforward pattern, not prone to off-by-one errors of typical imperative, looping based approaches. These binary functions can be re-used for different collection types, including streams of live data.

Chapter 5 introduces non-Gaussian state space models by relaxing the form of the observation distribution of the models introduced in chapter 4. The Kalman filter can no longer be used for these non-Gaussian models. In this case the particle filter can be used to produce a weighted approximation of the filtering distribution and the marginal log-likelihood to perform pseudo marginal Metropolis-Hastings algorithm. A model for data with a large amount of outliers using the Student's-t distribution is developed which works well for the temperature data considered in section 5.8. A Gibbs sampler using a conditionally Gaussian Student's-t model by considering the Student's-t distribution as an Inverse-Gamma mixture of Normals, this MCMC scheme was found to be more efficient than a PMMH scheme.

Chapter 6 considers a class of composable models for irregularly observed time series data. This chapter emphasizes the streaming nature of modern time series data. Akka streams are used to implement model composition, particle filtering and the PMMH inference algorithm. The class of composable models forms a monoid with an associative composition operator, this allows the specification of complex state space models with built in seasonality and drift. The inference algorithm is chosen to be flexible and decoupled from the model specification, this allows users to explore a wide range of models without having to develop bespoke inference algorithms.

Chapter 7 considers adding an additional latent-state to model the evolution of the covariance of a Gaussian state space model using a factor stochastic volatility model. A novel dynamic model with a time dependent system matrix evolving according to a factor stochastic volatility model is proposed. This allows us to model non-stationary time series.

Chapter 8 considers modelling several nearby temperature sensors using a Gaussian process with a dynamic mean function represented by a DLM. The DLM is able to capture the temporal seasonality present in the temperature data, however when examining the difference between the posterior fitted values and the observed data, two sensors located in the north-east appeared to read higher than the other sensors. This discrepancy was considered to be a result of their location and hence modelling a spatial relationship between the sensors was natural. Some readings at these two sensors were four times as large as the other nearby sensors. These measurements were considered to be anomalous. This resulted in a high measurement variance at these temperature sensors. It was found that the model was able to smooth the forecasts at these locations providing a more accurate representation of the temperature.

9.1 Further Work

This section highlights some possible future directions for the field of time series modelling and probabilistic programming highlighted and developed upon in this thesis.

In a probabilistic program, although inference algorithms are decoupled from the statistical model, they still have to be chosen by the programmer. A program which chooses the most efficient inference algorithm using the graph representation of the statistical model could be an interesting research direction. This graph representation is usually already computed when calculating the gradient of the un-normalised log-posterior in HMC. In addition, this graph structure could assist with selecting the optimal inference algorithm by identifying sub-models which can be fit independently in parallel. Or, if a hierarchical model is specified using conditionally conjugate structure, then an efficient Gibbs sampler could be used.

Straightforward model implementations and fast inference schemes are required for exploration both in research and industrial applications. The flexibility and power which come with specifying models using embedded monadic domain specific languages are typically worth the time investment of learning functional programming.

This thesis has focused on a subset of the sensors deployed in the Urban Observatory. Although the inference methods are computationally intensive, the size of the datasets is not a challenge and they typically fit comfortably in memory of a single compute node. The Urban Observatory has many more deployed sensors and new sensors measuring the same things are being added regularly. An interesting challenge would be to fit a multivariate space-time model to all the sensors measuring specific variables in order to determine the smoothed value of CO or NO for example. This could result in the amount of data being processed being too large, hence new strategies could be developed such as developing faster inference algorithms. Another, more straightforward strategy, could be to simply drop all readings until the inference algorithm catches up and determine how this affects the accuracy of the inferences. Model coupling could be used in order to fit simple univariate models for each sensor which are re-coupled at a later date [Berry and West, 2018].

Two examples of Student's-t distributed state space models were considered in sections 5.7 and 5.8. The model residuals for the one-step forecasts indicated that the more robust distribution provided a better fit. A multivariate Student's-t state-space model could be used for the sensor measuring temperature, NO, CO and humidity. A multivariate DLM which uses a Student's-t distribution for selected time series can be implemented by only sampling the auxiliary variance parameters (described in section 5.4) for the required observations. An extension to choose which time series ought to be modelled using a Student's-t distribution could be considered by sampling the value of the degrees of freedom in a preliminary MCMC run, if a summary value (for instance the median) is greater than a threshold value then the observations can be considered to be Gaussian.

In this thesis, several competing models are considered for the same sensor. Model selection is performed by analysing the model diagnostics, such as the residuals of the one-step forecasts. Model selection could be automated by incorporating an indicator variable into the MCMC algorithms, or some other Bayesian model selection criteria. This was not performed as it was considered to be too computationally intensive.

The dynamic Gaussian process model for ten temperature sensors considered in chapter 8 could be improved upon by using a more suitable covariance function and not the infinitely smooth squared exponential function. In addition, expanding the model to include more sensors over a greater area could be used to inform future positioning of sensors.

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Christophe Andrieu and Johannes Thoms. A tutorial on adaptive MCMC. *Statistics and computing*, 18(4):343–373, 2008.

Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010.

Apache. Apache Hadoop, 2018a. URL `https://hadoop.apache.org/`.

Apache. Apache Spark - Unified Analytics Engine for Big Data, 2018b. URL `http://spark.apache.org`.

Yves F Atchadé, Jeffrey S Rosenthal, et al. On adaptive Markov chain Monte Carlo algorithms. *Bernoulli*, 11(5):815–828, 2005.

World Bank. United nations population division. world urbanization prospects, 2014. data retrieved from World Urbanization Prospects, `https://data.worldbank.org/indicator/SP.URB.TOTL.IN.ZS`.

Michael Barr and Charles Wells. *Category theory for computing science*, volume 49. Prentice Hall New York, 1990.

Brian Beckman. Kalman Folding, Part 1, Extracting Models from Data, One Observation at a Time. *preprint; vixra*, 2016. URL `http://vixra.org/abs/1606.0328`.

Nicola Bellotto and Huosheng Hu. People tracking with a mobile robot: A comparison of Kalman and particle filters. In *Proc. of the 13th IASTED Int. Conf. on Robotics and Applications*, pages 388–393, 2007.

Lindsay Berry and Mike West. Bayesian forecasting of many count-valued time series. *arXiv preprint arXiv:1805.05232*, 2018.

Michael Betancourt. A conceptual introduction to Hamiltonian Monte Carlo. *arXiv preprint arXiv:1701.02434*, 2017.

Gerald J Bierman. *Factorization Methods for Discrete Sequential Estimation.* Courier Corporation, 2006.

G. E. P. Box and Mervin E. Muller. A Note on the Generation of Random Normal Deviates. *Ann. Math. Statist.*, 29(2):610–611, 06 1958. doi: 10.1214/aoms/1177706645. URL `https://doi.org/10.1214/aoms/1177706645`.

Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. *Handbook of Markov chain Monte Carlo.* CRC press, 2011.

Richard S Bucy and Peter D Joseph. *Filtering for stochastic processes with applications to guidance*, volume 326. American Mathematical Soc., 2005.

RR Cacciola, M Sarva, and R Polosa. Adverse respiratory effects and allergic susceptibility in relation to particulate air pollution: flirting with disaster. *Allergy*, 57(4):281–286, 2002.

Lilian Calderon-Garciduenas, Antonieta Mora-Tiscareno, Lynn A Fordham, Gildardo Valencia-Salazar, Charles J Chung, Antonio Rodriguez-Alcaraz, Rogelio Paredes, Daina Variakojis, Anna Villarreal-Calderon, Lourdes Flores-Camacho, et al. Respiratory damage in children exposed to urban pollution. *Pediatric pulmonology*, 36(2):148–161, 2003.

Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A Brubaker, Jiqiang Guo, Peter Li, Allen Riddell, et al. Stan: A probabilistic programming language. *Journal of Statistical Software*, 20(2):1–37, 2016.

Chris K Carter and Robert Kohn. On Gibbs sampling for state space models. *Biometrika*, 81(3):541–553, 1994.

Carlos M. Carvalho, Michael S. Johannes, Hedibert F. Lopes, and Nicholas G. Polson. Particle Learning and Smoothing. *Statistical Science*, 25(1):88–106, 2010a.

Carlos M Carvalho, Michael S Johannes, Hedibert F Lopes, Nicholas G Polson, et al. Particle learning and smoothing. *Statistical Science*, 25(1):88–106, 2010b.

Noel Cressie. Statistics for spatial data. *Terra Nova*, 4(5):613–617, 1992.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

Pierre Del Moral. *Feynman-kac formulae.* Springer, 2004.

J. L. Doob. The Brownian Movement and Stochastic Equations. *Annals of Mathematics*, 43(2):351–369, 1942. ISSN 0003486X. URL `http://www.jstor.org/stable/1968873`.

A. Doucet, M. K. Pitt, G. Deligiannidis, and R. Kohn. Efficient implementation of Markov chain Monte Carlo when using an unbiased likelihood estimator. *Biometrika*, 102(2): 295, 2015.

Arnaud Doucet, Nando De Freitas, Kevin Murphy, and Stuart Russell. Rao-Blackwellised particle filtering for dynamic Bayesian networks. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pages 176–183. Morgan Kaufmann Publishers Inc., 2000.

Arnaud Doucet, Nando De Freitas, and Neil Gordon. *Sequential Monte Carlo methods in practice.* Springer, 2001. doi: 10.1007/978-1-4757-3437-9.

Simon Duane, Anthony D Kennedy, Brian J Pendleton, and Duncan Roweth. Hybrid Monte Carlo. *Physics letters B*, 195(2):216–222, 1987.

Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, September 1960. ISSN 0001-0782. doi: 10.1145/367390.367400. URL `http://doi.acm.org/10.1145/367390.367400`.

Daniel P Friedman and David Stephen Wise. *Unwinding structured recursions into iterations.* Indiana University, Computer Science Department, 1974.

Sylvia Frühwirth-Schnatter. Data augmentation and dynamic linear models. *Journal of time series analysis*, 15(2):183–202, 1994.

Dani Gamerman and Hedibert F Lopes. *Markov chain Monte Carlo: stochastic simulation for Bayesian inference.* Chapman and Hall/CRC, 2006.

Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. *Bayesian data analysis.* CRC press, 2013.

Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. In *Readings in Computer Vision*, pages 564–584. Elsevier, 1987.

Z Ghahramani and Mi Jordan. Factorial hidden Markov models. *Machine learning*, 273: 245–273, 1997. ISSN 0885-6125.

Michele Giry. A categorical approach to probability theory. In *Categorical aspects of topology and analysis*, pages 68–85. Springer, 1982.

Neil J Gordon, David J Salmond, and Adrian FM Smith. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. In *IEE Proceedings F-Radar and Signal Processing*, volume 140, pages 107–113. IET, 1993.

Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in haskell. In Donald Sannella, editor, *Programming Languages and Systems — ESOP '94*, pages 241–256, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. ISBN 978-3-540-48376-2. doi: 10.1007/3-540-57880-3_16.

W Keith Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A convenient category for higher-order probability theory. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12. IEEE, 2017.

Carl Hewitt, Peter Bishop, and Richard Steiger. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute, 1973.

Matthew D Hoffman and Andrew Gelman. The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15 (1):1593–1623, 2014.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, volume 1, page 3, 2017.

Stefano M Iacus. *Simulation and inference for stochastic differential equations: with R examples.* Springer Science & Business Media, 2009.

E L Ionides, C Bretó, and A A King. Inference for nonlinear dynamical systems. *Proceedings of the National Academy of Sciences of the United States of America*, 103(49): 18438–43, 2006.

PM James, RJ Dawson, N Harris, and J Joncyzk. Urban Observatory Environment. Newcastle University, 2014a. URL `http://dx.doi.org/10.17634/154300-19`.

PM James, RJ Dawson, N Harris, and J Joncyzk. Urban Observatory Transport. Newcastle University., 2014b.

Mark P Jones and Luc Duponcheel. Composing monads. Technical report, Technical Report YALEU/DCS/RR-1004, Department of Computer Science. Yale, 1993.

Simon J Julier and Jeffrey K Uhlmann. New extension of the Kalman filter to nonlinear systems. In *AeroSense'97*, pages 182–193. International Society for Optics and Photonics, 1997.

Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

Gordon M Kaufman and S James Press. Bayesian factor analysis. 1973. URL `http://hdl.handle.net/1721.1/1870`.

Bohdan B Khomtchouk, Edmund Weitz, Peter D Karp, and Claes Wahlestedt. How the strengths of Lisp-family languages facilitate building complex and flexible bioinformatics applications. *Briefings in bioinformatics*, 19(3):537–543, 2016.

Sangjoon Kim, Neil Shephard, and Siddhartha Chib. Stochastic volatility: likelihood inference and comparison with ARCH models. *The review of economic studies*, 65(3): 361–393, 1998.

Aaron A. King, Dao Nguyen, and Edward L. Ionides. Statistical Inference for Partially Observed Markov Processes via the R Package pomp. *Journal of Statistical Software*, 69(12):1–43, 2015.

Oleg Kiselyov and Chung-Chieh Shan. Embedded probabilistic programming. In *Domain-Specific Languages*, pages 360–384. Springer, 2009.

Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In *ACM SIGPLAN Notices*, volume 48, pages 59–70. ACM, 2013.

Peter E Kloeden and Eckhard Platen. Higher-order implicit strong numerical schemes for stochastic differential equations. *Journal of statistical physics*, 66(1-2):283–314, 1992.

Andreas Krause, Ajit Singh, and Carlos Guestrin. Near-optimal sensor placements in Gaussian processes: Theory, efficient algorithms and empirical studies. *Journal of Machine Learning Research*, 9(Feb):235–284, 2008.

Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M Blei. Automatic differentiation variational inference. *The Journal of Machine Learning Research*, 18(1):430–474, 2017.

Maria V Kulikova and Julia V Tsyganova. Improved discrete-time Kalman filtering within singular value decomposition. *IET Control Theory & Applications*, 11(15):2412–2418, 2017.

Yingying Lai, Andrew Golightly, and Richard Boys. Sequential bayesian inference for spatio-temporal models of temperature and humidity data. *arXiv preprint arXiv:1806.05424*, 2018.

Jonathan Law and Darren J Wilkinson. Composable models for online Bayesian analysis of streaming data. *Statistics and Computing*, pages 1–19, 2017.

F William Lawvere. The category of probabilistic mappings. *preprint*, 1962.

Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM, 1995.

Jane Liu and Mike West. Combined parameter and state estimation in simulation-based filtering. In *Sequential Monte Carlo methods in practice*, pages 197–223. Springer, 2001.

David J. Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. WinBUGS - A Bayesian modelling framework: Concepts, structure, and extensibility. *Statistics and Computing*, 10(4):325–337, 2000a. ISSN 1573-1375.

David J Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. WinBUGS-a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and computing*, 10(4):325–337, 2000b.

Oleksandr Manzyuk. A simply typed λ-calculus of forward automatic differentiation. *Electronic Notes in Theoretical Computer Science*, 286:257–272, 2012.

Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of functional programming*, 18(1):1–13, 2008.

AR Meenakshi and C Rajian. On a product of positive semidefinite matrices. *Linear algebra and its applications*, 295(1-3):3–6, 1999.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

Bartosz Milewski. Category theory for programmers, 2018. URL `https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/`. Accessed: 2018-10-25.

Gordon E Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

Kevin P Murphy. Conjugate Bayesian analysis of the Gaussian distribution. *Decisions in Economics and Finance*, 1:16, 2007. ISSN 1593-8883.

Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

Christopher JL Murray, Majid Ezzati, Abraham D Flaxman, Stephen Lim, Rafael Lozano, Catherine Michaud, Mohsen Naghavi, Joshua A Salomon, Kenji Shibuya, Theo Vos, et al. Gbd 2010: design, definitions, and metrics. *The Lancet*, 380(9859):2063–2066, 2012.

Lawrence M. Murray. Bayesian State-Space Modelling on High-Performance Hardware Using LibBi. *Journal of Statistical Software*, 67(10):1–36, 2015.

Lawrence M Murray, Anthony Lee, and Pierre E Jacob. Parallel resampling in the particle filter. *Journal of Computational and Graphical Statistics*, 25(3):789–805, 2016.

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. Probabilistic inference by program transformation in Hakaru (system description). In *International Symposium on Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, pages 62–79. Springer, 2016. doi: 10.1007/978-3-319-29604-3_5. URL `http://dx.doi.org/10.1007/978-3-319-29604-3_5`.

Radford M Neal et al. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2(11), 2011.

NECA. Open Data Service — North East Combined Authority, 2016. URL `https://www.netraveldata.co.uk/`.

J. A. Nelder and R. W. M. Wedderburn. Generalized Linear Models. *Journal of the Royal Statistical Society. Series A (General)*, 135(3):370–384, 1972. ISSN 00359238. URL `http://www.jstor.org/stable/2344614`.

Christopher Nemeth, Paul Fearnhead, and Lyudmila Mihaylova. Particle approximations of the score and observed information matrix for parameter estimation in state–space models with linear computational cost. *Journal of Computational and Graphical Statistics*, 25(4):1138–1157, 2016.

Yurii Nesterov. Primal-dual subgradient methods for convex problems. *Mathematical programming*, 120(1):221–259, 2009.

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

Anthony O'Hagan. Curve fitting and optimal design for prediction. *Journal of the Royal Statistical Society: Series B (Methodological)*, 40(1):1–24, 1978.

Bernt Oksendal. *Stochastic differential equations: an introduction with applications*. Springer Science & Business Media, 2013.

Michael Osborne and Stephen J Roberts. Gaussian processes for prediction. *Technical Report PARG-07-01*, 2007.

Sungwoo Park, Frank Pfenning, and Sebastian Thrun. *A probabilistic language based upon sampling functions*, volume 40. ACM, 2005.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS-W*, 2017.

Giovanni Petris, Sonia Petrone, and Patrizia Campagnoli. *Dynamic linear models with R*. Springer, 2009.

Mark Pitt and Neil Shephard. Time varying covariances: a factor stochastic volatility approach. *Bayesian statistics*, 6:547–570, 1999a.

Michael K Pitt and Neil Shephard. Filtering via simulation: Auxiliary particle filters. *Journal of the American statistical association*, 94(446):590–599, 1999b.

Martyn Plummer et al. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the 3rd international workshop on distributed statistical computing*, volume 124. Vienna, Austria, 2003.

Nicholas G Polson, James G Scott, et al. On the half-Cauchy prior for a global scale parameter. *Bayesian Analysis*, 7(4):887–902, 2012.

C. Arden Pope III and Douglas W. Dockery. Health effects of fine particulate air pollution: Lines that connect. *Journal of the Air & Waste Management Association*, 56(6):709–742, 2006. doi: 10.1080/10473289.2006.10464485. URL `https://doi.org/10.1080/10473289.2006.10464485`.

Raquel Prado and Mike West. *Time series: modeling, computation, and inference.* CRC Press, 2010.

R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2015. URL `https://www.R-project.org/`.

Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *ACM SIGPLAN Notices*, volume 37, pages 154–165. ACM, 2002.

Brian D Ripley. *Stochastic simulation.* John Wiley & Sons, Inc., 1987.

Gareth O Roberts and Jeffrey S Rosenthal. Optimal scaling of discrete approximations to Langevin diffusions. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 60(1):255–268, 1998.

Gareth O Roberts, Andrew Gelman, Walter R Gilks, et al. Weak convergence and optimal scaling of random walk Metropolis algorithms. *The annals of applied probability*, 7(1): 110–120, 1997.

Håvard Rue, Sara Martino, and Nicolas Chopin. Approximate Bayesian inference for latent Gaussian models by using integrated nested Laplace approximations. *Journal of the royal statistical society: Series b (statistical methodology)*, 71(2):319–392, 2009.

Ramiro Ruiz-Cárdenas, Elias T Krainski, and Håvard Rue. Direct fitting of dynamic models using integrated nested Laplace approximations—INLA. *Computational Statistics & Data Analysis*, 56(6):1808–1828, 2012.

John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science*, 2:e55, 2016.

Adam Ścibior, Zoubin Ghahramani, and Andrew D Gordon. Practical probabilistic programming with monads. In *ACM SIGPLAN Notices*, volume 50, pages 165–176. ACM, 2015.

Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. Functional programming for modular Bayesian inference. *Proceedings of the ACM on Programming Languages*, 2:83, 2018.

Neil Shephard and Michael K. Pitt. Likelihood Analysis of Non-Gaussian Measurement Time Series. *Biometrika*, 84(3):653–667, 1997. ISSN 00063444. URL `http://www.jstor.org/stable/2337586`.

Chris Sherlock, Gareth Roberts, et al. Optimal scaling of the random walk Metropolis on elliptically symmetric unimodal targets. *Bernoulli*, 15(3):774–798, 2009.

Dan Simon. *Optimal state estimation: Kalman, H infinity, and nonlinear approaches.* John Wiley & Sons, 2006.

Andrew Simpson. *A fully Bayesian approach to financial forecasting and portfolio selection.* PhD thesis, Newcastle University, 2002.

G. Storvik. Particle filters for state-space models with the presence of unknown static parameters. *IEEE Transactions on Signal Processing*, 50(2):281–289, Feb 2002. ISSN 1053-587X. doi: 10.1109/78.978383.

Stripe. Rainier: Bayesian inference in Scala, 2018. URL `https://github.com/stripe/rainier/`.

Wouter Swierstra. Data types à la carte. *Journal of functional programming*, 18(4): 423–436, 2008.

TensorFlow. Tensorflow probability, 2018. URL `https://www.tensorflow.org/probability/`.

Catherine L Thornton and Gerald J Bierman. UDUT Covariance Factorization for Kalman Filtering. In *Control and Dynamic Systems*, volume 16, pages 177–248. Elsevier, 1980.

Adrien Todeschini, François Caron, and Marc Fuentes. Biips · Bayesian inference with interacting particle systems , 2017. URL `http://biips.github.io/`.

K Triantafyllopoulos. Dynamic generalized linear models for non-Gaussian time series forecasting. *arXiv preprint arXiv:0802.0219*, 2008.

Typelevel. Cats: Lightweight, modular, and extensible library for functional programming., 2018. URL `https://github.com/typelevel/cats/`.

Uber. Pyro: Deep universal probabilistic programming with Python and PyTorch, 2018. URL `https://github.com/uber/pyro`.

Rui Vieira and Darren J. Wilkinson. Online state and parameter estimation in Dynamic Generalised Linear Models. *arXiv preprint arXiv:1608.08666*, 2016. URL `http://arxiv.org/abs/1608.08666`.

Philip Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.

Fei Wang, Xilun Wu, Gregory Essertel, James Decker, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *arXiv preprint arXiv:1803.10228*, 2018.

Liang Wang, Gaëtan Libert, and Pierre Manneback. Kalman filter algorithm based on singular value decomposition. In *[1992] Proceedings of the 31st IEEE Conference on Decision and Control*, pages 1224–1229. IEEE, 1992.

Larry Wasserman. Bayesian model selection and model averaging. *Journal of Mathematical Psychology*, 44(1):92 – 107, 2000. ISSN 0022-2496.

Noel Welsh. Differentiable Functional Programming, 2018. URL `https://slideslive.com/38908798/differentiable-functional-programming`. Accessed: 2018-10-25, Presented at Scala Days 2018.

Mike West and Jeff Harrison. *Bayesian forecasting and Dynamic Models.* Springer Science & Business Media New York, 1997. doi: 10.1007/b98971.

Mike West, P Jeff Harrison, and Helio S Migon. Dynamic generalized linear models and Bayesian forecasting. *Journal of the American Statistical Association*, 80(389):73–83, 1985.

Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis.* Springer-Verlag New York, 2016. ISBN 978-3-319-24277-4. URL `http://ggplot2.org`.

Darren J Wilkinson. Parallel Bayesian computation. In *Handbook of parallel computing and statistics (ed. Kontoghiorghes E. J., editor.)*, page 481–512. New York, NY: Marcel Dekker/CRC Press, 2005.

Christopher KI Williams and Carl Edward Rasmussen. Gaussian processes for machine learning. *the MIT Press*, 2(3):4, 2006.

Max A Woodbury. Inverting modified matrices. *Memorandum report*, 42(106):336, 1950.

Changye Wu, Julien Stoehr, and Christian P Robert. Faster Hamiltonian Monte Carlo by Learning Leapfrog Scale. *arXiv preprint arXiv:1810.04449*, 2018.

Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.